

Debugging Activity Blueprint

1st Valentin Bourcier

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL
Lille, France
valentin.bourcier@inria.fr

2nd Alexandre Bergel

RelationalAI
Bern, Switzerland
alexandre.bergel@me.com

3rd Anne Etien

Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRISTAL
Lille, France
anne.etien@univ-lille.fr

4th Steven Costiou

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL
Lille, France
steven.costiou@inria.fr

Abstract—Empirically analyzing debugging activity is notoriously difficult. In particular, aggregating data (obtained from either observation or event logging) to verify hypotheses on developers’ behavior is known to be challenging. Overall, the difficulty of studying debugging activities contributes to the need for more empirical evidence on how practitioners use debuggers.

We propose *debugging activity blueprint* as a visual tool to analyze and navigate through all the events recorded in a programming environment. Our blueprint is a polymetric view representing the interaction between debuggers and other programming tools. Our blueprint highlights the flow of a debugging activity across the tools an IDE offers. An exploratory use case over three participants and two debugging tasks indicates that our blueprint supports a fine-grained analysis of complex debugging scenarios.

Index Terms—Debugging, Debugging behavior, IDE, visualization, polymetric views

I. INTRODUCTION

Debugging is a crucial activity in software development that involves identifying, analyzing, and removing bugs from a software system. Despite its relevance, in today’s programming and software development environments, we still need to understand how practitioners use debuggers at a fine and coarse-grain level. Not being able to characterize the behavior of programmers when debugging may hamper the future development of debugging tools and methodologies.

For example, a debugging activity does not solely happen in a debugger. Previous research efforts [1], [2] show that practitioners navigate through the base source code of an application being debugged. However, we still need to understand the developer intent behind this phenomenon. Just as we need to provide answers to essential questions such as *What programming tools does a practitioner need when debugging?* or *What information is important to a practitioner that is not provided by a debugger?* and correlate them with developers’ intent.

Considered problem. The problem addressed in this paper is **how to characterize a debugging activity?** As far as we are aware of, no tools or methodologies proposed by the academic community allow to fully to grasp the intent behind

developers’ debugging actions. Our blueprint is a first step toward answering this question.

Contribution. This paper proposes *debugging activity blueprint*, a multi-scale polymetric view [3] to visualize a debugging activity conducted by a programmer. Our blueprint is built from a seamless event logging of the programming environment and provides visual support to analyze the behavior of the monitored debugging activity. Our blueprint is multi-scale because it helps observe behavior at a very fine grain (e.g., representing a sequence of debugging actions like *step into* and *step over*) and at a coarse grain (e.g., moving from one tool to another). Our blueprint is postmortem, meaning that our visualization is built once the debugging activity is deemed completed.

Evaluation. We have applied our blueprint to two different debugging tasks performed by three participants. We were able to fully understand and explain a complete session. We have identified a visual vocabulary from recurrent visual structures and new questions raised about debugging activities.

Outline. The paper is structured as follows: Section II provides the necessary background to our work. Section III describes our debugging activity blueprint. Section IV presents the case studies and the results we obtained by applying our blueprint. Section V presents the visual vocabulary our blueprint defines and uses this vocabulary in the case studies presented earlier. Section VI discusses relevant points of our approach. Section VII summarizes the threats to validity we identified. Section VIII discusses the work related to our effort. Section IX concludes and highlights our future work.

II. BACKGROUND: PHARO PROGRAMMING TOOLS

We applied our blueprint to debugging sessions monitored in the Pharo programming environment [4]. As such, it is relevant to give an overview of the programming tools commonly used with a debugger since our blueprint visualizes the interaction between these tools.

Inspector. An object in object-oriented programming consists of values following the structure specified by the class of the object. An inspector is a tool that shows the values of an

object’s variables, commonly referred as the dynamic state of that object. An object inspector allows a practitioner to navigate through the graphs of objects.

Code browser. In Pharo, the code browser is the main tool to view and edit source code. One can execute code snippets or unit tests from a code browser. A code browser also displays the breakpoints associated with a line of code.

Debugger. This is the main tool used by Pharo developers to explore and navigate the execution state of a program. The Pharo debugger is similar to traditional debuggers structured along a method call stack. It shows the current frame with the run-time values of its defined variables, and the source code associated to that current frame. In addition, Pharo’s debugger supports live methods edition, i.e., methods can be modified without requiring the running program to be restarted.

Queries. One can query the Pharo system to answer questions about the source code structure. Querying a simple method `printOn: signature` lists (i) the implementations of the method `printOn:` found in the system or (ii) existing methods that call `printOn:`.

Commonality between Pharo tools. Programming tools provided by Pharo have a number of commonalities. First, one can have multiple instances of each tool. It is common in a debugging activity to have *several debuggers opened at the same time*. Pharo does not impose any restriction on the number of instances of code browsers, inspectors, queries, or debuggers one can use at the same time.

Second, all the tools can be opened and used at any given time. For example, it is common for a practitioner to navigate or edit code using a code browser and debug a given code snippet (which has the effect of executing the snippet within a debugger).

Third, no programming tool dominates other tools, and all the tools are equally accessible through a keyboard shortcut and contextual menu.

Differences with VSCode and Eclipse. Properties of the Pharo tools contrast with tools offered by major programming environments, including VSCode and Eclipse. In these environments, the source code dominates other tools since they are almost always present as window tabs. One cannot open more than one debugger at a given time. The enforced code editing phase vs execution phase implies that one cannot arbitrarily open a debugger during code editing. Pharo does not enforce tool dominance or restrict code editing to a separate activity from program execution.

III. DEBUGGING ACTIVITY BLUEPRINT

The debugging activity blueprint is a visualization that shows the tools used by a developer during a debugging task. Our blueprint displays the relationships between the tools and the activities performed within these tools.

A. In a nutshell

The *debugging activity blueprint* is a post-mortem visualization representing a debugging activity exercised by developers

in the Pharo development environment (or Pharo IDE). We refer to *debugging activity* as a portion of a programming session in which a developer mostly concentrates on fixing or understanding software behavior.

When debugging, every action performed by developers in the Pharo IDE is recorded as a log. Once the debugging session is deemed finished by the programmer, our visualization uses these logs to represent the activities of developers in the debugging tools from the IDE. Tool activities and their interactions are represented in a structured fashion, as illustrated by Figure 1.

