Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**

Empirical Evaluation and Novel Design of Object-Centric Debuggers to Improve the Debugging of Object-Oriented Programs

Évaluation Empirique et Conception Novatrice de Débogueurs Centrés sur les Objets afin d'Améliorer le Débogage des Programmes Orientés Objet

THÈSE

présentée et soutenue publiquement le 30 Octobre 2025

pour l'obtention du

Doctorat de l'Université de Lille

spécialité Informatique

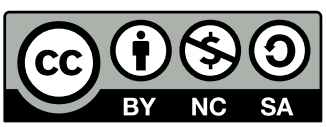
par

Valentin Bourcier

Composition du jury

Présidents :	Walter Rudametkin	Professeur des Universités Centre Inria de l'Université de Rennes
Rapporteurs :	Elisa Gonzalez Boix	Professeure Vrije Universiteit Brussel
	Jannik Laval	Maître de conférence (HDR) IUT Lumière de Lyon
Examinateurs :	Christophe Dony	Professeur des Universités Université de Montpellier
Directeur de thèse :	Steven Costiou	Chargé de recherche (HDR) Centre Inria de l'Université de Lille

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe



Acknowledgments

First, I would like to thank the thesis reviewers, Elisa Gonzalez Boix and Jannik Laval, for reading this dissertation and bringing their expertise to provide valuable feedback on my work, despite a tight schedule. I would also like to thank Christophe Dony for being part of my thesis committee and Walter Rudametkin for chairing it. Your comments and our scientific discussions during the defense were very helpful in formulating this final version of my thesis disseration.

I would like to express my gratitude to my supervisor, Steven Costiou, for giving me the opportunity to pursue my interest in scientific research and complete this thesis. I am especially grateful for his dedication to my work and his constant availability, which enabled us to have many conversations during which he offered advice, shared his expertise, and encouraged me. His consistent support over the past three years was essential to the completion of this thesis and my entry into the academic world. I extend my gratitude to Anne Etien for her collaboration on some of my work and for her valuable professional advice.

I would also like to thank my research team and its director, Stéphane Ducasse, for welcoming me into the team and for our enjoyable interactions. I am also grateful to all the doctoral students I have met, who have supported and advised me with their own experience, for all the moments of friendship and conversations they shared with me. I would like to warmly thank the people who accompanied me on a daily basis, especially Clotilde, Faouzi, and Soufyane for the positive conversations we had over puzzles during our lunch breaks. I would also like to thank Rémi, for his help and the many scientific and political discussions we had, and Alexandre for the shared memories of our first conference. Thanks also to Sebastian and Nahuel for facilitating those friendly after-work gatherings. Finally, I would like to give a special mention to Gabriel, with whom I began my thesis. I am grateful for the moments that built our friendship and made the past three years more enjoyable, and also for his support which greatly contributed to the completion of this thesis.

Last but not least, I owe my deepest gratitude to my family, my parents Jean-Marc and Sylvie, and my sister Tessa, for the unconditional support they have always given me, no matter what projects I undertake. I thank them and my close friends, Alexandre and Calista, for staying close despite the distance. Finally, I would like to give special thanks to my partner, Odeth, for her support and encouragement throughout this journey.

This thesis was funded by the ANR JCJC OCRE Project (https://anr.fr/Project-ANR-21-CE25-0004).

Abstract

Debugging software remains challenging and costly due to the difficulty of understanding program execution and tracing the origin of problems, which are often obscured by system complexity. To debug object-oriented program, developers are traditionnally provided with debuggers featuring breakpoints, steps, and a call-stack. They use these features to navigate the source code and methods involved in a program execution. However, research suggests that, when maintaining object-oriented programs, developers think not only in terms of source code and methods but also consider the objects' behavior, their state transitions, and interactions with other objects.

To address the discrepancy between the available debugging tools and the information developers need while debugging, researchers have proposed *object-centric debugging*. Object-centric debugging is an approach that proposes the use of debugging tools focused on specific objects rather than on classes and methods. These tools include object-centric breakpoints, which pause program execution for specific objects. They also include omniscient debuggers that allow developers to capture the behavior of objects during execution and then explore it. Although evaluations of this approach have been proposed, mostly through benchmarks and case studies, further research is needed to learn the impact of object-centric debugging.

The objective of this thesis is therefore to investigate the impact of object-centric debugging and propose solutions to improve the approach. To this end, we conducted an empirical evaluation of object-centric breakpoints. Our results suggest that these breakpoints could help developers save time while debugging. However, they also reveal limitations in scenarios involving bugs in object initialization. Therefore, we compared these results with those of object-centric debugging solutions based on omniscient debuggers. To do so, we conducted a case study analyzing debugging sessions with object-centric breakpoints and their omniscient equivalent, Time-Traveling Object-Centric Breakpoints (TTOCB). Our observations suggest that TTOCB would help developers debug scenarios in which object-centric breakpoints seem limited, and reduce the number of actions performed with the debugger. Finally, we compared the characteristics of existing object-centric debuggers with the debugging methodologies described in the literature. We discovered a lack of technological support from debuggers for the systematic debugging method. To address this issue, we propose Scopeo, an omniscient, queryable, and object-centric debugger featuring *exploration scopes*. Exploration scopes are subparts of program execution that developers can create and reuse with queries. To illustrate the feasibility of debugging with an exploration scope, we presented a case study of three debugging scenarios. We conclude this dissertation by identifying potential avenues for future research.

Keywords Object-Oriented Programming, Object-Centric Debugging, Object-Centric Breakpoints, Empirical Evaluation, Time-Traveling Debugger, Query-Based Debugging, Debugging Method, Debugging Workflow, Object Relationships, Debugging, Breakpoints.

Résumé

Le débogage des logiciels reste fastidieux en raison des difficultés liées à la compréhension de l'exécution des programmes et à la recherche de l'origine des problèmes qu'ils présentent. Pour déboguer un programme orienté objet, les développeurs disposent traditionnellement d'outils de débogage dotés de fonctionnalités qui leur permettent de naviguer dans le code source des méthodes impliquées dans l'exécution d'un programme. Cependant, des recherches suggèrent que, lors de la maintenance de programmes orientés objet, les développeurs ne pensent pas seulement en termes de code source, mais se questionnent aussi sur le comportement des objets, leurs interactions et changements d'états.

Pour remédier à l'écart entre les outils de débogage et les informations dont les développeurs ont besoin lors du débogage, les chercheurs ont proposé *l'object-centric debugging*, le débogage centré sur les objets. Le débogage centré sur les objets est une approche qui propose d'utiliser des outils de débogage axés sur des objets spécifiques plutôt que sur des classes et des méthodes. Ces outils comprennent les *object-centric breakpoints*, des points d'arrêt qui suspendent l'exécution de programme pour des objets spécifiques. Ils incluent aussi des débogueurs omniscients, qui permettent aux développeurs de capturer le comportement des objets pendant l'exécution, puis de l'explorer. Bien que des évaluations de cette approche aient été proposées, des recherches supplémentaires sont nécessaires pour connaître l'impact du débogage centré sur les objets.

L'objectif général de cette thèse est donc d'étudier l'impact de l'object-centric debugging et de proposer des solutions pour améliorer cette approche. Pour ce faire, nous avons mené une évaluation empirique des object-centric breakpoints. Nos résultats suggèrent que ces points d'arrêt pourraient aider les développeurs à gagner du temps lors du débogage, mais révèlent des limites pour certains scénarios. Nous avons comparé ces résultats avec l'équivalent omniscient des object-centric breakpoints, les Time-Traveling Object-Centric Breakpoints (TTOCB). Nos observations suggèrent que les TTOCB aideraient les développeurs à déboguer des scénarios pour lesquels les breakpoints object-centric semblent limités et à réduire le nombre d'actions nécessaires pour déboguer. Enfin, en étudiant la littérature, nous avons constaté que les debuggers object-centric n'apportent pas d'outils pour aider les développeurs à suivre la méthode de débogage systématique. Pour remédier à cela, nous proposons Scopeo, un debugger omniscient, et centré sur les objets, qui introduit les *exploration scopes*. Ces *scopes* sont des sous-parties de l'exécution du programme que les développeurs peuvent créer et réutiliser via des requêtes. Nous présentons ensuite une étude de cas illustrant la faisabilité du débogage à l'aide de Scopeo et terminons en identifiant des pistes de futures recherches.

Contents

Listings

List of Tables

List of Figures

Chapter 1

Introduction

Contents

Debugging is known to be a tedious and time-consuming part of the development and software maintenance process [Tassey 2002, Telles 2001, Vessey 1986, Zeller 2009, O'Dell 2017]. It consists in reproducing the behavior of a program, understanding how it came to be and implementing the best possible solution to rectify failures [Zeller 2009]. Among the challenges related to debugging, the great diversity of bugs makes choosing an appropriate debugging strategy difficult [Cotroneo 2025, Ahmed 2023, Gyimesi 2021]. Additionally, some bugs can be difficult to handle by nature [Perscheid 2017, Yin 2011, Costiou 2018]. The process of debugging can also be particularly costly due to the need for developers to maintain mental models of the program under investigation and the potential for false assumptions leading to incorrect debugging hypotheses [Détienne 2001, Yoon 1998, Vessey 1986, Gilmore 1991, Ko 2005, Spinellis 2018]. Additionally, it seems that developers today are mostly self-trained [Beller 2018, Perscheid 2017], even though teaching debugging seems to improve developers' skills in this process [Hassan 2024, Yang 2025]. At the same time, developers report that the tools are perceived as difficult to use, forcing developers to rely on lower-level instructions, such as *print statement* [Beller 2018].

1.1 Context

Debugging object-oriented programs presents an additional challenge. Among the debugging tools provided by Integrated Development Environments, breakpoints are the ones most commonly used by developers [Beller 2018], particularly to initiate an analysis of a program's behavior [Fontana 2021]. These debugging breakpoints are flags allowing developers to interrupt the execution of a program at any point in the source code. First, developers examine the source code to determine where to place a breakpoint. After placing a first breakpoint, they start

the execution of the program. The system stops the program execution when it reaches a breakpoint and displays the execution context in a call-stack debugger. Usually, to present the execution context, debuggers display the stack of executed operations (i.e., method calls) from the beginning of the program execution up to the breakpoint location. This enables developers to inspect and analyze the program's state and behavior, and possibly set additional breakpoints in the source code [Zeller 2009, Perscheid 2017, Petrillo 2019, Beller 2018]. However, research suggests that these breakpoints and especially call-stack debuggers are not suitable for debugging object-oriented programs [Lienhard 2006, Ressia 2012b]. These works highlight a conceptual gap between the perspective of standard debugging breakpoints (*i.e.,* the source code and the method call-stack) and the questions developers ask when debugging object-oriented programs. When debugging object-oriented programs, developers tend to ask questions about the behavior of the objects living in the execution context of the program [Sillito 2006, Kubelka 2014], rather than about source code. Therefore, when debugging a single suspect object, developers may experience unwanted interruptions because the execution is stopped every time an object executes the same code as the suspect object, on which the breakpoint was set. This does not help developers to reason about the specific objects they are investigating about.

Consequently, Ressia *et al.* proposed *object-centric breakpoints* [Ressia 2012b] that developers can use on objects rather than on the source code. With object-centric breakpoints, developers can select a single object and place a breakpoint on that object. The breakpoint is then active only for that particular object even if that object shares source code with other objects.

1.1.1 Object-Centric Debugging

The approach of focusing the debugging perspective on objects has been coined *object-centric debugging* [Lienhard 2006, Ressia 2012b]. Object-centric debuggers are expected to help analysis [Lienhard 2006, Lienhard 2007, Lienhard 2009, Thiede 2023a, Thiede 2023b], understanding [Willembrinck 2021, Willembrinck 2022b], and debugging [Hinkle 1993, Ressia 2012b, Corrodi 2016, Costiou 2018, Costiou 2020b] of object-oriented programs. In particular, these debuggers aim at addressing the discrepancy between the conventional representation of a software execution as a linear control flow (with the stack of method calls) and the abstraction of such a flow that is offered by object-oriented programs [Ressia 2012b]. These programs represent the execution as a directed graph, wherein the nodes are objects (representing a concept with a state and behavior) and the edges are messages and commands that are sent from one object to another. Therefore, we qualify as object-centric solutions those that aim to help developers follow the interactions and state transitions of objects, tasks that would otherwise be chal-

lenging using standard debugging tools, potentially demanding a greater number of actions [Ressia 2012b] or leading to program crashes [Hinkle 1993]. Some of these solutions enhance standard debuggers with object-centric operations such as breakpoints [Hinkle 1993, Ressia 2012b, Corrodi 2016, Costiou 2022], run-time behavioral adaptation [Ressia 2012a, Papoulias 2017, Costiou 2018], and support for object-centric operators [Costiou 2020b]. Others enhance back-in-time and time-travel debuggers with new object-centric views and operators [Lienhard 2006, Lienhard 2007, Lienhard 2009, Yang 2011, Willembrinck 2021, Willembrinck 2022b, Thiede 2023a, Thiede 2023b, Bourcier 2024b, Bourcier 2024c].

1.2 Research Questions

While object-centric debugging has been shown to be effective in certain scenarios [Ressia 2012b], there is still a lack of empirical evaluation of its impact on the activity of debugging. This lack of evaluation makes it difficult for researchers and developers to understand the true benefits and limitations of object-centric debugging. Therefore, we cannot determine whether it is worthwhile to continue making engineering efforts and conducting research to develop practical object-centric debugging approaches.

Most studies have focused on illustrating the approach through case studies [Lienhard 2009, Gestwicki 2005, Hofer 2006, Ressia 2012b, Phang 2013], or have used benchmarks to measure efficiency in terms of calculation and memory space [Hinkle 1993, Ressia 2012b]. However, few studies have involved developers directly in the evaluation process. Whyline, has been empirically evaluated through several user studies [Ko 2004, Ko 2008, Ko 2009]. These evaluations strongly suggest the superiority of Whyline over traditional tools when it comes to facilitating debugging. However, the evaluation designs have limitations that make confirming the results important. Willembrinck *et al.* also provides an empirical evaluation related to object-centric debugging [Willembrinck 2021]. They investigated a time-travel debugger using 14 program comprehension tasks. Two of these tasks required the use of the object-centric options of the debugger. Even though their evaluation shows significant results when considering all tasks, the authors could not conclude about the object-centric tasks of their experimental design. Without an empirical evaluation, we cannot assess the impact of object-centric debugging on the debugging process. Therefore, we cannot determine whether it is worthwhile to continue researching this topic or in which direction the research should proceed to best help developers.

Hence our main research question:

> **RQ$_1$**. How does object-centric debugging impact the activity of debugging?

Later in our thesis we focus on the evaluation of object-centric breakpoints as they represent the most researched object-centric debugging solution with a stable implementation. However, many contributions to the field of object-centric debugging propose using omniscient debuggers to explore program executions in an object-centric manner. Therefore, in this thesis, we are also interested in understanding the practical differences between object-centric approaches based on breakpoints and those based on omniscient debuggers.

> **RQ$_2$**. What are the practical differences between object-centric approaches based on breakpoints and those based on omniscient debuggers?

Object-oriented programs can manipulate a large number of different objects. While debugging these programs, developers often wonder about the behavior of the objects being manipulated [Sillito 2008, Kubelka 2019]. Developers follow an iterative debugging methodology that includes hypothesis formulation, verification, and refinement to accomplish this task [Gilmore 1991, Zeller 2009, Spinellis 2018]. This methodology allows them to reduce the search space for objects and potential flaws in their behavior. Existing object-centric debugging approaches, particularly omniscient ones, partially support this cycle. In our thesis, we wonder if we can provide technological support to debugging cycles, and how such support would impact the debugging of object-oriented program.

> **RQ$_3$**. Can we provide technological support for debugging cycles, and how does such support impact the debugging of object-oriented programs?

1.3 Thesis Outline

This thesis investigates the impact of object-centric debugging on the debugging of object-oriented programs and how to improve existing object-centric debugging approaches. Below, we outline the structure of our dissertation.

Chapter 1: State of the Art

We begin our dissertation by identifying general debugging challenges from the literature before presenting the main challenge to debugging object-oriented

programs. We then list the existing object-centric approaches, their limits, and conclude this chapter by highlighting the lack of empirical evidence supporting this approach.

Chapter 2: Evaluating the Impact of Object-Centric Breakpoints on Debugging

In this chapter, we present the first user study on object-centric breakpoints. We designed and conducted a controlled experiment involving 81 developers to investigate the impact of object-centric breakpoints on the debugging of object-oriented programs. The results show that object-centric breakpoints can facilitate debugging, but might also hinder it when bugs are located at program initialization.

Chapter 3: A Case Study of Object-Centric Breakpoints and Time-Traveling Queries

In this chapter, we introduce omniscient debuggers and time-traveling queries as means to address the limitations of object-centric breakpoints highlighted by the results of the empirical experiment. We present a case study that compares the debugging sessions of one developer across four tasks with object-centric breakpoints and with Time-Traveling Object-Centric Breakpoints, an equivalent of object-centric breakpoints leveraging the potential of omniscient debuggers.

The results show that omniscient debugging approaches have the potential to facilitate the debugging of programs in which defects are executed during the initialization process. We also highlight how time-traveling queries facilitate the implementation of object-centric debugging tools.

Chapter 4: Scopeo, Towards Object-Centric Debuggers Supporting Debugging Cycles

In this chapter, we introduce Scopeo, our proposal for supporting debugging cycles. Scopeo is an omniscient debugger that automates the exploration of program execution history by executing queries. We introduce the concept of an *exploration scope* with Scopeo. An exploration scope is a representation of subparts of the execution history obtained as a result of queries. It supports debugging cycles by being also reusable as input for new exploration phases.

We evaluated our approach using a case study, reusing scenarios from empirical experiment presented in chapter 3. The results provide insights to how the exploration scope can be used to support debugging cycles, and draw the limitations of the current prototype of Scopeo.

1.4 Contributions

1.4.1 Publications

International conferences - peer-reviewed

Valentin Bourcier, Pooja Rani, Maximilian Ignacio Willembrinck Santander, Alberto Bacchelli, Steven Costiou, "Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs". In: *ACM International Conference on the Foundations of Software Engineering FSE*. 2025. URL: https://hal.science/hal-04948470v1

Valentin Bourcier, Alexandre Bergel, Anne Etien, Steven Costiou, "Debugging Activity Blueprint". In: *2024 IEEE Working Conference on Software Visualization (VISSOFT)*. 2024, pp. 48–58. DOI: 10.1109/VISSOFT64034.2024.00015. URL: https://ieeexplore.ieee.org/abstract/document/10794853

Valentin Bourcier, Steven Costiou, "Scopeo: an Object-Centric Debugging Approach for Exploring Object-Oriented Programs". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Rovaniemi, Finland, 2024. DOI: 10.1109/SANER60148.2024.00040. URL: https://inria.hal.science/hal-04627606

International journals - peer-reviewed

Valentin Bourcier, Maximilian Willembrinck, Adrien Vanègue, Stéphane Ducasse, Anne Etien, Steven Costiou, "Time-Traveling Queries: Extensible Tools for Faster Program Comprehension." In: *The Journal of Object Technology* 23.1 (2024), p. 1. URL: https://www.jot.fm/issues/issue_2024_01/article7.pdf

Valentin Bourcier, Steven Costiou, Maximilian Ignacio Willembrinck Santander, Adrien Vanègue, Anne Etien, "Time-traveling object-centric breakpoints". In: *Journal of Computer Languages* (2024). DOI: 10.1016/j.cola.2024.101285. URL: https://inria.hal.science/hal-04629161

Workshop - peer-reviewed

Adrien Vanègue, Valentin Bourcier, Fabio Petrillo, Steven Costiou, "Debugging Video Games: A Systematic Mapping". In: *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques*. 2023, pp. 23–30. DOI: 10.1145/3605155.3605865. URL: https://inria.hal.science/hal-04139070v1

1.4.2 Awards

Technology Award for The Pharo Debugger - ESUG 2023

At the 2023 European Smalltalk User Group (ESUG) International Conference, we received an award for presenting advanced debugging tools in Pharo. These tools served as the foundation for the development of the debugging tools and the research presented in this thesis.

1.4.3 Software

Scopeo [Bourcier 2025a]

Scopeo is an omniscient debugger that allows developers to ask questions in the form of queries that collect objects and events related to those objects. Scopeo allows developers to save a query's results in an exploration scope that can be reused as subject for new, more refined queries. This supports the refinement of hypotheses that developers make during debugging.

Chapter 2

Background and State of the Art

Contents

In this chapter, we present the challenges associated with debugging. We begin with a general overview of these challenges before diving into the details of the challenges specific to the debugging of object-oriented programs. We then present existing object-centric debugging approaches. First, we present approaches based on breakpoints and outline their limitations. Then, we present omniscient approaches, which are an alternative to breakpoint-based approaches. We conclude this chapter by listing the evaluations of object-centric approaches before discussing their limitations, explaining why these limitations constitute one of the motivations for this thesis.

2.1 Debugging, a Challenging Task

A bug is characterized by unexpected behavior, also known as a *failure* of a software program at runtime. A bug originates from a *cause*, or *defect*, in the code [Zeller 2009]. When a defect is executed, it can corrupt the state of the program. This corrupted state does not necessarily lead immediately to the program's

failure. In this case, the execution continues, and the corrupted state might propagate to future program states until the program ends or fails [Zeller 2009]. Debugging involves identifying and understanding the sequence of *infections* from the cause to the failure, also known as the *cause-effect* chain, and correcting the defect [Zeller 2009, Beller 2018]. Of these two aspects of debugging, identifying and understanding the cause-effect chain is seemingly the most complex [Azadmanesh 2017, Zeller 2009, Yoon 1998, Gilmore 1991]. Although one might expect *Automatic Program Repair* approaches to help developers avoid this complexity, researchers report otherwise. Winter et al. [Winter 2023] surveyed 386 software developers to investigate their attitude toward Automatic Program Repair. They found that developers tend not to trust automated debugging approaches due to challenges in code comprehension [Böhme 2017] and difficulties in understanding automatically generated fixes [Parnin 2011]. Therefore, an interesting question to consider is what common obstacles or challenges make it difficult to understand code and programs? Fortunately, the research community has already reported on the challenges that hinder code and program comprehension.

Bug diversity: Some bugs such as concurrent bugs or design related bugs are inherently complex [Perscheid 2017, Yin 2011]. Researchers described several difficult-to-correct bugs from the literature [Costiou 2018]. These are the Heisenbug [Gray 1986, Eisenstadt 1997, Zeller 2009], the Stealth-bug [Eisenstadt 1997], the Mandelbug [Grottke 2007] and the Schröedinbug [Raymond 1996]. These bugs have one thing in common: their defects do not leave clues of their existence after corrupting the program. They may even remain undetected during the testing phase and only appear in specific scenarios within the program [Costiou 2018]. As of today, the literature contains taxonomies of bugs that are relevant to the majority of software systems [Catolino 2019, Tan 2014]. However, the research community continues to study the types of bugs associated with specific programming paradigms, such as non-deterministic and concurrent programming [Leesatapornwongsa 2016], event-driven programming [Tchamgoue 2011], and domains like the gaming industry [Coppola 2024, Butt 2023, Lewis 2010]. Studies have also been conducted on bugs encountered when using specific programming languages [Cotroneo 2025, Ahmed 2023, Gyimesi 2021]. This research help to develop bespoke debugging strategies and design and assess specific debuggers or automatic fault detection solutions. However, the variety of bug categories, languages, and tools can hinder the selection of an effective debugging strategy for any given situation.

Cognitive load: When debugging, developers maintain mental models of the program under investigation [Gilmore 1991, Yoon 1998, Azadmanesh 2017,

O'Dell 2017]. They have a mental model of their understanding of the program specifications and its expected behavior, as well as a mental representation of the problem [Gilmore 1991, Yoon 1998, Détienne 2001]. While debugging, developers formulate implicit hypotheses about the differences between a program's actual state, behavior, and their mental model of it [Vessey 1986, Gilmore 1991, Yoon 1998, Spinellis 2018]. To verify these hypotheses, they ask "why?", "what?", or "how?" questions [Sillito 2006, Sillito 2008, Kubelka 2019], to which they respond by modifying the code or using debugging tools [Layman 2013, Perscheid 2017]. Once they have an answer, they update their mental model with the newly acquired knowledge and start again the process until they find the cause of the bug [Vessey 1986, Gilmore 1991, Yoon 1998, Spinellis 2018]. Since mental models are approximations of reality used for reasoning about programs [O'Dell 2017], and because individuals can only handle so much cognitive load, developers' mental models of a program may contain false assumptions [Gilmore 1991, Ko 2005]. False assumptions lead to incorrect hypotheses, forcing developers to repeat the debugging process. This cycle makes debugging a particularly costly cognitive activity since it continues until the developers have a sufficient understanding of the program to identify the source of the bug or experience a cognitive breakdown (which could lead to further bugs [Ko 2005]).

Debugging education: In a user study, Yin et al. reported that about 27% of erroneous bug corrections are made by developers who never contributed to the code containing the defect [Yin 2011]. When developers have a strong understanding of the code involved in a bug, they can create a more accurate mental model of the issue, increasing the chances of proposing the right fix. Gugerty et al. found that more experienced developers demonstrate a stronger ability to understand code and programs behavior [Gugerty 1986]. If debugging skills play a role in understanding the program [Jeffries 1982, Vessey 1985, Gugerty 1986, Chmiel 2004, Albluwi 2021], then an interesting question arises: How are these skills acquired? Recent studies indicate that most developers learn debugging skills independently rather than during their computer science education [Perscheid 2017, Beller 2018]. Perscheid et al. highlight that among the participants to their survey, those who did not receive an education to debugging were performing similarly to those who received debugging education [Perscheid 2017]. This suggests that as of today, developers mostly learn to debug through practical experience. Yang et al. highlight that although teaching debugging can be challenging because *"students may lack confidence in executing debugging strategies* [...] *may perceive that the strategies are slowing them down ..."* [Yang 2025], there is evidence that, through teaching, students can learn how to debug more accurately [Yang 2025] and understand unfamiliar code [Hassan 2024]. Therefore, it seems that during their studies many developers miss out on exercises and knowledge that could help them to debug.

Tools complexity: Many developers are unaware of advanced debugging tools [Perscheid 2017, Beller 2018]. They continue to use *print statements* (code operations that log chains of characters) for debugging when needed, even though they believe debuggers are more efficient [Beller 2018]. It's only natural to wonder what hinders the adoption of advanced debugging tools and why developers resort to basic, low-level debugging approaches. It seems that, although some advanced tools effectively improve users'comprehension strategies, they can also hinder users' ability to understand programs when they present insufficient information or are not well-suited to the task at hand [Storey 2000]. This forces users to change their debugging strategies, which adds cognitive load. In their user study involving 458 participants, Beller et al. reported that developers *"admitted that debuggers are not easy to use"* and that they *"did not want more debugging features, but for existing ones to be made easier to use"* [Beller 2018]. This contradicts the idea that debugging is a situational, fast-paced, and complex activity [Beller 2018]. In order to be adopted, debugging tools should help resolve hard problems in the easiest way possible [Zayour 2001, Beller 2018]. Without that, developers might continue using low-level approaches even when it is considered less efficient.

2.2 Debugging Object-Oriented Programs, Specific Challenges

In section 2.1, we have presented the challenges to debugging in general, without going into detail about languages or programming paradigms. Each programming paradigm, whether it is procedural, object-oriented, functional or logic programming, uses a different approach to abstract the concepts developers want to implement. A change in programming paradigms alters how developers think about programs, organize and navigate source code, and seek information during development or debugging [Wiedenbeck 1999]. The focus of this thesis is to make debugging programs based on the object-oriented paradigm easier. Hence, in this section, we detail the main challenges that developers face when debugging object-oriented programs.

2.2.1 Observing Particular Objects

Most dynamic debugging tools shipped by default with Object-Oriented programming languages face an important limitation. They provide developers with a main perspective, the methods *call-stack* which is the sequence of methods executed before reaching a certain point in the debugged execution. However, when developing Object-Oriented programs, developers do not only think in terms of code or methods, but also in terms of objects, their state transitions and interactions with other objects [Sillito 2006, Sillito 2008, Kubelka 2019]. Indeed, the

model representing the life-cycle of objects during program run time is closer to a graph with the nodes representing objects and the edges representing their interactions, than to a sequence of methods. Ressia et al. described this discrepancy between the representation of objects and the perspectives of debugging tools as a *conceptual gap* [Ressia 2012b].

Practical consequences. When developers want to know how objects behave during the program execution they modify the lines of code and use low level debugging operations [Beller 2018]. These operations such as print statements or *breakpoints* are code instructions that pause the execution and open a traditional call-stack based debugger, showing the program state before pausing. However, in object-oriented programming languages the lines of codes defining methods behavior, hence objects' behavior are written in classes. During program execution, all objects instantiated from a class share the same methods. This means that the low level instructions will be activated for every object that intends to perform the behavior defined by methods containing a debugging instruction. This increases the cognitive load for developers and the number of actions that they must perform to filter out debugging information unrelated to the objects they are analyzing [Hinkle 1993, Ressia 2012b]. In some object-oriented environments, particularly live programming environments, these debugging instructions may be activated so frequently that they cause the program to crash [Hinkle 1993].

2.2.2 Accessing Particular Objects

Developers must gain access to an object if they want to observe it. However, while trying to reach some objects in the execution, developers have to search within the entire living memory of the program. For the objects developers are trying to reach and observe, often an object that presents the symptom of a bug, we can often find many similar objects instantiated from the same class [Hinkle 1993]. Developers must search within these objects, all potential candidates to the observation.

Practical consequences. To perform this research of objects that are relevant to them among all potential candidates, developers have to use breakpoints, to pause the execution where they suppose relevant objects come into action. Because this process relies on assumptions of where relevant objects are involved during the execution, developers might have to place several breakpoints. When their assumptions prove to be wrong, meaning that the breakpoints they used are misplaced or missed a critical point when navigating forward in the execution, developers have to restart the program, using new breakpoints and careful navigation steps [Willembrinck Santander 2023]. When the program being debugged is non-deterministic, manipulates or create unpredictable values such as

randomly generated identifiers, time stamps, environment variables, inputs from external resources, two executions of that program might differ. The objects manipulated during a first execution might be different in a second execution, that is, if they are created again.

2.3 Debugging Object-Oriented Programs, the Requirements

In this section, we list the requirements a debugging approach must fulfill in order to address the challenges of debugging object-oriented programs.

2.3.1 Requirement for Observing Particular Objects

As aforementioned, to better understand the behavior of objects during program execution, developers need to monitor, or modify the attributes and interactions of objects at various stages of program execution [Sillito 2006, Sillito 2008, Kubelka 2019]. These facilities align with the definition of *Object-Centric Debugging*, an approach that consist in focusing debugging tools and operations on particular objects [Ressia 2012b]. Therefore, in the following of this document, we will use the adjective *object-centric* when referring to debugging tools or methodologies providing developers with such features. To illustrate the concept of object-centric debugging, Ressia et al. introduced the *object-centric breakpoints*. Unlike conventional breakpoints, which are not specific to one object, these breakpoints pause the execution and open a debugger when a chosen object's methods are invoked or its attributes modified during execution [Ressia 2012b].

Object-centric versus conditionals. One could argue that any debugging tool supporting the execution of object-oriented programs should be considered as object-centric. However, the challenge for observing particular objects, which Object-Centric Debugging aims at addressing, is due to the fact that debugging tools shipped by default with object-oriented programs often lack sufficiently high level features to effectively interact with and visualize how individual objects behave. Below, we illustrate this *conceptual-gap* [Ressia 2012b] between object-centric and conventional debugging tools. Ressia et al. proposed an object-centric breakpoint named the *Halt on call*. This breakpoints was designed to help developers finding when a method is being invoked on a particular object [Ressia 2012b] by selecting the target object. To find a method invocation on a particular object with a conventional breakpoint, developers would have to specify a condition [Beller 2018] in the source code so that this breakpoint only activates when the object executing the targeted method matches the chosen object. To write

such conditions developers refer to the object executing a method at runtime using language features such as `this` or `self`. To compare such object with the object they want to pause the execution for, developers also use a reference or an identifier (e.g., a hash). Ultimately a condition pausing the execution for a particular object can look like the following example: `self hash == my_object_hash`. Because object-centric breakpoints allow developers avoiding the need to write a condition, they are expected to require less effort from developers to debug object-oriented programs than conditional breakpoints . Additionally, while our *Halt on call* example seems easy to implement with a conditional breakpoint, there is more complex examples like the *Halt on read* or *Halt on interactions*, for which developers would potentially need to apply several conditional breakpoints and think about more complex conditions under which to pause the execution for chosen objects. However, thinking under which condition a particular object of the program can be observed might be difficult [Beller 2018], and error-prone [Damevski 2016], which makes conditional breakpoints impractical for debugging objects.

