



HAL
open science

Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs

Valentin Bourcier, Pooja Rani, Maximilian Ignacio Willembinck Santander,
Alberto Bacchelli, Steven Costiou

► To cite this version:

Valentin Bourcier, Pooja Rani, Maximilian Ignacio Willembinck Santander, Alberto Bacchelli, Steven Costiou. Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs. 2025. hal-04948470

HAL Id: hal-04948470

<https://hal.science/hal-04948470v1>

Preprint submitted on 14 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs

VALENTIN BOURCIER, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, France

POOJA RANI, University of Zurich, Switzerland

MAXIMILIAN WILLEMBRINCK, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, France

ALBERTO BACCHELLI, University of Zurich, Switzerland

STEVEN COSTIOU, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, France

Debugging consists in understanding the behavior of a program to identify and correct its defects. Breakpoints are the most commonly used debugging tool and aim to facilitate the debugging process by allowing developers to interrupt a program's execution at a source code location of their choice and inspect the state of the program.

Researchers suggest that in systems developed using object-oriented programming (OOP), traditional breakpoints may be a not effective method for debugging. In OOP, developers create code in classes, which at runtime are instantiated as object—entities with their own state and behavior that can interact with one another. Traditional breakpoints are set within the class code, halting execution for every object that shares that class's code. This leads to unnecessary interruptions for developers who are focused on monitoring the behavior of a specific object. As an answer to this challenge, researchers proposed *object-centric debugging*, an approach based on debugging tools that focus on objects rather than classes. In particular, using *object-centric breakpoints*, developers can select specific objects (rather than classes) for which the execution must be interrupted. Even though it seems reasonable that this approach may ease the debugging process by reducing the time and actions needed for debugging objects, no research has yet verified its actual impact.

To investigate the impact of object-centric breakpoints on the debugging process, we devised and conducted a controlled experiment with 81 developers who spent an average of 1 hour and 30 minutes each on the study. The experiment required participants to complete two debugging tasks using debugging tools with vs. without object-centric breakpoints. We found no significant effect from the use of object-centric breakpoints on the number of actions required to debug or the effectiveness in understanding or fixing the bug. However, for one of the two tasks, we measured a statistically significant reduction in debugging time for participants who used object-centric breakpoints, while for the other task, there was a statistically significant increase. Our analysis suggests that the impact of object-centric breakpoints varies depending on the context and the specific nature of the bug being addressed. In particular, our analysis indicates that object-centric breakpoints can speed up the process of locating the root cause of a bug when the bug can be replicated without needing to restart the program. We discuss the implications of these findings for debugging practices and future research.

Data and materials: <https://doi.org/10.5281/zenodo.14844897>

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Empirical software validation**; **Software testing and debugging**; **Object oriented development**; **Maintaining software**.

Additional Key Words and Phrases: Debugging, Debugging tools, Breakpoints, Object-oriented programming, Object-centric debugging, Object-centric breakpoints, Empirical evaluation, Controlled experiment

Authors' Contact Information: [Valentin Bourcier](mailto:valentin.bourcier@inria.fr), Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France, valentin.bourcier@inria.fr; [Pooja Rani](mailto:pooja.rani@ifi.uzh.ch), University of Zurich, Zurich, Switzerland, rani@ifi.uzh.ch; [Maximilian Willembreinck](mailto:maximilian-willembreinck-santander@inria.fr), Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France, maximilian-ignacio.willembreinck-santander@inria.fr; [Alberto Bacchelli](mailto:alberto.bacchelli@ifi.uzh.ch), University of Zurich, Zurich, Switzerland, bacchelli@ifi.uzh.ch; [Steven Costiou](mailto:steven.costiou@inria.fr), Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France, steven.costiou@inria.fr.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

1 Introduction

Debugging is known to be a tedious and time-consuming part of the development and software maintenance process [39, 55, 56, 59, 64]. It consists in reproducing the behavior of a program, understanding how it came to be and implementing the best possible solution to rectify failures [64].

