UMEÅ UNIVERSITET
Institutionen för Datavetenskap

8 maj 2023

[5DV230] Objektorienterad programmering (Java)

OU2 Robot 2.0

version 2.0

Namn	Daniel Hylander
CS-användare	ens21dhr
UMU-id	dahy0007

Personal
Johan Eliasson, Anton Degerfeldt, James Eriksson
Abdulsalam Aldahir, Oscar Kamf, Michael Minock, Mohammad Mshaleh

Innehåll

1 Problembeskrivning

Problemet bygger på att utöka ett robot navigations program med nya klasser och ett gränssnitt för robot klasserna. Programmet simulerar hur olika robotar traveserar en labyrint, vilket skulle göras med ett antal förbestämda klasser. Total var det sex klasser, en *Maze* (labyrint), *Position* (position), *Robot* (gränssnitt), med tre olika robotar *RandomRobot*, *RightHandRuleRobot* och *MemoryRobot*.

I föregående uppgift skapades *Position*, *Maze* och *RandomRobot* klasserna, som skulle utökas på med två nya robotar *RightHandRuleRobot* och *MemoryRobot*. Ett gränssnitt Robot skulle skapas som alla robotar skulle implementera inklusive *RandomRobot*.

2 Introduktion

I denna rapport, beskriver vi hur programmet ska användas och dess struktur. Vi visar algoritmerna som används för att bestämma hur robotarna rörs sig genom en labyrint, programmets begränsningar följt av hur programmet testades. Vi avslutar med reflektioner om projektet.

3 Användarhandledning

För att använda programmet måste källkoden för programmet först laddas ner, vilket kan göras via följande git-länk[4]. När källkoden är nedladdad behöves den kompileras, refera till sektionen om kompilering. Efter kompilering är klar kan man exekvera MazeSimulator, och ett GUI öppnas.

Använda programmet med MazeSimulator:

1. Välj vilken labyrint fil som ska användas under simulationen.
2. Navigera till rot mappen av projektets källkod.
3. Från rot mappen navigera till "*test_files/mazeTest*" och välj en labyrint.
4. Nu bör en labyrinten vissas i fönstret, och man kan välja vilken/vilka robot;ar som ska köras.

Man kan även använda andra än dem givna labyrinterna att simulera. Det kan göras genom att skapa en ny text fil som följer programmets labyrint format, refera till javadoc för Maze klassens konstruktör[2]. Sedan väljer man den nya text filen när man exekverar MazeSimulator.

3.1 Kompilering

För att kompilera källkoden behöver man följande programvara installerad:

- Java Development Kit (JDK) version 17 eller högre

Sedan för att kompilera källkoden, utför följande steg:

1. Öppna en terminal och navigera till rot mappen av projektets källkod.

2. Kör följande kommando i terminalen
   "*javac java_files/MazeSimulation.java java_files/model/*.java*"

3. Programets main fil kommer att ligga vid "*java_files/MazeSimulation.class*".

4 Systembeskrivning

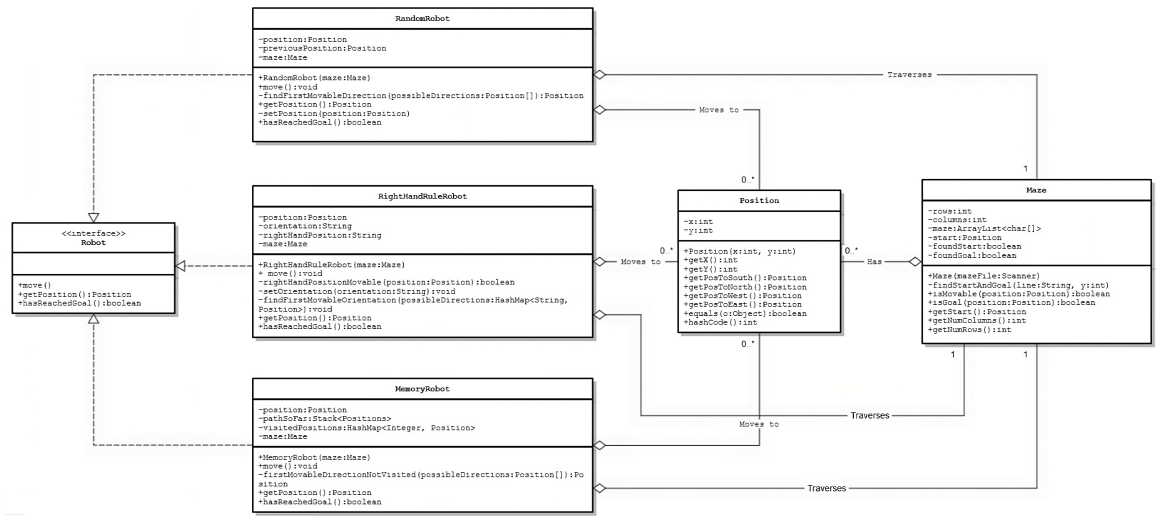
Den interna strukturen av programmet består av sex klasser, tre olika robot klasser, ett gränssnitt, en position klass och labyrint klass. Dem tre grund klasserna kommer först att beskrivas: position klass, labyrint klass och gränssnittet följt av en ny sektion om robot klasserna.

