Swarm Debugging: towards a shared debugging knowledge

Fabio Petrillo[1], **Guilherme Lacerda**[1,2], **Marcelo Pimenta**[1] **e Carla Freitas**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

[2]Faculdade de Informática - Centro Universitário Ritter dos Reis (Uniritter)

{fspetrillo,gslacerda,mpimenta,carla}@inf.ufrgs.br

Abstract. *Debugging is a tedious and time-consuming task since it is a methodical process of finding causes and reducing the number of errors. During debugging sessions, developers run a software project, traversing method invocations, setting breakpoints, stopping or restarting executions. In these sessions, developers explore some project areas and create knowledge about them. Unfortunately, when these sessions finish, this knowledge is lost, and developers cannot use it in other debugging sessions or sharing it with collaborators. In this paper, we present Swarm Debugging, a new approach for visualizing and sharing information obtained from debugging sessions. Swarm Debugging provides interactive and real-time visualizations, and several searching tools. We present a running prototype and show how our approach offers more useful support for many typical development tasks than a traditional debugger tool. Through usage scenarios, we demonstrate that our approach can aid developers to decrease the required time for deciding where to toggle a breakpoint and locate bug causes.*

1. Introduction

Developers usually spend over two thirds of their time investigating code – either testing or doing dynamic investigation using a debugger or reading, statically following methods' calls or using other source browsing tools [LaToza and Myers 2010]. Since debugging is a tedious and time-consuming task[Fleming et al. 2013], in the last 30 years, numerous approaches and techniques have been proposed to help software engineers in debugging. However, recent work showed that empirical evidence of the usefulness of many automated debugging techniques is limited, and they are rarely used in practice[Parnin and Orso 2011]. Thus, if developers spend a lot of time debugging code, why should this human effort be lost? Could we collect debugging session information to use in the future, creating visualizations and seaching tools about this information? Why is a breakpoint toggled? What is its purpose?

To address these debugging issues, we claim that *collecting and sharing debugging session information can provide data to create new visualizations and searching tools to support programmers tasks, decreasing the time for developers deciding where to toggle the breakpoints as well as improving project comprehension.* To support this statement, we propose a new approach named Swarm Debugging. Swarm Debugging is a technique to collect and share debugging information, and to provide visualizations and searching tools for supporting the debugging process.

The remainder of the paper is structured as follows. Section 2 describes Swarm debugging, its structure, visualizations and searching tools. Section 3 discusses its results

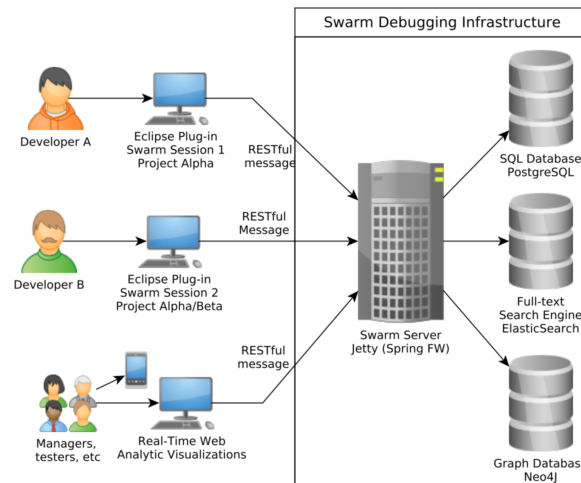

Figure 1. The Swarm Debugging Infrastructure

and main contributions compared to related work, and in Section 5 we draw some final comments and present future work.

