Visualizing sequences of debugging sessions using Swarm Debugging

Eduardo A. Fontana, Fabio Petrillo
Laboratoire d'Informatique Formelle
Universite du Quebec a Chicoutimi
Chicoutimi, Canada
fontanadnb@gmail.com, fabio@petrillo.com

***Abstract*—In Software Engineering, one of the most important activities is debugging. Debugging is a set of techniques to detect, locate, and correct faults in a computer program. Modern *Integrated Development Environments* (IDEs), such as *Eclipse* or *Visual Studio*, provide infrastructure to support interactive debugging, during which a developer explores the source code of the system under development or maintenance. Although IDEs encourage developers to work collaboratively, debugging is still an individual activity. Furthermore, interactive debugging activity is limited by IDE debugging features that do not store previous debugging sessions. This condition forces developers to repeat debugging execution sessions to review the debugging information. In this paper, using the concept of *Swarm Debugging*, we present the Sequence Debugging Session View (SDV) tool. The primary goal is to capture the debugging information from a developer IDE (as *Visual Studio*) and store it. Then, the tool enables developers to retrieve the data in 3D interactive visualization and understand software behavior through the analysis and sharing of debugging session data. The main contribution of the tool is to assist on program comprehension and to reduce effort during software maintenance. To validate the solution, we performed two usage studies in real situations at a software house. The feedback from the evaluation of the tool suggests that the team could be helped on the software arrangement.**

***Index Terms*—Debugging, software visualization, swarm debugging, software comprehension, threejs, 3D visualization.**

I. Introduction

Debugging a software is one of the main activities of software developers. Developers debug code to find bugs, to test changes or to understand a software system [1]. Most of the *Integrated Development Environments* (IDEs) have an integrated debugger infrastructure that allows developers to explore a software line by line, inspect variables, as well as other features. However, when a debugging session is finished, the data from the session, such as methods, variables, stepping code and stack traces, are lost. It happens because the IDEs do not store the debugging information along the debugging sessions. This condition forces the developers to repeat the debugging execution to review the debugging information. Furthermore, recurring software problems sometimes require similar debugging sessions, which can be performed at different times by different developers, recreating the understanding of the system architecture from scratch.

In this paper, we present **Sequence Debugging Session View** (SDV), a visualization tool based on *Swarm Debugging* [2] that captures the debugging data from an IDE debugging infrastructure, storing it in a database, and promotes the recovery of this data through 3D visualization. The SDV is an interactive diagram inspired by the UML sequence diagram [3] that shows a temporal sequence of events (stepping and breakpoints) during multiple debugging sessions, allowing the developer to view and replay a debugging.

The main contribution of this paper is to present a visualization tool that improves developers understanding of a software system based on previous debugging sessions. The tool recovers and shares debugging data over time, besides replaying debugging execution. The debugging data history and the replay feature provide insights to help developers learn about software behavior.

This paper is outlined as follows: Section II presents the *Swarm Debugging* concepts. Section III details the tool, being Subsection III-A to present the 3D visualization elements, and Subsection III-B to show how the solution is structured. Then, Section IV demonstrates two real cases where the tool was used. Section V shows the users opinion and a discussion about the issues that were raised. Section VI contains the conclusion.

II. Background

A. Debugging and Interactive Debugging

Debugging is a process where developers make hypotheses about the root cause of a problem or defect and verify these hypotheses by examining different parts of the source code of the program. *Interactive debugging* consists of using a tool, *i.e., a debugger*, to detect, locate, and correct a fault in a program. It is a process also known as *program animation*, *stepping*, or *following execution* [4]. Developers often refer to this process simply as *debugging*, because several IDEs provide debuggers to support it. However, it must be noted that while *debugging* is the process of finding faults, *interactive debugging* is one particular debugging approach in which developers use interactive tools. Expressions such as *interactive debugging*, *stepping* and *debugging* are used interchangeably, and there is not yet a consensus on what is the best name for this process.

Generally, breakpoints allow pausing intentionally the execution of a program for debugging purposes. It is a means of acquiring knowledge about a program during its execution, for example, to examine the call stack and variable values when the control flow reaches the locations of the breakpoints. Thus,

a breakpoint indicates the location (line) in the source code of a program where a pause occurs during its execution.