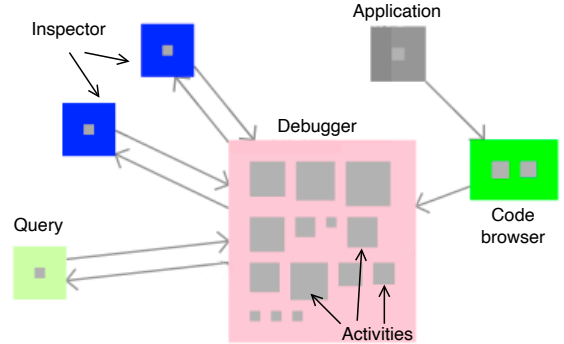


Fig. 1: *Debugging activity blueprint*: tool activities and interactions.

Representing Pharo tools. Our blueprint represents a Pharo tool as a colored box. Figure 1 shows the different kinds of Pharo tools available displayed by the blueprint with their interactions. The color of the boxes indicates the kind of tool involved in the activity. The blueprint uses the following arbitrary color encoding: **Debugger** (pink), **Inspector** (blue), **Query** (green), **Browser** (green), **Application** (grey) (which corresponds to the application being debugged).

Flow between tools. An arrow $T \rightarrow S$ between two boxes T and S indicates at least once transition that occurred between the tool T to the tool S . This transition means that the developer used T during the debugging activity, and then switched to the tool S . This transition may happen several times during the debugging activity.

Note that our tool does not represent the liveness of the Pharo tools. A transition $T \rightarrow S$ does not indicate that T was removed from the IDE or S was opened. Instead, it simply means that the user changed their focus to another tool. Tool opening or closing is not represented in the blueprint.


Since each of the inspectors and the query forms a cycle with the debugger, we deduce that the programmer temporarily moved away from the debugger.

Interaction block. Within a given tool, developers typically perform several actions before switching to a different tool. We refer to *interaction block* as an uninterrupted sequence of actions performed in the same tool. A sequence of actions interrupted by switching programming tools results in two interaction blocks. An interaction block is represented as a

smaller gray box located in a box. The size of an interaction block reflects the number of actions performed by developers. Many actions can be performed in the debugger, including inspecting variables' values, navigating in the method call stack, adding new breakpoints, and stepping to the next instruction.

Figure 1 shows that all tools have one or several interaction blocks. Interaction blocks within a tool are read from left to right and top to bottom, as in most Indo-European natural languages. Figure 1 reveals that the programmer performed many actions in the debuggers since the debugger contains 14 blocks, which means that the programmers move away from and back to the debugger 13 times. Some debugger's blocks are large, indicating the programmer did many actions without leaving the debugger (e.g., `step into` or `step over`). The query and inspectors have a small block each, indicating that the programmer did not do much with these tools.

B. Interactions

An interaction block is a linear sequence of actions. If at least one debugging action related to the control flow or program counter (i.e., `step-over`, `step-into`, `add a breakpoint`, etc.) is among an interaction block's actions, then the interaction block has a thin black border (i.e., .

When moving the mouse over a user interaction, such as in Figure 2, a tooltip appears showing the duration of the activity and the debugging actions performed during this interaction. We encode debugging actions into symbols to indicate the actions performed during an interaction block. Table I shows the Pharo debugging actions with their associated symbols.

TABLE I: Symbols for Debugging Actions and breakpoints.

Action	Symbol	Breakpoints	Symbol
Over	>	Add	+b
Into	∨	Hit	*b
Proceed	P	Remove	-b

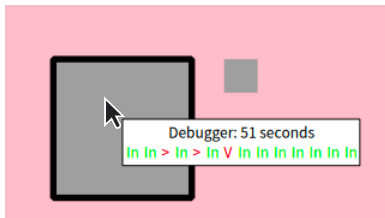


Fig. 2: Interactions.

The `In` symbol (see Figure 2) corresponds to inspections of code, execution contexts, objects, etc. We do not consider inspections as debugging actions but as exploration actions. Within the same tool, developers may execute different exploration actions in different subtools. It is therefore common to have successive basic interactions without leaving the tool. A classical example is the inspection of an object: developers may dive into the object's graph from its instance variables, which may open sub-inspections or trees of properties. This kind

of tool's inner interactions spawns different basic interactions within a tool visualization.

Temporality is not explicitly represented in the blueprint. Instead, interactions offered by the blueprint are key to revealing the different order of sequentiality of events. We visualize temporality using color highlighting. When pointing the mouse over an interaction, the visualization highlights the previous interaction in orange and the next interaction in red. We can see an example in Figure 3. The mouse is moved over the fifth square of the debugger (in pink). The developer was previously in the orange interaction in the same debugger, before arriving at the current interaction. A tooltip appears over the current interaction and shows the sequence of actions performed by the developer. The developer inspected (`In`) three elements of the debugger, then added a breakpoint (`+b`), inspected two elements (`In`), resumed the execution (`P`) which hit a breakpoint (`*b`). Just after the breakpoint hit, the developer moved to the red interaction in the code browser at the left of the debugger. We then interpret that the cause of moving to the code browser was the breakpoint hit.

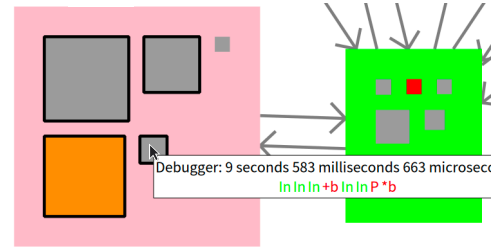


Fig. 3: Temporality.

C. Exploration and annotation of blueprints

To enhance and archive the blueprint understanding, we complement the visualization with fine-grained logs and annotations associated with tools and their interactions.

Access to the logging model. When generating a blueprint from logs, we maintain a mapping between the visual element (e.g., an interaction) and the logs from which it was materialized. Combined with the visual elements, these logs enable fine-grained comprehension of the actions performed by developers.

For example, in Figure 3 we know from the blueprint that the developer installed a breakpoint. When we select the `+b` symbol in the tooltip, an inspector opens on the log from which that visual element was materialized. In Figure 4, we can observe an excerpt of the inspected breakpoint installed in Figure 3. We then obtain the knowledge that the developer put a breakpoint on the method `personName` of a class `OCDPerson`, and more specifically on the line that returns the person's name (the `node` element in Figure 4).

```
▶ | method OCDPerson>>#personName
▶ | node   personName ^personName
```

Fig. 4: Excerpt of a breakpoint log example.

Annotations. To save the knowledge acquired when exploring a blueprint, each tool (i.e., color boxes) and each interaction block (i.e., inner grey boxes) can be annotated with texts that persist in the blueprint model. To do so, we need to select a tool or a tool interaction and add an annotation in the pane that opens (Figure 5).

