Obligatorisk Uppgift 2: Roboten 2.0

Objektorienterad programmeringsmetodik

5DV113VT20

Kursansvarig: Johan Eliasson(anders.broberg@cs.umu.se)

Handledare/övriga lärare: Jakub Jagiello, kuba@cs.umu.se , Klas af Geijerstam(klasa@cs.umu.se), Fredrik Peteri, fredrikp@cs.umu.se
Lennart Steinvall, dva95lsl@cs.umu.se
Anders Broberg, bopspe@cs.umu.se

namn: William Danielsson
E-post: William99danielsson@hotmail.com
cs-user: id18wdn

datum: 2020-04-27

Innehållsförteckning

1.0 Inledning/problembeskrivning sida: 3

2.0 Körning sida: 3

3.0 Systembeskrivning sida: 4-9

3.1 Position sida: 5

3.2 Maze sida: 6

3.3 Robot sida: 6

3.4 MemoryRobot sida: 7

3.4.1 Public move(), djupet-först sökning sida: 7

3.5 RandomRobot Sida: 8

3.5.1 Public move(), Slumpmässig-traversering sida: 8

3.6 RightHandRuleRobot sida: 9

3.6.1 Public move(), RightHandRule-algoritm sida: 9

4.0 Testning sida: 10

4.1 JUnit5 Tester sida: 10-11

4.2 MazeSimulation Testkörningar sida: 11

5.0 Reflektion och Sammanfattning sida: 12

1.0 Inledning/problembeskrivning

Målet med denna uppgift var att skapa 3 robotar som ska kunna läsa in en labyrint via en textfil och stega sig igenom labyrinten från starten till målet med hjälp av en djupet-först traversering, RightHandRule-algoritm och via endast slumpmässig traversering. Labyrinten ska vara en textfil bestående av nått tecken t.ex “*” eller “#” som väggar och “ “ (mellanslag) som förflyttningsbara vägar. Roboten ska alltid börja på startpositionen markerad med ett “S” i labyrinten och den ska röra sig på ett giltigt sätt (alltså inte gå igenom väggar eller teleportera) till slutet markerad med ett “G”. En exempel labyrint kan till exempel se ut som följande.

```
###############S####    S = startpositionen
#   #                #
# ###### ##  ## #  #
#   #    #  #####  #
# #   ## #  #   #  #
# #####  #### # ####
#   #         #    #    G = målet
##################G#
```

Programmet ska vara uppbyggt med hjälp av objektorientering i JAVA och använda de funktioner som är definierad i gränsytan till specifikationen.

2.0 Körning

Programmet kan kompileras och köras i de flesta utvecklingsmiljöer men går såklart också att köra direkt från terminalen. På windows kan koden kompileras genom att ställa in rätt path där java/bin mappen ligger.
path = “insert_path_here”.

