UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport obligatorisk uppgift

Objektorienterad programmeringsmetodik 7.5 p

5DV133

Obligatorisk uppgift nr

2

Namn	Simon Asp	
E-post	id14sap	@cs.umu.se
Datum	15.04.28	
Handledare	Adam Dahlgren, Andrew Wallace, Filip Allberg, Niklas Fries, och Lina Ögren	

Innehållsförteckning

1. Problembeskrivning
2. Användarhandledning
3. Systembeskrivning
4. Lösningens begränsningar
5. Testkörningar
    - 5.1 Ytterligare test
    - 5.2 Resultat
    - 5.3 Diskussion:

1. Problembeskrivning

Uppgiften går ut på att låta två robotar tävla i en labyrint som läses in från en fil. Robotorna har olika algoritmer för att lösa labyrinten, och labyrinten kan vara godtyckligt stor. För att båda robotorna garanterat ska fungera så måste startpositionen och målpositionen vara ihopkopplade längs en vägg. Labyrinten ska vara konstruerad så att väggar markeras med "*", och start- och målposition markeras med "S" respektive "G". Roboten kan sedan röra sig på positioner markerade med mellanslag " " (se bild 1.)

Den ena roboten måste använda sig av en "right hand rule"-algoritm. Detta betyder att den måste hålla sin högra hand längs en vägg ända tills den kommer fram till målet. Den andra måste använda sig av en "memory robot"-algoritm, som fungerar så att den kommer ihåg var den har varit, och också markerar var den varit. Om den sedan fastnar så stegar den tillbaka för att hitta ett nytt ställe att gå på, men den får inte gå där den har markerat.

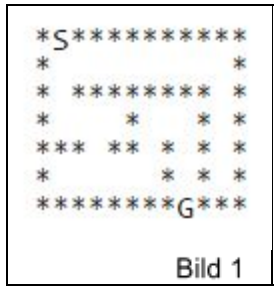

Bild 1

Labyrinten ska fungera i ett koordinatsystem med x- och y-koordinater som roboten kan röra sig på och veta var den befinner sig. Målet är i slutändan att mäta robotarnas prestanda och se vilken som är snabbast i en given karta.

2. Användarhandledning

Lösningen ligger i mappen `~/id14sap/edu/5DV133/ou2/` där det finns ett antal java-filer och även kompilerade .class-filer. Själva lösningen ligger i filen MazeRun.java där prestandan mäts för robotorna.

För att köra MazeRun så skriver man
`# java MazeRun maze.txt` (för att köra robotarna i kartan i bild 1.)
Vill man köra ett större labyrint skriver man
`# java MazeRun hugemaze.txt` där labyrinten är 2000x2000 tecken stor.

3. Systembeskrivning

Systemet består i grova drag av en abstrakt klass **Robot** (som har specifika underklasser) som håller reda på vart den befinner sig, vart den kan gå och om den kommit i mål i klassen **Maze** med hjälp av klassen **Position.**

Klassen **Maze** skapar en labyrint utifrån en fil som användaren anger (Som bild 1) och sparar den i ett attribut `mazeData`. Labyrinten är skapad med hjälp av en tvådimensionell array av typen Char som lagrar tecknen från filen till platser i arrayen. Här kan roboten sedan förflytta sig genom arrayens index, med hjälp av klassen Position.