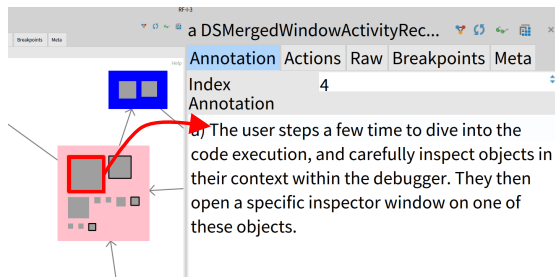


Fig. 5: An interaction (in red) with an annotation.

IV. EXPERIMENTAL DESIGN AND RESULTS

We performed an exploratory experiment with three participants to study how the debugging activity blueprint can help in understanding how developers debug. We simulate a between-participant experiment for the experimentation of a debugger extension (namely, an object-centric debugger [5]) by giving developers real debugging tasks, one for which they do not use the debugger extension and the others for which they use it. The goal of the experiment is not to study the impact of the object-centric debugger but to explore the visualization’s potential for understanding debugging activities and especially the impact a debugger may have.

Study settings. We asked three participants of different profiles and experiences (see Table II) to perform a set of debugging tasks in Pharo. The tasks are composed of one control task and one treatment task. In both tasks, we ask participants to solve a bug. During the control task, participants use the standard debugging tools available in Pharo. During the treatment task, participants use an object-centric debugger [5].

We configured the environment to import the debugging tasks and record participants’ actions required to generate the visualization. Participants never leave the Pharo IDE as the experimental framework with the task descriptions is included in Pharo. Participants record their screens but do not follow any protocol to comment on what they are doing, e.g., they do not think aloud.

Participants. Table II details the participant’s profiles. Participants are one associate professor with a 10-year-long industry experience in software development (SD), one computer science PhD student with 4 years of SD experience, and one engineering intern with 4 years of SD experience. All participants were either familiar with or already heard of the additional tool used in the treatment task. All participants are proficient with Pharo tools and are Pharo practitioners, in particular, they are used to debug with Pharo. In the following, participants are referred to as *users*.

TABLE II: Participants: three Pharo practitioners.

Alias	Soft. dev. experience	Position
User-A	4 years	Engineering intern
User-I	4 years	PhD student
User-V	> 10 years	Associate professor

Debugging tools. Participants used two kinds of tools during the experiment. During control tasks, they use the standard tools available in Pharo and described in Section II. During treatment tasks, participants use the standard tools complemented by an object-centric debugger. This debugger provides breakpoints that automatically scope to specific objects without the need to write conditionals.

Tasks. The tasks assigned to participants are named *Atom* and *Reflectivity*. The *Atom* task consists in solving a bug in a small graphical application composed of colored squares named atoms. The *Reflectivity* task consists in fixing a unit test of *Reflectivity* [6], the reflective layer of Pharo. For each task, participants have to provide a fix and an explanation for the bug.

Tasks are randomly assigned to users in one of the two sequences detailed in Table III. The first tasks (1) are always used as control, while the other task (2) are always using the treatment (i.e., the object-centric debugger).

TABLE III: Tasks sequences.

Sequence	Task 1	Task 2
1	Atom	Reflectivity
2	Reflectivity	Atom

Results. Table IV shows for each task the time taken by participants to complete the task and if participants fixed the task’s bug. All participants fixed the two bugs. On the measured times we can observe the following:

- User-A (novice) spent as much time solving the control task (*Atom*) and the treatment task (*Reflectivity*).
- User-I (novice) spent about twice the time on the control task (*Atom*) compared to the treatment task (*Reflectivity*).
- User-I (novice) spent about the same time solving the *Reflectivity* task as treatment as User-V (expert) did for the same task as control.
- User-V (expert) spent about more time solving the treatment task (*Atom*) than the control task (*Reflectivity*).

These observations, if repeated with many participants, might provide statistical evidence that the treatment tool has or has not an effect on the debugging efficiency of the participants. However, such quantitative evaluation cannot help us to understand the mechanisms at play in the observed effects. In the next section, we use the Debugging Activity Blueprint to study the participants’ debugging behavior during these tasks.

TABLE IV: Results of the experiment for each task, by user. Data is computed from the logs. The times are displayed in minutes with seconds (i.e., 1'15 = one minute and fifteen seconds). (C) = control task, (T) = treatment task.

Alias	Tasks (in order)	Time (in min)	Bug fixed
User-A	(C) Atom	36'49	yes
	(T) Reflectivity	37'08	yes
User-I	(C) Atom	43'58	yes
	(T) Reflectivity	20'47	yes
User-V	(C) Reflectivity	20'54	yes
	(T) Atom	33'41	yes

V. VISUALIZING DEBUGGING ACTIVITIES

In this section, we study the blueprints of the debugging activities extracted from the participations to our experiment. Figures 6, 7 and 8 respectively show the debugging activity blueprints of participants User-A, User-I, and User-V for their control and treatment tasks. These figures have been manually annotated. We report notable observable instances of similar tool interactions that we categorize into a visual vocabulary. We use this vocabulary to do a summary analysis of each participant’s debugging session. We discover that our vocabulary could become patterns of debugging that could be used for systematic analyses of debugging activity blueprints. Finally, we analyze in detail a specific blueprint and observe that our understanding of the debugging session from the blueprint matches what happens in the corresponding video.

A. Visual Vocabulary

In this section, we define the debugging activity blueprint vocabulary from what we observe in the visualizations from Figures 6, 7 and 8. We observe three kinds of recurring notable instances of tool interactions and navigation: *chains*, *hubs*, and *ping-pongs*.

Chain. A chain is a single sequential flow going through three or more tools of any kind (including start and stop). The flow is unidirectional, starts from any kind of tool, and may return or not to its starting point. For example, a long chain of debuggers, manually annotated C2, can be seen on the right side part of Figure 7.

Hub. A hub is a window with a notable concentration of activities, from which other tools are navigated back and forth. Visually, a hub appears central to the debugging activity and catches the eye. The window is bigger and contains more activities than most of the other navigated tools. For example, two hubs can be seen in the left side task of Figure 8: a debugger (pink) and a code browser (green) marked with h1 and h2.

Ping-pong. A ping-pong is a central window of one kind (e.g., a debugger) from which two or more tools (e.g., inspectors) are navigated back and forth. The navigated tools have a unique activity, with a unique incoming and outgoing flow from and to the central windows. Figure 8, in the Atom Task, shows a ping-pong example with a debugger as the central window and

several other tools (six inspectors, one code browser, and one debugger). A ping-pong is marked with p1 in the figure.

B. Visualizations exploration

In this section, we use our vocabulary to explore the visualizations. First, we describe the occurrences of the observable vocabulary instances. Second, we interact with the live visualization to explore the meaning of these instances in the context of each debugging task.