Among the debugging tools provided by Integrated Development Environments (IDEs), breakpoints are the ones most commonly used by developers [2], particularly to initiate an analysis of a program's behavior [21]. 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 program execution system stops the program execution when it reaches a breakpoint and displays the execution context in a debugger. Usually, to present the execution context, debuggers display the stack of executed operations (e.g., 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 [2, 43, 44, 64].

Even though breakpoints are a widely used debugging tool [2], research suggests that they are not suitable for debugging object-oriented programs [36, 49]. 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 [32, 53], rather than about source code. Therefore, when debugging a single suspect object, developers may experience unwanted interruptions because 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 or reasoning about.

Consequently, Ressia *et al.* proposed *object-centric breakpoints* [49] 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. The approach of focusing the debugging perspective on objects has been coined *object-centric debugging* [36, 49]. Object-centric debuggers are expected to help analysis [35–37, 57, 58], understanding [61, 62], and debugging [14, 15, 18, 27, 49] of object-oriented programs. We quantify this from two angles: the debugging time, as they would provide faster ways to observe an object involved in a bug, and the number of actions with the debugger required to make that observation, which would decrease.

These works showcase innovative debuggers with interesting potential in improving the debugging of object-oriented programs. However, object-centric debuggers do not come with strong empirical evaluations but only with anecdotal evidence of their benefits. Therefore, we need to better understand the impact of object-centric debugging before engaging in significant research and engineering efforts to develop the approach.

In this paper, we present the first empirical evaluation of the impact of object-centric breakpoints on debugging. We devise and conduct a controlled experiment following a between-participants design with 81 developers, 54% of whom reported being professional software developers and 30% students. 85% of participants reported having more than three years of programming experience. Each participant is asked to complete two unrelated debugging tasks that we order randomly. The first task has to be completed using standard debugging tools (control) and the second using object-centric breakpoints in addition (treatment). We measured the tasks' correctness (i.e., participants

correctly explained/fixed the bugs), the number of debugging actions, and the time needed by our participants to debug each bug.

We found that, whenever a bug can be repeated without requiring the restart of the program, object-centric breakpoints seem to lower the time needed to find the root cause. However when the bug cannot be reproduced without a program restart, object-centric breakpoints seem to be inefficient and can even slow down the process of debugging. Additionally, we found no significant correlation between object-centric breakpoints and the number of actions required to debug, nor with the correct understanding of the bug. Overall, our investigation suggests that the effect of object-centric breakpoints depends on the nature of the observed bug. It confirms the potential of the approach, yet more research is needed to learn about the exact scenarios where object-centric debugging applies.

2 Background

In this section, we provide information on object-centric debugging and object-centric breakpoints—a central implementation of the approach. We highlight the differences between object-centric breakpoints and conditional breakpoints.

2.1 Object-Centric Debugging

Object-centric debugging is an approach aiming to address 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 [49]. 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 challenging using standard debugging tools, potentially demanding a greater number of actions [49] or leading to program crashes [27]. Some of these solutions enhance standard debuggers with object-centric operations such as breakpoints [14, 17, 27, 49], run-time behavioral adaptation [15, 42, 48], and support for object-centric operators [18]. Others enhance back-in-time and time-travel debuggers with new object-centric views and operators [8, 9, 35–37, 57, 58, 61–63]. Yet only Willebrink *et al.* includes an empirical evaluation related to object-centric debugging [61]. They set to investigate a time-travel debugger using 14 program comprehension tasks; two of the tasks require to use the object-centric options of the time-travel back end. Even though their evaluation shows significant results when considering all tasks, no conclusions could be drawn about the two object-centric tasks in isolation. Our work focuses on the evaluation of object-centric breakpoints as they represent the most researched object-centric debugging solution with a stable implementation.

