Towards Understanding Interactive Debugging

Fabio Petrillo*†, Zéphyrin Soh†, Foutse Khomh†, Marcelo Pimenta*, Carla Freitas*, Yann-Gaël Guéhéneuc†
*Federal University of Rio Grande do Sul, RS, Brazil
†École Polytechnique de Montréal, QC, Canada
E-Mails: fabio@petrillo.com,{mpimenta, carla}@inf.ufrgs.br,
{zephyrin.soh,foutse.khomh,yann-gael.gueheneuc}@polymtl.ca

***Abstract*—Debugging is a laborious activity in which developers spend lot of time navigating through code, looking for starting points, and stepping through statements. Yet, although debuggers exist for 40 years now, there have been few research studies to understand this important and laborious activity. Indeed, to perform such a study, researchers need detailed information about the different steps of the interactive debugging process. In this paper, to help research studies on debugging and, thus, help improving our understanding of how developers debug systems using debuggers, we present the Swarm Debug Infrastructure (SDI), with which practitioners and researchers can collect and share data about developers' interactive debugging activities.**

We assess the effectiveness of the SDI through an experiment that aims to understand how developers apply interactive debugging on five true faults found in JabRef, toggling breakpoints and stepping code. Our study involved five freelancers and two student developers performing 19 bug location sessions. We collect videos recording and data about 6 hours of effective debugging activities. The data includes 110 breakpoints and near 7,000 invocations. We process the collected videos and data to answer five research questions showing that (1) there is no correlation between the number of invocations (respectively the number of breakpoints toggled) during a debugging session and the time spent on the debugging task; $\rho = -0.039$ (respectively 0.093). We also observed that (2) developers follow different debugging patterns and (3) there is no relation between numbers of breakpoints and expertise. However, (4) there is a strong negative correlation between time of the first breakpoint ($\rho = -0.637$); and the time spent on the task, suggesting that when developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints too quickly.

We conclude that the SDI allows collecting and sharing debugging data that can provide interesting insights about interactive debugging activities. We discuss some implications for tool developers and future debuggers.

***Index Terms*—Software Maintenance, Interactive Debugging, Debugging Patterns.**

I. Introduction

> ***Debug.*** *To detect, locate, and correct faults in a computer program. Techniques include the use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operations, and traces.*
> —IEEE Standard Glossary of SE Terminology—

Debugging is a common activity during software development, maintenance, and evolution [1] during which developers use debugging tools to detect, locate, and correct faults. Debugging tools can be *interactive* or *automated.*

Interactive debugging tools, *a.k.a. debuggers*, such as *sdb* [2], *dbx* [3], or *gdb* [4] have been used by developers for decades. Modern debuggers are often integrated in development environments, *e.g.,* DDD [5] or the debuggers of Eclipse, Netbeans, Intellij IDEA, Visual Studio Integrated Development Environments (IDEs). With debuggers, developers navigate through the code, looking for locations to place breakpoints, and stepping into statements. While stepping, developers can traverse method invocations, toggle one or more breakpoints, stop and/or restart executions. This exploration process allows developers to gain knowledge about programs and the causes of faults, allowing them to fix the faults.

Automated debugging tools require both successful and failed runs and do not support programs with interactive inputs [6]. Consequently, they have not been widely adopted in practice. Moreover, automated debugging approaches are often unable to indicate the "true" locations of faults [7]. Other more interactive approaches, such as slicing and query languages, help developers but, to date, there have been no evidence that they significantly ease developers' debugging activities.

Roßler [7] advocates for the development of a new family of debugging tools that are context-aware and that rely on true scenarios (even though Ceccato *et al.* [8] recently showed that automatically-generated test-cases are as useful for debugging as manual test cases). To build context-aware debugging tools, researchers need an understanding of developers' debugging activities and the contextual factors of fault fixing activities. Thus, researchers need tools to collect and share data about developers' interactive debugging activities.

This paper presents SDI and aims to use SDI to collect debugging activities and study how developers perform such activities. The Swarm Debug Infrastructure (SDI), an open-source infrastructure[1] integrated into Eclipse allows practitioners and researchers to collect and share fine-grained data about developers' interactive debugging activities. The understanding of debugging activities could help practitioners and researchers to develop new families of debugging tools that are more efficient and/or adapted to the particularity of each debugging activity. Moreover, assessing whether developers follow debugging patterns could be the first step toward recommending locations to toggle breakpoints that must reduce debugging effort and thus improve developers productivity.

The understanding of how developers perform debugging activities and/or debugging patterns is an example of use of debugging activities collected by SDI. In addition to the per-

[1] https://github.com/SwarmDebugging

spective studied in this paper, the data collected by SDI could be useful to assess the developers' interest and knowledge of the code: When developers toggle breakpoint in a program element, the element seems relevant to the task at hand. By collecting debugging activities, we could know the program elements interested by the developer to resolve the task. Thus, these debugging activities could help to assess the developers' knowledge of the code.

Thus, this paper makes the following three contributions:

- We introduce a novel approach for debugging named swarm debugging and summarise the infrastructure that support the approach;
- We present an infrastructure, the Swarm Debug Infrastructure (SDI), to gather, store, and share data about interactive debugging activities;
- We conduct an experiment showing the relation between tasks' elapse time, developers' expertise, and debugging patterns.

The paper is organized as follows. Section II provides some background about debugging. Section III presents the swarm debugging approach and Section IV describes the Swarm Debugging Infrastructure. Section V details our experiment that evaluates the SDI, testing whether it can help answer research questions pertaining to developers' debugging activities. Finally, Section VII concludes the paper and outlines some avenues for future work.

II. Background

This section provides background details about interactive debugging. It also presents some related studies about (1) program understanding (finding starting method, locating and recommending relevant program elements to developers) and (2) debugging tools to support program understanding

A. Interactive Debugging

Debugging is the process of finding and resolving faults that prevent correct operations of software programs [9]. Thus, any process that aims at finding faults can be considered "debugging". The software engineering community usually focuses on one kind of particular debugging process, which consists in using a tool called a debugger: interactive debugging. Yet, developers have used many other processes and tools to debug programs. These processes and tools range from control flow graph, profiling tools, logs [10], to static analyses [11].

In this paper, we focus on interactive debugging, which consist in using a debugger and which is also known as *program animation*, *stepping*, or *following execution*. Also, many developers refer to this process as simply *debugging*, because several IDEs provide debuggers to support *debugging*. However, while *debugging* is the process of finding faults, *interactive debugging* is one particular debugging process in which developers use tools to interactively investigate the execution of a program. We use the expressions *interactive debugging* or *stepping*, but there is not yet a consensus on what is the best name for this debugging process.

B. Program Understanding

Previous work studied and reported how developers understand programs and provided tools to support program understanding. Maalej *et al.* [12] observed and surveyed developers to comprehend how they understand program. They reported that to understand the program, developers need to acquire runtime information and frequently execute the application using a debugger. Ko *et al.* observed that developers spend large amounts of times navigating between program elements. They modeled program understanding as a process of searching, relating, and collecting relevant information [13]. To prevent redundant navigation between program elements, Coblenz provided the tool JASPER to collect and display elements relevant to the current element [14].