1) *Instances of vocabulary:* We counted the number of occurrences of each vocabulary instance and reported the numbers in Table V. To count, we went over each visualization and looked for all possible instances of one or more of our vocabulary definitions. To simplify a possible check by the readers, Figures 6, 7 and 8 representing, both the control and the treatment tasks, have been manually annotated after analysis.

TABLE V: Instance count of each vocabulary element by user, for their control (C) task visualization and their treatment (T) task visualizations.

User	Chain	Hub	Ping-Pong
A (Fig. 6)	C 9	C 4	C 2
	T 4	T 3	T 2
I (Fig. 7)	C 8	C 5	C 0
	T 4	T 0	T 0
V (Fig. 8)	C 10	C 2	C 2
	T 3	T 1	T 1

First observations lead to the conclusion that these patterns are independent of the task, the user, and the type of used debugger. Indeed, vocabulary instances are observed multiple times for each user, and in each task except the treatment task of User-I for which we observe no hubs and no ping-pongs. We therefore hypothesize that our vocabulary may actually represent *patterns*, i.e., “recurring solutions to standard problems.” [7].

The visualizations in the scope of the current work are insufficient to conclude about such patterns. Broader experiments should be conducted to empirically explore these pattern meanings. However, before this experiment, it was very difficult for us to explain or decompose a debugging session. The debugging activity blueprint allowed us to observe these possible patterns and formulate a hypothesis about them. More targeted use cases or case studies using the blueprint could help researchers observe oddities, leading to new patterns, in the ways developers use their debuggers and interact with their IDE when debugging. From such observations, we can then formulate new hypotheses to explore in real empirical evaluations and learn more about debugging sessions.

2) *Summary analysis of the debugging sessions:* Considering User-A (Figure 6), the debugging activity blueprint enables us to observe that the debugging sessions for the control and the treatment tasks seem to share some properties: no apparent structure is obvious, and there are multiple instances of many tools. These sessions are long and complex in terms of tool interaction.

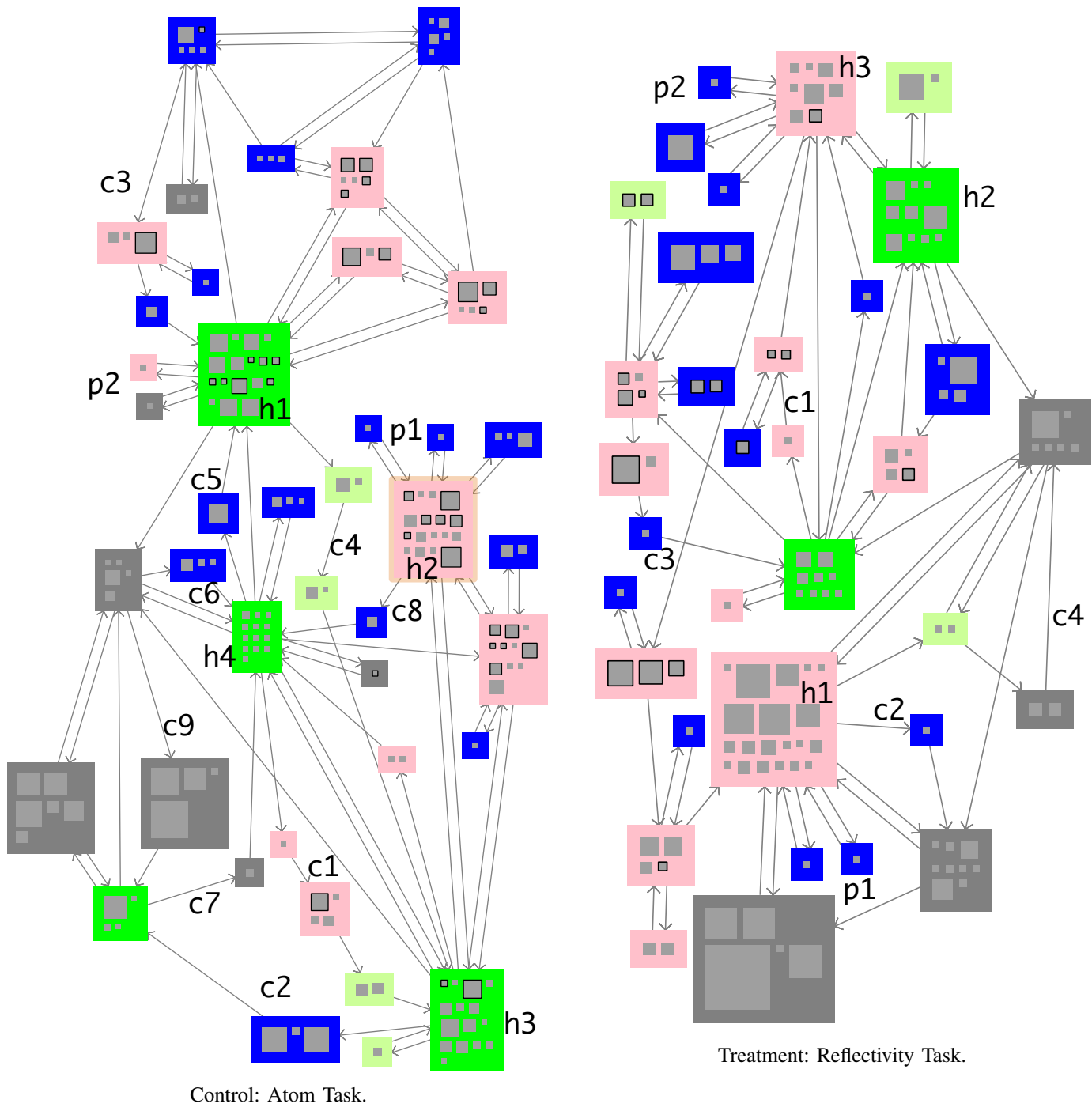


Fig. 6: User-A (novice): visualization of debugging activities for control and treatment tasks. c = chain, h = hub, p = ping pong.

Considering User-I (Figure 7), the debugging activity blueprint enables us to observe that the control and the treatment tasks seem significantly different. The debugging session of the treatment task is simpler, relatively sequential, and without any hub or ping-pong. On the opposite, the control task looks like the debugging sessions of User-A.

Considering User-V (Figure 8), the debugging activity blueprint shows that the debugging session for the control task looks like those of the two other participants even if User-V is an expert. This visualization has hubs, ping-pongs, and chains. His control task is nevertheless simpler implying that experience has an impact. In addition as for User-I, the debugging session for User-V's treatment task is simpler compared to both his control task and the ones of the two other participants.