2.2 Object-centric Breakpoints

Object-centric breakpoints [14, 17, 27, 49] provide operations and APIs to interrupt a program execution when an object specified by the developer gets called or modified. Implementing object-centric breakpoints is challenging, as it requires 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). To do so in the most efficient way (*i.e.*, without a condition to filter the targeted object), solutions such as *Aspect-Oriented Programming* (AOP) [31] and the setup of object Proxies [16] must be employed. It requires virtual-machine support to exchange the target object references with a proxy. Such a switch is possible in Python [45] and in Smalltalk variants, such as Squeak [5] or Pharo [6] thanks to their metaprogramming capabilities. We focus on the Pharo IDE [6] for

the presentation of breakpoints and the realization of our contribution, since it is the only object-oriented programming environment with object-centric breakpoints in its production version, since 2019.

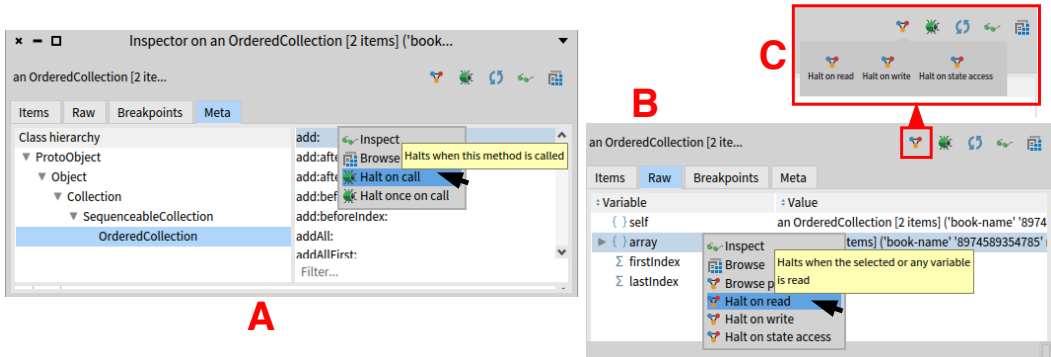


Fig. 1. The Pharo *Inspector* opened on a collection object, showing the integration of different object-centric breakpoints. Developers can install: (A) object-centric method breakpoints on the objects' methods, (B) object-centric field breakpoints on the objects' variables, and, from the top menu, (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 content of the instance variables, Figure 1 B) and the methods it can execute (Figure 1 A). The inspector is embedded in the debugger and displays the values of variables in use within the currently interrupted method. Developers can place the following object-centric breakpoints on the inspected objects:

Method-specific breakpoint. This breakpoint interrupts the execution before the target object executes a specific method. Ressler *et al.* [49] 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 interrupting 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 1 A) of the *Meta* pane.

Field breakpoint. This breakpoint interrupts 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 [49]. Developers install an object-centric field breakpoint through the contextual menu of an instance variable, by selecting *Halt on read/write* (Figure 1 B) of the *Raw* pane. The *Halt on state access* (Figure 1 B) interrupts 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 interrupts 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 1 C).

2.3 Object-centric Breakpoints Versus Conditional Breakpoints

Common competing arguments against the use and implementation of object-centric breakpoints is that most IDEs provide *conditional breakpoints* that can achieve the same feature. Conditional breakpoints offer the possibility to define conditions under which breakpoints interrupt execution. Developers can write conditions that check for the identity of the executing object to decide if

the breakpoint should interrupt execution or not, thus implementing an object-centric breakpoint. However, writing conditionals is done rarely [2], can be difficult [2], error-prone [19] and can induce undesirable performance overheads [2, 19], especially if lots of instances execute the condition just to filter out one single object. Additionally, the system must provide a reliable way to refer to objects, such as a reference or an identifier (*e.g.*, a hash), which they must use to write an identity check (*e.g.*, `hash == my_object_hash`). Overall, object-centric breakpoints are expected to be more efficient and to require less debugging code (*i.e.*, conditionals) from developers when tracking object interactions and modifications.

3 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.

3.1 Research Questions

We structured our investigation around three research questions.

RQ₁. 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 [49], we formulate the following null hypothesis:

H₀₁ object-centric breakpoints do not affect the ability of developers to fix bugs.

RQ₂. How do object-centric breakpoints affect the number of debugging actions?

Current evaluation scenarios [49] 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:

H₀₂ Object-centric breakpoints do not affect the number of debugging actions developers perform to fix bugs.

