Objektorienterad programmering med Java
VT 2019

1DL028 10hp

Uppgift 4

Simulering av ekologisk värld

av

Tobias Isaksson

Department of Information Technology

Uppsala Universitet

Datorpost:

tobbe.isaksson@gmail.com

Innehållsförteckning

1. Inledning

1.1. Versioner

Oracles Java JDK version 11.0.2
Javac version 11.0.2
java version "11.0.2" 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)
Notepad++ v7.6.6

2. Programmet

2.1. Kompilera och köra programmet

Programmet kompileras med `javac -encoding utf-8 Param.java`.
För att starta programmet finns två vägar. Vill man enbart starta simuleringen med default-parametrar så kör man `java Pasture`. Önskar man istället ändra simuleringens parametrar startar man med `java Param`.

2.2. Programmet och dess uppbyggnad

Lösningen av uppgiften slutade med 11 olika klasser. Sex av dessa är till för de olika Entiteter som kan förekomma i hagen. Av dessa sex är en superklass för samtliga entiteter, ytterligare en är superklass för entiteter som kan förflytta sig och de fyra kvarvarande är en för var typ av entitet. De två klasserna `PastureGUI.java` och `Engine.java` är helt oförändrade från de exempelfiler som gavs med uppgiften. De övriga klasserna bygger mer eller mindre även de på de exempel som gavs.

2.3. A.I.

Får och vargar har en mycket simpel artificiell intelligens som styr deras beteende. Gemensamt för de båda är att de före en förflyttning ”tittar” sig omkring för att se om det finns något ätbart inom dess angivna synfält. Finner den något ätbart så ändrar den riktning för att förflytta sig dit. Finns mat i flera riktningar väljs en av dem slumpvis.
Fåren skall även fly undan vargar och det skall vara viktigare än att äta. Så i fårens fall så kontrollerar de omgivningen för vargar efter att ha kollat efter mat. Finner den en varg ändrar den riktning i motsatt riktning.
Här finns utrymme för en del förbättringar vilka jag nämner lite om i en senare punkt.

2.4. Balans i ekosystemet

I uppgiften nämns att simuleringen körd med default-parametrar skall resultera i någorlunda balans i det lilla ekosystemet. Efter att ha lagt mycket tid på att laborera med parametrarna så verkar det vara mycket svårt att uppnå. Det faktum att alla entiteters startplacering i hagen slumpas ut påverkar för mycket. Parametrar som vid en körning kan ge skaplig balans och låter simuleringen pågå länge utan att någon typ av entitet tar över eller blir utrotad kan vid nästa körning ge helt annat resultat. Efter samråd med kursens lärare har parametrarna istället försökts sättas så att var körning inte ger samma utfall.

3. Klasser och dess metoder

Klasserna `PastureGUI.java` och `Engine.java` kommer inte tas upp då de är oförändrade från exemplen som gavs i uppgiften. De metoder som är oförändrade från exemplen kommer inte heller gås igenom eller förklaras.
I uppgiftens simulerade värld förekommer fyra olika entiteter: Får, vargar, gräs och staket. Var och en av dessa fyra har en egen klass vilka är: `Sheep.java`, `Wolf.java`, `Plant.java` och `Fence.java`. Till dessa finns för alla entiteter superklassen `Entity.java` och för de entiteter som förlyttar sig superklassen `Moving.java`.

3.1. Superklass Entity.java

Superklass till samtliga fyra typer av entiteter som kan förekomma i simuleringen. Denna var i den givna exempelkoden given som ett interface. I denna lösning ändrades den till en klass då det fanns en tanke om att flytta upp vissa metoder i den. Nu vart inte det fallet men den lämnades definierad som en klass istället för interface i alla fall.

3.2. Klass Fence.java

Denna klass representerar den enklaste av de fyra entiteter som förekommer i simuleringen, staketet. Klassen `Fence.java` ärver från `Entity.java`.

Konstruktor Fance

Konstruktorn tar som enda parameter ett objekt i from av den pasture det skall sättas in.

Metod getImage

Returnerar en `imageIcon` innehållande den bild som representerar staket.

Metod isCompatible

Metoden används för att se om denna entitet kan förekomma på samma ruta som en annan entitet. Som parameter anges den entitet som kontrollen skall göras mot. I detta fall returneras alltid `false` eftersom inga andra entiteter får förekomma på samma ruta som ett staket.

Metod tick

Denna metod anropas en gång per tidsenhet i simuleringen. Eftersom staket i denna simulering vare sig åldras, förökar sig eller flyttar på sig görs ingenting i denna metod.

