# Requirements

*Debugger++: Trace-based Debugging for Java Development Environments*

Team 57: Robin Lai, Juntong Luo, Jane Shi, Arya Subramanyam

Client: The Reliable, Secure, and Sustainable Software Lab (ReSeSS)

CPEN 491: Capstone

Dr. Pieter Botman

April 11th, 2023

# Table of Contents

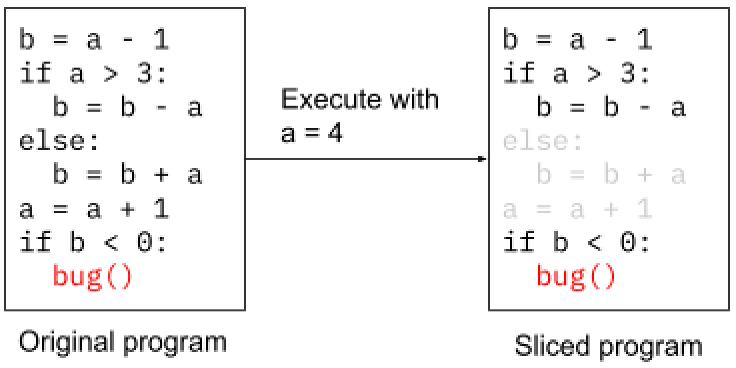
# List of Figures

# List of Tables

# 1. Background

All programmers must eventually contend with the frustrating process of debugging, the process of diagnosing and fixing software defects. It is hard to overstate the cost of debugging: this phase of software development “can easily range from 50 to 75 percent of the total development cost” (Hailpern and Santhanam 4). An even slightly improved debugging experience could have a far-reaching impact.

Von Mayrhauser and Vans estimate that software engineers debugging large programs spend up to half their time developing what they term a “situation model”, a bottom-up picture of the data flow and control flow that influenced the bug at hand (von Mayrhause and Vans). This suggests that a tool to present programmers with only the relevant part of the program they are debugging, as well as information gathered about the flow of data through the program, might help.



```
b = a - 1
if a > 3:
  b = b - a
else:
  b = b + a
a = a + 1
if b < 0:
  bug()
```

```
b = a - 1
if a > 3:
  b = b - a
else:
  b = b + a
a = a + 1
if b < 0:
  bug()
```

*Figure 1: Dynamic slicing on an example program.*
*The line “bug()” is selected as the slicing criteria.*

Fortunately, such tools exist: Program slicing is a technique to identify the subset of the statements of a program that affect some value of interest, known as the slicing criterion. This technique enables developers to fast-track their debugging process.

The Reliable, Secure, and Sustainable Software Lab (UBC ReSeSS Lab), our client, has created a tool, Slicer4J (Ahmed et al. 1570), for the Java programming language. Slicer4J performs dynamic slicing, i.e., executing the target program to collect a program trace (list of executed statements) and performing dependency analysis on the trace to produce a subset of the trace which affects the slicing criterion (as shown in Figure 1). Note that the slicing results for the same program can change if the program is nondeterministic (has possible changing execution paths), because Slice4J executes the target program when slicing.

The output slice of Slicer4J consists of the line numbers of the statements that affect the slicing criterion, as well as the control and data dependencies between the statements. For any two statements S1 and S2, S1 is control dependent on S2 if and only if S2 determines whether S1 executes, and S1 is data dependent on S2 if a variable used in S1 is defined in S2.

Currently, Slicer4J does not have a user-friendly interface. It only has a command-line interface and can only output the slice and dependency information as a set of log files. Last year, a capstone team worked on creating a standalone IntelliJ plugin that let users view the slicer's output in the IDE. The team conducted a user study at the end of the project, comparing users using IntelliJ's traditional debugger to ones using the slice viewer exclusively. Unfortunately, the results of the study indicated that participants were more effective in debugging programs using the traditional IntelliJ debugger.