B. Swarm Debugging

Swarm Debugging (SD) [2], [5] is an approach that uses swarm intelligence applied to interactive debugging data to create knowledge to support software development activities. SD works by (1) capturing debugging contextual information, (2) sharing it, and (3) reusing it across debugging sessions and developers. Swarm Debugging emerges from a context where many developers, performing debugging sessions independently, are in fact building collective knowledge, which can be shared and reused with adequate support. To provide such support, Swarm Debugging includes Swarm Debugging Infrastructure (SDI), with which practitioners and researchers can collect and share data about developers' interactive debugging sessions.

III. The Tool

A. The 3D visualization

Sequence Debugging Session View (SDV) is inspired by the UML sequence diagram [3]. The view is composed of a group of objects that represent the sequence diagram disposed in a 3D structure. Our visualization uses the 3D axes as follows. First, the Y axis represents the temporal sequence of events (breakpoint hitting and stepping) in a debugging session. Second, the X axis represents the sequence of *StepInto* calls during the session. Finally, the Z axis represents the temporal sequence of session debugging to a task. *Starting point* (1) indicates where the debugging flow starts. The *Base* (2) is the main architecture stack to which the project is associated, in this case .NetFramework. The *Breakpoint* (3) is a sphere that represents the breakpoint added to the IDE line of code, positioned in the sequence of the debugging events. The *Pathnode* (4) represents a sequential path during an interactive debugging session, i. e., the checkpoints called nodes through which the debugging session went. The nodes are composed by events, such as *Step events* (5), i.e. *StepInto*, *StepOver* and breakpoint addition. *Types* (6) represent a type, frequently associated with an object or a procedure file. The height of the *types* expands according to the number of events it represents in the debugging sessions. The *Group of types* (7) is a representation of type grouping from the same directory, package or project.

B. Architecture

We built the SDV solution in two parts. The first one is a *Visual Studio Extension* written in *C#*. The module is based on Swarm Debugging Infrastructure [5] and it is responsible for collecting data from debugging sessions of the current project and sending it over the *Internet* to the *Web Portal*. The second is a *Web Portal*, also in *C#*, that serves two purposes. It is responsible for receiving the debugging session data through a *Web Service* and storing it in a database. Besides that, through the *Front-End* solution built in *Three.js* and *JavaScript*, the *Web Portal* allows the developers to retrieve the collected data.

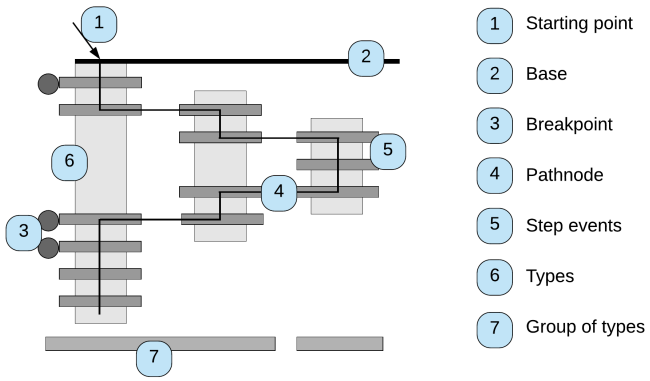


Fig. 1. Visualization elements schema

After retrieving it, the developers can filter the tasks, projects and session sequences. Then, the SDV shows the debugging sessions data sequenced in a 3D interactive visualization. In Fig. 2, we present the high level architecture of SDV, the extension and the server solution with the *Front-End* 3D Visualization.