Positions klassen representera en *(x, y)* koordinat inuti en labyrint. Från ett positions objekt kan man hämta positionen brevid i alla fyra kardinal riktningar, där varje positions har en unik hash-kod. För mer detaljer av metoderna refera till javadoc för Position.java[3].

Labyrint klassen skapar en labyrint från en text fil och validerar att labyrinten har rätt format. Labyrinten traveseras sedan av en robot. Från ett labyrint objekt kan man se om en positions i labyrinten är gå bar. Man kan också se om en position är labyrintetns mål eller start position. För mer detaljer av metoderna refera till javadoc för Maze.java[2].

Varje robot använder sig av en gränssnitt som har tre abstrakta metoder: *move()*, *getPosition()* och *hasReachedGoal()*. För att flytta roboten ett steg i labyrinten används *move()* metoden medans *getPosition()* ger robotens nuvarande position och *hasReachedGoal()* kollar om roboten har nått slutet av labyrinten. En utförliga beskrivning av robotarna beskrivs nedan.

Relationerna mellan klasserna är illusterat med UML-diagrammet i figur 1.



Figur 1: UML-diagrammet som illustrerar relationerna mellan klasserna för programmet.

4.1 Robot klasser och Algortimer

Programmet består av tre olika robot klasser *RandomRobot*, *RightHandRuleRobot* och *MemoryRobot*. Där *move()* metod för robotarna bestämmer hur dem traveserar en labyrint. Vi kommer att gå igenom robotarnas syfte och algoritmen for *move()* metoden för vardera robot.

RandomRobot skapar en robot som traveserar en labyrint genom att välja en slumpmässig position bredvid sig för varje steg. Positionen får inte vara lika med dess föregående position om en sådan existerar, annars går den tillbaka till den föregående positionen. Algorimten för *move()* metoden kan ses nedan:

```
Skapa en array och sätt in varje position brevid roboten
i slumpmässig ordning
För varje position i arrayen
    Om positionen är gå bar och den inte är lika med
    föregående position
        Sätt föregående position till nuvarande position
        Flytta roboten till positionen
Om ingen position hittades
    Spara nuvarande postion i en temp variablen
    Flytta roboten till föregående position
    Sätt föregående positon till positionen i temp variablen
```

RightHandRuleRobot skapar en robot som traveserar en labyrint på ett sådant sätt som en människa hade gjort om den navigerat runt i labyrinten och hela tiden hållit höger hand mot en vägg i labyrinten. Algoritmen för *move()* metoden kan ses nedan:

```
Skapa ett hashtabel och sätt in varje position brevid roboten
men en nyckel lika med positionens hash kod
Om positionen till höger om roboten är gå bar
    Flytta roboten till positionen
Annars
    Tills en gå bar positions hittas
        Om positionen framför roboten inte är gå bar
            Om orientera roboten
    Flytta roboten till positionen
```

MemoryRobot skapar en robot som traveserar en labyrint genom en djupet-förstsökning av labyrinten, genom att den håller reda på positionerna den besökt- samt vägen den gått hittills. Algoritmen för *move()* metoden kan ses nedan:

```
Skapa en stack för robotens väglista
Skapa en hashtabel för alla besökta positioner
Skapa en array och sätt in varje position brevid roboten
För varje position i arrayen
    Om positionen är obesökt och gå bar
        Lägg till positionen i stacken
        Flytta roboten till positionen
Om ingen position hittades och stacken inte är tom
```

```
    Ta ut första positionen i stacken och flytta roboten
    till positionen
Annars
    Flytta roboten till nuvarande positionen
Lägg till nuvarnde postionen i hastabelen om den inte
redan är med
```

För mer detaljer om metoderna för robot klasserna refera till javadoc för RandomRobot.java, RightHandRuleRobot.java och MemoryRobot.java[1].

4.2 Lösningens begränsningar

Även om programmet erbjuder den funktionalitet den behöver, finns det vissa begränsningar som behöves åtgärdas för att programmet ska nå sin fulla potential. I den här sektionen beskrivs dessa begränsningar med rekommendationer för att åtgärda dem.