Object-centric definition. Following this discussion about object-centric versus conditionals, we precise our definition of object-centric. Not only an object-centric debugging solution must help developers monitoring or modifying objects' attributes and interactions, but it also relies on mecanisms for manipulating objects at runtime or capturing traces of their behavior. This more precise definition excludes from the term "object-centric", approaches tied to source code or call stack representation such as conditionnal breakpoints, even though they can serve as a low-level foundation for implementing object-centric debugging solutions.

> **Requirement 1: object-centric**. We argue that, to support the challenge of observing particular objects with a program execution, a debugging tool or methodology must be object-centric.

2.3.2 Requirements for Accessing Particular Objects

Developers can only observe specific objects if they have references to them. However, as previously mentioned, obtaining such references is challenging (cf. subsection 2.2.2) because debugging is error-prone and often requires restarting the program multiple times (cf. section 2.1) to target objects exhibiting unexpected behavior. This becomes especially difficult with non-deterministic programs, where unpredictable values cause each restart to produce a potentially non-reproducible execution path.

Omniscient debuggers overcome this limitation by recording execution histories [Lewis 2003], capturing non-deterministic values in the process. Capturing

these values ensures that developers will observe the same execution after reproducing the bug once. This avoids the need for developers to restart their exploration repeatedly to search for objects to debug.

> **Requirement 2: omniscient**. We argue that, to support the challenge of accessing particular objects within a program execution, a debugging tool or methodology must be omniscient.

Existing literature has proposed multiple omniscient debugging approaches, however the terminology used to categorize these approaches seems inconsistent. Consequently, in this thesis we identify the two distinct categories of omniscient debugging approaches, which are outlined below. *Back-in-time* approaches [Ko 2004, Hofer 2006, Pothier 2007, Ko 2008, Ko 2009, Pothier 2009, Thiede 2023a, Thiede 2023b] perform a full record of a program execution and provide ways to explore it afterward. *Time-traveling* approaches [Lienhard 2009, Wang 2014, Willembrinck 2022b] perform a complete or partial recording of an execution and replay that execution to allow live exploration of the program.

The ability of omniscient approaches to address the challenge of accessing particular objects within the execution depends on what features they provide to help developers exploring the execution history. Indeed, retrieving a specific object within the recorded execution of an omniscient debugger is a manual process that can be as tedious as when using non-omniscient debugging tools, especially for large program executions [Willembrinck Santander 2023].

> **Requirement 3: scoping features**. We argue that to support the challenge of accessing specific objects during program execution, a debugging tool must provide features that help developers identify and access objects in an execution history.

2.3.3 Requirement for Reducing the Cognitive Load

Object-centric debugging approaches aim to lower the cognitive effort needed to debug by proposing debugging perspectives that align more closely with the abstractions and mental models of programs provided by object-oriented programming languages [Ressia 2012b]. However, as described earlier in section 2.1, part of the cognitive load hindering the debugging activity is also due to the iteration process developers go through while debugging [Vessey 1986, Gilmore 1991, Yoon 1998, Spinellis 2018]. This iteration process, is mainly due to the difficulty of maintaining mental models [Gilmore 1991, Ko 2005, O'Dell 2017] of programs and to the assumption that these models contain [Ko 2005]. In particular, develop-

ers report forgetting part of their program exploration and previous conclusions, requiring them to repeat debugging actions to try answering again previous hypotheses [Sillito 2006, Sillito 2008].

Requirement 4: debugging iterations support. We argue that, to further reduce the cognitive load associated with debugging of object-oriented programs, object-centric debugging tools should provide better object-oriented perspectives while helping developpers maintaining their train of thoughts and the state of their inquiry throughout the debugging session.

2.4 Object-Centric Debugging, the State of the Art

In this section, we explore the object-centric debugging approaches identified in existing research. We examine each approach to determine how well it satisfies the requirements specified in section 2.3. From the literature, three categories of object-centric debugging approaches can be distinguished. These categories include object-centric breakpoints and instrumentations, interrogative or scriptable debugging tools, and debugging tools that provide graphical visualizations.

2.4.1 Object-Centric Breakpoints and Instrumentations

Object-centric breakpoints [Ressia 2012b] and instrumentation solutions [Kiczales 1997, Yin 2013, Dupriez 2019, Costiou 2022], enable interaction with specific objects involved in an ongoing program execution. Because of this latter aspect, these types of approaches are sometimes referred to as live, online, or interactive debugging approaches.

Object-Centric Breakpoints As mentioned in section 2.3, object-centric breakpoints pause the execution when a chosen objects are called or modified during execution [Hinkle 1993, Ressia 2012b]. *Declarative breakpoints* extend the same idea to a set of objects [Corrodi 2016]. When proposing object-centric breakpoints, Ressia et al. specify that these breakpoints can only augment traditional debugging tools, not replace them [Ressia 2012b], because they apply directly to an object's reference. Indeed, the most common way to find such a reference is to place a standard breakpoint in the code where the chosen objects can be used. Then, once the call-stack debugger opens, use the call-stack to search for references to those objects. While they do not apply directly on objects, and therefore are not entirely consistent with our definition of object-centric, advanced breakpoints, such as *Stateful* [Bodden 2011a] and *Temporal* [Pasquier 2023] breakpoints

also help narrowing the *conceptual gap* described by Ressia [Ressia 2012b]. *Stateful breakpoints* [Bodden 2011a] enable developers to reference an object as a criterion for combining and activating different line breakpoints, thereby expressing pausing conditions across several locations in the source code [Bodden 2011a]. *Temporal breakpoints* [Pasquier 2023], go further by enabling developers to define breakpoints based on the programming language and through concurrent executions.

Object-Centric Instrumentations Instrumentation involves modifying a program's source code to add instructions for monitoring, diagnosing errors, and understanding the program's internal state. Yin et al. proposed *Pointcut*, a language allowing developers to express pausing conditions by describing expected objects' state and behavior [Yin 2013]. Pointcut is based on *Aspect-Oriented Programming (AOP)* [Kiczales 1997]. In object-oriented programs, *aspects* encapsulate the instrumentation instructions to apply on methods. Other techniques rely on object *proxies and anonymous subsclasses* [Costiou 2020a, Costiou 2022]. Object proxies are attached to objects to intercept the interactions coming to that object. These proxies can be combined with *message passing control* [Ducasse 1999] to install instrumentation only when the behavior to be instrumented is activated. By migrating objects to anonymous subclasses [Hinkle 1993, Costiou 2020a, Costiou 2022], one can override the original behavior of objects so that they execute the instrumentation instructions. When combined, proxies, anonymous subclasses, and message-passing control can constitute the foundation for building a *metaobject protocol* [Kiczales 1991], which allows developers to extend fundamental aspects of object-oriented programs.

Requirements and Limitations To the best of our knowledge, object-centric debugging tools based on breakpoints and instrumentation do not provide support for debugging *non-deterministic* programs, as they do not enable developers to observe similar objects throughout several program executions. Although *Temporal breakpoints* [Pasquier 2023] enable developers to reason about the past and future of non-deterministic concurrent execution, to the best of our knowledge, they do not address other possible sources of non-determinism, such as system calls. Additionnaly, the approaches mentioned in this section do not provide support for developers when they need to iterate over the breakpoints they place during a debugging session to verify their assumptions, as described in section section 2.1. Furthermore, as they cannot guarantee the manipulation of the same objects within an execution, object-centric breakpoint and instrumentation solutions do not offer developers scoping features to facilitate isolation and observation of particular objects.

2.4.2 Object-Centric Omniscient Debugging Approaches

In this subsection, we present the object-centric omniscient debugging approaches that we found in the literature, and explain how they address our list of requirements for supporting the debugging of object-oriented programs. Table 2.1 summarises the positioning of the different approaches that we present. These approaches provide features that help developers explore execution histories by offering queries, enabling scripted exploration, or providing graphical views.

Table 2.1: This is a mapping of object-centric omniscient debugging approaches with the requirements for supporting the challenges of debugging object-oriented programs. A checkmark (✓) indicates that the corresponding approach supports the requirement. A line (–) indicates partial support. A cross (×) indicates that the requirement is not supported.

Object-centric debugging tools	Omniscient	Scoping features	Support iterations
Interrogative and scriptable debugging tools			
Query based OOP debugging[1]	✓	–	×
Unstuck[2]	✓	–	×
Whyline[3]	✓	–	✓
TOD[4]	✓	–	✓
Expositor[5]	✓	–	×
Seeker[6]	✓	✓	–
Trace Debugger[7]	✓	–	×
Debugging tools providing visualizations			
JIVE[8]	✓	–	×
Compass[9]	✓	–	✓

References

[1] [Lencevicius 1997, Lencevicius 2003]
[2] [Hofer 2006]
[3] [Ko 2009, Ko 2008, Ko 2004]
[4] [Pothier 2009, Pothier 2007]
[5] [Phang 2013]
[6] [Willembrinck 2021, Willembrinck 2022a] [Willembrinck 2022b]
[7] [Thiede 2023a, Thiede 2023b]
[8] [Gestwicki 2005]
[9] [Lienhard 2009]

Interrogative and scriptable debugging tools *Expositor* [Phang 2013] combines scripting and time-travel debugging to allow programmers to automate debugging tasks. From an execution, Expositor generates traces that developers manipulate as lists with operations such as map and filter. Such operation can help developers scoping their exploration around particular objects within an execution. *Whyline* [Ko 2009, Ko 2008, Ko 2004] generates and answers *"Why did? and Why did not?"* questions. By selecting instance variables of objects developers can ask question about the origin of the instance variable value. For instance, developers can ask the question *"Why did the color of this table change to blue?"*, where the color is referenced by the instance variable `color` of an object of type `Table`. *TOD* [Pothier 2009, Pothier 2007] allows developers to ask similar *Why?* questions about the origin of an instance variable value. TOD does not displays the answers in the form of a graphical visualization as Whyline does (cf. section 2.4.2), but brings the highlight to the line of code from which the value originate. While JIVE puts the accent on the visualizations of object-oriented programs' executions, like the tools we presented previously, it also allows searching for the origin of an instance variable. *Unstuck* [Hofer 2006] provides a language, an API, that expose the recorded events' properties. This API allows developers to express queries in order to filter the call-stack of the program execution. Although Unstuck and TOD do not share the same querying feature, they are similar debuggers. They both record and present the same execution events in a similar manner. Both record the same events and present them in a similar manner. *Seeker* [Willembrinck 2022b] supports the execution of *Time-traveling queries (TTQs)*. TTQs allow developers to formulate questions about the execution of a program and to time-travel between the results provided by the debugger. Seeker provides a pre-built list of object-centric queries. The set of queries is extensible so developers can write their own to customize how they explore the execution. *Trace Debugger* [Thiede 2023a, Thiede 2023b] supports queries about specific objects using objects' protocol of supported messages, or queries about views of the object state changes. Lencevicius et al. proposed a debugger allowing to express queries to answer to before running a program [Lencevicius 1997]. While we do not consider this approach as an omniscient debugging approach since it cannot access all program states after its first execution, this omniscient capabilities where brought later on by the authors in a second proposition [Lencevicius 2003].

Debugging tools providing visualizations *Compass* [Lienhard 2009] tracks and displays the flow of objects in program execution. The flow of an object refers to the events (*e.g.*, an object is created, passed as a parameter, etc.) that the object encounters during its life cycle. Compass uses green arrows to highlight an object's flow. These arrows show the object's path in a execution tree visualization representing all the activated methods and their relationships during program execution. As mentioned in the previous paragraph,

Whyline [Ko 2009, Ko 2008, Ko 2004] is primarily a querying approach. However, it provides developers with a graphical view of the sequence of interactions that answer their queries. The *Java Interactive Visualization Environment (JIVE)* [Gestwicki 2005] provides developers with detailed and compacted UML visualizations such as the *Sequence Diagram* or *Object Diagram*. While *JIVE* [Gestwicki 2005] seems to be primarily a debugging approach based on visualizations, it also provide a querying interface, supporting PractQL, a variant of SQL [Blanton 2012] and declarative queries [Czyz 2007], providing updated visualizations as answers.

Requirements and Limitations Of the approaches we mentioned, *Seeker* [Willembrinck 2022b] is the closest to fully supporting a scoping feature that enables developers to retrieve a specific object within an execution history. Thanks to its predefined list of time-traveling queries, such as *Find all instance creations of a class*, *Seeker* is ready to help developers search for objects, out of the box [Willembrinck Santander 2023]. To the best of our knowledge, the other interrogative debugging tools we mentioned in this section, *Query based OOP debugging* [Lencevicius 1997, Lencevicius 2003], *Unstuck* [Hofer 2006], *Expositor* [Phang 2013], and *JIVE* [Gestwicki 2005] require developers to write scripts or conditions to search the history for objects of interest. Since learning a new protocol and writing scripts or conditions incurs additional cognitive cost [Beller 2018, Damevski 2016] compared to using ready-to-use features, we argue that these tools only partially support the requirement for a scoping feature. Tools such as *Whyline* [Ko 2009, Ko 2008, Ko 2004], *TOD* [Pothier 2009, Pothier 2007], and *Trace Debugger* [Thiede 2023a, Thiede 2023b] provide ready-to-use queries to help developers analyze the flow of objects. However, *Whyline and TOD*, are limited to *Why? and Why did not?* queries, and *Trace Debugger* is limited by the methods supported by the objects present in execution histories. This may also limit the ability of developpers to search for particular objects. Finally, although *Compass* [Lienhard 2009] enables developers to visualize the evolution of a given object during execution, to the best of our knowledge, it does not provide scoping features that would help developers filter the execution history and focus on particular objects.

Whyline, *TOD* and *Compass* support developers' debugging iterations. These tools allow developers to place bookmarks that reference events within the execution history. These bookmarks can serve as summaries of debugging sessions and as ways to revisit parts of the exploration process. *Seeker* enables developers to execute follow-up queries from previous query results, allowing them to refine their debugging exploration. Although *Seeker* provides a wider range of queries than *Whyline* and *TOD*, it does not allow developers to mark or navigate to previous stages of their exploration. Therefore, it offers partial support for de-

bugging iterations. As far as we know, the other tools we mentioned do not allow to execute follow-up requests based on previous query results without first storing those results and updating the initial query's conditions. Additionally, these approaches do not enable developers to mark a stage of their exploration so that they can return to it later. To the best of our knowledge, none of the other tools we mentioned enable the execution of follow-up requests based on previous query results without first storing those results and updating the initial query's conditions. Furthermore, these approaches do not allow developers to save a stage of their exploration to return to later.

2.5 Object-Centric Debugging Evaluations

As mentioned at the beginning of this chapter in section 2.1, previous work has highlighted that the difficulties associated with debugging are partly due to the tools themselves. These debugging tools are sometimes considered too complex and unsuitable, and they are often absent from educational training [Perscheid 2017, Beller 2018]. In order to make it easier for developers to debug object-oriented programs and to determine which research direction to focus on, in this section we present a brief overview of the studies conducted for each of the object-centric debugging approaches we listed earlier. This overview is summarized in Table 2.2 and use the following terminology of research design from Creswell et al. [Creswell 2017]. *User studies* are empirical experiments involving developers. They can focus on qualitative (QL) aspects of debugging, using observations and interviews, or quantitative (QT) aspects, when monitoring metrics such as time to debug or number of actions. *Case studies* are extensive descriptions of a real debugging scenario (i.e., with a real bug) using a given approach that discuss the approach's impact and limitations. We classify those focusing only on presenting a debugging tool as *evaluations through example*.

From Table 2.2 we can observe that the majority of approaches have been evaluated with case studies or examples demonstrating and discussing how tools that implement these approaches work. Some researchers also provided benchmarks [Lencevicius 1997, Lencevicius 2003, Hofer 2006, Pothier 2007, Pothier 2009, Ressia 2012a, Yin 2013] to illustrate the capacity of their approach to process a maximum number of cases efficiently in terms of calculation but especially in terms of memory space, an aspect often criticized in omniscient approaches. In our litterature of object-centric debugging, two approaches, *Whyline* [Ko 2009, Ko 2008, Ko 2004] and *Seeker*[Willembrinck Santander 2023] have been evaluated using a user study design involving developers. The subsections below detail these evaluations and their results. Our goal is to determine if and how the object-centric debugging approach can help developers debug in practice.

Table 2.2: Mapping of object-centric debugging approaches and their evaluation methods, indicated by a checkmark (✓). Several checkmarks on the same line indicates that the debugging approach has been evaluated under different research design.

Object-Centric Debugging Tools	User Study		Case Study	Example	Benchmark
	QL	QT			
Breakpoint and instrumentation debugging tools					
Stateful[1]				✓	
Object-Centric breakpoints[2]				✓	✓
Pointcut[3]				✓	✓
Reflectivity[4]				✓	
Temporal breakpoints[5]			✓		
Interrogative and scriptable debugging tools					
Query based OOP[6]				✓	✓
Unstuck[7]				✓	✓
Whyline[8]	✓	✓			✓
TOD[9]				✓	✓
Expositor[10]			✓		
Seeker[11]		✓			
Trace Debugger[12]			✓		
Debugging tools providing visualizations					
JIVE[13]				✓	
Compass[14]				✓	✓

References

[1] [Bodden 2011b]
[2] [Hinkle 1993, Ressia 2012b]
[3] [Yin 2013]
[4] [Costiou 2020a, Costiou 2022]
[5] [Pasquier 2023]
[6] [Lencevicius 1997, Lencevicius 2003]
[7] [Hofer 2006]
[8] [Ko 2009, Ko 2008, Ko 2004]
[9] [Pothier 2009, Pothier 2007]
[10] [Phang 2013]
[11] [Willembrinck 2021, Willembrinck 2022a] [Willembrinck 2022b]
[12] [Thiede 2023a, Thiede 2023b]
[13] [Gestwicki 2005]
[14] [Lienhard 2009]

2.5.1 Whyline

The authors of Whyline [Ko 2004] performed an empirical experiment with Master's students, 4 participants in control, 5 participants in treatment. Participants were videotaped and followed the think-aloud protocol so that their goals, strategies and intentions could be interpreted later. The experiment seems to follow a between participants design where the control group is compared to the treatment group. Following this experiment the authors reported that participants can solve 40% more tasks, and debug eight times faster when using Whyline.

Later, the authors performed another user study [Ko 2008] during which 9 participants debugged a color slider scenario. The bug caused the slider to not produce the expected color when moved. The selected population is heterogeneous, with participants coming from all kinds of backgrounds, from non-programmers to experts. The authors compared the results of the 9 participants with the results of 18 self-proclaimed Java experts who participated in a prior study [Ko 2006] with the same scenario. The authors concluded that, with Whyline, the participants were twice as fast as the experts from the prior study [Ko 2006] would be without Whyline.

The last empirical evaluation of Whyline [Ko 2009] involved 20 participants, 10 in control and 10 in treatment. For the control and treatment tasks, the authors used real bugs, from ArgoUML. Treatment group seemed to be faster and have a better ability to solve bugs than the control group. During these user studies, Ko et al. also made qualitative observations. They reported on some of the questions that developers ask when debugging the various features of the debugger [Ko 2004, Ko 2008] and discussed the differences between novice and experienced developers [Ko 2008]. The Whyline authors also reported participants' perceptions during the aforementioned evaluation, which suggests that developers appreciate Whyline's features [Ko 2008].

Overall, the results obtained by the authors of *Whyline* show that omniscient object-centric debugging approaches based on querying features and providing exploration bookmarks are promising a solution for helping developers debug. However, further experimentation is needed to better understand the usefullness of the object-centric debugging approach. In particular because *Whyline's* queries are limited to *Why? and Why didn't?* and therefore represent only a subset of the object-centric debugging questions as proposed by Ressia et al. [Ressia 2012b]. Moreover, in the study [Ko 2006] that precedes [Ko 2008], we could not find a mention to the 18 participants announced in [Ko 2008]. To our knowledge, only 10 of the announced 18 participants are mentioned. Additionally, in their latest study [Ko 2009], the authors reported *"The debugger used in our control condition imposed limitations on participants' ability to control the live program and both conditions were disallowed from editing the program"*. Therefore, the results of the control group are not perfectly representative of developers' usual behavior (i.e.,

the baseline for comparison) and may have influenced the authors' observations in favor of Whyline. To verify the results of Whyline's evaluations and ensure their generalizability, we suggest conducting additional empirical experiments.

2.5.2 Seeker

Willembrinck *et al.* investigated the impact of *Seeker* and the Time-Traveling Queries on program comprehension, using 14 program comprehension tasks. Of these tasks, two required to use the object-centric options of the time-travel back end [Willembrinck 2021]. The experiment followed a repeated-measures design [Seltman 2015] with 34 participants. The authors measured for each task the time taken required by participants understand the task, the precision of the participant's answer, and the number of debugging actions. They compared the results obtained with Time-Traveling Queries and those obtained with standard Pharo[Black 2009] debugging tools. The evaluation showed shows significant improvement in program comprehension and debugging performance (i.e., developers needed less time to debug using TTQs). However, it was not possible to draw any conclusions about the two object-centric tasks in isolation. Conducting further studies with different bugs and focusing on Time-Traveling Queries related to object behavior, could complement the results of this experiment and bring additional knowledge about object-centric debugging.

2.6 Conclusion

In this chapter, we first identified the general challenges of debugging software. Then, we discussed the conceptual gap between object-oriented programs and the debugging features of common call-stack-based debuggers.

We created a list of requirements that a debugging approach must fulfill in order to facilitate the debugging of object-oriented programs. In particular, we presented Object-Centric Debugging, an approach that fits this objective, as it aims to bridge the conceptual gap by providing developers with advanced tools to observe object behavior. Next, we identified debugging solutions in the literature that offer similar or identical features.

From this literature, we learned that most existing object-centric tools do not offer ready-to-use queries to enable developers to find and inspect the behavior of objects during execution, nor do they allow developers to return to previous steps in their debugging sessions. We argue that these features are essential to facilitate debugging object-oriented programs and encourage adoption of object-centric debugging tools. In this chapter, also we highlight that most existing object-centric tools have been evaluated using examples, case studies, and benchmarks. However, the lack of user studies involving developers means that the scientific com-

munity lacks knowledge about the real impact of object-centric debugging approaches on debugging. For example, it is possible that object-centric debugging is a misguided approach, or, as we believe, that developers are missing an opportunity due to a lack of solutions adapted to real-world situations.

For these reasons and to facilitate debugging object-oriented programs, the following chapters of this thesis aim to provide additional empirical knowledge on the impact of object-centric debugging on debugging activities. They will also compare interrogative approaches with approaches based on breakpoints or instrumentation using case studies. Finally, the last chapter presents our proposed debugging solution, which aims to meet the requirements we defined.

Chapter 3

Evaluating the Impact of Object-Centric Breakpoints on Debugging

Contents

To investigate the impact of the object-centric debugging approach on developers' debugging activities, we designed and conducted a controlled experiment involving 81 developers. In order to evaluate its impact, we needed to select an object-centric debugging tool that implements the object-centric debugging approach. We selected object-centric breakpoints because they are the only debugging tool among those listed in our state-of-the-art research that has an integrated implementation in a production version of a language and development environment. In particular three object-centric breakpoints were integrated into the Pharo IDE [Black 2009] in 2019 and are still available for use in new Pharo

versions ever since. Object-centric breakpoints are complex to implement (illustrated later in chapter 4). They require the creation of a back end to intercept all method invocations on the object, as well as all accesses and assignments to its instance variables (object attributes). Using a stable version of the debugging tools allowed us to start the experiment without any preliminary engineering work.

In our experiment, participants completed two debugging tasks, one using object-centric breakpoints and another without, taking an average of one hour and thirty minutes to complete the entire study. Our analysis of the experiment results, detailed in this chapter, shows that object-centric breakpoints seems to facilitate debugging. However, we also found that object-centric breakpoints might hinder developers' debugging approaches when the bugs are located at program or software initialization. In this chapter, we discuss the implications of these results for developers and detail research perspectives for further studying the impacts of the object-centric debugging approach.

3.1 Object-Centric Breakpoints

Below, we present the implementation of object-centric breakpoints [Hinkle 1993, Ressia 2012b] that we evaluate in our empirical experiment. This implementation has been integrated in the production version of the Pharo IDE [Black 2009] since 2019.

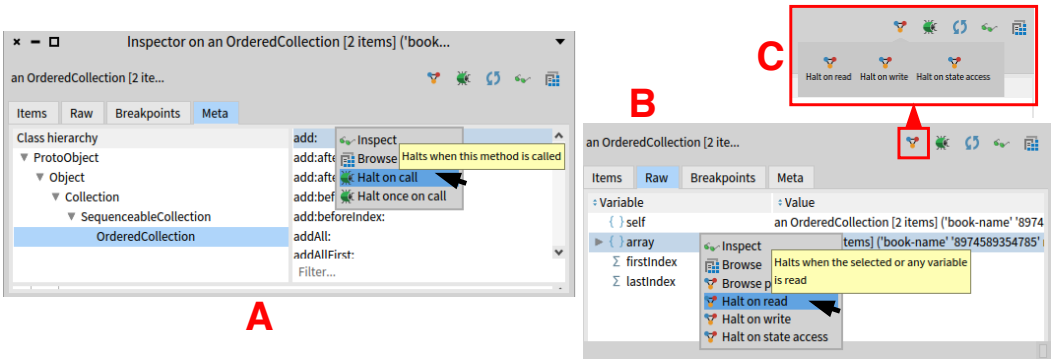


Figure 3.1: The Pharo *Inspector* opened on a collection object, showing the integration of different object-centric breakpoints. Developers can install: (A) object-centric method breakpoints on the objects' methods, (B) object-centric field breakpoints on the objects' variables, and, from the top menu, (C) general field breakpoints.

To apply a breakpoint on a given object, the Pharo environment relies on another debugging tool—the inspector. The latter allows a developer to observe any object in the memory of the executed program, in terms of its state (*i.e.*, the con-

tent of the instance variables, Figure 3.1 B) and the methods it can execute (Figure 3.1 A). The inspector is embedded in the debugger and displays the values of variables in use within the currently pauseed method. Developers can place the following object-centric breakpoints on the inspected objects:

Method-specific breakpoint. This breakpoint pauses the execution before the target object executes a specific method. Ressia *et al.* [Ressia 2012b] have proposed it as way to help when many instances of the same class execute the same method (*e.g.,* in a loop), but developers are interested in pauseing the execution only when a particular object executes that method. Developers install an object-centric method breakpoint through the contextual menu of a method, by selecting *Halt on call* (Figure 3.1 A) of the *Meta* pane.

Field breakpoint. This breakpoint pauses the execution when one of the instance variables, attributes of a particular object, is accessed or modified. It is intended to help when many instances of the same class are modified during the execution, but developers are interested in tracking state changes in only one target object [Ressia 2012b]. Developers install an object-centric field breakpoint through the contextual menu of an instance variable, by selecting *Halt on read/write* (Figure 3.1 B) of the *Raw* pane. The *Halt on state access* (Figure 3.1 B) pauses the execution when the selected instance variable is being either read or written to.

General field breakpoint. This breakpoint is a generalization of the field breakpoint. It pauses the execution when any attribute of a target object is accessed or modified. In Pharo, these breakpoints can be activated using the top menu of the inspector (Figure 3.1 C).

3.2 Research Methodology

Our overarching goal is to make a step towards understanding whether and how object-centric debugging improves the debugging of object-oriented programs. Towards this goal, we aim to study, through a controlled experiment, the impact of object-centric breakpoints on (1) the ability of developers to fix bugs, (2) the number of interactions/actions with the debugging tools required to fix bugs, and (3) the time taken to fix bugs.

3.2.1 Research Questions

We structured our investigation around three research questions.

RQ$_1$. How do object-centric breakpoints affect developers' ability to fix bugs?

For object-centric breakpoints to be beneficial to the process of debugging, it is essential not to hinder the ability of developers to fix bugs. Since object-centric breakpoints are claimed to improve debugging [Ressia 2012b], we formulate the following null hypothesis:

H0$_1$ Object-centric breakpoints do not affect the ability of developers to fix bugs.

RQ$_2$. How do object-centric breakpoints affect the number of debugging actions?

Current evaluation scenarios [Ressia 2012b] suggest that debugging with object-centric breakpoints reduces the number of actions that developers perform with debugging tools. Since we want to evaluate the effect of object-centric breakpoints, whether positive or not, we formulate the following null hypothesis:

H0$_2$ Object-centric breakpoints do not affect the number of debugging actions developers perform to fix bugs.

RQ$_3$. How do object-centric breakpoints affect the time taken to debug?

If object-centric breakpoints are expected to reduce the number of debugging actions, we also expect they shorten the time needed for debugging. Therefore, we formulate the following null hypothesis:

H0$_3$ Object-centric breakpoints do not affect the time developers take to fix bugs.

3.2.2 Experiment Flow

To study our hypotheses, we conducted an empirical experiment with 81 participants. Figure 3.2 presents the sequence of the seven steps we asked our participants to follow within the Pharo IDE in autonomy, using their own computer, operating system, and work environment of choice. The steps are carried out in order by a wizard tool [Steven Costiou 2021], and material is available in the replication package [Bourcier 2025b].

(1) Welcome instructions We inform our participants about the objective of the experiment and announce its estimated duration to be approximately one hour and thirty minutes. We provide the instructions needed to setup the experimental

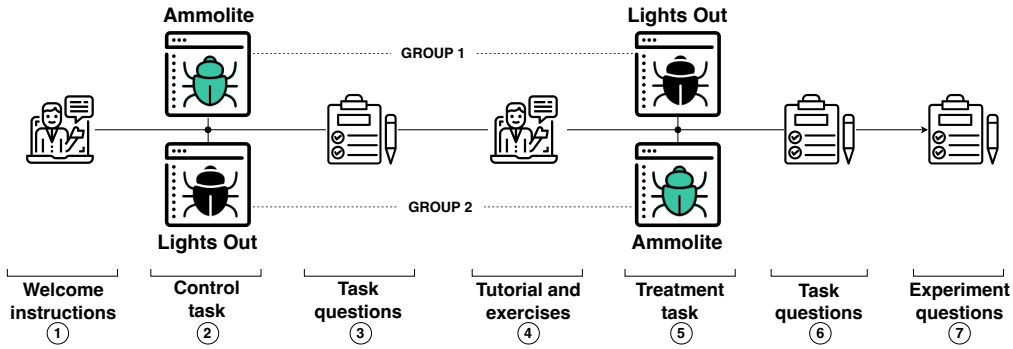


Figure 3.2: Sequence of the seven steps followed by the experiment participants.

environment (*i.e.,* the wizard), detail the steps and explain how to use the experiment wizard tool [Steven Costiou 2021]. When starting the experiment, the participants are asked to consent to their data being collected (with a default opt-out). If they accept, they can proceed with the experiment and are randomly assigned to one of the groups (Figure 3.2). To ensure that participants only use standard tools, they are explicitly instructed not to import advanced tools, such as plugins or libraries into the IDE. Finally, the participants were informed that they had flexibility to work at their own pace and according to their own schedule. However, to ensure the accuracy of the results, we requested that they try to complete the experiment in a single uninterrupted session whenever possible.

(2) Control task The wizard presents the participants with a description of their first bug, either in Ammolite or Lights Out, depending on the participants' group (step 1). They are asked to fix the bug using the standard debugger, inspector, and code navigation and modification tools. After completing the task, participants are asked to explain the cause of the bug and how they resolved it.

(3) and (6) Post-task questions After control (step 2) and treatment (step 5) tasks we ask participants a set of questions. The poll of questions comprises open questions (asking for complementary information on participants' understanding of the bug), questions with multiple choices (e.g., to evaluate how long participants were interrupted while performing the task), and Likert scale questions to get feedback on the task and on the user experience the tool offers.

(4) Tutorial and exercises As soon as a participant starts this step of the experiment, the wizard activates the object-centric breakpoints. Participants are then presented with a tutorial explaining with text and video support the concepts behind object-centric breakpoints and how to use them in Pharo, as presented in section 3.1. We ask participants to complete two exercises to apply the knowledge gained from the tutorial.

(5) Treatment task The wizard provides the description for the second bug of the experiment. As for the control task (step 2), the bug assigned in treatment depends on the participants' group (step 1). Participants are encouraged to try using the object-centric breakpoints, using the object-centric version of the inspector (Figure 3.1) in addition to the standard debugging and source code modification tools. Similarly to the control task (step 2), participants are asked to explain the cause of the bug and how they resolved it.

(7) Post-experiment questions Once both tasks are completed, we ask participants to provide demographic information and any additional feedback on the experiment.