2. The swarm debugging approach

Swarm debugging (SD) is an approach inspired by three challenges in software engineering: the time and effort spent during debugging sessions, the difficulties of deciding where to set a breakpoint, and how collective intelligence can be used to improve software development. Firstly, developers spend long periods in debugging sessions, finding bugs or understanding a software project [LaToza and Myers 2010]. During these sessions, while using traditional debugging tools, they collect a lot of information and create a mental model of the project [Murphy et al. 2006]. Unfortunately, several studies have shown that developers quickly forget details when they explore different points within the source code, loosing parts of the mental model [Tiarks and Röhm 2013]. The second challenge is finding suitable breakpoints when working with the debugger [Tiarks and Röhm 2013]. Developers have to find the adequate breakpoints in order to reproduce the data and control flow of the program. Tiarks and Röohms [Tiarks and Röhm 2013] observed that developers encountered problems to find the appropriate breakpoints, because deciding where to toggle a breakpoint is an "extremely difficult" task. They found that programmers applied a strategy: they set a lot of breakpoints in the beginning, then they start the debugger, and after that, they remove the breakpoints that turned out to be irrelevant. They noticed some programmers that wrote notes on a piece of paper or used an external editor as a temporary memory. Moreover, the study suggested that this strategy causes significant overhead. When a developer does not know exactly where an error is located, he or she chooses a breakpoint as a starting point or a line of code near the point that is an hypothesis for this error. Thus, this point is usually an strategic area in the software, and this knowledge is lost in the current debugging tools. The third challenge is how collective intelligence can be used in software development. Software development is a cooperative effort [Fuggetta 2000], and Storey et al. [Storey et al. 2014] claim that collective intelligence is an open field for new software development tools.

Swarm Debugging is based on three works: 1) the declarative and visual debugging environment for Eclipse called JIVE[Czyz and Jayaraman 2007], 2) a novel user in-

terface to support feature location named In Situ Impact Insight (I3) [Beck et al. 2015], and 3) the Kononenko et al.'s experience with ElasticSearch for performing modern mining software repositories research [Kononenko et al. 2014]. Firstly, JIVE is an Eclipse plug-in that is based on the Java Platform Debugging Architecture (JPDA), which allows the analysis of a Java program execution. JIVE requests notification of certain runtime events, including class preparation, step, variable modification, method entry and exit, thread start and end. Our approach also uses JPDA to collect debug information. Secondly, I3 [Beck et al. 2015] introduces a novel user interface, which allows developers to retrieve details about the textual similarity of a source code and to highlight code in the editor, as well as augment it with *in situ* visualizations. These visualizations also provide a starting point for following relations to textually similar or co-changed methods. Swarm Debugging uses textual similarity in its searching tools on software projects repositories. Finally, as Kononenko et al. [Kononenko et al. 2014], we use ElasticSearch to create a powerful search mechanism for the collected data during the debugging sessions, providing a shared information retrieval system for software developers. So, as debugging is a foraging process [Fleming et al. 2013], the Swarm Debugging main idea is to collect information during the sessions, storing breakpoints, reachable paths and debugging behaviors. Finally, these data are transformed into knowledge by visualizations and searching tools, creating a collective intelligence about software projects. In the next sections, we describe how Swarm Debugging implements this idea.

2.1. Architecture and infrastructure

Integrated Development Environments (IDEs) are widely used tools to develop software [Minelli et al. 2014]. We chose Eclipse[1] to implement our prototype. The Java Development Tools (JDT) is an important component of Eclipse, which includes debugging support in the JDT Debug module. Eclipse uses JPDA in order to provide traditional debugging capabilities such as setting of breakpoints, stepping through execution, or examining variables in the call stack. The Swarm Debugging prototype (Figure 1) is an Eclipse Plug-in, which captures Java Platform Debugging Architecture (JDPA) events in a debugging session.

When a developer creates a session, the SD starts two listeners, and it uses RESTful messages to communicate with a Swarm Server Infrastructure. These messages are received by a Swarm Server instance that storages them in three specialized persistent components: a SQL database (PostgreSQL), a full-text search engine (ElasticSearch) and a graph database (Neo4J). So, the Swarm Server provides an infrastructure for sharing information between developers, searching tools, and, as a web container (Jetty), for creating our visualizations (see section 2.2). During the debugging session, for each breakpoint or *Step Into* event, stack trace items are analysed by a tracer, extracting method invocations. For each pair of invocation, the tracer obtains an invoking and invoked method, creating an invocation entry.

