**CS 1120 Computer Science II (with Java),** *Spring 2019*

Department of Computer Science, Western Michigan University

Instructor: *Fola Olagbe* Lab TA: *Hisham Saleh*

Ryan Hopkins

**SOFTWARE LIFE CYCLE REPORT – FOR LAB ASSIGNMENT 2**

# PHASE 1: SPECIFICATION

Create an application Maze Solver that can:

1. Generate a maze (a 2d array of strings)
   1. Number of rows and columns are given by the user and must be in the range [5,10]. Number of rows cannot equal number of columns.
   2. Must validate input. If input is invalid, repeatedly ask for valid input.
2. Randomly fill the maze with numbers in the range [1,100] and #’s.
   1. The number is the amount of coins the robot can pick up by passing through that cell.
   2. A ‘#’ represents a blocked position the robot cannot pass through.
   3. ‘S’ is the starting point for the robot and is at the top left corner of the maze.
   4. ‘D’ is the robot’s final destination (the end of the maze) and is at the bottom right corner of the maze.
   5. **The total number of #’s cannot be greater than a third of the total number of positions in the maze.**
3. Print out the maze to the screen once it is generated.
4. Attempt to find an available path from S to D.
   1. The robot can move up, right, left, or down.
   2. Use A\* pathfinding algorithm.
   3. **This program must use recursion.**
5. If the robot is not able to find a path, print a corresponding message to the screen.
6. If the robot finds a path, then:
   1. Print out the maze with the path marked by +’s.
   2. Sum up all the numbers on this path and print out the result as the number of coins found.
7. **This program is required to run steps 2 – 6 above 3 times.**

**Example output:**

Enter number of rows in range [5, 10]:

1

Invalid input!

Enter number of rows in range [5, 10]:

a

Input must be an integer. Re-enter:

5

Enter number of columns in range [5, 10].

This must be different from number of rows:

5

Number of columns is the same as number of rows!

Enter number of columns in range [5, 10].

This must be different from number of rows:

6

\*\*\*\*\*\*Maze #1\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # 6 # 4 100

27 78 # # 51 #

27 # # # 3 #

47 29 47 # 53 #

32 29 48 56 47 D

Congratulations! I found a solution for this maze as below:

S # 6 # 4 100

+ 78 # # 51 #

+ # # # 3 #

+ 29 47 # 53 #

+ + + + + D

The amount of coins collected: 313

\*\*\*\*\*\*Maze #2\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # # # 86 47

83 # 22 # 34 #

# 24 48 # # 17

# 31 # 24 67 1

21 77 99 87 95 D

Sorry, no solution can be found for this maze!

\*\*\*\*\*\*Maze #3\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # 57 71 5 37

88 # # 72 # #

# 68 79 # # #

88 69 47 # # #

55 35 89 99 26 D

Sorry, no solution can be found for this maze!

Test the application by performing the following steps:

1. Enter invalid inputs for number of rows / columns and ensure program asks user to re-enter inputs if they are invalid.
2. Ensure application generates correct output for mazes based on example output above.
   1. Ensure application finds path if one exists.
   2. Ensure program outputs “Sorry, no solution can be found for this maze!” if one does not.

# PHASE 2: DESIGN

**2.1 Modules and Their Structure**

The program will have two classes: the class Maze acts as a template for Maze type objects and the class MazeSolver instantiates these objects. These two classes are not related by the class hierarchy (none is derived from the other).

The two classes are specified below by the **UML diagram**.

|  |  |  |
| --- | --- | --- |
| MazeInput |  | LA2Main |
|  |  |  |
| + initializeMazeSolver(solver: MazeSolver): void  + getNumRows(): int  + getNumCols(numRows: int): int |  | + main(args: String[]): void |

|  |
| --- |
| MazeSolver |
| String[][] maze  boolean[][] visited  int numRows  int numCols |
| + drawMaze(): void  + printMaze(): void  + isSolvable(): Boolean  + solveMazeRecursively(row: int, col: int): Boolean  + printResult(): void  + setNumRows(numRows: int): void  + getNumRows(): int  + setNumCols(numCols: int): void  + getNumCols(): int |

**2.2 Pseudocode for the Modules**

**2.2.1 Pseudocode for** LA2Main

1a) LA2Main Pseudocode Refinement #1:

// instantiate MazeSolver object

// initialize MazeSolver object using MazeInput.initializeMazeSolver

// print maze

// determine if maze is solvable

// if maze is solvable

// solve maze

// print result

// if maze is not solvable

// print appropriate message

// repeat 3 times

**2.2.2 Pseudocode for** MazeInput

2a) MazeInput Pseudocode Refinement #1:

// --METHODS--

// initializeMazeSolver(solver: MazeSolver): void

// getNumRows(): int

// getNumCols(numRows: int): int

2b) MazeInput Pseudocode Refinement #2:

// --METHODS--

// initializeMazeSolver(solver: MazeSolver): void

// initialize numRows and numCols in MazeSolver

// initialize maze parallel arrays in MazeSolver

// getNumRows(): int

// asks for a number in range [5, 10] from the user as the number of rows

// validate input

// return valid input

// getNumCols(numRows: int): int

// asks for a number in range [5, 10] from the user as the number of cols

// validate input (cannot be the same as numRows)

// return valid input

**2.2.3 Pseudocode for** MazeSolver

3a) MazeSolver Pseudocode Refinement #1:

// --FIELDS--

// String[][] maze

// boolean[][] visited

// int numRows

// int numCols

// --METHODS--

// drawMaze(): void

// printMaze(): void

// isSolvable(): Boolean

// solveMazeRecursively(row: int, col: int): Boolean

// printResult(): void

// setNumRows(numRows: int): void

// getNumRows(): int

// setNumCols(numCols: int): void

// getNumCols(): int

3b) MazeSolver Pseudocode Refinement #2:

// --FIELDS--

// String[][] maze <- array with coins and obstacles

// boolean[][] visited <- parallel array to maze stores visited cells

// int numRows

// int numCols

// --METHODS--

// drawMaze(): void

// initialize maze parallel arrays

// create maze and set to maze field

// fill cells with “#” and random numbers from 1-100

// set top left cell to “S” (starting point)

// set bottom right cell to “D” (destination)

// total number of “#”s cannot exceed 1/3 of total cells

// printMaze(): void

// print out maze

// isSolvable(): Boolean

// calls solveMazeRecursively(row, col) and prints result

// solveMazeRecursively(row: int, col: int): Boolean

// if maze is already solved

// returns false if there is no solution and true otherwise

// if there is a solution

// solves maze recursively using A\*

// replaces cells in shortest path from S to D with + in maze field

// record number of coins collected

// printResult(): void

// prints solved maze along with appropriate message

// setNumRows(numRows: int): void

// getNumRows(): int

// setNumCols(numCols: int): void

// getNumCols(): int

**PHASE 3: RISK ANALYSIS**

No risks (to timetable, cost, human health, etc.) are identified by me.

# PHASE 4: VERIFICATION

The algorithm has a recursive execution path. Correctness of this path has been verified by me by analyzing its steps, and their completeness w.r.t. the Specification.

# PHASE 5: CODING

## 5a) Code Refinement #1

**File LA2Main.Java:**

**package** maze\_solver;

**public** **class** LA2Main {

**public** **static** **void** main(String[] args) {

MazeSolver solver = **new** MazeSolver();

// gets row and col data from user, creates maze, sets fields in MazeSolver

MazeInput.*initializeMazeSolver*(solver);

solver.printMaze();

**if** (solver.isSolvable()) {

solver.solveMazeRecursively(solver.getNumRows(), solver.getNumColumns());

solver.printResult();

} **else** {

System.***out***.println("Sorry, no solution can be found for this maze!");

}

}

}

**File MazeInput.Java:**

**package** maze\_solver;

**public** **class** MazeInput {

// I chose to make these methods static because they act more like functions

**public** **static** **void** initializeMazeSolver(MazeSolver solver) {

}

**private** **static** **int** getNumRows() {

**return** 0;

}

**private** **static** **int** getNumCols() {

**return** 0;

}

}

**File MazeSolver.Java:**

**package** maze\_solver;

**public** **class** MazeSolver {

**private** String[][] maze;

**private** **boolean**[][] visited;

**private** **int** numRows;

**private** **int** numColumns;

**public** **void** drawMaze() {

}

**public** **void** printMaze() {

}

**public** **boolean** isSolvable() {

**return** solveMazeRecursively(**this**.getNumRows(), **this**.getNumColumns());

}

**public** **boolean** solveMazeRecursively(**int** row, **int** col) {

**return** **true**;

}

**public** **void** printResult() {

}

// Getters and setters

**public** **int** getNumRows() {

**return** numRows;

}

**public** **void** setNumRows(**int** numRows) {

**this**.numRows = numRows;

}

**public** **int** getNumColumns() {

**return** numColumns;

}

**public** **void** setNumColumns(**int** numColumns) {

**this**.numColumns = numColumns;

}

}

## 5b) Code Refinement #2

**File LA2Main.Java:**

**package** maze\_solver;

**public** **class** LA2Main {

**public** **static** **void** main(String[] args) {

MazeSolver solver = **new** MazeSolver();

// gets row and col data from user, creates maze, sets fields in MazeSolver

MazeInput.*initializeMazeSolver*(solver);

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println("\*\*\*\*\*\*Maze #" + i + "\*\*\*\*\*\*");

System.***out***.println("Start drawing the maze...");

System.***out***.println("The maze is as below:");

solver.printMaze();

**if** (solver.isSolvable()) {

solver.solveMazeRecursively(solver.getNumRows(), solver.getNumColumns());

solver.printResult();

} **else** {

System.***out***.println("Sorry, no solution can be found for this maze!");

}

System.***out***.println();