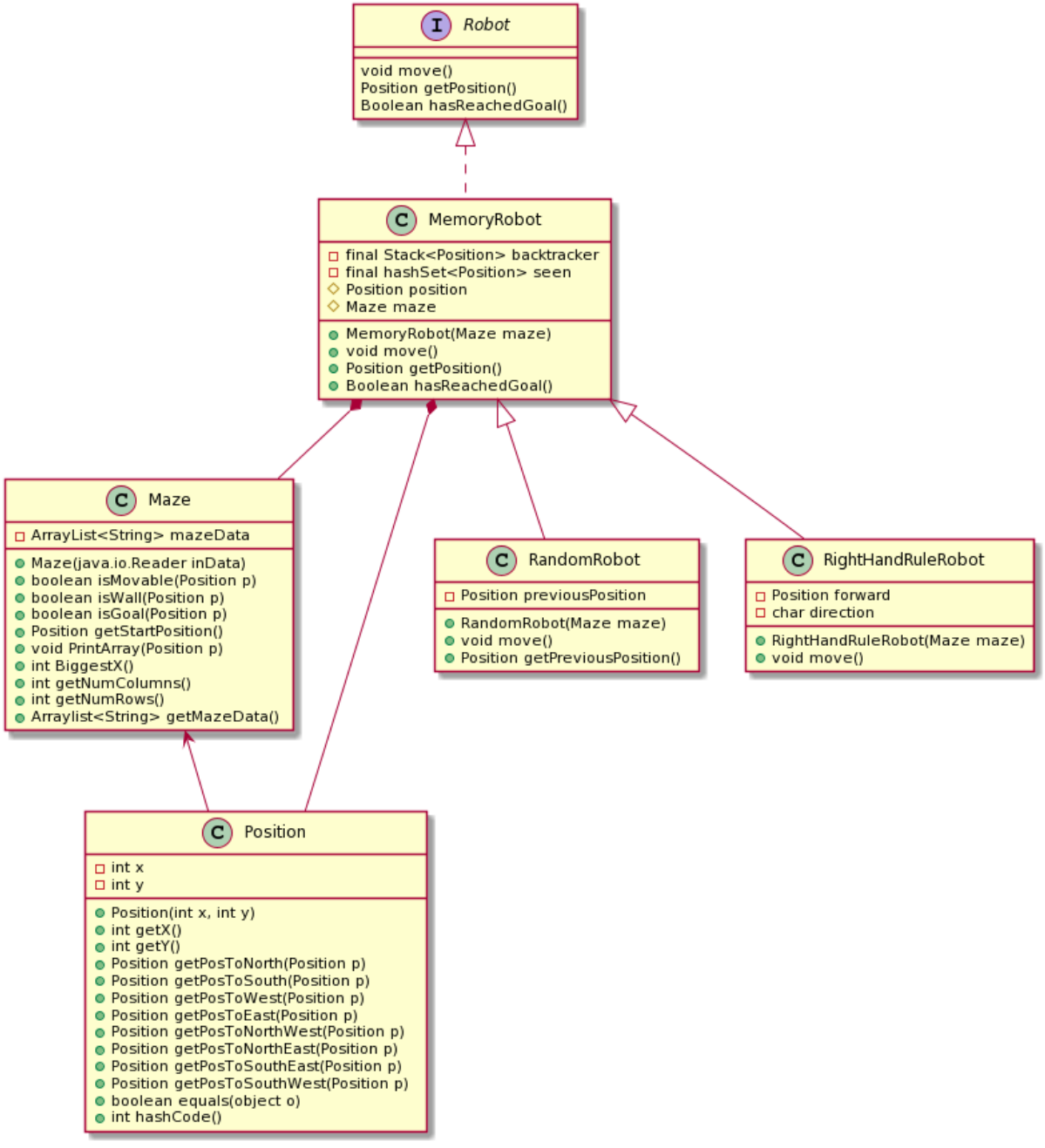
Sen för att kompilera från mappen där filerna ligger skriver man
cd “insert_path_here”

Sedan kompileras koden genom att skriva
javac MazeSimulation.java Position.java Maze.java MemoryRobot.java RightHandRuleRobot.java RandomRobot.java Robot.java

Nu är programmet kompilerat, för att sedan köra det skrivs
java MazeSimulation
Nu är programmet igång och för att läsa in en labyrint klickar man på “Open” knappen högst upp i det vänstra hörnet och sen kan användaren bläddra bland filer och välja en labyrint-fil i txt format.

3.0 Systembeskrivning

UML-Klassdiagram över programmet och beskrivning av klassernas ansvarsområden samt viktigaste metoder beskrivs nedan.



3.1 Position

Klassen Position är lägst i klass-hierarkin och används för att implementera datatypen Position som består av ett x och y värde. Positions viktigaste metoder är som följande.

public getPosToNorth(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg högre i Y-led än den som togs in.

public getPosToSouth(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg lägre i y-led än den som togs in.

public getPosToEast(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg högre i x-led än den som togs in.

public getPosToWest(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg lägre i x-led än den som togs in.

public getPosToNorthWest(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg lägre i x-led och ett steg högre i Y-led än den som togs in.

public getPosToNorthEast(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg högre i x-led och y-led än den som togs in

public getPosToSouthWest(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg högre i x-led och ett steg lägre i y-led än den som togs in.

public getPosToSouthEast(Position p) Tar in en Position och returnerar en uppdaterad koordinat som nu är ett steg högre i x-led och y-led än den som togs in.

