Understanding Interactive Debugging with Swarm Debug Infrastructure

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. In this paper, we present the Swarm Debug Infrastructure (SDI) with which researchers can collect and share data about developers' interactive debugging activities. SDI allows collecting and sharing debugging data that are useful to answer research questions about interactive debugging activities. We assess the effectiveness of the SDI through an experiment to understand how developers apply interactive debugging.**

I. Introduction

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 *gdb* [2] have been used by developers for decades. Modern debuggers are often integrated in development environments, *e.g.,* DDD [3] 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.

In this paper, we propose the Swarm Debug Infrastructure (SDI), an open-source infrastructure[1] integrated into Eclipse, which allows practitioners and researchers to collect and share fine-grained data about developers' interactive debugging activities. Through an experiment conducted with 10 participants and three real faults in JabRef, we show how the data collected using the SDI can improve our understanding of developers' debugging activities.

II. 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.

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

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

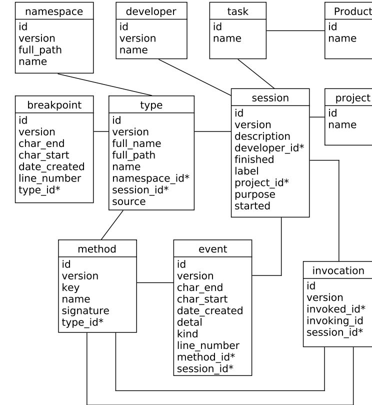


Fig. 1: The SDI metadata

A. Swarm Debug Services

The Swarm Debug Services (SDS) provide the infrastructure needed by the Swarm Debug Tracer (SDT) to store and, later, share debugging data from and between developers. The SDT sends RESTful messages that are received by a SDS instance that stores them. We choose and define domain concepts to model software projects and debugging data. Figure 1 shows the meta-model of these concepts using an entity-relationship representation. The concepts are inspired by the FAMIX Data model [4].

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.

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.

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-

[3]http://projects.spring.io/spring-boot/

[4]https://www.elastic.co/

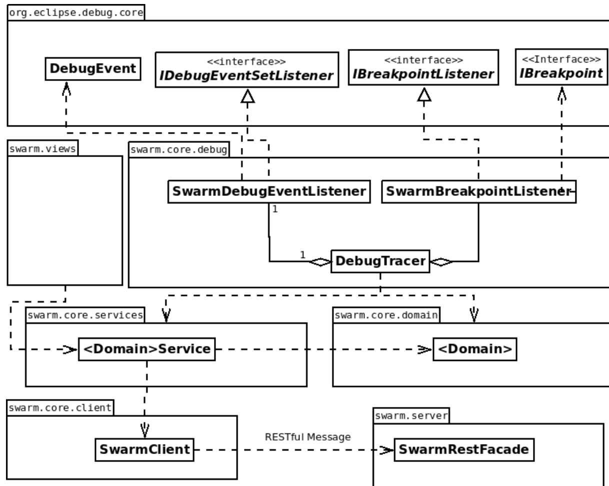


Fig. 2: The Swarm Tracer architecture

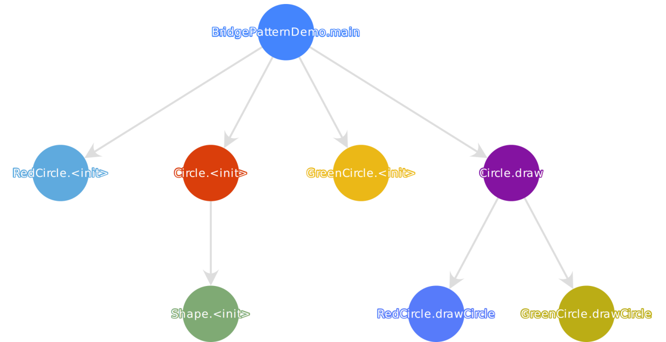


Fig. 3: Method call graph for Bridge design pattern

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.

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 2 shows the SDT architecture. After an authentication process, developers create a debug session and toggle breakpoints. Stepping events as *Step Into*, *Step Over* or *Step Return* are caught and stack trace items are analyzed by the Tracer, extracting method invocations.

Following the foraging approach [5], 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.

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 [6], as shown by Figure 3, 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 [7] 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.

In summary, through SDI, we provide a technique and model to collect, store and share debugging session information, contextualizing breakpoints and events during these sessions. We also provide a tool for visualizing context-aware debugging sessions using call graphs and a map where developers can see areas visited by his or her colleagues and know who has already explored a project. Using web technologies, we created real-time and interactive visualizations, providing an automatic memory for developer explorations. SDI also provides a tool for searching starting points and breakpoints for software projects based on shared session information collected by developers. Moreover, dividing software exploration by sessions and its call graphs are easy to understand because only intentional visited areas are shown on these graphs.

With SDI, one can flow through the execution of a program and see only the points that are relevant to developers, which is in sharp contrast with traditional visual debugging tools that provide only data about the whole execution and one must manipulate plenty of irrelevant paths.

III. Experimental Study with SDI

The SDI provides data collections, services, and tools for researchers to understand developers' interactive debugging activities. To evaluate the effectiveness of the SDI, we present a study that uses the SDI to collect data to answer three research questions. We now present the context, the design and the results of our experimental study.

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. For the sake of space and simplicity, to illustrate the use of the SDI, we formulate the following two research questions:

RQ1: How many breakpoints does a developer toggle by task?
RQ2: Do developers follow similar navigation patterns through method invocations during debugging?

