Verification and Validation

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. Overview

The following document outlines our team's verification and validation methods. The verification section includes multiple technical tests that will be implemented to ensure the technical robustness of our solution. The validation section outlines a Usability Study and its results to ensure that Debugger++ achieves user needs by enabling faster and more efficient debugging using dynamic slicing.

2. Verification

All tests can be found in the *src/test* folder of our repository.

2.1 Testing Instructions

For the unit tests, simply run "./gradlew test" in the project root directory. The unit tests for SubGraphBuilder cannot be run in parallel because they all need to read from the same dot file so we have to run them separately.

For the integration test, first, open the Debugger++ repository in IntelliJ IDEA, and make sure you build Gradle correctly. Secondly, open the test repository by running the runIdeForUiTest command. Finally, go to the UITest.java located in src/test/java/team57/debuggerpp/dbgcontroller/pages/UITest.java, and run the test.

2.2 Unit Test Descriptions

The *UI* component, the *DebugController* component, and the *DynamicSliceDebuggerRunner* component cannot be tested individually as they require an active instance of our project's Plugin to be running. They are highly dependent on the current project and other IDE properties and states which cannot be easily mocked. Due to this, we will test these components through Integration Tests. Unit Tests will include the *JavaSlicer* and *SubgraphBuilder* Components and some utility functions which are well-isolated and not dependent on the IDE properties, state, and current projects. All unit tests are automated.

2.2.1 JavaSlicer Component Tests

The unit tests for *JavaSlicer* includes tests for the *JavaSlicer* class (which invokes Slicer4J) and the *ProgramSlice* class (which parse the output of Slicer4J).

The unit tests of *JavaSlicer* can be found in *src/test/kotlin/team57/debuggerpp/slicer/JavaSlicerTest.kt*. Table 1 includes a summary of the programs on which JavsSlicer is tested. For each program, the test invokes all functions of the JavaSlicer (except the ones dependent on an active instance of IDE) to execute the complete process of slicing (i.e. from instrumenting the program to producing the dynamic slice).

No.	Program	Description
2.2.1.1	TestProjectBasic.jar	This program aims to test a basic case which consists of a single class with 3 functions.
2.2.1.2	TestProjectTiny.jar	This program aims to test a basic case which consists of a tiny program with 2 functions.
2.2.1.3	TestProjectException.jar	This program aims to test the case in which the value of the variable in the slicing criteria comes from an exception.
2.2.1.4	TestProjectStaticVariable.jar	This program aims to test the case in which the value of the variable in the slicing criteria comes from a static variable.
2.2.1.5	TestProjectMultithreading.jar	This program aims to test the case in which the value of the variable comes from another thread.
2.2.1.6	TestProjectMultipleClasses.jar	This program aims to test the case in which the program has multiple classes.

Table 1: JavaSlicer Component Tests

The unit tests for *ProgramSlice* can be found in *src/test/kotlin/team57/debuggerpp/slicer/ProgramSliceTest.kt*. Table 2 summarizes the primary functions that are tested with the test description.

No	Function	Description
2.2.1.7	ProgramSlice/dependencies	By inputting the DynamicSlice produced by Slicer4J, the expected output is a map containing the correct

		dependencies information between lines in the source file.
2.2.1.8	ProgramSlice/firstLine	By inputting the DynamicSlice produced by Slicer4J, the expected output is the source location of the first statement in the slice.

Table 2: ProgramSlice Component Tests

2.2.2 SubgraphBuilder Component Tests

These tests can be found in *src/test/java/team57/debuggerpp/trace/SubGraphBuilderTest.java*. This table includes a summary of the primary functions that are tested with the test description.