3.2 Maze

Klassen Maze är ett steg högre i klass-hierarkin, maze skapar en arraylist mazeData och läser in labyrint-filen med hjälp av en Reader. mazeData kan sedan användas för att söka efter specifika positioner i labyrinten och avgöra vilka positioner som är giltiga för roboten att gå till. Maze viktigaste metoder är som följande.

public isMovable(Position p) Tar in en position och avgöra ifall den positionen är på en plats som är giltig, alltså inte utanför arrayen och är antingen ett “ “ (mellanslag) eller “G” (målet). Returnerar true respektive false beroende på positionen.

public isWall(Position p) Tar in en position och avgöra ifall den positionen är på en vägg, Returnerar true respektive false beroende på positionen.

public isGoal(Position p) Tar in en position och kollar ifall den positionen har tecknet “G”, returnerar antingen true respektive false beroende på positionen. Denna metod är trivial eftersom den avgör ifall roboten har utfört sitt jobb eller inte.

public getStartPosition() Loopar igenom hela arraylistan och letar efter den positionen där “S” ligger på. Returnerar sedan positionen för “S”, denna metod används i början av programmet för att sätta roboten på rätt plats från början innan den börjar röra på sig.

3.3 Robot

Robot är ett interface som definierar strukturen för hur en robot ska vara uppbyggd. Det är sen en robots-klass jobb att implementera de 3 olika metoderna nedan.

void move()

Position getPosition()

Boolean hasReachedGoal()

3.4 MemoryRobot

MemoryRobot är den klassen som är först med att implementerar från interfacet Robot. MemoryRobot har som uppgift att göra en djupet först sökning i labyrinten och använder därför attribut som, Position, Maze, hashSet och en Stack.

3.4.1 public move(), djupet-först sökning

Detta en är den metoden som gör mest arbete i hela programmet, alla andra metoderna har endast varit byggstenar för att denna move metod ska kunna fungera.

Teorin går ut på att roboten börjar på startpositionen och sen väljer den en riktning (North,West,East eller South) och kollar ifall den platsen är movable genom att helt enkelt kalla på isMovable metoden i maze och ta in t.ex getPosToNorth som argument. Ifall det är en giltig position alltså är ett mellanslag eller “G” förflyttar sig roboten dit. Men sen sparar även roboten sin position i en hashset som kommer ihåg just den positionen och förutom det lägger den även positionen överst i en stack. Detta måste göras för att djupet-först sökningen ska fungera, roboten måste minnas de positioner den har varit på så att den inte hoppar mellan två eller flera positioner för alltid utan att komma vidare, sen måste roboten även kunna backa ifall den går in i en återvändsgränd. Hashseten tar hand om minnet och stacken ser till att roboten kan backa till en tidigare position.

I min kod är detta implementerat genom att varje gång roboten vandrar ett steg sparar den positionen både i en hashSet och i en stack. När roboten går kollar den ifall positionen den försöker gå till ligger i hashseten redan eller inte, ifall positionen redan ligger där betyder det att den redan har besökt den platsen förut och därför inte ska gå dit igen. Ifall roboten misslyckas att gå åt alla 4 hållen kollar den i stacken för den tidigare positionen och backar dit istället. Detta kommer göra att ifall roboten inte längre hittar en väg backar den tills att den hittar en väg den inte tidigare besökt och fortsätter gå igenom den vägen istället. Detta återupprepas till målet är hittat eller tills varenda plats har blivit besökt. Ifall varje plats har blivit besökt gör programmet en System.exit(-1) och säger till användaren att det inte fanns någon väg till målet.

3.5 RandomRobot

RandomRobot är en simplare version av MemoryRobot som har samma metoder förutom att move() funktionen är annorlunda. Eftersom de båda använder ungefär samma attribut och att 2 av 3 metoder är samma så ärver RandomRobot från MemoryRobot och endast ändrar på Move() metoden och har några andra attribut.

3.5.1 public move(), Slumpmässig-sökning

Precis som i MemoryRobot är detta den metod som gör den största delen av jobbet. RandomRobot börjar med att kolla ifall den kan gå åt ett slumpmässigt håll. Ifall den kan det går den dit och sparar sin föregående position, ifall det inte gick att gå dit provar den ett annat håll som inte var det den precis provade och kollar ifall den kan gå dit tills den lyckas att gå åt något håll.

När den har lyckats gå ett steg gör den samma sak igen bara att den inte får gå till samma position som den precis gick ifrån. Detta är för att den inte ska gå fram och tillbaka onödigt mycket. När den sen har gått ett till steg så skriver den över den första föregående positionen med den nya föregående positionen. Denna Robot har alltså inte minne och vet endast om den positionen den precis var vid och sin nuvarande position.

