Case Western Reserve University

Design Document

JEdu

October 15, 2019

Date	Version Number	Note
October 13	V 1.0	Initial Release Documentation
October 13	V 1.1	Team Reviewed and Signoff

1. Scope and Vision

1.1 Background

Major Java IDEs, like most other IDEs, are developed for professional programmers and have specific requirements for project structures. When using the IDEs, programmers need to meet these requirements or will have unsuccessful compilation and code execution. Although such project structure allows the programmers to conveniently manage their projects, it is not instinctive for most students taking EECS 132.

DrJava is a Java programming environment favored by one of the EECS132 professors, Prof. Harold Connamacher. Over many semesters, thousands of students picked up DrJava as their first Java programming environment. They benefited from its powerful interactive functionalities such as the interaction pane, the auto JUnit testing component, the integrated JDB debugger, as well as the JavaDoc generator. The need for this project arises from the fact that while JDK continues to release new updates, DrJava no longer supports the latest version of JDK and will soon become outdated due to the lack of maintenance.

As students who decided to major in computer science because of Prof. Connamcher's lectures, we are determined to help develop a substitute for DrJava, so that students in future semesters can benefit from the easy DrJava-like style of programming. After researching different Java text editors, we chose jEdit as the platform for our project. jEdit is an editor that allows convenient open source contributions through the integration of jEdit plugins, which we are utilizing to achieve some of the features that DrJava has.

1.2 Objectives, Success Criteria and Measures

No	Component	Success Criteria	Measure	Priority
1	JDB Plugin	The plugin is integrated with a seamless UI.	Implement a GUI for the JDB plugin that is consistent with the JEdit UI.	2
2	JDB Plugin	The plugin supports all of the necessary functionalities of a debugger.	The plugin features setting breakpoints, "step over", "step into", "step out", "resume", as well as querying variable contents.	1
3	JUnit Plugin	The plugin is developed as a functional testing plugin with a consistent GUI and does not require the presence of a project.	The plugin should allow the users to create test files and input their test cases formatted in the JUnit standard.	1
			The plugin features two displays. One will show each test with a fail/pass indicator, the other one will give the	1

			important error messages for the tests that fail.	
			The plugin should be able to test all of the other Java files that are saved under the same directory as the test file.	1
4	Interaction Pane	The interaction pane component should feature the same behavior as that of DrJava's interaction pane.	The interaction pane should take basic Java expressions and statements as inputs, then interpret and evaluate; the result should be displayed.	2

1.3 Vision Statement

We aim to improve the development experience of jEdit by implementing plugins that support functionalities similar to those of the JUnit, JDB, and interaction pane in DrJava. To be more specific, we are planning to:

1. Build a JUnit plugin that allows running customized JUnit tests and displaying test outputs readable by novice programmers without the need of a project structure.
2. Implement a JDB plugin with an intuitive UI and make it accessible to novice programmers.
3. Integrate an interaction pane plugin by providing a GUI for the JShell feature.

1.4 Features

1.4.1 Major Features

These are the features that we think are the most important and can likely be completed within a semester.

FE-1	Not requiring a project structure for a program to compile and run.
FE-2	Auto-detect class files under the same directory, therefore eliminating the need to open all files manually.
FE-3	JUnit plugin: allows running customized JUnit tests and displaying easy-to-understand test outputs (pass/fail + error reports) without the need of a project structure.
FE-4	JDB plugin: provide standard debugging operations: setting breakpoints, step into, step out, step over, resume, restart, etc. The plugin should also allow the users to investigate the values of parameters before and after the line is executed and trace the method calls.

1.4.2 Stretch Features

These are the less important features that we would like to complete if we have time left.

SE-1	Interaction pane plugin: takes basic Java expressions, such as (1+1), class instantiations, method calls, and package imports as inputs. The expressions will then be interpreted and evaluated, and the result will be displayed.

1.5 Scope and Limitations

Scope: This project will focus on making jEdit an easy-to-use tool for novice programmers by creating plugins. We will have a project that allows for easy and intuitive unit testing and debugging.

