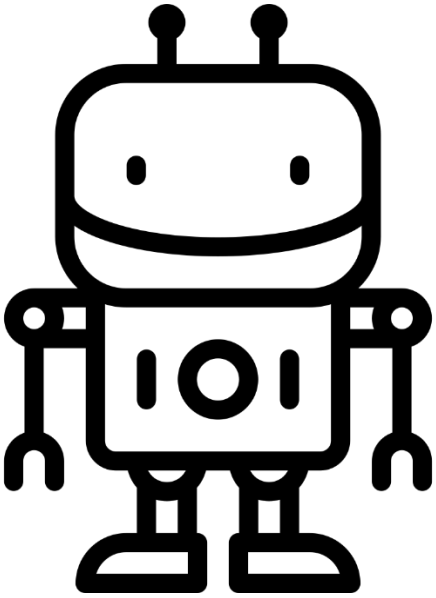
Laborationsrapport OU2

Objektorienterad programmeringsmetodik 5DV133

Jonas Sjödin
josj0105@cs.umu.se
2017-04-19



Kursansvarig
Anders Broberg

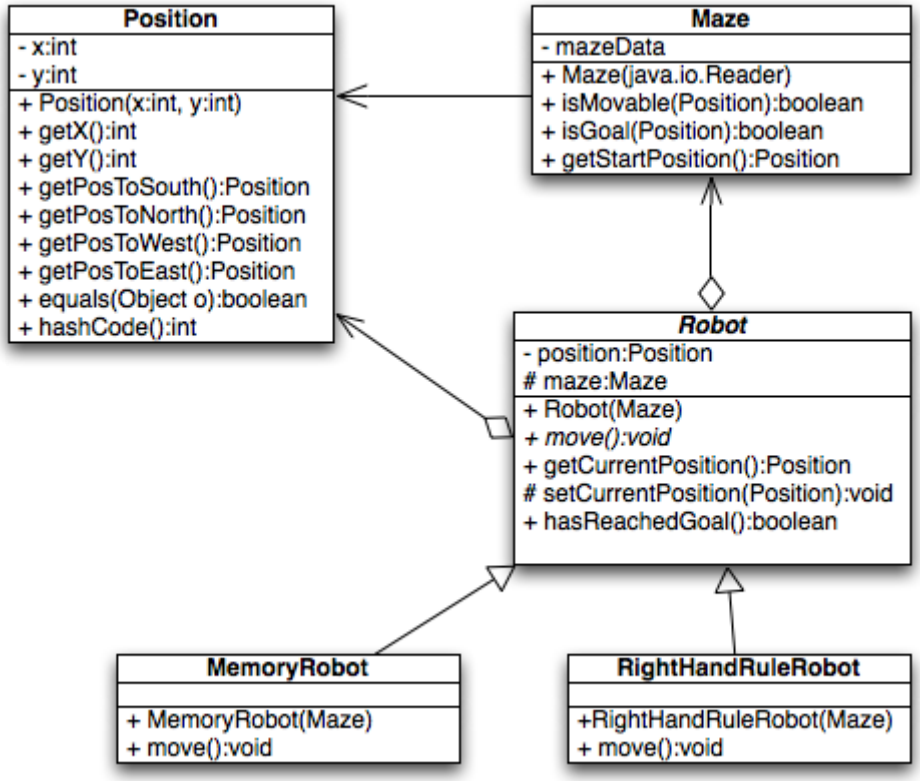
Handledare
Adam Dahlgren
Jakob Vesterlind
Sebastian Sandberg
Didrik Lindqvist
Daniel Harr

Innehåll

1. Inledning och problem

Denna obligatoriska uppgift handlar om att skriva klasser som representerar en labyrint, positioner och två robotar genom att designa de efter ett givet UML-diagram. Sedan ska robotarna tävla mot varandra och vilken som är snabbast i en given labyrint ska visas och deras rörelser och förflyttningar ska analyseras.

