Obligatory assignment 2

5DV133 Object oriented programming

Thomas Sarlin <thsa0043/id15tsn>
thsa0043@gapps.umu.se

Teachers:
Anders Broberg
Nicklas Fries
Adam Dahlgren
Jonathan Westin
Erik Moström
Alexander Sutherland

Date: 19/4 – 2016

Additional files: thsa0043ou2.zip

Introduction

In the course Object oriented programing we are set to create a guiding system for theoretical robots. Involving file-handling and reading. The assignment is to create three main classes “Maze” “Position” and “Robot”. From which we are to create the guiding system. The maze should be created from an input file with a specified structure. The Robot-class is an abstract class used to create different types of robots. To specify how each individual robots movement differs from others they all have their own “move” method. To navigate through the maze the use of the Position class is applied. Which is used to specify the x and y coordinates of a specified Position in the maze, and also check which positions are to the north, south, east and west. We are to implement at least two robots, one that follows the right hand rule and the other that is a memory robot that uses the knowledge of where it has been to do a depth first search of the maze.

Table of Contents

Executive instructions trough terminal

1. Unzip the archive “thsa0043ou2.zip” in a folder you are able to reach.

2. Guide yourself to the folder where the extracted files are located trough the commando:
   cd “extraction folder../thsa0043ou2/”

3. When located in the correct folder you can write the commando “ls” to see which files that are located in the folder.

4. Compile the specified test program by running:
   javac RobotTest.java

5. To run the program you write the following:
   java RobotTest “testMaze/maze1.txt” “testMaze/maze2.txt” … “testMaze/mazeN.txt”

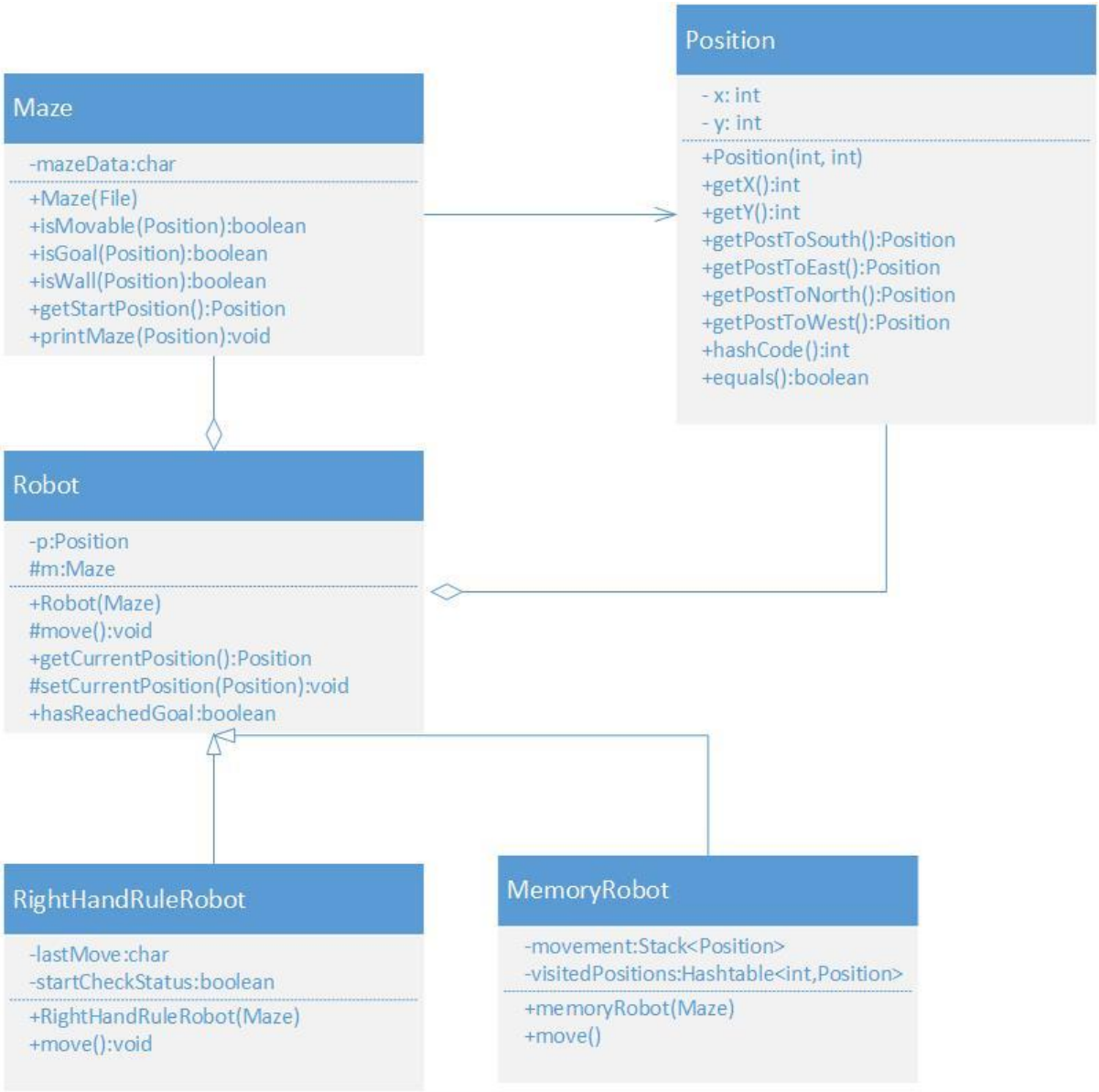
   The maze-arguments are Files with information on how the maze is constructed. There is no limit on the number of mazes you can add to the program. But for easier usage no more than 10 is recommended. To see how to implement your own mazes check the system description. There are currently 6 premade mazes in the testMaze folder that is included in the zip-archive, labelled maze1.txt to maze6.txt. To see how they are mapped, check the test chapter.