3.2.3 Experimental Design

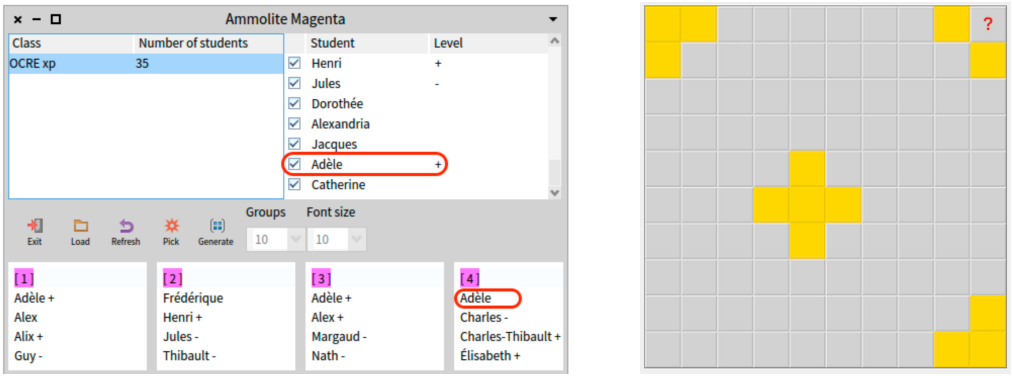
Every participant underwent a control condition using standard Pharo tools to debug a task and a treatment condition using object-centric breakpoints to debug another task (Figure 3.2 steps 2 and 5).

Although this resembles a within-participants design [Christensen 2015], comparing each participant under different conditions presents risks related to the unknown impact of object-centric breakpoints and to the difficulty of assessing task complexity [Rein 2023]. First, object-centric breakpoints are uncommon tools and we do not know what effect they might have on debugging. The tasks must be complex enough: if they are too trivial, the object-centric breakpoints may not show any difference from standard tools. This limits the number of tasks that participants can perform, as adding too many tasks risks making the experiment excessively long. Second, in a within-participant design with a limited number of tasks, the tasks have to be different enough to minimize learning effects, yet similar enough to allow for meaningful comparisons across different conditions.

Given our current understanding of the impact of object-centric breakpoints, we cannot ensure that two different tasks, even if similar in some aspects, would be comparable under different conditions. We therefore opted for a between-participants design [Christensen 2015] where independent measures (control and treatment) are then compared per task.

3.2.4 Experimental Tasks

We designed the control and treatment tasks to mirror the object-centric breakpoints scenarios [Ressia 2012b] described in section 3.1. To ease the presentation of the tasks, we label these scenarios as follows: Scenario I corresponds to the *Object-centric debugging field breakpoint* scenario, while Scenario II encompasses both the *Object-centric debugging method-specific breakpoint* and *Object-centric debugging general field breakpoints* scenarios.



(a) Ammolite (b) Lights Out

Figure 3.3: Ammolite (a) and Lights Out (b) applications with the bug symptom highlighted in red.

Object-centric debugging assumes that developers can access the problematic object within the program execution [Ressia 2012b, Costiou 2020b]. To ensure this assumption is met, we selected applications with graphical user interfaces, and bugs with a single problematic object. We made these problematic objects accessible by right-clicking on the graphical interface. Figure 3.3 presents the applications of the tasks.

Ammolite (a) It is a real application for teachers to create balanced groups of students based on their grades, indicated by markers (+ for above average and – for below average). To generate and display the groups, the user clicks the "Generate" button on the graphical interface.

A bug was discovered in the production version, where a student's marker was missing in the generated groups display. This bug matches an object-centric debugging scenario because it requires accessing the problematic object's `marker` attribute, to observe its updates. This observation is challenging to do using a standard debugger due to the attribute's declaration in multiple methods (Scenario I). Additionally, fixing this bug requires understanding how the problematic object is displayed on the graphical interface and therefore identifying the methods executed by the problematic object and their caller. The latter is complicated by the presence of identical student names and markers in the promotion (Scenario II).

Lights Out (b) Originally a training exercise for learning Pharo, participants may have seen an implementation of this game before the experiment. However, they were unaware of the bug we introduced since we devised it from scratch. The game features a grid where each tile represents a light, on when yellow and

off when gray. Clicking on a tile turns it on along with the adjacent lights (top, right, bottom, and left). The goal is to turn all lights on.

The bug symptom is that one of the corner lights is not switchable, its color never changes. The symptom shifts to a new corner each time the application is restarted. Similarly to Ammolite, this bug corresponds to an object-centric debugging scenario as it demands accessing the problematic object's `color` attribute to monitor its usage and updates (Scenario I). In addition, it requires comprehension of how the switch feature of the lights operates, which involves identifying the methods executed by the problematic object. This bug is complicated by the 100-tile grid, where each corner has an equal chance of exhibiting the bug (Scenario II).

Task differences We designed the tasks so that one should not take significantly longer than the other to solve using standard debugging tools, and ensured the bugs were of a type commonly encountered in Pharo. However, the tasks differ in how the bug is reproduced. In Ammolite, each click on the "Generate" button re-executes the section of code containing the bug. In contrast, Lights Out requires restarting the entire application to re-execute the erroneous code. In Lights Out, the fault may lie deep within the initialization of the application, requiring a combination of object-centric breakpoints to understand the symptom and standard debugging tools to locate the root cause.

3.2.5 Experimental Variables

Table 3.1 presents the variables we use for our analysis of the data. We consider object-centric breakpoints as the single independent variable of the experiment. We seek to measure the impact of object-centric breakpoints on three dependent variables, the *correctness* of participants answers to the tasks as a proxy to study $H0_1$, the *debugging actions* performed by participants as a proxy to study $H0_2$, and the *debugging time* spent by participants to complete a task as a proxy to study $H0_3$. We qualify as debugging actions the absolute number of debugging-related actions performed by each participants to complete a task. These actions are: the number of added and removed breakpoints, of added, modified and removed methods, of custom code execution (scripts), of call-stack navigation and of step-by-step code execution action.

To control participants' development experience which could influence the dependent variables, we assign randomly participants to the different groups (Figure 3.2). To control interruptions, *i.e.,* time anomalies due to participants being interrupted during a task, we correct the time measures using automatic computation from logs and participants' feedback on interruptions.

Table 3.1: Variables used for the statistical analysis model.

Variables	Description
Independent variables	
Treatment	The introduction of object-centric breakpoints to standard debugging tools.
Dependent variables	
Correctness	Whether the participant correctly or incorrectly explained the root cause of the bug or fixed it.
Debugging actions	The number of actions the participant performed with the debugger during the task.
Debugging time	The total time the participant took to complete the task.
Control variables	
Experience	The participant's development experience, measured in years.
Interruptions	Duration of interruptions participants experienced while debugging.

3.2.6 Data Collection, Selection, and Correction

Participants used the Pharo 9 IDE [Black 2009] for the experiment. Our experimental framework instruments the IDE immediately after the *Welcome instructions* (Figure 3.2, step 1) so that the IDE immediately starts generating usage logs. For the data analysis, we collect the logs to reconstitute every participation. We automatically extract information from the usage logs and compute from them the *time* and *debugging actions* metrics. We delete incomplete participations (*e.g.,* incomplete tasks) without processing the related data. To ensure the accuracy of the measurements, we screen-recorded two pilot participations and manually verified that the logs matched the video recordings. Metrics such as the *Correctness* and the *Interruptions* are extracted from the questionnaires answers. The data are transmitted to, processed and stored on an institutional server. The aforementioned process, in addition to the nature of the data collection through the use of logs and questionnaires, was approved by the ethical committee of our research institution.

Once extracted and computed, metrics data have to be adjusted to cope with inconsistencies (*e.g.,* time anomalies) and for information that require a human decision (*e.g.,* deciding if a task's answer is correct). We devised protocols for manual data adjustment [Bourcier 2025b], which we followed to select data, assess tasks' correctness and correct time anomalies. Each time, one of the authors first

performed the time anomalies analysis and correction and a second author double-checked the decisions.

Data selection We excluded the treatment tasks from our analysis where object-centric breakpoints were not used, as comparing control and treatment is only valid if the treatment condition is met. To do so, we created a script to automatically scan the logs and reject tasks where no object-centric breakpoint events were detected. In the process, we also rejected the tasks with incomplete data (*e.g.,* due to a data collection error) which we also consider as invalid.

Correctness assessment For each task, we manually analyzed participants' answers and compared them against our knowledge of the bug. We considered a task correct if the bug was fixed or when the provided answer explained the bug root cause (*i.e.,* the participant understood the bug). For both Ammolite and Lights Out, we believe it's enough to explain the root cause, because once understood, bug fixing is trivial.

Correction of time anomalies We determined manually the amount of interruption time for each task (from participants' interruptions declared in surveys and from computed time gaps from the logs), then we subtracted this interruption time from the total time of the corresponding task. For example, if we computed a time gap of 20 hours and the time declared by the participants in the questions was "more than 10 minutes", we simply removed these 20 hours from the task time.

3.2.7 Required Number of Participants and Their Recruitment

To estimate how many participants we should recruit, we performed an *a priori* power analysis using the *G-Power* software [Faul 2009]. We chose a large effect size of 0.7 with a target statistical power of 0.8 because of the examples presented in the literature [Ressia 2012b] suggesting that object-centric breakpoints have a strong potential for facilitating debugging. For instance, in one scenario [Ressia 2012b] understanding a bug would require 48 debugging operations with standard tools but only two operations with object-centric breakpoints. Other scenarios highlight the considerable effort required to scope breakpoints to one specific object among many of the same kind, and the implications of failing to do so (system crashes [Hinkle 1993], massive debugger noise (unwanted interruptions) [Costiou 2020b]). Each example suggests a significant impact on the number of debugging actions needed to understand a bug, and consequently, on the time spent debugging. We employed a two-tailed *Mann-Whitney U* test

(suitable for normal and non-normal distributions), with a significance threshold of 0.05 and a balanced number of participants between control/treatment groups. Under the assumption that our dependent variables would follow a normal distribution, the results of the analysis showed that we needed to recruit at least 70 participants, 35 in each group.

To recruit participants, we sent invitation letters by email to our professional contact lists, to the Pharo community users channels (mailing lists and Discord servers) and to public channels (Twitter). In addition, we contacted people directly from the community who have a public Pharo development activity. To convince people to participate, we told them beforehand that the goal was to evaluate the (yet unknown) impact of a Pharo tool, and that they will be manipulating that tool during the experiment. Participants were not compensated. We ran a pilot with 11 participants, developers and researchers from our research group. They reported problems and instabilities. We took all this feedback into account in the actual design and proceeded with the experiment. The results of the first participants were not included in our study.

3.3 Results

The experiment involved 81 participants. After data selection, we obtained 76 valid participants for Ammolite (42 in control, 34 in treatment) and 72 valid participants for Lights Out (38 in control, 34 in treatment). Overall, participants described their role as full-time developers (37), part-time developers (7), students (25), self-employed (3), unemployed (2), and other (7). Participants were evenly distributed across tasks in both conditions (Figure 3.4), based on their self-reported programming experience. Statistical tests on demographic data comparing both conditions yielded no significant results. In this section, we report the results of our investigation by research question.

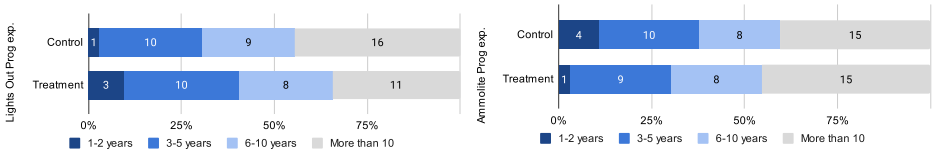


Figure 3.4: Programming experience for Lights Out (left) and Ammolite (right) for each condition.

3.3.1 Statistical Tests

In this section, we conduct a preliminary analysis of the data to ensure that we chose appropriate statistical tests. For every statistical test we performed in the

experiment, we considered a level of significance $\alpha = 0.05$ which is the common threshold used to mitigate type I errors.

The time and the number of debugging actions performed are continuous variables. To test if their distribution follows a normal distribution and to choose appropriate statistical tests for our analysis, we performed *Shapiro-Wilk* tests for both tasks. All tests output *p-values* <= 0.001 except for time metric under the treatment condition (Ammolite: $p = 0.011$, Lights Out: $p = 0.105$). These results suggest that the data has little chance to be normally distributed.

The $\mathbf{H0}_1$ hypothesis concerns the correctness, which is a categorical variable. To search for associations of this variable with the conditions of the experiment (control and treatment), we propose to use contingency tables and χ^2 tests. The $\mathbf{H0}_2$ and $\mathbf{H0}_3$ hypotheses concern the time and debugging actions which are continuous variables. Since normality tests suggest that they do not follow a normal distribution, we propose to analyze the differences between the control and treatment measures of these variables using *Mann-Whitney U* tests.

We report the *Vovk-Sellke maximum p-ratio* (VS-MPR) to help minimize the risks for type I errors. VS-MPR represents the maximum odds of obtaining a given *p-value* under the alternative hypothesis (as opposed to under the null hypothesis) [Sellke 2001]. It gives an idea of how confident we can be when rejecting a null hypothesis based on a *p-value*, especially when the *p-value* is close to the 0.05 threshold.

We report the rank-biserial correlation coefficient R_{rb}[1] as an appropriate measure of the effect size for the Mann-Whitney U test [Jané 2024], *i.e.,* the magnitude of the difference between the control and treatment groups. We interpret R_{rb} as the arithmetic difference between the proportion of data supporting the hypothesis that the values in control are greater than in treatment and the proportion suggesting the opposite [Kerby 2014]. For example, for the time variable, an effect size $R_{rb} = 0.2$ ($r = 0.6 - 0.4$) would indicate that 60% of participants' results suggest that it takes more time to debug using standard tools compared to object-centric breakpoints, while 40% suggest otherwise. We consider that such a 60% proportion of data in the context of our experiment would constitute evidence that the tool is worthy of more research efforts. Therefore, and following the most recent R_{rb} interpretation guidelines [Funder 2019], we consider effect sizes to be small when $|R_{rb}| < 0.2$, medium when $0.2 < |R_{rb}| < 0.3$ and large when $|R_{rb}| > 0.3$.

3.3.2 RQ1 - Ability to Fix the Bug—Correctness

We initially assumed that participants from the control and treatment groups would give the same number of correct answers $\boldsymbol{H0}_1$. However, the results in-

[1] We use the R library *effectsize* [Ben-Shachar 2020] to calculate R_{rb}.

clude incorrect answers in both groups.

In control, Ammolite ($N = 42$) had 40 correct answers, while in treatment ($N = 34$), had 32 correct answers. The χ^2 test shows no significant difference ($p = 1$) in correctness between control and treatment conditions for the Ammolite task .

Lights Out shows 32 correct answers in the control group ($N = 38$) and 24 correct answers in the treatment group ($N = 34$). The χ^2 test shows no significant difference ($p = 0.269$) in correctness between control and treatment conditions for the Lights Out task.

The tests did not reveal a significant association between the object-centric breakpoints and the number of participants that successfully fixed or explained the bugs, *therefore we cannot reject* $\boldsymbol{H0}_1$.

Finding 1. Object-centric breakpoints did not affect developers' ability to fix bugs.

3.3.3 RQ2 - Number of Actions to Debug

The introduction of object-centric breakpoints reduced Ammolite debugging actions by 49% on average, while increasing Lights Out actions by 40% on average. The box plots in Table 3.2 illustrate the distributions of the results that support these tendencies of a positive effect observed for Ammolite and a negative effect for Lights Out. Indeed, the number of debugging actions required to debug Ammolite appears to be lower and varies less when using object-centric breakpoints compared to without. Conversely, getting Lights Out fixed required a slightly higher (and less consistent) number of debugging actions from the participants. The distributions of the results in the control and treatment groups show a striking similarity, particularly in the case of Lights Out. Therefore, it is not possible to draw statistically meaningful conclusions solely from the distribution of the results. In the next step, the Mann-Whitney U tests allow us to conclude.

The difference in debugging actions between the control and the treatment condition is small for both Ammolite ($R_{rb} = 0.204$) and Lights Out ($R_{rb} = -0.112$). Moreover, the impact on the debugging actions for both Ammolite ($p = 0.130$) and Lights Out ($p = 0.417$) is not statistically significant. Since the odds for observing such results under an alternative hypothesis is low (VS-MPR: 1.388 and 1.000), *we decided not to reject* $\mathbf{H0}_2$.

Finding 2. Object-centric breakpoints did not have a significant effect on the number of debugging actions required to fix bugs.

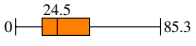
Table 3.2: Descriptive statistics (1) of debugging actions required to debug Ammolite and Lights Out in control (C) and treatment (T), and results of the Mann-Whitney U tests (2) assessing the significance of the differences revealed by the descriptive statistics.

	(1) Group descriptives			(2) Mann-Whitney U test		
Task	N	Mean	Distribution	p-value	VS-MPR	R_{rb}
Ammolite	42 (C)	149.238		0.130	1.388	0.204
	34 (T)	73.294				
Lights Out	38 (C)	157.447		0.417	1.000	-0.112
	34 (T)	220.912				

3.3.4 RQ3 - Time to Debug

While debugging Ammolite, participants were on average 41.5% minutes faster in treatment than in control. In contrast, they were 29.5% slower to complete Lights Out (Table 3.3). The box plots in Table 3.3 present the distributions of the results that support these trends. A positive effect is evident for Ammolite, while we can observe a negative effect for Lights Out. The time required to debug Ammolite appears to be reduced and varies less when object-centric breakpoints are added to the debugger. In contrast, resolving Lights Out's bug required a longer amount of time from participants.

Table 3.3: Descriptive statistics (1) of the time in minutes required to debug Ammolite and Lights Out in control (C) and treatment (T), and results of the Mann-Whitney U test (2) assessing the significance of the differences revealed by the descriptive statistics.

	(1) Group descriptives			(2) Mann-Whitney U test		
Task	N	Mean	Distribution	p-value	VS-MPR	R_{rb}
Ammolite	42 (C)	34.238		0.014	6.262	0.329
	34 (T)	20.014				
Lights Out	38 (C)	29.352		0.031	3.412	-0.296
	34 (T)	38.016				

The results of the Mann-Whitney U test performed on the data confirm these observations (Table 3.3). In Ammolite, a proportion of 66% of the participants debugged in less time using object-centric breakpoints than without ($R_{rb} = 0.329 =$

$0.664 - 0.335$). Conversely, a proportion of 65% of the participants needed more time to debug Lights Out ($R_{rb} = -0.296 = 0.352 - 0.648$) when using the object-centric breakpoints. The effect sizes are large (close to 0.3) and statistically significant for both Ammolite ($p = 0.014$) and Lights Out($p = 0.031$). Since there are 6.262 and 3.412 more chances for observing such results under the alternative hypothesis that our expectations about object-centric breakpoints are correct than under $\mathbf{H0}_3$, *we take the decision to reject* $\mathbf{H0}_3$.

Finding 3. Object-centric breakpoints decreased the time needed to fix Ammolite and increased the time needed to fix Lights Out.

3.3.5 Participants' Perception

According to post-experiment feedback responses (Figure 3.2, step 7), 45% respondents described the experiment as easy, while 30% were neutral and 25% reported the experiment as difficult. A similar proportion of 45% respondents described the experiment as long, the other stayed neutral (30%) or disagreed (25%). 76% of respondents found object-centric breakpoints easy to learn and 94% that it would be useful to improve the process of debugging. 90% of respondents anticipated using the object-centric breakpoints in the future.

Task easiness and length Respondents to the post-task questions (Figure 3.2, step 3 and 6) perceived Ammolite as an easy task (54% in control and 63% in treatment) and Lights Out as a difficult task (37% in control and 47% in treatment). Similarly, in treatment participants perceived Ammolite as short (59% of the answers) and Lights Out as long (61% of the answers). As pointed out by some participants, the randomness in the object presenting the bug in Lights Out might be one of the reasons for its perception: "However, the cell that can not be turned on changes randomly. It is hard to find which will be the faulty cell.". Additionally, the complexity of the task seems to be increased by the fact that the bug is injected through the initialization of the graphical components: "Sadly, the complexity of black-box frameworks [...] doesn't really make debugging much easier even if we know the object.", "It was difficult to find because the method that cause this behavior was injected in the morph package." (morph and framework refer to Pharo's graphical components system). Lastly, it seems that participants encountered unexpected behaviour, possibly due to the randomness aspect of the bug: "I got stuck because of a bug which meant that sometimes my 4 sides were colored and sometimes not, whereas there should always have been only 3 of them colored", "I cannot get the game bug anymore so this is difficult to debug it.".

Debugging experience After debugging Ammolite or Lights Out using object-centric breakpoints, the majority of participants perceived the debugging experience as enjoyable, efficient, intuitive, and the new breakpoints easy to use and learn. Over 63% of the post-task questions repondents (Figure 3.2, step 3 and 6) agreed with these statements for Ammolite and over 57% for Lights Out.

Object-centric breakpoints were highly valued by participants working with Ammolite, with 82% finding them helpful including 72% considering them extremely helpful. For Lights Out, participants were less enthusiastic, with 61% finding the tool helpful including 35% describing it as extremely useful. In their feedback several participants gave reasons for this difference: "I was surprised at the Lights Out task, as object-centric breakpoints does not particularly help there. Once you have the object in question, it is too late to set any breakpoint as the damage has been done." "Because the bug occurs at initialization of the game, it's not easy to use object-centric debugger. The 'wrong' state is already here when one can install a breakpoint [...] when one understands this, the object-centric is not useful anymore and one needs to switch to standard debugging and static analysis."

3.3.6 Analysis of the Difference between Ammolite and Lights Out

We found that object-centric breakpoints have contradictory effects on debugging, benefiting Ammolite and hindering Lights Out (subsection 3.3.3 and subsection 3.3.4).

Task difficulty Participants perceived Lights Out as harder to debug than Ammolite (subsection 3.3.5) because of the random appearance of the symptom on one of the corner lights. However, we designed the tasks so that they would require a similar number of debugging actions and time to complete. Since the pilot run confirmed this (no difference in task difficulty was reported), we tested the task difficulty aspect with the following null hypotheses $H_{eq1}0$: the tasks require the same number of debugging actions to complete and $H_{eq2}0$: the tasks take the same time to complete. Overall, the data distribution for time and debugging actions for both tasks ($N_{Ammolite} = 42$, $N_{LightsOut} = 38$) appears to be part of the same population which is in favor of $H_{teq}0$ and $H_{aeq}0$. We tested $H_{eq1}0$ and $H_{eq2}0$ using Mann-Whitney U tests to compare time and action data under the control condition with Ammolite and Lights Out. We cannot reject $H_{eq1}0$ ($p = 0.919$, $R_{rb} = -0.014$) nor $H_{eq2}0$ ($p = 0.962$, $R_{rb} = 0.007$). The observed data does not support the hypothesis of a different difficulty level between the tasks, and the high p-values even suggest that to debug Ammolite and Lights Out developers need a similar number of debugging actions and a similar amount of time.

Task characteristics Participants reported that the characteristics of Lights Out bugs may contribute to the perceived difficulty in debugging Lights Out (subsection 3.3.5). Notably, reproducing the bug in Ammolite requires to press a button on the graphical interface, whereas restarting the application is necessary to reproduce the Lights Out bug. As highlighted by participants, it implies that to identify the problematic code in Lights Out developers have to go through an additional step. They first need to understand the symptoms of the bug and realize that the bug occurs during the initialization process. Following this, developers must switch to the code browser and analyze the source code specific to that initialization process. Therefore, we analyzed their tool usage behavior to further understand the differences between the tasks and conducted appropriate tests to find the differences, if there are any.

Table 3.4: Results of the Mann-Whitney U tests for significant changes in the usage, in terms of time (T) and activations (A), of the development tools in control and treatment when debugging Ammolite (1) and Lights Out (2).

(1) Ammolite tools usage				**(2) Lights Out tools usage**			
Tool	p	VS-MPR	R_{rb}	Tool	p	VS-MPR	R_{rb}
Task app (T)	0.020	4.781	0.302	Inspector (T)	0.005	13.211	-0.375
Browser (T)	0.006	11.520	0.356	Inspector (A)	0.009	8.763	-0.351
Debugger (T)	0.023	4.282	0.307	Spotter (A)	0.008	9.951	0.630
Wizard (T)	0.006	12.215	0.448				
Implems (A)	0.019	4.798	0.377				
Senders (T)	0.028	3.722	0.449				

Task app: the application window of the task to debug, *i.e.,* Ammolite (1) or Lights Out (2).

Wizard: the tool presenting the experiment tasks and surveys to participants during the experiment, *cf.,* subsection 3.2.2.

Implems and senders: tools displaying the implementations and callsites of a given method.

Activations: the number of times participants activated (opened or entered) a given tool window to use it.

As a proxy to measure the tool usage, we recorded the number of tool activation and amount of time spent in all the tools of the Pharo IDE, such as browser, wizard, and debugger. Then for each tool we performed a Mann-Whitney U test to compare its usage in control and treatment, for both tasks. Since we are conducting multiple tests, the likelihood of finding a statistically significant result purely by chance increases. We applied the Benjamini-Hochberg procedure [Benjamini 1995] to control for false discovery rate using 10% as an acceptable threshold, adjusting the significance levels for the Mann-Whitney U tests to $p \leq \alpha = 0.028$ for Ammolite and $p \leq \alpha = 0.009$ for Lights Out. Table 3.4 presents

only the results for which Mann-Whitney U tests' results satisfied this requirement.

Overall for Ammolite, the introduction of object-centric breakpoints seems to lower the usage of all the IDE tools ($R_{rb} > 0$). In particular, the results suggest a notable decrease in the time spent in the code browser $VS - MPR = 11.520$ and in the wizard tool $VS - MPR = 12.215$. In contrast, with Lights Out, the usage of the inspector is significantly higher with the object-centric breakpoints than without ($R_{rb} < -0.296$ and $VS - MPR > 8$). Conversely, the usage of the spotter, a static tool for code exploration, has significantly decreased ($R_{rb} < 0.630$ and $VS - MPR = 9.951$). Lastly, while we observe a decrease in the usage of the debugger when debugging Ammolite with object-centric breakpoints ($R_{rb} = 0.307$), for Lights Out under the same condition, the Mann-Whitney U test reveals a decrease $R_{rb} = -0.296$. Although, this last result ($p = 0.035$) is above the false discovery rate of 10%, it is consistent with our previous observations that the usage of dynamic tools is more intensive with Lights Out in treatment. These results show that depending on the bug being addressed, Ammolite or Lights Out, object-centric breakpoints alters how developers utilize IDE tools in a different manner.

Finding 4. Even though Ammolite and Lights Out can be fixed with similar amount of time and actions using standard IDE tools, depending on the bug, introducing object-centric breakpoints has changed how developers use the IDE tools.

3.4 Discussion

In this section, we discuss the implication of our findings.

Influence of object-centric breakpoints on debugging actions We did not measure any significant impact caused by the use of object-centric breakpoints on the number of debugging actions our participants performed. This result might indicate that the actual effect size is smaller than what Ressia *et al.* [Ressia 2012b] expected. Indeed, we calibrated our statistical power analysis to determine the necessary sample sizes for detecting large effects (details in subsection 3.2.7). Future research can be devised and conducted to test whether the potential for debugging improvement showcased by Ressia *et al.* holds for smaller effect sizes, or if the expected large effect size can be observed in other debugging scenarios.

Influence of object-centric breakpoints on the time to debug We measured a statistically significant impact caused by the use of object-centric breakpoints on time to debug. Specifically, we measured decreased debug time for Am-

molite, but increased debug time for Lights Out. As mentioned in subsection 3.3.6, the nature of the bugs could be the root cause for this difference. Object-centric breakpoints appear to be beneficial when bugs can be reproduced without restarting the application; conversely, they seem to lead developers to spend more time within the dynamic tools of the IDE (*i.e.,* the debugger, the object inspector) when fixing bugs requiring to restart the application to be reproduced (subsection 3.3.6).

Research has shown that developers who are debugging spend in average 14% [Beller 2018] and 16% [Alaboudi 2023] of their time in the debugger itself. We observe about twice the time spent in the debugger, with 30% (control) and 29% (treatment) in average for Ammolite, and 30% (control) and 38% (treatment) for Lights Out. We attribute this difference from the literature to the nature of Pharo and live programming, where emphasis is placed on dynamic tools. Alaboudi *et al.* also observe that "source code remains the central anchor point in debugging tasks" [Alaboudi 2023]. Our results show that participants navigate the source code for, in average, 45% (control) and 22% (treatment) of their time for Ammolite, and 40% (control) and 21% (treatment) for Lights Out. While for both task in control and consistently with [Alaboudi 2023], navigating source code seems to be the prominent activity, it becomes a less important activity after the introduction of object-centric breakpoints. This could indicate that object-centric breakpoints may succeed in swapping the debugging perspective from the IDE standards to an OOP perspective, with the impact that we observed on the debugging time. This opens new questions on the design and implementation of debugging tools: can we shape our tools to adopt the underlying programming languages perspective and how does that impact the debugging activity?

Do object-centric breakpoints improve the debugging of object-oriented programs? The diverging results in terms of debug time in one case (faster for Ammolite) vs. the other case (slower for Lights Out) open a new research discourse. Compared to Lights Out, Ammolite presents several technical points that may have played a role in the observed effectiveness of the object-centric breakpoints. While past work considered strategies and techniques for debugging [Böhme 2017, LaToza 2020], no study yet investigated whether a tool is best suited for applying a specific strategy. Literature on bug classification focuses on categorizing the type, source, cause and technical characteristics of bugs in a large variety of contexts, such as bug classification in general [Catolino 2019], in Java [Osman 2014, Osman 2016] and Javascript [Gyimesi 2021], for performance [Sánchez 2020], security [Wei 2021], and compilers [Rahman 2023]. These studies aim at helping to identify bugs and understand their consequences, but further research is needed to investigate and provide insights on how to choose appropriate debugging tools and techniques. From the perspective of studying object-centric breakpoints, a first step would be to identify bug types and technical context that can be productively debugged with object-centric breakpoints.

Technical differences observed between Ammolite and Lights out In Ammolite, the defective object is deterministically initialized and remains constant throughout the execution. This presents two advantages: participants always observe the same buggy object when reproducing the bug and breakpoints set on that object remain active until uninstalled. In Lights Out, because participants lose the buggy object when restarting the program for reproducing the bug, they lose all breakpoints set on that object and have to set them again on a different object. This might have advantaged participants using the object-centric breakpoints to debug Ammolite (treatment), as they were not slowed down by this technical limitation.

A single object-centric debugging step is enough to identify the root cause of the Ammolite bug. The bug is a parsing error of the buggy object's properties held into a field of the object (*i.e.,* an instance variable). This root cause can be directly observed by setting an object-centric method breakpoint on the property's getter method, or a field breakpoint on the instance variable holding the property. It seems reasonable to think that in this case, the object-centric breakpoints helped by minimizing the gap between the bug's symptom (a property not displaying correctly) and its root cause (improper parsing of the property). In contrast, in Lights Out, the object-centric breakpoints only serve to understand that something happened during initialization, then participants have to switch back to standard tools to find the root cause. The necessity to recognize when to switch between object-centric breakpoints and standard tools may contribute significantly to slower debugging times in scenarios like Lights Out. While we did not explore this aspect in this study, further studies could be conducted to determine if recognizing the need for this switch impacts developers' effectiveness. If this is the case, additional training and tool support could be developed and assessed to help participants better identify when a switch is necessary.

3.5 Threats to Validity

Internal validity To avoid self-report biases when removing the interruption times declared by participants and deciding on the correctness of each task, we devised a decision protocol and conducted a double-checking process with two authors to ensure accuracy and consistency.

External validity Our study is specific to the Pharo language and environment which threatens the generalization of our results. First, we selected our participants from the Pharo community which could prevent the generalization of our results to developers in general. However, the results show that our participants have various programming experience and backgrounds, including students, researchers, and industry professionals, mitigating this bias. Second, it is known

that Pharo developers frequently use the inspector to learn about the structure of the objects in the program [Kubelka 2018]. This may have influenced the actions performed by participants during the experiment, potentially differing from those that would be performed in other object-oriented languages or environment. However, future work could recreate our bugs in other similar programming languages and environment and extend the results.

We acknowledge that the tasks we used in our experiment may not be representative of common bugs encountered by developers. However, the Ammolite bug was a real bug of Ammolite's application, and we created the Lights Out bug to match with scenarios of UI development where one component does not behave as expected which was illustrated in Ressia's research [Ressia 2012b]. Furthermore, one participant mentioned after the experiment that they frequently work on the Pharo codebase and found both the Ammolite and Lights Out bugs to be similar to those they regularly encounter in the Pharo system.

