Final Report

Braden Meyerhoefer, David Alejos, Jiawei Zhang

Section 1:

Braden Meyerhoefer:

- Execution tracing functionality
- CLI
- Visualization
- Integration

David Alejos:

- File interaction functionality

Jiawei Zhang:

- Summarization functionality

Section 2:

Reading and understanding other people's code is really hard, but it is a necessary skill for developers to have as they move from project to project. Right now, a lot of time is wasted onboarding new developers as they read documentation and code to try and get to a point where they can start contributing. Codenser is a tool that aims to speed up that process by giving developers insight into how applications are working, and what pieces of code are the most important to understand.

Codenser does so using three core functionalities: execution tracing, summarization, and file interaction. Execution tracing shows the flow of how a program is running, summarization gives the location and quick summary of pieces of code, and file interaction shows how the various files of the application use each other. All of these modules are integrated into useful visualizations to assist new developers in understanding their new project.

Section 3:

https://meyerhoeferb.github.io/sd_final_submission/

Website, as requested, contains: images of teammates, brief biography of each teammate, video demo, links to writing, this report

Section 4:

Java libraries used: spoon, ASM, apache.commons.IO

Python libraries used: os, networkx, matplotlib, pandas, plotly

Section 5:

For the file interaction module, the first thing that happens is that the names and locations of all the java files in the codebase are extracted using the apache commons IO library. Then, the java files are scanned, and all method calls with relevant information are extracted using the Spoon java library. The module then outputs a csv file containing the adjacency matrix for all the files, where files are adjacent if they call a method from another file. For the execution tracing module, we designed a java agent that runs in the Java Virtual Machine and outputs a log of when methods are called and returned. This is then processed by a Python script to output a weighted adjacency matrix, where weight is how many times one method calls another. For pseudocode generation which gives users a brief summary about some specific parts of the code, it first takes in the entire source code and breaks it down into a parsing tree. Each command and keyword are stored as the node of the parsing trees. Based on the relationship of these keywords and commands, we can find its left or right children under the node. Then based on the node information, we generate a brief English sentence summary. For example, if the node is a “for” statement, it will generate: “There is a for loop”, and based on its child information, we find the condition of the for loop as well as the operation within it. All of this information will be stored for later use. In addition, information such as line number, method name, and file name will also be stored with the code information in a csv file.

All of this output data is then processed by Python scripts to visualize in graph form, with heatmapping for the nodes. Heatmapping is calculated by incoming weight of each node. On hovering over the node, the user gets the relevant information about the node (class,

method, file, etc.). All of this runs in a command line interface, which is simply a Python script using the terminal as a back end. Through the os library, the script calls the various functions as if from the command line, just automated so the user can do each part in a single command.

Section 6:

David: Working on this project has made me realize how important it is to pick the right tools for the job and the importance of being flexible when it comes to redoing a part of a project even if it is almost done. If I had to start again, I would pick more effective tools to create what I needed, and I would also not hesitate to start over some parts when a better tool or idea comes along. Something else I would do differently is that instead of trying to work starting from the user facing components I would try to focus more on components that the user cannot see in the beginning and then worry more about the user facing parts later on. That is what I would do differently if I started this project again.

Braden: Execution tracing was not a new concept to me, we had done it by hand so many times in various computer science classes and it's a pretty standard feature in many debugging tools. But to see how it works at a low level was so interesting, I had no idea a "premain" method was a thing, or even really that tools existed to modify code at the bytecode level. Working with such a complex library forced me to be creative in how I solved the problems I was faced with, figuring out how to do it in the simplest way I could (like having my part print to standard error and point standard error to a file instead of figuring out how to implement file IO at a bytecode level). It was also beneficial to build the CLI, as I hadn't really done one for a serious application before, and having to call various functionalities written in different languages and different files was an interesting problem to solve. Managing a complex file structure like that was also new for me as well, and honestly quite satisfying. I wouldn't really change anything if I went back, I'm proud of my work and the design/technical choices I made. I would make my part so much better though, knowing what I know now, because so much of the early weeks were spent finding ASM. If I had known about it from the start, I could have made my module much more robust.

Jiawei: If I can do the project again, I wish that I can figure out the direction of the project a little bit earlier. I have spent a lot of time trying to figure out the purpose and the approach for my part and I tried many different ways to accomplish the goal of the project. However, many of these attempts are not exactly the same as the original goal and my mentor has to correct me and provide me suggestions every single time. I personally think that I have wasted some time when trying to figure out the direction and during that period of time, I was really struggling and anxious even though I spent a lot of time on the project.

Section 7:

The video on our website is not a final presentation, since we didn't have to do one, but is rather me (Braden) doing a live demo and explanation of Codenser. Hopefully that is helpful for future groups. Everything should work out of the box for our project, make sure you run the setup script first to update the python packages and then the CLI should be good to go. Also read the README, I put some important developer notes and instructions on running it in there that I won't type out again here. In short, run codenser.py and it's kind of straight forward from there. The video explains how to analyze your own test files.

Summarization is one section that really needs some work, and then integration with the visualization. I was imagining each node of the execution trace having the line number and file it is located at, along with maybe some basic summarization if possible (which is really hard as we found out, but line number and file is definitely possible). If the test program writes to standard error itself that is potentially an issue for execution tracing. I know how to fix it but haven't gotten around to doing so, so I leave that to future teams (hint: it would be a change in make_graph.py). The ASM tool was pretty tricky to use, so I went for the simplest execution trace. With that as a starting point, future teams could make it more complicated like weighting by runtime or something.

A good suggestion a classmate made was that this could be a module integrated into an IDE like Eclipse. That would be a cool thing for a future group to do. Or make it a GUI, although I don't think it really adds anything to this project.