6. After running just follow the instructions in the test-program. To see how the test-program works check the test-program chapter.

7. **Additionally there are jUnit test programs.**
   <u>If you want to compile these you need to locate your jUnit.jar and run following:</u>
   javac –cp “../junit.jar” MazeJunitTest.java
   or
   javac –cp “../junit.jar” PositionJunitTest.java

8. Run by writing:
   java MazeJuniTest
   or
   java PositionJunitTest

Class Diagram



System description

This is a system made to let a robot guide itself through a maze. By dividing the responsibilities as follows.

Position:
Responsible of keeping track of coordinates of specific points in the Maze class, should also be able to know which coordinates that are adjacent to the specified position. Used to mainly to check if a specific point in the maze is movable or end-position. Also used in the Robot-subclasses class to keep track and set the robots current position.

Maze:
Responsible to keeping track of the mazes structure, if the maze-file is corrupt or implemented wrong it is responsible for throwing and alerting the user. Maze should also be able to check if a position is start/goal, movable position or a wall.
Correct structure of maze-file:
1) Walls are indicated by asterisk (*)
2) Movable locations are indicated by space.
3) Should have a start-point, indicated by an 'S'.
4) Should have at least one goal, indicated by 'G'

Example:

```
*S*********
*         *
* ****  ****
* *       *
*********G*
```

Robot:
The robot has to move through the maze in a specified pattern, the pattern is implemented by the programmer. I will show two different types of robots in the following section.
The responsibilities of the robot is to keep track of its current position in the maze and move in the specified pattern. The limits of the robots should be specified.

Implementations of robots

RightHandRuleRobot:

It follows the right hand rule, i.e. it puts its theoretical right hand on the wall and moves forward never letting go off the wall until it reaches the end of the maze. If the start-location is not located near a wall, the robot will move north to the closes wall and then start following the right-hand rule pattern.
Restrictions: Goal needs to be a part of the same wall structure as the start.

MemoryRobot:

Memory Robot uses a hash table to keep track of the positions that have been visited before and a stack to be able to backtrack when a dead-end is reached.
It follows the depth first algorithm, the robot moves forward and pushes each move onto the stack. Always checking adjacent positions for movable positions. If the robot reaches an intersection with multiple movable positions it puts all of those positions on the stack and keeps moving to the route last added to the stack. If the robot reaches a dead end the robot starts to pop values from the stack to back-trace until the last intersection then follows the next route in the intersection repeating the pattern until goal is reached. If the stack becomes empty it's an indication that the maze probably has no goal with a connecting path to the start-position.

Restrictions: Is able to comprehend mazes that are constructed according to the criteria mentioned in the "maze" part of the section. Mazes larger than 36x123 elements has not been tested, no guarantees are given that it will work with larger mazes but in theory it should work fine.

Classes

A brief description of the public methods and constructors, some private methods that need some explanation are listed. There are additional private methods beside those listed who are mainly to make the code easier to read and help the public methods.

Position.java

Constructors:

Position(int x, int y) – Creates a Position with an x & y value.