The study's focus on evaluating a new Pharo tool and the way it was presented to participants may have unintentionally influenced participant selection and responses. Specifically, it may have attracted individuals who are naturally more enthusiastic about new tools (self-selection bias [Heckman 1990]), making them more inclined to favor object-centric breakpoints. Additionally, some participants may have responded in ways they believed the researchers expected rather than providing fully candid reflections on their experiences (moderator acceptance bias [Furnham 1986]). This bias could lead to an overestimation of the benefits and adoption likelihood of object-centric breakpoints.

However, our analysis of participant feedback aligns with the experimental results. Participants expressed appreciation for object-centric breakpoints when they seemed useful based on our findings, such as in the Ammolite bug scenario. Conversely, they showed less interest when these breakpoints seemed less suitable for the task, as observed in the Lights Out bug scenario. This consistency of the responses with our experimental results suggests that, despite potential biases, participants provided meaningful insights into the practical value of the tool.

Another limitation is that we did not control participants' work environments, meaning external factors (such as computer hardware, software versions, network stability, or surrounding noise) could have influenced their experience. To mitigate this, we provided detailed guidance on the expected setup and clear instructions on how to complete the experiment. Additionally, we remained available whenever possible to assist with technical issues, particularly those related to the preset environment (e.g., the Pharo image), rather than the tasks themselves.

Construct validity Our findings are limited to the breakpoints described in section 3.1. We provided five out of the eight breakpoints proposed by Ressia [Ressia 2012b]. Future work could implement and re-evaluate the impact of object-centric breakpoints with the missing ones and verify the consistency with our findings.

All participants went through the treatment task after the control task. This could induce learning and fatigue effect. However, given the negative results we observe for Lights Out in treatment, we believe that the information participants gathered from Ammolite in control did not help them in treatment. Moreover, the bug differences highlighted in the paper are also a mitigating factor to the learning effect between Lights Out in control and Ammolite in treatment. Even though Lights Out was perceived as difficult, the results of the treatment group for Ammolite are better than those of the control group, mitigating the fatigue effect induced by Lights Out. However, since the treatment group was less effective when debugging Lights Out than the control group, the possibility of a fatigue effect induced by Ammolite in control on Lights Out in treatment remains.

Prior to using object-centric breakpoints under the treatment condition, participants went through a tutorial (subsection 3.2.2, step 4) to learn how to use them. This step was essential for conducting the experiment, but it might have introduced a learning effect that may amplify the effect size of our statistical tests.

In the design, we strongly encouraged participants to debug using the object-centric breakpoints under the treatment condition (subsection 3.2.2, steps 4-5), which might be a factor for the results we obtained with Lights Out in treatment. There is a possibility that because of this instruction, participants stayed stuck in the debugger trying to use the object-centric breakpoints before realizing they should switch back to the standard tools at some point (as reported in section subsection 3.3.5). However, this instruction was necessary so that participant use the object-centric breakpoints, allowing us to measure a difference between standard tools with and without object-centric breakpoints.

Conclusion validity Even though we reached the number of participants required by our *a priori* statistical power analysis, it is possible that the tests we performed were of low statistical power, preventing us from detecting the expected effect. We assumed normal distributions and used a standard *Cohen's d* effect size estimate corresponding to a large effect size [Cohen 1988]. Since the results yielded non-parametric distributions (subsection 3.3.1), we used the non-parametric Mann-Whitney U test for which we chose the rank-biserial correlation as an appropriate effect size measure [Jané 2024]. These violated assumptions and the different statistical tests we used make it difficult to assess if we detected the predicted effect size. Since we detected no significant effect for the debugging actions subsection 3.3.3 we must consider being underpowered. Because we ran identical tests for debugging action and time to debug,

it is also possible that our tests for time are underpowered despite detecting a large significant effect size. In this case, we could suffer from effect size inflation [Ioannidis 2008, Gelman 2014, Lu 2019, Rochefort-Maranda 2021]. This could imply that the true effect of object-centric breakpoints is actually smaller than the one we detected. We recommend to perform future studies with increased sample sizes to enable the detection of smaller effects.

3.6 Conclusion

We investigated the effect of object-centric breakpoints on the debugging process, focusing on the time and actions required to fix two bugs in two distinct tasks. Contrary to our initial expectations based on past literature [Ressia 2012b], we could not measure a significant impact of object-centric breakpoints on debugging actions. However, in one of the two tasks (Ammolite) we could measure a statistically significant reduction of debugging time for participants who used object-centric breakpoints vs. those using traditional debugging tools. Conversely, for another task (Lights Out) we found an increased debugging time for participants using object-centric debugging. Based on further analysis of the data, including the feedback of our participants, it seems reasonable to attribute this divergence to the different nature of the tasks and the distinct steps necessary to reproduce the bugs. In particular, when the bug is located in the initialization process of the objects, as in Lights Out, object-centric breakpoints seems to hinder the debugging process. Overall, our findings suggest the need for additional research to gain a deeper understanding of object-centric debugging and breakpoints, particularly to identify the scenarios where they are most effective. Meanwhile, we recommend that developers use object-centric breakpoints when they have access to faulty objects and can reproduce the bug without needing to restart the application.

Chapter 4

A Case Study of Object-Centric Breakpoints and Time-Traveling Queries

Contents

In chapter 2 we discussed the practical limitations of breakpoint based approaches for object-centric debugging. We emphasized the complexity of defining conditions for pausing and investigating programs, especially for programs generating or utilizing non-deterministic values. As a result of the empirical experiment we presented in chapter 3, we found that object-centric breakpoints seem

inadequate to help developers debugging certain types of scenarios. In particular, object-centric breakpoints can hinder the debugging of a program when a defect is executed during the initialization process of the software or application being debugged. Indeed, if the defect is part of the code that initializes the objects of a software, or application, its execution cannot be directly observed using object-centric breakpoints because these breakpoints can only be applied to objects after their creation and initialization. Therefore, in this scenario, when using object-centric breakpoints, developers must realize that the defect is located in the initialization process and switch back to standard breakpoints and call-stack based debugger. As discussed in chapter 3, we argue that this additional cognitive cost brought by the object-centric breakpoints hinders the process of debugging. In chapter 2, we described omniscient debuggers as those able to address issues in programs that produce or use non-deterministic values. Since they are capable of reaching any state of program execution, omniscient debuggers should also allow developer searching for events happening at the initialization of the program, thus addressing that latter limitation.

Problem: In practice, we are unaware of the differences in implementation, usage, and performance between object-centric breakpoints and omniscient approaches. Our knowledge of these differences is solely based on the theoretical aspects of these solutions. Consequently, our conclusions and recommendations may not be reliable in real-world situations and differ from developers' experience.

Research questions: Therefore, in this chapter, we explore the practical differences between object-centric breakpoints and omniscient approaches. As a subject for comparison with the object-centric breakpoints, we use the omniscient debugger *Seeker* [Willembrinck Santander 2023]. Seeker is currently the only omniscient debugger implementation we can find in the Pharo Smalltalk development environment and therefore the best possible candidate for a comparison against the object-centric breakpoints implementation that were subjects of our previous experiment. Seeker implements the *Time-Traveling Queries* mechanism to provide developers with a way to explore a program execution automatically using queries [Willembrinck Santander 2023]. Time-Traveling queries are extensible. To extend the set of existing time-traveling queries, the mechanism provides an API built to abstract away the technical complexity of collecting execution details. Theoretically this makes time-traveling queries an interesting foundation to simplify the process of building object-centric tools.

Therefore, we hypothesize that they facilitate the implementation of object-centric debugging tools and ask the corresponding research question:

> **RQ$_1$**. What are the practical differences between **implementing** object-centric breakpoints and object-centric time-traveling queries?

Omniscient debugging approaches present an overhead during execution because of the information they record about the execution [Lienhard 2008, Willembrinck Santander 2023]. Object-centric breakpoints do not have this limitation, which is why we wonder:

> **RQ$_2$**. How does the performance of object-centric time-traveling queries compare to that of object-centric breakpoints?

Omniscient debuggers and time-traveling queries allow developers to explore and navigate execution backward and forward while object-centric breakpoints only allow forward exploration. Additionally, automatically exploring the entire execution using queries is a one-click process, whereas object-centric breakpoints require at least to proceed with the execution every time a breakpoint pauses it. Therefore, these debugging approaches theoretically produce different user experiences. We hypothesize that breakpoints implemented using time-traveling queries reduce the number of actions needed to debug and help developers collect more information about the program execution compared to object-centric breakpoints. This leads us to our second research question:

> **RQ$_3$**. What are the practical differences between **debugging** with object-centric breakpoints and object-centric time-traveling queries?

Outline: In this chapter, we first present the time-traveling queries mechanism. Using an example, we then compare how to implement object-centric breakpoints and their equivalent *Time-Traveling Object-Centric breakpoints* (TTOCB). Next, we perform 7 benchmarks to measure the differences between object-centric breakpoints and their TTOCB equivalent in terms of performance. Finally, we present a case study experiment re-using the object-centric scenarios from Willembrinck et al.'s experiment [Willembrinck 2021] and comparing how they would be solved with object-centric breakpoints and TTOCB. The results of these studies suggest that despite the performance overhead, TTOCB would allow developers debugging object-centric scenarios requiring less actions from developers and that they would in certain cases allow developers to get more complete information about their programs.

4.1 Time-Traveling Queries, an Overview

The following section introduces the *Time-Traveling Queries* mechanism. The purpose of this overview is to familiarize readers with the research work of Willembrinck et al.[Willembrinck Santander 2023], and serves as an introduction for understanding our thesis's contributions presented in the next sections of this chapter.

4.1.1 Time-Traveling Queries

Time-Traveling Queries is a mechanism that automatically explores program executions to collect execution data called *program-states* (subsection 4.1.2). In order to collect program states, the approach requires a debugging backend that controls execution and collects information about it while it is running. Theoretically, any omniscient debugging back-end is eligible for such a task. However, Willembrinck et al. proposed to use a time-traveling debugger [Willembrinck Santander 2023]. Such time-traveling debugger annotates the program-states with time index, i.e., timestamps, to enable navigation to any point in the running execution of the program where program-states were collected. Time-traveling queries filter the program-states of the program execution to retrieve specific information. They achieve this by using a selection function that determines the relevance of a program state to the query's objective. Later in our dissertation, in Table 4.1, we present the selection functions we used to implement time-traveling object-centric breakpoints.

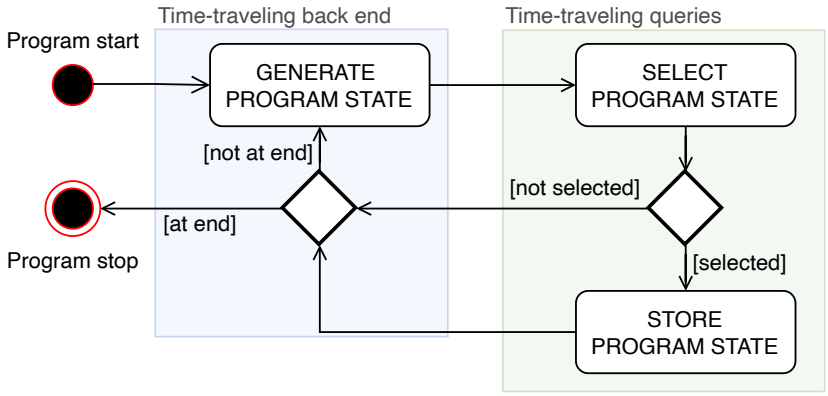


Figure 4.1: A simplified view of the *Time-Traveling Query* system and its time-traveling back end.

As illustrated by Figure 4.1, when a time-traveling query is activated, it triggers the execution of the program by the time-traveling back end. While executing the program, the backend returns program states that are fed back to the query to be either selected then stored as part of the results or simply ignored.

4.1.2 Program States

As introduced by Willembrinck et al. [Willembrinck Santander 2023], we can qualify program states as facades to the traditional debugging call-stack frames. A call stack frame is a representation of a program execution step. It contains references to the instruction being executed and the execution environment of that instruction. In other words, it contains all the variables accessible in the instruction's scope.

In summary, program states provide an abstraction that enables the creation of queries without the need to write code to extract data from lower-level call stack frames.

4.1.3 Making Program States Object-Specific

Due to *reverse-replay* implementation of the underlying time-travel debugger backend, objects are not consistently identifiable across different queries results [Willembrinck Santander 2023]. This means that by themselves program states do not enable the creation of object-centric queries. Willembrinck et al. solved this problem by proposing the *Object Unique Identifier* (OID) to identify objects across executions [Willembrinck Santander 2023]. We will not dive into the implementation details of such unique identifier as they come to answer the constraints of the underlying time-traveling debugger backend, Seeker, and its reverse-replay implementation. However, we note that, to enable developers to write object-centric queries, the authors added to the program states an API method to retrieve the OID of objects.

4.2 Time-Traveling Object-Centric Breakpoints

Before conducting our case study, we completed the list of object-centric time-traveling queries implemented by Willembrinck et al. [Willembrinck Santander 2023] so that there would be a time-traveling equivalent to every object-centric breakpoints originally described by Ressia et al. [Ressia 2012b]. We detail the time-traveling queries equivalents to object-centric breakpoints which we called *Time-traveling Object-centric Breakpoints* (TTOCB) in Table 4.1 below. In particular, we present a high level description of the TTOCB' selection function (cf. Figure 4.1).

Table 4.1: Mapping between object-centric breakpoints [Ressia 2012b] and their TTOCB counterpart. With a white background, we highlight the TTOCB already implemented in previous work [Willembrinck Santander 2023], and with a gray color the TTOCB we implemented.

	Object-Centric Breakpoints	**Time-Traveling Object-Centric Breakpoints**
1	Halt on read	Selects the program states corresponding to an **instance variable read**, where the oid of the **receiver object** matches with the oid of the selected object.
2	Halt on write	Selects the program states corresponding to an **instance variable assignment**, where the oid of the **receiver object** matches with the oid of the selected object.
3	Halt on call	Selects the program states corresponding to a **message send**, where the oid of the **receiver object** matches with the oid of the selected object.
4	Halt on invoke	Selects the program states corresponding to a **message send**, where the oid of the **sender object** matches with the oid of the selected object.
5	Halt on creation	Selects the program states corresponding to an **instanciation message**, where the **name of the class** about to be instantiated matches with the one selected.
6	Halt on object in call or invoke	Selects the program states corresponding to a **message send**, where the oid of the selected object matches with one of the **arguments oids**.
7	Halt on interaction	Selects the program states corresponding to a **message send**, where the oid of the **sender or the receiver** matches with the oid of two selected objects.

Out of all TTOCB presented in Table 4.1, three of them were already part of Willembrinck et al. contribution [Willembrinck Santander 2023], the *2. Halt on write*, *3. Halt on call* and *5. Halt on creation*. We therefore completed the list of object-centric debugging breakpoints adding the *1. Halt on read*, *4. Halt on invoke*, *6. Halt on object in call or invoke* and *7. Halt on interaction* as highlighted with

a gray color in Table 4.1. Later in this chapter, in subsection 4.3.2 we provide a detailed example of our implementation of the *7. Halt on interaction* TTOCB.

4.3 Differences in Implementation Approaches

To implement object-centric breakpoints in Pharo, we use reflexive techniques, while to implement time-traveling object-centric breakpoints, we rely on the time-traveling query mechanism. To illustrate the differences between these two approaches, we present a comparison of the *7. Halt on interaction* implementation in both versions. Unlike the other breakpoints listed in Table 4.1, which only apply to a single object, this breakpoint is particularly complex because it applies to two objects. Specifically, it pauses execution each time an object sends or receives a message to or from another object.

4.3.1 Implementation Based-on Reflection Techniques

To implement our *7. Halt on interaction* breakpoint in Pharo [Black 2009], we leverage the concepts of anonymous subclasses or lightweight classes [Hinkle 1993, Ducasse 1999] and dynamic sub-method behavioral reflection [Denker 2008, Costiou 2020a]. We illustrate the installation process of an anonymous subclass with Figure 4.2 a figure extracted from [Costiou 2022], and explain it below.

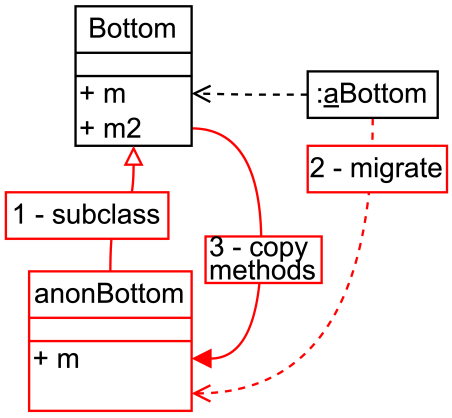


Figure 4.2: Illustration of the anonymous subclassing process for object-centric instrumentation, extracted from [Costiou 2022], Section 3.3, Figure 2.

Figure 4.2 illustrates the process of anonymous subclassing, and how it can constrain code instrumentation to specific objects. In other words, it shows how developers can create object-centric instrumentation. The authors of this approach devised the process in three steps:

- **Step 1:** Create anonymous classes dedicated to the objects to instrument. These classes must be positioned in the hierarchy as subclasses of the original classes of the targeted objects. In the example, Figure 4.2, `anonBottom` is subclassing `Bottom`, the original class of the object `:aBottom`.
- **Step 2:** Migrate the targeted objects from their original classes to their previously created anonymous classes. In Pharo, this step requires a primitive operation such as `class adoptInstance: object`.
- **Step 3:** Copy all methods implemented in the class hierarchy of the targeted objects into their anonymous classes. The objects now have an override for every method they are susceptible to execute, i.e., object-specific method overrides.

This process is simplified and should contain additional steps to tackle object-oriented language execution specificities, such as the lookup of methods following usages of the `super` keyword [Costiou 2022]. After *Step 3*, we apply instrumentation on every code instructions (AST nodes) sending a message. For this, we use the *Reflectivity* [Costiou 2020a] library on each copied method as illustrated by Listing 4.1. We first do so for one of the *7. Halt on interaction* selected objects. With this instrumentation, we intend to capture each message sent by this object, and verify whether the message's recipient matches the other selected object. If there is a match, it means we have identified an interaction between the objects of the breakpoint, and that we must pause the program execution.

```
1  HaltOnInteraction >> #instrument: aMethod
2
3    | metalink|
4    metalink := Metalink new.
5    metalink metaObject: [ :messageReceiver |
6      messageReceiver == target ifTrue: [ Halt now ]
7    ].
8    metalink control: #before.
9    metalink arguments: #( messageReceiver ).
10
11   aMethod ast sendNodes do: [ :node |
12     node link: metalink
13   ]
```

Listing 4.1: Code snippet for instrumenting all message sending nodes of a method to *7. Halt on interaction* between two objects.

As illustrated in the code snippet Listing 4.1, *Reflectivity* uses a metalink, which is an object to describe the instrumentation to apply. To this metalink, we pass a block closure using the method `metaObject:` (line 5). This closure will

be responsible to check if the receiver of the message to be sent matches with our *7. Halt on interaction* breakpoint designated object (line 6). If the verification is correct, then we pause the execution using the `Halt now` (standard breakpoint) statement (line 6). Lines 8-9 only contain parameters for the instrumentation, indicating that the instrumentation is to execute before any message send and that the receiver of each message must be passed as an argument of the closure at run time. Finally, we install the instrumentation on every message send instruction, using the AST (lines 11-13).

At this stage of the process we have applied the instrumentation on the methods of only one of the *7. Halt on interaction* selected objects, we call it `A`. That instrumentation checks for the messages sent by `A` to the other object, `B`. Since both selected objects can send a message to the other one, this procedure must be repeated so that `B` checks for the messages it sends to `A`. The installation of these object-centric instrumentation must be synchronized to avoid missing interactions between the objects. Additionally, because this object-centric breakpoint targets two objects, it requires new integration with the system, to store the reference to one of the objects while developers seek in the call-stack debugger the second object on which to apply the breakpoint.

4.3.2 Implementation Based-on Time-Traveling Queries

Using the example of the *7. Halt on Interaction* we illustrate how straightforward it is to implement an object-centric breakpoint equivalent with the support of an omniscient debugger and the time-traveling queries mechanism. To do that, we write a query selection function, *i.e.*, the selection step of Figure 4.1. We first write the following `value:` method of the selection function. This selection function takes a program state object as a parameter and returns `true` if that program state represents an interaction between our objects.

```
1  SelectAllMessagesBetweenObjects >> #value: pState
2    | expectedOids messageOids |
3
4    pState isMessageSend ifFalse: [ ^ false ].
5
6    expectedOids := { firstOid . secondOid }.
7    messageOids := {
8      pState oidOf: pState receiver .
9      pState oidOf: pState messageReceiver
10   }.
11
12   ^ messageOids first ~~ messageOids second and:
13     [ expectedOids includesAll: messageOids ]
```

Listing 4.2: Implementation of the selection function of the *Halt on interaction* time-traveling object-centric breakpoint.

Listing 4.2 shows our implementation of selection function for the *Halt on interaction* time-traveling object-centric breakpoint. This implementation uses the target object OIDs (line 6) obtained as arguments of the query at its creation and stored as instance variables, `firstOid` and `secondOid`. These two OIDs identify the objects for which the query seeks the interactions. In the last lines of our selection function, we will compare our objects OIDs to those of the sender and the receiver of message represented by the program state provided in argument. We store theses OIDs as a pair in the `messageOids` variable (line 7-10). Note that the keyword `receiver` (line 8) refers to the execution context receiver, which the case of a message send in Pharo is the sender of the message. In our selection function we perform three checks:

1. Line 4: we check that the current program state corresponds to a message send (*i.e.*, an interaction between objects),
2. line 12: to reject messages sent by an object to itself, we check that the message sender (*i.e.*, the first of the pair) is different from the object that receives the message (*i.e.*, the message receiver, second of the pair),
3. line 13: we check that the sender and receiver OIDs correspond to the expected objects OIDs.

Willembrinck et al. already provided a uniform way to add an argument to a query [Willembrinck Santander 2023] by prompting popups to the user. So it becomes possible to use the query we just implemented as is, without further steps of integration to the environment. Our query will be actionable via every menu displaying objects in Seeker debugger, the first object being selected through a right click and the second through a popup as the one illustrated bellow by Figure 4.3.

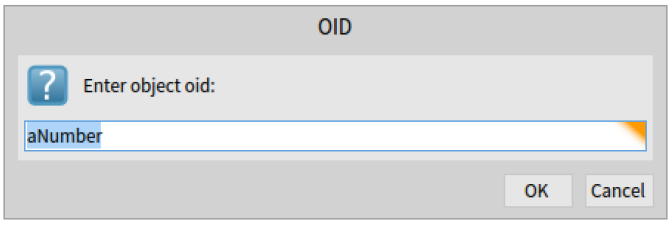


Figure 4.3: Screenshot of the popup for selecting an OID in Seeker [Willembrinck Santander 2023].

4.3.3 Observations

Using the time-traveling queries to implement the TTOCB, we needed to understand the concept of OID and define a selection function to filter program states.

In contrast, when using reflective techniques to implement OCB, we needed to understand and combine the advanced concepts of anonymous subclasses and sub-method partial behavioral reflection [Denker 2008, Costiou 2020a]. Time-traveling queries enabled us to implement object-centric breakpoints with less code and fewer steps than reflective techniques, because there was no need for code instrumentation.

Thanks to the program's API, which abstracts the complexity of behavioral instrumentation, time-traveling queries help implement object-centric tools with seemingly less effort than traditional instrumentation approaches based on reflection.

4.4 Differences in Implementation Performances

To highlight the differences between the performances of object-centric breakpoints and time-traveling object-centric breakpoints, we performed benchmarks using the following unit test in Listing 4.3. We performed seven benchmarks, one for each type of object-centric breakpoint implemented in Pharo, *i.e.* breakpoints on variable read, writes, and method call (see Table 4.2). For each benchmark we took 30 measures running the unit test, first using Pharo OCB implementation and then with the TTOCB implementation.

```
1 TestClass >> #testStudentPrinting
2   | group |
3   group := AMGroup new.
4   self students do: [ :s | | str |
5     str := WriteStream on: String new.
6     group textPrintStudent: s on: str.
7     self assert: (#(+ -) includes: str contents last)
8   ]
```

Listing 4.3: A unit test of the Ammolite application.

All our benchmarks were executed using *ReBench* [Marr 2018], with Pharo 11 image (Pharo-11.0.0+build.714) on an Apple M1 Max, 32GB RAM, running on Darwin Kernel Version 22.5.0. To ensure that we can run the benchmarks in a way that produces comparable data, we use the following settings.

4.4.1 Object-centric breakpoints settings

An object-centric breakpoint is supposed to pause the execution. To create a runnable benchmark, we modified the breakpoints' behavior to avoid trigger an execution interruption and return just before breaking. This way, the breakpoint

mechanism is executed, but the execution interruption is only simulated and the execution continues.

Additionally to measure the execution time of object-centric breakpoints we need to ensure that they get activated during execution. To do so, we manually provide the object to debug, one of the objects representing a student in Listing 4.3 (cf. line 4 and 6), and we install the breakpoint on it at the beginning of the benchmark. The time required to install the breakpoint is included in the measures. However, it is negligible over the whole benchmark execution (less than 10 milliseconds).

4.4.2 Time-traveling object-centric breakpoints settings

Similarly to object-centric breakpoints, we must provide an object to ensure that we collect the correct execution time for time-traveling object-centric breakpoints. To obtain comparable results, the object must be the same.

However, to provide the object to the time-traveling object-centric breakpoints, we have to run the debugger a first time, over the entire execution so that the object OID is known by the debugger. Similarly to the installation of object-centric breakpoints, this step is inevitable and is taken into account in the results. Next, we execute the query corresponding to the evaluated breakpoint.

4.4.3 Benchmarks Results

We detail the results of the benchmarks in the following Table 4.2. We observe that the average time required to execute the test with object-centric breakpoints is between 140 and 240 milliseconds. The average time for the equivalent operation using TTOCB is 97004 milliseconds. This performance evaluation shows that on average TTOCB are 602 times slower than object-centric breakpoints.

4.4.4 Observations

These benchmarks confirm the statement we made in the chapter introduction, there is an overhead to TTOCB execution. On average, TTOCB execute 602 times slower than OCBs to provide the same information to developers.

This implies that TTOCB are applicable to smaller programs than OCBs. Furthermore, beyond a certain execution time, TTOCB may be unusable in practice. For instance, a program that takes one minute to execute with an OCB would require developers to wait approximately ten hours for the TTOCB equivalent of the breakpoint to execute.

Table 4.2: Time in milliseconds required to execute a controlled unit test (Listing 4.3) with the Pharo object-centric breakpoints and the equivalent time-traveling object-centric breakpoints. *(*1 var) means the breakpoint is set on one specific variable, *1 selector* means the breakpoint is set on one specific selector.

Breakpoint type	**Execution time OCB (in ms)**	**Execution time TTOCB (in ms)**	**TTOCB / OCB overhead**
Halt on state access	170 ± 0	97512 ± 961	×574
Halt on read	163 ± 2	102331 ± 692	×629
Halt on write	150 ± 0	93131 ± 851	×621
Halt on call (1 selector)	240 ± 1	91927 ± 554	×382
Halt on state access (1 var)	150 ± 0	99570 ± 1229	×664
Halt on read (1 var)	149 ± 1	98413 ± 689	×660
Halt on write (1 var)	140 ± 0	96146 ± 1232	×687

4.5 Differences in Debugging Procedures

In this section, we highlight the practical differences when debugging between object-centric breakpoints (OCB) and time-traveling object-centric breakpoints (TTOCB). We present the case study we conducted using four tasks. We asked a developer to perform the tasks first with OCB and then with TTOCB, and to report the actions needed to complete the tasks. We detail our research methodology and present the details of the four comparisons of debugging procedures with OCB and TTOCB. We end the section by summarizing the results

4.5.1 Research Methodology

To highlight the differences between the use of OCB and TTOCB, we conducted a case study in which a single developer performs a set of tasks twice, once with

OCB and once with TTOCB. Our objective is to compare the practical differences between the two kind of breakpoints. We selected four tasks from an empirical study [Willembrinck 2021] on program comprehension, in which the developer has to answer four questions (one per task). These questions require the developer to adopt an object-centric perspective.

Participant We asked a student in software engineering to first perform the tasks (and answer the questions) with OCB, then to try answering the same questions again but with the help of TTOCB. At the time of the experiment, the student was currently at the end of a two years long internship on software debugging tool development and used Pharo daily.

Observation protocol Both OCB and TTOCB allowed the participant to find the elements required to answer the program comprehension questions. However, the methods used to reach these elements are different. To compare these debugging methods we counted the number of debugging actions and breakpoints the participant used to obtain the correct answers for these four questions. Additionally, we reported the information observed by the participant at every stage of the procedure.

We qualify as debugging action any execution action (putting a breakpoint, stepping the execution, resuming the execution, inspecting an object, selecting a variable) or observation action (*i.e.*, deducing information that helps to answer the question from the debugger). For example in the table describing the debugging procedures for the *Question 3* (Table 4.5), the third step of the procedure with OCB reports the following action: *Click Proceed. The participant observes that* `pc` *is assigned 29*. We count two actions: first, the *Proceed* action resuming the execution, and second the observation of the assignment of the value `29` to the variable `pc`. When counting the number of actions, we consider that setting an OCB is equivalent to executing its TTOCB counterpart. We apply this procedure for each line of the report tables and add up the numbers.

We describe the debugging procedure starting from the next page.

4.5.2 Detailed Debugging Procedures

In these section, we report in Table 4.3, 4.4, 4.5, 4.6, our observations of the debugging procedures followed by the participant to our experiment when debugging the four comprehension tasks with OCB and TTOCB. We analyze these observations of the debugging procedures in the following subsection 4.5.3.

Table 4.3: Procedure to answer the question *"When (=from which methods) is the first object in the `shapes` collection receiving the #color: message? What are the values of the arguments in each message?"*. The answer provided by [Willembrinck 2021] is: *The `color:` message is called only once on the first object of the `shapes` collection, within the method `RSNormalizer>>#normalize` that is called from the test, with the color green as an argument.*

Question 1

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `RSNormalizerTest>> #testBasic`	Debug the following test with Seeker: `RSNormalizerTest>> #testBasic`
Do 1 *step-over* **[Action 1]**, then 3 *step-through* **[Actions 2-4]** then 1 *step-over* **[Action 5]** to execute the method `#new` on `RSBox` in the block closure	Select the class `RSBox` **[Action 1]**, right-click and execute the query `All Instances Creation of class named as selection` **[Action 2]**
Select `thisContext` **[Action 6]** in the debugger inspector and select the `RSBox` instance **[Action 7]** at the 3rd position in the stack	Seeker shows all created instances in the execution, in a table. Time-travel **[Action 3]** to the first instance creation
In the inspector, select the object class (`RShape`) **[Action 8]**, right-click on the `color:` method and set an object-centric breakpoint on that method **[Breakpoint 1]** .	Do 1 *step-over* **[Action 4]** and 1 *step-into* **[Action 5]** to reach the `RSBox` instance, and execute the query: `All messages sent to self` **[Breakpoint 1]** (`self` in this context represents the `RSBox` instance.)
Click *Proceed* **[Action 9]**. The breakpoint hits. Observe **[Action 10]** that `#color:` is called within the method `RSNormalizer>>#normalize` that is called in the test, with the color green as an argument.	Seeker shows all messages sent to the `RSBox` instance, in a table. Observe **[Action 7]** that `#color:` is called only once. Time-travel **[Action 8]** to the corresponding step, and observe **[Action 9]** that the method is called from the test within the method `RSNormalizer>>#normalize`, with the color green as an argument.

