A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

2020-08-23

Objektorienterad Programmerings metodik

5DV133

Several thin, curved lines in dark blue and light grey that sweep upwards from the bottom left corner of the page.

Namn: [Gustaf Söderlund](#) CS-andändarnamn: [et14gsd](#)
PERSONAL: Johan Eliasson, Jakub Jagiello, Lennart
Steinvall, Fredrik Peteri, Klas af Geijerstam

Innehållsförteckning

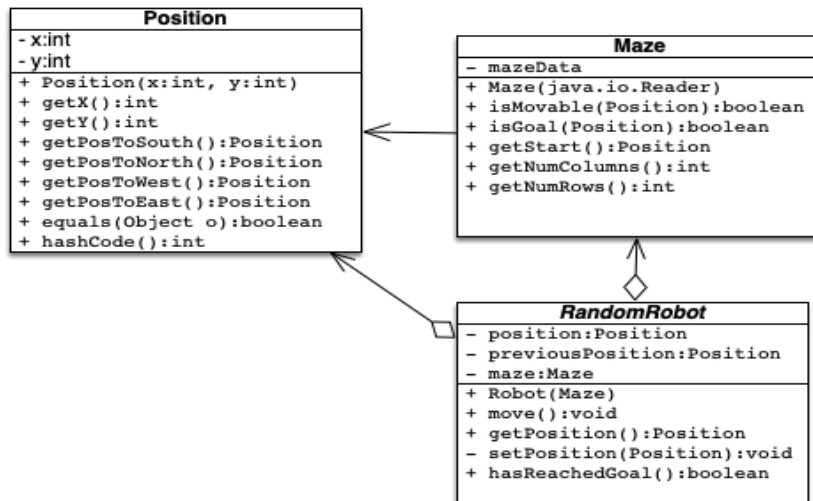
| | |
|---|-----------|
| 1. Introduktion | 2 |
| 2. Problembeskrivning..... | 3 |
| 3. Åtkomst och användarhandledning, inkl. kompilering och körning i terminal. | 4 |
| 4. Systembeskrivning..... | 5 |
| 4.1 Position..... | 6 |
| 4.2 Maze | 7 |
| 4.3 Robot..... | 7 |
| 4.4 AbstractRobot..... | 7 |
| 4.5 RandomRobot..... | 8 |
| 4.5.1 Move()-algorithm..... | 8 |
| 4.6 MemoryRobot..... | 9 |
| 4.6.1 Move()-algorithm..... | 9 |
| 4.6.2 checkIfMovable(Position)..... | 9 |
| 4.6.3 moveRobot(Compass)..... | 9 |
| 4.7 RightHandRuleRobot..... | 10 |
| 4.7.1 Move() - algorithm..... | 10 |
| 4.7.2 Corner(Compass) | 10 |
| 5. Testning..... | 11 |
| 5.1 Mazetest..... | 11 |
| 5.2 PositionTest | 12 |
| 5.3 RobotTest | 12 |
| 5.5 TestKörningar | 13 |
| 6. Reflektioner | 14 |

1. Introduktion

Syftet med den andra obligatoriska uppgiften är att utveckla tidigare robotuppgift. Uppgiften fokuserar på att refaktorisera och bygga vidare på befintlig kod samt att öva på begrepp och implementation av arv. Vältäckande enhetstester för alla modellklasser skrivna med JUnit5 inkluderas.

2. Problembeskrivning

Under den första obligatoriska uppgiften skapades fyra klasser i Java för att en robot skulle orientera sig igenom en labyrint från start till mål. Klasserna som skapades var RandomRobot, Maze, RobotTest och Position. RobotTest var mainprogrammet och klasserna skapades utifrån detta UML-Diagram.



Figur.1 Figuren visar ett UML-diagram från Obligatorisk uppgift 1.