We do not consider these three participants as representative. Still, the debugging activity blueprint encourages us to think that (i) with traditional debuggers, whatever the experience of the user, the debugging sessions reflects the complexity of the task, (ii) using an object-centric debugger may reduce the time and number of actions needed to debug; (iii) the object-centric can differently simplify a debugging session and (iv) some patterns appear in the debugging sessions whatever the used debugger, object-centric or not.

Further investigations are needed, for example, to understand the impact of using the object-centric debugger. We have three participants, for one, using such a debugger seems to have no incidence, for the two others it seems it has, by simplifying, but the blueprints are different. We were completely in the dark without the debugging activity blueprint. Now, we have a tool to analyze better and understand debugging sessions. New questions arise such as: (i) are the debugging sessions with traditional tools always so complex and not fluid? (ii) does using an object-centric debugger or more generally, a specific debugger simplify and fluidize a debugging session? (iii) does the nature of the task influence the debugging session? (iv) does the expertise of a participant on a project influence a debugging session more than using a specific debugging tool? and so on. With the debugging activity blueprint, new perspectives are opened to understand debugging sessions.

3) Focus: fine-grained analysis of a debugging session:

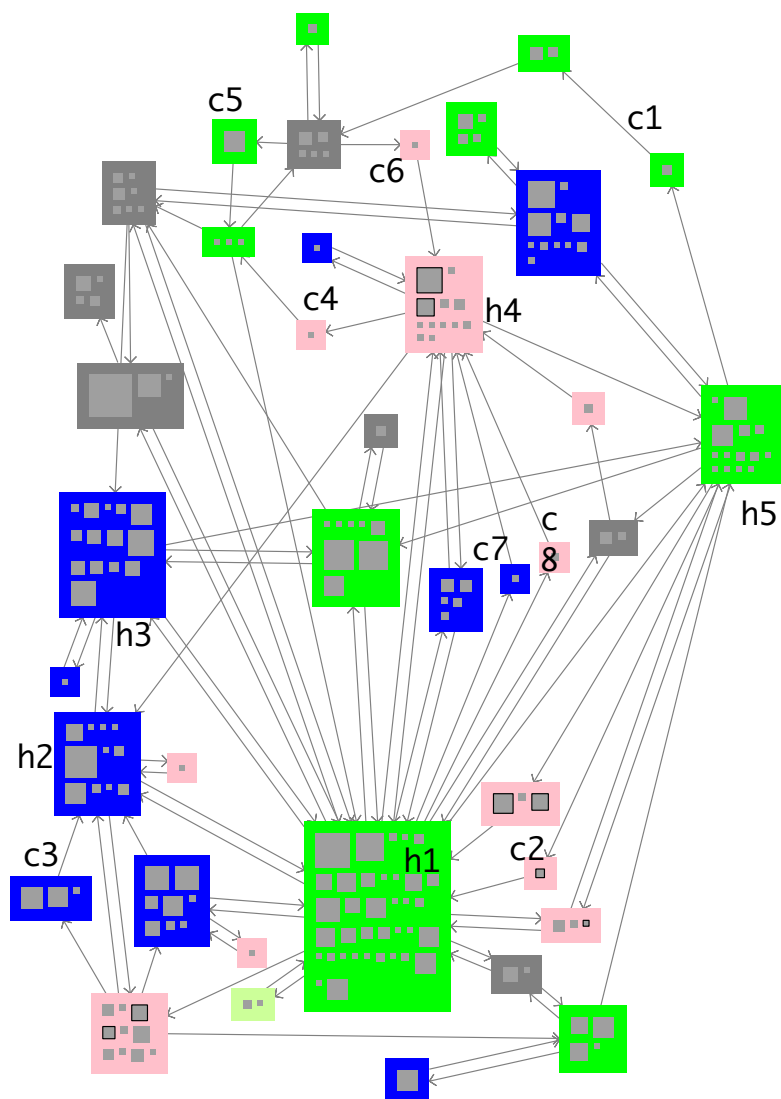
In the following, we detail the debugging session of User-I on their treatment task (Figure 7). We chose this particular visualization because it is radically different from all others. This visualization corresponds to a session where User-I, a novice developer, uses an additional tool named *object-centric debugger*. User-I finished the task (i.e., correctly fixed the bug) in about 21 minutes. User-V (expert, Figure 8) finishes the same task also in 21 minutes but without the additional tool (i.e., in control). User-I's treatment visualization is simple, has no hubs, no ping-pongs, but only chains, and the time to finish the task matches the expert's performance. This is not the case for the novice User-A (Figure 6) on the same task as treatment.

Methodology. We used the live visualization side-by-side with the video recording of User-I's treatment task. We used the interactive flow of the visualization to track User-I's navigation

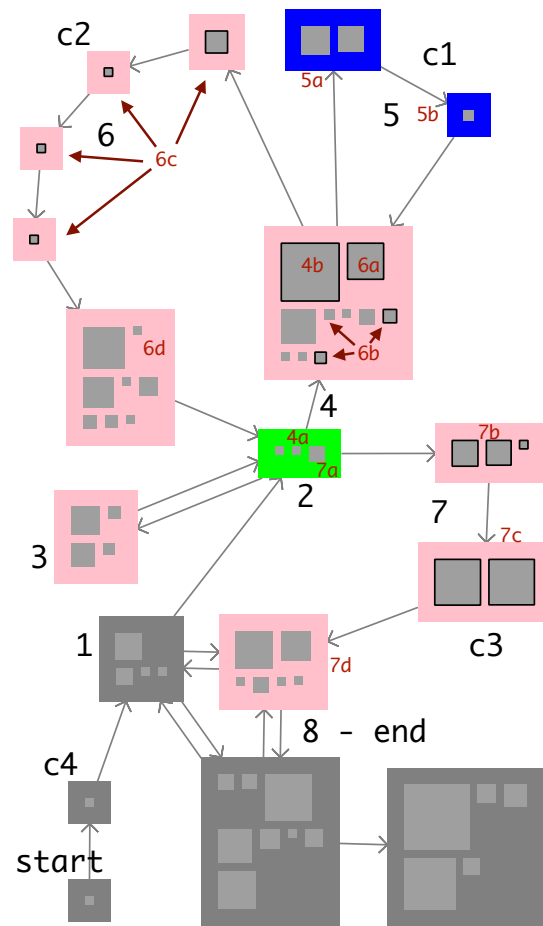
through different activities, the interaction popups to observe the performed actions (e.g., debugger steps), and the detailed actions in the logs associated with each activity to understand the semantics of the performed actions. When we had doubts about interpreting the visualization, we looked at the video recording and investigated both side-by-side. Throughout this analysis, we manually annotated the visualization (see Figure 7 in treatment) with numbers representing the important steps of the debugging session and we described our understanding of these steps in the blueprint using its annotation tool. Finally, after finishing the analysis we watched the entire video recording with our notes and the visualization and compared.