Table 4.4: Procedure to answer the program comprehension question *"What instance variables of `RSBox b1` are modified during this test?"* The answer provided by [Willembrinck 2021] is: *The instance variables of `RSBox b1` that are modified during this test are `encompassingRectangle` (twice), `connectedLines`, `parent`, `entryIndex`, and `isDirty`.*

Question 2

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `RSAttachPointTest>> #testVerticalAttachPoint`	Debug the following test with Seeker: `RSAttachPointTest>> #testVerticalAttachPoint`
Do 4 *step-over* **[Actions 1-4]** to do the assignment `b1 := RSBox new size: 20`	Do 4 *step-over* **[Actions 1-4]** to do the assignment `b1 := RSBox new size: 20` to get the object of interest `b1`
Inspect the object of interest `b1` **[Action 5]**, do not select any instance variable and set an object-centric breakpoint **[Breakpoint 1]** on writings of all instance variables	Select the variable `b1` **[Action 5]**, right-click and execute the query `All the assignments of instance variables, of the object currently pointed by the selected variable` **[Breakpoint 1]**
Click *Proceed* **[Action 6]** and a breakpoint is hit. Observe **[Action 7]** that an instance variables was modified. Repeat these actions (*Proceed* and observe) until the execution ends, which happens 3 more times **[Actions 8-9, 10-11, 12-13]**. The following variables, in this order, were modified: `encompassingRectangle`, `connectedLines`, `encompassingRectangle`, `parent`	Seeker shows all the assignments of all instance variables of the object `b1`, in a table. Observe **[Action 6]** that the instance variables are modified in the following order: `paint`, `isFixed`, `matrix`, `shouldUpdateLines`, `baseRectangle`, `baseRectangle`, `encompassingRectangle`, `path`, `baseRectangle`, `encompassingRectangle`, `path`, `encompassingRectangle`, `connectedLines`, `encompassingRectangle`, `parent`

Table 4.5: Procedure to answer the question *"What are the different values of the `pc` instance variable of the `newContext` object during this test?"* The answer provided by [Willembrinck 2021] is: *The values of the `pc` instance variables of the `newContext` object during this test are 29 and `nil`*

Question 3	
Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `ContextTest>> #testSteppingReturnSelfMethod`	Debug the following test with Seeker: `ContextTest>>#testSteppingReturnSelfMethod`
Do 9 *step-over* **[Actions 1-9]** to do the assignment `newContext := aMethodContext step`	Select the variable `newContext` **[Action 1]** and execute the query `All the assignments of variable with selected name` **[Breakpoint 1]**
Inspect the object of interest `newContext` **[Action 10]**, select the instance variable `pc` **[Action 11]**, and set an object-centric breakpoint on writings of this instance variable **[Breakpoint 1]** (via right-click on `pc`)	Seeker shows all assignments of the variable `newContext`. Time-travel **[Action 2]** to the first assignment and do 1 *step-over* **[Action 3]** to get the object of interest `newContext`
Click *Proceed* **[Action 12]**. Observe **[Action 13]** that `pc` is assigned 29	Select the variable `newContext` **[Action 4]**, right-click and execute the query `All assignments of instance variables of the object currently pointed by the selected variable` **[Breakpoint 2]**
Click *Proceed* **[Action 14]**. Observe **[Action 15]** that `pc` is assigned `nil`	Seeker shows all assignments of all instance variables of the object `newContext`, in a table. To filter the results to get only the assignments of the instance variable `pc`, time-travel **[Action 5]** to an assignment of the instance variable `pc`, select the variable `pc` **[Action 6]**, right-click and execute the query `All the assignments of a selected instance variable, of self` **[Breakpoint 3]**. Observe **[Action 7]** that the instance variable `pc` is assigned 4 times, the values: 25, 28, 29, `nil`.
Click *Proceed* **[Action 16]**, the test ends its execution.	

Table 4.6: Procedure to answer the program comprehension question *"How many times is `generator>>#atEnd` called on the `generator` object and from which methods?"* The answer provided by [Willembrinck 2021] is: *The method `Generator>>#atEnd` is called 7 times: 4 times from the method `GeneratorTest>>#testAtEnd` and 3 times from the method `Generator>>#next`.*

Question 4

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `GeneratorTest>>#testAtEnd`	Debug the following test with Seeker: `GeneratorTest>>#testAtEnd`
Do 2 *step-over* **[Actions 1-2]** to do the assignment `generator := self numbersBetween: 1 and: 3`	Do 2 *step-over* **[Action 1-2]** to do the assignment `generator := self numbersBetween: 1 and: 3` to get the object of interest `generator`
Inspect the object of interest `generator` **[Action 3]**, select the object class `Generator` **[Action 4]** and right-click on the method `#atEnd` and set an object-centric breakpoint on that method **[Breakpoint 1]**	Select the variable `generator` **[Action 3]**, right-click and execute the query `All messages sent to the selected object` **[Breakpoint 1]**
Click *Proceed* 7 times **[Action 5-11]** until the test ends its execution. Observe that the 7 breakpoints are hit (one at each breakpoint) **[Action 12-18]** of the method `#atEnd` being called: 4 times from the method `GeneratorTest>>#testAtEnd` and 3 times from the method `Generator>>#next`	Seeker shows all the messages sent to the object `generator`, in a table. Observe **[Action 4]** that the object `generator` receives the `atEnd` message 7 times: 4 times from the method `GeneratorTest>>#testAtEnd`, 3 times from the method `Generator>>#next`

4.5.3 Observations

Below, we summarize the observations we made on the debugging procedures presented in subsection 4.5.2. The following Table 4.7 presents the differences in the number of actions performed with the debugger and the number of breakpoints used, between the debugging sessions of our participant with OCB and TTOCB.

Table 4.7: Count of installed breakpoints and performed debugging actions when answering 4 program comprehension questions with OCB and TTOCB (see subsection 4.5.2).

	OCB	TTOCB
Question 1		
Actions	10	9
Breakpoints	1	1
Question 2		
Actions	13	6
Breakpoints	1	1
Question 3		
Actions	16	7
Breakpoints	1	3
Question 4		
Actions	18	4
Breakpoints	1	1
Total		
Actions	57	26
Breakpoints	4	6

TTOCB Compared to OCB Regarding Debugging Actions

In order to complete all the comprehension tasks, our participant had to perform approximately twice as many actions with the OCB as with the TTOCB. The difference in the number of breakpoints used for all the tasks is only two. Specifically, the participant used two more TTOCB than OCB to complete Question 3 Table 4.5. On average, the developer performed 7 actions before using OCBs and only 3 before using TTOCB. Therefore, it seems that the main difference lies in how the objects of interest for program comprehension are accessed before applying breakpoints.

The procedure in Question 3 Table 4.5 illustrates this well. The participant performed his first TTOCB after only one action, compared to 11 actions in the OCB

case. This initial action enabled him to use the breakpoint `All the assignments of variables with selected names` to obtain the desired object. The participant then made two additional requests to answer the comprehension task.

It seems that TTOCB reduce the effort required to obtain a reference to the objects to be debugged compared to OCBs, which could facilitate their adoption.

TTOCB Compared to OCB Regarding the Access to Program Information

Within this chapter, we have formulated the hypothesis that time-traveling object-centric breakpoints would enable more comprehensive information to be obtained regarding program execution, particularly with execution details occurring during software initialization, and therefore during the initialization of objects themselves.

The details of the debugging procedures followed by our participant to answer *Questions 2: "What instance variables of RSBox b1 are modified during this test?"* Table 4.4 and *Question 3: "What are the different values of the pc instance variable of the newContext object during this test?"* Table 4.5 seem to support this hypothesis. For both questions, our participant collected more information about the evolution of the designated objects' instance variables when using TTOCB. As we can observe from the tables, when using the object-centric debugging breakpoints our participant first needed to use multiple *step-throughs* commands to navigate forward within the execution until being able to apply breakpoints to designated objects. For Questions 2, our participant accessed the object of interest after the execution of `b1 := RSBox new size: 20` using the value of the variable `b1`. Similarly, for Question 3, after the execution of `newContext := aMethodContext step` using the value of the variable `newContext`. Only from that point in the execution the object-centric breakpoints could highlight the objects' modifications of instance variables.

That is how our participant missed some instance variable writes. They were performed before accessing the variables within the execution details on the right-hand side of the assignments. Using TTOCB allowed our participant to collect such modifications in the scope of the entire program execution history as soon as the object could be found.

Time-traveling, object-centric breakpoints seem better suited to helping developers access more information about program execution than traditional object-centric breakpoints. In practice, this could provide developers with information that could improve their understanding of program execution.

4.5.4 Discussion

Our observations suggest that time-traveling object-centric breakpoints are easier to implement than object-centric breakpoints based on reflective implementa-

tions. We also made some observations in terms of a practical approach to debugging. Specifically, TTOCB appear to reduce the number of actions required for debugging compared to traditional object-centric breakpoints. Additionally, the manual search for objects made our participant miss information happening prior to finding objects of interest with OCB, whereas TTOCB provided all the details of execution about these objects. These observations support our hypothesis that TTOCB can overcome the limitations of OCBs when debugging programs whose defect lies in the initialization of objects. As discussed in the introduction, there are also theoretical differences between TTOCB and OCB. TTOCB can be applied to non-deterministic programs, allowing developers to move to any point of interest during execution. On the other hand, non-deterministic programs can make OCB innefective if restarting execution is required to debug.

However, our benchmarks reported that TTOCB take 602 times longer to execute than OCBs. Furthermore, if TTOCB can easily provide the complete list of object instantiations, message sends, or assignments to any instance variable of an object in one execution, the number of results can be significant. Filtering these results is then an additional burden for developers. This mitigates our observation that TTOCB significantly reduce the number of required action to debug. TTOCB provides two ways to support results filtering. We can filter the list of results based on a string representation (*e.g.*, by filtering method names). However, to do this, it is necessary to know what we are looking for, which is not always the case when debugging. We can also time-travel to one of the results and execute another TTOCB based on contextual information. However, in this situation, we also need to know when to time-travel and on what new elements base our TTOCB exploration. For this reasons, we must consider the size of the programs to be debugged when making our observations. If the program takes too long to execute and contains too much information, the benefits of TTOCBs will be reduced.

Therefore, we recommend prioritizing the implementation and use of TTOCBs when developers have specific needs and want adapted object-centric debugging tools rapidly. We also recommend TTOCB in cases where the program is non-deterministic or where a bug is suspected to be present during object initialization. These recommendations only apply if a time-traveling debugger with an API similar to that of program states is available and the program executes quickly.

4.5.5 Threats to validity

TTOCB is based on Time-Traveling Queries, for which an empirical experiment [Willembrinck 2021] has shown that developers using queries did 38% less debugging actions than developers using the standard Pharo debugger. While our results seem consistent with the latter experiment, they are based on only four examples, which limits our ability to generalize our conclusions. These results

should be investigated more thoroughly to confirm them. However, they suggest that further research based on empirical experience would be worthwhile.

In our analysis, we counted only the debugging actions that we could observe from our participant's report. Therefore we do not know how many implicit actions the developer may or may not have performed to decide when to step into a message send or when to step over it, etc. While this could have an impact on the results we reported, as we followed the same counting protocol for both the OCB and TTOCB possible biases should be mitigated and not in favor of one particular approach.

In this experiment our proxy to interpret the effort required for debugging is the number of actions performed by developers with the debugger. However, the number of actions does not reflects how difficult it is to perform an action. Some actions might be related to a higher cognitive load than others. Additionally, our participant performed each comprehension task in the same order: first with OCB and then with TTOCB. Although the tools are different, the participant may have learned details about the task while debugging with OCB that may have helped with TTOCB. This could have potentially biased our observations.

Therefore, future work should evaluate OCB and TTOCB in a controlled manner. *Think-Aloud* protocol [Ericsson 2017] should be used to capture and associate developers' intentions with their actions. They should also be asked about their perception of the debugging difficulty. To mitigate learning effects, future evaluations should consider randomly switching the order of the tasks (i.e., OCB first, then TTOCB, and vice versa). This would provide a more accurate comparison between OCB and TTOCB.

4.6 Conclusion

In this chapter we explore the practical differences between object-centric breakpoints (OCB) and time-traveling object-centric breakpoints (TTOCB), their equivalent based on time-traveling queries. We illustrate how TTOCB abstract away low-level concerns, saving developers from having to understand and use complex language reflection techniques for implementing and extending object-centric breakpoints. Our user study across four example scenarios indicates that TTOCB could reduce the number of actions needed for debugging and offer more detailed information about program execution than traditional object-centric breakpoints. This suggests a promising improvement of TTOCB in terms of debugging effort and support to program comprehension. In contrast to OCB, TTOCB appears helpful to developers in debugging scenarios in which bugs are located in the initialization process of objects. However, as the 7 benchmarks we performed highlighted, the advantages of TTOCB over OCB come with a performance tradeoff. TTOCB is 602 times slower than OCB, implying that it is less suited to help developers debug large programs. Overall, our findings suggest the need for more re-

search to confirm our observations, with thorough empirical designs that include a wider range of real-world scenarios. Additionally, research addressing the limitations of OCB and TTOCB in terms of implementation, usability, and performance remains a worth the efforts. In the meantime, we recommend that developers prefer time-traveling object-centric breakpoints over object-centric breakpoints for small program executions.

Chapter 5

Scopeo, Towards Object-Centric Debuggers Supporting Debugging Cycles

Contents

In the second chapter, chapter 3, we demonstrated through empirical experimentation that, while object-centric breakpoints have the potential to help developers debug their object-oriented programs, the limitations of these breakpoints can also hinder debugging in some cases. In chapter 4, we then showed how using omniscient approaches with queries to explore program execution history compare to object-centric breakpoints, and how they address limitations in certain scenarios.

In this chapter, we continue to seek for improvements of the approaches for debugging object-oriented programs. We detail our motivation, derived from analyzing various existing debugging methodologies and relating them to the object-centric debugging approaches listed in the state of the art section of chapter 2. Specifically, we point out that object-centric debugging approaches do not provide technological support for the iterative process of hypothesis and exploration in which developers engage during debugging.

To facilitate the debugging process, we argue that it is necessary to provide developers with technological support for these iterations. In this chapter, we present Scopeo, an omniscient debugger that supports the execution of queries to automate the exploration of program execution histories. The main contribution of Scopeo is the *exploration scope*, a representation of subparts of the execution history, obtained as result of the queries and reusable as input for new exploration phases, i.e., new queries.

We evaluated our approach using a descriptive case study research design. We used Scopeo to debug three practical scenarios while observing and describing the limitations of the approach. For two cases of our study, we reused the scenarios from the empirical experiment presented in chapter 3. This enables us to gain further insight into the potential of omniscient debuggers to address bugs that object-centric breakpoints seem limited to help with.

In the final section, we discuss the limitations of Scopeo and our analysis, and we outline potential improvements and areas for future research.

5.1 Background and Motivations

To research what the next generation of object-centric debuggers could look like, we first examine the different stages that comprise existing debugging models. Then, in this section, we present how the object-centric debugging solutions listed in chapter 2 support the different stages. This allows us to identify and discuss the gaps in support that motivate our approach.

5.1.1 Common Models of the Debugging Process

Studies on debugging models teach us that debugging is product of both program comprehension and a systematic method of acquiring knowledge about how programs behave [Gilmore 1991, Zeller 2009]. This simplified scientific method is often referred to as the debugging process [Vessey 1985, Gilmore 1991, Yoon 1998, Spinellis 2018]. Gilmore has schematized this process, illustrated by Figure 5.1.

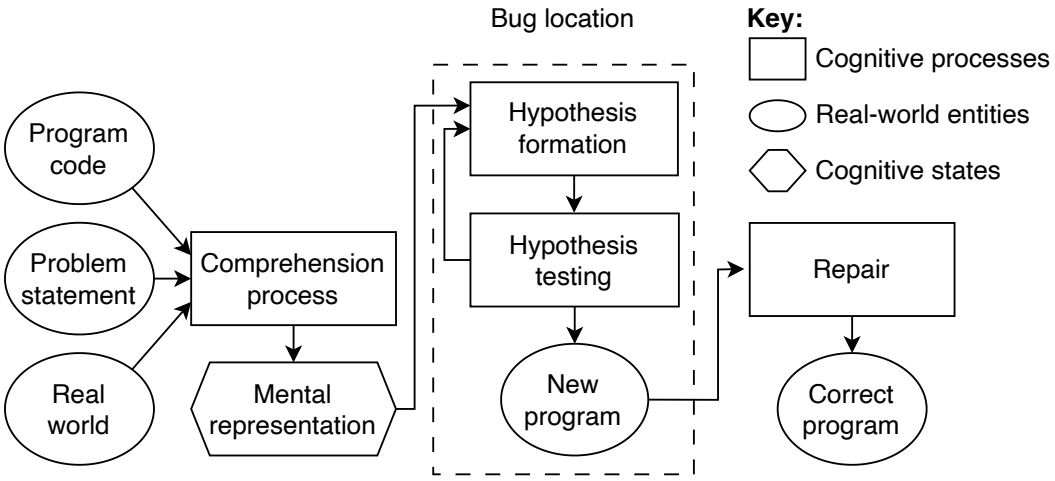


Figure 5.1: Schematic view of many models of debugging, extracted from [Gilmore 1991], *Introduction*, Figure 1. In this diagram, *Real-World* stands for real-world knowledge.

Although there are more recent and detailed versions of Figure 5.1's model [Spinellis 2018], Gilmore's version is particularly interesting because it highlights the comprehension process as well as the mental representation of the program in the debugging model. In Figure 5.1 we can observe that the comprehension process and the mental representation stages occur before the iterative process of formulating and testing hypotheses about the problem to find the fault location. This fault localization process is illustrated in Figure 5.1 by the *Bug Location* block, which shows a loop consisting of *Hypothesis Formation* followed by *Hypothesis Testing*. The loop ends when a hypothesis about the bug location is validated. However, because of this separation between the comprehension process and fault-localization process, this model cannot represent how a *conceptual bug* co-existing with a *teleological bug* can be debugged [Gilmore 1991]. Gilmore provides the following definition for these two types of bugs *"In the latter case the program contains a traditional error, but in the former the program may be syntactically and semantically correct, the error arising because the program solves the wrong problem."*. In this explanation, we assume the *traditional error* refers to the raise of a software exception or a program crash.

5.1.2 Program Comprehension In the Debugging Process

Following the presentation of the general debugging model's limitations (as we previously outline in subsection 5.1.1), Gilmore conducted an analysis of debugging data from a controlled experiment involving 80 participants who debugged ten versions of two programs, each of which contained two bugs. Gilmore then proposed the debugging model depicted in Figure 5.2.

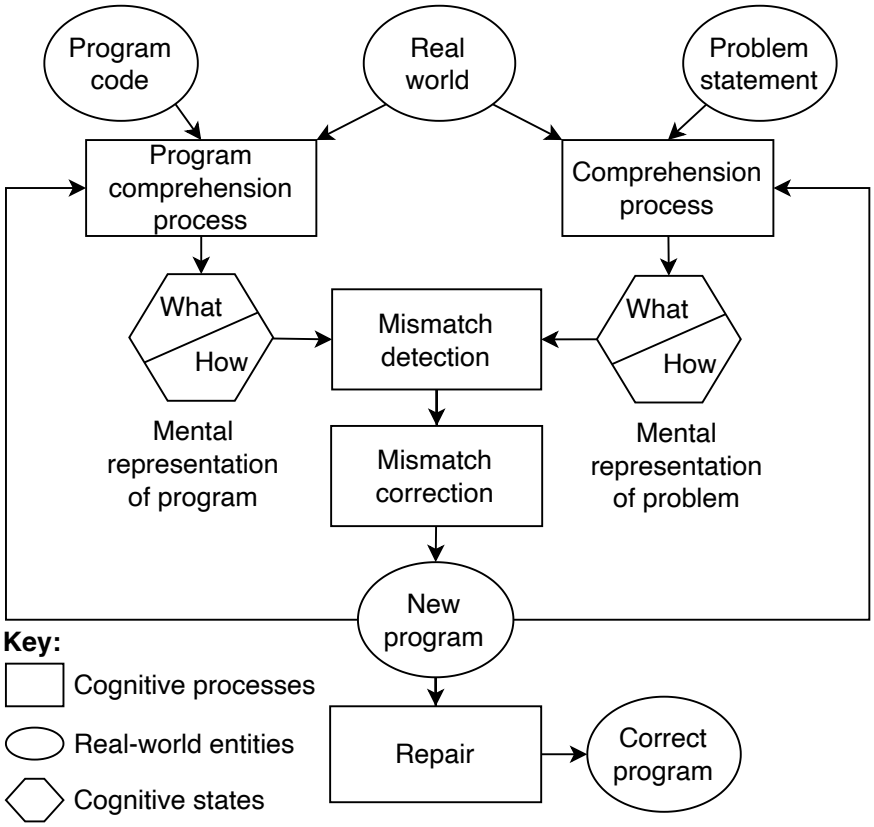


Figure 5.2: Schematic view of debugging viewed as design, extracted from [Gilmore 1991], *A model of debugging*, Figure 4. In this diagram, *Real-World* stands for real-world knowledge.

The Gilmore debugging model, illustrated in Figure 5.2, includes program and problem comprehension in the debugging iteration loop, as the initial stage. This stage is then following by the updating of mental representations of the program and the observed problem [Gilmore 1991]. After completing this phase, developers compare their mental representations with the reality of program execution to identify potential mismatches (represented in Figure 5.2 by the *Mismatch detection* and *Mismatch correction* processes). This iterative process yields a revised program, the *New program* at each iteration. In contrast to the preceding model

(Figure 5.1), in which the iterations terminate once developers identify the defect, in this new model, developers exit the debugging iteration cycle when they cannot detect further differences between their mental representation of the program (and that of the problem) and the behavior they observe at execution. Following this newly acquired comprehension of the program, developers can initiate the *Repair* process to produce a *Correct program*. Gilmore categorizes this model among program design activity [Gilmore 1991], as it does not strictly differentiate between program understanding and fault localization.

5.1.3 Object-Centric Debugging Support to the Debugging Process

In this chapter, we explore ways to improve the debugging of object-oriented programs. We find the aspects of debugging that are less supported by existing object-centric approaches to be more interesting to work on. To identify these least-supported aspects, we compare the object-centric debugging approaches presented in chapter 2 with the debugging models presented in Figure 5.1 and Figure 5.2.

It appears that many object-centric debugging approaches primarily focus on supporting developers in validating their hypothesis regarding program execution. For example, tools such as *Whyline* [Ko 2004, Ko 2008, Ko 2008, Ko 2009], *Unstuck* [Hofer 2006], *Seeker* [Willembrinck 2021, Willembrinck Santander 2023], or *TraceDebugger* [Thiede 2023a, Thiede 2023b] facilitate the answering of questions related to events that occur during program execution. Therefore, the stages *Hypothesis formation* and *Hypothesis testing* depicted in Figure 5.1, are already supported.

Tools that automate code repair have been proposed to help developers debug code, thereby supporting the *Repair* stage of the debugging models we presented [Parnin 2011, Böhme 2017, Winter 2023]. However, as aforementioned in chapter 2, these tools appear to pose significant challenges for developers because developers require an understanding of the proposed repairs in order to integrate them into the program code [Winter 2023]. Although none of the object-centric debugging approaches we reviewed in our state-of-the-art chapter 2, have been found to provide automated code repair features, it seems wiser to focus on program comprehension itself rather than automated repair mechanisms given that this type of approach also requires support for comprehension.

As Gilmore illustrated in his debugging model Figure 5.2 and as we have also shown in our current state of the art chapter 2, it appears necessary to diminish the cognitive load associated with understanding programs to facilitate debugging. To reduce the cognitive load of object-oriented programs, we can work on two aspect of Gilmore's model. The first, the mental representation of programs, is addressed by all object-centric approaches. Indeed, these approaches

offer perspectives aimed at bridging the conceptual gap between object-oriented programs and debugging tools, as described in chapter 2. The second aspect is the debugging cycle itself, the iterative process of program comprehension. To the best of our knowledge, these debugging cycle is the least-covered aspect of debugging by object-centric debugging approaches. Compass [Lienhard 2009] and TOD [Pothier 2009] provide bookmarks that allow developers to save key points in their exploration. While this prevents them from repeating actions, they still need to remember what these key points correspond to and how their different debugging iterations are related. Whyline [Ko 2008, Ko 2009] allows to execute follow-up queries based on the answers to previous queries and thus establish links between debugging iterations. However, the queries supported by Whyline, *Why?* and *Why Not?*, are not extensible and focus on fault localisation, which can limit program comprehension.

For the remainder of this chapter, we will focus our research on supporting the debugging cycle to facilitate understanding of programs.

5.2 Scopeo, the Approach in a Nutshell

In the previous section, we identified the debugging cycle, i.e. the iterative process of program comprehension, as the aspect of debugging that is the least supported by object-centric approaches. We argue that integrating support for this iterative process would make debugging programs easier. Hence, in this section we present our proposal, Scopeo, an omniscient debugger supporting the execution of queries for program exploration and designed to provide support for the iterative process of program comprehension. Figure 5.3 illustrates how we combine elements from Gilmore's debugging models Figure 5.1 and Figure 5.2 to propose a debugging method that supports the debugging cycle.

Below, we present Scopeo's features, following the order of stages in the proposed method (Figure 5.3).

(1) Reproduced execution - Omniscient back end. In our approach, we propose incorporating the debugging activity into the framework of an omniscient debugger. This is necessary to potentially benefit from the following aspects. It allows addressing bugs in programs that manipulate or produce non-deterministic values, as discussed in the state-of-the-art chapter chapter 2. As we observed in chapter 4, the omniscient debugger should enable comprehensive exploration of execution, including the initialization process. Additionally, an omniscient debugger should allow developers to perform all necessary debugging iterations without replaying bugs several times [Willembrinck Santander 2023]. Thus, this first stage in Scopeo's debugging method is to reproduce the bug with an omniscient debugger activated, so that the execution is recorded.

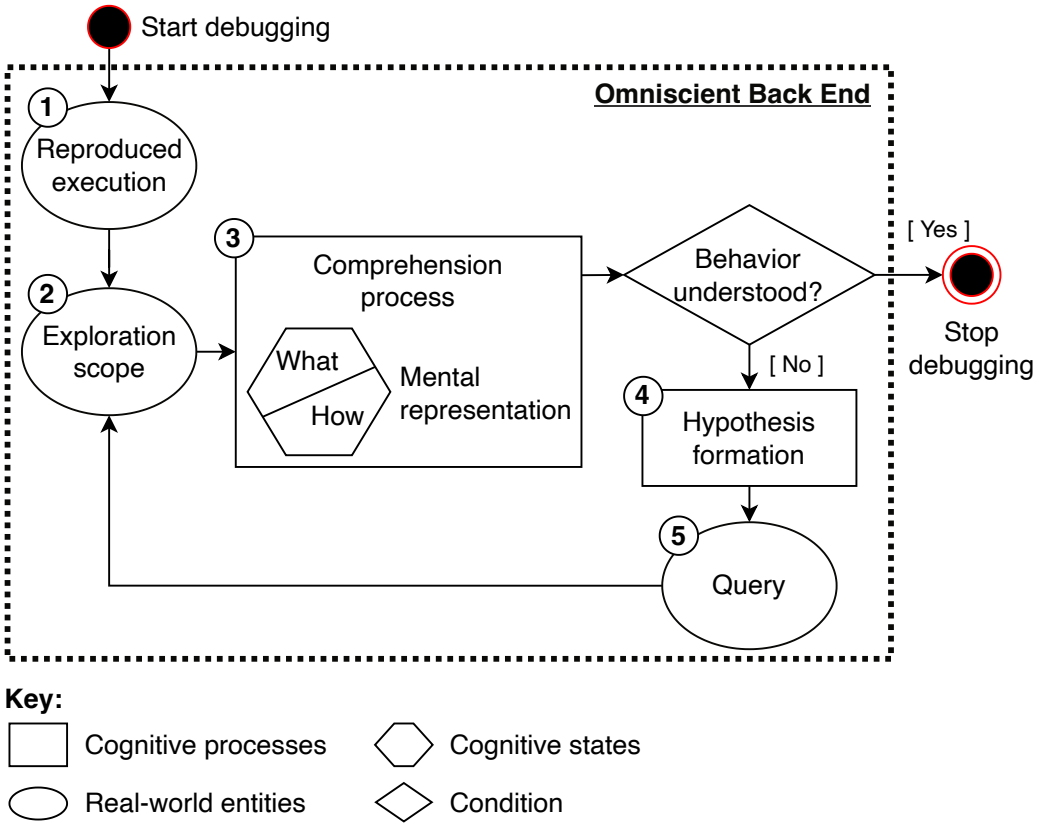


Figure 5.3: Schematic view of our proposal, the Scopeo debugging model, which was inspired by Gilmore's model, *A model of debugging* (Figure 4), [Gilmore 1991].

(2) Exploration scope We propose to provide developers with the exploration scope as a unit of reasoning. The exploration scope is a representation of program execution structured as a tree. This tree contains all events related to the execution of object-oriented programs, including method executions, variable assignments, and message sends between objects. This makes the exploration scope an enhanced version of the traditional call stack that is closer to the graph model of the object-oriented paradigm. We chose a tree structure for the exploration scope because it theoretically allows for the implementation of queries that can traverse the tree recursively and filter its content while maintaining the order and hierarchy of the filtered information. We argue that such queries are easier to implement using a tree structure than a graph structure because a tree structure does not allow for cyclical information, which could lead to an infinite loop when executing queries. Furthermore, since the tree structure contains all the execution related information, it can still be transformed into a graph if necessary in the future. We will discuss these queries in further detail in our explanation of stage five below. At this second stage of the method, the omniscient debugger creates an initial exploration scope for the developers to work with in the third stage. This scope encompasses the entire execution resulting from the reproduction of the bug in the previous step.

(3) Comprehension process - Object-centric mental representation The comprehension process is a cognitive activity during which developers observe and attempt to understand the program, while also considering their understanding of the problem [Gilmore 1991]. According to Gilmore et al.'s model, this process yields an updated version of the mental representations that developers have of the program and the problem [Gilmore 1991]. In our proposition, we encompass the update of the mental model as part of the comprehension process. We make this choice because we argue that bringing to developers object-centric perspectives will closely align with developers' mental models of object-oriented programs [Ressia 2012b], ultimately supporting the cognitive process. The object-centric perspectives we propose with Scopeo are the *object flow*, *object trace* or *object history*, i.e. the list of methods, messages, or variable assignments used by or using a particular object, as offered in *Compass* [Lienhard 2006], *Unstuck* [Hofer 2006] or *Trace Debugger* [Thiede 2023a]. Following the comprehension process, developers can decide to end their debugging session if they understand the problem, or proceed to stage four.

(4) Hypothesis formation Most debugging approaches involve generating hypotheses, testing them, and repeating the process as long as cause of the bug is not identified as illustrated by Figure 5.1 and discussed in chapter 2. We propose to integrate the same stages of *Hypothesis formation* and *Hypothesis verifi-*

cation to Scopeo's debugging model. The only difference as compared to traditional approaches is that developers iterate over the comprehension process as suggested by Gilmore et al.'s instead of focusing the iterations on the bug location [Gilmore 1991]. Therefore, at this stage of the process, developers formulate hypotheses about the program's behavior, the possible cause of the bug, and their own understanding of the program.

(5) Hypotheses testing with queries. To help developers verifying their hypotheses, we propose building upon existing work such as *Whyline* [Ko 2004, Ko 2008, Ko 2008, Ko 2009], *Unstuck* [Hofer 2006], *Seeker* [Willembrinck 2021, Willembrinck Santander 2023], or *TraceDebugger*. These works use queries to search a program execution for elements that developers can then compare to their hypotheses. For example, if a developer wants to test the hypothesis that a method is called three times, they can run a query to find all the events representing calls to that method during the execution. The number of results will allow them to verify the hypothesis. Therefore, the ability of our approach to help developers verify their hypotheses depends primarily on the queries they can execute. These queries are presented later in this chapter in section 5.4.1. Queries are executed within an exploration scope and generates a new exploration scope containing only the elements filtered by the query. This new exploration scope is a subtree of the scope selected for the query. Upon its creation, every new exploration scope is stored and displayed to the developer, who can then begin a new program comprehension cycle. Developers can return to an exploration scope at any time, thus avoiding the loss or forgetting of steps in their debugging session.

5.3 Presentation of The Scopeo Debugger

In this section, we will introduce the Scopeo debugger. It implements our approach to supporting program comprehension iterations during the debugging process. The Scopeo debugger is an omniscient debugger, and more precisely, it is a back-in-time debugger. To debug a program with Scopeo, one must first reproduce the bug while Scopeo is activated. This allows the debugger to record every instruction executed during program runtime (cf. section 5.2). These instructions are reified as traces in Scopeo's record and are structured in a tree representing the relationships between the methods, messages, and assignments that occurred during execution. For the remainder of this chapter, we will use the term *traces* to refer to these instructions that occurred during program execution.