Sillito *et al.* identified the questions that developers ask when finding and extending starting methods [15]. They described how developers answer these questions during software maintenance activities.

Feature and fault location approaches are used to identify and recommend program elements that are relevant to a task at hand [16]. To recommend relevant elements to developers, these tools used the bug report [17], domain knowledge [18], version history and bug report similarity [16]. In contrast to these approaches, Mylyn uses developers' activities (interaction traces collected during maintenance tasks) to reduce developers information overhead and to show in the developers' IDE only program elements that may be relevant to the task [19]. Mylyn interaction traces have been used to study work interruption [20], editing patterns [21], [22], and program exploration patterns [23]. In addition to Mylyn, other tools collect data during maintenance tasks. Eclipse UDC (Usage Data Collector) [24] collects data on developers usage of the Eclipse (*i.e.,* activating views, editors, etc.). The usage data have been used to study the copy/paste behaviour of developers [25].

To the best of our knowledge, Mylyn and UDC are the only Eclipse plugin that monitor and collect data during maintenance tasks. None of the previous work consider collecting and use debugging data. Thus, the SDI complements previous approaches by collecting data that allow researchers to study how developers find starting methods, possibly restoring parts of the developers' contexts after interruption by recalling previous breakpoints, for example.

C. Debugging Tools for Program Understanding

When maintaining and evolving programs, developers must locate and understand the causes of the faults. Some developers tend to print pieces of text (*e.g.,* `System.out.print()` in Java) to identify faulty program elements. Debugging tools have been developed to help developers locate and–or understand relevant elements.

Araki *et al.* described the debugging activity as an interactive process where developers make hypotheses, then verify them by examining the problems [26]. Developers then refute or validate their hypotheses until the tasks are resolved.

Katz and Anderson [27] conducted several experiments to study how developers debug programs. They observed that the participants' understanding is affected by their debugging behaviour. Participants' ability to fix faults is not affected by their debugging behaviour but participants faced difficulties locating faulty program elements.

Romero *et al.* [28] extended the work by Katz and Anderson [27] and identified high-level debugging strategies, *e.g.,* stepping and breaking execution paths and inspecting variable values. They reported that, according to their background and their level of expertise, developers use differently the information available in the debugging environment.

Zayour and Hamdar [29] studied the difficulties faced by developers when debugging in industrial IDEs. They reported that the nature of the IDE affects the time spend by developers during debugging activities.

Although the software engineering community provides debugging tools to improve fault localisation and program understanding, none of these collect debugging activities data to help understand debugging activities with the goal of improving software debugging tools and–or program understanding. The SDI provides an opportunity to collect and share debugging data, to study and improve debugging tools.

III. Swarm Debugging

To understand and support interactive debugging, we introduce the approach of swarm debugging, which aims at addressing three challenges in software engineering: the time and effort spend by developers during debugging activities, the difficulties of deciding where to set breakpoints, and the possibility of leveraging the developers' collective intelligence to improve debugging activities.

First, developers spend long periods of times in debugging sessions to find faults or understand a program [30]. During these sessions, using traditional debugging tools, they gather a lot of information and create mental models of the program, [31], [32]. Unfortunately, several studies have shown that developers quickly forget details when they explore a different location in the source code, losing parts of their mental models [33]. In a recent research, Tiarks and Röhms [33], who investigated the behaviour of 28 professional developers, observed that to recall parts of their mental models, some developers write notes on pieces of papers or use external editors as short-term memory.

Second, developers must find suitable breakpoints when working with debuggers [33] to reproduce the data and control flow of the program. Tiarks and Röhms [33] observed that professional developers encounter problems to set adequate breakpoints and that deciding where to toggle breakpoints is an "extremely difficult" task. They also observed that, often, developers set a lot of breakpoints in the beginning of their debugging sessions to then discard irrelevant breakpoints while they debug the program, which causes a significant overhead to developers.

Third, no previous work leverages the developers' collective intelligence [34] to improve debugging activities even though software development is, in general, a cooperative effort [35]. Bruch *et al.* [36] and Storey *et al.* [37] claim that collective intelligent is an open-field for new software development tools. Bruch *et al.* [36] argue that actual Integrated Development Environments (IDEs) only integrate tools for and knowledge of a single developer and leave out other developers. Moreover, developers use IDEs only because they integrate all tools necessary to browse, manipulate, and build programs. If developers have questions about a particular piece of code, they must go outside of their IDEs to find answers, for example by asking colleagues or searching on-line. After they found an answer, the newly gained knowledge is *usually lost* inside the IDEs. In addition, Storey *et al.* [37] argue that new developers expect collaborations: the newer generation of developers is proficient in social media, for communication and learning. They are opened, transparent, and expect to share their knowledge.

Swarm debugging aims at addressing these challenges and is founded on three studies: (1) the declarative and visual debugging environment for Eclipse called JIVE [38], (2) a novel user interface to support feature location named In Situ Impact Insight (I3) [39], and (3) ElasticSearch for mining and performing research on software repositories [40]. First, JIVE is an Eclipse plug-in that is based upon the Java Platform Debugging Architecture (JPDA) and can analyse Java program executions. JIVE requests notification of certain runtime events, including class preparations, step, variable modifications, method entries and exits, thread starts and ends. Our infrastructure also uses JPDA to collect debug data. Second, I3 [39] introduces a novel user interface on which developers can retrieve and visualise textual similarity in source code. The visualisations also provide a starting point for following relations between textually similar or co-changed methods. Similar to I3, our infrastructure provides novel visualisations. Third, it uses ElasticSearch to allow developers searching the data collected during debugging sessions.

Thus, as debugging is a foraging process [41], [42], swarm debugging wants to collect data during debugging sessions; store this data (breakpoints, reachable paths, and so on), transform this data into knowledge through visualisations and searching tools, and share this knowledge among developers to create a collective intelligence about programs, through debugging. The next section details our infrastructure.

IV. The Swarm Debug Infrastructure

The Swarm Debug Infrastructure (SDI) provides tools for collecting, sharing, and retrieving debugging data collected during developers' debugging activities using the Eclipse IDE[2] and its integrated debugger. It is organized in three main modules: (1) the Swarm Debug Services; (2) the Swarm Debug Tracer; and, (3) Swarm Debug Views.

A. Swarm Debug Services

The Swarm Debug Services (SDS) provide the infrastructure needed by the Swarm Debug Tracer (SDT) to store and,

[2] https://www.eclipse.org/

later, share debugging data from and between developers. Figure 1 shows the architecture of this infrastructure. The SDT sends RESTful messages that are received by a SDS instance that stores them in three specialized persistence mechanisms: an SQL database (PostgreSQL), a full-text search engine (ElasticSearch), and a graph database (Neo4J).