I den andra obligatoriska uppgiften som denna rapport sammanfattar ska två ytterligare robotar skapas. RightHandRuleRobot som ska ta sig från start till mål, men har kravet att alltid ha höger hand på en vägg som inte får släppas. MemoryRobot som ska ta sig från start till mål, men får inte gå på de positioner som roboten redan besökt. Roboten ska kunna backa om den fastnar i en återvändsgränd. Ett Robot interface ska skapas som innehåller de publika metoder som användes i RandomRobot. Arv får fritt bestämmas hur det ska användas.

3. Åtkomst och användarhandledning, inkl. kompilering och körning i terminal.

För att få åtkomst till den färdiga koden kan man besöka <https://git.cs.umu.se/et14gsd/ou2>.

I programmet så används en labyrintfil som måste vara av formatet ".maze". Labyrinten som ska användas har några regler som måste följas annars går det ej att kompilera programmet. Labyrinten måste bestå av en startposition som markeras med "S", en eller flera målpositioner som markeras med "G" och väggarna i labyrinten markeras med "*". Vägen som en robot kan gå på är " ", dvs blanksteg. Se bild nedan som ett exempel på en fungerande labyrint.

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

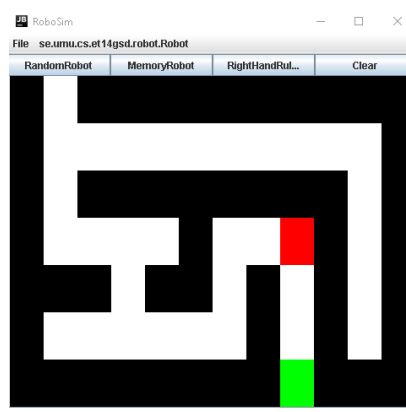
Figur 1 exempel på en labyrint

För att kompilera programmet så behöver man tillsammans med en labyrintfil och de andra nödvändiga filerna lägga allt i en och samma filmapp på en dator. Öppna terminal från en dator eller via terminalen via programmet IntelliJ 11 och ange följande kommando i terminalen:

```
javac MazeSimulation.java (tryck enter)
```

```
java MazeSimulation.java (tryck enter)
```