Limitation: With the limited time period, we will not be able to create all the features suitable for an IDE for the novice. Future projects could be done to build on what we will have.

2. Overall Description

2.1 Product Perspectives

This is a single-user project as it doesn't support any sharing. We only need to design for the usage of a single user at a time.

2.2 User Classes and Characteristics

The immediate target users for the project is Prof. Connamacher and future students taking his EECS 132 classes. The students are not familiar with Java, its project structure, unit testing, or debugging. The project needs to make it easy for them to get started.

Potential users include anyone using the jEdit tool. The general users could range from novice programmers to advanced programmers (who are probably writing other plugins). The project needs to be self-explanatory to use.

2.3 Operating Environments

OE-1: The plugins that we implement need to be run-able in all the operating systems that JEdit supports, namely Windows, OS X, and Linux. Because the plugins will be written in Java, we do not foresee any obstacles to achieving this goal.

2.4 Design and Implementation Constraints

CO-1: All code should be written in Java and compilable by JSE11 or later.

CO-2: All source code should be compatible with the JEdit environment.

2.5 Assumption and Dependencies

AS-1: Users will install their own JDK.

AS-2: User will be able to use JEdit (we may create a guide for that).

DE-1: JEdit plugin dev toolkit.

DE-2: JUnit library.

DE-3: Reference to JEdit's existing plugin source codes for UI designs.

3. User Case View

3.1 System Inputs and Outputs

3.2 JUnit Use-Case Detailed Descriptions

With some Java files and Java unit test files under one folder, instead of building a project, the user should be able to use JUnit plug-in to run tests and see test results for each test case in the JUnit UI. JUnit plugin can be accessed by Plugins > JUnit > JUnit.

3.2.1 Setting up test files

The test files should follow the general requirement of a JUnit class. Currently, JUnit Plugin can support JUnit API provided by package org.junit, which includes JUnit core classes like Runners class and Assert class, as well as JUnit annotations like ignore and test. In JUnit UI windows, the user can select test class by typing a class name or select by the drop-down menu.

3.2.2 Opening the JUnit window

In the dropdown menu on the top of JEdit window, select Plugins>Junit>Junit to display the JUnit window. The window is floating by default but can be made to dock into the view in the Docking pane of the Global Options dialog box.

3.2.3 Running in non-project setting

Currently, JUnit Plugin can only run tests in a project setting. Users should set up classpath for a project: to do so, users should click “Set Junit Path” button on the right-top corner and explicitly select the classpath directory. In the future release, JUnit plugin should be able to run JUnit tests for files that are not in a project setting. There should be a UI for users to select which test file(s) to run, and a script to look for the file and migrate it with current test running source code.

3.2.4 Running the Junit tests

After setting the classpath, JUnit tests will automatically find all the test classes by searching the classpath. By clicking “Select a Test Class”, JUnit plugin is able to generate a list of class names for selection. Also, a class name could be manually entered in the text field. Currently, JUnit Plugin does not support the selection of multiple tests at once, therefore to run multiple tests, the user needs to create test suites and include all unit tests in the suite. After that, the JUnit Plugin will run the unit test or test suite and display the results, including success, failure, ignore, and error/exception status, after clicking “Run Tests” button.

3.3 JDebugger Use-Case Detailed Descriptions

3.3.1 Debug Mode

In the menu bar for JEdit, there is a plugin option, which shows a drop-down menu of all the available plugins, the user should navigate to the Java Debugger plugin. Upon selection, the user will be presented with the debugging mode. In this debugging mode, the user will be able to perform standard debugging operations. There are several ways to exit the debugging mode. If an interruption occurred such as the file was compiled or there is an error in the code, the debugging mode will automatically be terminated. Another way to exit the debugging mode is when we have gotten to the end of the file.