2.2. Visualization techniques and searching tools

Swarm Debugging (SD) implemented the visualizations using the CytoscapeJS [Saito et al. 2012], a JavaScript Graph API framework. As a web application, the SD

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

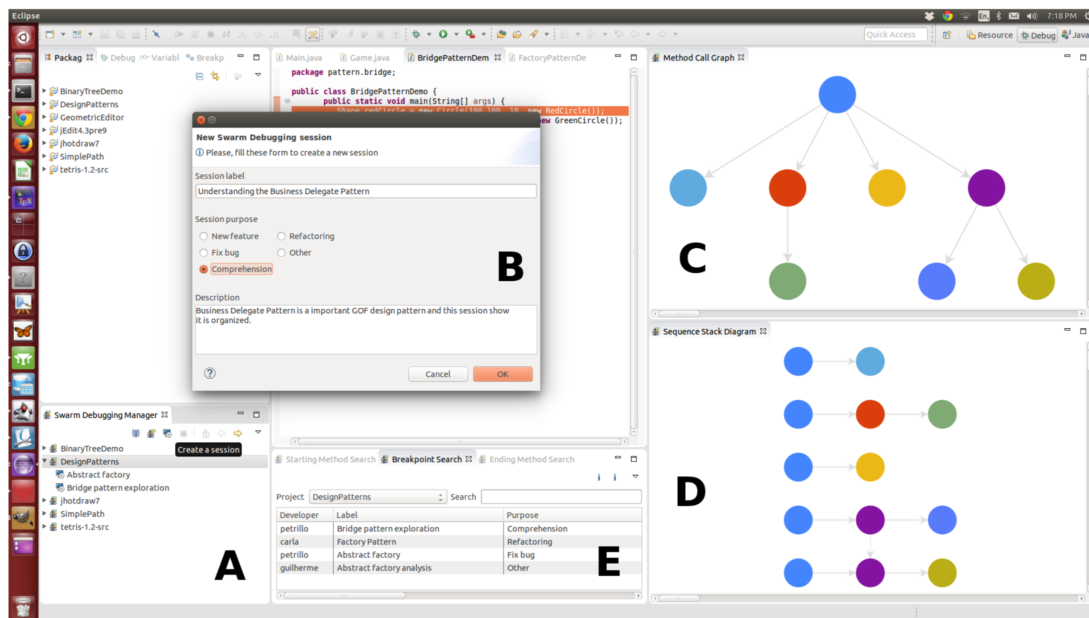

Figure 2. Swarm Debugging View

visualizations can be integrated into the Eclipse view as a SWT Browser Widget, or accessed through a traditional browser as Mozilla Firefox or Google Chrome, or web browsers for tablets and smartphones. Using these technologies, SD provides several visualizations about collected information: 1) Sequence stack diagram; 2) Dynamic method call graph; and 3) Swarm dashboard. Figure 2 shows the SD interface.

2.2.1. Sequence stack diagram

The sequence stack diagram is a novel way to represent a sequence of methods invocations (Figure 2-D). We use circles to represent methods and arrows to represent invocations. Each line is a complete stack trace, without returns. The first node is a Starting point (non-invoked method), and the last node is an Ending point (non-invoking method). If the invocation chain continues to a non-starting point method, a new line is created, the current stack is repeated, and a dotted arrow is used to represent a return for this node. In addition, developers can directly go to a method in the Eclipse Editor by double-clicking on its node in the diagram.

2.2.2. Dynamic method call graph

The dynamic method call graph is based on directed call graphs [Grove et al. 1997] (Figure 2-C) for explicitly modeling the hierarchical relationship induced by invocated methods. This visualization uses circles to represent methods (nodes) and oriented arrows to express invocations (edges). Each session generates a call graph and all invocations collected during the session are shown in this visualization. The starting points (non-invoked methods) are represented at the top of a tree, and the adjacent nodes represent the invocation sequence. In addition, the developer can navigate along the sequence of invocation

methods by pressing the F9 key (forward) or the F10 key (backwards). Finally, developers can go directly to a method in the Eclipse Editor by double-clicking on its node in the diagram.

2.2.3. Swarm dashboard

ElasticSearch technology brings us a powerful dashboard tool named Kibana. Using Kibana, we created an online panel to display project information. Several charts can be built to show the number of invocations, breakpoints or events by developers; the number of breakpoints by project and purpose; number of events by minute, etc.