No.	Function	Description
2.2.2.1	SubGraphBuilder/generate SubGraph(currentLine)	By inputting the specific line number of the program, the expected output would be the subgraph containing all executed lines that are control or data dependencies of the input line
2.2.2.2	SubGraphBuilder/generate SubGraph(validLine)	Given a valid line number, the subgraph should be successfully generated.
2.2.2.3	SubGraphBuilder/generate SubGraph(invalidLine)	Given an invalid line number, the subgraph should not be generated.
2.2.2.4	SubGraphBuilder/generate SubGraph(lineNotInSlice)	Given a line number that is not in the slice, the subgraph should not be generated.
2.2.2.5	SubGraphBuilder/generate SubGraph(validLine)	Given a valid line number, check if the number of nodes in the subgraph produced is correct.

Table 3: SubgraphBuilder Component Tests

2.2.2 Utility Functions Tests

In addition to the unit tests for major components, the project includes the unit tests for utility functions. These tests can be found in *src/test/kotlin/team57/debuggerpp/util*.

2.3 Integration Test Descriptions

Since our components and their integrations are highly dependent on the IDE environment and properties since our project involves building an extension to the IntelliJ IDE Debugger tool, it is not possible for us to easily mock these for the purpose of Integration Testing. Therefore, we

have chosen to conduct a manual test by running our Project Plugin to provide the IDE environment and properties required.

This is the code that we will use to create our integration tests, this code also can be found in src/test/TestProject/src/Main.java.

```
public class Main {
    private static int a = 5;

    public static void main(String[] args) {
        int a = 3;
        int b = 5;
        if (a > b) {
            System.out.println("ddd");
        }
        int t = test(2, 15);
        System.out.println(t);
    }
    public static int test(int x, int y) {
        int z = y - 5;
        int r = z + 5;
        int k = Integer.parseInt("4");
        if (x > 0)
            z = x + y;
        else
            z = x - y;
        if (k > 3) {
            a = 5;
        }
        if (r != 15) {
            return 1;
        } else {
            return z;
        }
    }
}
```

Figure 1: Program Used for Integration Tests

2.3.1 Debugger++ Integration Tests

These tests can be found in src/test/java/team57/debuggerpp/dbgcontroller/pages/UITest.java. This table includes a summary of the primary functions that are tested with the test description.

No.	Function	Description	Input	Expected Output
2.3.1.1	openProject()	Because we are highly dependent on the live environment of running IntelliJ IDE, we first need to set up the environment before testing. In this test, we checked the users' target Java version and then set up the remote robot that can generate the User Interface structure of the running IntelliJ IDE. Then we open the test file by retrieving the directory of the target test project. We also check if the Debugger++ logo appears after IDE finishes indexing.	N/A	N/A
2.3.1.2	testDebuggerppBtnWithoutSlicingCriteria()	In this test, we check the edge scenario where the user starts using Debugger++ without setting the slicing criteria.	N/A	The expected output is that there is a warning window pops up, indicating that the user did not select slicing criteria.
2.3.1.3	testDebuggerppBtnNothingMsg()	In this test, we check the edge scenario where the user clicks on the Debugger++ button located in the bottom tool tab.	N/A	The expected output would be a pop-up window, displaying the text “nothing to show”.
2.3.1.4	testRightClick()	In this test, we check the user is able to select slicing criteria and set the breakpoint in the expected place when right-clicking on the line they are interested in.	The slicing criteria is set to line 27, “return z”;.	The expected output would be a breakpoint on line 10 “int t = test(2, 15);”, and start a Debugger++ session.
2.3.1.5	checkGreyedOutLines()	This test is a follow-up test for testRightClick(). The purpose of this test is to determine if the lines are greyed out based on the slicing criteria that the user selects in the testRightClick() test.	N/A	Since we set the slicing criteria at line 27, the expected output is a set of line numbers that are greyed out, which is {2,5,6,7,8,9,11,16,19,20,21,22,23,25}