3.3.2 Basic Operations

There are several basic operations that can be done in the debugging mode. The most important operation is setting breakpoints. The breakpoint is used in the debugging mode such that upon encountering the breakpoint, the program will stop running. However, the debugging mode will not exit at the breakpoint. This gives rise to the second important feature: resume running in debugging mode. The resume option will not be allowed in the beginning since that would not make sense. Only when the running program has come to a stop at the breakpoint would the resume button be enabled to restart running the rest of the program.

3.3.3 Advanced Operation

When hitting a stop at some breakpoints, three other operations will be available to the user besides the resume option. The user can further inspect their code by using the step in, step out, and step over operations. Step in operation allows the user to go to the definition of a function call, essentially showing the user what would be the next step in the java call stack. If the user is already inspecting the function execution and desires to stop doing so, using the step out operation will take the user back to where they started. The stepover operation carries the user to

the next line of code in the same file without showing them the details of how the current function call is executed.

3.3.4 Extended Functionalities

The operations described above lays the foundation for a debugger to operate. The most helpful functionality for users will be a display window where the value of the variables is displayed. The display window will only be activated in the debugging mode and is updated every time the program runs to a stop at a breakpoint. Only the variable within the current scope will be displayed in the window. Moreover, when the user uses any of the steps in, step over function, the variable value display will also be updated since these operations indicate that a line of code has been executed.

3.4 Interaction Pane Use-Case Detailed Descriptions

3.4.1 Compile

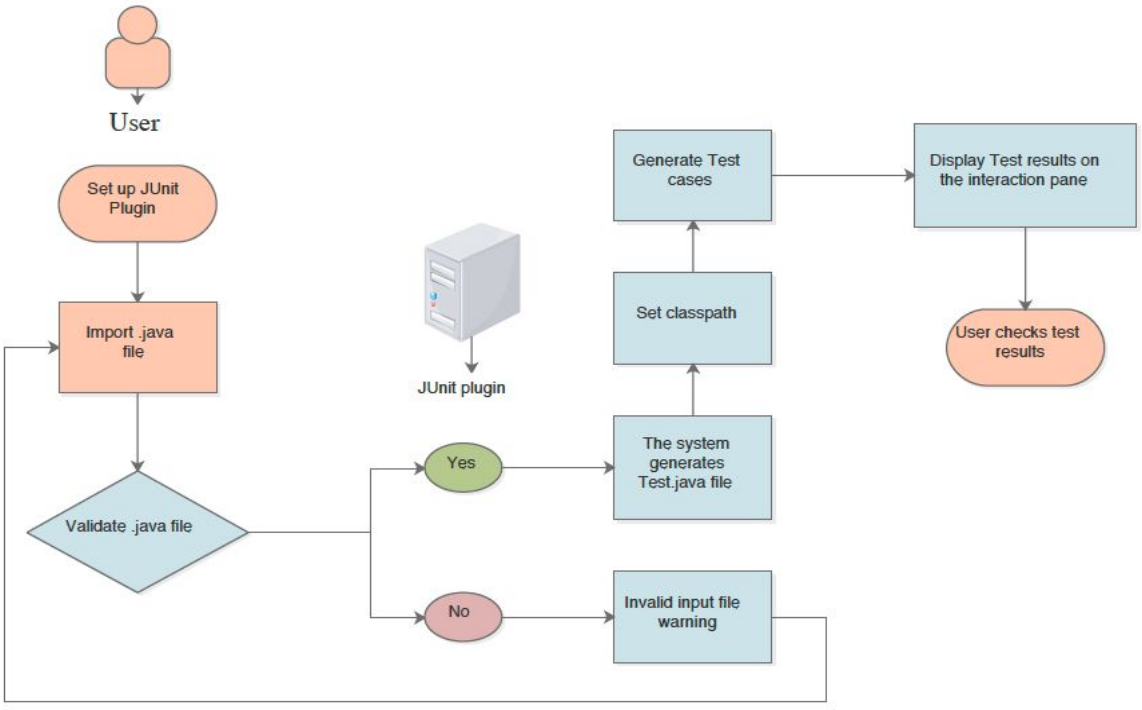
The user needs to compile the file to initialize the interaction pane for the corresponding file. After compile is hit, the interaction pane allows the user to achieve the result of having a main method in the java file. The interactions pane is a convenient way for new users to test their programs as they do not need a concrete class structure. A good interaction pane will have internal functions set up to enable better presentation, such as a List data structure will be presented in a row of numbers where the iteration was done internally in the background.