Fine-grained analysis results. The participant started the task and reached the task application window, where the task was described. This analysis is consistent with the screen recording of User-I. Steps listed below are indicated in the blueprint (Figure 7, right side).

- 1) The user spent under a minute reading before moving on to the class mentioned in the task description and accessing the code browser.
- 2) The user executed the unit test pointed out by the task description, which failed and opened a debugger.
- 3) In the opened debugger, the user inspected the failure context and the objects within, in multiple activities in the same debugger window. They performed no actions other than inspecting the execution state for about a minute. They then left the debugger and went back to the central code browser.
- 4)
 - a) The user put a breakpoint at the beginning of the unit test and started debugging the test.
 - b) The user stepped a few times to dive into the code execution and carefully inspected objects in their context within the debugger. Then, the user opened a specific inspector window on one of these objects.
- 5)
 - a) The user performed two inspection activities within the same inspector window. They navigated two different panes, one showing an object and its properties and the other showing the details of the selected properties. They navigated and inspected the properties of the object inspected from 4. They opened a new inspector window on one of these objects.
 - b) The user briefly inspected the newly opened object, then left the inspector to get back to the debugger.
 - a) Based on their previous observation, the user selected a set object in the debugger on which they placed two object-centric breakpoints on methods used to add elements to sets (`Set>>#add:and Set>>#addIfNotPresent:ifPresentDo:`). They proceeded with the execution, which hit a first breakpoint in the same control flow (i.e., in the same debugger).
 - b) The user performed inspection activities until proceeding with the execution again, which hit the breakpoint again.



Control: Atom Task.



Treatment: Reflectivity Task.
(Manually annotated after analysis)

Fig. 7: User-I (novice): visualization of debugging activities for control and treatment tasks. c = chain, h = hub, p = ping pong.

- c) The user proceeded through a chain of four object-centric breakpoint hits (chain 2), which ended up in the last debugger of the chain.
- d) Arrived in this debugger from four object-centric breakpoint hits, the user removed their breakpoints and performed careful inspections of the execution context and its objects within the same debugger. They finally closed the debugger and went back to the central code browser.
- 7) a) In the central code browser, the user debugged the test again and restarted the process.
- b) In the debugger, the user precisely stepped and looked for a particular object (a set) on which they put again an object-centric breakpoint. However, this time they only put it on the `add`: message. They proceeded and the breakpoint hit, opening

- a new debugger. This breakpoint was reached in under a minute.
- c) For 2 minutes, the user inspected the breakpoint context in two debugger activities. They did not seem to find what they were looking for and ended up proceeding and hitting the breakpoint again.
- d) The user inspected the newly opened debugger and spent about 2 minutes inside in multiple activities, inspecting objects and the code context. They made multiple navigations to the task application window where they wrote the correct answer that finished the task.

General conclusions. Our analysis of the blueprint matches accurately what is observable in the video. Once annotated we can understand what is happening in the debugging session by reading the blueprint alone. Video recordings were only needed

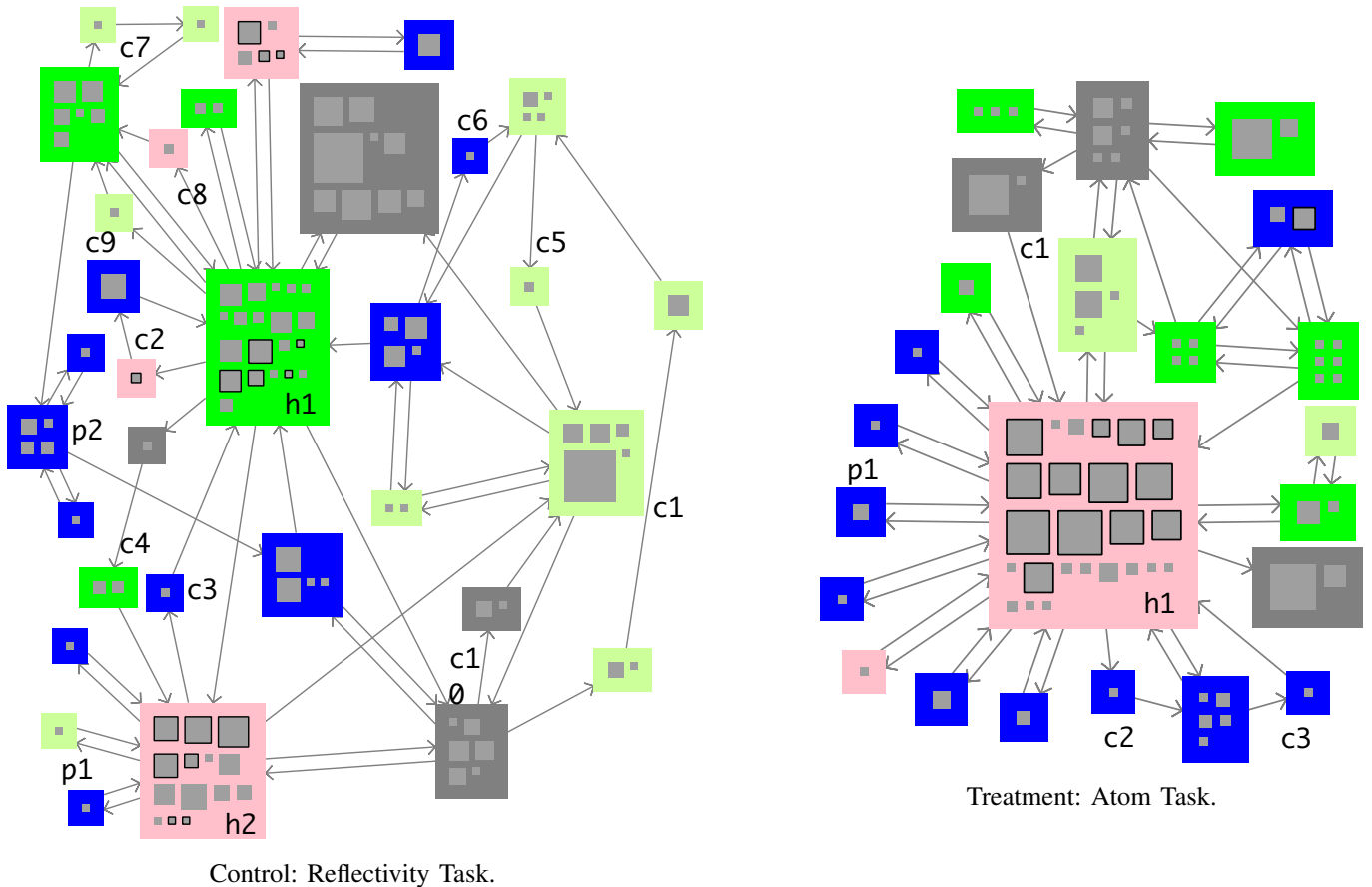


Fig. 8: User-V (expert): visualization of debugging activities for control and treatment tasks. c = chain, h = hub, p = ping pong.

to understand some blueprint oddities, like small interactions in a tool without any connections to other tools. The screen recordings showed that participants sometimes left the tool’s window briefly (for less than 500ms) and then returned, causing these small interactions.