3.6 RightHandRuleRobot

Denna robot precis som RandomRobot ärver också attribut och metoder från MemoryRobot och ändrar move() funktionen till en annan algoritm. Den algoritmen som denna robot skulle ha vara RightHandRule som menas att man ska tänka att roboten håller sin högra hand mot väggen till höger och håller kvar den där hela tiden och bara fortsätter gå längst med väggen tills att den har gått igenom hela labyrinten.

Denna algoritm såg simpel ut på papper men var helt klart den mest komplicerade av dem alla. Det är så mycket som händer när roboten vänder på sig, eftersom vart "höger" är är relativt beroende på vart roboten tittar vid ett visst tillfälle.

3.6.1 public move(), RightHandRule-algoritm

Detta är den metod som gör mest jobb i denna robot precis som i dem andra robottyperna. Jag implementerade detta genom att tänka att roboten alltid kollar åt något håll t.ex South. Då betyder det att roboten har sin högra hand åt Väster och att ifall den går framåt så går den söderut. Roboten kollar ifall det finns en vägg på sin västra sida och ifall den kan gå framåt och isåfall går den ett steg framåt. Den gör detta tills att det antingen inte finns en vägg på den västra sidan eller om den inte kan gå framåt längre. När detta händer kollar den ifall väggen tog slut och om den måste gå runt väggen eller ifall den kom till ett hörn/återvändsgränd och byter direction beroende på vilket av dem det var. Vi säger att den byter direction till East. Detta betyder nu att framåt är åt öster och att den då har sin högra hand mot väggen söderut. Och samma algoritm upprepas och samma undantag kollar fast nu med andra väderstreck eftersom robotens rotation har ändrats med 90 grader. För att göra detta möjligt blev jag tvungen att implementera 4 nya metoder i Position som även kollar diagonala rutor. Detta är så att roboten kan se när en vägg tar slut.

Denna Robot har inte heller något minne men den är betydligt smartare än RandomRobot eftersom med denna algoritm kommer roboten ha besökt alla platser i labyrinten ganska effektivt. Efter ett visst tag kommer den ha gått ett helt varv runt och ifall den inte hittade målet betyder det att det antingen inte finns något mål eller att målet ligger mitt i labyrinten långt ifrån en vägg mitt i det öppna.

Jag spånade lite på att göra så att den avslutar när den har gått runt hela labyrinten och inte hittat målet så att den inte går runt i all evighet. Men det tog för mycket tid och jag kände att jag inte skulle hinna resten av uppgiften om jag skulle behöva implementera en sådan.