3.4.2 Initialize objects, calling functions, etc.

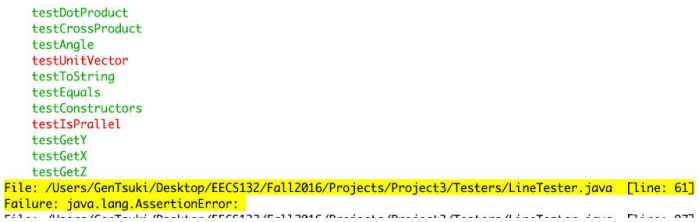
Once a Java file is compiled successfully, the interaction pane is ready for some testing. The user may write Java expressions to test the result. This includes but not limited to initialize objects, calling functions (static or non-static), running main methods, running tests, etc.

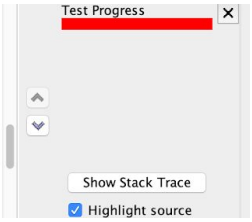
3.5 Use-Case Diagram and UI Example

3.5.1 JUnit

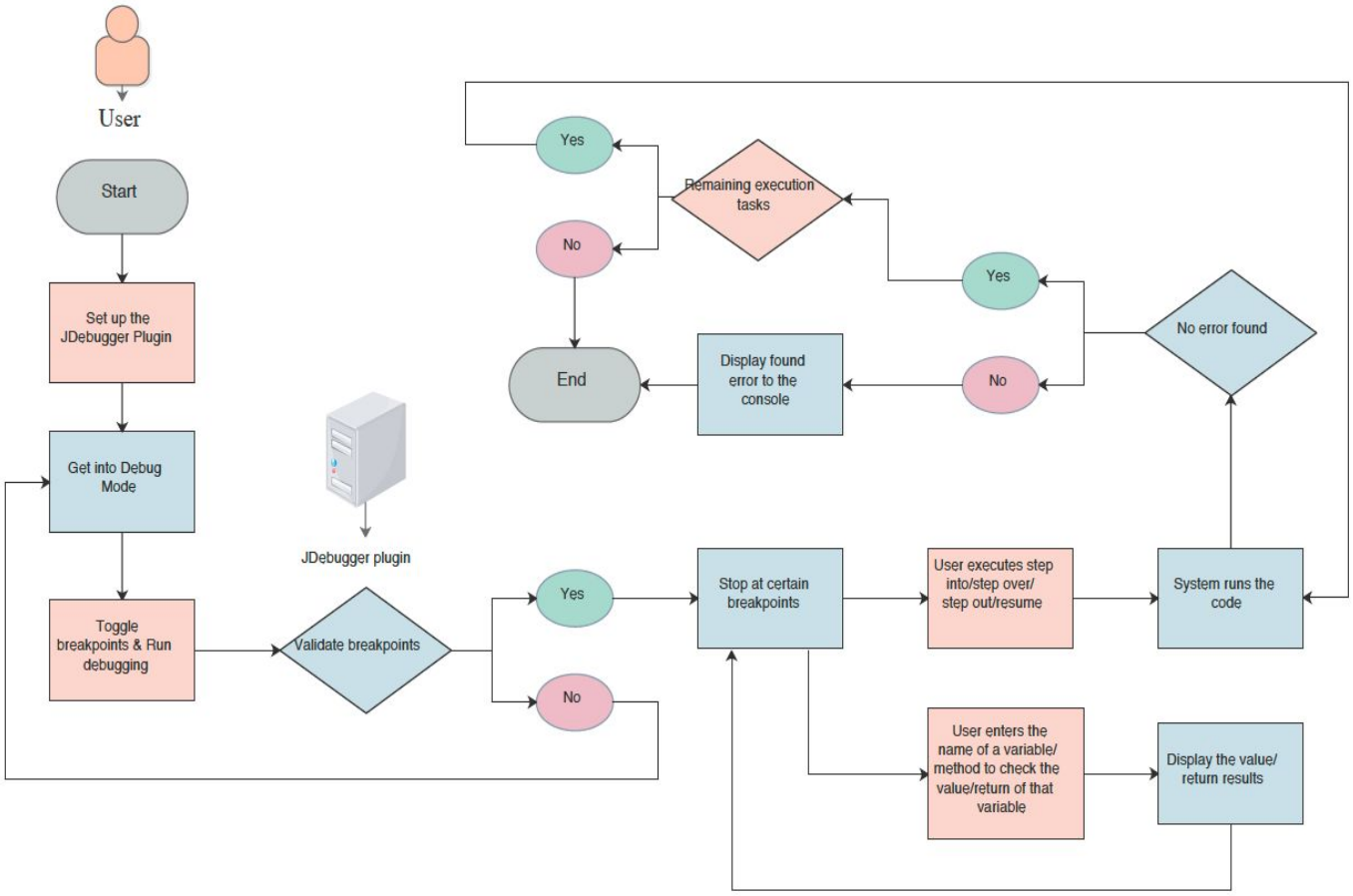


(Remarks: Pink squares represent the workflow of users; Blue squares represent the workflow of JDebugger system; Pink diamonds represent condition statements of users; Blue diamonds represent condition statements of JDebugger system.)