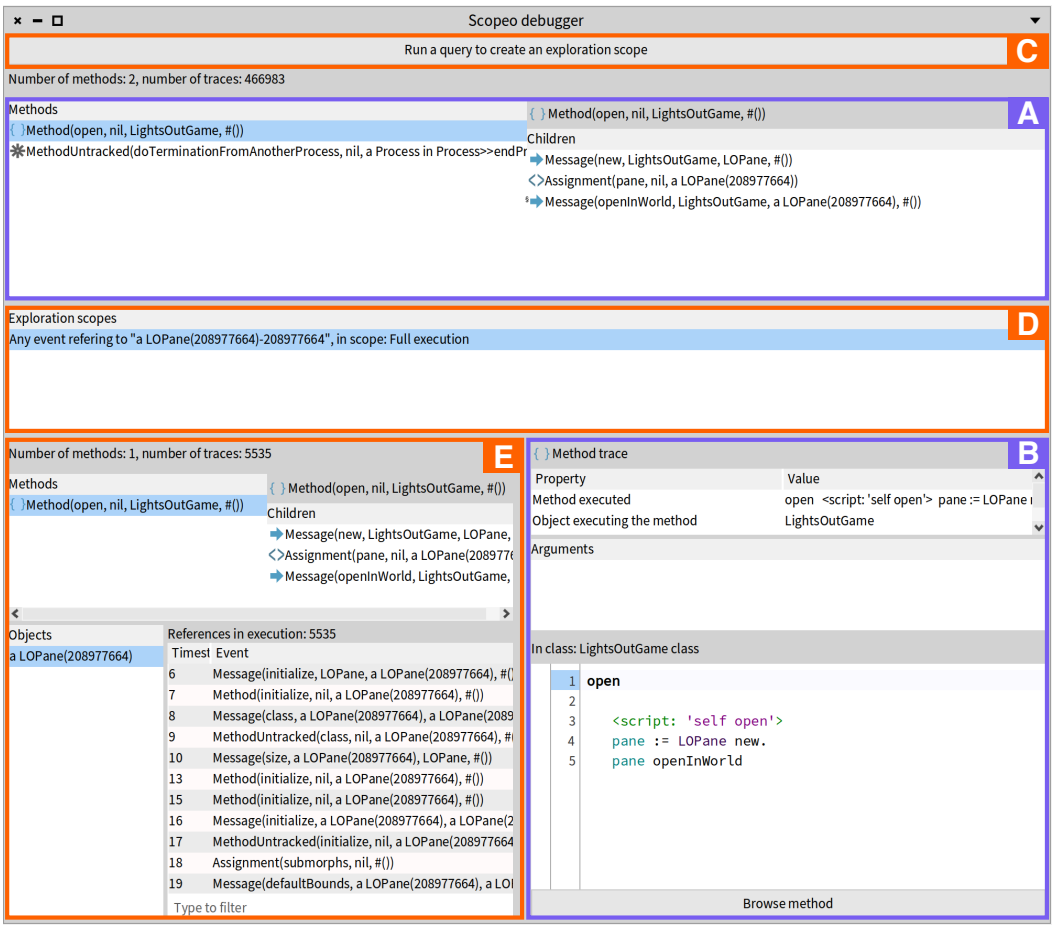


Figure 5.4: Screenshot of the Scopeo debugger with color-coded rectangles highlighting its capabilities. The light blue rectangles ■ show the features that Scopeo has in common with traditional, call-stack based debuggers. The orange rectangles ■ highlight the features that are unique to the Scopeo debugger.

The Scopeo interface presents the program execution tree using the *miller list* component labeled A in Figure 5.4. This view, similarly to call-stack-based debuggers, allows developers to navigate the program's execution traces. The component is divided into two panels that both display traces representing method calls, variable assignments, and messages sent from one object to another. The left panel displays traces that execute before those in the right panel. More precisely, the traces are displayed according to their hierarchy. The left panel shows the parent level, and the right panel shows the child level. The right panel always shows traces that originate directly from the selected trace in the left panel at run time. When a new selection is made in the left panel, the right panel automatically updates. However, when a trace is selected in the right panel, that panel shifts to the left. This pushes the previous panels into a non-visible area. Then, a new panel is created to take the previous role of the child traces panel. When panels move into the non-visible area, the debugger displays a horizontal scroll bar that allows the developer to navigate the different levels of execution.

Following the selection of a trace, from anywhere in the Scopeo debugger interface, the pane labeled B gets updated to display the trace's details. When the trace reifies a message or method execution, the pane will display the object executing the method or receiving the message, its arguments, the return value, and so on. For an assignment, the pane displays the variable affected, as well as its values before and after the assignment. Additionally, this view shows the source code of the method from which the trace was recorded during execution. In the case of messages and assignments, it also highlights the corresponding line of code. In terms of functionalities, this view is equivalent to the combination of the code editor and the *Inspector* of the Pharo standard debugger (cf. section 3.1).

The orange rectangles in Figure 5.4 highlight the debugger components specific to Scopeo. The component labeled as C is a button that, when activated, opens a panel that allows the configuration of a query to run in order to explore the execution of the program being debugged. The configuration of a query with Scopeo is a three-step process. We will illustrate this process in section 5.4.1. After performing a query, a new exploration scope is created. Upon its creation, the scope appears in the list highlighted by the D label. Selecting a scope in this list refreshes the E component to show the details of the selection.

The E component is where the object-centric perspective is provided to developers. This component is divided into two parts, a top part and a bottom part. The top part is an execution tree, visually identical to component A. However, the execution tree of an exploration scope only contains the traces filtered by the query from which the exploration scope originated. Therefore, it is a subtree of the complete execution tree shown by component A.

The bottom part of the E component, contains the object-flow panels. The list on the left contains all the objects involved in the execution tree resulting

from the query that created the exploration scope. The list on the right displays all the methods, messages, and assignments involving the object selected in the left pane.

5.4 Case Study

This section shows how developers can use Scopeo in a debugging session. We conduct our case study using the scenarios of our empirical experiment, chapter 3. We then compare the solution to the proposed scenarios using object-centric breakpoints with the solutions we propose using Scopeo.

5.4.1 First Debugging Scenario: LightsOut

As shown in Figure 5.5, LightsOut is a game consisting of a 10x10 grid of cells. Each cell represents a light that can be in two different states: lit, which is indicated by yellow, or off, which is indicated by gray.

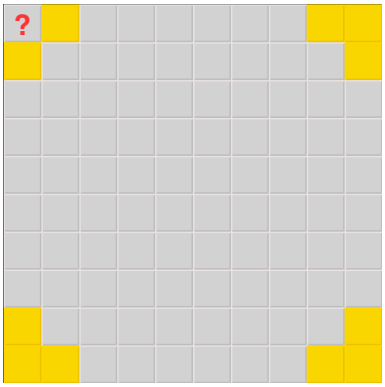


Figure 5.5: Screenshot of the Lights Out game graphical interface, with the bug symptom highlighted by red . A larger screenshot is available in subsection 3.2.4.

Due to a bug in the game, one of the lights placed in a corner of the grid never switches to the lit (yellow) state. Each time we restart the game, the bug's symptoms appear in a new corner of the grid. This information suggests that the bug is part of the grid initialization process, which will guide our debugging session.

Initiating the Debugging Session

To debug LightsOut, we first launch the game via Scopeo. In the current version of our prototype, this involves executing the following code.

```
ScopeoDebugger debug: [ LightsOut open ]
```

Running this code triggers the opening of the LightsOut interface. Clicking on the cells in the corners of the grid reveals that the top left corner at coordinates 1 @ 1 (in Pharo coordinate notation), presents the bug's symptoms. In addition to opening the LightsOut interface, running the sample code above also opens the Scopeo interface, which is shown in Figure 5.6.

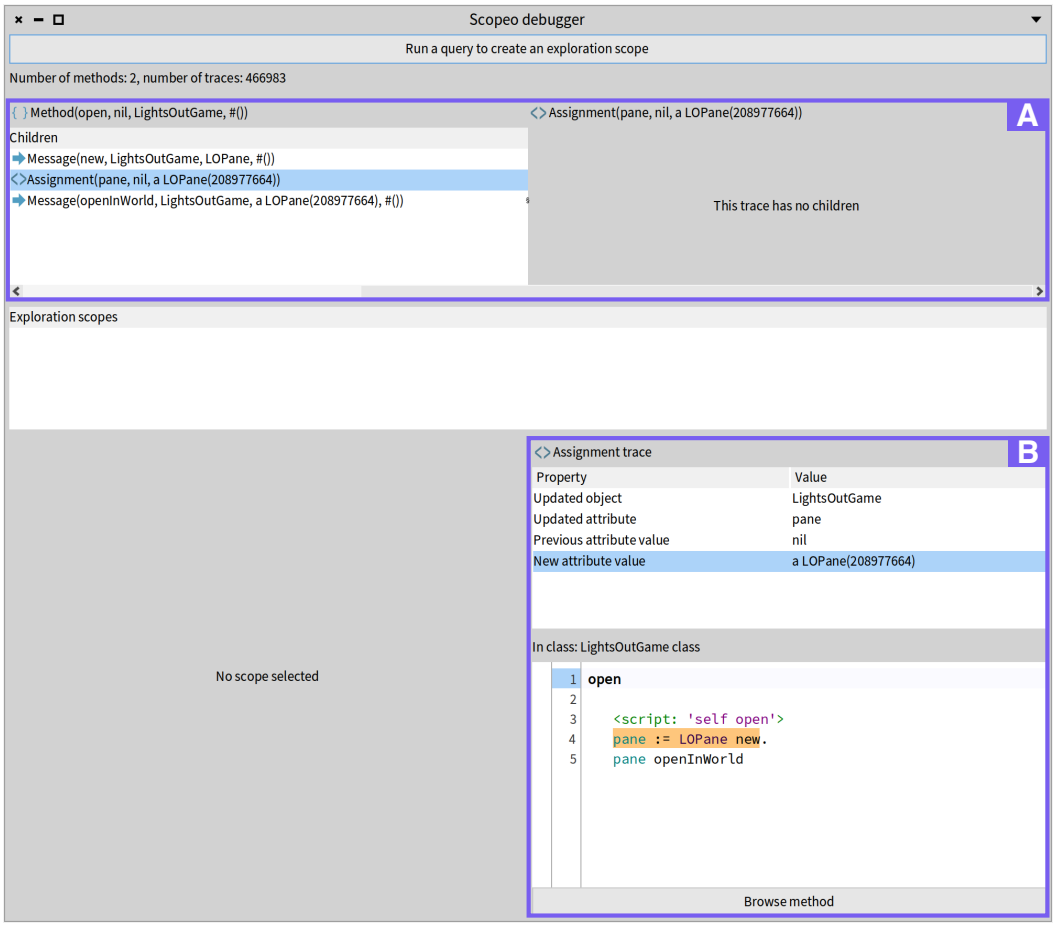


Figure 5.6: Screenshot of the Scopeo debugger opened on the execution of LightsOut. The label A designates the execution tree, while B designates the detailed view of the traces.

In the LightsOut execution tree A, Scopeo displays the first executed method by default. In our case, this method named open is visible thanks to the detailed view of execution traces B, illustrated in Figure 5.6. The code for this method shows the creation of a new LOPane object, which is then assigned as the value of the pane variable. The line of code responsible for the assignment is highlighted because, after this observation, we selected it in the execution tree A. Based on this initial observation, we assume that this code is responsible for creating the LightsOut grid. To verify this, we open the Pharo *Inspector* to view the details of the LOPane object. To perform this action, as showcased by Figure 5.7, we use the context menu of the trace detail view B. The inspector opens, allowing us to see the game's grid of cells exactly as it in Figure 5.5. Therefore, our assumption is correct, the LOPane object represents the grid.

Performing a Query

We would now like to retrieve all instructions involving the LOPane object created at the beginning of the execution. Rather than manually exploring the execution, we can use one of Scopeo's queries to do that. Before doing so, however, there is an additional step. We need to mark the LOPane object as the object about which we are going to ask the question. In Scopeo, we call the objects marked as input for future queries the *subjects*.

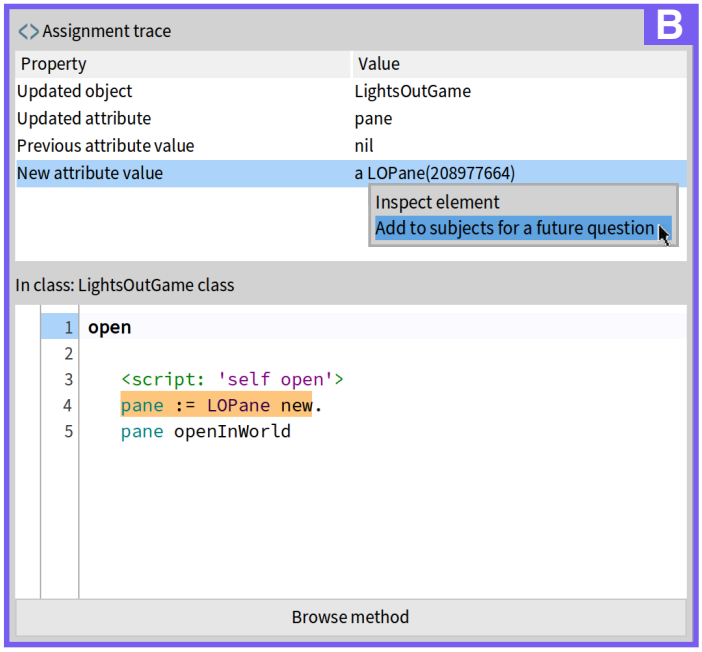


Figure 5.7: Screenshot of an assignment detailed view, labeled B, showing how to mark an object as a subject for future queries using the context menu.

For every possible type of trace, the detailed view B enables the opening of a context menu with the possibility of inspecting the objects referenced by the different fields of the trace. For a method or message trace, it is possible to inspect the arguments and the result. For assignments, we can inspect the values before and after they are assigned to a variable. To add an object as a subject for a future query we must use the option named *Add to subjects for a future question* from the context menu as visible in Figure 5.7. To perform a query, we must first click on the *Run a query to create an exploration scope* button in Scopeo's interface C. This action opens the query configuration window. This window will display a new panel for each of the three configuration steps illustrated by Figure 5.8 that we need to complete.

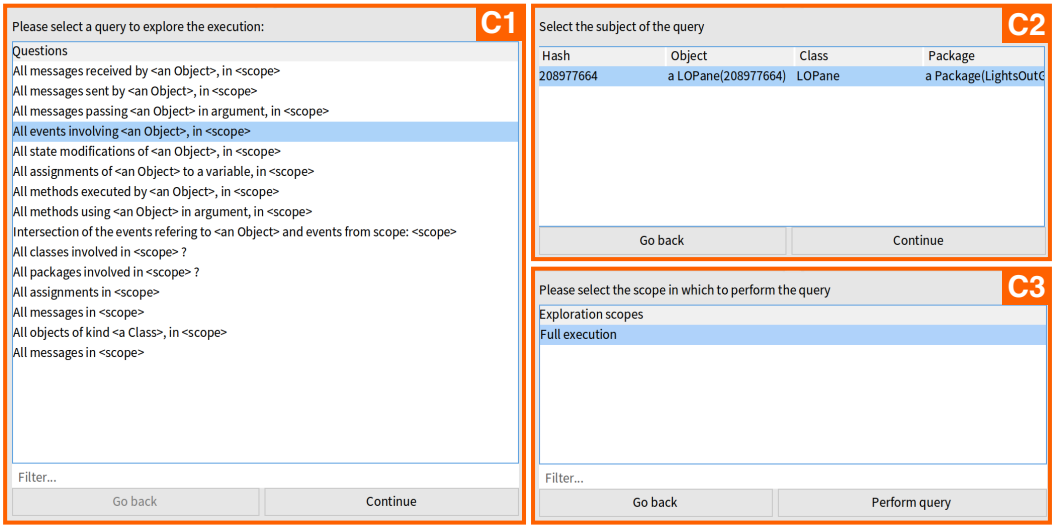


Figure 5.8: Screenshot of the configuration panels of a query. C1 is the pane for selecting a query, C2 for selecting a query's subject and C3 the exploration scope of delimiting the query's execution.

Of the pane showcased by Figure 5.8, the first one to be displayed in the query configuration window is the query selection pane C1. As its name states, this pane lists all queries currently supported by Scopeo. Scopeo provides a majority of object-centric queries, such as *All messages received by an object in a selected exploration scope*. While the object-centric queries allow developers to collect precise information about the behavior of objects they want to observe, in order to perform such queries, it is first necessary to collect objects of interest. To enable the collection of such objects of interest, we propose shipping Scopeo with queries for *finding initial focus points* [Sillito 2006, Sillito 2008, Kubelka 2019]. On example of such a query is *All classes involved in a selected exploration scope*, which is provided by *Seeker* [Willembrinck Santander 2023]. To retrieve all instructions

involving the `LOPane` object we are interested in, we select in C1 the query *All events involving <an Object> in <scope>*. In Scopeo's queries, words between brackets, such as *<an Object>*, *<scope>* and so on, indicate a parameter. These parameters can be selected using C2 for the subject of a query and C3 for the exploration scope. In our selected query, the subject is the object of interest, `LOPane`. Since we previously marked this object as subject for future queries (cf. section 5.4.1) it is available in C2, (cf. Figure 5.8). Therefore, we select it in the table and click *Continue* to select the final parameter required to configure our query, the exploration scope. Since this is the first query of our debugging session, the only available exploration scope is the default one, the *Full execution* scope, as displayed in C3. The Full execution scope contains the entire execution tree. Visualizing this scope would result in the same view displayed by the A component in Figure 5.4 and Figure 5.6. After selecting the exploration scope named *Full execution* and clicking the *Perform query* button, Scopeo closes the query configuration window and computes the query results, yielding a new exploration exploration scope that is a subtree of *Full execution*.

Investigating the Exploration Scope

After running the query, the resulting exploration scope is displayed in the following list displayed by Figure 5.9 and labeled D.

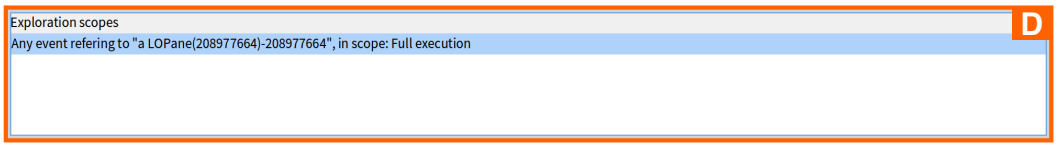


Figure 5.9: Screenshot of the list of the available exploration scopes, labeled D as in the full debugger screenshot Figure 5.4.

As shown in Figure 5.9, the parameters *<an Object>* and *<scope>* of our query *All events involving <an Object> in <scope>* are replaced by the string representation of the subject and the exploration scope that we selected for the query in the previous subsection. To help developers distinguish between exploration scopes originating from queries about objects of the same type, the string representation of each object is concatenated with its unique numerical identifier. For example, `LOPane(208977664)-208977664`. As in the latter example, the unique identifier may appear twice in the string representation. Indeed, by default objects provide their identifier in parentheses when asked for a string representation. We duplicate and concatenate again this number, because we must account for possible overrides of this default behavior.

To explore our query result we only need to select the corresponding exploration scope in the list of component D. We now can observe the results displayed in the following screenshot of the component E, Figure 5.10.

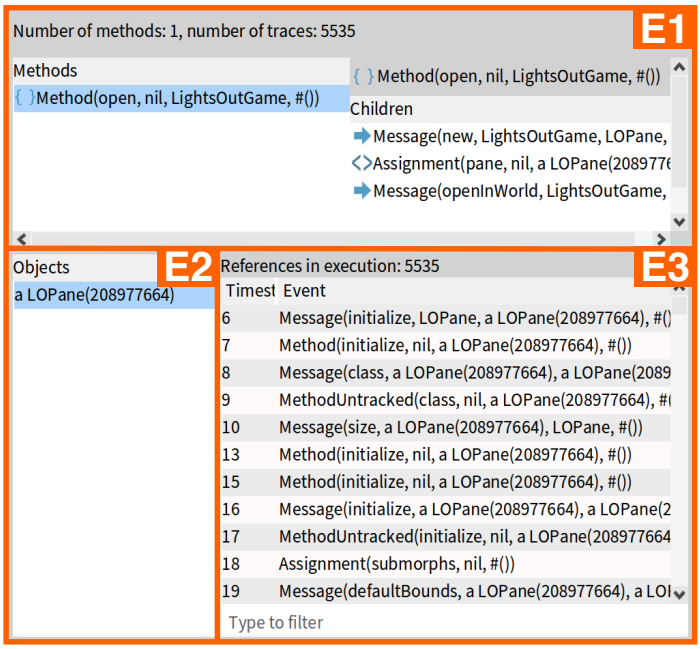


Figure 5.10: Screenshot showing the details of an exploration scope. E1 is the pane for exploring the scope execution tree, E2 for exploring the objects involved in the scope, E3 is the list of traces involving the selected object, i.e., its object-flow E2.

As shown in the screenshot Figure 5.10, the first traces involving our object of interest, `LOPane(208977664)`, are representing a message, `initialize`, that the object received, followed by the execution of a method of the same name. Selecting the latter trace updates the A, B, and E1 panes to show information about it. In the B pane we therefore can observe the source code of the method `initialize` executed by our object of interest. We read the code to understand how the `LOPane`, i.e., the grid of the LightsOut game is initialized. In this method, we can observe the following loop (cf. Listing 5.1) that initializes all the cells of the grid.

```
1 matrix := Array2D rows: size columns: size.
2 cells := matrix indicesCollect: [ :x :y |
3     self newCellAt: x @ y
4 ]
```

Listing 5.1: Listing of the code initializing LightsOut's grid of cells

In this initialization loop, we can observe that the message `newCellAt:` is sent to `self`, i.e. the `LOPane` (line 3). The next step in this debugging session is to get a reference to the cell presenting the bug, i.e., the cell that fails to switch to yellow.

Accessing the Object of Interest

The cell presenting the bug is the one in the top left corner (1 @ 1) of the LightsOut grid, and is initialized by the `newCellAt:` method. To obtain a reference to the object representing the faulty cell, and investigate about its behavior, we will first reduce our previous exploration scope. To do so, we type `newCellAt:` in the filter under the object-flow of the `LOPane` object, visible in the E3 window (cf. Figure 5.10). The filter updates the list to display only the traces containing anything related to the string `newCellAt:`. We obtain 203 results out of the 5535 traces involving the `LOPane`. At the beginning of the list, we select the trace that correspond to a `newCellAt:` message taking as argument the coordinates `1 @ 1`. Then, in component B, within the displayed details of the message we selected, we click on the value returned by the message, the object `LOCell(804933888)`. Similarly to what we did for our first query in Figure 5.7, we select this value as a subject for future queries.

Narrowing Down the Exploration Scope

We now would like to narrow down our exploration scope to observe only the interactions between our object of interest, the cell presenting the bug and the `LOPane`, the object responsible for its creation. For this, we perform a new query. In particular, we select the query *Any event refering to <an Object>, in <scope>:* in the query selection pane (annotated C1 in Figure 5.8). We then choose the `LOCell(804933888)` object as subject for our query using the dedicated component C2, as showcased by Figure 5.8.

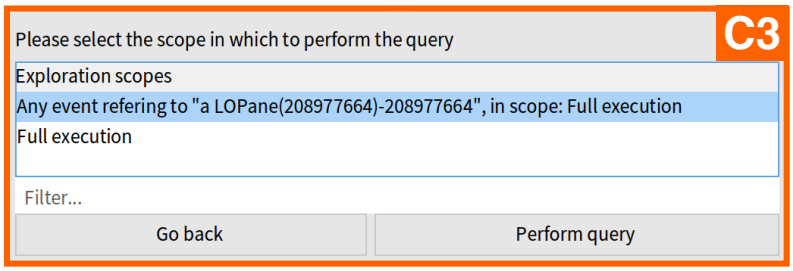


Figure 5.11: Screenshot showing the reuse of an exploration scope in a new query. C2 refers to the pane for selecting the exploration scope to perform a query on as in Figure 5.8.

The exploration scope that we select for this new query, in Figure 5.11, is the one yielded by our first query (cf. section 5.4.1). After performing this new query, Scopeo updates the E1 window to display a new exploration scope named *Any event refering to "LOCell(804933888)-804933888", in: Any event refering to "a LOPane(208977664)-208977664", in: Full execution.* The details of this new exploration scope are illustrated in below Figure 5.12.

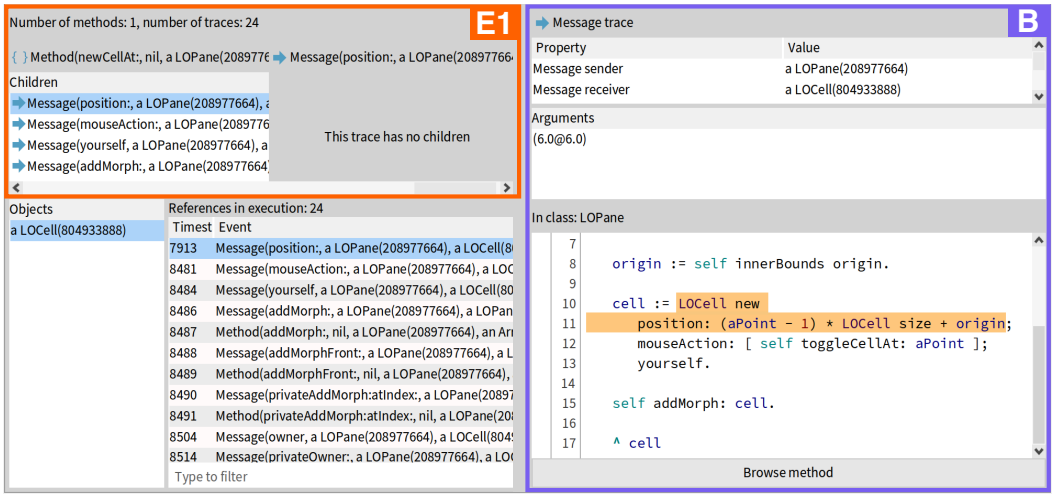


Figure 5.12: Screenshot of by a screenshot of the bottom part of the Scopeo debugger interface after selecting in D (cf. Figure 5.4) updates the exploration scope named *Any event refering to "LOCell(804933888)-804933888", in: Any event refering to "a LOPane(208977664)-208977664", in: Full execution.* Component B shows the details of the trace selected in E1.

The list of events involving the `LOPane` and `LOCell` all seem to occur during the initialization process of the game. As we can observe in components E1 and B in Figure 5.12, they take place in the context of the `newCellAt:` method (cf. Listing 5.1), where the `LOPane` creates the `LOCell` object, set its position, the action to perform on mouse click, and finish by adding the `LOCell` to the interface (using the `addMorph:` message).

Navigating Outside the Exploration Scope

Since the bug appears related to the positionning of the cell, we decide to explore the method executed after the `position:` message selected in E1. However, the E1 component only shows the sub tree of the execution containing the events that involve both the `LOPane` and `LOCell` objects. As the trace corresponding to the `position:` message has no children in displayed in E1, we deduce that only the `LOCell` is involved in the execution of the `position:` method. While this

provides us with some information, it does not help us to inspect the execution of this `position:` method. Therefore, to access the trace of the `position:` method execution, we use the complete execution tree as displayed by Figure 5.13.

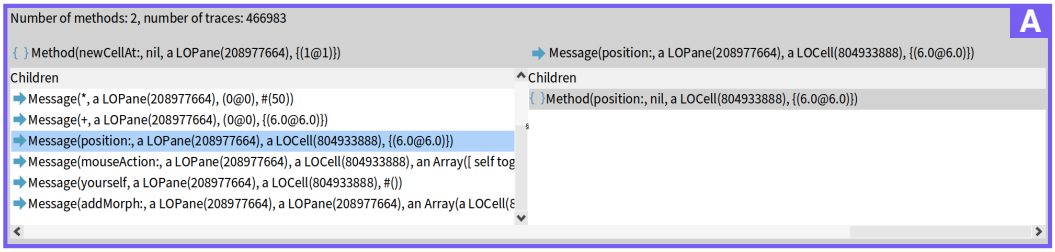


Figure 5.13: Screenshot of the entire execution tree, labeled A, where a trace is being selected.

Because the A component shows the execution of the program in an unfiltered manner, it allows us to access the trace of the `position:` method following the trace of the `position:` message (in the execution).

Identifying the Cause of the Bug

As the selection of the latter trace in component A triggers the update of the detailed view of the traces (i.e., component B in previous screenshots), we observe the following source code:

```
1 LOCell >> #position: aPoint
2   super position: aPoint.
3   aPoint = self class getCorner ifTrue: [
4     onColor := offColor.
5   ]
```

This source code contains a condition that can evaluate to true after calling the `getCorner` method. If this condition evaluates to true, the cell's `onColor` attribute takes the value of the `offColor` attribute. In other words, the yellow color representing the cell's lit state becomes identical to the gray color representing the off state (line 4). For this cell presenting the bug, it is impossible to differentiate between the lit and off states. Removing this method and rerunning the game supports the hypothesis that this `position:` method is responsible for the bug, because the symptom is no longer present after that.

5.4.2 Discussing the LightsOut Debugging Scenario

Through a debugging session of LightsOut using Scopeo, we illustrate how object-centric debugging tools based on an omniscient debugger allow developers debugging programs containing bugs that execute during initialization.

With Scopeo, we defined an exploration scope encompassing the initialization phase of the LightsOut grid and cells. We created this first scope from the object representing the grid. Then, we created a new exploration scope, that displayed only the initialization of the cell exhibiting the bug.

Using object-centric breakpoints, we would have applied either *Halt on call* or *Halt on state access* to the faulty cell. This would have paused the execution whenever the cell was updated or executed a method. To apply these breakpoints, we would have first needed to navigate the execution using stepping instructions, until the affectation to the `cell` variable (cf. component B in Figure 5.12). However, after executing the assignment instruction, the (faulty) `position:` method would have already been executed. This implies that, by trying to use object-centric breakpoints, we would have missed the cause of the bug. Ultimately, we would have had to restart the debugging process, and navigate to the faulty cell using standard breakpoints and stepping instructions, until the execution of the `position:` method.

This debugging scenario with Scopeo corroborates the observations we made in chapter 4, according to which omniscient solutions are better suited than object-centric breakpoints for observing the initialization process of objects.

The exploration scope which we narrowed down in this scenario is equivalent to the *All interactions between objects* query, which we implemented in the previous chapter chapter 4. Therefore, debugging experiences with Scopeo and Time-Traveling Object-Centric Breakpoints may be similar for developers in this scenario. A future comparison of the two approaches would be interesting for exploring the different uses of the *All interactions between objects* query. However, it is currently not possible because the TTOCB backend does not support executing programs that open graphical interfaces.

5.4.3 Second Debugging Scenario: Ammolite

Ammolite is our second scenario for exploring Scopeo's ability to support debugging iterations. Similarly to LightsOut, it is taken from our empirical experiment material (cf. subsection 3.2.4).

Ammolite, illustrated by Figure 5.14, is an application allowing teachers to create groups of students balanced according to their level. In the application, the level of a student is indicated by a marker, + or −. For a promotion of students, when generating the groups, the marker of one student named Adèle is disappearing in the list of groups.

Initiating the Debugging Session

To debug this scenario, we first launch the following code with Scopeo to record an execution that contains the bug. This code initializes Ammolite, and generates

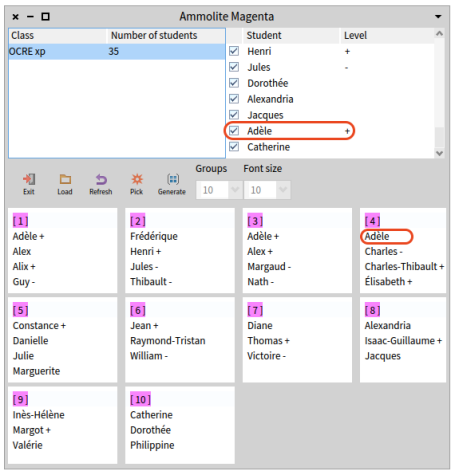


Figure 5.14: Screenshot of the Ammolite application with the bug symptom highlighted by a red question mark. A larger screenshot is available in subsection 3.2.4.

balanced groups of students. It also and opens Scopeo, similarly to Figure 5.6 for LightsOut.

```
ScpDebugger debug: [ AmmoliteMagenta run generate. ]
```

As before, our first objective in this exploration is to obtain a reference to an object of interest. In this scenario, the object of interest is Adèle, the student presenting the bug symptom. We begin by observing Ammolite's initialization process using the entire execution tree component A in Figure 5.15.

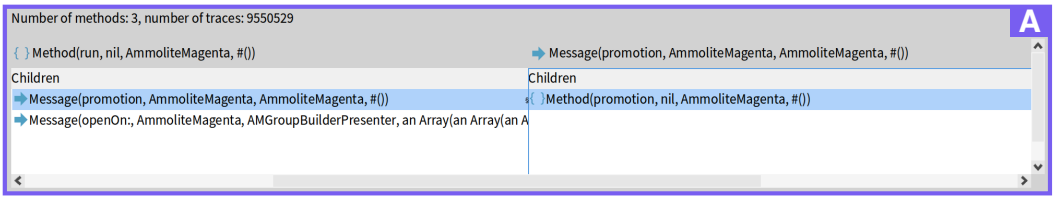


Figure 5.15: Screenshot of the execution tree (A) after Scopeo's opening on the Ammolite scenario.