The UBC ReSeSS Lab noticed that it may be more beneficial for users to access the Slicer4J functionality through the traditional IntelliJ debugger itself, instead of a Standalone Tool. Based on this, the goal of this project is to research how best to integrate the information generated by Slicer4J into the traditional IntelliJ line-by-line Debugger, build a working prototype (integrated into the IntelliJ Debugger), and evaluate the effectiveness of the system. Success for the project will be measured based on the delivery of the above-mentioned goals.

The significance of our project to our client is to make Slicer4J's dynamic-slicing functionality available to the average developer. Currently, Slicer4J requires extensive setup which a beginner-level programmer may have difficulty with. By providing an extension to the IntelliJ debugger, called Debugger++, all users of IntelliJ can conveniently take advantage of Slicer4J's functionality. Our client will benefit from the successful project by making Debugger++ available as an easily accessible, superior debugging solution to enable faster bug resolution for Java programs.

# 2. Project Outcome

## 2.1. Impacts

A successful project will:

1. Explore ideas for debugger-Slicer4J integration.
2. Develop the prototype Debugger++, an implementation of those ideas for IntelliJ which will allow Java programmers to debug problematic programs more efficiently by reducing the size of the program they debug (greying-out non-slice lines) and presenting the dependencies between lines.
3. Test Debugger++'s effectiveness as a debugging aid and present the findings to UBC ReSeSS Lab.
4. These findings will aid future research by the researchers in the ReSeSS Lab.

## 2.2. Stakeholders

### *2.2.1. UBC ReSeSS Lab*

The audience of this project includes the client, Professor Julia Rubin, who is the principal investigator of this project, and Sahar Badihi, who is a Ph.D. candidate supervised by Professor Rubin and is the main technical contact of this project.

### *2.2.2. Users of Debugger++*

The audience also includes any future users of Debugger++, which consists of Java developers with widely varying degrees of experience. These developers benefit by more quickly identifying bugs in their programs, or by using Debugger++ for program comprehension. They require installation instructions and a user manual.

### *2.2.3. Future Developers of Debugger++*

Additionally, the audience includes future engineers maintaining the system, depending on whether the client wishes to develop the project further or to maintain it. They require the schematics of our design, our project's source code, and our project's key documents in order to get a comprehensive understanding of the system design and implementation details of our project.

#### *2.2.4. Participants of Usability Study*

Finally, the audience includes participants of our usability study, which likely consists of other computer engineering students. They require information on how touse Debugger++ to test our implementation of the project.

# 3. Functional Requirements

The product should be a tool, integrated into IntelliJ IDEA, that enhances Java debugging workflows with the program slice produced by Slicer4J. Specifically, the product has the following functional requirements.

1. Allows the user to (1) select a slicing criterion, (2) generate a slice based on the criterion, and (3) start a debugging session with the slice.
2. In the source code viewer, (1) grays out program statements that are not in the slice, and (2) adds a breakpoint to the first line in the slice.
3. Allows the user to (1) use functionalities of the original debugger, with the difference that (2) lines not in the slice are automatically skipped when stepping through the program.
4. For the current line, presents the (1) control dependencies, and (2) data dependencies. The user can also (3) scroll to the line with the selected control dependencies and data dependencies by clicking on it in the slice info window.
5. Presents (1) graph visualizations of the dependencies between statements in the slice. The graph will (2) show only the subgraph for the dependencies of the current line.

# 4. Non-Functional Requirements

1. Performance
   The client has expressed that there are no critical performance NFRs (e.g. time for starting the debugging session). However, we have accounted for non-functional requirements required for good usability.
   1.1. Control dependencies and data dependencies of the current statement should be presented within 5 seconds.
   1.2. Dependencies graph should be presented within 5 seconds of a click in the GUI.
2. Usability

- 2.1. The interface design such as fonts and icons is consistent with the IntelliJ Platform design.
- 2.2. The interface is clear and simple, following the principles of subtractive design and visual hierarchy.
- 2.3. Each function should be accessible within 5 clicks from the main debugging view.
- 2.4. 70% or more users are satisfied with the usability of Debugger++ features including:
  - 2.4.1. Data Dependencies
  - 2.4.2. Control Dependencies
  - 2.4.3. Dependencies Graph
  - 2.4.4. Line Graying
  - 2.4.5. Modified Debug Actions
  - 2.4.6. Modified Breakpoints
- 2.5. 70% or more users are satisfied with the usability of Debugger++, overall.
- 2.6. 70% or more users find Debugger++ is easy to learn.

