Umeå University
Department of Computing Science

Object-Oriented Programming Methodology 7.5 p
5DV133

OU2 Robots and Labyrinths

Submitted	2016-04-25
Author:	Lorenz Gerber (`dv15lgr@cs.umu.se` `lozger03@student.umu.se`)
Instructor:	Anders Broberg / Niklas Fries / Adam Dahlgren / Jonathan Westin / Erik Moström / Alexander Sutherland

Contents

1	Problem Description	1
2	Usage Instructions	1
3	System Description	1
3.1	UML class diagram	1
3.2	Class Responsibilty and Collaborations	1
3.3	Specific Implementation details	1
3.4	Move Algorithms of the Robots	1
4	Known Limits	2
5	Testing	2
5.1	JUnit tests	2
5.2	Run tests and Robot tests	2
	References	3

1 Problem Description

Aim of this laboration was to implement a number of class that allow to simulate robots in a labyrinth. The assignment suggested three base classes *Maze*, *Position* and *Robot* [1]. Further, it was defined that two specializations of the *Robot* class had to be implemented: One robot that always follows with his ‘right hand’ along the wall until he finds the goal of the maze. And another one that can remember positions where he has been and hence look for new unexplored positions until he finds the exit.

The classes were to be tested in a *main* program. More specific, the two specialized robot classes should be evaluated in a competition against each other. The maze shall be provided as text file in the command line arguments.

2 Usage Instructions

3 System Description

3.1 UML class diagram

3.2 Class Responsibilty and Collaborations

3.3 Specific Implementation details

3.4 Move Algorithms of the Robots

Right Hand Rule Robot

The right hand rule robot needed a number of help methods besides the specified *move()* method. This was mainly a consequence for giving the robot a memory for the direction of movement. This was seen as a measure to model the behaviour of the robot as close as possible how a physical implementation would work: If a robot has a *right hand*, it also has a direction and all operations should relate to the direction of the robot. The private methods *moveDirection()*, *wallDirection* and *movableDirection* are used to lookup geographic directions from robot directions. The actual translation is done in the *move()* method using addition and subtraction followed by a modulo operator on a direction value that is initialized by the *setInitialDirection()* method.

Try...catch statements are used to handle *ArrayIndexOutOfBounds* exceptions when the robot is at the edge of the maze. The edge is interpreted as a wall.

Memory Robot

The memory robot needs just the specified *move()* method. It uses a *stack* (*java.util.Stack*) for implementing the back tracking and a hash table (*java.util.HashMap*) for the memory of already visited places. For this work, the *hashCode()* method of the *Position* Class had to be implemented with an override. It was chosen to concatenate the X and Y position with an arbitrary unlikely-to-appear number (‘9999’ was chosen here).

The *move()* method of the memory robot can be roughly separated in two parts. In the first part, the robot checks in every four directions the possibility to move based on the condition *isMovable* from the *Position* class and whether the *hashCode* of the potential new position is already stored in the *positionHistory* hash map. In the current version, a fixed sequence of direction is implemented (North, East, South, West). As soon as a viable directin is found, the robot makes the move. This includes pushing the current position on the *backtrack* stack, storing the hash code of the current position to the *positionHistory* hash map and finally, changing location to the new position. After making the move, the method is left with a return statement without reaching the second part. Each of the direction checking code blocks is enclosed in a *try...catch* statement that handles the *ArrayIndexOutOfBounds* exception when the robot is at the edge of the maze. Hence positions ‘beyond’ the maze are interpreted as wall.

The second part of the *move()* method in the memory robot is concerned with the back-tracking. It consists just of reading the stack for the last position, and then popping the top of the stack. As mentioned above, this second part is just reached when no direction offered a viable move.

4 Known Limits

The robots are implemented such that they accept the border of the maze as a wall. Hence a valid maze without any wall can be created.

In the current version, the *Memory Robot* will not detect when there is no solution to the maze. This could be implemented by checking for the start position after back tracking.

5 Testing

5.1 JUnit tests

As requested, JUnit tests were implemented for the classes *Maze* and *Position*. The test were implemented post development of the main code.

Maze

The JUnit test for maze verify whether a correct Maze can be loaded. Then each a test to check for exceptions when a maze without start or goal position, or a maze with uneven row length is attempted to load.

Position

JUnit tests to check whether a *Position* object can be instantiated and to verify that the *equals()* method works were implemented. Further, it is checked whether different positions yield different hash codes.

5.2 Run tests and Robot tests

The robot testing was implemented in a small main program. A static variable *ROUNDS* determine how many rounds the competition shall last. This mode was decided as it would be more difficult to detect adhoc whether a maze can be ’solved’ in general or by a specific robot particularly. So, limiting to a fixed number of rounds was a mean to prevent infinite looping. A typical situation where an infinite loop could occur is when a robot starts within a compartment that has no connection to the compartment with the *Goal* position.

Move counters for each robot are used. In each round, it is first checked whether either of the robots has reached the *Goal*. Then the move counter is advanced by one and the the *move()* method of each robot is called. After the predefined number of rounds are over, it is evaluate whether the robots reached the goal and if so, which one reached it first.

For testing purposes, *toString()* methods for *Position* and *Maze* were implemented. They were used during development to verify the correctness of the algorithms.

The text files *maze1.txt* and *maze2.txt* work with both robots, while *maze3.txt* results in an exception for the right hand rule robot as the starting position is not at a wall. The other provided maze text files are used for the JUnit testing and produce exceptions.

References

[1] Umeå University, 5dv133 obligatorisk uppgift 2. `http://www8.cs.umu.se/kurser/5DV133/VT16/uppgifter/ou2/`, 2016. accessed: 2016-04-24.