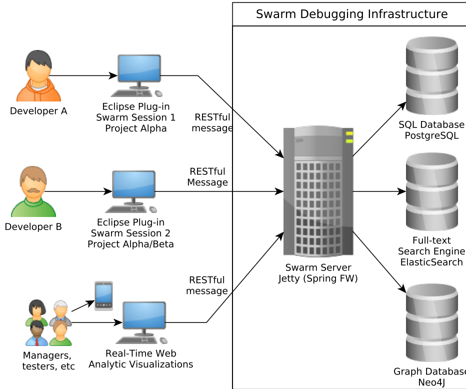


Fig. 1: The Swarm Debug Services architecture

The three persistence mechanisms use similar sets of concepts to define the semantics of the SDT messages. We choose and define domain concepts to model software projects and debugging data. Figure 2 shows the meta-model of these concepts using an entity-relationship representation. The concepts are inspired by the FAMIX Data model [43]. The concepts include:

- **Developer** is a SDT user. She creates and executes debugging sessions.
- **Product** is the target software product. A product is a set of Eclipse projects (1 or more).
- **Task** is the task to be executed by developers.
- **Session** represents a Swarm Debugging session. It relates developer, project, and debugging events.
- **Type** represents classes and interfaces in the project. Each type has a source code and a file. SDS only considers types that have source code available as belonging to the project domain.
- **Method** is a method associated with a type, which can be invoked during debugging sessions.
- **Namespace** is a container for types. In Java, namespaces are declared with the keyword *package*.
- **Invocation** is a method invoked from another method (or from the JVM, in case of the `main` method).
- **Breakpoint** represents the data collected when a developer toggles a breakpoint in the Eclipse IDE. Each breakpoint is associated with a type and a method if appropriate.
- **Event** is an event data that is collected when a developer performs some actions during a debugging session.

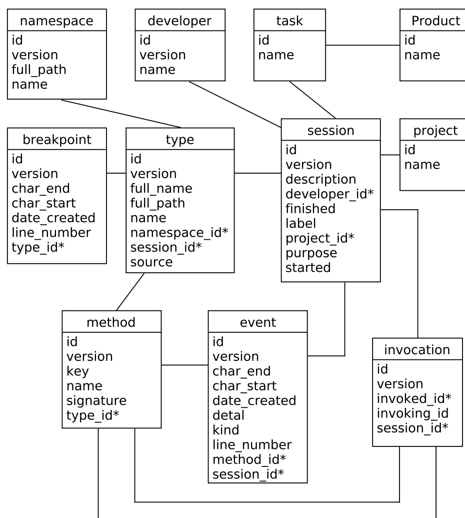


Fig. 2: The Swarm Debug metadata [44]

The SDS provides several services for manipulating, querying, and searching collected data: (1) Swarm RESTful API; (2) SQL query console; (3) full-text search API; (4) dashboard service; and (5) graph querying console.

1) Swarm RESTful API: The SDS provides a RESTful API to manipulate debugging data using the Spring Boot framework[3]. Create, retrieve, update, and delete operations are available through HTTP requests and respond with a JSON structure. For example, upon submitting the HTTP request:

```
http://swarmdebugging.org/developers/
search/findByName?name=petrillo
```

the SDS responds with a list of developers whose names are "petrillo", in JSON format.

2) SQL Query Console: The SDS provides a console available at http://db.swarmdebugging.org to receive SQL queries (SQL) on the debugging data, providing relational aggregations and functions.

3) Full-text Search Engine: The SDS also provides an ElasticSearch[4], which is a highly scalable open-source full-text search and analytic engine, to store, search, and analyse the debugging data. The SDS instantiates an instance of the ElasticSearch engine and offers a console for executing complex queries on the debugging data.

4) Dashboard Service: The ElasticSearch allows the use of the Kibana dashboard. The SDS exposes a Kibana instance on the debugging data. With the dashboard, researchers can build charts describing the data. Figure 3 shows a Swarm Dashboard embedded into Eclipse as a view.

5) Graph querying console: The SDS also persists debugging data in a Neo4J[5] graph database. Neo4J provides a query language named Cypher, which is a declarative, SQL-inspired language for describing patterns in graphs. It allows researchers to express what they want to select, insert, update, or delete from a graph database without describing precisely

[3]http://projects.spring.io/spring-boot/
[4]https://www.elastic.co/
[5]http://neo4j.com/

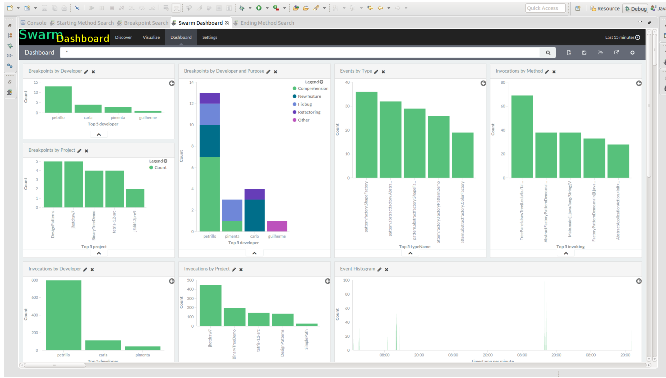


Fig. 3: Swarm Debug Dashboard

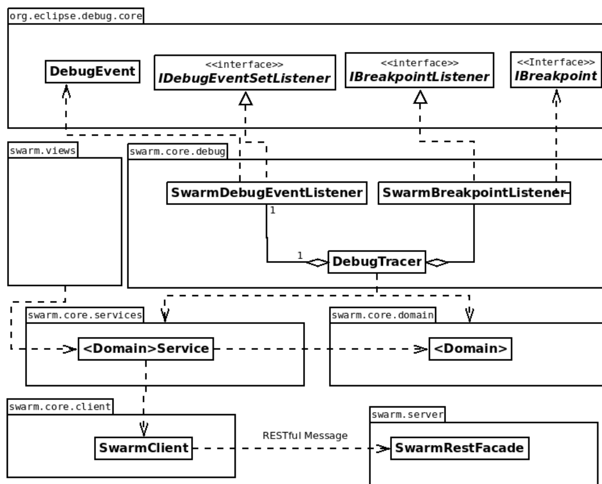


Fig. 4: The Swarm Tracer architecture [44]

how to do it. The SDS exposes the Neo4J Browser and creates an Eclipse view.

B. Swarm Debug Tracer

Swarm Debug Tracer (SDT) is an Eclipse plug-in that listens to debugger events during debugging sessions, extending the Java Platform Debugging Architecture (JDPA). Using the Eclipse JPDA, events are listened by our DebugTracer that implements two listeners: `IDebugEventSetListener` and `IBreakpointListener`. Figure 4 shows the SDT architecture.

After an authentication process, developers create a debug session using a view Swarm Manager and toggle breakpoints, trigger stepping events as *Step Into*, *Step Over* or *Step Return*. These events are caught and stack trace items are analyzed by the Tracer, extracting method invocations.