In Figure 5.15, we observe a call to the `promotion` method. Using the detailed trace view (component B in previous screenshots) we read the following source code of the `promotion` method.

```
1  AmmoliteMagenta >> #promotion
2
3    | promotion students |
4
5    promotion := AMPromotion new.
6    promotion name: 'Ammolite promotion'.
7
8    students := (self students splitOn: Character cr)
9      collect: [ :studentDataString |
10       AMStudent readFromDataString: studentDataString
11     ].
12   promotion students: students.
13
14   ^ Array with: promotion
```

Listing 5.2: Source code of the `promotion` method in Ammolite.

In Listing 5.2 at line 10, we observe that in Ammolite, students are described by the class `AMStudent`.

Accessing the Object of Interest

Therefore, to retrieve our object of interest, we perform a first query, *Objects which are of class <a Class> in scope: <scope>*. After selecting the query in the list of query of component C1 , we select the class `AMStudent` in the list of possible subjects in component C2 , as illustrated by Figure 5.16.

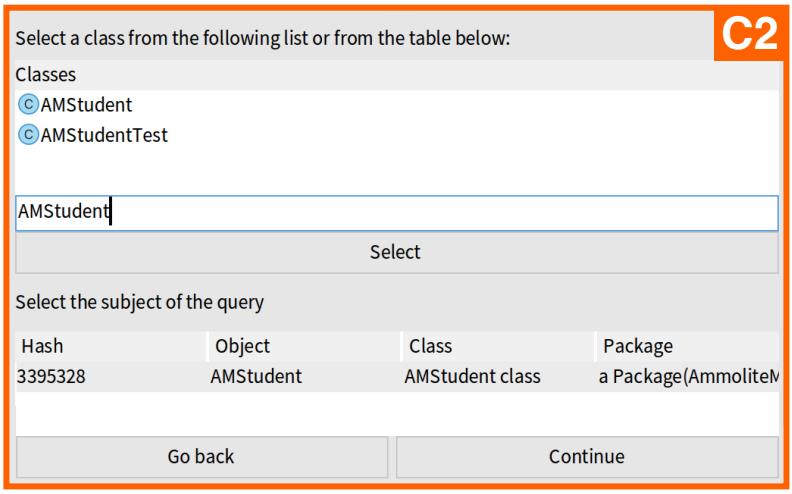


Figure 5.16: Screenshot of the subject selection pane, labeled C2 , showing how the interface automatically adapts to allow selection of a query parameters. In this example, a class is being selected for the query *Objects which are of class <a Class> in scope: <scope>*.

We perform the query in the *Full execution* scope, which yields as a result the new exploration scope *Objects which are of class "AMStudent" in scope: Full execution.* As a result, we obtain all the instanciated objects E2 and their associated object-flow E3, as displayed by Figure 5.17.

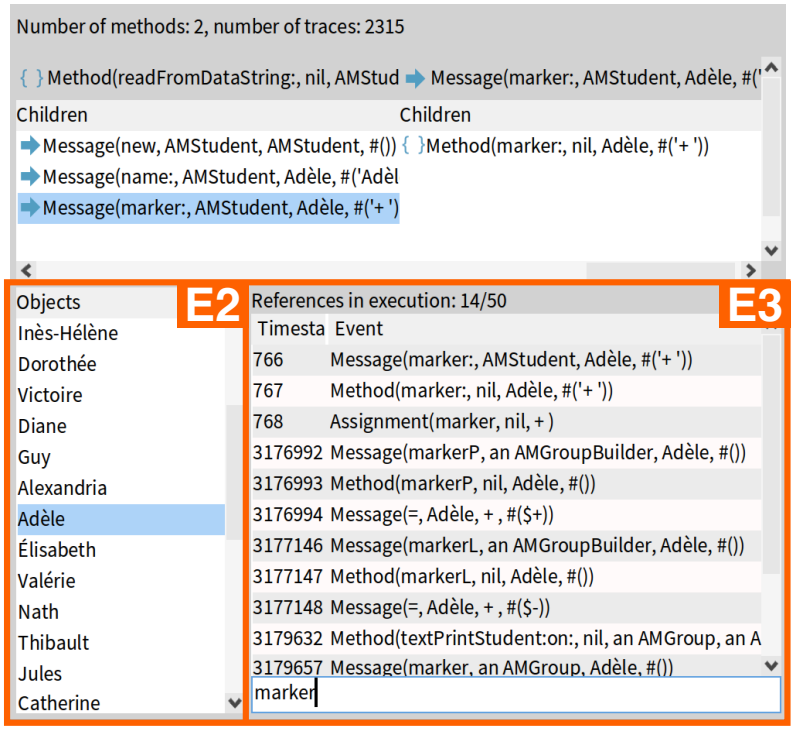


Figure 5.17: Screenshot of the exploration scope name *Objects which are of class "AMStudent" in scope: Full execution.* E2 shows the list of all students created in the execution of Ammolite. E3 displays the events involving the student named Adèle, selected in E2.

Identifying the Cause of the Bug

As showcased in Figure 5.17, we selected in the list the student named Adèle and filtered in its object flow all the events refering to the `marker`. Among the 14 resulting traces, we can observe in the first 3 ones the assignment of the string + with a trailing space. A little bit further in the list we also observe an call to the accessor of the marker made from the `textPrintStudent:on:` method showed in Listing 5.3.

```
1 AMStudent >> #textPrintStudent: student on: stream
2
3   student textPrintOn: stream.
4   ({ '+'. '-' } includes: student marker) ifFalse: [
5     ^ self
6   ].
7
8   stream space.
9   stream << student marker asString
```

Listing 5.3: Source code of the `textPrintStudent:on:` method in Ammolite.

As we can observe in the source code of the `textPrintStudent:on:` method, a verification is performed against the marker (line 4). If the marker does not matches with a plus or a minus symbol, the method returns immediately (line 5), without concatenating the marker to the string representation of the student (line 8 and 9). Since the value affected to the marker of our object of interest presents a trailing space, it does not satisfies the condition, which is the reason why it does not appear in Ammolite's interface. After adding an instruction to trim the value before affecting it to the marker, the marker is correctly printed on the interface, in the list of generated groups, suggesting that we correctly fixed the bug.

5.4.4 Discussing the Ammolite Debugging Scenario

As in the case of LightsOut, Ammolite is based on experimental materials from our empirical study evaluating the object-centric breakpoints on debugging (cf. chapter 3). Therefore our observations and the procedure we followed while conducting the debugging session can be influenced by this prior knowledge of the scenario.

In the empirical experiment, participants were being provided with a reference to the object presenting the symptom of the bug. Without such advantage, to debug Ammolite using object-centric breakpoints, we would need to step through the execution until the complete creation the list of students displayed in Ammolite.

Using Scopeo, with a single query, we could list all potential objects of interest. This suggest that having dedicated queries to initiate debugging sessions [Sillito 2008, Kubelka 2019] operating within the framework of an omniscient debugger could help developers debugging. This observation is consistent with the empirical results obtained by studying the impact of Seeker [Willembrinck Santander 2023], where similar queries were used.

After identifying the object exhibiting the symptom, we used only its object-flow view to continue our exploration. Filtering the view allowed us to find all references to the problematic `marker` attribute. Although equivalent to setting the object-centric breakpoint that pauses execution when an attribute is accessed

or modified (i.e., the *General Field Breakpoint* in section 3.1), we argue that this manipulation is more straightforward. Indeed, in comparison, to place the equivalent breakpoint on the `marker` attribute of the object, we would have needed to open an *Inspector* on the object, navigate to the view for the attributes, and then select the breakpoint through the menu.

Instead of directly exploring the filtered object-flow view, we could have executed new, more precise queries. Although our choice may have been influenced by our prior knowledge of Ammolite, we consider a set of 14 filtered results to be small enough to navigate directly. These results allowed us to observe the bug's symptoms and the helpful source code, so it appears that combining the object-flow view with filters can minimize the number of queries required to understand program execution.

5.4.5 Third Debugging Scenario: Microdown

Microdown [Ducasse 2020] is a variant of the Markdown markup language used to support Pharo [Black 2009] documentation. Microdown documents can contain JSON blocks of metadata within the document header. However, a bug was introduced to the Microdown parser, as illustrated by Listing 5.4. Since then, whenever the parser reads a valid metadata block (on the left of the listing), the resulting block ((on the right side of the listing)) only contains the last line. As we can observe on the right of the listing, the contents of line 2 has disappeared.

```
1 {                                  {
2  "authors":"S. Ducasse" ,
3  "title":"Pharo by Example"         "title":"Pharo by Example"
4 }                                  }
5
6 # Pharo by Example.                # Pharo by Example.
7
8 Text content...                    Text content...
```

Listing 5.4: Example of a Markdown document with JSON metadata in the header, that highlights the Microdown bug. The source code on the left shows the original block of metadata. The result of the parsing is shown on the right, with missing information on line 2.

The symptom of this bug is caught by the following unit test in Listing 5.5. When executed the unit test fails with the error, *KeyNotFound: key "authors" not found in Dictionary.*

```
1  MicMetaDataBlockTest >> #testAtKey
2
3   | source root metadata |
4
5   source := '{
6     "authors" : "S. Ducasse" ,
7     "title" : "Pharo by Example"
8   }'.
9
10  root := parser parse: source.
11  metadata := root children first.
12
13  self assert: (metadata atKey: 'authors')
14       equals: 'S. Ducasse'.
15
16  self assert: (metadata atKey: 'title')
17       equals: 'Pharo by Example'
```

Listing 5.5: Source code of the `testAtKey` unit test that highlights the Microdown bug.

The error is triggered by the code in the first assertion (lines 13 and 14). After parsing the string of metadata defined at the beginning of the test code (lines 5–8), the resulting object stored in the `metadata` variable does not contain the expected key, `authors`. Therefore, this unit test precisely reflects the example of Listing 5.4, with the error triggered by the absence of data on line 2 (in the example) after Microdown parsing.

Initiating the Debugging Session

We begin debugging this scenario in the same way as previous ones. Using Scopeo, we launch a code snippet to reproduce and capture an execution containing the bug symptoms.

```
1  [ MicMetaDataBlockTest new setUp; testAtKey; tearDown ]
2    on: Error do: [ :e | e crTrace ]
```

In the above code snippet, we prepare, launch the unit test (line 1) and catch the error resulting from the Microdown bug (line 2) to avoid pausing the execution and therefore the recording performed by Scopeo.

Our goal is to isolate the behavior dedicated to the parsing of the metadata, were the bug should be located. To do so, we first seek all the events in the execution in which the Microdown parser is involved. We begin by skimming the execution down to the execution of `testAtKey`, the unit test method, until reaching the message statement `parser parse: source` (line 10). To navigate in the execution until that line, we used the component A (cf. Figure 5.18), as presented across previous scenarios.

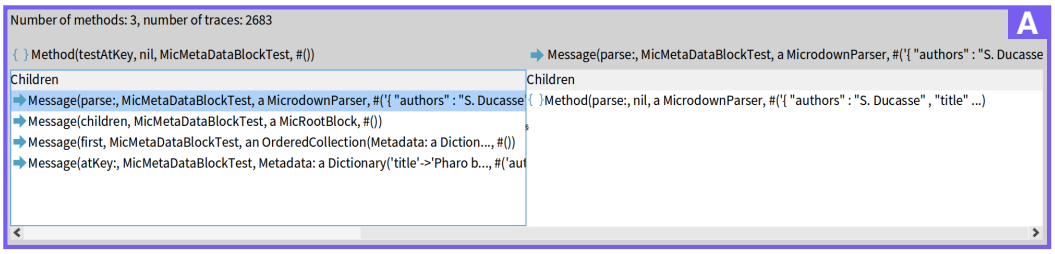


Figure 5.18: Screenshot of the execution tree (A) after Scopeo's opening on the Microdown scenario.

At this point in the execution we add the object referenced by the variable `parser` to the list of subjects for a future query, using the detailed view of the message trace B as shown in Figure 5.19.

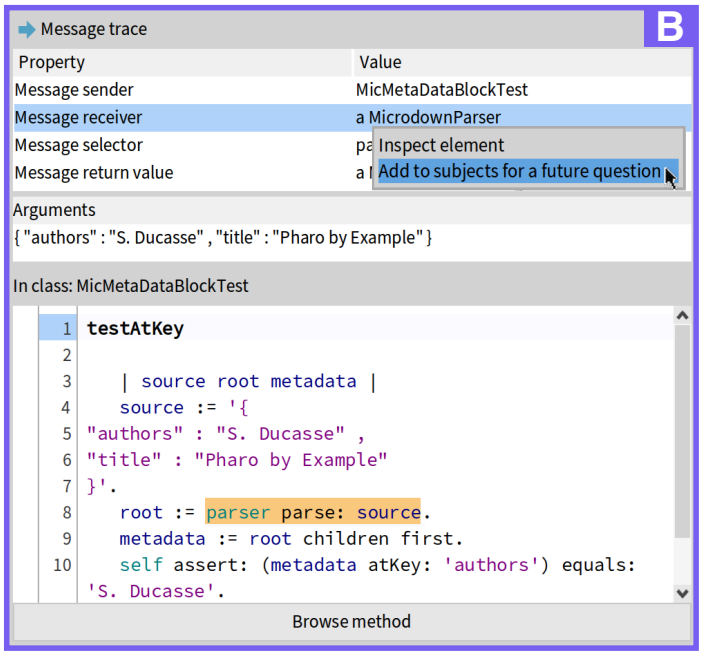


Figure 5.19: Screenshot of an message trace detailed view, labeled B, showing how we mark the value of the `parser` variable as a subject for future queries using the context menu.

We then perform the query *Any event refering to <an Object>, in scope: <scope>*, with our selected subject, the object referenced by the variable `parser` and in the scope of the *Full execution* (using the C components previously described, C1, C2 and C3).

Narrowing Down the Exploration Scope

Among these events where the `parser` object is involved we are interested only in those that also involve the results we obtain from the parsing, pointed by the `metadata` variable. The underlying question we are wondering about is how the parser creates the reified version of the metadata. To answer this question we follow the same steps as we just did to create our exploration scope with all the `parser` related events. We navigate the execution to reach the line number 11 using component A, as in Figure 5.18. Then we select the value of the `metadata` variable as subject for a query, as shown by Figure 5.19. Once again we perform the query *Any event refering to <an Object>, in scope: <scope>*. But this time our subject is the value of the `metadata` variable, and the exploration scope on which to perform the query is the one resulting from the last step. The new resulting scope entitled *Any event refering to "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution* is illustrated by Figure 5.20.

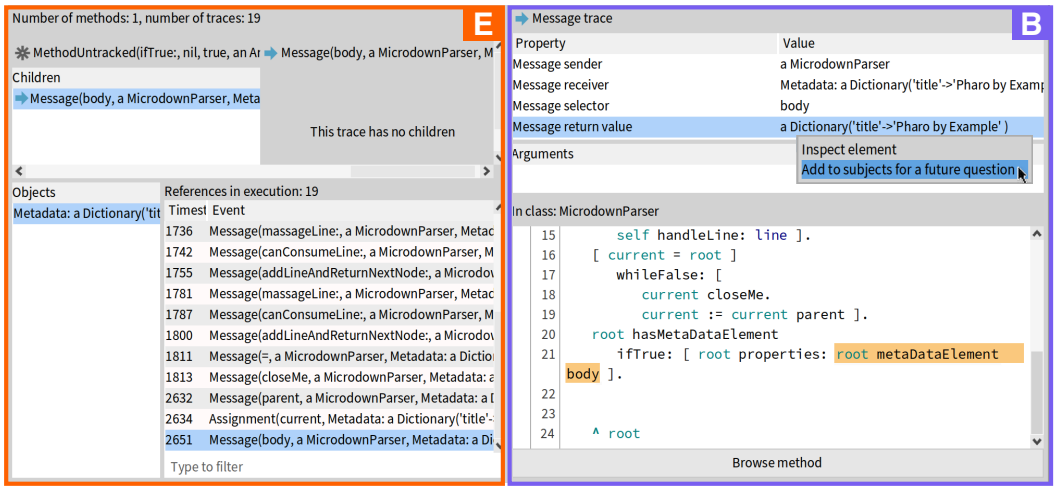


Figure 5.20: Screenshot of the exploration scope E named *Any event refering to "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution*. The detailed trace view B shows the event corresponding to the message body selected in component E.

When examining the events involving both the `metadata` and `parser` objects using the E component in Figure 5.20, we observe various parsing-related events. These events do not provide insightful information about the cause of the bug. However, we observe that the 19th result is a message that returns a dictionary matching the corrupted version of the metadata in our example Listing 5.4. We therefore wonder how does the `parser`, the `metadata` object and this dictionary interact with each other. To answer that question, we perform, once again a the query *Any event refering to <an Object>, in scope: <scope>*. As a subject to this query, we select the dictionary returned by the `body` message, using the context menu shown in component B in Figure 5.20. For the scope on which to execute the query, we select our last exploration scope. Performing this query yields the following exploration scope named *Any event refering to "a Dictionary('title'->'Pharo by Example')-107969012", in scope: Any event refering to "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution.*

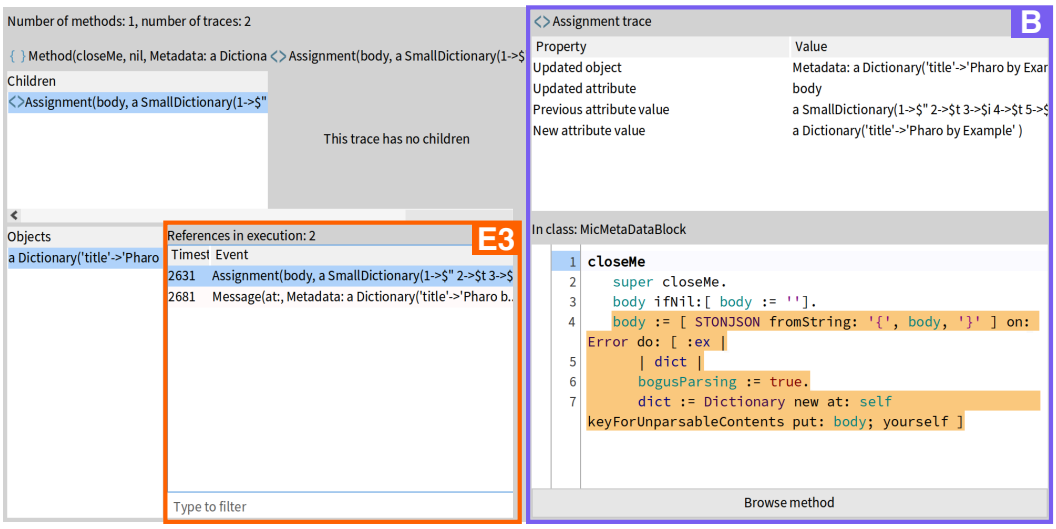


Figure 5.21: Screenshot of the exploration scope yielded by the query *Any event refering to "a Dictionary('title'->'Pharo by Example')-107969012", in scope: Any event refering to "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution.* E3 highlights the object-flow of the object a `Dictionary('title'->'Pharo by Example')` and B shows the source code of the assignment trace selected in E3.

Figure 5.21 shows the exploration scope resulting from the query. In the source code of the assignment, visible in B (line 4), we observe the conversion of a string into a JSON object, which is interesting since Microdown metadata are in JSON. This conversion is made by the statement `STONJSON fromString: '(', body, ')'`. However, at this point in the execution, as we can observe in the pane at the top of the component B, that the value of the `body` variable is an object of type `SmallDictionary`. From these observations, it seems that the bug is due to the fact that the body of the `metadata` object is a dictionary rather than a string of characters.

Creating a New Exploration Scope

To confirm this hypothesis, we search for all the assignments to the `body` variable in order to observe the evolution of its values. To do so we perform the query *All state modifications of the object <an Object>, in scope: <scope>*. For this query we reuse the dictionary object that we previously selected as subject for queries and the full execution for the exploration scope parameter. This creates the new exploration scope named *All state modifications of the object "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution.* In the object-flow view (component E3) of the object `Metadata: a Dictionary('title'->'Pharo by Example')-216975360`, we use the filter to show only the assignments refering to the `body` variable.

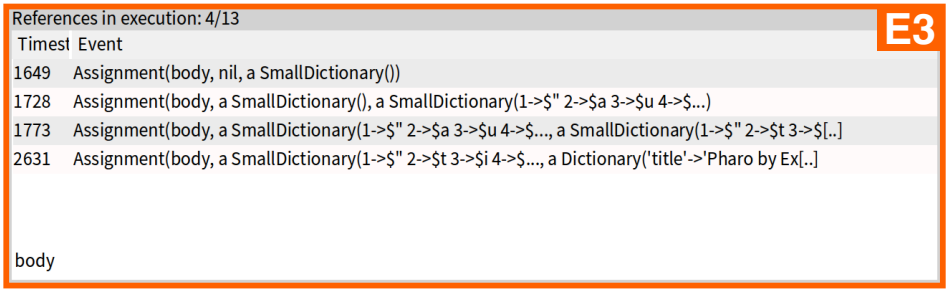


Figure 5.22: Screenshot of the object flow (component E3) of `Metadata: a Dictionary('title'->'Pharo by Example')-216975360`, in the exploration scope yielded by the query named *All state modifications of the object "Metadata: a Dictionary('title'->'Pharo by Example')-216975360", in scope: Full execution.*

As the results in Figure 5.22 indicate, the object is modified four times in the execution.

Identifying the Cause of the Bug

When looking at the details using component E3, we observe a first assignment of the value of a `SmallDictionary` to the `body` variable. Then, two more modifi-

cations occur. Using the component B , we realize that both these modifications result from a concatenation operation between the dictionary and the content of the metadata from the unit test (line 6 and 7). This concatenation happens in the following source code.

```
1 MicMarkupBlock >> #bodyFromLine: line
2   body := body
3     ifNil: [ line ]
4     ifNotNil: [  body , String cr , line ]
```

After each execution of this code, the variable `body` contains a dictionary object containing as key-value pair, the indexes and letters of the concatenated string, e.g. `1->", 2->t, 3->i, 4->t, 5->r, 6->e, 7->"`. From this observation we learn that concatenating a dictionary with a string of characters will always yield a new dictionary with these aforementioned key-value pair of indexes and letters. Once again, it seems that body should be of a type `String` and not `SmallDictionary`. To verify that, we remove the first assignment to the `body` variable after reaching its source code by a click in the object-flow cf. Figure 5.22. We execute the unit test again. The test now passes which suggest that we correctly fixed the bug.

5.4.6 Discussing the Microdown Debugging Scenario

Of the three scenarios of our investigation, the Microdown scenario is the only one that is not derived from the empirical experiment presented in the first chapter of this thesis, chapter 3.

To resolve the Microdown scenario, we narrowed down the exploration scope twice. As a result, we obtained a new exploration scope containing all interactions between the Microdown `parser`, the `Metadata` object referencing the metadata after parsing, and the metadata itself. This process of refining the exploration scope allowed us to highlight the `body` attribute of the `Metadata` object as potentially part of the problem.

Next, we created a new exploration scope that included only changes to the state of the `Metadata` object. Unlike other omniscient debuggers in the literature that base program execution exploration on queries (such as *Seeker* [Willembrinck Santander 2023] and *Trace Debugger* [Thiede 2023a]), performing a new query does not erase the exploration scope (or results) originating from previous investigations. With Scopeo, it is always possible to return and inspect a previous exploration scope, such as with *Compass* [Lienhard 2006] or *Whyline* [Ko 2008].

The last query we executed is equivalent to the object-centric breakpoint that pauses execution when an attribute is modified. We previously refered to this breakpoint as the *Field Breakpoint* in section 3.1 and the *Halt on Write* in section 4.2. However, for the same reasons as in the LightsOut scenario, this object-centric breakpoint would not have returned the assignment of a dictionary to the `body` variable, which is the cause of the bug.

5.5 Discussion

In this section, we present additional observations from our scenarios. We discuss differences with object-centric breakpoints, the impact of the exploration scope and limitations of our approach.

5.5.1 Differences Between Scopeo and Object-Centric Breakpoints

To illustrate how Scopeo's debugging support compares to object-centric breakpoints, we reused the scenarios from our empirical experiment. Although our prior knowledge of LightsOut and Ammolite may have influenced the solutions we brought to these scenarios using Scopeo, it was the only way to make such a comparison without conducting a dedicated experiment. As mentioned in the discussion of our scenarios, thanks to its omniscient nature, Scopeo allows developers to explore the initialization process of the objects they want to debug without restarting program execution or switching to other debugging tools. When debugging an object with object-centric breakpoints, however, developers seem to access the object through a variable or method that references it during execution. The problem with this method of access is that the object has a greater chance of being fully initialized, so developers cannot observe the initialization process. These observations corroborate the results of our previous experiments in chapter 3 and chapter 4. Therefore, if developers want to address a bug that appears to be part of an object's initialization process, we recommend using an omniscient debugging solution.

5.5.2 Impact of the Exploration Scope

We used the exploration scope in two out of three of our scenarios, LightsOut, and Microdown. Scopeo supported the verification of our hypotheses as they became more detailed, especially with Microdown, by storing exploration scopes and allowing their reuse for new queries. Specifically, using exploration scopes enabled us to reduce the amount of information that needed to be inspected to verify our hypotheses, thus facilitating the understanding of bug scenarios. This suggests

that the exploration scope can help reducing the cognitive cost of program comprehension. In comparision, other approaches, such as *Unstuck* [Hofer 2006] or *Seeker* [Willembrinck Santander 2023], offer queries that filter only complete program executions. These queries can yield a considerable number of results, requiring more effort for analysis than a narrowed down exploration scope. However, our three scenarios do not provide insight into the cognitive effort required to narrow down the exploration scope or how often this occurs. Therefore, more research is needed to determine the actual impact of the exploration scope on debugging activities.

5.5.3 Problems and Limitations

Our approach may have limitations and problems that could hinder its adoption and evaluation compared to other approaches, which we discuss in this subsection.

Bug reproduction With our current prototype, reproducing a bug in Scopeo requires executing the code that is suspected to be faulty. For small programs, like those in our scenarios, it does not seem to be problematic. However, for large programs, prior knowledge of the program is required to isolate a code sample that can reproduce the problem while keeping it as small as possible. This sample must be as small as possible because omniscient debuggers consume additional hardware resources, including a lot of memory, to capture the program's execution history.

Additionally, some bugs arise from a specific sequence of user actions through the graphical interface, such as mouse movements or button presses. Since Scopeo requires code as input, these actions must be translated into a programmatic version. This code can be challenging to develop because it often necessitates an understanding of how the graphical interface was designed. This represents an additional cognitive cost that may hinder the adoption of Scopeo. This raises the question of how to capture a bug without writing code.

User experience The E1 component displays the execution subtree of the exploration scope selected in the D component. When executing our scenarios, we did not use the E1 component, therefore we question its usefulness for developers when debugging. However, we may have rarely used this component because our scenarios did not require it or because the tree-like representation of the exploration scope was not helpful to developers.

To execute queries with Scopeo, we implemented a multi-step workflow. First, we pre-select an object as a potential subject of a query from a point in the execution. Then, we select the query, the subject, and the exploration scope in which to

search and filter the elements. Repeating the steps of this workflow can generate cognitive costs added to the cognitive overhead required to understand the program. Although we argue that this additional cost is inherent to every advanced debugging tool, more research is necessary to determine its magnitude.

In our prototype, the exploration scope names are a concatenation of the name of the query from which they originate, and the names of that query's parameters. As the exploration scope is refined, the names of new exploration scopes become increasingly long as they include the names of previous exploration scopes. Identifying these scopes can therefore become difficult, and may hinder the adoption of our approach in the future.

If future experiments with scoped exploration prove promising, addressing these aspects will be essential.

Limited Set of Queries As our queries selection is guided by our research focus on object-centric debugging, the set of queries integrated into Scopeo is limited. The queries we have included are equivalent to almost every object-centric breakpoint described in section 4.2, except for *Halt on Read* due to technical maturity issues. The additional queries are intended to help developers begin exploring their programs. It is possible that this set of queries we have chosen is not minimal or sufficient for developers to debug their object-oriented programs.

Throughout our scenarios, we identified that most queries we used were aimed at reducing the exploration scope rather than augment it. This can limit the impact of the exploration scope by forcing developers to create a new scope when they can no longer reduce the existing one. Adding queries that, for example, would allow developers searching for common elements between two exploration scopes, could enable them keeping the same exploration scope throughout their debugging session by updating it, rather than recreating new ones. We believe this could reduce the cognitive cost of debugging because the exploration scope would be a reification of the developer's journey through the execution history of the debugged program.

5.6 Conclusion

We presented Scopeo, an object-centric queryable omniscient debugger supporting debugging iterations by allowing developers to narrow down their exploration to subparts of the execution history called exploration scope. The exploration scopes are obtained and can be iterated over using queries.

In this chapter, we performed a case study on three scenarios to observe in practice the impact of introducting the exploration scope as an element to manipulate during debugging sessions. Our study suggest that it is possible to debug using such exploration scope, even though more research is needed to understand

its impact on debugging. Furthermore, this study confirms the observations we made in the previous chapter, suggesting the potential of query-based omniscient debuggers to address limitations encountered with object-centric breakpoints in the debugging scenarios of the experiment.

However, we also identified practical challenges. Currently, using Scopeo requires translating the steps to reproduce the bug into programmatic code to ensure the bug is captured. Compared to traditional call-stack based debuggers, which usually only require a click to activate, this is an additional burden for developers. Additionally, the limited set of queries integrated into Scopeo primarily focuses on narrowing the exploration scope, which can affect the workflow for exploring the execution and contribute to the cognitive cost of debugging.

To challenge our observations, future research should empirically evaluate the quantitative and qualitative [Creswell 2017] impact of the exploration scope on debugging activity. This will help us understand how developers would benefit from and perceive the tool. Setting up such experiment might require researchers to conduct preliminary experiments to confirm the limitations we identified and ensure the prototype is mature enough to consider launching the evaluation of the exploration scope on a larger scale. Based on our observations, the following work may be necessary: developing methods for capturing bugs without translating programmatically the reproduction steps, streamlining the query execution workflow, and expanding the range of available queries.

Chapter 6

Conclusion

Contents

In this chapter, we present the results of our research and offer our perspectives on future work.

6.1 General conclusion

The goal of this thesis is to improve debugging, particularly for object-oriented programs. To this end, we studied object-centric debugging, an approach that enhances traditional debugging tools by offering perspectives focused on objects rather than only on the source code.

Chapter 2 We explained the main challenge in debugging object-oriented programs, the conceptual gap between traditional debugging tools and the object-oriented paradigm. We also presented existing object-centric debugging approaches and their characteristics that enable them to address this challenge. After analyzing existing evaluations of these approaches, we emphasized the need for empirical research on the impact of object-centric debugging.

Chapter 3 To answer our first research question $\boldsymbol{RQ}_1$. *How does object-centric debugging impact the activity of debugging?*, we conducted an empirical experiment involving 81 developers. This experiment investigated in particular the impact of object-centric breakpoints on the activity of debugging. The study revealed the nuanced impact of object-centric breakpoints, showing that depending on the nature of the debugging task, they can be beneficial to developers or hinder the debugging. To the best of our knowledge this experiment is the first evaluating the impact of object-centric breakpoints. This highlights the need for more

research, particularly about the different types of bugs and how object-centric approaches affect the debugging of these bugs. Such research would help determine the situations in which these approaches are most helpful for developers.

Chapter 4 Following this experiment, we hypothezied that omniscient debugging approaches can address limitations of object-centric breakpoints. Therefore, to provide insight to our second research question $\boldsymbol{RQ_2}$. *What are the practical differences between object-centric approaches based on breakpoints and those based on omniscient debuggers?* we conducted a case study to compare both approaches. In particular, we introduced the Time-Traveling Object-Centric Breakpoints, an equivalent for object-centric breakpoints, based on the existing concept of Time-Traveling queries. We then compared how a developer would approach debugging with these different tools. Our observations support our hypothesis, however the benefits of query-based omniscient debuggers over object-centric breakpoints comes at a performance cost. Although research aimed at improving the performance of omniscient debuggers has already been conducted [Pothier 2007, Lienhard 2008], we recommend using these tools for smaller executions.