Det givna klass-diagrammet:



2. Användarhandledning

2.1 Körning av programmet i terminalen

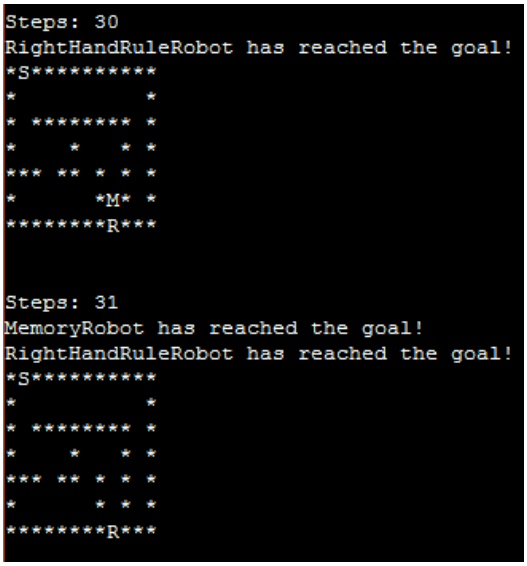
MR = MemoryRobot

RHRR = RightHandRuleRobot

För att kunna enkelt simulera en körning av programmet har jag skapat en main-fil som skapar en maze och en RHRR och en MR och låter dessa gå tills de båda har kommit i mål. För att köra denna fil från terminalen skriver man följande:

Först navigerar man till mappen i terminalen med kommandot ”cd java/ou2/src” fast med den väg där dina filer ligger. När man sedan navigerat fram till rätt mapp ska man skapa en mapp som tillexempel heter ”output” så slipper man ha de kompilerade filerna i samma mapp som sina .java filer. Sedan skriver man ”javac -d output Main.java Maze.java Position.java Robot.java MemoryRobot.java RightHandRuleRobot.java” för att kompilera koden. Sedan skriver man ”java -cp output Main maze.txt” för att köra de kompilerade filerna där ”maze.txt” är den angivna labyrinten i detta fall. Då kommer programmet att köras och skriva ut robotarna och dess position i terminalen tills de båda gått i mål.

När programmet körs kommer det att se ut som följande:



För att kompilera jUnit-testerna måste man först navigera till mappen där jUnit är lokaliserad. T.ex. så går man först till root mappen och skriver i terminalen ”javac -cp /usr/share/java/junit4.jar File.java FileTest.java” för att kompilera filen. Sedan skriver man ”java -cp .:/usr/share/java/junit4.jar org.junit.runner.JUnitCore FileTest”. Där de filer som ska köras i detta fall är Position & PositionTest och Maze & MazeTest. För att testerna ska lyckas krävs det också att de fyra maze-filerna maze.txt, maze2s.txt, mazeNoG.txt och mazeNoS.txt finns lokaliserade i samma mapp som de kompilerade JUnit-testerna.

3. Systembeskrivning

3.1 Position

Denna klass motsvarar en position med en x- och en y-koordinat. I denna klass finns publika metoder som hämtar positionens x- och y-koordinat. Klassen har även fyra metoder som returnerar positionen norr, en som returnerar söder, en som returnerar väst och en som returnerar öst om den nuvarande positionen. Sedan finns även hashcode och equals skapade i klassen.

3.2 Maze

Denna klass läser in en fil som innehåller en labyrint i **konstruktorn**. Sedan läser den in varje rad i textfilen och sparar varje enskilt tecken i en tvådimensionell arraylist så att varje cell i arraylisten motsvarar en arraylist med tecknena från motsvarande rad i labyrintfilen. Sedan stängs filen. Denna klass har även en funktion som kollar om en given position går att flytta till. Om den går det returnerar den true, annars false. Den har även en funktion som kollar om en given position är målpositionen G. Om den är det returneras true, annars false.

Denna klass har funktionen **getStartPosition** som returnerar startpositionen S där en robot ska starta. Den sista funktionen **printMazeRobot** i klassen printar ut labyrinten med roboten i till terminalen.

3.3 Robot

Denna klass är en abstrakt klass som har en maze och en position som sätts till labyrintens startposition i konstruktorn. Klassen har även en abstrakt **move** funktion som måste skapas i en annan klass som ärver från denna klass. Sedan har klassen en **getCurrentPosition** och en **setCurrentPosition** som hämtar och sätter robotens position till ett **Position**-värde. Det finns även en funktion här som kollar om roboten har nått målet vid namn **hasReachedGoal**.