To use the SDT, a developer must open the view "Swarm Manager" and establish a connection with the Swarm Debugging Services. If the target project is not into the Swarm Manager, she can associate any project in her Workspace into Swarm Manager. This association consists of linking a Swarm Session with a project in the Eclipse workspace. Second, she must create a Swarm session. Once a session is established,

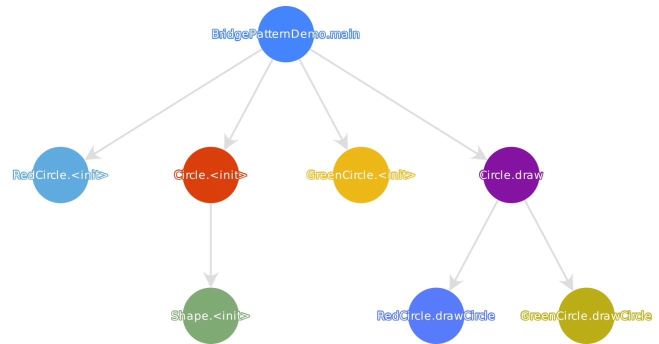


Fig. 5: Method call graph for Bridge design pattern [44]

she can use any feature of the regular Eclipse debugger, the SDT collects developers' interaction events in the background, with no visible performance decrease.

Typically, the developer will toggle some breakpoints to stop the execution of the program of interest at locations deemed relevant to fix the fault at hand. The SDT collects the data associated to these breakpoints (locations, conditions, and so on). After toggling breakpoints, the developer runs the program in debug mode. The program stops at the first reached breakpoint. Consequently, for each event, such as *Step Into* or *Breakpoint*, the SDT captures the event and related data. It also stores data about methods called, storing invocations entry for each pair invoking/invoked method. Following the foraging approach [42], the SDT only collects invoking/invoked methods that were visited by the developer during the debugging session, ignoring other invocations. The debugging activity continues until the program run finishes. The Swarm session is then completed.

To avoid performance and memory issues, the SDT collects and sends the data using a set of specialised *DomainServices* that send RESTful messages to a *SwarmRestFacade*, connecting to the Swarm Debug Services.

C. Swarm Debug Views

On top of the SDS, the SDI implements and proposes several tools to search and visualise the data collected during debugging sessions. These tools are integrated in the Eclipse IDE, simplifying their usage. They include, but are not limited to:

1) Dynamic method call graphs: which are direct call graphs [45], as shown by Figure 5, to display the hierarchical relations between invoked methods. They use circles to represent methods and oriented arrows to express invocations. Each session generates a graph and all invocations collected during the session are shown on these graphs. The starting points (non-invoked methods) are allocated on top of a tree and adjacent nodes represent invocations sequences. Researchers can navigate sequences of invocation methods pressing the F9 (forward) and F10 (backward) keys. They can also directly go to a method in the Eclipse Editor by double-clicking on nodes in the graphs.

2) Breakpoint search tool: which researchers and developers can use to find suitable breakpoints [41] when working with the debugger. For each breakpoint, the SDS captures the type and location in the type where the breakpoint was toggled. Thus, developers can share their breakpoints. The breakpoint search tool allows *fuzzy*, *match*, and *wildcard* ElasticSearch queries. Results are displayed in the Search View table for easy selection. Developers can also open a type directly in the Eclipse Editor by double-clicking on a selected breakpoint.

3) Starting/Ending method search tool: which allows searching for methods that (1) only invoke other methods but that are not explicitly invoked themselves during the debugging session and (2) that are only invoked by others but that do not invoke other methods.

Formally, we define Starting/Ending methods as follows. Given a graph $G = (V, E)$, where V is a set of vertexes $V = \{V_1, V_2, \ldots, V_n\}$ and E is a set of edges $E = \{(V_1, V_2), (V1, V3), \ldots\}$. Then, each edge is formed by a pair: $< V_i, V_j >$, were V_i is the *invoking* method and V_j is the *invoked* method. If α is the subset of all vertexes *invoking* methods and β is the subset of all vertexes *invoked* by methods, then the Starting and Ending methods are:

$$StartingPoint = \{V_{SP} \mid V_{SP} \ \in \alpha \ and \ V_{SP} \notin \beta\}$$

$$EndingPoint = \{V_{EP} \mid V_{EP} \ \in \beta \ and \ V_{EP} \notin \alpha\}$$

Locating these methods is important in a debugging session, because they are the entries and exits points of a program at runtime.

4) Source code full-text search tool: which expands the Eclipse IDE "default" search tool, using the full-text search features provided by ElasticSearch.

In summary, through SDI, we provide a technique and model to collect, store and share interactive debugging session data, contextualizing breakpoints and events during these sessions. We created real-time and interactive visualizations using web technologies, providing an automatic memory for developer explorations. Moreover, dividing software exploration by sessions and its call graphs are easy to understand because only intentional visited areas are shown on these graphs, one can through the execution of a project and see only the important areas that are relevant to developers.

Currently, the Swarm Tracer is implemented in Java, using Eclipse Debug Core services. However, SDI provides a RESTful API that can be accessed independently, and new tracers can be implemented for different IDEs or debuggers.

V. Experimental Study with SDI

We present the study on the use of SDI to collect and share debugging activities. This study aims to evaluate how the data collected by SDI could be useful. Thus, we use the data collected by SDI to answer five research questions. We first present the context of the study. Then, we explain the design and report the results of the study.

Task	Time (min.)
318	13
667	31
669	11
993	28
1026	21

TABLE I: Elapse time by task (average)

A. Context

Studies and discussions about interactive debugging are scarce in the literature pertaining to program comprehension, so we could elaborate many research questions to better understand such important software development activity, *i.e.,* debugging. To illustrate the use of the SDI, we formulate the following five research questions:

RQ1: Is there a correlation between the numbers of invocations and tasks' elapsed time?
RQ2: Is there a correlation between the numbers of breakpoints and tasks' elapsed time?
RQ3: Do developers explore/debug in different ways a task?
RQ4: Is there a correlation between the numbers of breakpoints and developers' expertise?
RQ5: Is there a correlation between time of first breakpoint and task's elapsed time?

To answer the research questions above, we run the experiment designed in the next section.

B. Study Design

To answer the research questions, we proceeded as follows[6]:

1) Tasks Definition: We had to choose debugging tasks to trigger participants' debugging activities. We chose to ask participants to find the locations of true faults in an independent, open-source program. We selected JabRef[7] as target program, which is an open-source bibliography reference manager developed in Java. We chose JabRef because it has faults publicly reported in its issue tracker and its domain was easy to understand by the participants. We picked 5 faults reported against JabRef v3.2 in its issue tracker and asked participants to find the locations of the faults described in issues 318, 667, 669, 993 and 1026.

In order to estimate task's effort, we calculated averages of elapse time for each task by participant. Table I shows the average time (in minutes) for each task. Furthermore, in average, participants spent **21 minutes** to complete the bug location tasks.

2) Participants: In order to reproduce a realistic industry scenario, we recruited 5 professional freelancer developers[8]. Among them, 2 Java experts and 3 intermediates, 100% were male, 100% used Eclipse and 100% used debuggers frequently. As many other experimental studies, we asked 2 volunteer students at Polytechnique Montréal to participate in our experimental study.

[6] All artifacts on http://swarmdebugging.org/publications/qrs2016.
[7] http://www.jabref.org/
[8] https://www.freelancer.com/

3) Artifacts: We provided participants with instructions by two documents. The first document was an experiment tutorial[9] which explained how to install and configure all tools to perform a warm-up task and the experimental study. We also used the warm-up task to confirm that the participants' environments were correctly configured and that the participants understood the instructions. The warm-up task was described using a video to guide the participants. We make available this video on-line[10].

