Step #1: Introduce a New Dependency

1. **Why is it preferable to use libraries such as Apache CLI or Log4J to support things like parsing the command line or printing log statements instead of writing code from scratch?**
   Apache CLI is a preferrable library to process command line arguments because it provides more customizable and multi-functional abilities than the just using the regular command line. Apache CLI provides the Options class which enables the developer to set up checks for which flags can be included in the command line and if those flags can be accompanied by a body. Apache CLI then provides multiple functions from the Options class to check if the flag is present, if the flag comes with a body, and other functions to process the data. From scratch, the arguments array provided by Java only reads the command flags in order and does not provide for the customization that can be setup by Apache CLI ("Apache Commons…", 2023a: "Using Apache…", 2023b). Log4J provides more detailed logging statements and tracking capabilities for log statements at runtime. The Log4J dependency comes with multiple settings of information, such as Info, Error, Fatal, and more. When these type of log statements are printed, it helps to notify the user what type of log the data is, which can help to identify processes in large programs. Additionally, when errors and exceptions arise, error loggers print the exact exception and where the exception arose to help find the error and debug the problem. Log4J compared to system functions as a result helps to code and debug with more descriptive logging information (Narkhede, 2017).

2. **What are the advantages/inconveniences of using a logging approach?**
   The advantages of logging as defined above is that logging can provide more detail to understand where the logging statement came from, what type of log statement it is for (Error, Fatal, etc.), and in terms of errors, what the error is and where it was generated. An inconvenience in using a log approach is that it requires defining a state variable to be used throughout the program and importing dependencies to be able to access the logger, which can be annoying as it adds more lines of code than preferred.

Step #2: Setting Up Your Walking Skeleton

1. **How did you identify these abstractions?**
   I identified these abstractions for my walking skeleton from the business specifications. In the business specifications, we read the maze file path and the user guess using the i and p flags in the terminal. I thought to make a Configuration class to process these two flags. The business specifications also state that we must solve the maze. I created the MazeToGrid class so that the read in text file can be converted to a data structure for maze solving, and I created the MazeSolver class to use this data structure to solve the maze. Additionally, to process the user input and compare it to a solution path, I created a UserPathResult class to deal with this functionality. The Main class deals with the printing out to the user.

2. **What makes you think they are the right ones? Justify your design choices.**
   There are three primary tasks to make this program functional. The first is to process the flags that indicate what the user wants to do. The second is to solve the maze if there is no p flag. The third is to verify if the user's path is correct. The Configuration

class completes the first primary task of the program. The MazeToGrid and MazeSolver classes solve the maze when there is no p flag competing the third primary task of the program. And lastly, the UserPathResult will compare the user's guess to the solution from MazeSolver to see if the user is correct, completing the third primary task. Therefore, all the program tasks are completed by these design abstractions.

Step #3: Identify Features

1. **How did you identify these features?**
   My features to print are the solution path in a factored and non-factored format, to print correct path if the user's guess is correct or to print incorrect path if the user's guess is incorrect. I identified these features from the business specifications. If the p flag is not present, we are required to print out a factored and non-factored path. If the p flag is present, then we are required to print correct path for a correct user's guess or incorrect path for an incorrect user's guess.

2. **How will you ensure that they model visible value for the end user?**
   These features ensure visible value for the end user because they directly fulfil the business specifications of the program. It shows the solution path of the maze, and it shows the verification of the correctness of the user's guess.

Step #4: Minimal And Viable Product

1. **What made your MVP *viable*? Justify why your choices make your mvp release a viable one.**
   My mvp is viable because it can solve the straight path maze. By choosing to allow my program to solve a straight maze, it demonstrates that my right-hand solving algorithm is functioning correctly, and that my general program is performing the required three fundamental steps. The first step is recognizing that there is no p flag and I just solve the maze. The second step is converting the text file to a data structured maze and solving the maze. The third step is printing out a final path. The completion of these steps demonstrates that my code is functioning according to the basic business specifications, which makes my mvp viable.

2. **What made your MVP *minimal*? Justify why this version is minimal.**
   My mvp is minimal because it can only solve one maze, the straight path maze. This is minimal because it can only solve a single type of maze while still demonstrating proper business specification logic in the program.

Step #5: Algorithm Polymorphism

1. **How did you encapsulate (information hiding) your maze exploration? How does it interact with your maze representation?**
   I encapsulated my maze exploration by using a recursive implementation of the right-hand rule algorithm in the function called Solution. Solution does not return the traversal path but instead returns whether the next space is the exit tile or a space tile. In this way, Solution traverses the maze and when the exit is reached, returns true

terminating the exploration. During the traversal however, I add the cardinal direction the explorer is facing while walking the solution maze path to a private class string variable. It is then a second method that runs Solution and returns the final correct path. Therefore, if another class were to call Solution directly, they would simply get a true for finding the exit tile of the maze and not the final path. Only by calling the second function are they able to get the final path and without knowing how the path was calculated or traversed, effectively implementing encapsulation.