# 5. Constraints

The project must

1. Make minimal, if any, changes to Slicer4J for our solution.
2. Utilize the Slicer4J Tool to obtain the Slice-Based Debugging Information.
3. Be integrated into the current traditional IntelliJ debugger.
4. Have consistent GUI elements with the rest of the IntelliJ IDE's Interface.

Note that because dynamic slicing requires a concrete execution path that includes the slicing criterion, Debugger++ has to, before starting the debugging session, run the debuggee program (i.e. the program that is being debugged by the user) once and rely on this execution trace for dynamic slicing and slice-related functionalities. As a result, if the program execution path deviates from this trace when being debugged (the program is nondeterministic), the dynamic slice becomes invalid and Debugger++ will not be able to handle it.

# 6. Use Cases

For all use cases, we assume the IntelliJ IDEA IDE is running with the Debugger++ plugin loaded.

## 6.1 Use Case 1: Starting a debugging session with dynamic slicing

Preconditions**:** A Java project and a source file have been opened.
Main Scenario**:**

1. The user right-clicks a line in the source code editor.
2. A context menu pops up, with “Start Slicing from Line” being one of the options.
3. The user clicks the “Start Slicing from Line” option.
4. If other debugging sessions are running, they are terminated.
5. The IDE compiles source code, instruments compiled class files, runs the instrumented class files to get a trace, and calls the slicer to generate a program slice.
6. The IDE enters the debugging mode, grays out program lines in the slice in the source code viewer, and puts a breakpoint in the first slice line.

## 6.2 Use Case 2: Toggling breakpoints

Preconditions**:** A Java project has been opened and a Debugger++ session is running.
Main Scenario:

1. The user clicks on the space to the left of a line in the source code viewer.
2. Depending on whether there is already a breakpoint on the selected line:
    a. If there is already a breakpoint, the breakpoint is removed.
    b. If there is not already a breakpoint, a breakpoint is set.
3. If the breakpoint is in the slice, the program should be paused when it hits a breakpoint set by the user. Otherwise, the breakpoint is ignored.

## 6.3 Use Case 3: Stepping through the program

Preconditions: A Java project has been opened, a Debugger++ session is running, and the program is paused.
Main Scenario:

1. The user clicks one of the following 6 buttons: Step Over, Step Into, Force Step Into, Step Out, Run To Cursor, and Force Run To Cursor.
2. For (Force) Run To Cursor, if the cursor lies on a line that is not in the slice, a toast message pops up informing the user that they cannot run to a line not that is in the slice in the debugging with dynamic slicing mode. Otherwise, the program continues running and pauses if it hits the line pointed by the cursor.
3. For stepping buttons, the behaviour is identical to the stepping debug actions of the original debugger, except with the difference that lines that are not in the slice are automatically skipped.
4. The data dependencies and control dependencies of the current statement are displayed in the info window.

## 6.4 Use Case 4: Using other functionalities in the original debugger

Preconditions: A Java project has been opened, a Debugger++ session is running, and the program is paused.
Main Scenario:

1. The user uses a function that is present in the original debugger.
2. The behaviour of Debugger++ should be the same as the original debugger.

## 6.5 Use Case 5: Scroll to the line on pressing control or data dependency

Preconditions: A Java project has been opened, a Debugger++ session is running, and the program is paused.
Main Scenario:

1. The user clicks on a data or control dependency from the Data or Control Dependencies sub-tab within the Slice Info Tab.
2. The source code viewer scrolls to the position where the clicked line is.