3.3. Klass Plant.java

I det simulerade ekosystemets hage skall det växa gräs vilket representeras av denna klass. Klassen ärver från `Entity.java`.

Konstruktor Plant

Parametrar för att skapa en ny instans av klassen `Plant` är vilken `pasture` den skall sättas in, vid vilken ålder den skall föröka sig första gången och hur ofta den därefter skall föröka sig.

Metod getImage

Returnerar en `imageIcon` innehållande den bild som representerar gräs.

Metod isCompatible

Precis som för klassen `Fence` så används denna metod för att kontrollera om denna entitet kan förekomma på samma ruta som en annan. Parametern som skall skickas med är en instans av en entitet-klass. I detta fall returneras `false` om entiteten som jämförs mot är `Fence` eller `Plant`, annars `true`. Gräs kan växa på alla rutor utom där det finns staket eller redan växer gräs.

Metod tick

Som för alla entitet-klasser så anropas denna metod en gång per tidsenhet i simuleringen. Innan något annat utförs i denna metod kontrolleras om aktuell entitet fortfarande finns kvar i hagen. Gräset kan ha blivit uppätet av ett får och då skall inget alls utföras. När det gäller gräs så skall det enbart hållas koll på om det är dags att föröka sig. Detta görs genom att variabeln `age` i kontstruktorn sätts till antalet tidsenheter som skall passera innan denna entitet förökar sig första gången. Denna variabel räknas sedan ned med ett för var tidsenhet och när noll nås förökar sig entiteten. När detta utförts så sätts `age` istället till antalet tidsenheter mellan förökningar.
Själva förökningen sker genom att anropa metoder i `pasture` för att hitta ledig ruta/rutor och sätta in en ny instans av denna entitet där.

3.4. Superklass Moving.java

Entiteterna `Wolf` och `Sheep` har förmågan att förflytta sig och här en denna klass som genemsam superklass. Denna klass används för att hålla de metoder som är gemensamma just för entiteter som kan förflytta sig i hagen. `Moving` ärver från klassen `Entity`.

Metoden randomDirection

Slumpar fram värden i spannet -1 till 1 för rörelseriktning i X-led och Y-led. Dessa sätts in i en variabel av typen `Point` som håller aktuell entitets rörelseriktning.

Metoden movement

Tar vilken pasture aktuell entitet finns i som parameter och anropar dess metoder för att förflytta entiteten. Misslyckas förflyttningen sätts en ny slumpmässig riktning och `false` returneras. Lyckas förflyttningen returneras `true`.

Metoden findFood

Med hjälp av tre de tre parametrarna: klasstyp att hitta, pasture att leta i och synfält letar denna metod på angränsade rutor av rätt typ och sätter riktningen mot en av dem.

3.5. Klass Wolf.java

Simuleringens rovdjur, varg, representeras av denna klass. Denna entitet är lite mer komplicerad än `Plant` då den förutom att föröka sig även flyttar på sig, letar föda och äter. Klassen ärver från superklasen `Moving`.

Konstruktor Wolf

Konstruktorn för denna entitet tar som parametrar vilken pasture den skall sättas in i, vid vilken ålder den kan föröka sig, hur ofta den skall föröka sig, hur fort den förflyttar sig, hur länge den klarar sig utan att äta och hur stort synfält den har.

Metod getImage

Returnerar en `imageIcon` innehållande den bild som representerar entiteten varg.

Metod tick

Metoden anropas en gång per tidsenhet i simuleringen och har hand om allt för den entitet som påverkas av tidens gång. Innan något annat görs kontrolleras att denna entitet finns kvar i hagen. Finns den inte kvar skall inget alls utföras och metoden lämnas direkt.
Var gång denna metod anropas räknas tre variabler ned med ett: `age`, `speed` och `lastEat`. Den första, `age`, används för att hålla koll på entitetens ålder så att det går att veta om det är dags att föröka sig, `speed` används för att kontrollera om det är dags för en förflyttning och `lastEat` räknar ned tiden sedan senast entiteten åt.
Efte att det konstaterats att entiteten finns kvar i hagen så kontrolleras om den står på samma ruta som en entiet av klassen `Sheep`. I så fall skall den entiteten ätas. Denna kontroll görs med hjälp av metoderna i instansen av `Pasture` för att få en lista över entiteter i en specifik ruta. Finns det en instans av klassen `Sheep` i denna lista så plockas den bort, tiden `lastEat` återställs och `hasEaten` sätts till `true` vilket tillåter att denna entitet förökar sig.
Nästa steg i metoden `tick` är att kontrollera att entiteten ätit i tid. Har den inte det så tas den bort och metoden avslutas direkt.
Vidare kontrolleras om det är dags för reproduktion. Om så är fallet läggs en ny entitet av klassen `Wolf` till i hagen och tiden för reproduktionshastighet sätts till startvärde igen.
Sista steget i metoden är att kontrollera om det är dags för förflyttning. Är det dags för förflyttning så kontrolleras först om det finns mat inom synfältet, i detta fall letas efter entiteter av typen `Sheep`. Hittas mat i synfältet så ändras riktningen mot denna och ett försök till förflyttning görs. Misslyckas förflyttningen så återställs inte timern för förflyttning till startvärde. Detta gör att ett nytt försök till förflyttning görs omedelbart nästa tidsenhet.