3.4 MemoryRobot

Denna klass är en robot som använder sig av bredden först sökning (**BFS**) för att leta sig igenom en given labyrint. När en instans av klassen skapas körs **konstruktorn** som tar en given labyrint och sätter roboten i labyrinten. Den sätter även startplatsen som besökt. Klassen har även en **move** funktion som kör för att flytta på roboten. I denna klass så kollar roboten på sin omgivning och om rutorna bredvid inte är besökta och om de går att gå på så läggs de till i robotens stack med platser som ska besökas. Sedan byter metoden robotens nuvarande position till nästa obesökta i stacken och sätter den nuvarande positionen till besökt.

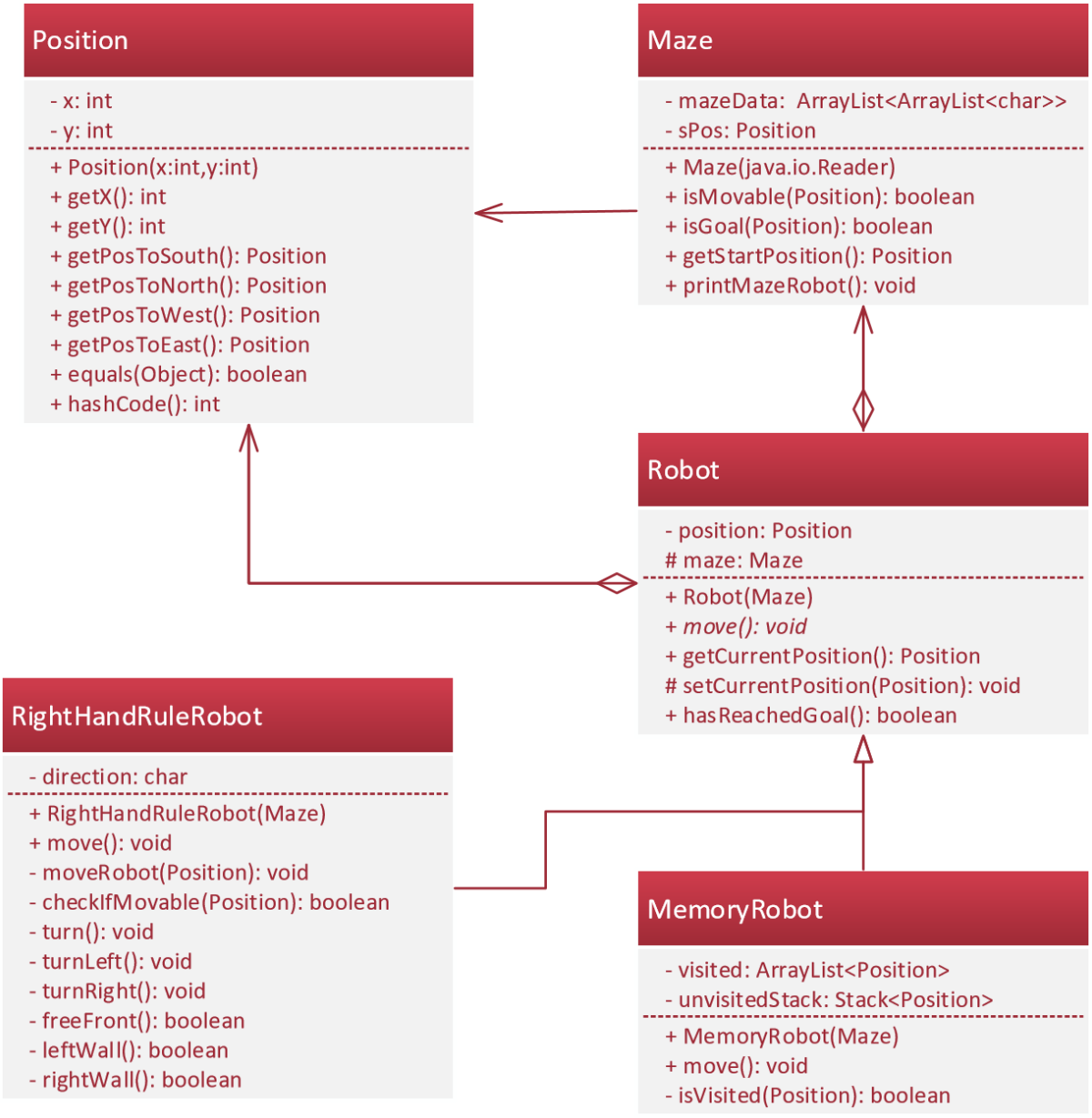
3.5 RightHandRuleRobot

Denna klass är en robot som hela tiden följer en vägg på sin högra sida. När en instans av klassen skapas körs **konstruktorn** som sätter den klassglobala char-variabeln **direction** till rätt riktning för att roboten ska vara i rätt riktning gå när funktionen **move** körs. När **move** sedan körs fyra kollar på omgivningen.