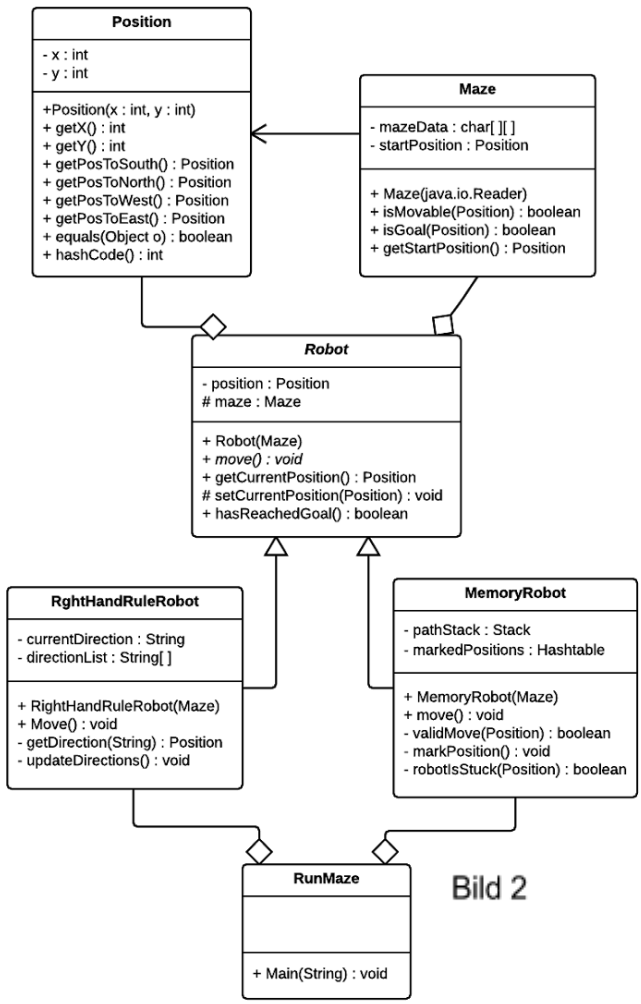

Bild 2

Position består av ett x- och ett y-värde som kan användas för att positionera en viss plats i arrayen. Eftersom arrayen är indexerad från noll för både rader och kolumner från samma ställe (se bild 3), så kommer en förflyttning söderut innebära att y-värdet måste öka, och vice versa för att förflytta sig norrut. Koordinaterna är alltså inte som ett kartesiskt system som man kan vara van vid, utan ett tabellsystem där 2,1 motsvarar kolumn 2 rad 1.

	Col 1	Col 2	Col 3	Col 4
Row 1	0, 0	0, 1	0, 2	0, 3
Row 2	1, 0	1, 1	1, 2	1, 3
Row 3	2, 0	2, 1	2, 2	2, 3
Row 4	3, 0	3, 1	3, 2	3, 3

Bild 3

Maze har en metod för att se var roboten kan röra sig - `isMovable()` (alltså på mellanslag-tecken i labyrinten), som också håller koll på att roboten är inom rätt intervall, alltså innanför arrayens index. Skulle roboten försöka gå utanför detta intervall eller att en position är en asterisk, så returnerar metoden false, annars true. En Maze kan inte skapas om inte start- och målpositionen finns med i filen. Om inte detta finns med kommer ett undatag kastas.

Robot har alltså en position som den befinner sig på, och en move-metod som flyttar roboten ett steg i labyrinten. Hur roboten ska röra sig utges ifrån de två underklasserna till robot. RightHandRuleRobot och MemoryRobot.

RightHandRuleRobot går efter en algoritm som jag hittade på ett forum[1], som lyder: Om ett av dessa steg är möjliga, gå dit: höger, fram, vänster, bak.

[1] http://www.dreamincode.net/forums/topic/328670-cases-for-right-hand-rule-maze-solver/ (15-04-28) svar nr 5.

Roboten kollar alltid dessa steg, men den gör det utifrån var den själv pekar. Så om den pekar åt söder så kommer "höger" inte vara detsamma som att gå höger om roboten pekar åt norr. För att hålla koll på var roboten pekar finns ett attribut `currentDirection`, och för att veta vad "höger" eller "vänster" är gentemot vart roboten pekar så används en array av typen String, som uppdateras beroende på var roboten pekar.