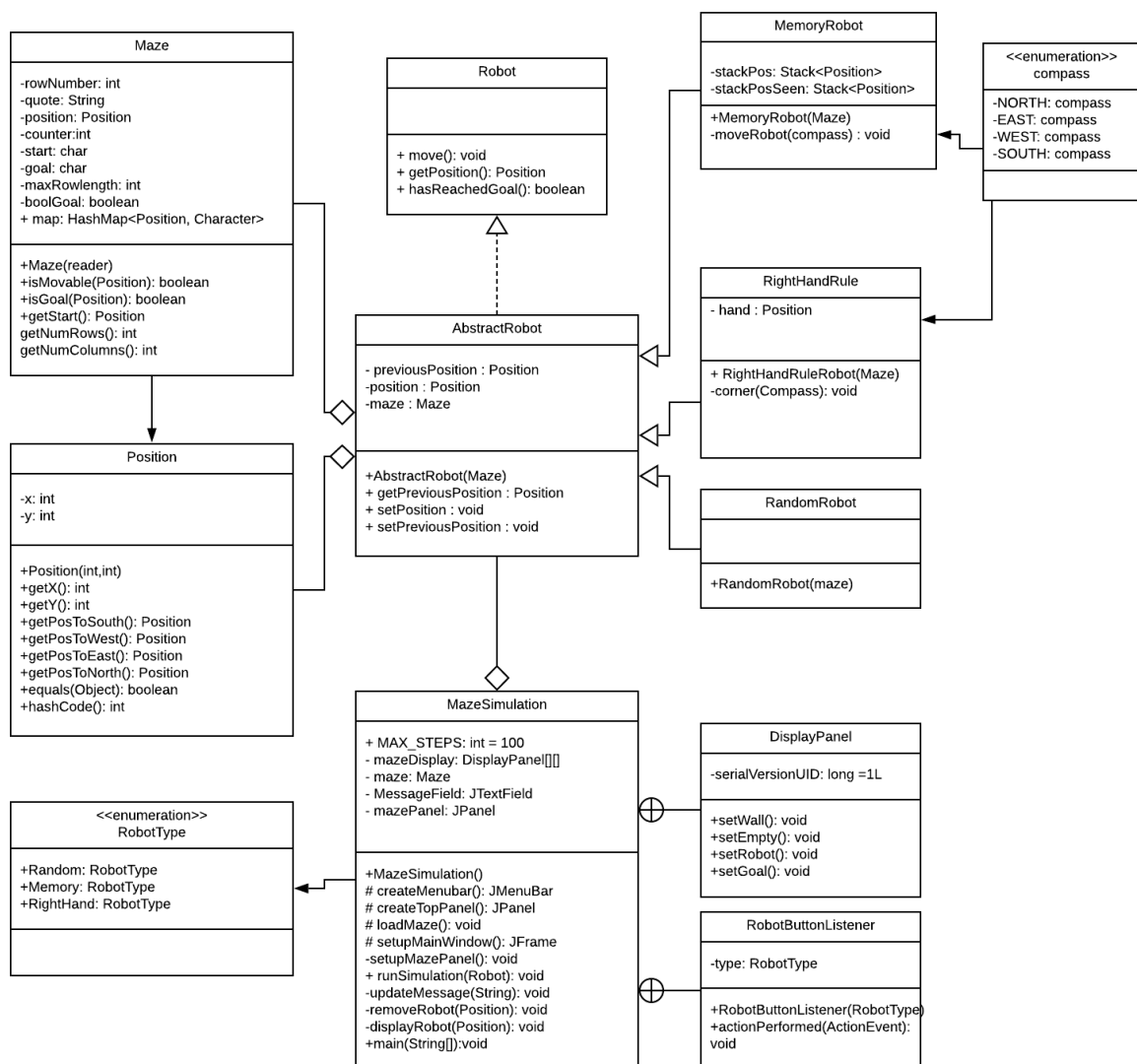
När man kompilerat programmet så kommer användaren få en förfrågan att ange en labyrintfil. Ange då en labyrintfil som uppfyller de krav på en fungerande labyrint. I programmet får användaren välja vilken av de tre robotarna som ska ta sig genom labyrinten. Användaren får då en två dimensionell yta som representerar labyrinten. Den gröna rutan representerar mål, röd ruta representerar roboten och de svarta rutorna representerar Väggar där roboten inte kan röra sig. Här nedan ser man en figur som representerar hur det kan se ut under en körning.



Figur 2: exempelbild tagen på en körning.

4. Systembeskrivning

Ett interface som är namnsatt till Robot har skapats utifrån RandomRobots publika metoder ifrån den första obligatoriska uppgiften. Utifrån interfacet har en superklass ” Två nya robotklasser har skapats MemoryRobot, RightHandRuleRobot. RandomRobot klassen är modifierad så att den ska fungera med interface samt nya metoder. Klasserna Maze och Position är ej förändrade från första obligatoriska uppgiften. MazeSimulation var given inför uppgiften och är mainprogrammet. Ett UML Diagram är skapat för att ge en blick över systemet. Se figur nedan.



Figur 3 UML-Diagram.

4.1 Position

Klassen Position har ansvaret att ge positionernas koordinater i labyrinten. Klassen konstruerades efter UML-diagram som var givet i första obligatoriska uppgiften. Klassen använder sig av attributen X och Y som är heltal. Det finns inga privata metoder i klassen. De publika metoderna är:

Position(int x, int y)

När man skapar en position så tar konstruktorn in två heltals värden, det första värdet blir x-koordinaten och det andra värdet blir y-koordinaten.

getX()

Returnerar int värdet på positionens x koordinat.

getY()

Returnerar int värdet på positionens y koordinat.

getPosToSouth()

Returnerar en ny Position som är en position söder om den nuvarande positionen.

getPosToWest()

Returnerar en ny Position som är en position väster om den nuvarande positionen.

getPosToNorth()

Returnerar en ny Position som är en position norr om den nuvarande positionen.

getPosToEast()

Returnerar en ny Position som är en position öster om den nuvarande positionen

equals(object O)

Returnerar en boolean om två objekt är likadan. Sant om de är likadan annars falskt.

hashCode()

Returnerar x och y värdet på hashkods positionen.