The second document presented the 5 issues with a description and some piece of information to reproduce the faults. To reduce the participants' effort to reproduce the faults, we offered videos demonstrating step-by-step how to reproduce the faults. We also provided the participants with an electronic form to report whether they were tired or not at the end of the experiment.

For this experimental study, we used Eclipse Mars 2 and Java 8, the SDI and its Swarm Debug Tracer plug-in, and two Java projects: an small Tetris game for the warm-up task and JabRef v3.2 for the experimental study. All participants received the same workspace, provided by our artifact repository.

4) Data Collection: After installing the environment (Eclipse and the SDI), each participant executed the warm-up task. This tasks consisted in starting a debugging session, toggling a breakpoint, and debugging a Tetris program to locate a given method. After the warm-up task, each participant executed debugging sessions to find the location of faults described in the five selected issues. We did not set a time constraint but suggested 20 minutes by fault. We asked participants to control their fatigue, asking them to go to the next task if they felt tired while informing us of this situation in their reports. Finally, each participant filled a report to provide their answers and other information, whether they completed the tasks successfully or not.

All services were available on our server[11] during this debugging sessions and the experimental data were collected in the course of 8 days. We also collect the video capture for the participants. The experiment tutorial contained the instruction to install and set the OBS (Open Broadcaster Software), an open source system for live streaming and recording[12]. Participants were asked to provide the video captured during the experiment. A video was recorded for each task, providing about **6 hours** of effective developer's activities. We had 19 completed tasks by 5 developers, 110 collected breakpoints and more then 6000 invocations.

5) Data Analysis: After the participants completed the debugging sessions (successfully or not), we used the tools provided by the SDI on the data collected to answer each research question. To answer RQ1 and RQ2, we used SQL queries, with which we can extract all the invocations and breakpoints set during each session and find a relationship between breakpoints and tasks. The example of SQL to extract data to RQ2 is:

```
select t.id taskId,
s.id sessionId, count(*)
from breakpoint b, task t,
type tp, session s
where b.type_id = tp.id
and tp.session_id = s.id
and s.task_id = t.id
group by s.id, t.id order by s.id
```

Finally, to answer RQ3, we plotted the call graph of each debugging session using the SDI. We organized these graphs by tasks and by numbers of invocations, analyzing each graph to identify navigation patterns. The SQL to extract data to RQ3 is:

```
select s.developer_id, tsk.title,
s.id, count(*) as invocations
from  product p, task tsk,
session s, invocation i
where p.id = 1
and p.id = tsk.product_id
and tsk.id = s.task_id
and i.session_id = s.id
group by tsk.title, s.id
order by tsk.title,invocations
```

[9]http://swarmdebugging.org/publications/experiment/tutorial.html
[10]https://youtu.be/U1sBMpfL2jc
[11]http://server.swarmdebugging.org
[12]https://obsproject.com

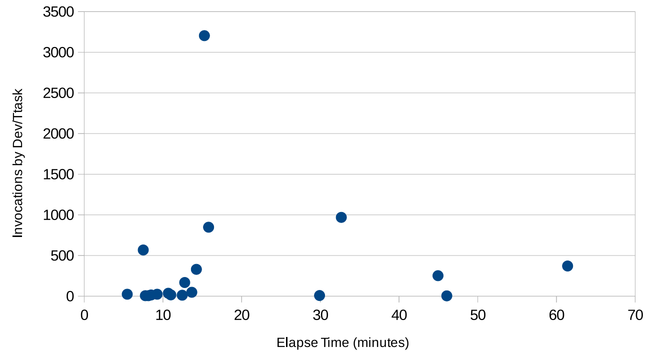


Fig. 6: Invocations (Dev/Task) by Elapse Time

C. Results

We now report the results of our analyses.

RQ1: Is there a correlation between the numbers of invocations and tasks' elapsed time?

By analyzing the elapse time of each task executed by developer and invocations , we can plot Figure 6 in which we can observe that **there is not a correlation between the numbers of invocations and elapse task time**. This conclusion can be strengthened by the Pearson's correlation coefficient ($\rho = -0.039$) lower than 0.1.

RQ2: Is there a correlation between the numbers of breakpoints and tasks' elapsed time?

By analyzing the elapse time of each task executed by developer and breakpoints, we can plot Figure 7 in which we can observe that **there is not a correlation between the numbers of toggled breakpoints and elapse task time**. This conclusion can be strengthened by the Pearson's correlation coefficient ($\rho = 0.093$) lower than 0.1.

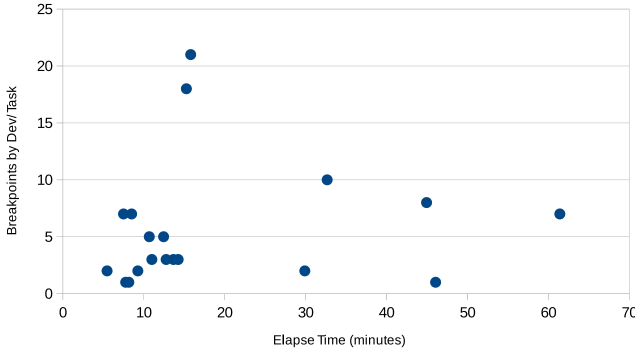


Fig. 7: Invocations (Dev/Task) by Elapse Time

RQ3: Do developers explore/debug in different ways a task?

We observed two distinct debugging navigation patterns: (1) a *fuzzy* debugging pattern and (2) a *straight* debugging pattern. In the fuzzy debugging pattern, the call graph presents several branches, showing that participants used a foraging approach. Figure 9 shows two typical fuzzy debugging graphs. In the straight debugging pattern, participants followed a straight or quasi-straight set of invocations, as shown in Figure 10.

Furthermore, we identified a strong correlation between expertise and navigation patterns: the more expert the participants, the more straightforward their navigation patterns. Future work will further study this correlation to confirm its existence and provide explanations and, possibly, recommendations to developers during debugging activities.

RQ4: Is there a correlation between the numbers of breakpoints and developers' expertise?

By relating the numbers of breakpoints toggled during debugging tasks and developers' expertise, we can conclude that **there is no relation between numbers of breakpoints and expertise**. Although this result may seem counter-intuitive, because the more expert a participants, the less breakpoints she could need, we explain this result three ways. First, the numbers of breakpoints is possibly more related to task complexity. Second, all participants were newcomers to JabRef. Third, the chosen program and issues are not representative of all programs and debugging tasks.

RQ5: Is there a correlation between time of first breakpoint and task's elapsed time?

Breakpoints are key for interactive debugging, and an important breakpoint is a first toggle breakpoint during a session. We analysed 19 interactive debugging sessions in which 73% (14/19 sessions) started a first debugger execution after lower than 3 minutes of toggled first breakpoint, and 52% (10/19 sessions) started a first debugger immediately (lower than 10 seconds) after had defined a first breakpoint. In conclusion, a first breakpoint is an important decision on an interactive debugging session.