2.3.1.6	testDebuggerppActions()	This test is a follow-up test for testRightClick(). The purpose of this test is to check if the debug action of Debugger++ is running as the user expected. Specifically, we test step into, step over, and run to cursor.	N/A	After clicking on the step into button, the current execution line goes from line 10 to 14. After clicking on the step over button, the current execution line goes from line 14 to 15. Clicking on line 27 first and then clicking on the run to cursor button, the current execution line jumps from line 15 to 27.
2.3.1.7	testDebuggerppElement()	This test is a follow-up test for testRightClick(). In this test, we check if the data dependencies, control dependencies, and graph tab are displayed as expected. And clicking on the text in the two dependency tabs (if present) will cause a scroll to the line specified in the dependencies tab.	N/A	The tab of data dependencies, control dependencies, and graph are displayed. Specifically, after clicking the step into button 2 times, we should be able to see “To Line 15 (Main.java): int r=z+5; ”in the data and control dependencies tab
2.3.1.8	testDebuggerSkipNonSliceLine()	This test is a follow-up test for testRightClick(). In this test, we check if the non-slice line can be skipped in the Debuger++ session.	N/A	After clicking the step over button once and step into button 2 times, we should expect the Debugger++ to skip line 16, and currently in line 17.

Table 4: Integration Tests for Debugger++

2.3.2 Existing IntelliJ Debugger Integration Tests

These tests can be found in src/test/java/team57/debuggerpp/dbgcontroller/pages/UITest.java.

This table includes a summary of the primary functions that are tested with the test description.

No.	Function	Description	Input	Expected Output
2.3.2.1	useOriginalIDebugger()	In this test we check if the normal debugger is working as we	Set the breakpoint in line 10	The current execution line will jump from 10 to 14 after clicking on the step into button, and move to line 16 after clicking on the step over button. Then we move our cursor to line 27, the current execution line will jump to line 27.

		expected.		

Table 5: Integration Tests for Existing IntelliJ Debugger

2.4 Regression Testing

The repository of our project has a GitHub action that automatically runs all unit tests and builds the plugin when a new commit is pushed. The author of the commit will receive an email notification if a unit test fails or if the plugin fails to build. This achieves Regression Testing by verifying all preexisting functionality with every new change.

3. Validation

3.1 Introduction

In order to validate our product against its Non-Functional Requirments, our team has conducted an extensive Usability Study. The study aims to determine whether our solution, Debugger++, provides superior usability for developers. Since the motive behind this project is to provide an improved debugging experience, we concentrate on a multitude of key usability metrics to confirm whether the project and its features meet the users' needs and requirements. Participants in the study are asked to debug a program, during which they are required to use Debugger++ as their debugger. After the exercise, each participant fills out a survey to determine their satisfaction with product features and user experience.

3.2 Participants

Our participants consisted of 7 students studying at the University of British Columbia. As a control, we only gathered users that are familiar with the IntelliJ Debugger in some capacity. The set of participants consisted of 3 beginners, 3 proficient and 1 expert users in level of experience with using the IntelliJ Debugger.

Furthermore, our participants were split equally between students in Computer Engineering in their final year of undergraduate studies and in their first year of their Masters degree, except for

1 Computer Science student. Their ages range from 20 to 26, with the majority of our participants being 23 years old.

3.3 Experimental Procedure

The entire experiment was moderated by a Capstone team-member with the participants, in person. The testing environment was in a private room and the test was conducted on the moderators' computer, where the participant accessed a running version of Debugger++. At first, we took user consent for participation in the study. They were informed of its purpose and were briefed on what what was required of them during the study, including debugging a program and completing a retrospective survey after.

Since it is probable that our demographic of users may be unfamiliar with the analysis technique behind our project, dynamic slicing, they were shown an introductory video to help understand the concept. Additionally, this video also walked through the basic functions and benefits of Debugger++. This video was formulated using snippets from our team's Capstone Video.

Considering that this is the very first time that they were using Debugger++, participants were allowed 2 minutes to explore the tool and ask their moderator any questions about product use or features. Following this, the debugging task was conducted. The participants were given a maximum of 30 minutes to attempt to successfully complete the debugging tasks, after which they were made to fill out a usability survey.