Chapter 5 Based on our understanding of the literature, we highlighted a gap in the support brought by object-centric debugging tools to the iteration of investigation and comprehension that constitute the activity of debugging. Therefore, we asked the following question $\boldsymbol{RQ_3}$. *Can we provide technological support for debugging cycles, and how does such support impact the debugging of object-oriented programs?*. To provide an answer to this question, we developed Scopeo [Bourcier 2025a], a query-based omniscient debugger incorporating exploration scopes to support debugging cycles. We highlighted through a practical case study how Scopeo's exploration scopes can support the debugging cycle. We observed promising results in narrowing down the search space for exploring the execution. However, our study revealed potential challenges and limitations in the bug reproduction and query creation workflows, as well as in the expressiveness of the queries themselves. We recommend addressing these limitations prior to conducting full scale empirical experiments to learn about the impact of Scopeo's exploration scope on debugging activity.

Our goal is to facilitate the debugging of object-oriented programs by developing object-centric debugging approaches. To this end, we investigated the impact of these approaches on debugging, compared the two main types of approaches: omniscient and breakpoint based. We have also opened a new avenue of research by developing Scopeo, an object-centric debugger that explores ways to support the debugging cycle. Our work calls for further research in object-centric debugging and establishes a foundation for future studies in this area.

6.2 Future Work

In this section, we offer multiple perspectives to further explore the impact of object-centric debugging and facilitate empirical experimentation in this field.

6.2.1 Replicate the Object-Centric Breakpoints Experiment

The conclusions drawn from the experiment presented in chapter 3 apply to the Pharo Smalltalk language. Pharo relies on a live development environment that allows developers to inspect and modify any object in a program or the environment itself while it is running. Therefore, Pharo developers are accustomed to observing objects in programs. This threaten the generalizability of our conclusions. To better understand the impact of object-centric breakpoints and verify that our conclusions apply to more general contexts, we must replicate the experiment with other object-oriented languages such as Python and Java.

Furthermore, our findings suggest that the impact of object-centric breakpoints varies depending on the bug being addressed. This raises the following question: in which cases these breakpoints can facilitate debugging and, conversely, hinder the process. Working on a bug taxonomy to answer this question would also open up a new avenue of research: preemptively identifying debugging techniques adapted to a situation. In the case of object-centric breakpoints, we wonder whether it is possible, before debugging begins, to identify characteristics that would indicate whether object-centric breakpoints will facilitate the comprehension of a given bug.

6.2.2 Empirically Evaluate Scopeo

Our case study on Scopeo does not allow us to determine its practical impact on debugging sessions led by developers. To learn more on this subject, we need to conduct an empirical experiment like the one reported in chapter 3. For this experiment, we propose using a within-participants design in which developers first perform a sequence of debugging tasks without Scopeo and then with it. We have identified two specific research questions to explore.

> **RQ**$_1$. How does the exploration scope affect developers' comprehension?
>
> **RQ**$_2$. How does the exploration scope affect the debugging process?

Our hypothesis is that developers will be able to gain a better understanding of programs with the exploration scope. By conducting this experiment with developers, we will gather information that provides new insights into omniscient, object-centric approaches. This information could help us identify limitations to

work on, and also potential benefits, which could open up new opportunities for developers. Based on our experience with experiments and the preliminary work required to prepare Scopeo for evaluation, we estimate that this experiment will take approximately one year to complete.

6.2.3 Support Debugging Experiment with Visualizations

As discussed in this dissertation, debugging is a challenging activity that requires significant cognitive effort. This makes conducting empirical experiments on debugging difficult, as it involves recruiting developers and requiring them to make considerable cognitive efforts.

Therefore, it is important to collect as much information as possible about the entire process for each experiment conducted. This involves both quantitative and qualitative information. Quantitative information includes time spent debugging and the number of actions performed. Qualitative data includes the tools used, the graphical elements manipulated in the debugging environment, and the developers' intentions and perceptions when using various tools. Because of its large volume, it is difficult to correlate the different metrics and interpret this data.

In parallel with the work presented in this document, we have started preliminary work to address this challenge and proposed the *Debugging Activity Blueprint* [Bourcier 2024a]. The Debugging Activity Blueprint is a visualization that allows one to observe how developers navigate the development environment during debugging. The goal of this visualization was to identify patterns that would enable us to explain, and subsequently analyze, debugging sessions. As shown in Figure 6.1 below, we identified certain patterns using three case studies. One pattern is the *chain*, which is a sequence of actions with different graphical windows of the same tool.

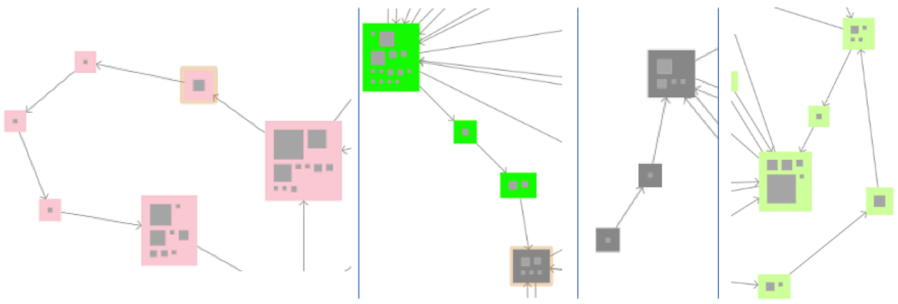


Figure 6.1: Screenshot of the *chain* pattern identified using the *Debugging Activity Blueprint* [Bourcier 2024a]. The pink color represents debugger windows, green highlights code browsers, gray refers to the debugged application and the light green to method implementors.

However, these three case studies do not provide enough evidence to confirm that the patterns we identified are actually patterns. Furthermore, the current state of this work necessitates that researchers interpret the developers' intentions from which the patterns emerge. This introduces a bias that could lead to false conclusions about the developers' intentions during debugging.

In future research, we propose conducting debugging sessions that follow the think-aloud protocol. This protocol requires developers to formulate their thoughts during debugging sessions. Afterwards, we propose to analyze the data used to generate the visualization using statistical tools such as Markov chains [Damevski 2016]. This initial analysis should allow to confirm or refute the patterns we've identified. Next, we propose mapping the audio samples from the think-aloud protocol to the identified patterns. This should enable researchers to link the developers' intentions to the visual patterns.

This work aims to accelerate the analysis of empirical debugging experiments results by proposing a tool that allows researchers to visually synthesize complex debugging sessions. Continuing this work would open up the possibility of conducting debugging experiments with a large number of developers while maintaining the ability of researchers to analyze the resulting data. Ultimately, the goal is to maximize the value of experimental research efforts, better understand developer behavior during debugging, and propose better research avenues and debugging tools.

Bibliography

[Ahmed 2023] Shibbir Ahmed, Mohammad Wardat, Hamid Bagheri, Breno Dantas Cruz and Hridesh Rajan. *Characterizing Bugs in Python and R Data Analytics Programs*, 2023. `https://arxiv.org/abs/2306.08632`.

[Alaboudi 2023] Abdulaziz Alaboudi and Thomas D LaToza. *What constitutes debugging? An exploratory study of debugging episodes.* Empirical Software Engineering, vol. 28, no. 5, page 117, 2023.

[Albluwi 2021] Ibrahim Albluwi and Haley Zeng. *Novice Difficulties with Analyzing the Running Time of Short Pieces of Code.* In Proceedings of the 23rd Australasian Computing Education Conference, ACE '21, page 1–10, New York, NY, USA, 2021. Association for Computing Machinery.

[Azadmanesh 2017] Mohammadreza Azadmanesh and Matthias Hauswirth. *How did the failure come to be?* In Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems, CoCoS 2017, page 13–18, New York, NY, USA, 2017. Association for Computing Machinery.

[Beller 2018] Moritz Beller, Niels Spruit, Diomidis Spinellis and Andy Zaidman. *On the Dichotomy of Debugging Behavior Among Programmers.* In Proceedings of ICSE 18: 40th International Conference on Software Engineering, 2018.

[Ben-Shachar 2020] Mattan S. Ben-Shachar, Daniel Lüdecke and Dominique Makowski. *effectsize: Estimation of Effect Size Indices and Standardized Parameters.* Journal of Open Source Software, vol. 5, no. 56, page 2815, 2020.

[Benjamini 1995] Yoav Benjamini and Yosef Hochberg. *Controlling the false discovery rate: a practical and powerful approach to multiple testing.* Journal of the Royal statistical society: series B (Methodological), vol. 57, no. 1, pages 289–300, 1995.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Blanton 2012] Ethan Blanton, Demian Lessa, Lukasz Ziarek and Bharat Jayaraman. *Ji.Fi: visual test and debug queries for hard real-time.* In Proceedings of the 10th International Workshop on Java Technologies for Real-Time

and Embedded Systems, JTRES '12, page 155–164, New York, NY, USA, 2012. Association for Computing Machinery.

[Bodden 2011a] Eric Bodden. *Stateful breakpoints: A practical approach to defining parameterized runtime monitors.* In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 492–495, 2011.

[Bodden 2011b] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati and Mira Mezini. *Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders.* In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[Böhme 2017] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe and Andreas Zeller. *Where is the bug and how is it fixed? an experiment with practitioners.* In Eric Bodden, Wilhelm Schäfer, Arie van Deursen and Andrea Zisman, editeurs, Proceedings of Foundations of Software Engineering ESEC/FSE, pages 117–128. ACM, 2017.

[Bourcier 2024a] Valentin Bourcier, Alexandre Bergel, Anne Etien and Steven Costiou. *Debugging Activity Blueprint.* In 2024 IEEE Working Conference on Software Visualization (VISSOFT), pages 48–58, 2024.

[Bourcier 2024b] Valentin Bourcier and Steven Costiou. *Scopeo: an Object-Centric Debugging Approach for Exploring Object-Oriented Programs.* In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland, 2024.

[Bourcier 2024c] Valentin Bourcier, Steven Costiou, Maximilian Ignacio Willembrinck Santander, Adrien Vanègue and Anne Etien. *Time-traveling object-centric breakpoints.* Journal of Computer Languages, 2024.

[Bourcier 2025a] Valentin Bourcier. *Scopeo (https://github.com/scopeo-project)*, 2025. `https://inria.hal.science/hal-04880231/` (Accessed: 2025-09-01).

[Bourcier 2025b] Valentin Bourcier, Pooja Rani, Maximilian Ignacio Willembrinck Santander, Alberto Bacchelli and Steven Costiou. *Replication Package for "Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs"*, February 2025. `https://doi.org/10.5281/zenodo.14844897`.

[Butt 2023] Nigar Azhar Butt, Salman Sherin, Muhammad Uzair Khan, Atif Aftab Jilani and Muhammad Zohaib Iqbal. *Deriving and Evaluating a Detailed Taxonomy of Game Bugs*, 2023. `https://arxiv.org/abs/2311.16645`.

[Catolino 2019] Gemma Catolino, Fabio Palomba, Andy Zaidman and Filomena Ferrucci. *Not all bugs are the same: Understanding, characterizing, and classifying bug types.* Journal of Systems and Software, vol. 152, pages 165–181, 2019.

[Chmiel 2004] Ryan Chmiel and Michael C Loui. *Debugging: from novice to expert.* Acm Sigcse Bulletin, vol. 36, no. 1, pages 17–21, 2004.

[Christensen 2015] Larry B. Christensen, R. Burke Johnson and Lisa A. Turner. Research methods, design, and analysis; 12th edition, global edition. Pearson, Boston, Mass y 2015, 2015.

[Cohen 1988] Jacob Cohen. Statistical power analysis for the behavioral sciences. routledge, 1988.

[Coppola 2024] Riccardo Coppola, Tommaso Fulcini and Francesco Strada. *Know Your Bugs: A Survey of Issues in Automated Game Testing Literature.* In 2024 IEEE Gaming, Entertainment, and Media Conference (GEM), pages 1–6, 2024.

[Corrodi 2016] Claudio Corrodi. *Towards Efficient Object-Centric Debugging with Declarative Breakpoints.* In SATToSE 2016, 2016.

[Costiou 2018] Steven Costiou. *Unanticipated behavior adaptation : application to the debugging of running programs.* PhD thesis, Université de Bretagne occidentale - Brest, November 2018.

[Costiou 2020a] Steven Costiou, Vincent Aranega and Marcus Denker. *Submethod, partial behavioral reflection with Reflectivity: Looking back on 10 years of use.* The Art, Science, and Engineering of Programming, vol. 4, no. 3, February 2020.

[Costiou 2020b] Steven Costiou, Mickaël Kerboeuf, Clotilde Toullec, Alain Plantec and Stéphane Ducasse. *Object Miners: Acquire, Capture and Replay Objects to Track Elusive Bugs.* The Journal of Object Technology, vol. 19, pages 1:1–32, July 2020.

[Costiou 2022] Steven Costiou, Vincent Aranega and Marcus Denker. *Reflection as a Tool to Debug Objects.* In International Conference on Software Language Engineering (SLE), pages 55–60, 2022.

[Cotroneo 2025] Domenico Cotroneo, Giuseppe De Rosa and Pietro Liguori. *PyResBugs: A Dataset of Residual Python Bugs for Natural Language-Driven Fault Injection.* In 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), pages 146–150, 2025.

[Creswell 2017] John W Creswell and J David Creswell. Research design: Qualitative, quantitative, and mixed methods approaches. Sage publications, 2017.

[Czyz 2007] Jeffrey K. Czyz and Bharat Jayaraman. *Declarative and visual debugging in Eclipse.* In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange, eclipse '07, page 31–35, New York, NY, USA, 2007. Association for Computing Machinery.

[Damevski 2016] Kostadin Damevski, David C Shepherd, Johannes Schneider and Lori Pollock. *Mining sequences of developer interactions in visual studio for usage smells.* IEEE Transactions on Software Engineering, vol. 43, no. 4, 2016.

[Denker 2008] Marcus Denker. *Sub-method Structural and Behavioral Reflection.* PhD thesis, University of Bern, May 2008.

[Détienne 2001] Françoise Détienne. Software design–cognitive aspect. Springer Science & Business Media, 2001.

[Ducasse 1999] Stéphane Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk.* Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999.

[Ducasse 2020] Stéphane Ducasse, Laurine Dargaud and Guillermo Polito. *Microdown: a Clean and extensible markup language to support Pharo documentation.* In Proceedings of the 2020 International Workshop on Smalltalk Technologies, 2020.

[Dupriez 2019] Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega and Stéphane Ducasse. *Sindarin: A Versatile Scripting API for the Pharo Debugger.* In International Symposium on Dynamic Languages (DLS'19), pages 67–79. ACM, 2019.

[Eisenstadt 1997] Marc Eisenstadt. *My Hairiest Bug War Stories.* Commun. ACM, vol. 40, no. 4, pages 30–37, 1997.

[Ericsson 2017] K Anders Ericsson. *Protocol analysis.* A companion to cognitive science, pages 425–432, 2017.

[Faul 2009] Franz Faul, Edgar Erdfelder, Axel Buchner and Albert-Georg Lang. *Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses.* Behavior Research Methods, vol. 41, no. 4, pages 1149–1160, November 2009.

[Fontana 2021] Eduardo Andreetta Fontana and Fabio Petrillo. *Mapping breakpoint types: an exploratory study.* In 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021.

[Funder 2019] David C Funder and Daniel J Ozer. *Evaluating effect size in psychological research: Sense and nonsense.* Advances in methods and practices in psychological science, vol. 2, no. 2, pages 156–168, 2019.

[Furnham 1986] Adrian Furnham. *Response bias, social desirability and dissimulation.* Personality and individual differences, vol. 7, no. 3, pages 385–400, 1986.

[Gelman 2014] Andrew Gelman and John Carlin. *Beyond power calculations: Assessing type S (sign) and type M (magnitude) errors.* Perspectives on psychological science, vol. 9, no. 6, pages 641–651, 2014.

[Gestwicki 2005] Paul Gestwicki and Bharat Jayaraman. *Methodology and architecture of JIVE.* In Proceedings of the 2005 ACM symposium on Software visualization, pages 95–104, 2005.

[Gilmore 1991] David J Gilmore. *Models of debugging.* Acta psychologica, vol. 78, no. 1-3, pages 151–172, 1991.

[Gray 1986] Jim Gray. *Why Do Computers Stop and What Can Be Done About It?* In Symposium on Reliability in Distributed Software and Database Systems, pages 3–12, 1986.

[Grottke 2007] Michael Grottke and Kishor S. Trivedi. *Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate.* Computer, vol. 40, pages 107–109, 2007.

[Gugerty 1986] L. Gugerty and G. Olson. *Debugging by skilled and novice programmers.* In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '86, page 171–174, New York, NY, USA, 1986. Association for Computing Machinery.

[Gyimesi 2021] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc and Ali Mesbah. *BUGSJS: a benchmark and taxonomy of JavaScript bugs.* Software Testing, Verification and Reliability, vol. 31, no. 4, page e1751, 2021.

[Hassan 2024] Mohammed Hassan, Grace Zeng and Craig Zilles. *Evaluating How Novices Utilize Debuggers and Code Execution to Understand Code.* In Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1, ICER '24, page 65–83, New York, NY, USA, 2024. Association for Computing Machinery.

[Heckman 1990] James J Heckman. *Selection bias and self-selection.* In Econometrics, pages 201–224. Springer, 1990.

[Hinkle 1993] Bob Hinkle, Vicki Jones and Ralph E Johnson. *Debugging objects.* In The Smalltalk Report, 1993.

[Hofer 2006] Christoph Hofer, Marcus Denker and Stéphane Ducasse. *Design and Implementation of a Backward-In-Time Debugger.* In Proceedings of NODE'06, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[Ioannidis 2008] John PA Ioannidis. *Why most discovered true associations are inflated.* Epidemiology, vol. 19, no. 5, pages 640–648, 2008.

[Jané 2024] Matthew B Jané, Qinyu Xiao, Siu Kit Yeung, Mattan S Ben-Shachar, Aaron R Caldwell, Denis Cousineau, Daniel J Dunleavy, Mahmoud Elsherif, Blair T Johnson, David Moreau*et al. Guide to effect sizes and confidence intervals*, 2024. `https://doi.org/10.17605/OSF.IO/D8C4G`.

[Jeffries 1982] Robin Jeffries. *A comparison of the debugging behavior of expert and novice programmers.* In Proceedings of AERA annual meeting, volume 10, pages 1–7, 1982.

[Kerby 2014] Dave S Kerby. *The simple difference formula: An approach to teaching nonparametric correlation.* Comprehensive Psychology, vol. 3, pages 11–IT, 2014.

[Kiczales 1991] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. The art of the metaobject protocol. MIT Press, 1991.

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* In Mehmet Aksit and Satoshi Matsuoka, editeurs, European Conference on Object-Oriented Programming (ECOOP'97), pages 220–242. Springer-Verlag, June 1997.

[Ko 2004] Andrew J. Ko and Brad A. Myers. *Designing the whyline: a debugging interface for asking questions about program behavior.* In Proceedings of the 2004 conference on Human factors in computing systems, pages 151–158. ACM Press, 2004.

[Ko 2005] Amy J Ko and Brad A Myers. *A framework and methodology for studying the causes of software errors in programming systems.* Journal of Visual Languages & Computing, vol. 16, no. 1-2, pages 41–84, 2005.

[Ko 2006] Amy J Ko, Brad A Myers, Michael J Coblenz and Htet Htet Aung. *An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks.* IEEE Transactions on software engineering, vol. 32, no. 12, pages 971–987, 2006.

[Ko 2008] Andrew J. Ko and Brad A. Myers. *Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior.* In In Proceedings of the 30th International Conference on Software Engineering, ICSE 08, 2008.

[Ko 2009] Amy J Ko and Brad A Myers. *Finding causes of program output with the Java Whyline.* In Proceedings of the SIGCHI conference on human factors in computing systems, pages 1569–1578, 2009.

[Kubelka 2014] Juraj Kubelka, Alexandre Bergel and Romain Robbes. *Asking and Answering Questions During a Programming Change Task in the Pharo Language.* In Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[Kubelka 2018] Juraj Kubelka, Romain Robbes and Alexandre Bergel. *The Road to Live Programming: Insights from the Practice.* In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pages 1090–1101, New York, NY, USA, 2018. ACM.

[Kubelka 2019] Juraj Kubelka, Romain Robbes and Alexandre Bergel. *Live Programming and Software Evolution: Questions During a Programming Change Task.* In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 30–41, 2019.

[LaToza 2020] Thomas D LaToza, Maryam Arab, Dastyni Loksa and Amy J Ko. *Explicit programming strategies.* Empirical Software Engineering, vol. 25, no. 4, pages 2416–2449, 2020.

[Layman 2013] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline and Gina Venolia. *Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers.* In 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, pages 383–392, 2013.

[Leesatapornwongsa 2016] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu and Haryadi S. Gunawi. *TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.* SIGPLAN Not., vol. 51, no. 4, page 517–530, March 2016.

[Lencevicius 1997] Raimondas Lencevicius, Urs Hölzle and Ambuj K. Singh. *Query-based debugging of object-oriented programs.* SIGPLAN Not., vol. 32, no. 10, page 304–317, October 1997.

[Lencevicius 2003] Raimondas Lencevicius, Urs Hölzle and Ambuj K Singh. *Dynamic query-based debugging of object-oriented programs.* Automated Software Engineering, vol. 10, no. 1, pages 39–74, 2003.

[Lewis 2003] Bill Lewis. *Debugging Backwards in Time.* In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03), October 2003.

[Lewis 2010] Chris Lewis, Jim Whitehead and Noah Wardrip-Fruin. *What went wrong: a taxonomy of video game bugs.* In Proceedings of the Fifth International Conference on the Foundations of Digital Games, FDG '10, page 108–115, New York, NY, USA, 2010. Association for Computing Machinery.

[Lienhard 2006] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba and Oscar Nierstrasz. *Capturing How Objects Flow At Runtime.* In Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06), pages 39–43, 2006.

[Lienhard 2007] Adrian Lienhard, Stéphane Ducasse and Tudor Gîrba. *Object Flow Analysis — Taking an Object-Centric View on Dynamic Analysis.* In Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07), pages 121–140, New York, NY, USA, 2007. ACM Digital Library.

[Lienhard 2008] Adrian Lienhard, Tudor Gîrba and Oscar Nierstrasz. *Practical Object-Oriented Back-in-Time Debugging.* In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[Lienhard 2009] Adrian Lienhard, Julien Fierz and Oscar Nierstrasz. *Flow-Centric, Back-In-Time Debugging.* In Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009, volume 33 of *LNBIP*, pages 272–288. Springer-Verlag, 2009.

[Lu 2019] Jiannan Lu, Yixuan Qiu and Alex Deng. *A note on Type S/M errors in hypothesis testing.* British Journal of Mathematical and Statistical Psychology, vol. 72, no. 1, pages 1–17, 2019.

[Marr 2018] Stefan Marr. *ReBench: Execute and Document Benchmarks Reproducibly*, aug 2018. Version 1.0.

[O'Dell 2017] Devon H O'Dell. *The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills.* Queue, vol. 15, no. 1, pages 71–90, 2017.

[Osman 2014] Haidar Osman, Mircea Lungu and Oscar Nierstrasz. *Mining frequent bug-fix code changes.* In 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pages 343–347, 2014.

[Osman 2016] Haidar Osman, Manuel Leuenberger, Mircea Lungu and Oscar Nierstrasz. *Tracking Null Checks in Open-Source Java Systems.* In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 304–313, 2016.

[Papoulias 2017] Nick Papoulias, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *End-User Abstractions for Meta-Control: Reifying the Reflectogram.* Science of Computer Programming, vol. 140, pages 2–16, 2017.

[Parnin 2011] Chris Parnin and Alessandro Orso. *Are automated debugging techniques actually helping programmers?* In Proceedings of the 2011 international symposium on software testing and analysis, pages 199–209, 2011.

[Pasquier 2023] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux and Loïc Lagadec. *Temporal breakpoints for multiverse debugging.* In Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, pages 125–137, 2023.

[Perscheid 2017] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel and Robert Hirschfeld. *Studying the advancement in debugging practice of professional software developers.* Software Quality Journal, vol. 25, no. 1, pages 83–110, 2017.

[Petrillo 2019] Fabio Petrillo, Yann-Gaël Guéhéneuc, Marcelo Pimenta, Carla Dal Sasso Freitas and Foutse Khomh. *Swarm debugging: The collective intelligence on interactive debugging.* Journal of Systems and Software, vol. 153, pages 152–174, 2019.

[Phang 2013] K. Y. Phang, J. S. Foster and M. Hicks. *Expositor: Scriptable time-travel debugging with first-class traces.* In 2013 35th International Conference on Software Engineering (ICSE), pages 352–361, may 2013.

[Pothier 2007] Guillaume Pothier, Éric Tanter and José Piquer. *Scalable omniscient debugging.* ACM SIGPLAN Notices, vol. 42, no. 10, pages 535–552, 2007.

[Pothier 2009] Guillaume Pothier and Éric Tanter. *Back to the future: Omniscient debugging.* IEEE software, vol. 26, no. 6, pages 78–85, 2009.

[Rahman 2023] Akond Rahman, Dibyendu Brinto Bose, Farhat Lamia Barsha and Rahul Pandita. *Defect Categorization in Compilers: A Multi-vocal Literature Review.* ACM Computing Surveys, vol. 56, no. 4, pages 1–42, 2023.

[Raymond 1996] Eric S Raymond and Guy L Steele. The new hacker's dictionary. Mit Press, 1996.

[Rein 2023] Patrick Rein, Tom Beckmann, Eva Krebs, Toni Mattis and Robert Hirschfeld. *Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools.* In 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), pages 254–265, 2023.

[Ressia 2012a] Jorge Ressia. *Object-Centric Reflection.* PhD thesis, Institut fur Informatik und angewandte Mathematik, 2012.

[Ressia 2012b] Jorge Ressia, Alexandre Bergel and Oscar Nierstrasz. *Object-Centric Debugging.* In Proceeding of the 34rd international conference on Software engineering, ICSE '12, 2012.

[Rochefort-Maranda 2021] Guillaume Rochefort-Maranda. *Inflated effect sizes and underpowered tests: How the severity measure of evidence is affected by the winner's curse.* Philosophical Studies, vol. 178, pages 133–145, 2021.

[Sánchez 2020] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo and Sergio Segura. *TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs.* IEEE Access, vol. 8, pages 107214–107228, 2020.

[Sellke 2001] Thomas Sellke, María Jesús Bayarri and James O Berger. *Calibration of ρ values for testing precise null hypotheses.* The American Statistician, vol. 55, no. 1, pages 62–71, 2001.

[Seltman 2015] Howard J Seltman. *Experimental design and analysis.* Retrieved from, 2015.

[Sillito 2006] J. Sillito, G.C. Murphy and K. De Volder. *Questions Programmers Ask During Software Evolution Tasks.* In Proceedings of International Symposium on Foundations on Software Engineering (FSE), pages 23–34. ACM, 2006.

[Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task.* IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 434–451, jul 2008.

[Spinellis 2018] Diomidis Spinellis. *Modern debugging: The art of finding a needle in a haystack.* Communications of the ACM, vol. 61, no. 11, pages 124–134, 2018.

[Steven Costiou 2021] Valentin Bourcier Steven Costiou Maximilian Willembrinck. *Pharo experiment wizard*, 2021. `https://github.com/Pharo-XP-Tools/Phex` (Accessed: 2024-08-01).

[Storey 2000] M-AD Storey, Kenny Wong and Hausi A Müller. *How do program understanding tools affect how programmers understand programs?* Science of Computer Programming, vol. 36, no. 2-3, pages 183–207, 2000.

[Tan 2014] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou and Chengxiang Zhai. *Bug characteristics in open source software.* Empirical software engineering, vol. 19, pages 1665–1705, 2014.

[Tassey 2002] Gregory Tassey. *The economic impacts of inadequate infrastructure for software testing.* National Institute of Standards and Technology, 2002.

[Tchamgoue 2011] Guy Martin Tchamgoue, Ok-Kyoon Ha, Kyong-Hoon Kim and Yong-Kee Jun. *A Taxonomy of Concurrency Bugs in Event-Driven Programs.* In Tai-hoon Kim, Hojjat Adeli, Haeng-kon Kim, Heau-jo Kang, Kyung Jung Kim, Akingbehin Kiumi and Byeong-Ho Kang, editeurs, Software Engineering, Business Continuity, and Education, pages 437–450, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Telles 2001] Matthew Telles and Yuan Hsieh. The science of debugging. Coriolis Group Books, 2001.

[Thiede 2023a] Christoph Thiede, Marcel Taeumel and Robert Hirschfeld. *Object-Centric Time-Travel Debugging: Exploring Traces of Objects.* In Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming, Programming '23, pages 54–60, New York, NY, USA, 2023. Association for Computing Machinery.

[Thiede 2023b] Christoph Thiede, Marcel Taeumel and Robert Hirschfeld. *Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging.* In Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2023, pages 89–102, New York, NY, USA, October 2023. Association for Computing Machinery.

[Vessey 1985] Iris Vessey. *Expertise in debugging computer programs: A process analysis.* International Journal of Man-Machine Studies, vol. 23, no. 5, pages 459–494, 1985.

[Vessey 1986] Iris Vessey. *Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols.* IEEE Transactions on Systems, Man, and Cybernetics, vol. 16, no. 5, pages 621–637, 1986.

[Wang 2014] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta and Iulian Neamtiu. *Drdebug: Deterministic replay based cyclic debugging with dynamic slicing.* In Proceedings of annual IEEE/ACM international symposium on code generation and optimization, page 98. ACM, 2014.

[Wei 2021] Ying Wei, Xiaobing Sun, Lili Bo, Sicong Cao, Xin Xia and Bin Li. *A comprehensive study on security bug characteristics.* Journal of Software: Evolution and Process, vol. 33, no. 10, page e2376, 2021.

[Wiedenbeck 1999] SUSAN Wiedenbeck and VENNILA RAMALINGAM. *Novice comprehension of small programs written in the procedural and object-oriented styles.* International Journal of Human-Computer Studies, vol. 51, no. 1, pages 71–87, 1999.

[Willembrinck Santander 2023] Maximilian Ignacio Willembrinck Santander. *An interactive debugging approach based on time-traveling queries.* Theses, Université de Lille, November 2023.

[Willembrinck 2021] Maximilian Willembrinck, Steven Costiou, Anne Etien and Stéphane Ducasse. *Time-Traveling Debugging Queries: Faster Program Exploration.* In International Conference on Software Quality, Reliability, and Security, Hainan Island, China, December 2021.

[Willembrinck 2022a] Maximilian Willembrinck, Steven Costiou, Anne Etien and Stéphane Ducasse. *Time-Traveling Queries for Faster Debugging and Program Comprehension.* Journées Nationales du Génie de la Programmation et du Logiciel 2022, June 2022. Poster, `https://inria.hal.science/hal-03738585`.

[Willembrinck 2022b] Maximilian Willembrinck, Steven Costiou, Adrien Vanègue and Anne Etien. *Towards Object-Centric Time-Traveling Debuggers.* In International Workshop on Smalltalk Technologies (IWST 22), Novi Sad, Serbia, August 2022. ACM Digital Libraries.

[Winter 2023] Emily Winter, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, Vesna Nowack and John Woodward. *How do Developers Really Feel About Bug Fixing? Directions for Automatic Program Repair.* IEEE Transactions on Software Engineering, vol. 49, no. 4, pages 1823–1841, 2023.

[Yang 2011] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen and Binyu Zang. *ORDER: Object centRic DEterministic Replay for Java.* In 2011 USENIX Annual Technical Conference (USENIX ATC 11), 2011.

[Yang 2025] Stephanie Yang, Miles Baird, Eleanor O'Rourke, Karen Brennan and Bertrand Schneider. *Decoding debugging instruction: A systematic literature review of debugging interventions*. ACM Transactions on Computing Education, vol. 24, no. 4, pages 1–44, 2025.

[Yin 2011] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy and Lakshmi Bairavasundaram. *How do fixes become bugs?* In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 26–36, 2011.

[Yin 2013] Haihan Yin, Christoph Bockisch and Mehmet Aksit. *A pointcut language for setting advanced breakpoints*. In Proceedings of the 12th annual international conference on Aspect-oriented software development, pages 145–156, 2013.

[Yoon 1998] Byung-Do Yoon and O.N. Garcia. *Cognitive activities and support in debugging*. In Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems, pages 160–169, 1998.

[Zayour 2001] I. Zayour and T.C. Lethbridge. *Adoption of reverse engineering tools: a cognitive perspective and methodology*. In Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, pages 245–255, 2001.

[Zeller 2009] Andreas Zeller. Why programs fail: a guide to systematic debugging. Elsevier, 2009.