4.2 Maze

Klassen maze har ansvaret att representera en labyrinth. Klassen är konstruerad efter UML-diagramet som var givet i första obligatoriska uppgiften. Klassen använder sig av attributet map som är en hashmap, klassen använder följande publika metoder:

Maze(Reader) är konstruktorn. Maze skapas genom att ta in ett Reader-objekt som läser av labyrinthfilen som används. Labyrinthfilen läses av genom att ta en position i taget och lägga in i en HashMap där positionen är nyckel och symbolen är värdet.

isMovable(Position) metoden är en Boolean som kontrollerar om roboten kan röra sig på en viss position. Metoden returnerar falskt om det finns en vägg på positionen eller om positionen stäcker sig utanför labyrintens område. Annars returnerar den sant.

isGoal(Position) Metoden tar in en position och kontrollerar om x- och y-koordinaten har tecknet "G". Finns det ett "G" på positionen så returneras True annars False.

getStart() Metoden används för att ta fram startpositionen i labyrinten. Returnerar en position som har tecknet "S".

getNumRows() är en metod som används för att räkna antalet rader i labyrinten. Returnerar ett heltal.

getNumColumns() är en metod som används för att räkna antalet kolumner i labyrinten. Returnerar ett heltal.

4.3 Robot

Klassen Robot är ett interface som innehåller de publika metoder som användes i RandomRobot från första uppgiften. De metoder som används i interfacet är:

move() returnerar en void.

getPosition() Returnerar ett object av typen Position.

hasReachedGoal() Returnerar en True eller false.

4.4 AbstractRobot

AbstractRobot är en abstrakt klass till subklasserna MemoryRobot, RighthandRuleRobot och RandomRobot. AbstractRobot ärver från interfacet Robot och använder följande attribut - previousPosition : Position, - position :Position och # maze : Maze

Move() är fortfarande abstrakt och oimplementerad från interfacet

getPosition() returnerar den nuvarande positionen som roboten befinner sig på.

getPreviousPosition() Returnerar den förgående positionen som roboten befann sig på.

hasReachedGoal() Returnerar sant då robotens nuvarande position är en målposition dvs ett "G" annars falskt.

setPosition() Tar in en position och placerar roboten på den positionen.

setPreviousPosition() Tar in en position och sätter den nuvarande positionen som förgående position.

4.5 RandomRobot

I subklassen RandomRobot så är move metoden implementerad på ett sätt så att roboten går till random positioner, men den får inte gå på förgående position. Roboten får bara gå tillbaka till sin förgående position om det fastnar i en återvändsgränd.

4.5.1 Move()-algorithm

Denna metod används i RandomRobot för att få roboten att förflytta sig genom labyrinten. Först skapar man en tom lista och kontrollerar vilka håll roboten kan gå. Kan roboten gå norr, öst, väst eller söder så läggs de tillgängliga väderstrecken in i den tomma listan, det går inte att lägga in den förgående positionen i listan. Sen väljer man ett slumpmässigt väderstreck och förflyttar roboten till den nya positionen och sätter förra positionen som förgående position. Hamnar roboten i en återvändsgränd så kommer roboten att backa ett steg till förgående position.

4.6 MemoryRobot

I subklassen memoryrobot så är move metoden implementerad på ett sätt så att roboten kommer ihåg alla besökta positioner genom att lägga till dem i två olika stackar. Den ena stacken håller reda på robotens nuvarande position och den andra håller koll på vilken väg roboten har vandrat genom labyrinten. Roboten tar sig till mål genom att gå på positioner som ej finns i stackarna. Skulle roboten få problem med att fastna i en återvändsgränd kan roboten gå ett steg tillbaka och ta bort den senaste/översta positionen ur stacken. I memoryrobot används enums *compass(NORTH, EAST, WEST* och *SOUTH)* för att senare användas i en switch.