## 6.6 Use Case 6: View dependencies subgraph for a statement

Preconditions: A Java project has been opened, a Debugger++ session is running, and the program is paused.
Scenario:

1. The user clicks the Dependencies Graph sub-tab within the Slice Info Tab.
2. The user can see the dependencies subgraph for the current statement.

### 6.7 Use Case 7: Terminating the debugging session

Preconditions: A Java project has been opened, and a Debugger++ session is running.

Scenario:

1. The user clicks on the Stop button in the top-right section of the window.
2. The current debugging session is terminated and the debugger view dismisses.
3. Grayed-out lines in the source code viewer are restored.

# 7. Traceability Matrix

| Requirement | Design Element | Test Case | Use Case |
|---|---|---|---|
| FR 1.1 “select slicing criteria” | 6.2 User Interface | 2.3.1.4 | UC 1.1-1.3 |
| FR 1.2 “generate a slice” | 6.1 JavaSlicer | 2.2.1.1-2.2.1.6 | UC 1.5 |
| FR 1.3 “start a debugging session” | 6.4 Dynamic Slice Debugger Runner | 2.3.1.4, 2.3.2.1 | UC 1.6 |
| FR 2.1 “gray out  non-slice statements” | 6.2 User Interface | 2.3.1.5, 2.3.1.8 | UC 1.6 |
| FR 2.2 “adds a breakpoint to the first line in the slice” | 6.3 Debug Controller | 2.2.1.8 | UC 1.6 |
| FR 3.1 “use functionalities of the original debugger” | 6.3 Debug Controller | 2.3.2.1 | UC 2, 4, 7 |
| FR 3.2 “skip non-slice lines” | 6.3 Debug Controller | 2.3.1.8 | UC 3.2, 3.3 |
| FR 4.1 “show control dependencies” | 6.1 JavaSlicer, 6.2 User Interface | 2.2.1.7, 2.3.1.7 | UC 3.4 |
| FR 4.2 “show data dependencies” | 6.1 JavaSlicer, 6.2 User Interface | 2.2.1.7, 2.3.1.7 | UC 3.4 |
| FR 4.3 “scroll to line when clicked” | 6.2 User Interface | 2.3.1, 2.3.1.7 | UC 5.2 |

| | | | |
|---|---|---|---|
| FR 5.1 “graph visualizations” | 6.1 JavaSlicer, 6.2 User Interface, 6.5 Subgraph Builder | 2.3.1.7 | UC 6.1, 6.2 |
| FR 5.2 “show subgraph only” | 6.1 JavaSlicer, 6.2 User Interface, 6.5 Subgraph Builder | 2.3.1.7 | UC 6.2 |
| NFR 1.1 “info window performance” | 6.1 JavaSlicer, 6.2 User Interface | N/A | UC 5, 6 |
| NFR 1.2 “Graph visualizations performance” | 6.1 JavaSlicer, 6.2 User Interface | N/A | UC 6 |
| NFR 2.1 “consistent as fonts and icons” | 6.2 User Interface | N/A | N/A |
| NFR 2.2 “clear and simple interfaces” | 6.2 User Interface | 3. Validation (Usability Test) | All |
| NFR 2.3 “accessible within 5 clicks” | 6.2 User Interface | N/A | All |
| NFR 2.4 “users are satisfied with usability of Debugger++ features” | All | 3. Validation (Usability Test) | All |
| NFR 2.5 “users are satisfied with usability of Debugger++, overal” | All | 3. Validation (Usability Test) | All |
| NFR 2.6 “easy to learn” | All | 3. Validation (Usability Test) | All |

*Table 1: Traceability Matrix*

## Bibliography

Ahmed, Khaled, et al. “Slicer4J: a dynamic slicer for Java.” *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1570-1574.

Hailpern, Brent, and Padmanabhan Santhanam. “Software debugging, testing, and verification.” *IBM Systems Journal*, vol. 41, no. 1, 2002, pp. 4-12.

von Mayrhauser, A., and A. M. Vans. “From program comprehension to tool requirements for an industrial environment.” *IEEE Workshop on Program Comprehension*, 1993.