In order to analyse if there is a relation between time of first breakpoint and task elapsed time, for each session, we normalized our data dividing each first breakpoint time by task elapsed time, and we associated these ratio with its respective elapsed time, plotting Figure 8.

Analysing Figure Figure 8, it is clear that **there is a strong correlation between time of first breakpoint** ($\rho = -0.637$), and task elapsed time is **inversely proportional** to the time of task's first breakpoint, following a correlation function:

$$f(x) = \frac{\alpha}{x^{\beta}} \tag{1}$$

where $\alpha = 125$ and $\beta = 0.72$.

On the whole, results show that **whether developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints quickly**.

D. Discussions

As any empirical study, this experimental study is subject to limitations that threaten the validity of its results. The first limitation pertains to the number of participants involved in the study. With 7 participants, we can not claim generalization of the results. However, we accept this limitation because the goal of the study was to show the effectiveness of the data collected by the SDI to obtain insights about developers' debugging activities. Future studies with larger numbers of participants and more systems and tasks are needed to confirm or infirm the results of this study. The SDI and all the material used in our experimental study are publicly available at http://swarmdebugging.org/publications/qrs2016.

Other threats to the validity of our results concern their internal, external, and conclusion validity. We accept these threats because the aim of the experimental study was to show the effectiveness of the SDI to collect and share data about developers' interactive debugging activities, not to answer with strong statistical significance the research questions. Future work is needed to perform in-depth experimental studies with these research questions and other, possibly drawn from the questions that developers asked found by Sillito *et al.* [15].

VI. Implications for the Research on Software Debugging

We now discuss some implications of our work for SE researchers, developers, debuggers' developers and educators. SDI is an open and freely available infrastructure that SE researchers can use to perform new empirical studies about debugging and–or software static and dynamic analysis.

Developers can use SDI to record their debugging patterns in order to identify debugging strategies that are more efficient

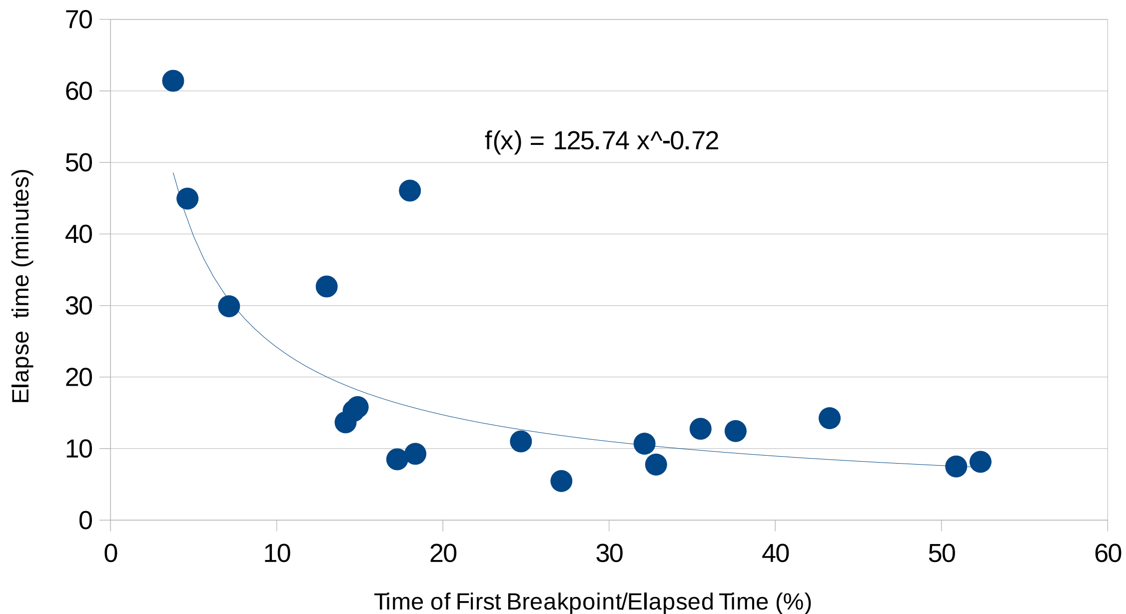


Fig. 8: Relation between time of first breakpoint and task elapsed time

in the context of their project, allowing them to improve their debugging skills.

Developers can share their debugging activities, such as breakpoints and–or invocations to improve collaborative work and ease software maintenance. While developers usually work on specific tasks, there are sometimes re-open issues and–or similar tasks that need to understand or toggle breakpoints on the same entity. Thus, using breakpoints previously toggled by a developer could help to assist another developer working on similar task. For instance, the breakpoint search tools can be used to retrieve breakpoints from previous debug sessions, which could help speed up a new debugging session, providing developers with valid starting points. Therefore, the breakpoint search tool can decrease the time spent to toggle a new breakpoint.

Debugger's developers can use SDI to understand IDE users' behaviours and requirements. This knowledge base is important to create new tools, using novel data-mining techniques, to integrate different data sources. SDI provides a transparent framework for developers to share debugging information, creating a collective intelligence about their software projects.

Last but not least, educators could leverage SDI tools to teach interactive debugging techniques, tracing their students' debugging sessions, and evaluating their performance. Data collected by SDI about debugging sessions of professional developers could also be used to educate students, *e.g.,* by showing them examples of good and bad debugging patterns.