3.4 Experimental Task

All participants were required to debug a deterministic Java Program with some bug in it. This debugging task was designed to ensure the requirement for debugger usage, considering that extremely simple bugs can often be debugged by eye. The program includes multiple lines which do not contribute to the main purpose of the function, present to confuse and distract the participant. Simple variable names were used, often based on the alphabet, to further confuse users and encourage the use of a debugger.

```
public class Main {
    public static void test(int a) {
        int n = 10;
```

```
        int s = a + n;
        int v = n - s;

        double j = v;
        int f = 0;
        int h = 1;
        for (int i = 0; i < a; i++) {
            f = (int)j - 1;
            h = f * 3;
        }

        int[] fib = new int[s - n];
        fib[0] = 0;
        v++;

        if (v < a) {
            j++;
            h = h + (int)j;
        }

        fib[1] = 1;
        for (int i = 2; i < a; i++) {
            int f1 = fib[1] + f;
            int f2 = fib[i - 2] + f;
            int f3 = fib[0] + v;
            fib[i] = fib[i - 1] + f2 - f;
            v = a - f1 - f3;
        }

        for (int i = 0; i < n; i++) {
            j = Math.random() * 10;
            double m = Math.sqrt(j);
        }

        System.out.println("The Fibonacci sequence up to " + a + " is: " +
fib[s - n - 2]);
    };

    public static void main(String[] args) {
        test(5);
    }
}
```

Figure 2: Buggy Fibonacci Sequence Class

The expected behaviour of the program is to print out the last number in the fibonacci sequence based on the argument provided to the `test` function. The bug is in the print statement, which incorrectly prints out the second last number in the sequence instead of the first. It can be fixed by changing the line `System.out.println("The Fibonacci sequence up to " + a + " is: " + fib[s - n - 2]);` to instead print out the value in `fib` of the correct index, which is `fib[s - n - 1]`.

3.5 Usability Survey

The usability survey was administered via Google Forms. Each participant was required to fill out the entire form after they had the opportunity to use Debugger++ during the debugging task. The usability metrics were formulated based on our Non-Functional Requirements, and are as mentioned in the table below, with the corresponding response options.

Usability Question	Response Options
How satisfied are you with the Data Dependencies?	Linear scale from 1-10 with 1 indicating Strongly Unsatisfied and 10 indicating Strong Satisfied
How satisfied are you with the Control Dependencies?	
How satisfied are you with the Dependencies Graph?	
How satisfied are you with the Line Graying?	
How satisfied are you with the Debug Actions (step into, step over etc.) skipping over non-slice lines?	
How satisfied are you with the breakpoints being ignored for non-slice lines?	
Overall, how satisfied are you with Debugger++?	
Debugger++ was easy to learn	Select one from Strongly Disagree, Disagree, Neutral, Agree and Strongly Agree
Debugger++ helped me debug faster	
Debugger++ helped me debug more efficiently	
What did you like about Debugger++?	Paragraph response
What did you not like about Debugger++?	

Table 6: Usability Survey Questions and Response Options

3.6 Results

Participants expressed a high level of satisfaction with Debugger++, overall, with an average satisfaction rating of 8.6/10. In particular, the survey indicated that users were most satisfied with the Line Graying functionality with 100% of participants indicating that they were strongly satisfied with the feature. Some participants expressed that it helped them “decrease debugger complexity” as it “shows a smaller program to debug”. A close second is the Debug Actions, with 6 of the users rating it 10/10, and the remaining 1 rating it a 9/10. The Data Dependencies and Breakpoint functionalities were well received with a majority of users choosing 8/10 for both, with average scores of 7.3 and 7.7 respectively. However, the Control Dependencies and Dependencies Graph were the least preferred, with majority of users expressing dissatisfaction with a score of 5 or less, for both. When asked what they did not like about Debugger++, most participants indicated that the Dependencies Graph is “too complex”.