Public methods:

getX() – Returns an integer representing the value of x

getY() – Returns an integer representing the value of y

getPosToSouth() – Returns a new Position with the x & y values of the position south of current position.

getPosToNorth() – Returns a new Position with the x & y values of the position north of current position.

getPosToEast() – Returns a new Position with the x & y values of the position east of current position.

getPosToWest() – Returns a new Position with the x & y values of the position west of current position.

equals(Position p) – Compares current position to Position p, returns true if they are equal.

hashCode() – Returns an integer representing the hash-value of the position.

Maze.java

Constructors:

Maze(File inputFile) – Reads in a file and converts it into a two-dimensional char-array. Uses other private methods checkMaze, calculateMaze and setMaze. To be able to throw invalid file structures, calculate the size of the array to avoid array exceptions and finally to add the characters to the array.

Public methods

isMovable(Position p) – checks if Position p is a movable Position, i.e the robot can move there, indicated with a space.

isGoal(Position p) – checks if Position p is goal, indicated with a 'G'

isWall(Position p) – checks if Position p is a wall, indicated with an asterisk

getStartPosition() – scans the maze for the start-position indicated with an 'S', when/if position is found it is returned.

printMaze(Position p) – Prints out the current maze with an Position p marked with an 'X', used to represent the maze and a Position you want to indicate. Used by Robot to show the current Position/movement of the robot.

Robot.java – Abstract class

Constructors:

Robot(Maze m) – Creates a robot with a specific maze and sets the current position of the robot to the start-position in the maze.

Public methods:

getCurrentPosition() – returns the current Position.

setCurrentPosition(Position p) – sets the current position to Position p.

hasReachedGoal() – Returns a Boolean representing if the robot has reached goal.

Abstract methods:

move() – Must be implemented by each sub-class, should move the robot one step in a pre-determined pattern set by the user.

Protected methods:

getNorth,getEast,getSouth,getWest – Methods to get a position from an adjacent direction, used to make the code easier to read in the implemented robots.

RightHandRuleRobot.java

Inherits from Robot

Constructors:

RightHandRuleRobot(maze) – inherited from Robot

Public methods:

Move() – Moves the robot one step in the right hand rule pattern. Uses several private functions to be able to move and make the code more understandable.

Algorithm:

Private methods:

startCheck() – checks if the robot is adjacent to a wall, if not the robot should move north until it reaches a wall(returns a boolean that is used by move, when this returns true the robot starts using the right hand rule).

tryGoing"direction" – several functions to try moving in a set direction, catches out of bound exceptions so that the program doesn't crash when the robot checks outside of the maze.

MemoryRobot.java

Inherits from Robot

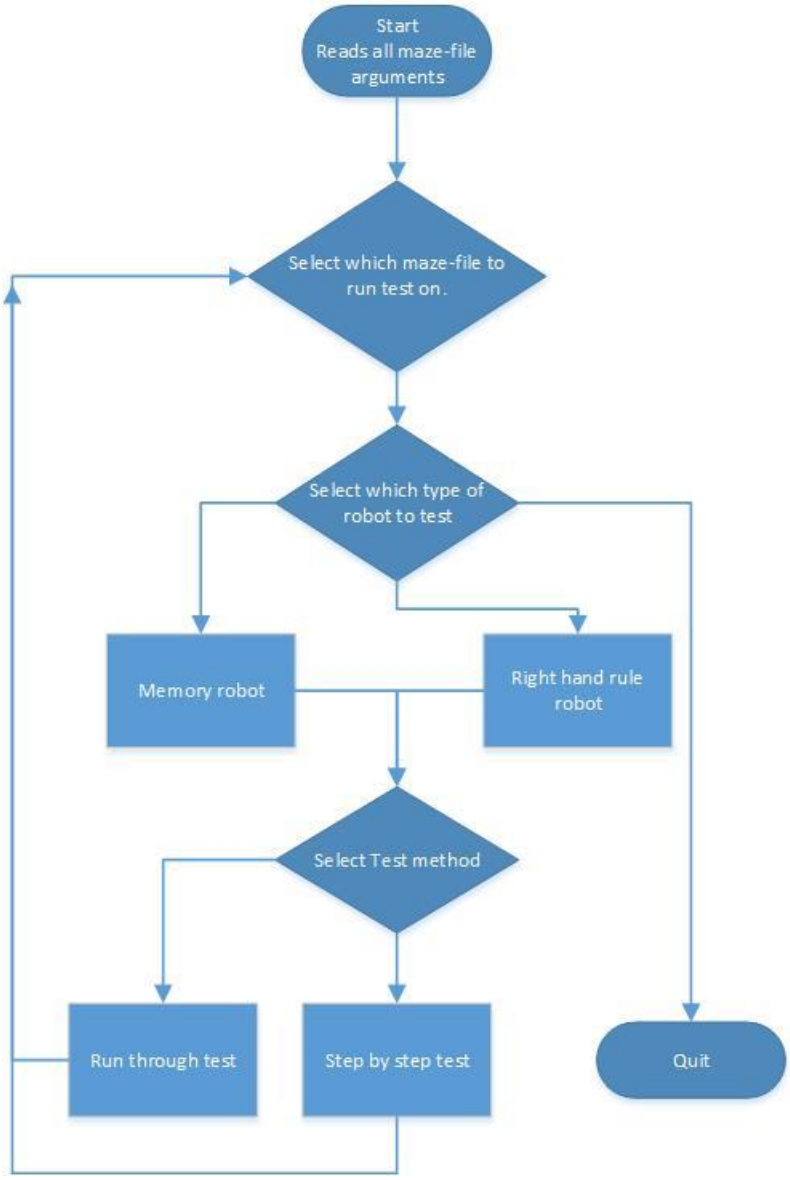
Constructors:

RightHandRuleRobot(maze) – inherited from Robot

Public methods:

Move() – Moves the robot according to the depth first algorithm, uses a stack to keep track of the recent movements of the robot and a hash table to keep track of which movables that have been visited.

Flowchart for robotTest program



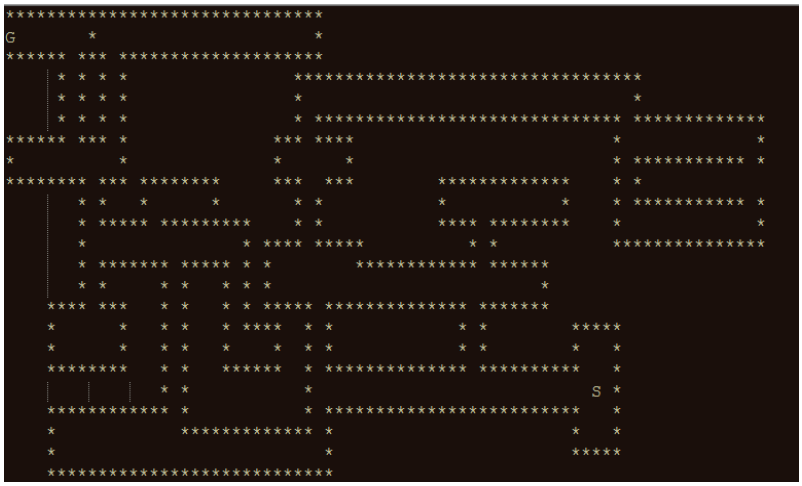
Test runs

Specified mazes

Maze1.txt

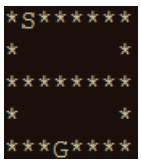
Maze2.txt

Maze3.txt

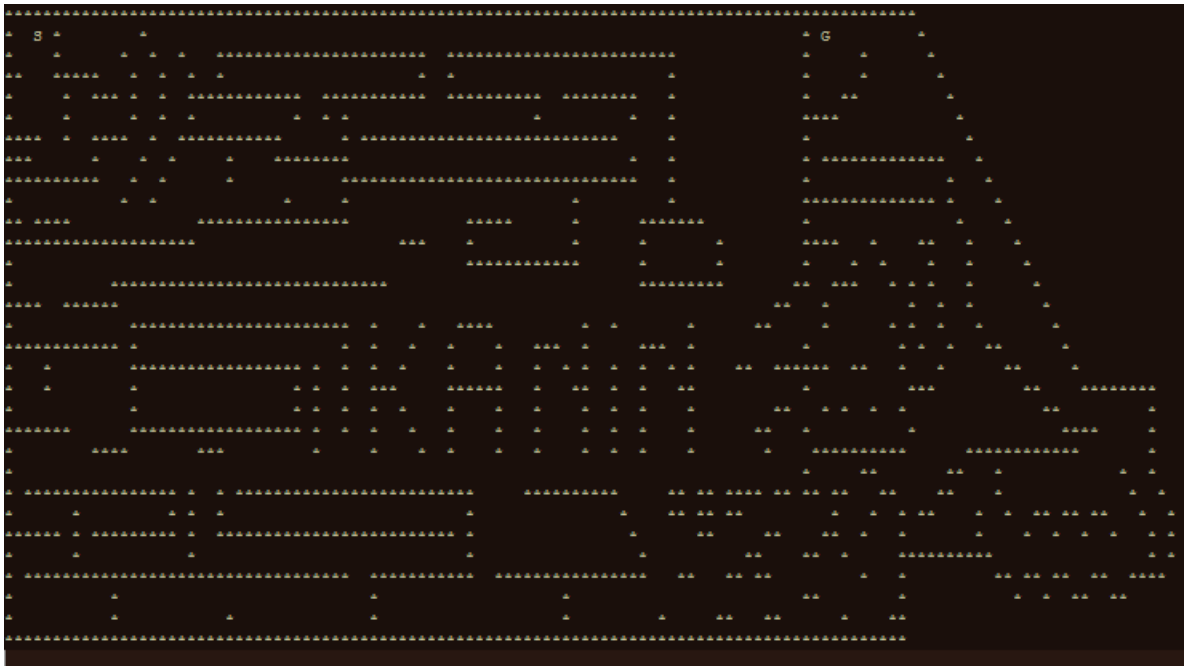


Maze4.txt

Maze5.txt



Maze 6



Run through test

The run through test is designed to force the robot to keep going until it finds the goal in the maze and returns in how many moves it was able to do so. Results are as follows:

Robot	Maze1.txt	Maze2.txt	Maze3.txt	Maze4.txt	Maze5.txt	Maze 6
RightHandRule	159 moves	61 moves	396 moves	Robot goes into an Inf. loop	Robot goes into an Inf. loop	1026 moves
Memory	284 moves	34 moves	312 moves	106 moves	Complains about no goal reachable, test program goes into inf. Loop not the robot itself.	6413 moves, really ineffective at larger mazes with wide paths as the robot "rolls" forward in wide spaces.

As you see the RightHandRule robot has some limitations. It cannot find a goal that is disconnected from a wall compared to the memory robot that checks all possible options along the way until goal is reached. The memory robot is not always the most effective but if there is a goal within reach regardless of position it will find it.

Step by step test

The step by step test is used to represent how the robot is moving by visually printing out the robots position in the maze after every move it makes.

The right hand rule robot works as expected, it keeps following the wall with the "right-hand" on the wall and keeps going. If no walls are nearby at the robots start-location it moves north until it hits a wall then goes by the rule.
It cannot find goal if the goal itself is not attached to the wall that is connected to the start-point or the first wall it hits. If you look at maze4.txt the robot moves up until it reaches the rectangle, then keeps going round it endlessly. In maze5.txt it just goes into an infinite loop.