VII. Conclusion

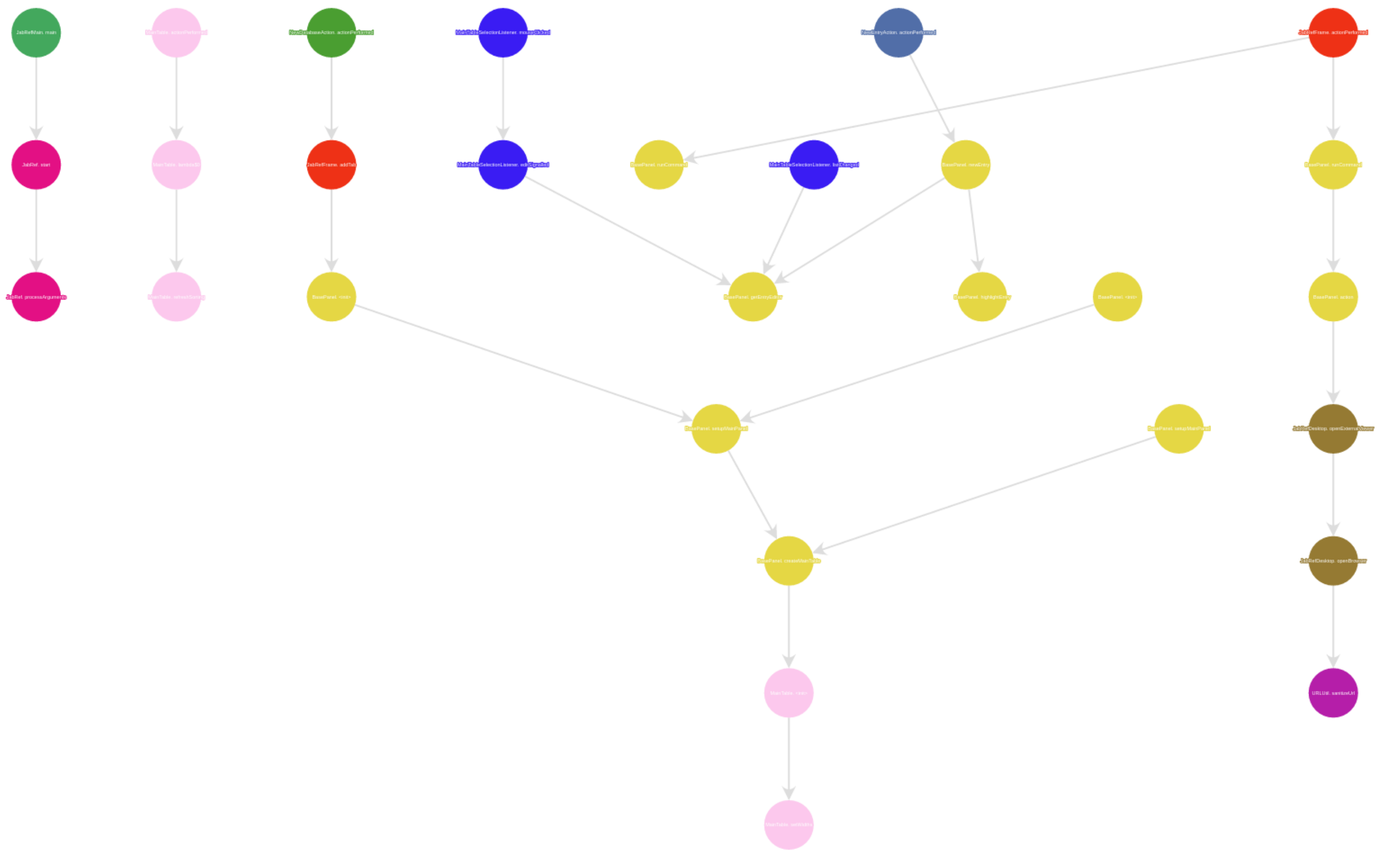
In this paper, we introduce the swarm debugging approach and the implemented infrastructure to support the swarm debugging, the Swarm Debug Infrastructure (SDI). SDI is an open source infrastrucre to collect and share interactive debugging activities. We conducted an experiment with three real maintenance task performed by seven developers on JabRef system. We aimed to evaluate how effective the data collected by SDI could be use to understand interactive debugging. We found that (1) there is not a correlation between the numbers of invocations and elapse task time ($\rho = -0.039$); (2) there is no correlation between numbers of breakpoints and elapse task time ($\rho = 0.093$); (3) developers follow different debugging patterns (4) there is no relation between numbers of breakpoints and expertise; (5) **whether developers toggle breakpoints carefully, they complete tasks faster than developers who toggle breakpoints quickly**.

The research community can leverage the SDI to conduct more studies to improve our understanding of developers' debugging activities, which could ultimately result into the development of whole new families of debugging tools that are more efficient and–or more adapted to the particularity of each debugging activity.

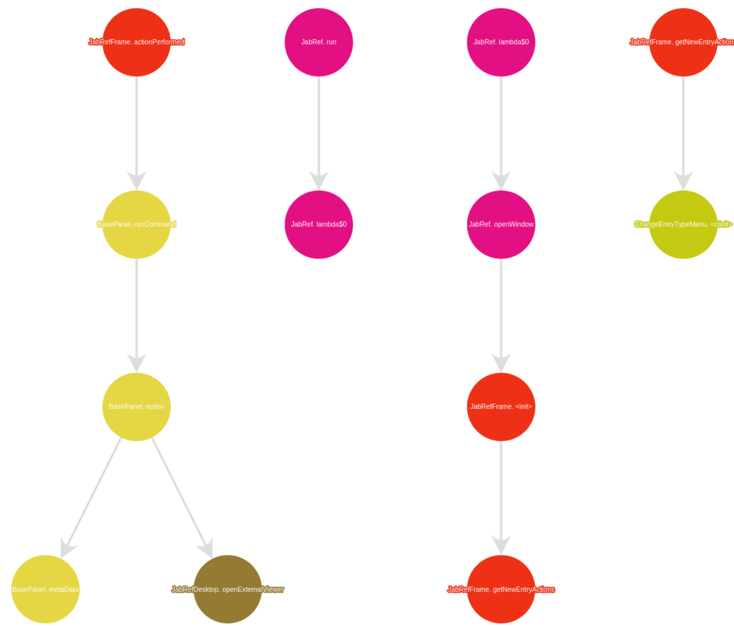
In future work, we plan to use SDI in real production environnement and survey developers about the usefulness of SDI. We would also ask the opinion of other developers of debugging tools (*e.g.,* Netbeans or gdb) to figure out whether SDI could be benefit to community of debugging tools and–or integrated with existing debugging tools.

Acknowledgment

We give our thanks to the programmers who participated in our case study, supporting and improving our work with several insightful ideas and discussions. This work has been partly supported by the Natural Sciences and Engineering