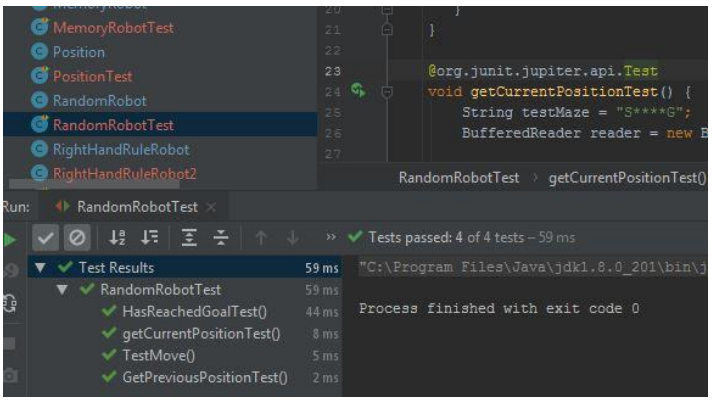
4.0 Testning

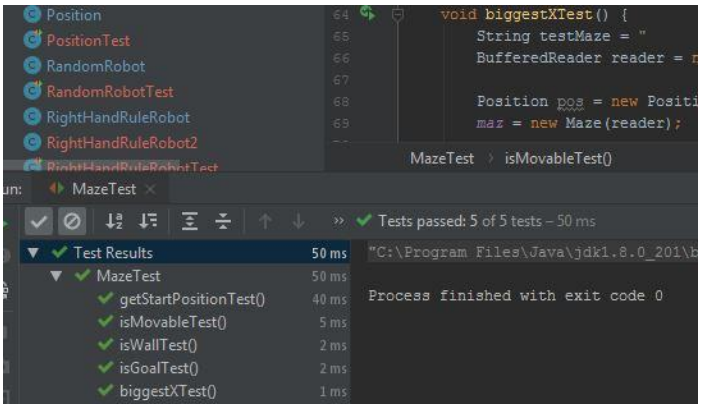
När det kom till testning använde jag mig av både JUnit5 tester för varje klass och mitt egna testprogram där jag har en metod i Maze som skriver ut labyrinten och robotens position där jag kan se den stega igenom labyrinten. Jag hade missat att handledarna hade kommit ut med ett eget testprogram som mer eller mindre var mitt testprogram fast med ett mycket snyggare grafiskt gränssnitt. Tyvärr fick jag inte så mycket tid att använda detta när jag programmerade eftersom jag inte visste om den och använde mitt egna istället.

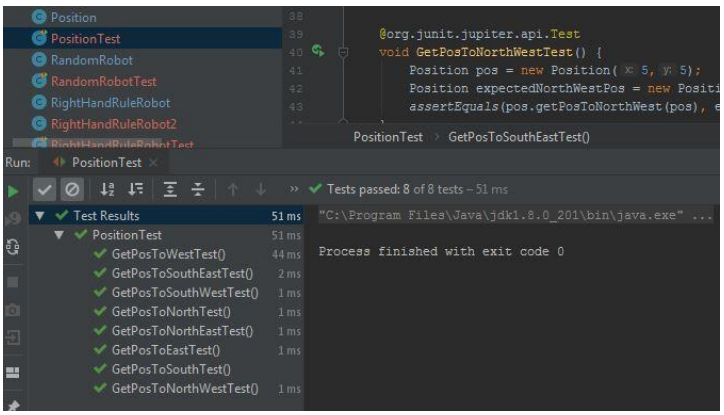
4.1 JUnit5 Tester

JUnit tester skapades för alla klasser och innehåller ett test per metod som testar att inget konstigt händer och att rätt värden blir tilldelade till rätt attributs osv.

I robotklassernas move() metod kollar jag bara att robotarna inte kan gå in i en vägg. Detta är för att det kändes väldigt komplicerat att skriva ett test som kollar ifall algoritmerna funkar, detta var lättare att kolla visuellt genom att köra mitt testprogram istället. Därför skrev jag bara ett litet test för move() så att jag kan se att den inte gör något den inte får göra när jag har gjort en ändring i klassen.







4.2 MazeSimulation testkörningar

MazeSimulation var en klass som vi fick färdig från handledarna som vi kan använda för att testa labyrinterna grafiskt. Jag har gjort några egna labyrinter som jag har skapat via en webbsida som genererar labyrinter och några labyrinter är tagna från förra årets kurs då vi fick labyrinter givna av handledarna.

Klicka på länken nedan för att se ett DEMO på 4 labyrinter som körs i MazeSimulation och där varje labyrint testas med mina 3 olika robottyper.

https://www.youtube.com/watch?v=YDZT_hDz7HA&feature=youtu.be - **Video Länk till DEMO**

5.0 Reflektion och Sammanfattning

Jag tycker att dessa två uppgifter ou1 och ou2 på denna kurs har varit de roligaste programmerings uppgifterna hittills. Väldigt svåra, men lärorika och lönsamma. Det var också ett roligt tillägg att få ett färdigt grafiskt testprogram som gjorde att man kunde se sitt verk i ett mer komplett sammanhang.

Det som har varit svårt på kursen var att skriva RightHandRuleRobot och att göra JUnit testerna. Det krävdes mycket tid för att förstå hur jag skulle kunna testa robotarna i JUnit utan att behöva läsa in en textfil. Fick klura ganska länge på hur jag kunde få min reader att läsa in en sträng istället för en fil men det gick bra tillslut.

Det var även svårt att lära sig att jobba med arv och interface. Jag hade inte arbetat med det så mycket innan och det tog lite tid att förstå hur super() fungerar och vad som får finnas i ett interface osv. Jag tycker att ni borde göra flera sådana här uppgifter som GameOfLife eller Roboten som man kan kolla grafiskt och faktiskt kunna se vad som händer.

När det kommer till programmeringsspråket tycker jag att JAVA är betydligt lättare att använda än C. Det känns mera optimerat och inte lika primitivt som C är. Jag kommer helt klart att fortsätta jobba i JAVA i framtiden!