RQ₃. 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:

H₀₃ Object-centric breakpoints do not affect the time developers take to fix bugs.

3.2 Experiment Flow

To study our hypotheses, we conducted an empirical experiment with 81 participants. Figure 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 [54], and material is available in the replication package.

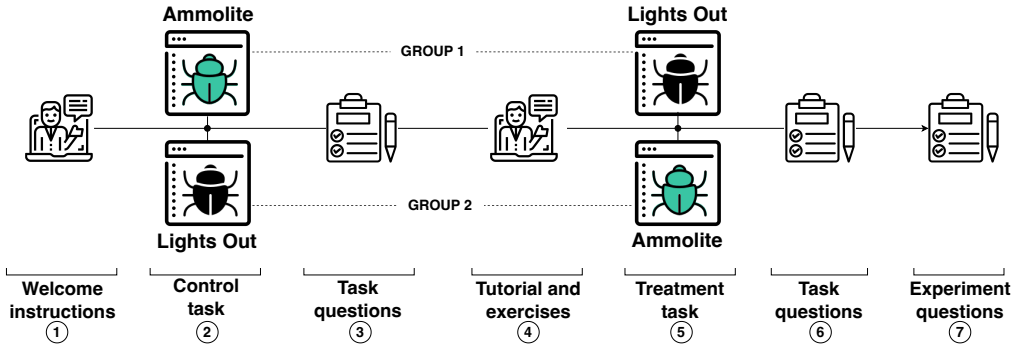


Fig. 2. Sequence of the seven steps followed by the experiment participants.

(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 environment (*i.e.*, the wizard), detail the steps and explain how to use the experiment wizard tool [54]. 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 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 subsection 2.2. 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 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.

3.3 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 2 steps 2 and 5).

Although this resembles a within-participants design [12], 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 [47]. 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 [12] where independent measures (control and treatment) are then compared per task.

3.4 Experimental Tasks

We designed the control and treatment tasks to mirror the object-centric breakpoints scenarios [49] described in subsection 2.2. 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.

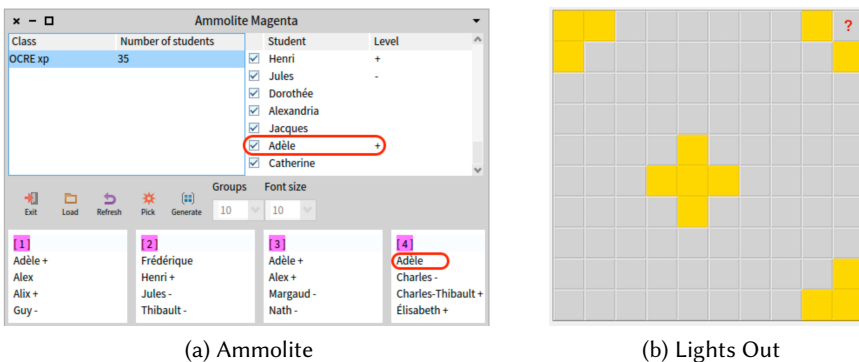


Fig. 3. Ammolite (a) and Lights Out (b) applications with the bug symptom highlighted in red.

Object-centric debugging assumes that developers can access the problematic object within the program execution [18, 49]. 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](#) 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.

3.5 Experimental Variables

[Table 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 2](#)). To control interruptions, *i.e.*,

Table 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.

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.

3.6 Data Collection, Selection, and Correction

Participants used the Pharo 9 IDE [6] for the experiment. Our experimental framework instruments the IDE immediately after the *Welcome instructions* (Figure 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 [10], 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.

3.7 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 [20]. 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 [49] suggesting that object-centric breakpoints have a strong potential for facilitating debugging. For instance, in one scenario [49] 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 [27], massive debugger noise (unwanted interruptions) [18]). 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.

4 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 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.