The SDV *Front-end* 3D visualization part is a web application in *JavaScript* witch allows the developers of a team to select the session data of the current work task and analyze it. As shown in Fig. 6, the left part is a filter (1) that enables to select debugging sessions by task in the project and by the developers involved. The sessions to select are presented in descending order. In each one the amount of breakpoints and events referred on the view is shown. Breakpoints are displayed in red spheres and events in green spheres. Beside them, there is an eye icon that enables hiding or showing the session loaded on the visualization. Next to the filter is the view tool box (2). The tool box allows interactive manipulation of the visualization, such as resetting the camera, moving objects closer, resizing the scale, hiding or showing elements, and changing the color pallete and the background color. The third section is the 3D Visualization (3), composed by loaded sessions. That visualization is interactive, making it possible to zoom it in and out, read the session debugging element information by putting the mouse over it, and select other elements, such as the types, events or breakpoints. The last one is the source code (4) associated with the selected type, event or breakpoint. This section shows the exact source code captured when and where the debug cursor occurred in the IDE. Still in this section it is possible to replay the sequence of events in the same order they occurred, because the cursor on the source code is displayed synchronously on the 3D visualization. Thus, it is possible to revisit the debugging session after it occurred.

The SDV is an open source tool and is available on GitHub (https://github.com/eduardoafontana repositories *SwarmClientVS* and *SwarmServerAPI*), and a video showing the usage study can be found on YouTube (https://www.youtube.com/watch?v=YkXaiD60OFU).

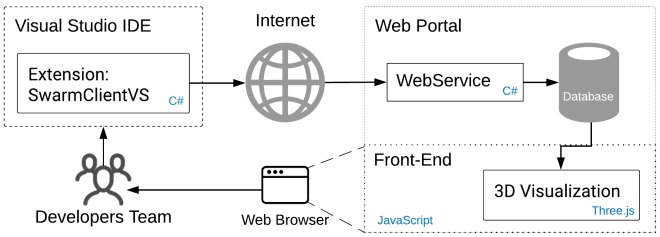


Fig. 2. SDV high level architecture

IV. Case Studies

To evaluate our visualization tool, we performed two case studies. The first case consisted of an issue at a software house. Using SDV, the issue was investigated by a senior developer. After that, a junior developer solved the issue. The second case consisted of behavior insights from a research analysis obtained from inspecting the sessions data collected in typical issue-related tasks at the software house.

A. Case 1 - Capturing session data and inspecting a bug

*a) **Issue context**:* The experiment consisted of applying the SDV to an issue in a software house and dividing the work into two steps. On the first phase, a senior developer analyzed the bug by capturing session data through SDV. On the second phase, a junior developer got the session data from SDV and tried to fix the issue. The software is a legacy system built on .Net Framework 3.5 with *web forms*, which repeatedly brings bugs. Because of that, the team considered the possibility of this bug having been already fixed.

*b) **Usage**:* On the first phase, the senior developer received the issue, prepared the conditions to run the system and then started debugging using SDV. The developer had a recent history with fixing bugs in this system. Hence, he suspected which file to add the breakpoints to and debug. He was able to replay the bug. He collected two sequences of debugging sessions in SDV, as shown in Fig. 3.

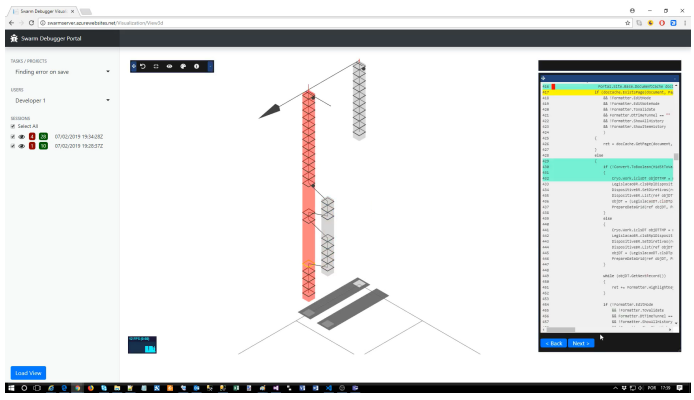


Fig. 3. Visualization of debugging sessions

On the second phase, the junior developer received the same issue and the session data collected by the senior developer. The junior developer saw the debugging sessions on SDV, and he inspected the source code by clicking on the first breakpoint bullet on the debugging path. After that, he replayed the session in SDV and highlighted the methods which contained the breakpoints and events. **Then, he added a breakpoint to the project on the same lines that he had observed on the visualization.** After debugging once, he did not find the cause of the bug, but could simulate it. After inspecting the visualization one more time, he found a breakpoint in a method that had not yet been observed. He put the breakpoint in this method inside the project and started debugging. When the method was hit, he stepped into a method that had not yet been explored, not even by the senior developer. This method was inside another class where he found the cause of the issue, as shown in Fig. 7.