Under programmets utveckling och testning har följande begränsningar hittats:

- Om labyrinten har en större öppen yta (4x4 celler) utan väggar kommer *RightHandRuleRobot* att gå i cirklar.

För att åtgärda dessa begränsningar är följande rekommenderat:

- Kolla så att roboten alltid har en vägg mot sin högra sida, annars kan man låta roboten gå fram till en vägg är hittat.

5 Testning

Testerna som kördes under programmets uppbyggnad utfördes för att validera programmets funktionalitet. I den här sektionen beskrivs test strategin, vilka tester som utfördes och testens resultaten.

Testmetoden som användes under programmets utveckling var enhetstester, som utfördes med JUnit5.

Följande test utfördes:

- *Position*
  - Hämta sin egen position: för att säkerställa att ett positions objekt kan hålla reda på sin egen position.
  - Hämta alla positionerna i kardinal riktningarna: för att säkerställe att den ger rätt position till en robot som vill flytta sig i en specifik riktning.
  - Kolla om två positions objekt vid samma position är ekvivalenta.
  - Kolla om två positioner har unika hash koder: för att kunna få unika nycklar i en hashtabel för en position.
- *Maze*

  - Kolla start och mål positionen: för att säkerställa att labyrinten har rätt format.
  - Kolla rad och kolumn antalet.
  - Kolla om en maze kan hitta gå bara positioner: för att säkerställa att en robot kan flytta sig en en labyrint.

- *RandomRobot*
  - Kolla start positionen för roboten: för att säkerställa att roboten start på rätt plats.
  - Kolla att roboten kan röra sig: för att säkerställa att roboten kan travesera en labyrint.
  - Kolla att roboten kan gå till den föregående positionen om det inte finns någon annan gå bar position: för att säkerställa att roboten följer specifikationen.
  - Kolla att roboten kan travesera olika typer av labyrinter.

- *RightHandRuleRobot*
  - Kolla start positionen för roboten: för att säkerställa att roboten start på rätt plats.
  - Kolla att roboten kan röra sig: för att säkerställa att roboten kan travesera en labyrint.
  - Kolla att roboten håller sig till höger sidan vid varje förflyttning: för att säkerställa att roboten följer specifikationen.
  - Kolla att roboten kan travesera olika typer av labyrinter.

- *MemoryRobot*
  - Kolla start positionen för roboten: för att säkerställa att roboten start på rätt plats.
  - Kolla att roboten kan röra sig: för att säkerställa att roboten kan travesera en labyrint.
  - Kolla att roboten inte går till en position som den redan varit på: för att säkerställa att roboten följer specifikationen.
  - Kolla att roboten kan travesera olika typer av labyrinter.

Programmet passerade alla tester som var implementerade förutom RightHandRuleRobot som inte kunde travesera en labyrint typ, men resultaten vissade att programmet ändå möter specifikationen för funktionaliteten. Som sagt vissa begränsningar hittades under testningen, refera till sektion 3.2 om begränsningar.

6 Reflektioner

Det var en stor skillnad att skriva i java i jämförelse men C. Istället för att behöva implementera varje metod eller funktionalitet som programmet skulle utföra kunde man hitta en klass i java som utförde samma uppgift. Det har gjort koden mer lättläst och mer kompakt. Problemet uppstår då man behöver veta vilka klasser som man kan använda och vilka existerar i java.

Det var väldigt mycket nytt som man behövde lära sig under denna uppgift, inte bara var det java men också att skriva tester i JUnit och använda git. Själv använde jag mig av VScode istället för Intelij, vilket gav upphov till en del små problem att lösa. Något som jag inte lyckades få till var att se var hur mycket täckning av koden mina JUnit test hade i VScode. Men i det stora hela hade jag inga större problem, bara en hiskligt många små problem, som tillsammans tog sin tid att fixa.

Referenser

[1] Daniel Hylander. *javadoc*. URL: https://people.cs.umu.se/ens21dhr/ou2_doc/model/package-summary.html. (accessed: 27.04.2023).

[2] Daniel Hylander. *Maze.java javadoc*. URL: https://people.cs.umu.se/ens21dhr/ou2_doc/model/Maze.html. (accessed: 27.04.2023).

[3] Daniel Hylander. *Position.java javadoc*. URL: https://people.cs.umu.se/ens21dhr/ou2_doc/model/Position.html. (accessed: 27.04.2023).

[4] Daniel Hylander. *Project git-repository*. URL: https://git.cs.umu.se/ens21dhr/ou2_robot2.0. (accessed: 27.04.2023).