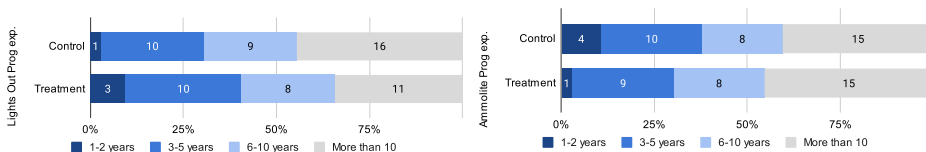


Fig. 4. Programming experience for Lights Out (left) and Ammolite (right) for each condition.

4.1 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) [52]. 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 [29], *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 [30]. 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 [22], we consider effect sizes to be small when $|R_{rb}| < 0.2$, medium when $0.2 < |R_{rb}| < 0.3$ and large when $|R_{rb}| > 0.3$.

4.2 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 $\mathbf{H0}_1$. However, the results include 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 $\mathbf{H0}_1$.*





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

¹We use the R library *effectsize* [3] to calculate R_{rb} .

4.3 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 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.

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

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





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 $H0_2$.

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

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

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

The results of the Mann-Whitney U test performed on the data confirm these observations (Table 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 H_0_3 , we take the decision to reject H_0_3 .

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

4.5 Participants' Perception

According to post-experiment feedback responses (Figure 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 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 respondents (Figure 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.”

4.6 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 4.3 and subsection 4.4).

Task difficulty. Participants perceived Lights Out as harder to debug than Ammolite (subsection 4.5) 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 4.5). 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.

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

Table 4. Results of the Mann-Whitney U tests for significant changes in the usage 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 time	0.020	4.781	0.302	Inspector time	0.005	13.211	-0.375
Browser time	0.006	11.520	0.356	Inspector active	0.009	8.763	-0.351
Debugger time	0.023	4.282	0.307	Spotter active	0.008	9.951	0.630
Wizard time	0.006	12.215	0.448				
Implementors active	0.019	4.798	0.377				
Senders time	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](#).

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

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 [4] 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 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.

5 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.* [49] expected. Indeed, we calibrated our statistical power analysis to determine the necessary sample sizes for detecting large effects (details in [subsection 3.7](#)). 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 Ammolite, but increased debug time for Lights Out. As mentioned in [subsection 4.6](#), 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 4.6](#)).

Research has shown that developers who are debugging spend in average 14% [2] and 16% [1] 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” [1]. 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 [1], 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 [7, 34], 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 [11], in Java [40, 41] and Javascript [25], for performance [51], security [60], and compilers [46]. 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.

6 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 [33]. 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 [49]. 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 [26]),

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 [23]). 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 [subsection 2.2](#). We provided five out of the eight breakpoints proposed by Ressia [49]. 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](#), 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](#), 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 4.5](#)). 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 [13]. Since the results yielded non-parametric distributions ([subsection 4.1](#)), we used the non-parametric Mann-Whitney U test for which we chose the rank-biserial correlation as an appropriate effect size measure [29]. 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 4.3 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 [24, 28, 38, 50]. 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.

7 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 [49], 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. 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.

8 Data Availability

All data, software, and scripts used for analysis are available in our replication package [10].

Acknowledgments

This work was funded by the ANR JCJC OCRE Project (<https://anr.fr/Project-ANR-21-CE25-0004>). We also gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project 200021_197227. We express our warmest thanks to Vincent Aranega, associate professor at University of Lille for the early discussion and exploration of object-centric breakpoints implementation. Finally, we would like to express our gratitude to Tristan Coignon, Ph.D. student at INRIA in the Spirals team, France, and Oscar Nierstrasz, researcher at Feenk, Switzerland, for their valuable time and expertise in reviewing our manuscript and providing helpful feedback prior to its submission.

