A1 Report

Step 1

Using external libraries make it easier and faster to implement certain features. If the project did not utilize Apache CLI to gather the command line options, it would have been much more time consuming to create a class manually that performs the same feature. This also holds true for the Log4J library, since it reduces the amount of work needed to implement that specific feature. Overall, it allows us to focus on the goal instead of focusing on smaller, non-important features.

A logging approach provides a way to track where certain lines of code are executed. For example, we can easily tell from which method of the class (in this case Main) the code was executed from, and it also indicates what type of logging it is, such as plain info, or an error that needs to be addressed. One disadvantage of logging is that it may not be useful when you need to print a line to the user for them to read, and you don't want all the extra information to go with it.

Step 2

The abstractions used in this project were Algorithm, Maze, Player and Vector. The abstracts can vary greatly, but as long as they serve the function correctly, they are viable. In this case, the Algorithm class was chosen, since the program will most likely utilize several different algorithms to solve the maze. This makes it easy to add different algorithms by creating an abstract class. The Maze class was added to store the maze and hide certain implementation details by the use of methods. The Player class was used as a "pin" to move across the maze. Lastly, the Vector class was used to abstract certain complex position and direction details. Overall, these classes have served well win one another, since they made development easier by simplifying certain process, but at the same time they don't abstract too much away to the point where the project is "over-engineered" or the program is too constricted.

Step 3

Many features were added to the Kanban board for this project. It was difficult to identify things that are only visible to the end-user due to the technical complexity of this project, however, the business logic specification of the project provided a general description of what is to be expected of this project, which outlines all the features that should be visible to the end-user. This made it easy to identify features needed in the command line interface, which are what the user use to navigate the program.

This ensured that the features only model visible value to the user, because the business specifications do describe the "how", or "why" each feature is implemented, but "what" features need to be implemented. The user is not concerned about the technical process involved in achieving a certain feature, so that was also taken into consideration when labeling the names for each new "issue" in the Kanban board. For example, the feature "Verify path provided by the user" does not describe the process in which it would

occur, it simply identifies that this is what to expect as one of the utility features in the script.

Step 4

The version of the program under the"mvp" tag on the GitHub repository represents the minimum viable product for this project. It is considered viable, because it achieves the main functionality of the program, which is to provide instructions to solve a maze. It is minimal because it does so in the most simplest form: Solving a straight maze where all you need to do is move forward. This resulted in the first working version of the program, which can be useful when you need to showcase functionality as soon as possible.

Step 5

Encapsulation was heavily used in the Maze class. The internal variables such as the maze's pattern, file, positions and start directions were all private variables that are inaccessible to external classes. They were able to be read through methods, however, the methods that set some of those variables were private. This ensured that they could not be modified from the outside, thus being encapsulated. Several "helper" methods such as "isPositionEmpty" were also used to define common use cases of those variables without modification, which reduced the complexity of those tasks. This ensured that the other classes, such as Algorithm classes could implement their path finding logic without causing issues.

The implemented code follows many of the SOLID principles criteria that ensure it is written well. Firstly, all of the classes have only one responsibility and do not extend further beyond that. The Open-Closed Principle was used extensively in the Algorithm class, since new algorithms can be added without modifying the main program and how it works. The code also adheres to the Liskov Substitution Principle using the abstract classes for the Algorithm class to implement new logic without causing errors. This can be seen through the use of polymorphism when deciding the type of algorithm used in the solving of a maze. Interface Segregation principle was not needed in this current implementation, since most of the components of the design were quite simple. Dependency Inversion Principle was implemented since the main program does not directly use the Maze class to determine the path. It is instead done through an abstract Algorithm class.

The code that was implemented supports the use of new algorithms through the creation of a new abstract class. This class has all the methods common for the class type, such as the ability to move, turn, have a position and direction, however, each new abstraction must define its own "solveMaze(Maze maze)" method, where it determines the path to complete the maze provided in the arguments. In the main file, the program selects the algorithm to use based on the "-method" option and selects the correct abstract algorithm. Then, polymorphism is used to call the method, regardless of what instance it is. This means that it is easy to add another algorithm just by creating a new abstract Algorithm class and adding it to the main file.

Iterative and Incremental principles were used throughout this project. Initially, a walking skeleton was set up within the code to detail the different classes and methods that were going to be used. Then, a minimal viable product was developed by solely achieving the functionality of solving the simplest possible maze: A straight maze. Then, the final product was developed in the next iteration of the design process. Evidently, there was much refactoring involved, but this is typically unavoidable due to the nature of this specific design process. Each iteration took approximately one week, but the amount of work needed to complete the iteration was minimal since it was spread out throughout several weeks.