Metod isCompatible

Jämförelse metod för att se om denna entitet kan förekomma på samma ruta som en specifik annan entitetstyp. Returnerar `false` för entiteter av typen `Fence` och `Wolf` eftersom vargar inte kan förekomma på samma ruta som staket eller andra vargar.

3.6. Klass Sheep.java

Simulering av hagens gräsätare, får, representeras av denna klass. Precis som vargen är denna entitet är lite mer komplicerad än `Plant` då den förutom att föröka sig även flyttar på sig. Utöver att precis som `Wolf` flyttar på sig, letar föda och äter så försöker den även undvika entiteter av just klassen `Wolf`. Klassen ärver från superklasen `Moving`.

Konstruktor Sheep

Konstruktorn för denna entitet tar som parametrar vilken `pasture` den skall sättas in i, vid vilken ålder den kan föröka sig, hur ofta den skall föröka sig, hur fort den förflyttar sig, hur länge den

klarar sig utan att äta och hur stort synfält den har.

Metod getImage

Returnerar en `imageIcon` innehållande den bild som representerar entiteten får.

Metod tick

Metoden anropas en gång per tidsenhet i simuleringen och har hand om allt för den entitet som påverkas av tidens gång. Innan något annat görs kontrolleras att denna entitet finns kvar i hagen. Finns den inte kvar skall inget alls utföras och metoden lämnas direkt.
Var gång denna metod anropas räknas tre variabler ned med ett: `age`, `speed` och `lastEat`. Den första, `age`, används för att hålla koll på entitetens ålder så att det går att veta om det är dags att föröka sig, `speed` används för att kontrollera om det är dags för en förflyttning och `lastEat` räknar ned tiden sedan senast entiteten åt.
Efte att det konstaterats att entiteten finns kvar i hagen så kontrolleras om den står på samma ruta som en entiet av klassen `Plant`. I så fall skall den entiteten ätas. Denna kontroll görs med hjälp av metoderna i instansen av `Pasture` för att få en lista över entiteter i en specifik ruta. Finns det en instans av klassen `Plant` i denna lista så plockas den bort, tiden `lastEat` återställs och `hasEaten` sätts till `true` vilket tillåter att denna entitet förökar sig.
Nästa steg i metoden `tick` är att kontrollera att entiteten ätit i tid. Har den inte det så tas den bort och metoden avslutas direkt.
Vidare kontrolleras om det är dags för reproduktion. Om så är fallet läggs en ny entitet av klassen `Sheep` till i hagen och tiden för reproduktionshastighet sätts till startvärde igen.
Sista steget i metoden är att kontrollera om det är dags för förflyttning. Är det dags för förflyttning så kontrolleras först om det finns mat inom synfältet, i detta fall letas efter entiteter av typen `Plant`. Hittas mat i synfältet så ändras riktningen mot denna. Innan ett försök till förflyttning inleds så görs ytterligare en kontroll i synfältet. Denna gång letas det efter entiteter av klassen `Wolf` vilka skall undvikas. Hittas en entitet av klassen `Wolf` i synfältet så sätts riktningen att peka från denna och ett försök till förflyttning görs. Misslyckas förflyttningen så återställs inte timern för förflyttning till startvärde. Detta gör att ett nytt försök till förflyttning görs omedelbart nästa tidsenhet.

Metod isCompatible

Jämförelse metod för att se om denna entitet kan förekomma på samma ruta som en specifik annan entitetstyp. Returnerar `false` för entiteter av typen `Fence` och `Sheep` eftersom får inte kan förekomma på samma ruta som staket eller andra får.

3.7. Klass Parameters.java

Denna klass har till uppgift att hålla programmets startparametrar och default-parametrar. Den har även ett metod för att validera inmatningen. Samtliga instansvariabler är deklarerade som publika för enkel åtkomst hos de klasser som behöver det. Klassen `Param` skapar en instans av `Parameters` för att visa defaultparametrarna och ta emot eventuella förändringar användaren gör. `Param` skickar sedan denna information vidare till konstruktorn i klassen `Pasture` där simuleringen sätts upp.