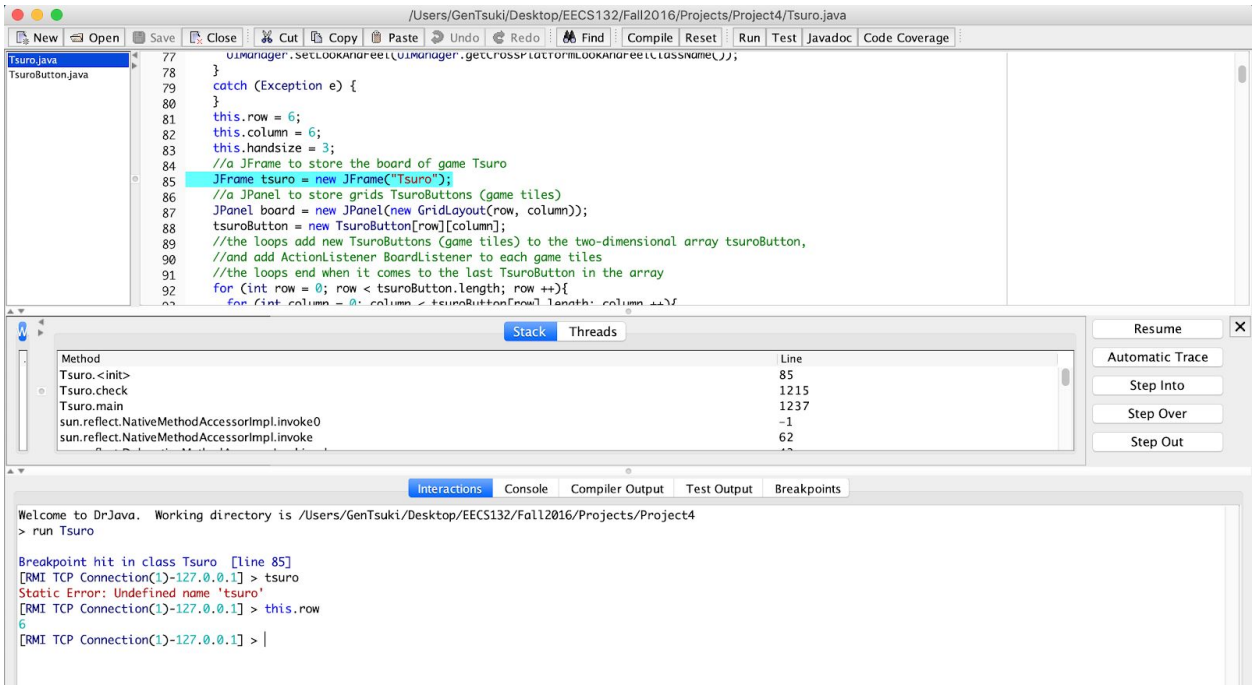




3.5.2 JDebugger



(Remarks: Pink squares represent the workflow of users; Blue squares represent the workflow of JDebugger system; Pink diamonds represent condition statements of users; Blue diamonds represent condition statements of JDebugger system.)



4. Non-Functional Requirements

4.1 Performance Requirements

PR-1: The average execution time of each unit testing iteration would be no longer than 2 seconds unless for bad user cases.

PR-2: The average responding time of each debug session should be no longer than 3 seconds.

4.2 Quality Attributes

Availability-1: The JEdu plugin pack should be available as a .jar file and compatible with the latest version of JEdit.

Integrity-1: The JEdu plugin pack should include the debugger plugin, the unit testing plugin, and the interaction pane plugin if given enough time.

5. Project Progress

5.1 System and Feature Design

By the submission of this report, the system and feature design of both the JDB plugin and JUnit plugin has been completed. We filled the project in with features according to the use-case flow listed in the Use Case View section above and found open-source projects that can support the integration of our development.

5.2 User Interface and User Experience Design

Although the JEdu plugin pack will be integrated to JEdit, as the ultimate goal of the project is to empower most of the convenient user experience of Drjava, which is soon becoming outdated, we designed the UI/UX to best accommodate the existing features of DrJava as listed in the above Use Case View and Use Case Diagram sections.

Notably, not all of the features presented in the workflow will be implemented due to the limitation of time. More details about our goals and non-goals before the upcoming second progress report will be addressed in the Updated Project Plan section.

5.3 Existing Code Review