}

}

}

**File MazeInput.Java:**

**package** maze\_solver;

**import** java.util.Scanner;

**public** **class** MazeInput {

// I chose to make these methods static because they act more like functions

**public** **static** **void** initializeMazeSolver(MazeSolver solver) {

// Instantiated reader in initializeMazeSolver method because

// closing a Scanner object also closes the input source

Scanner reader = **new** Scanner(System.***in***);

solver.setNumRows(*getNumRows*(reader));

solver.setNumColumns(*getNumCols*(reader, solver.getNumRows()));

solver.drawMaze();

reader.close();

}

**private** **static** **int** getNumRows(Scanner reader) {

**int** numRows = -1;

**while** (numRows == -1) {

System.***out***.println("Enter number of rows in maze in range [5, 10]:");

**try** {

**int** input = Integer.*parseInt*(reader.nextLine());

**if** (input < 5 || input > 10) {

**throw** **new** IllegalArgumentException("Input must be between 5 and 10");

} **else** {

numRows = input;

}

} **catch** (NumberFormatException e) {

System.***out***.println("Input must be an integer");

} **catch** (IllegalArgumentException e) {

System.***out***.println(e.getMessage()); // Input not in range

}

System.***out***.println();

}

**return** numRows;

}

**private** **static** **int** getNumCols(Scanner reader, **int** numRows) {

**int** numCols = -1;

**while** (numCols == -1) {

System.***out***.println("Enter number of columns in maze in range [5, 10]:");

System.***out***.println("Number of columns cannot be equal to number of rows entered above");

**try** {

**int** input = Integer.*parseInt*(reader.nextLine());

**if** (input < 5 || input > 10) {

**throw** **new** IllegalArgumentException("Input must be between 5 and 10");

} **else** **if** (input == numRows) {

**throw** **new** IllegalArgumentException("Input cannot be equal to number of rows entered above");

} **else** {

numCols = input;

}

} **catch** (NumberFormatException e) {

System.***out***.println("Input must be an integer");

} **catch** (IllegalArgumentException e) {

System.***out***.println(e.getMessage()); // Input not in range or input==numRows

}

System.***out***.println();

}

**return** numCols;

}

}

**File MazeSolver.Java:**

**package** maze\_solver;

**public** **class** MazeSolver {

**private** String[][] maze;

**private** **boolean**[][] visited;

**private** **int** numRows;

**private** **int** numColumns;

**public** **void** drawMaze() {

**int** totalCells = getNumRows() \* getNumColumns();

**int** totalObstacles = 0;

**this**.maze = **new** String[getNumRows()][getNumColumns()];

**this**.visited = **new** **boolean**[getNumRows()][getNumColumns()];

**for** (**int** i = 0; i < getNumRows(); i++) {

**for** (**int** j = 0; j < getNumColumns(); j++) {

**if** (totalObstacles >= totalCells / 3) {

**this**.maze[i][j] = String.*valueOf*((**int**) (Math.*random*() \* 99 + 1));

} **else** {

**if** (Math.*random*() < .25) {

**this**.maze[i][j] = "#";

totalObstacles++;

} **else** {

**this**.maze[i][j] = String.*valueOf*((**int**) (Math.*random*() \* 99 + 1));

}

}

}

}

**this**.maze[0][0] = "S";

**this**.maze[getNumRows() - 1][getNumColumns() - 1] = "D";

}

**public** **void** printMaze() {

**for** (**int** i = 0; i < getNumRows(); i++) {

**for** (**int** j = 0; j < getNumColumns(); j++) {

System.***out***.print(maze[i][j] + " ");

}

System.***out***.println();

}

}

**public** **boolean** isSolvable() {

**return** solveMazeRecursively(**this**.getNumRows(), **this**.getNumColumns());

}

**public** **boolean** solveMazeRecursively(**int** row, **int** col) {

**return** **true**;

}

**public** **void** printResult() {

System.***out***.println("Congratulations! I found a solution for this maze as below:");

printMaze();

System.***out***.println("The amount of coins collected:");

}

// Getters and setters

**public** **int** getNumRows() {

**return** numRows;

}

**public** **void** setNumRows(**int** numRows) {

**this**.numRows = numRows;

}

**public** **int** getNumColumns() {

**return** numColumns;

}

**public** **void** setNumColumns(**int** numColumns) {

**this**.numColumns = numColumns;

}

}

## 5c) Code Refinement #3

# PHASE 6: TESTING

This program has multiple execution paths. It will be necessary to derive tests for entering valid and invalid input. Invalid input that must be tested for includes: entering the wrong type of data, entering numRows equal to numCols, and entering numbers not in the appropriate range.

I verified that this output satisfies program requirements.

**PHASE 7: REFINING THE PROGRAM**

No refinements are needed. In this program, I have already included all required features.

# PHASE 8: PRODUCTION

I prepared a copy of the entire program for Lab TA’s evaluation, as specified by the TA. Then, I sent electronically the copy to the Lab TA.

# PHASE 9: MAINTENANCE

To fully benefit from the program evaluation feedback received from the Lab TA, I will perform program maintenance. This means that I should use all TAs feedback to improve my program.

© 2012-2019 by Leszek T. Lilien and Jeremy Doornbos