4.6.1 Move()-algoritm

Denna metod används för att förflytta Memoryrobot genom labyrinten. Först kontrolleras alla riktningar som roboten kan vandra. Om roboten kan vandra en riktning så ska man kalla på den privata metoden moveRobot med det enum som innehåller värdet av väderstrecket som input. Kan roboten inte gå någonstans så kommer roboten att gå samma väg tillbaka genom att kolla efter den översta positionen i stacken och gå tillbaka ett steg. När roboten har backat så plockas översta positionen i stacken som håller koll på vart roboten befinner sig bort.

4.6.2 checkIfMovable(Position)

Denna privata metod tar in en position som ska kontrollera om det är möjligt att röra sig dit. Metoden returnerar tillbaka en bool sant om det är möjligt att röra sig till den positionen och om positionen inte har besökts tidigare. Annars falskt.

4.6.3 moveRobot(Compass)

Denna private metod använder sig av en switch och enums som tidigare har skapats. Returnerar inget. Men flyttar roboten till en position beroende på vilket enum som sätts in i switchen. moveRobot() pushar även in den nuvarande positionen in i en stack för komma ihåg vägen den gått.

4.7 RightHandRuleRobot

I subklassen RightHandRuleRobot så är move metoden implementerad på ett sätt så att roboten ska förflytta sig från start till mål men på ett villkor, att endast hålla sin högra hand mot en vägg under hela simuleringen. Det enda undantaget är att om roboten startar mitt i en labyrinth där det inte finns några väggar i närheten. Då har det beslutats att roboten ska vandra rakt norrut tills man når en vägg och därefter fortsätta vandra med höger hand på väggen. Detta gör roboten genom att alltid kontrollera så att man kan placera en hand på höger sida vid nästa steg med hjälp av detta så kan roboten gå runt ett hörn.

RightHandRuleRobot använder sig av enums (*NORTH*, *EAST*, *WEST* och *SOUTH*) för att senare användas i en switch för att ta sig runt ett hörn.

4.7.1 Move() - algoritm

Denna metod används för att förflytta RightHandRuleRobot genom labyrinthen. Metoden börjar med att kontrollera ifall roboten kan runda ett hörn vilket den endast kommer göra om robotens högerhand inte är på en väggposition. Roboten kommer att gå runt hörnet olika beroende på vilket håll den kommer ifrån dvs att den kollar av förgående position. När roboten vet vilket håll den kommer ifrån så skickas ett enum med ett värde på det väderstreck som roboten ska gå, till den privata metoden corner för att ta sig runt hörnet.

Sen kommer roboten att kontrollera sin miljö och röra sig framåt så länge högerhand är på en väggposition. Varje gång roboten har förflyttat sig ett steg fram så ska handen förflyttas med till nästa position.

Skulle roboten vandrat in i en återvändsgränd så kommer det att kontrollera vilket håll den kan gå åt och backa tillbaka ett steg och förflytta högerhand med sig.

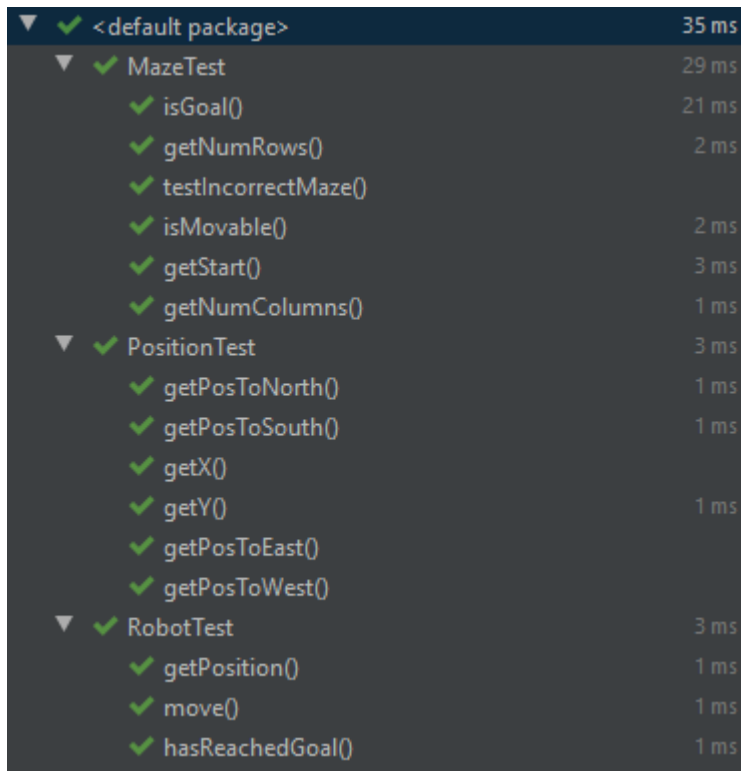
Om robotens startposition är mitt i en labyrinth där det inte finns några väggar omkring sig så ska roboten börja vandra norrut tills den hittar en vägg.