VI. DISCUSSION

Concerning the visualization. The layout may have an incidence on the comprehension of the visualization and possibly on the identification of patterns. To minimize the incidence of the layout, all the visual elements of the visualization are manually draggable. Moreover, the debugging activity blueprint is not a single visualization but a richer tool enabling interactions and annotations, and providing other views to complete the blueprint.

Debugging sessions are complex. Consequently, the visualization cannot be completely simple, even if it provides an abstraction. In particular, it was easy to analyze in detail the User-I Treatment task and to realize that the visualization can completely replace the video. However, for more complex sessions as the ones of the control tasks, we need to see to what extent it can replace the video.

The debugging activity blueprint enables the discovery of patterns, which we graphically identified. Their semantics and

implications need to be further investigated. On the opposite, we have not studied the sequence of debugging activities that could have led to the identification of other patterns, that not graphically emerged.

Concerning the analysis of a debugging session. The debugging activity blueprint opened doors concerning the analysis of debugging sessions. For example, it seems that the use of object-centric debuggers or perhaps more largely specific debuggers has an incidence on the flow of debugging sessions. Similarly, we can study if the experience of the users, their knowledge of the project to debug, or whatever criteria have an incidence on the debugging session, not only in terms of results (resolving the bug or not), or time, but also in the flow of the activities inside the debugging session. We have no answer yet, but we are convinced that now we have the tool to better investigate debugging sessions.

VII. THREATS TO VALIDITY

Conclusion validity. Since we used a small number of sessions, a gap may exist between our observations and what would be found in a representative and a large set of observations. The tasks given to participants were debugging tasks, and as such, we have not studied micro-debugging activities that may occur when adding a new software feature.

Construct validity. Debugging tasks were conducted using the Pharo programming language. As such, the patterns we discovered may be intimately related to Pharo.

Color blindness. Depending on the population, color blindness affects about 8% of the population of men. The most common form of color blindness is the red-green color vision deficiency. We used an online simulator¹ to verify if visual cues due to our coloring remain perceivable in the presence of the red-green color vision deficiency. The simulator indicates that the visual cues are perfectly distinguishable for the red-weak (Protanomaly) and green-weak (Deuteranomaly) visual deficiency.

Visualization correctness. An important premise of our exploratory experiment is that the visualization accurately represents what happens in a real debugging session. We do not formally prove that this is the case, however, we tame this problem by manually comparing logs and our blueprint analyses with screen recordings.

First, we analyzed the raw logs of the six studied debugging sessions and compared them to their associated screen recordings. For each session, we took samples of the screen recording and we compared what was visually happening in the video to the logs of events happening at the same time. The events in the logs always matched the screen recordings. In the scope of this paper, we therefore consider that the logs represent the debugging sessions with enough precision for our exploratory experiment.

Second, our detailed analysis of Figure 7 (treatment part) shows that the blueprint represents a developer activity that matches what happened in the entire debug session shown in the screen recording. This improves our trust in the blueprint accuracy although we did not do this detailed analysis for all the blueprints.

VIII. RELATED WORKS

A common approach to opening up new research paths for improving the development and debugging process is to collect data on how developers interact with their environment while performing development and debugging activities. There are different approaches to collecting data on developers' needs and interactions.

Collecting data through observations or interviews. Fontana and Petrillo [8] mapped breakpoints available to developers from the literature and their observations of existing debuggers. Although they performed an analysis of developers' understanding of these breakpoints, this contribution does not provide insight of how the breakpoints are employed in practice. Alaboudi and LaToza [9]–[12] observed and analyzed developers' actions while debugging, using live observations, or video and audio records. Through interviews with professional engineers from Microsoft, Layman et al. [13] highlighted how practitioners use information and tools to debug. Similarly, Alaboudi and LaToza [14] interviewed 11 professional developers and observed that they frequently

switched debugging activities after a minute. Although these studies provide essential information for understanding the behavior adopted by developers when debugging programs, this mode of analysis is not scalable and seems limited to orders of magnitude between 8 and 77 participants according to the summary of prior studies from [13] or about 30 hours of activity [11].

Collecting data through event logs. Another approach to studying debugging activity is by collecting and aggregating log data. *The Mylar Monitor* [15] records fine-grained events on the Eclipse IDE, i.e. window events, selections, periods of inactivity, commands invoked through menus or key-binding, etc. Similarly, *Rabbit Eclipse* [16], *FeedBaG* [17], and *The ProM* [18] enable the analysis of the sequence of debugging events or actions performed by developers within an IDE, such as the most used commands and the interactions of these commands with files edited by developers. The last presented tools focus on the Eclipse IDE [15], [16] and the Microsoft Visual Studio IDE [17], [18]. The approach of collecting usage data from the IDE suffers a significant limitation. The amount of data to analyze depends on the precision of the recorded actions and the length of the debugging sessions. The more specific the action recorded (i.e. low-level, such as mouse movements), the greater the number of events generated.

Aggregating log data with algorithms. Damevski et al. [19], [20] provide an automatic approach for identifying (using the MG-FSM algorithm) and clustering developers' usage patterns from logs of Visual Studio IDE's usage. The authors identified 20 different clusters of usage patterns and found that developers are reluctant to use conditional breakpoints when debugging. *The Debugging Activity Blueprint* differs by offering visual support for understanding developers' behavior, comparing debugging sessions, and eventually identifying patterns.