Swarm Debugging also provides several searching tools for developers. To allow finding suitable breakpoints [Fleming et al. 2013] when working with the debugger, we developed a search tool. For each breakpoint toggle, SD captures the type and line of code where the breakpoint was toggled, storing this in the SD infrastructure databases. This way, all developers can share their breakpoints. The breakpoint search tool (Figure 2-E) combines *fuzzy*, *match* and *wildcard* ElasticSearch query strategies. Results are displayed in the search view table, which allows an easy selection. Finally, developers can open a type directly in Eclipse Editor by double-clicking on a selected breakpoint. The last search tool is a full-text search into the project source code. Eclipse IDE makes available a search tool, but it doesn't have full-text search features such as those provided by ElasticSearch. So, our source code search is a complementary tool for developers.

2.3. Swarm Debugging usage

This section describes the steps for using our approach. Firstly, using the Eclipse IDE, developers open a view “Swarm Manager", and establish a connection with the Swarm server (Figure 2-A). If the target project is not in the Manager, they associate a project from their workspaces to the Swarm Manager. Secondly, they create a Swarm session (Figure 2-A), informing a label, debugging purpose and a description for the new session (Figure 2-B). Automatically, the Eclipse Perspective is switched to *Debug*, and the visualizations *Method Call Graph* and *Sequence Stack Diagram* are shown (Figures 2-C and 2-D).

Then, with a session established, developers may need to toggle some breakpoints. For supporting this task, developers use breakpoint, starting point, or source search tools to find the correct location for their goal (Figure 2-E). Swarm Debugging captures and stores all breakpoints toggled by the developers. After adding all breakpoints, they start a conventional debugging execution, stopping in the first reached breakpoint. For each *Step Into* or *Breakpoint* event, SD DebugTracer captures the event and stores its method call, producing invocations entry for each pair invoking/invoked method. Automatically, the method call graph and sequence stack diagram visualizations are updated by the views' mechanisms.

The process continues until the execution is finished, when the Swarm Debugging session is completed. Finally, all collected information is made available for developers through visualization of the resulting graphs or breakpoints.

3. Discussion

During the SD implementation, we analyzed many usage scenarios, and we can summarize some practical observations. The breakpoint search tool is powerful and useful. After recording many breakpoints in the database, new debugging sessions were started much quickly because we found the significant breakpoints easily. In certain situations, although some breakpoints were not the exact points that we wanted, they were good points to start. So, the breakpoint search tool decreases the time spent to toggle a breakpoint.

Swarm sessions showed an important feature to divide the software complexity problem. In traditional approaches, as the JIVE approach, all data are used for visualizations and search. In addition, when the session finishes all data are lost. On the other hand, Swarm Debugging saves session information, and developers can divide massive projects in multiple diagrams, having a fine control to explore relevant or important portions of their software project. This feature improves the overall software project comprehension.

Nevertheless, it would be possible to argue that debugging information in different parts of the software would not be interesting to share. However, we believe that in certain situations, a developer can take advantage of his colleague's knowledge who has explored an unknown area before him. The problem to be solved might not be the same, but may be in the same region of the project, and knowing where some breakpoints were placed before (by himself or by other developers) could accelerate a decision of where to put another ones. However, since this is really a new proposal, for sure we need to evaluate it, and this is immediate future work.

One of the most important innovations in our proposal is that Swarm Debugging creates diagrams during a debugging session, and makes specific graphs for an execution. Furthermore, if we carefully observe a diagram, the redundancy for each stack line allows the developer to use zooming or panning to visualize exactly a sequence needed for understanding the software. We compared the sequence stack diagram with the UML sequence diagram, and our proposal is better for large sequences (we used JIVE to do this comparison). However, of course, we need to evaluate this experimentally to validate the hypothesis of usefulness of our approach.

4. Related work

In recent years, researchers have worked on improving debugger tools to address developers' issues. Our work is related to several aspects of software development, including debugging techniques, visual debugging, databases, and collective intelligence.