4.7.2 Corner(Compass)

Denna private metod använder sig av en switch och enums som tidigare skapats. Returnerar inget. Men flyttar roboten till en position beroende på vilket enum som sätts in i switchen och placerar handen på höger vägg.

5. Testning

Enhetstestning har genomförts för att kontrollera att publika metoderna i klasserna Maze, Position och Robot fungerar korrekt. Det har gjorts samtidigt som klasserna har konstruerat. Testerna är gjorda med Junit 5.



| | |
|-----------------------|-------|
| ▼ ✓ <default package> | 35 ms |
| ▼ ✓ MazeTest | 29 ms |
| ✓ isGoal() | 21 ms |
| ✓ getNumRows() | 2 ms |
| ✓ testIncorrectMaze() | |
| ✓ isMovable() | 2 ms |
| ✓ getStart() | 3 ms |
| ✓ getNumColumns() | 1 ms |
| ▼ ✓ PositionTest | 3 ms |
| ✓ getPosToNorth() | 1 ms |
| ✓ getPosToSouth() | 1 ms |
| ✓ getX() | |
| ✓ getY() | 1 ms |
| ✓ getPosToEast() | |
| ✓ getPosToWest() | |
| ▼ ✓ RobotTest | 3 ms |
| ✓ getPosition() | 1 ms |
| ✓ move() | 1 ms |
| ✓ hasReachedGoal() | 1 ms |

Figur 4 representerar en testkörning på de olika testerna.

5.1 Mazetest

Testet MazeTest testar klassen Maze för att ta reda på om funktionaliteten är korrekt.

Varje test skapar ett Maze objekt *maze* som lästs in från en fungerande labyrintfil. Nedan förklaras hur testen genomförts.

testIncorrectMaze() testar om filen som man valt att läsa in är av korrekt struktur eller inte. Om detta test fungerar vet man att filen som lästs in är felaktig. Och behöver justeras.

isMovable() skapar en position *posTrue* som redan är en känd position med koordinater som inte är en vägg i den inlästa labyrintfilen. När man använder `assertTrue` på `maze.isMovable` och *posTrue* så bekräftar man funktionaliteten hos `isMovable`.

isGoal() skapar en position *goalpos* som har samma koordinater som en känd målposition i en Labyrintfil där man vet att målpositionen finns. När man använder `assertTrue` på `maze.isGoal(goalpos)` så bekräftas funktionaliteten hos `isGoal`.

getStart() Skapar en position *startPos* som har samma koordinater som en känd position i en labyrintfil där man vet att start positionen finns. När man använder `AssertEquals` på `maze.getStart()` och *startPos* bekräftar man att positionerna är samma och `getStarts` funktionalitet.

getNumRows() skapar ett heltal *rows* som har samma värde som antalet rader i en känd labyrintfil. När man använder `assertEquals` på `maze.getNumRows()` och *rows* bekräftar man *getNumRows* funktionalitet.

getNumColumns() skapar ett heltal *columns* som har samma värde som den längsta raden i en känd labyrintfil. När man använder `assertEquals` på `maze.getNumColumns` och *columns* bekräftas funktionaliteten hos `getNumColumns`.