B. Study Design

To answer the research questions, we proceeded as follows[5]:

1) Tasks Definition, Participants and Artifacts: We had to choose debugging tasks to trigger participants' debugging activities. We chose to ask participants to find the locations of real faults in an independent, open-source program. We selected JabRef[6] 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 randomly picked three 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, and 669.

As many other experimental studies before ours, we asked volunteers among our undergraduate and graduate students at Polytechnique Montréal to participate in our experimental study. After sending a general call for volunteers, 10 students volunteered (80% male, 20% female). They were all experts or advanced developers (70%). They all used IDEs (70%) and debuggers (60%) frequently.

We provided participants with instructions by email and in two documents. The first document 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[7].

The second document presented the three 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 external data about their debugging activities, including their numbers of years of programming experience, their undergraduate or graduate program, their level of tiredness, and others.

For this experimental study, we used Eclipse Mars and Java 8, the SDI and its Swarm Debug Tracer plug-in, and two Java projects: an in-house 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.

2) Data Collection and Analysis: 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 fault. All participants completed this task successfully (mortality rate 0%). After the warm-up task, each participant executed debugging sessions to find the locations of the faults described in the three 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 three tasks successfully or not. All services were available on our server[8] during this debugging sessions and the experimental data were collected in the course of seven days.

After the 10 participants completed the debugging sessions (successfully or not), we used the tools provided by the SDI on the data collected by the SDI to answer each research question. To answer RQ1, we use the following SQL query, with which we can extract all the breakpoints set during each session and find a relationship between breakpoints and tasks:

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

Finally, to answer RQ2, 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.

C. Results

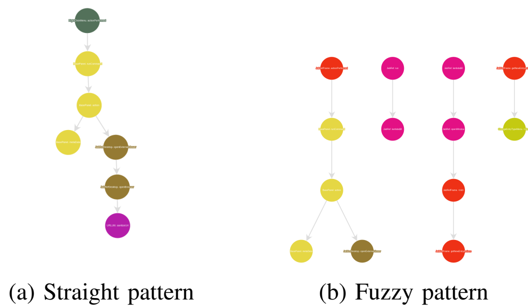
RQ1: How many breakpoints does a developer toggle by task?

Applying the analysis described previously, we conclude that participants toggled between one or two breakpoints by task, independently of success or failure of locating the source of the faults and independently of the complexity of the tasks. We explain this result by the relative simplicity of the chosen program, its domain, and its issues. Future works are necessary to replicate this experimental study to confirm or infirm this observation.

RQ2: Do developers follow similar navigation patterns through method invocations during debugging?

By visualising the call graphs generated using the SDI, we observed two distinct debugging navigation patterns: (1) a *fuzzy* debugging pattern and (2) a *straight* debugging pattern. In the fuzzy debugging pattern, the method invocation graph presents several branches, showing that participants adopted an

[5]All artifacts, tutorials, raw data, survey forms, and spreadsheets are available at http://swarmdebugging.org/publications/icpc2016.
[6]http://www.jabref.org/
[7]https://youtu.be/U1sBMpfL2jc
[8]http://server.swarmdebugging.org



(a) Straight pattern (b) Fuzzy pattern

Fig. 4: Examples of straight and fuzzy debugging patterns

exploratory approach. Figure 4b shows two typical fuzzy debugging graphs. In the straight debugging pattern, participants followed a straight or quasi-straight set of method invocations to achieve their tasks, as shown in Figure 4a.

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.

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 10 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/icpc2016.

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.* [8].

IV. Conclusion

In this paper, we introduce a novel approach for debugging (*i.e.,* swarm debugging) and propose the Swarm Debug Infrastructure (SDI), an open-source infrastructure integrated into Eclipse, to collect and share fine-grained data about developers' interactive debugging activities. The SDI collects data in the background during debugging activities without altering the performance of the IDE. The SDI stores data on a remote server using an asynchronous execution and, thus, does not suffer from performance or memory issues as could omniscient debuggers [9] or tracing-based approaches [10].

We described the SDI and showed through an experiment that we can use the SDI to collect and share data about the developers' debugging activities while fixing real faults. The experiment pertained to three real faults in JabRef and collected data from 10 participants. We used the collected data to answer two research questions showing that the SDI collect data relevant to the developers' debugging activities. From the research questions we observed that (1) developers toggled only one or two breakpoints by activity; (2) there is no correlation between numbers of breakpoints and developers' expertise; (3) there is no correlation between the numbers of breakpoints and task complexity, and (4) developers follow different debugging patterns.

This paper is only a first step towards fully understanding debugging activities. In future work, we plan to extend the SDI into a full context-aware debugger, adding new ways to share and visualise debugging sessions. We also plan to integrate the SDI with a bug tracking system, improve the breakpoint search, associate issues tracking information with breakpoints. Finally, we will perform new empirical studies on developers' debugging activities using the SDI.

Acknowledgment

We thank 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 Research Council of Canada and the Canada Research Chair on Patterns in Mixed-language Systems, and CNPq (Conselho Nacional de Desenvolvimento Cientfico e Tecnolgico), Brazil.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] G. Pothier and É. Tanter, "Back to the Future: Omniscient Debugging," *IEEE Software*, no. 6, pp. 78–85, nov.

[10] P. Ohmann and B. Liblit, "Lightweight control-flow instrumentation and postmortem analysis in support of debugging," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov, pp. 378–388.