Aggregating log data with visualizations. For our contribution, we record fine-grained data because we want to open up the possibility of identifying the developers' intent when using the different parts of the programming environment. As Kovarova et al. [21] suggest, we argue that aggregating the information extracted from debugging sessions through visualization would help explore records of developers' activity while debugging. *Swarm debugging* [22]–[25] proposes a *Method Call Graph*, a *Sequence Stack* diagram, and a list of *Step Into or Breakpoint* events for visualizing information obtained during debugging sessions. *Swarm debugging* aims at sharing information found on a given program during a debugging session among several developers and therefore differs from *The Debugging Activity Blueprint* which aims at helping researchers understand how developers debug. *DFlow* [26], [27] is a visualization of object-oriented program development sessions. The visualization focuses on classes, methods, and related events, i.e. navigation or edition. Whereas debugging and development are intricately activities, our *Debugging Activity Blueprint* differs from *DFlow* by providing more information on the debugging aspects and focusing first on the IDE tools and their usage by developers rather than the source code. *Ferax* [28] is a platform for

¹<https://www.color-blindness.com/coblis-color-blindness-simulator/>

recording developers' activities inside and outside the IDE and summarizing them into four views. *The Debugging Activity Blueprint* does not provide information on developers' activities outside the IDE. However, it offers more precise data on interactions between IDE tools and allows one to inspect tool-specific events.

IX. CONCLUSION

Studying debugging activities is complex and challenging due to the inherent difficulties in aggregating and interpreting data from event logs. This complexity underscores the need for more empirical evidence on how developers use debugging tools.

Our proposed debugging activity blueprint addresses this need by offering a visual tool that aids in the detailed analysis and navigation of events within a programming environment. The blueprint encapsulates the interactions between debuggers and other programming tools, effectively illustrating the flow of debugging activities within an IDE.

Through an exploratory use case involving three participants and two distinct debugging tasks, our blueprint showed itself to be useful in facilitating a fine-grained analysis of intricate debugging scenarios. Furthermore, it opens several open questions in the analysis of debugging sessions. This indicates the potential of our approach to enhance the understanding of debugging practices and support the development of more effective debugging tools. Further research and validation with larger samples and diverse tasks are recommended to confirm and extend these findings.

ACKNOWLEDGMENT

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

REFERENCES

- [1] J. Kubelka, R. Robbes, and A. Bergel, "Live programming and software evolution: Questions during a programming change task," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 30–41.
- [2] J. Kubelka, R. Robbes, and A. Bergel, "The road to live programming: Insights from the practice," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1090–1101. [Online]. Available: <https://doi.org/10.1145/3180155.3180200>
- [3] M. Lanza and S. Ducasse, "Polymetric views—a lightweight visual approach to reverse engineering," *Transactions on Software Engineering (TSE)*, vol. 29, no. 9, pp. 782–795, Sep. 2003.
- [4] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Square Bracket Associates, 2013. [Online]. Available: <http://books.pharo.org/deep-into-pharo/>
- [5] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceeding of the 34rd international conference on Software engineering*, ser. ICSE '12, 2012. [Online]. Available: <http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf>
- [6] S. Costiou, V. Aranega, and M. Denker, "Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use," *The Art, Science, and Engineering of Programming*, vol. 4, no. 3, Feb. 2020.
- [7] D. C. Schmidt, M. Fayad, and R. E. Johnson, "Software patterns," *Commun. ACM*, vol. 39, no. 10, pp. 37–39, Oct. 1996. [Online]. Available: <https://doi.org/10.1145/236156.236164>
- [8] E. A. Fontana and F. Petrillo, "Mapping breakpoint types: an exploratory study," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 1014–1023.
- [9] S. Baltés, O. Moseler, F. Beck, and S. Diehl, "Navigate, understand, communicate: How developers locate performance bugs," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.
- [10] A. Alaboudi and T. D. LaToza, "Rethinking debugging and debuggers," in *12th Annual Workshop at the Intersection of PL and HCI*, 2021.
- [11] —, "An exploratory study of debugging episodes," *arXiv preprint arXiv:2105.02162*, 2021.
- [12] —, "Edit-run behavior in programming and debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2021, pp. 1–10.
- [13] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging revisited: Toward understanding the debugging needs of contemporary software developers," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 383–392.
- [14] A. Alaboudi and T. D. Latoza, "What constitutes debugging? an exploratory study of debugging episodes," *Empirical Software Engineering*, vol. 28, pp. 1–34, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:261662669>
- [15] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, Jul. 2006. [Online]. Available: <http://dx.doi.org/10.1109/MS.2006.105>
- [16] C. Ioannou, A. Burattin, and B. Weber, "Mining developers' workflows from ide usage," in *CAiSE Workshops*, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:46970864>
- [17] S. Proksch, S. Amann, and S. Nadi, "Enriched event streams: A general dataset for empirical studies on in-ide activities of software developers," *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 62–65, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:30268042>
- [18] V. Shynkarenko and O. K. Zhevaho, "Development of a toolkit for analyzing software debugging processes using the constructive approach," *EngRN: Computer Engineering (Topic)*, 2020. [Online]. Available: <https://api.semanticscholar.org/CorpusID:228928155>
- [19] K. Damevski, H. Chen, D. C. Shepherd, and L. L. Pollock, "Interactive exploration of developer interaction traces using a hidden markov model," *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 126–136, 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:12367713>
- [20] K. Damevski, D. C. Shepherd, J. Schneider, and L. L. Pollock, "Mining sequences of developer interactions in visual studio for usage smells," *IEEE Transactions on Software Engineering*, vol. 43, pp. 359–371, 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:15000607>
- [21] A. Kovarova, M. Konopka, L. Sekerak, and P. Navrat, "Visualising software developers' activity logs to facilitate explorative analysis," *Acta Polytechnica Hungarica*, vol. 13, no. 2, 2016.
- [22] F. Petrillo, G. Lacerda, M. Pimenta, and C. Freitas, "Visualizing interactive and shared debugging sessions," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 2015, pp. 140–144.
- [23] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guéhéneuc, "Towards understanding interactive debugging," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 152–163.
- [24] F. Petrillo, Y.-G. Guéhéneuc, M. Pimenta, C. D. S. Freitas, and F. Khomh, "Swarm debugging: The collective intelligence on interactive debugging," *Journal of Systems and Software*, vol. 153, pp. 152–174, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121219300780>
- [25] E. A. Fontana and F. Petrillo, "Visualizing sequences of debugging sessions using swarm debugging," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 139–143.
- [26] R. Minelli and M. Lanza, "Visualizing the workflow of developers," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2013, pp. 1–4.
- [27] R. Minelli, L. Baracchi, A. Mocci, and M. Lanza, "Visual storytelling of development sessions," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 416–420.
- [28] G. di Rosa, A. Mocci, and M. D'Ambros, "Visualizing interaction data inside & outside the ide to characterize developer productivity," *2020 Working Conference on Software Visualization (VISSOFT)*, pp. 38–48, 2020. [Online]. Available: <https://api.semanticscholar.org/CorpusID:226292830>