COVER PAGE

Introduction: This assessment demonstrates the application of modular software design, version control, and rigorous testing methodologies to enhance the output of a maze-generating program. The project involves creating a more readable maze display using box-drawing characters by implementing a modular design that includes reading the maze from a file, converting walls into box-drawing characters, and formatting the final maze for console output.

A robust version control strategy is used, with branches such as MAZE, MAZEreader, MAZEconverter, and MAZEprinter showcasing modular development. Testing branches like MAZEreaderTest and MAZEconverterTest ensure functionality and reliability. This structured approach highlights the principles of good software engineering practices, from modularity and maintainability to comprehensive testing.

VERSION CONTROL

Branches used: MAZE, MAZEreader, MAZEconverter, MAZEprinter, MAZEreaderTest, MAZEconverterTest.

Why needed:

MAZE: Acts like a main branch where all the other branches are merged.

MAZEreader: Contains readMazeFromFile() method, which handles reading and loading of the maze from the file. This branch handles reading and loading the maze from a file.

MAZEconverter: Contains ConvertWallsToBoxDrawing() method and determineBoxDrawingChar() method, which converts maze walls to box-drawing characters. This branch is dedicated to wall conversion and box-drawing logic.

MAZEprinter: Contains printMAZE() method, which prints the maze to the console. This branch focuses on formatting and displaying the maze.

MAZEreaderTest: This branch is dedicated to testing the functionality of the main module MAZEconverter.

MAZEconverterTest: This branch is dedicated to testing the functionality of the main module MAZEreader.

MERGING STRATEGY:-

Make branches for each module and test and them all in the main branch, which is called MAZE

Modularity Design

Module 1: main

- **Purpose**: To serve as the entry point for the application, orchestrating the reading, conversion, and display of the maze.
- **Input**: None (though it reads from a file, maze_output.txt).
- **Output**: Printed ASCII representation of the maze on the console.
- **Design Decision**: The main method centralizes the sequence of operations, including error handling, to ensure the program can attempt all steps and handle any file-related issues. This keeps the process manageable and provides a single entry point to test the entire application flow.

Module 2: readMazeFromFile

- **Purpose**: To read the maze structure from a file and load it into memory for further processing.
- **Input**: String filename (in this case, "maze_output.txt").
- **Output**: A 2D array representing the maze structure, typically stored in a class-level variable.
- **Design Decision**: Reading the maze from a file separates file I/O from logic, allowing the file format to be handled independently of the maze processing. This approach makes it easier to change the file input format or location in the future if needed without affecting other modules.

Module 3: ConvertWallsToBoxDrawing

- **Purpose**: To convert wall cells in the maze to ASCII box-drawing characters, making the maze more visually intuitive.
- **Input**: A 2D array of maze cells.
- **Output**: A modified 2D array where wall cells have been replaced with box-drawing characters.
- **Design Decision**: The ConvertWallsToBoxDrawing module is separate from both file reading and display, allowing for easier maintenance of conversion logic. It ensures that the wall-conversion logic is encapsulated, making it easier to debug, modify, or test independently. This module is needed to make the maze more appealing by converting walls (#) to box-drawing characters.

Module 4: determineBoxDrawingChar

- **Purpose**: Determines the appropriate box-drawing character based on the surrounding cells (walls and paths).
- **Input**: Current cell coordinates (i, j).
- **Output**: Box-drawing character.
- **Design Decision**: The module checks the surrounding cells to identify the right character for drawing smooth connections between cells.

Module 5: printMaze

- **Purpose**: Prints the generated maze to the terminal.
- **Input**: None.
- **Output**: Displays the maze in the terminal.
- **Design Decision**: This input-output operation completes the program by displaying the final maze output.

CHECKLIST FOR MODULARITY IMPLEMENTATION			
Module	**Checklist Compliance**	**Issues Found**	**Actions for Improvement**
MAZEreader	Single Responsibility: Yes Clear Interface: Yes Encapsulation: Yes Error Handling: Partial	Graceful error handling for missing or empty files could be improved.	Add fallback behaviour or default outputs for missing/empty files.
MAZEconverter	Single Responsibility: Yes Testability: Partial Documentation: Partial Clear Interface: Yes	Some edge cases in boxdrawing conversion are not handled; there is a lack of inline documentation.	Implement additional test cases and provide detailed comments for the logic.
MAZEprinter	Single Responsibility: Yes Testability: Yes Clear Interface: Yes Documentation: Yes	None	No improvements were necessary.
MAZE (Main)	Single Responsibility: Partial (orchestrates multiple functions) Error Handling: Partial	Error messages are printed but do not provide fallback options for certain errors.	Refactor to delegate more tasks to specialized modules and include default behaviour for errors.