*c) **User's feedback**:* We received the feedback from the junior developer. He said that, based on the 3D visualization and the correspondent interactive replay in the source code, it was possible to infer where, in the file and the class, to set the breakpoint to try to replay the issue. However, this knowledge only provided the means to find the issue, not exactly the place where the issue would be, because its cause was in another class, deeper. He said that the senior developer could have performed some more debugging sessions. **The junior developer said that the replay function helped him understand the stack trace that was being performed by the system when the *Save button* was pressed (the issue happened when that button was pressed in the system)**.

B. Case 2 - Analyzing captured sessions and inducing insights

*a) **Context**:* In this case study, we present some behavior insights that a researcher obtained while inspecting and analyzing data sessions collected from some debugging sessions using SDV in maintenance of legacy systems at the software house.

*b) **First Insight**:* By analyzing SDV sessions from both the senior and the junior developers, the researcher realized that the junior developer explored the source code in a fuzzy debugging pattern [2] throwing more breakpoints on disperse points than the senior developer. Furthermore, the junior developer executed the same session stack trace repeatedly without changes on the position of the breakpoints, as shown in Fig. 4.

Furthermore, the researcher observed that the developers forgot to remove the breakpoints when they finished the task. Consequently, these forgotten breakpoints remained in future tasks. Clearly, SDV helped highlight that behavior.

*c) **Second Insight**:* Other situation that could be observed was in the initial process of searching the issue, i.e., in the first debugging session, when the senior developer added breakpoints to random places to get situated in the source code, but he did not navigate by *StepInto* or *StepOver*, going only through *Continue*, hopping between breakpoints. After replaying the bug and locating the region in the code, the senior developer executed debugging sessions by doing the *StepInto* and *StepOver* steps, as shown on Fig. 5.

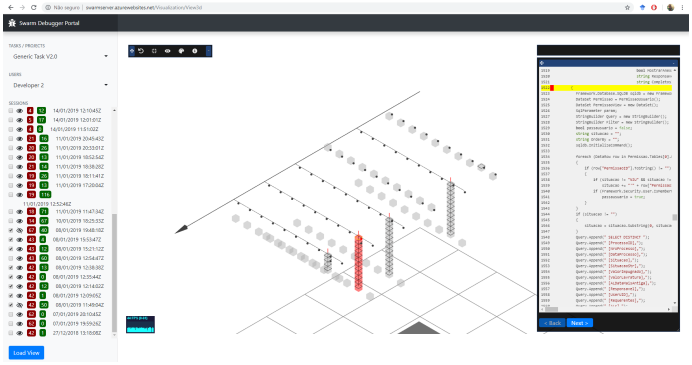


Fig. 4. Junior developer adding random breakpoints

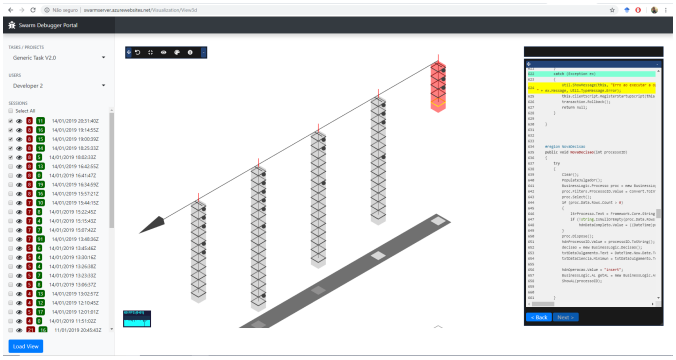


Fig. 5. Senior developer adding more breakpoints and performing the steps

*d) **Third Insight**:* The other point observed by the research team is that developers usually set a breakpoint after loops and press *Continue* to escape from long loops during the debugging sessions. Besides that, a similar behavior observed is that developers set breakpoints inside the method and press *Continue* or *StepOver* rather than pressing *StepInto*.

V. Discussion

In this section we discuss some initial aspects observed in case studies of SDV usage. SDV revealed some debugging behaviors, limitations, assistance and impacts. These aspects have implications on the work of developers, as shown.