MemoryRobot använder sig av en stack och en hashtabell för att hålla koll på var den har varit och vart den ska gå om den fastnar. Anta att roboten står på startpositionen. Då kommer den först markera den positionen i hashtabellen och lägga till positionen den står på i stacken, sedan leta efter ett ställe att gå till och går dit. Sedan repeterar den detta tills dess att den till exempel har kört fast i ett hörn, då backar den bakåt med hjälp av stacken tills dess att den kan röra sig igen, och "poppar" de värden i stacken som den lämnat (men bara om den verkligen inte kan gå någon annanstans än bakåt).

För att köra de två robotarna används **MazeRun**-klassen som läser in en fil, skapar en maze och sedan låter de två robotarna köra tills de nått målet, och samtidigt räknas tiden det tog för de att komma fram, och hur många steg det krävdes.

4. Lösningens begränsningar

För att kunna köra robotarna i labyrinten krävs en korrekt fil, med start och målposition, detta kollas och varnas för om det skulle hända att en fil inte har dessa positioner. Vad lösningen inte kollar är dock om start- och mål-positioner går att nå. Det skulle till exempel kunna vara så att start och/eller målpositionen har väggar runt sig och inte går att nå. Då kommer robotarna gå runt i en oändlig loop. Det antas alltså att filen har en lösning och att målet går att nå.

RightHandRuleRobot kräver att målet hänger ihop med starten, annars kan den också fastna i en oändlig loop där den snurrar runt runt. Detta sker när målet exempelvis svävar fritt och inte har nån vägg bredvid sig.

Min implementation av RightHandRuleRobot är kanske inte den bästa och snyggaste eftersom en lista hela tiden måste uppdateras när roboten ska flytta på sig. Det finns bättre metoder som jag hört talas om som använder modulus, men eftersom det här var lättast för mig att förstå när jag skulle skapa en algoritm så fick det bli så.

Att skapa mazen kan ta ganska lång tid eftersom den läser in rad för rad till en sträng och sedan bryter upp den strängen och sätter in i en matris. Det är lite onödiga steg, och jag kom senare på att man kan lagra strängraderna var för sig i exempelvis en lista. Dessutom blir det lite krångligt att hålla reda på koordinaterna eftersom en matris i Java

lägger in rader först och sedan kolumner, alltså måste man ange y-koordinaten först och sedan x, vilket man kanske inte är van vid när man tänker x- och y-koordinater.

5. Testkörningar

Positionsklassen var den första som utvecklades och det var naturligt att börja testa den på en gång. Testen är gjorda med JUnit4 och nedan är en beskrivning av de test som gjordes.

`testPositionNotNull()`

- En position med koordinaterna 0,0 skapas för varje test och här testas att positionen verkligen har skapats och inte är tom med `assertNotNull()`

`testGetX()`

- Här testas att man får rätt X värde när man kallar på `getX()`. X-värdet borde vara 0 eftersom det var det som lades in när positionen skapades. Resultatet kollas med `assertEquals()`

`testGetY()`

- Här är samma test som ovan med med `getY()`. Y-värdet borde också vara 0 eftersom det var det som lades in när positionen skapades.

`testGetPosToSouth()`

- Här kollas att positionen har gått söderut, alltså att Y-värdet har *ökat* (som rör sig på matrisens rader). Förväntar sig att få +1 på Y-värdet. Positionen hämtas med `getY()` och jämförs med +1 genom `assertEquals()`.‘

`testGetPosToNorth()`

- Detta är samma test som ovan men här förväntas Y-värdet blivit -1.

`testGetPosToWest()`

- Samma test som ovan men här förväntas X-värdet att minska, vi söker -1.

`testGetPosToEast()`

- Samma test igen men X-värdet borde nu vara +1.

`Equals()` och `Hashcode()` har inte testats eftersom de autogenererats med editorn IntelliJ IDEA 14.

När dessa test fungerade och garanterade funktionaliteten för Position så skapades test för Maze och utvecklingen började ske av den klassen. Dessa är testen för Maze:

`testMazeNotNull()`

- Här kollas att mazen som är skapad inte är tom, med `assertNotNull()`.