References

- [1] Abdulaziz Alaboudi and Thomas D LaToza. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering* 28, 5 (2023), 117. <https://doi.org/10.1007/s10664-023-10352-5>
- [2] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *Proceedings of ICSE 18: 40th International Conference on Software Engineering*. <https://doi.org/10.1145/3180155.3180175>
- [3] Mattan S. Ben-Shachar, Daniel Lüdtke, and Dominique Makowski. 2020. effectsize: Estimation of Effect Size Indices and Standardized Parameters. *Journal of Open Source Software* 5, 56 (2020), 2815. <https://doi.org/10.21105/joss.02815>
- [4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)* 57, 1 (1995), 289–300. <https://doi.org/10.1111/j.2517-6161.1995.tb02031.x>
- [5] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2007. *Squeak by Example*. Square Bracket Associates. <https://github.com/SquareBracketAssociates/SqueakByExample-english>

- [6] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. <http://books.pharo.org>
- [7] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of Foundations of Software Engineering ESEC/FSE*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 117–128. <https://doi.org/10.1145/3106237.3106255>
- [8] Valentin Bourcier and Steven Costiou. 2024. 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. <https://doi.org/10.1109/SANER60148.2024.00040>
- [9] Valentin Bourcier, Steven Costiou, Maximilian Ignacio Willembriñck Santander, Adrien Vanègue, and Anne Etien. 2024. Time-traveling object-centric breakpoints. *Journal of Computer Languages* (2024). <https://doi.org/10.1016/j.cola.2024.101285>
- [10] Valentin Bourcier, Pooja Rani, Maximilian Ignacio Willembriñck Santander, Alberto Bacchelli, and Steven Costiou. 2025. Replication Package for "Empirically Evaluating the Impact of Object-Centric Breakpoints on the Debugging of Object-Oriented Programs". <https://doi.org/10.5281/zenodo.14844897>
- [11] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181. <https://doi.org/10.1016/j.jss.2019.03.002>
- [12] Larry B. Christensen, R. Burke Johnson, and Lisa A. Turner. 2015. *Research Methods, Design, and Analysis; 12th Edition, Global Edition*. Pearson, Boston, Mass y 2015.
- [13] Jacob Cohen. 1988. *Statistical power analysis for the behavioral sciences*. routledge. <https://doi.org/10.4324/9780203771587>
- [14] Claudio Corrodi. 2016. Towards Efficient Object-Centric Debugging with Declarative Breakpoints. In *SATToSE 2016*. <https://doi.org/10.7892/boris.94640>
- [15] Steven Costiou. 2018. *Unanticipated behavior adaptation : application to the debugging of running programs*. Ph.D. Dissertation. Université de Bretagne occidentale - Brest.
- [16] Steven Costiou, Vincent Aranega, and Marcus Denker. 2020. Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020). <https://doi.org/10.22152/programming-journal.org/2020/4/5>
- [17] Steven Costiou, Vincent Aranega, and Marcus Denker. 2022. Reflection as a Tool to Debug Objects. In *International Conference on Software Language Engineering (SLE)*. 55–60. <https://doi.org/10.1145/3567512.3567517>
- [18] Steven Costiou, Mickaël Kerboeuf, Clotilde Toullec, Alain Plantec, and Stéphane Ducasse. 2020. Object Miners: Acquire, Capture and Replay Objects to Track Elusive Bugs. *The Journal of Object Technology* 19 (July 2020), 1:1–32. <https://doi.org/10.5381/jot.2020.19.1.a1>
- [19] Kostadin Damevski, David C Shepherd, Johannes Schneider, and Lori Pollock. 2016. Mining sequences of developer interactions in visual studio for usage smells. *IEEE Transactions on Software Engineering* 43, 4 (2016). <https://doi.org/10.1109/TSE.2016.2592905>
- [20] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. 2009. Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses. *Behavior Research Methods* 41, 4 (Nov. 2009), 1149–1160. <https://doi.org/10.3758/brm.41.4.1149>
- [21] Eduardo Andreetta Fontana and Fabio Petrillo. 2021. Mapping breakpoint types: an exploratory study. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. <https://doi.org/10.1109/QRS54544.2021.00110>
- [22] David C Funder and Daniel J Ozer. 2019. Evaluating effect size in psychological research: Sense and nonsense. *Advances in methods and practices in psychological science* 2, 2 (2019), 156–168. <https://doi.org/10.1177/2515245919847202>
- [23] Adrian Furnham. 1986. Response bias, social desirability and dissimulation. *Personality and individual differences* 7, 3 (1986), 385–400. [https://doi.org/10.1016/0191-8869\(86\)90014-0](https://doi.org/10.1016/0191-8869(86)90014-0)
- [24] Andrew Gelman and John Carlin. 2014. Beyond power calculations: Assessing type S (sign) and type M (magnitude) errors. *Perspectives on psychological science* 9, 6 (2014), 641–651. <https://doi.org/10.1177/1745691614551642>
- [25] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2021. BUGSJS: a benchmark and taxonomy of JavaScript bugs. *Software Testing, Verification and Reliability* 31, 4 (2021), e1751. <https://doi.org/10.1002/stvr.1751>
- [26] James J Heckman. 1990. Selection bias and self-selection. In *Econometrics*. Springer, 201–224. https://doi.org/10.1007/978-1-349-20570-7_29
- [27] Bob Hinkle, Vicki Jones, and Ralph E Johnson. 1993. Debugging objects. In *The Smalltalk Report*.
- [28] John PA Ioannidis. 2008. Why most discovered true associations are inflated. *Epidemiology* 19, 5 (2008), 640–648. <https://doi.org/10.1097/EDE.0b013e31818131e7>