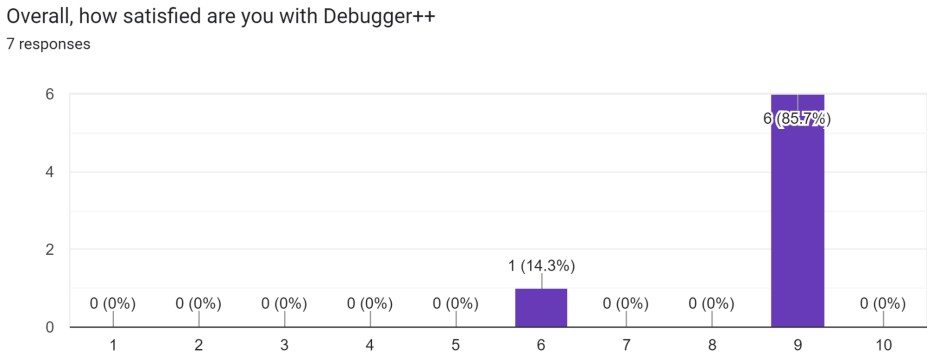

Figure 3: Usability Survey Responses - Overall Satisfaction

There was generally strong agreement that Debugger++ was easy to learn with 85.7% of participants strongly agreeing and the remaining 14.3% agreeing. Furthermore, 71.4% of participants strongly agreed that Debugger++ helped them debug faster and more efficiently, which are the major benefits the tool and its features aim to offer. This indicates that our implementation achieves these objectives successfully, proving that we indeed have built a faster and more efficient debugging tool compared to the traditional IntelliJ Debugger.