`testMazeException()`

- Här gjordes ett test för att kolla så att mazen kastar ett undantag om labyrintfilen saknar en startposition. En "dålig maze" (utifrån `badmaze.txt`) skapades och sedan så förväntade sig testet att få ett Exception eftersom klassen Maze ska kasta ett sådant om ingen start och/eller slutposition finns.

`testIsMovable()`

- Här testas vad som händer när en position som är en asterisk i mazen anropas genom `isMovable()`. Vad som händer då är att metoden ska returnera false, och detta försäkras genom `assertFalse()`.

`testOutOfBounds()`

- Här kollar testet vad som händer när en position befinner sig utanför mazens intervall. En position skapas på (-1, -1) och skickas in i `isMovable()` –metoden. Den metoden borde då returnera false, och detta kollas med `assertFalse()`.

`testGetStartPosition()`

- Startpositionen i `maze.txt` ligger på koordinaterna (1, 0) och dessa jämförs med var som kommer tillbaka från `getStartPosition()` med `assertEquals()`.

`testIsGoal()`

- Målpositionen i `maze.txt` ligger på koordinaterna (8, 6) och dessa stoppas in i `isGoal()` och borde då returnera true. Detta kollas med `assertTrue()`.

5.1 Ytterligare test

När mazen blivit skapad gjordes test att skriva ut det som fanns i mazeData för att försäkra sig om att det som låg där var lika det som matades in. Detta gjordes inte i ett JUnit-test utan endast i konsolen för editorn. Liknande test gjordes för robotarna för att se att de rörde sig på rätt sätt. Innan detta gjordes så skrevs även koordinaterna för varje robotförflyttelse så att man enkelt kunde se om roboten rörde sig till rätt ställen. Dessa test är inte med i den färdiga koden, vilket de skulle kunna vara, men eftersom det inte krävdes att dokumentera dessa test med till exempel JUnit så togs dessa test bort då det sparade tid.

5.2 Resultat

Jag körde tre labyrinter för att kolla prestandan på robotarna. Den ena är den labyrint som var med i specifikationen, och den andra är 100x100 tecken stor och den tredje 2000x2000 tecken stor. Denna kräver lite mer av robotarna och det var mer intressant att se vad som hände när labyrinten var så stor. Båda robotarna klarar av labyrinterna.

Här är resultaten i tid och antal steg för robotarna:

	Labyrint 1:	**Labyrint 2:**	**Labyrint 3:**
Memory robot tid:	0.0003 sec	0.0156 sec	1.3734 sec
Right hand rule tid:	0.0001 sec	0.0051 sec	0.1049 sec
Memory robot steg:	22	4968	1879435
Right hand robot steg:	21	5022	1948778

Som vi ser så tar memory robot alltid mer tid, vilket är förståerligt eftersom implementationen är mer komplex. Däremot så tar den färre steg på de två stora labyrinterna, vilket är lite intressant. Jag ville gärna tro att right-hand-rule alltid skulle vara snabbare, men det verkade inte vara så för just dessa två labyrinter. Detta beror såklart på hur labyrinten ser ut.

5.3 Diskussion:

Jag tänkte först göra tester som loopade igenom filen för att hitta start och målpositioner och använda i testGetStartPosition() och testIsGoal(). Men detta skulle ju egentligen testa själva filen, att den är rätt, och inte att dessa två metoder fungerar som de ska. Att ange koordinater som man vet är en start och målposition tycker jag är rimligt eftersom det testar om metoderna fungerar som de ska.

De enda test som gjordes för att försäka sig om att mazen är skapad är att kolla om den inte är tom och att den är komplett, alltså har start och målposition. Inget test gjordes för att kolla om mazen har en lösning eller om t.ex. alla positioner bara är stjärnor. Detta kunde gjorts genom att i alla fall försäkra sig om att det finns tillräckligt många mellanslag (giltiga positioner att gå på) för att ta den kortaste vägen i en given maze. Men eftersom det inte stod i beskrivningen av uppgiften så lämnade jag det och antog att den maze-fil man stoppar in har en lösning.