1. Om det finns en vägg på höger sida om roboten och platsen framför roboten i dess riktning så ska roboten flyttas ett steg fram.
2. Om det inte finns en vägg på höger sida om roboten ska roboten vända till höger och flyttas ett steg i den riktningen.
3. Om det finns en vägg till vänster och höger och roboten inte kan gå fram p.g.a. en vägg eller om positionen är utanför labyrinten så ska den vända på sig 180 grader.
4. Om roboten har en vägg på höger sidan och platsen framför inte går att gå till och platsen till vänster om roboten är ledig ska roboten vända sig till vänster och flyttas ett steg i den riktningen.

För att kunna utföra detta körs ett antal funktioner där **moveRobot** flyttar roboten ett steg i samma riktning. Sedan finns flera andra privata funktioner som berättar för roboten om dess omgivning och sköter riktningsändringar av roboten till höger, vänster eller bakochfram.

4. Klassdiagram



För att enkelt och tydligt visa hur detta program är uppbyggt presenteras här ett klassdiagram över mina 5 klasser som bygger upp grunden i mitt program.

5. Begränsningar

5.1 Labyrint

Denna lösning kräver en labyrint som ser ut på följande sett:

- `* = Vägg`
- `S = Start (Antal = 1)`
- `G = Mål (Antal >= 1)`

En labyrint kan då t.ex. se ut på följande sett:

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

5.2 Robotarna

MR klarar av att hantera alla labyrinter där det går att gå från start till mål. RHRR däremot kan endast hitta målet om den startar vid en vägg som är sammanbunden med målet. Utöver detta ska robotarna klara av alla labyrinter som följer ovanstående specifikation för en labyrint.

6. Tester

6.1 Klasstester

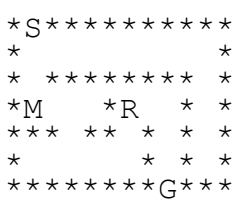
6.1.1 PositionTest

Denna JUnit-klass testar gränsytan i klassen Position så att den fungerar som den ska. Där testas det om det går att starta en ny position, om getX och getY returnerar det som förväntas. Klassen testar även om positionerna runtomkring en given position returnerar förväntat resultat. Den testar även om equals och hashCode fungerar som förväntat

6.1.2 PositionTest

Denna JUnit-klass testar gränsytan i klassen Maze. Den testar att skapa en Maze som har en start och ett mål. Den testar även om ett undantag kastas om en maze försöker skapas med 2 starter, ingen start eller inga mål. Klassen testar även maze metod isMovable där det kollas så att endast en position utanför mazen eller en position som har ett '*'-tecken inte är movable. Den testar även så att isGoal och getStartPos fungerar som de ska.

6.2 Testkörning av robotarna

Efter att robotarna tagit 8 steg har de flyttat sig såhär: 	Efter att robotarna tagit 16 steg har de flyttat sig såhär. 
Efter att robotarna tagit 21 steg har de flyttat sig såhär och RHRR har gått i mål. 	Efter att robotarna tagit 31 steg har de flyttat sig såhär och nu har även MR gått i mål. 

6.3 Analys av testkörning

Då min implementation av MR först kollar om den kan gå norr, sedan syd, sedan väst och sist öst kommer den alltid att gå öst om den kan, om inte kommer den gå väst, om inte kommer den gå syd och om inte det går gå norr och därför behöver min implementation av MR ta fler

steg för att hitta till mål i denna specifika labyrint än RHRR. Fördelen med MR är dock att om målet inte hänger ihop med en vägg så hittar denna robot tillslut mål medans RHRR inte kommer att göra det utan endast gå runt i cirklar. En annan fördel med MR är att den inte kan besöka en plats två gånger utan går bara då direkt till nästa obesökta plats i stacken. Den enda fördelen som RHRR har över MR är ifall labyrintens rätta väg är i stora drag att följa robotens högra vägg, som ovanstående är. Då den absolut snabbaste vägen i labyrinten är att ta 17 steg (Om en robot endast får flyttas ett steg i taget) så är RHRR riktigt nära med sina 21 steg.