Firstly, it is important to understand Solution interacts with an instance of the maze representation instead of a single maze shared throughout the program. I coded my maze representation algorithm such that it can be instantiated to hide the process of how it converted the text file into a data structure but still provide the service of sharing a maze map. This enables Solution to directly interact and modify the maze data structure because it won't affect anyone else's maze map. Secondly, the behaviour and processing of Solution is based on the cardinal directions. Solution always traverses left to right, meaning it begins on the west wall of the maze facing east. Solution always checks if the current tile is the solution tile. If not, then it proceeds to turn right if there is a space and if not then it moves forward, and if it cannot do either of those things it turns left. The notion of left, right and forward are relative to an object performing the movements. But the algorithm is not an object and does not understand these relative directions. So, to represent these relative movements, Solution records the cardinal direction it was facing after every step because the cardinal directions do not change relative to the object. In a different class, these cardinal directions are processed into the relative right, left and forward notions.

2. **How is your code design when assessed from a SOLID point of view?**
   My code design is very good when assessed from a SOLID perspective. Every class in my program is given a single specific task following the single-responsibility principle. The open-closed principle is also present in my program as the classes perform their tasks well, meaning there is no modification needed to produce a new feature. Instead, classes can be extended to add more functionality and features to improve future applications of the program. Since each class has been delegated a specific task, any subclass should be able to inherit the same behaviour and improve behaviour by adding new functionality, which also helps to promote the Open-Closed principle. Since my classes are well defined and processed, there should be no conflicts regarding the Liskov principle. The interfaces I have created in my program ensure that the classes do not have to use unwanted methods. Every class implementing an interface is using all the methods and not overriding unwanted methods, demonstrating my interface segregation is done correctly. My code does not violate the dependency inversion principle as all my objects follow the one dot rule.

3. **How will your code support new algorithms? If Jelle, a new developer (an expert in graph theory), joins your team in a few weeks, how will they implement new algorithms using your code architecture?**
   I have created 4 interfaces that will enable a new developer to create their own algorithms. I have a MazeCreator interface with the method mazeCreator that returns the maze as generic type T. I made the return type generic for the developer to represent the maze using whatever data structure is most optimal for their algorithm. I have a CoordinateMethods interface with a findLeftCoordinate and

findRightCoordinate method that returns a generic type T and takes in a maze as a parameter of type K. I made the return type and parameter type different because it is possible that the maze data structure made in mazeCreator is different than the type of coordinates that the user wishes to return. I have a SolverGeneric interface which contains the solution method and finalPath methods. Solution returns type T and takes in the maze of type K and row, column, and endRow coordinate of type Q to respect the type casting of the previous coordinateMethod interface. The solution functions traverses the maze and produces a path, which can be returned or not, depending on the algorithm and its implementation. The finalPath forcibly returns a String as this is the easiest type to process printing the user path. The interface PrinterBusinessSpec contains the pathCalculation and factored functions which both return a String that is printed to the user. Altogether, these interfaces provide the blueprints for another developer to fully implement a new maze solving algorithm.

Step #6: Module Interface Specification

1. **Which information from your MIS can easily be reflected in the Java source code? How?**
   The SolverGeneric interface is currently implemented in my Java source code. I wrote this interface intentionally to support algorithm polymorphism so future developers can write their own algorithms. As a result, my solution and finalPath functions and solutionPath variable are implemented and functioning exactly as intended. Additionally, when debugging the functionality for the implementation of my right-hand algorithm, I never encountered unexpected errors resulting me to conclude my code was sound enough that no exceptions will have to be thrown.

2. **Which information cannot be reflected immediately? Which workaround can you imagine?**
   As I stated in the previous question of the report, I have implemented the interface and MIS in my program. This means that all the information is reflected immediately, resulting in no work arounds to be performed.

References

n.a., n. a. (2023a, October 27). *Apache Commons CLI*. Apache Commons. https://commons.apache.org/proper/commons-cli/#:~:text=The%20Apache%20Commons%20CLI%20library,for%20example%20tar%20%2Dzxvf%20foo.

n.a., n. a. (2023b, October 27). *Using Apache Commons CLI*. Apache Commons. https://commons.apache.org/proper/commons-cli/usage.html

Narkhede, P. (2017, October 25). *WHY LOG4J IS BETTER THAN SYSTEM.OUT.PRINTLN? (WHAT IS DIFFERENCE BETWEEN LOG4J AND COMMON LOGGINGS?)*. Automation Talks. https://automationtalks.com/2017/06/04/why-log4j-is-better-than/

Appendix A

Module: SolverGeneric

Exported Routines:

Routine	In	Out	Exception
new solutionPath	--	String	--
solution	K, Q	T	--
finalPath	--	String	--

Semantics:

- **State Variable(s)**
  - solutionPath: String
  - row: Integer
  - column: Integer
- **State Invariant(s)**
  - row >= 0, row < |maze row amount|
  - column >= 0, column < |maze column amount|
  - maze = $\neg empty$
- **Assumption(s)**
  - There is an entry on the left wall and an exit on the right wall and vice versa. The row coordinate, column coordinate, and final row coordinate are all the same type.
- **Access Routine Semantics**
  - *solution(K mazeInput, Q row, Q column, Q endRow): T*
    - *Transition:* $solutionPath := solutionPath_{prev} + solutionPath_{new}$
    - *Output:* $out := mazeInput[row][column] = exit$
  - *finalPath(): String*
    - *Output: solutionPath: String*