We observe that replaying debugging sessions from SDV could help the team identify software arrangement involved in recurrences of bugs without the need for real debugging.

As a second point in the initial observation, we perceived that the reuse of debugging data by developers in similar problems could decrease development time and rework, as observed in our previous work [2]. Besides, we perceived that the debugging data collected from legacy systems by the senior developer ensured a knowledge base of the system over time. This knowledge could be reused by other developers, even as a resource to introduce new developers to the team and to understand the legacy system and software architecture.

As a limitation of the SDV, we observed that the use of SDV could be spread to the team only if the SDV IDE extension were compatible with *Visual Studio Code*. Similarly, we perceived the limitation of the SDV IDE extension when the developer was using more than one instance of *Visual Studio* at the same time.

As impacts on the team we highlight two relevant issues that were observed. The first one is that the filters and the visualization are not intuitive, therefore a previous demonstration was necessary so that the developers could understand the use of the tool. The second issue is that the visualization and source code working simultaneously, with the synchronization of events and the replay of debugging, helped the developer get situated in the original source code. However, we observed that the window where the source code was shown on SDV could be better visualized if it were a new tab of the browser, allowing it to be split onto two monitors.

VI. Conclusion

In this paper, we presented Sequence Debugging Session View (SDV), a 3D visualization tool based on *Swarm Debugging* and interactive diagram inspired on the UML sequence diagram that shows a temporal sequence of events during multiple debugging sessions, allowing replay of the debugging session.

The main result we obtained is that the ability to recover and share debugging data over time provided developers with support to understand the system and find the bugs. Furthermore, by analyzing the data collected from the case study (B), we gained some insights into developer behaviors. Moreover, we observed that replaying debugging sessions led the junior developer to locate the bug. Also, we perceived that SDV helped, over time, in how the legacy systems architecture evolves, assisting on software comprehension. However, we observed that our visualization demands a training session prior to its usage, because the *Swarm Debugging* concept and the metaphors are not intuitive, requiring a previous semiotic explanation.

As future work, we plan to implement the debugging extension for *Visual Studio Code* and other IDEs. On the 3D visualization, we plan on developing improvements and new features to make it more interactive and accurate, as well as improving the precision of the collected data. Furthermore, we plan to evaluate SDV by performing controlled experiments.

References

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

[2] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y. Guhneuc, “Towards understanding interactive debugging,” in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Aug 2016, pp. 152–163.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[4] I. Zayour and A. Hamdar, “A qualitative study on debugging under an enterprise IDE,” *Information and Software Technology*, vol. 70, pp. 130–139, feb 2016.

[5] F. Petrillo, Y.-G. Guéhéneuc, M. Pimenta, C. Dal Sasso Freitas, and F. Khomh, “Swarm Debugging: the Collective Intelligence on Interactive Debugging,” *arXiv e-prints*, p. arXiv:1902.03520, Feb. 2019. [Online]. Available: https://arxiv.org/abs/1902.03520