Metod validate

Metoden anropas för att kontrollera om värden som matats in ligger inom tillåtna gränser. Här finns också alla värden för vad som är tillåtna som minimum och maximum för var och en av parametrarna. Dessa gränser ligger direkt i koden som ”magiska nummer” något som normalt skall undvikas. Men då denna klass har som enda uppgift att hålla parametrar och dessutom är väldigt liten valde jag att göra ett avsteg från principen istället för att deklarera 34 privata instansvariabler i början av klassen.

3.8. Klass Param.java

För att användaren skall kunna mata in andra parametrar en default för simuleringen så används denna klass för att skapa ett fönster för inmatning. Klassen bygger helt på exempelfilen som gavs i uppgiften. Vilka värden som efterfrågas har ändrats och en validering av värdena har lagts till innan de skickas vidare.

3.9. Klass Pasture.java

Även denna klass bygger på exemplet som gavs i uppgiften. Klassen har dock blivit dubbelt så många rader lång när nya metoder lagts till och befintliga modifierats. I denna genomgång kommer främst beskrivas vad som ändrats i klassens olika metoder och varför. Metoder som är oförändrade kommer inte tas upp alls.

Konstruktor Pasture

Konstruktorn tar som parameter en instans av klassen `Parameters` innehållande de värden som simuleringen skall startas med. Dessa startvärden förs sedan över till privata klassvariabler. När detta är gjort anropas en ny metod `startNewPasture`.

Metod startNewPasture

Första delen av denna metod där det skapas instanser av `Engine` och `PastureGUI` samt sätts ut staket är oförändrad från det givna exemplet. Dock är de delarna flyttade hit från att i exemplet ha legat i konstruktorn.
Nästa steg är att sätta ut rätt antal av `Fence`, `Plants`, `Sheep` och `Wolf` på slumpmässiga platser i hagen.

Metod findNeighboursOfType

Detta är en modifering av exemplets metod `getFreeNeighbours` som returnerar en lista med entiteter av given typ istället för lediga rutor. Parametrar som krävs är vilken entitet det är som tittar sig omkring, vilken typ av entitet den letar efter och hur långt synfältet är. Är synfältet längre än 1 så börjar funktionen med att först titta i de närmast angränsade rutorna. Hittar den entieter av eftersökt typ där tittar den inte vidare längre ut. Finner den dock inte vad den söker utökas sökområdet ett steg och det letas vidare. Metoden används när får och vargar letar mat samt när fåren försöker undvika vargar. Finns det mat eller vargar på närmsta rutan så behöver man ju inte leta i hela synfältet.

4. Tänkbara Förbättringar

Det finns mycket i programmet som skulle kunna förändras, snyggas till och förbättras men som på grund av tidsbrist inte blev gjort. Nedan listas några av dessa.

4.1. A.I

De simulerade entiteterna har i dagsläget en väldigt simpel artificiell intelligens. När de entitetstyper som äter tittar sig omkring efter mat så förflyttar de sig slumpvis till en av de angränsade rutor som har mat. Här skulle man kunna ordna så att de först tittar i den riktning de redan var på väg och om mat finns däråt så fortsätter de i samma riktning. Man skulle till och med kunna få fåren att titta i sin förlyttningsriktning i hela sitt synfälts längd efter mat. Finns mat i den riktningen som fåret hinner nå utan att svälta ihjäl så skulle det kunna fortsätta i den riktningen och ignorera mat som är närmare.
När fåren i denna simulering flyr från vargarna är de inte speciellt intelligenta om de blir omringade av vargar. Hittar de vargar på samma avstånd i flera riktningar så flyr de slumpmässigt från en av dem. Här kan man tänka sig en implementering av någon form av viktning av flyktriktningar så fåret flyr på ett bättre sätt från alla vargar det ser samtidigt.

4.2. GUI för parameterinmatning

Det fönster som används för att låta användaren mata in startparametrar till simuleringen bygger i stort sett helt och hållet på givet exempel. Här skulle man kunna snygga till en del genom att gruppera inmatningsfälten och sätta in lite rubriker.

4.3. Koduppprepning

I framförallt klasserna för `Sheep` och `Wolf` förekommer en del koduppprepning. Det är några metoder som är väldigt/helt lika. Dessa skulle kunna städas upp, göras lite mer generella och flyttas upp i en superklass.