As mentioned above, we found the existing source code for both the JUnit and the JDB components. Both plugins are open-source and available at the JEdit plugin home page. However, while the JUnit plugin still functions as a concrete program, due to the invalidity of certain dependencies, the JDB plugin is no longer working. To be able to leverage our project on the existing source code, we reviewed the coding structure and dependencies carefully and adjusted our system design to be integrated into the interfaces of these plugins.

5.4 Project Construction

Upon the completion of our project design and code review, we have initiated our project source folder as well as all the necessary binaries. We separated our JEdu project to two major components: JDB and JUnit, each as an individual project. We are planning to pack the two projects together as one plugin by the end of our integration, but for clarity, we will be implementing the two projects separately.

We structured the project classes according to our functional and system design. In the meantime, we created interfaces for the legacy code of the two plugins we found. Finally, we have initialized a private git repository for source control purposes of our project. We created

separate branches for the JDB and JUnit code commit and will regularize strict group code reviews for any pull request into the master branch.

6. Updated Project Plan

6.1 Next Steps

6.1.1 Java Debugger

After inspecting the source code we intended to integrate into the jEdit IDE, we discovered that the major dependent libraries no longer exist. As a result, we decided to build our own debugger based on the structure of a similar project - another java debugger plugin made for integration. To utilize the structure of the new source code, we need to clean it up by identifying workable functions. As this debugger plugin is an open-source contribution for jEdit plugin, we will not be completing all the functionality of the full java debugger. We will be resolving the legacy code and laying out the structure for our debugger plugin. We will be implementing the basic and advanced operations which consist of setting breakpoints, stepping in, stepping out, etc. If time allows, we will also make the display window showing variable values at breakpoints. We will be leaving out all the necessary interface of our existing functionalities for further improvement.