The memory robot keeps remembers which positions it has visited and follows the depth-first algorithm. It keeps going forward and always checks the surroundings, if it can move in only one direction it does so. If it reaches an intersection it remembers all the options and keeps going until it reaches a dead end. When a dead end is reached it backtracks the movements to the last visited intersection and tries the next optional path. If the robot doesn't find an exit it throws exceptions that the stack is empty as expected (maze 5 for example of a maze where the robot can't find an exit).

Junit tests

mazeJunitTest.java

The test shows how the maze class throws exceptions if the maze is implemented wrong, the class throws exceptions as expected when an incorrect file is added. If Goal or Start is missing it throws an illegal state exception that tells the user that the file is missing necessary values. If it holds any symbols other than asterisk, space, G or S an exception is thrown as well. Maze should be able to get the correct start-point from the file as-well and an exception is thrown if the maze is unable to find a start-point and the test throws an exception if the getStartPosition returns the incorrect value. The test showed that the maze was able to return the correct x and y values of the start-position.
It also tests if the isMovable and isGoal methods works as intended, also successful.
Lastly a visual test is made to check if the maze class is able to print out the current maze with an X to mark the position representing the robot.

IMPORTANT: IF the test does not work as intended, the mazeTest folder might be in the wrong directory, you will get a FileNotFound exception and an error-message saying “The system cannot find the path specified”. Just check so that the extracted mazeTest folder is in the directory specified in the error message.

positionJunitTest.java

Tests so the class works as expected, both the constructor and the methods, both to get the x & y coordinates and all the positional getters works as expected. The Position class is able to return the coordinates of the position to the North, East, West and South. The test also covers the “equals” method, it checks so that the method is able to distinguish if two positions are equal. Finally the hash code method is tested to see if the correct numerical hash code is return from a set position.