TEST DESIGN

Black Box testing (Equivalence partitioning)

MAZE

Category	Test Data	Expected Result	Reasoning
Valid file	"ValidMaze.txt"	Properly transformed and printed	Ensures end-to-end processing works for valid files
Invalid file	"nonexistent.txt"	Error properly handled	Invalid files must not crash the program

MAZEreader

Category	Test Data	Expected Result	Reasoning
The file does not exist	"Nonexistent.txt"	IOException or error message	File not found must be handled
File is empty	"empty.txt"	Empty 2D array or meaning error	Empty files must not crash the program
The file contains invalid data	"invalid.txt"	Error message	Non-maze data must be gracefully rejected
File contains a valid maze	"ValidMaze.txt"	Valid 2D char array representing the maze	Valid maze files must be correctly parsed into 2D array

MAZEconverter

Category	Test Data	Expected Result	Reasoning
No walls in the maze	Input:[[' ', ' '], [' ', ' ']]	Output: unchanged	Empty mazes must remain unaltered.
No paths in the maze	Input: [['#', '#'], ['#', '#']]	Walls converted to ['┼', '┼'], ['┼', '┼']	The conversion ensures all walls are updated.
Mixed maze	Input: [['#', ' '], ['#', '#']]	Walls converted based on neighbours	Handles mixed inputs as per connection rules.

MAZEprinter

Category	Test Data	Expected Result	Reasoning
Empty maze	Input: []	No output	Empty inputs must not cause crashes.

Non-empty maze	Input: [['┼', '┼'], [' ', ' ']]	Outputs maze row by row (┼┼ followed by a space)	Ensure formatted maze printing works.

White Box Testing

MAZEreader

Test path	Test data	Expected Result
File does not exist	“nonexistent.txt”	IOException
The file exists but is empty	“empty.txt”	Return an empty array or provide a meaningful error
File exists, contains a valid maze	“ValidMaze.txt”	Correctly parsed 2D char array
The file exists, contains invalid content	“invalid.txt”	Error or fallback to handling, empty array or appropriate error message

- **File exists → Empty**: ○ Test Value: "empty.txt".
  - ○ Reason: Exercises the branch where the file exists but contains no data.
- **File does not exist**: ○ Test Value: "nonexistent.txt".
  - ○ Reason: Tests the branch for error handling when the file is not found.

- **File exists → Non-empty → Valid parsing**:
  ○ Test Value: "validMaze.txt"
  - ○ Reason: Ensures proper parsing when the file exists and contains valid maze data. Exercises the parsing logic and iterative loops.

- **File does exist**: ○ Test Value: "invalid.txt". ○ Reason: Detects symbols in the input file.

MAZEconverter

Test-Path	Test Data	Expected Result	Reasoning
No neighbour walls	Input: [['#']]	Converts to '|'	Tests standalone wall conversion.
The wall surrounded on all sides	Input: [['#', '#'], ['#', '#']]	Converts to ['┼', '┼'], ['┼', '┼']	Ensures correct junction conversion.
Mixed neighbours	Input: [['#', ' '], ['#', '#']]	Converts appropriately	Tests mixed path-wall handling.

- **Single isolated wall**: ○ Test Value: [['#']].
  ○ Reason: Tests the conversion logic for a wall with no neighbours. Exercises the branch where no connections exist.
- **Wall surrounded on all sides**: ○ Test Value: [['#', '#'], ['#', '#']].
    - Reason: Exercises the condition where all four sides have neighbours, resulting in a ┼ character.
- **Mixed neighbou rs**: ○ Test Value: [['#', ' '], ['#', '#']].
    - Reason: Covers paths where walls have partial connections (e.g., corner or Tjunction).

Test Implementation

Codes are in the zip file

-MAZEconverterTest

-Faced two failures, one in whiteBoxTestConversionLoop and others in BlackBoxTestConversion

-Tried to improve by changing the main MAZEconverter.java code and MAZEconverterTest.java code, but couldn't fix the error properly.

Summary of Work

Module Name	Module Complete?	Test Designed?	Test implemented?	Test successful?
Main	yes	Yes	yes	Yes
readMazeFromFile	yes	Yes	yes	Yes
ConvertWallsToBoxDrawing	yes	Yes	yes	Partially
determiningBoxDrawingChar	yes	Yes	Yes	Yes
printMaze	yes	yes	Yes	yes

Challenges Faced

1. **White Box Test Failures**: Faced issues with conversion logic during white-box testing for edge cases in the ConvertWallsToBoxDrawing module.
2. **Handling Invalid Maze Files**: Error handling for malformed maze files required significant adjustments.
3. **Version Control Merging Conflicts**: Resolved merging conflicts during branch integration, particularly with test branches.
4. Couldn't perform test implementation for MAZEreaderTest as the file was not being read properly.

Limitations

1. The ConvertWallsToBoxDrawing module still has unresolved edge errors that may lead to incorrect junction formatting.
2. Limited time for exhaustive testing across all possible maze configurations. **Improvements**

1. Introduce comprehensive logging for better insight into test failures.
2. Update the design to handle dynamic maze dimensions efficiently.
3. Implement additional test scenarios to cover edge cases and improve robustness.