- [29] 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. 2024. Guide to effect sizes and confidence intervals. <https://doi.org/10.17605/OSF.IO/D8C4G>
- [30] Dave S Kerby. 2014. The simple difference formula: An approach to teaching nonparametric correlation. *Comprehensive Psychology* 3 (2014), 11–17. <https://doi.org/10.2466/11.IT.3.1>
- [31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, Mehmet Aksit and Satoshi Matsuoka (Eds.). Springer-Verlag, 220–242. <https://doi.org/10.1007/BFb0053381>
- [32] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. 2014. 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 (Portland, Oregon, USA) (PLATEAU '14)*. ACM, New York, NY, USA, 1–11. <https://doi.org/10.1145/2688204.2688212>
- [33] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. ACM, New York, NY, USA, 1090–1101. <https://doi.org/10.1145/3180155.3180200>
- [34] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449. <https://doi.org/10.1007/s10664-020-09810-1>
- [35] Adrian Lienhard, Stéphane Ducasse, and Tudor Girba. 2007. Object Flow Analysis – Taking an Object-Centric View on Dynamic Analysis. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07)* (Lugano, Switzerland). ACM Digital Library, New York, NY, USA, 121–140. <https://doi.org/10.1145/1352678.1352686>
- [36] Adrian Lienhard, Stéphane Ducasse, Tudor Girba, and Oscar Nierstrasz. 2006. Capturing How Objects Flow At Runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*. 39–43.
- [37] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. 2009. Flow-Centric, Back-In-Time Debugging. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009 (LNBP, Vol. 33)*. Springer-Verlag, 272–288. https://doi.org/10.1007/978-3-642-02571-6_16
- [38] Jiannan Lu, Yixuan Qiu, and Alex Deng. 2019. A note on Type S/M errors in hypothesis testing. *Brit. J. Math. Statist. Psych.* 72, 1 (2019), 1–17. <https://doi.org/10.1111/bmsp.12132>
- [39] Devon H O'Dell. 2017. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue* 15, 1 (2017), 71–90. <https://doi.org/10.1145/3055301.3068754>
- [40] Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. 2016. Tracking Null Checks in Open-Source Java Systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 304–313. <https://doi.org/10.1109/SANER.2016.57>
- [41] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. 2014. Mining frequent bug-fix code changes. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 343–347. <https://doi.org/10.1109/CSMR-WCRE.2014.6747191>
- [42] Nick Papoulias, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. 2017. End-User Abstractions for Meta-Control: Reifying the Reflectogram. *Science of Computer Programming* 140 (2017), 2–16. <https://doi.org/10.1016/j.scico.2016.12.002>
- [43] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110. <https://doi.org/10.1007/s11219-015-9294-2>
- [44] Fabio Petrillo, Yann-Gaël Guéhéneuc, Marcelo Pimenta, Carla Dal Sasso Freitas, and Foutse Khomh. 2019. Swarm debugging: The collective intelligence on interactive debugging. *Journal of Systems and Software* 153 (2019), 152–174. <https://doi.org/10.1016/j.jss.2019.04.028>
- [45] Pyjack 0.3.2. 2011. *pythonhosted.org* (Accessed: 2024-08-01). <https://pythonhosted.org/pyjack/>
- [46] Akond Rahman, Dibendu Brinto Bose, Farhat Lamia Barsha, and Rahul Pandita. 2023. Defect Categorization in Compilers: A Multi-vocal Literature Review. *Comput. Surveys* 56, 4 (2023), 1–42. <https://doi.org/10.1145/3626313>
- [47] Patrick Rein, Tom Beckmann, Eva Krebs, Toni Mattis, and Robert Hirschfeld. 2023. 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)*. 254–265. <https://doi.org/10.1109/ICPC58990.2023.00040>
- [48] Jorge Ressia. 2012. *Object-Centric Reflection*. Ph. D. Dissertation. Institut für Informatik und angewandte Mathematik.
- [49] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-Centric Debugging. In *Proceeding of the 34th international conference on Software engineering (Zurich, Switzerland) (ICSE '12)*. <https://doi.org/10.1109/ICSE.2012.6227167>
- [50] Guillaume Rochefort-Maranda. 2021. Inflated effect sizes and underpowered tests: How the severity measure of evidence is affected by the winner's curse. *Philosophical Studies* 178 (2021), 133–145. <https://doi.org/10.1007/s11098-020-01424-z>