Debugging Canvas [DeLine et al. 2012] is a tool where developers debug their codes as a collection of code bubbles, annotated with call paths and variable values, on a two-dimensional pan-and-zoom surface. SD uses a similar idea, but it integrates source code and visualization in two distinct views. Estler et al. [Estler et al. 2013] discussed the *Collaborative Debugging* describing CDB, a debugging technique and integrated tool designed to support effective collaboration among developers during shared debugging sessions. Their study suggests that debugging collaboration features are perceived as important for developers, and can improve the experience in collaborative context. This result founded our approach, but differently of CDB (a synchronous debugging tool), SD is an asynchronous session debugging tool. Consequently, SD does not have several

collaboration issues. Minelli et al. [Minelli et al. 2014] presented a visual approach for representing development sessions based on the finest-grained UI events. The authors have collected, visualized, and analysed several development sessions for reporting their findings. SD is based on capturing UI events, but related to breakpoints.

5. Conclusions

In this paper, we have presented a new approach called Swarm Debugging aimed at visualizing and sharing information obtained from debugging sessions, and we have presented a novel tool as an Eclipse plugin. Our approach uses the developers workforce to create a context-ware visualization, automatically capturing and sharing knowledge with its context by observing developers' interactions that were discarded in traditional debugging tools. It allows programmers to find quickly breakpoints, starting points and *share their debugging experiences* about projects. By focusing context-aware sessions, each swarm debugging session captured only the **intentional exploration path**, focusing visualizations and searches on developer issues.

Our approach provides several contributions. First, a technique and model to collect, store and share debugging session information, contextualizing breakpoints and events during these sessions. Second, a tool for visualizing context-aware debugging sessions using call graphs and a map where developers can check areas that were visited by collaborators, and to know who already explored a project. Using web technologies, we created real-time and interactive visualizations, providing an automatic memory for developer explorations. Third, 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 graph are easy to understand because just intentional visited areas are shown in the graph. So, Swarm Debugging is used to support developers in their everyday work and on how software comprehension features can be integrated into the workflows depending on the task. With swarm debugging developers can follow the execution and see only the points that are relevant to programmers. In contrast, when using traditional visual debugging, one must see whole executions and manipulate plenty of irrelevant paths.

Despite the promising results we obtained, this paper is a preliminary work towards new ways to share and visualize information about debugging sessions. As future work, first, we plan to extend our catalog of visualizations and enlarge our dataset of debugging sessions. Second, we plan to integrate our tool with a bug tracking system, improving the breakpoint search, associating issues track information with breakpoints. Finally, we will perform new empirical studies to test our approach.

Acknowledgments This work has been partially supported by CNPq.

References

Beck, F., Dit, B., Velasco-madden, J., Weiskopf, D., and Poshyvanyk, D. (2015). Rethinking User Interfaces for Feature Location. In *23rd IEEE International Conference on Program Comprehension*, Florence. IEEE Comput. Soc.

Czyz, J. K. and Jayaraman, B. (2007). Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange - eclipse '07*, pages 31–35.

DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., and Reiss, S. P. (2012). Debugger Canvas: Industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073. IEEE.

Estler, H. C., Nordio, M., Furia, C. a., and Meyer, B. (2013). Collaborative debugging. *Proceedings - IEEE 8th International Conference on Global Software Engineering, ICGSE 2013*, pages 110–119.

Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. (2013). An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology*, 22(2):1–41.

Fuggetta, A. (2000). Software process: a roadmap. volume 97, pages 25–34.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call graph construction in object-oriented languages. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, pages 108–124.

Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. (2014). Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 328–331, Hyderabad, India. ACM.

LaToza, T. D. and Myers, B. a. (2010). Developers ask reachability questions. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:185–194.

Minelli, R., Mocci, A., Lanza, M., and Baracchi, L. (2014). Visualizing Developer Interactions. In *2014 Second IEEE Working Conference on Software Visualization*, pages 147–156.

Murphy, G., Kersten, M., and Findlater, L. (2006). How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4).

Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis ISSTA 11*, page 199.

Saito, R., Smoot, M. E., Ono, K., Ruscheinski, J., Wang, P.-l., Lotia, S., Pico, A. R., Bader, G. D., and Ideker, T. (2012). A travel guide to Cytoscape plugins. *Nature methods*, 9(11):1069–76.

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

Tiarks, R. and Röhm, T. (2013). Challenges in Program Comprehension. *Softwaretechnik-Trends*, 32(2):19–20.