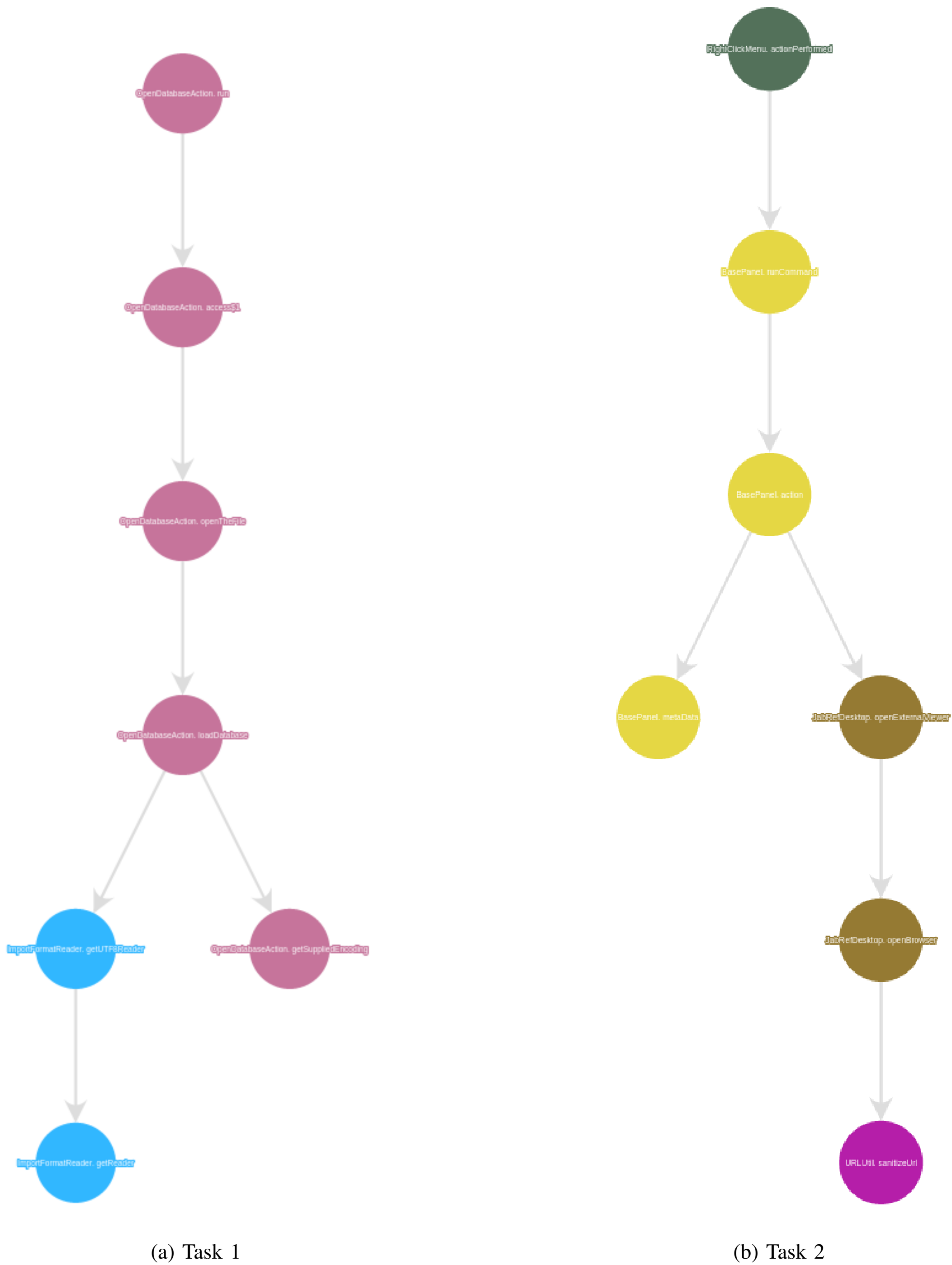


(a) Task 1



(b) Task 2

Fig. 9: Examples of fuzzy debugging patterns



(a) Task 1

(b) Task 2

Fig. 10: Examples of straight debugging patterns

Research Council of Canada and the Canada Research Chair on Patterns in Mixed-language Systems.

References

[1] A. S. Tanenbaum and W. H. Benson, "The people's time sharing system," *Software: Practice and Experience*, no. 2, pp. 109–119, apr.

[2] H. Katso, "Sdb: a symbolic debugger," in *Unix Prograammers Manual*, 1979.

[3] M. A. Linton, "The evolution of dbx," in *Proceedings of the Summer USENIX Conference*, 1990, pp. 211–220.

[4] Stallman, R. Pesch and S. Shebs, *Debugging with GDB - The GNU Source-Level Debugger*. GNU Press, 2002.

[5] P. Wainwright, "GNU DDD - Data Display Debugger," 2010.

[6] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, no. 12, pp. 971–987, dec.

[7] J. Rößler, "How helpful are automated debugging tools?" in *2012 1st International Workshop on User Evaluation for Software Engineering Researchers, USER 2012 - Proceedings*, no. Section V, 2012, pp. 13–16.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency," *ACM Transactions on Software Engineering and Methodology*, no. 1, pp. 1–38, dec.

[9] Ieee, "IEEE Standard Glossary of Software Engineering Terminology," p. 1.

[10] A. Blasciak and G. Parets, "System of debugging software through use of code markers inserted into spaces in the source code during and after compilation."

[11] T. Ball and S. K. Rajamani, "The slam project: Debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, no. 1, pp. 1–3, Jan. 2002.

[12] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2622669

[13] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transaction on Software Engineering*, vol. 32, no. 12, pp. 971–987, dec 2006.

[14] M. J. Coblenz, A. J. Ko, and B. A. Myers, "Jasper: An eclipse plug-in to facilitate software maintenance tasks," in *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '06, 2006, pp. 65–69.

[15] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July/August 2008.

[16] S. Wang and D. Lo, "Version history, similar report, and structure: putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*. New York, New York, USA: ACM Press, pp. 53–63.

[17] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun, pp. 14–24.

[18] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 689–699.

[19] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.

[20] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, 2015, pp. 251–260.

[21] A. Ying and M. Robillard, "The influence of the task on programmer behaviour," in *Proceedings International Conference on Program Comprehension*, june 2011, pp. 31–40.

[22] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study of the effect of file editing patterns on software quality," in *Proceedings Working Conference on Reverse Engineering*, 2012, pp. 456–465.

[23] Z. Soh, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, and B. Adams, "On the effect of program exploration on maintenance tasks," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 391–400.

[24] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. Nair, and D. Shepherd, "A practical guide to analyzing ide usage data," in *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015.

[25] T. M. Ahmed, W. Shang, and A. E. Hassan, "An empirical study of the copy and paste behavior during development," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, May 2015, pp. 99–110.

[26] K. Araki, Z. Furukawa, and J. Cheng, "A general framework for debugging," *Software, IEEE*, vol. 8, no. 3, pp. 14–20, May 1991.

[27] I. Katz and J. Anderson, "Debugging: An Analysis of Bug-Location Strategies," *Human-Computer Interaction*, no. 4, pp. 351–399, dec.

[28] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, no. 12, pp. 992–1009, dec.

[29] I. Zayour and A. Hamdar, "A qualitative study on debugging under an enterprise IDE," *Information and Software Technology*, pp. 130–139, feb.

[30] T. D. LaToza and B. a. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. New York, New York, USA: ACM Press, p. 185.

[31] G. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Elipse IDE?" *IEEE Software*, vol. 23, no. 4, 2006.

[32] M.-A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, no. 3, pp. 187–208, sep.

[33] R. Tiarks and T. Röhm, "Challenges in Program Comprehension," *Softwaretechnik-Trends*, no. 2, pp. 19–20, may.

[34] Wikipedia, "Collective intelligence."

[35] A. Fuggetta, "Software process: a roadmap," vol. 97, 2000, pp. 25–34.

[36] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini, "IDE 2.0: Collective Intelligence in Software Development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, 2010, p. 53.

[37] M.-a. Storey, L. Singer, B. Cleary, F. F. Filho, and A. Zagalsky, "The (R)Evolution of Social Media in Software Engineering," *FOSE*, 2014.

[38] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in Eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange - eclipse '07*, pp. 31–35.

[39] F. Beck, B. Dit, J. Velasco-madden, D. Weiskopf, and D. Poshyvanyk, "Rethinking User Interfaces for Feature Location," in *23rd IEEE International Conference on Program Comprehension*. Florence: IEEE Comput. Soc.

[40] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey, "Mining modern repositories with elasticsearch," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. Hyderabad, India: ACM, pp. 328–331.

[41] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks," *ACM Transactions on Software Engineering and Methodology*, no. 2, pp. 1–41.

[42] D. Piorkowski and S. Fleming, "The whats and hows of programmers' foraging diets," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems CHI '13*. Paris, France: ACM, pp. 3063—-3072.

[43] S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering approach combining metrics and program visualisation," *Reverse Engineering, 1999. ...*, no. Section 3.

[44] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guéhéneuc, "Understanding interactive debugging with swarm debug infrastructure," in *24rd IEEE International Conference on Program Comprehension*. Austin: IEEE Comput. Soc, 2016.

[45] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, pp. 108–124.