- [51] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2020. TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs. *IEEE Access* 8 (2020), 107214–107228. <https://doi.org/10.1109/ACCESS.2020.3000928>
- [52] Thomas Sellke, María Jesús Bayarri, and James O Berger. 2001. Calibration of p values for testing precise null hypotheses. *The American Statistician* 55, 1 (2001), 62–71. <https://doi.org/10.1198/000313001300339950>
- [53] J. Sillito, G.C. Murphy, and K. De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of International Symposium on Foundations on Software Engineering (FSE)*. ACM, 23–34. <https://doi.org/10.1145/1181775.1181779>
- [54] Valentin Bourcier Steven Costiou, Maximilian Willembinck. 2021. *Pharo experiment wizard*. <https://github.com/Pharo-XP-Tools/Phex>
- [55] Gregory Tassej. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002).
- [56] Matthew Telles and Yuan Hsieh. 2001. *The science of debugging*. Coriolis Group Books.
- [57] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. 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 (Tokyo, Japan) (Programming '23)*. Association for Computing Machinery, New York, NY, USA, 54–60. <https://doi.org/10.1145/3594671.3594678>
- [58] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. 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)*. Association for Computing Machinery, New York, NY, USA, 89–102. <https://doi.org/10.1145/3622758.3622892>
- [59] Iris Vessey. 1986. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics* 16, 5 (1986), 621–637. <https://doi.org/10.1109/TSMC.1986.289308>
- [60] Ying Wei, Xiaobing Sun, Lili Bo, Sicong Cao, Xin Xia, and Bin Li. 2021. A comprehensive study on security bug characteristics. *Journal of Software: Evolution and Process* 33, 10 (2021), e2376. <https://doi.org/10.1002/smr.2376>
- [61] Maximilian Willembinck, Steven Costiou, Anne Etien, and Stéphane Ducasse. 2021. Time-Traveling Debugging Queries: Faster Program Exploration. In *International Conference on Software Quality, Reliability, and Security*. Hainan Island, China. <https://inria.hal.science/hal-03463047>
- [62] Maximilian Willembinck, Steven Costiou, Adrien Vanègue, and Anne Etien. 2022. Towards Object-Centric Time-Traveling Debuggers. In *International Workshop on Smalltalk Technologies (IWST 22)*. ACM Digital Libraries, Novi Sad, Serbia. <https://inria.hal.science/hal-03825736>
- [63] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. 2011. ORDER: Object centRiC DEterministic Replay for Java. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.
- [64] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.