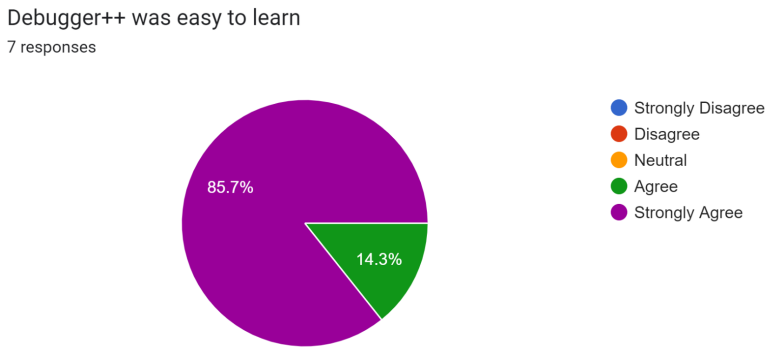

Figure 4: Usability Survey Responses - Ease of Learning

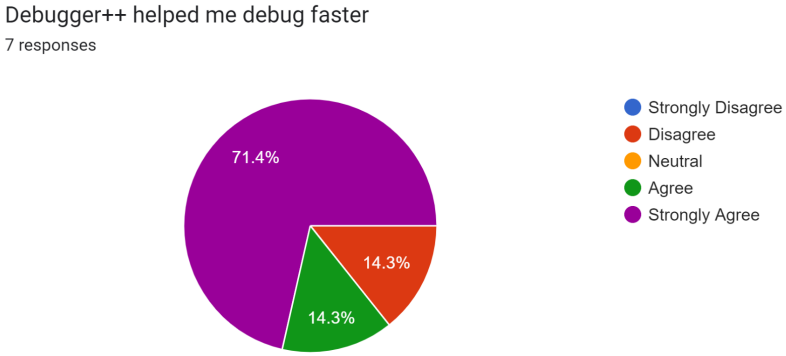

Figure 5: Usability Survey Responses - Debugging Faster

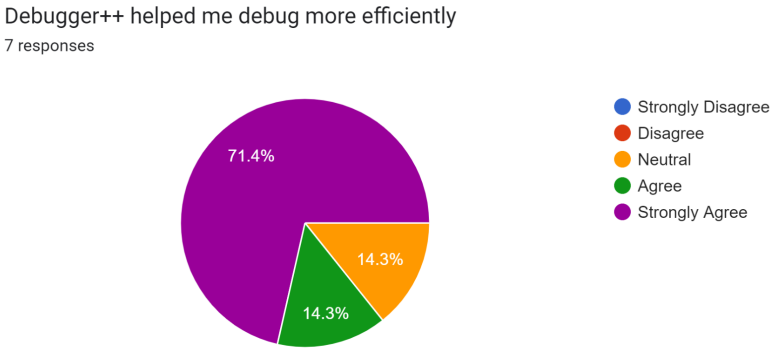

Figure 6: Usability Survey Responses - Debugging Efficiency

To view all raw data from the survey responses, please see *Appendix A: Survey Responses*.

3.7 Conclusion

Results of the usability survey indicate that the Debugger++ is a superior debugging tool which enables users to debug faster and more efficiently. Users find most features helpful, especially

the Line Graying and modification of Debug Actions which significantly helped reduce debugging complexity and increase efficiency. However, users would like to see improvements in the Dependencies Graph feature, including a reduction in its complexity. Generally, the tool is considered extremely easy to learn, although users may require some education regarding dynamic slicing prior to usage.

In conclusion, Debugger++ proves to be a helpful tool in debugging Java Programs, with a promising future. With additional refinement of features and improvements, users could be able to easily enjoy its benefits and reach new levels of debugging efficiency and speed. Possibly, more focus on isolating the relevant blocks of code like the Line Graying achieves could be greatly successful, along with a more accessible manner of displaying and visualizing dependencies.

Appendix A: Survey Responses

Please note that the results have been anonymized.

What is your name?							
Do you consent to participating in this study?	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Who was your moderator	Arya	Jane	Juntong	Juntong	Robin	Robin	Robin
What is your age	23	20	23	21	23	26	25
What year are you in	CPEN 4th / 5th Year	CPEN 4th / 5th Year	CPEN 4th / 5th Year	Computer Science 4th year	1st year in Meng	Master 1st	Master of Engineering in Computer Engineering
What is you experience with the IntelliJ Debugger?	Proficient	Proficient	Beginner	Proficient	Beginner	Beginner	Expert
How satisfied were you with the Data Dependencies?	7	10	10	8	2	5	9
How satisfied were you with the Control Dependencies?	4	8	10	8	2	1	5
How satisfied were you with the Dependencies Graph?	10	10	1	10	1	1	1
How satisfied were you with the line graying?	10	10	10	10	10	10	10
How satisfied were you with the debug actions (step into, step over etc.) skipping non-slice lines?	9	10	10	10	10	10	10
How satisfied were you with the breakpoints being ignored for non-slice lines?	4	8	9	10	5	8	10
Debugger++ was easy to learn	Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Debugger++ helped me debug faster	Disagree	Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Debugger++ helped me debug more efficiently	Neutral	Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree

Overall, how satisfied are you with Debugger++	6	9	9	9	9	9	9
What did you like about Debugger++	The greyed out function helps a lot of decrease the debuggin g complexit y.	It greys out unrelated lines	Good depende ncy design	It shows a smaller program to debug	line graying	Not bad, has good future.	the fact that irrelevant lines of code are grayed out
What did you not like about Debugger++	The program makes you feel a little too overdepe ndent on it in the sense that, normally when people debug, you would print out/check the target values section by section to isolate the buggy section of code quickly, then debug line by line from there. But	Graph becomes too complex	Can I zoom in the graph? ^^	The "TO Line xxx" label is confusing .	USELES S Depende ncies Graph	It's a good point to help debuggin g.	some irrelevent (have relationsh ips but do not change the value of target variables) lines are still highlighte d

	this tool makes you feel like you have to step through it line by line from beginning. That wastes a lot of time.						

Table 7: Survey Responses