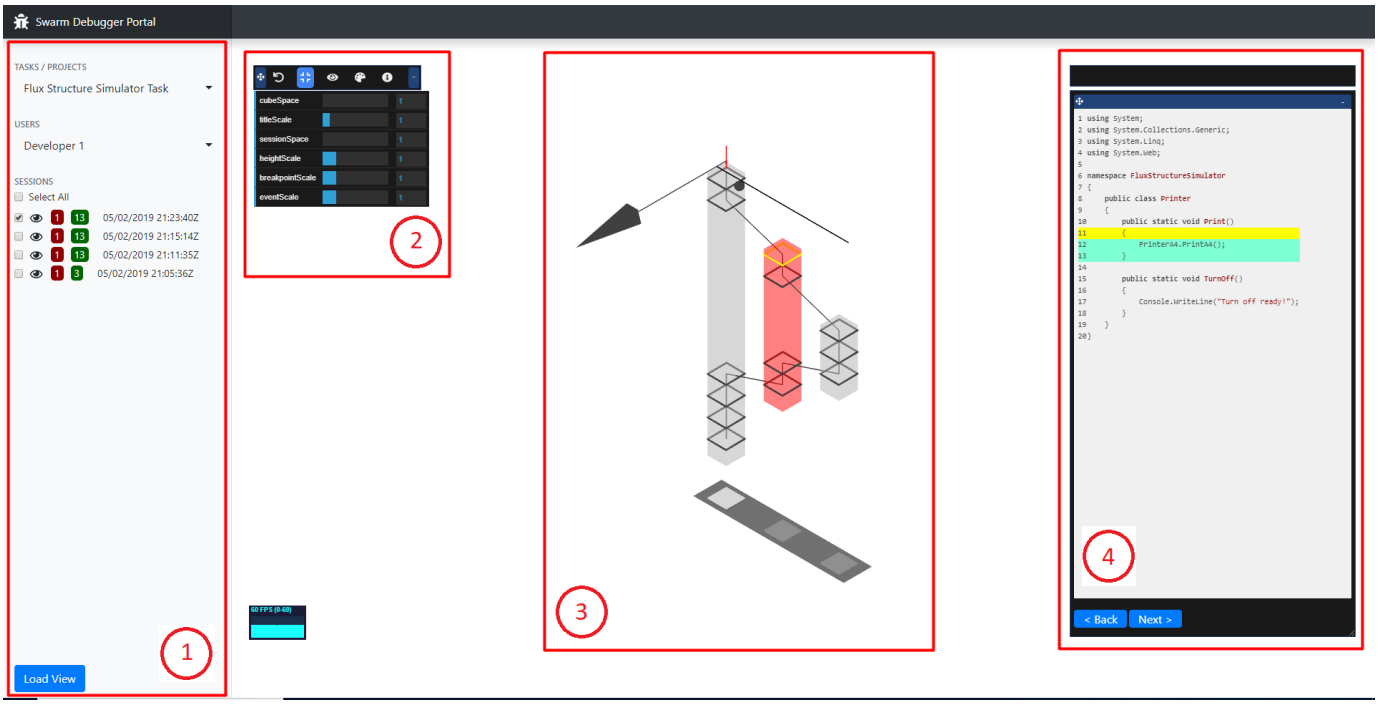
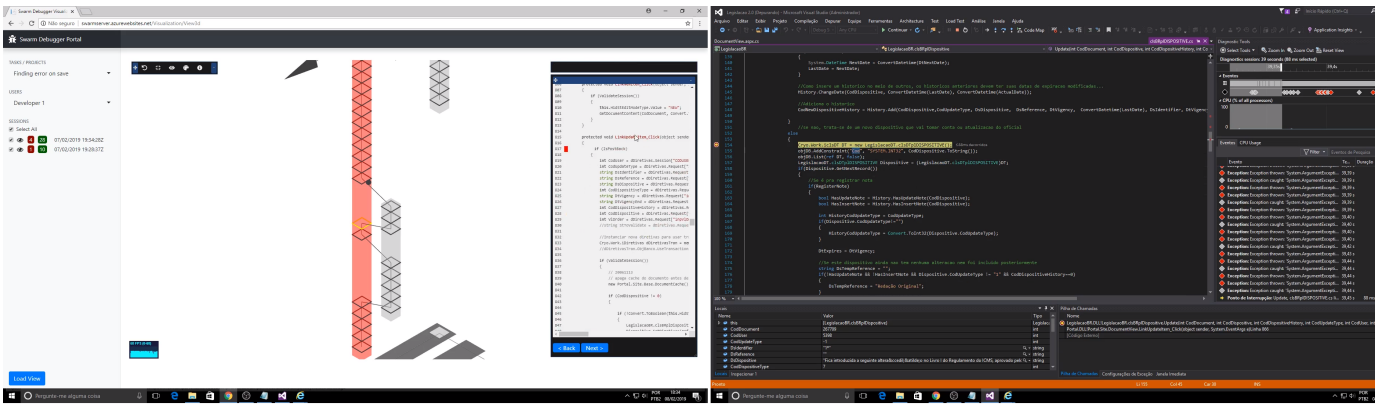


Fig. 6. 3D visualization front-end



Captured source code indicating where the bug is.

Bug location in system source code.

Fig. 7. The bug in the source code