6.1.2 JUnit

The current JUnit plugin for jEdit is workable, but the plugin itself does not provide a good user experience. We intend to improve the UI for the JUnit plugin for easier access. Moreover, an important feature that we will be adding to the existing JUnit plugin is the ability to compile files without a project structure. Since the goal for this improvement is to allow new users to learn java, the compiling should be done in an intuitive way - the user will be able to use the compile button in the menu, and a pop-up window will appear, showing a basic file selecting window available for all operating systems. The user will be able to compile a file that might not necessarily be the current file by selecting it in the UI.

6.2 Prioritized Deliverable List and Estimates

Pri	Name	Owner	Estimate	Description	Test/Validation	Status
1	Wireframe Deliverable	SG	2 weeks	Wireframe deliverable of the UIUX design for all of the plugins. The wireframe deliverable should demonstrate the end-to-end workflow of the three plugins.	ALL	Delivered
2	Functional Design Deliverable	ALL	4 weeks - done by the first	Functional design deliverable of the plugins. The deliverable should contain all of the details necessary for the	ALL	Delivered

			report	implementation of the plugins.		
3	JDB Functional Demo	SG, YG, YS	9 weeks	Functional demo of JDB plugin. The plugin should demonstrate: initialize debug mode, toggle breakpoints, step in, out, over, and resume debugging, exit debug mode.	[redacted]	In Progress
3	JUnit Functional Demo	JY, ML, RY	8 weeks	Functional demo of JUnit plugin. Besides the existing functionalities of the legacy JUnit plugin, our improved version should demonstrate a better GUI: test all cases in one click, locate the line of errors, etc. The plugin should not require a project structure.	[redacted]	In Progress
5	Interaction Pane Functional Demo	ALL	Idle, moved into stretch goal	Functional demo of interaction pane plugin. The plugin should demonstrate the full functionality of auto-loading files under the same directory, prompt the users to compile the files, and execute Java expressions or statements.	ALL	Pending
4	Final Deliverable	ALL	12 weeks	Final deliverable of all of the plugins. The deliverable should support the functionalities we specced out in our design doc, and for the non-goals, we should provide documents for programmers who wish to pick up our project.	[redacted]	Pending

6.2 Team Member Contribution

6.2.1 JDB Component

[redacted] - Managed group meeting and project progress layout. Researched dependencies for the current Java Debugger source code. Decided on the scope for the project and use-case. Designed the structure for our java debugger. Contributed to the non-functional requirements, project progress, and prioritized deliverable list sections for this report.

[redacted] - Organized usable dependencies for the existing source code. Decided on reusable source code. Created the basic project functions and key interfaces. Implemented part of the class structures for the debugger foundation.

[redacted] - Visualized the mockup of end-to-end user experience and interface. Drew the flowchart for project designs and the documentation of the existing source code. Debugged and tested each part of the existing code to determine usability. Created all the diagrams and visualization of the report.

6.2.2 JUnit Component

[redacted] - Set up github workflow and branches. Investigated and experimented on JUnit plugin use case. Try to run JUnit as with project structure. Spot problems and possible feature needed for our target users. Contributed to the JUnit design part of document.

[redacted] - Resolved the compatibility issue of the JUnit plugin. Investigated and experimented on the dependencies of JUnit Plugin. Went through the documentation and source code. Contributed to the JUnit design part of document.

[redacted] - Tested the functionality of existing JUnit Plugin by run JUnit Plugin with testing classes under different operating systems, with different jdk versions, and with different unit test packages. Wrote and modified JUnit user case description. Contributed to the background part of the document.

Appendix A: References

1. JEdu - Project Proposal.
2. JUnit Plugin for JEdit user guide.
3. S. Pestov. jEdit user guide. http://www.jedit.org/.
4. E. Gamma and K. Beck. JUnit user guide. http://www.junit.org/
5. R. Cartwright. Most recent version of Java on Mac. https://sourceforge.net/p/drjava/bugs/974/
6. Stoler, Brian Richard. "A framework for building pedagogic Java programming environments." (2002) Master's Thesis, Rice University.
7. B. Burke and A. Brock. Aspect-oriented programming and JBoss. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html, 2003.