5.2 PositionTest

Testet `PositionTest` testar klassen `Position` så att den fungerar korrekt. Nedan beskrivs hur testen genomförs.

getX() skapar en position *pos* som har givna x- och y-koordinat. När man använder `assertEquals` på `pos.getX()` och det givna värdet på X så bekräftas funktionaliteten hos `getX()`.

getY() skapar en position *pos* som får en x- och y-koordinat. När man använder `assertEquals` på `pos.getY()` och det givna värdet på Y så bekräftas funktionaliteten hos `getY()`.

getPosToSouth() skapar en position *pos* som får en x- och y-koordinat och en position *south* som har en position söder om *pos*. När man använder `assertEquals` på `pos.getPosToSouth()` och *south* så bekräftas funktionaliteten på `getPosToSouth`.

getPosToWest() skapar en position *pos* som får en x- och y-koordinat och en position *west* som har en position väster om *pos*. När man använder `assertEquals` på `pos.getPosToWest()` och *west* så bekräftas funktionaliteten på `getPosToWest`.

getPosToEast() skapar en position *pos* som får en x- och y-koordinat och en position *east* som har en position öster om *pos*. När man använder `assertEquals` på `pos.getPosToEast()` och *east* så bekräftas funktionaliteten på `getPosToEast`.

getPosToNorth() skapar en position *pos* som får en x- och y-koordinat och en position *north* som har en position Norr om *pos*. När man använder `assertEquals` på `pos.getPosToNorth()` och *north* så bekräftas funktionaliteten på `getPosToNorth`.

5.3 RobotTest

Testet `RobotTest` testar klassen `Robot` för att ta reda på om funktionaliteten är korrekt. Varje test skapar ett `Maze` objekt *maze* som lästs in från en fungerande labyrintfil. Nedan beskrivs hur testen genomförs.

move() skapar en `RandomRobot` *testRobot* i en labyrint. Med hjälp av `Position` *testPos1* sparas robotens nuvarande position och tillsammans med `testRobot.move()` förflyttas roboten. Genom att använda `assertNotEquals` på `testRobot.getPosition()` och *testPos1* bekräftar vi funktionaliteten hos `move`.

getPosition() skapar en `RandomRobot` *testRobot* i en labyrint. Genom att använda `testRobot.getPosition` och spara koordinaterna i en `Position` *testpos* så kan man jämföra tillsammans

med `assertEquals` på `testPos` och `maze.getStart()`. Om positionerna är samma så bekräftar man funktionaliteten hos `getPosition`.

`hasReachedGoal()` skapar en `RandomRobot` *testRobot* i en labyrinth. Så länge *testRobot* inte har nått mål så ska `testRobot.move()` användas. Position *goal* får en känd målposition i en känd labyrinth. När man använder `assertEquals` på `maze.isGoal(goal)` och `testRobot.hasReachedGoal()` vet man att roboten har nått mål och funktionaliteten hos `hasReachedGoal` är bekräftad.

5.5 TestKörningar

Testkörningar gjordes i jämna mellanrum för att kontrollera så att robotarnas beteende är korrekt enligt specifikationen. För att säkerställa att `move`-algoritmen fungerade korrekt testades ett flertal olika labrynter. Labrynter med flera/inga målpositioner, flera/inga startpositioner, olika former på labrynten och ett antal återvändsgränder från olika håll testades för att säkerställa att roboten inte fastnade. De tre robotarna klarade sig igenom alla tester vilket bekräftar deras funktionalitet.

6. Reflektioner

Jag tycker att uppgiften var intressant. Det krävdes mer kunskap och arbete för denna uppgift, vilket jag tyckte var positivt. Uppgiften var som sagt utmanande, men intressant. Jämfört med C-programmering så tycker jag att det är roligare och enklare att programmera i java.

De problem som jag stött på under projektet var att jag tyckte robotarna såg enkla ut att koda enligt specifikationen, men efter att man börjat koda och stött på första problemen så upptäckte jag att det var lite svårare än vad jag trodde. Jag hade lite problem med hur git fungerade i början, men fick lära mig efter lite research på nätet.