Objektorienterad programmering med Java, 10 hp
ST 2010

Inlämningsuppgift 4
Simulering av ekologisk värld

Sebastian Lundström (selu7901)

6 augusti 2010

1 Inledning

Denna rapport behandlar design och implementation av ett program som simulerar livet i en hage. I hagen kämpar olika varelser för sitt liv genom att jaga mat och föröka sig.

Lösningsförslag har diskuterats med Jon Borglund.

2 Installation

Dessa instruktioner förutsätter att du befinner dig i en terminal i samma mapp som programmets Makefile. Du förutsätts även ha tillgång till Java 5 (eller senare) och programmet make. Allt som beskrivs i denna sektion uttrycks i mer detalj i programmets Makefile.

För att både kompilera och köra programmet skrivs enklast (där $ representerar kommandoprompten):

```
$ make
```

Detta kommando kompilerar alla .java-filer och kör sedan det resulterande programmet. Vill du endast kompilera programmet skriver du:

```
$ make compile
```

Vill du endast köra programmet skriver du:

```
$ make run
```

Programmet går att konfigurera genom att ge parvisa argument:

```
$ java pasture.Main     \
    sheep.move.after 4  \
    sheep.visibility 3  \
    wolf.move.after  5  \
    wolf.visibility  4
```

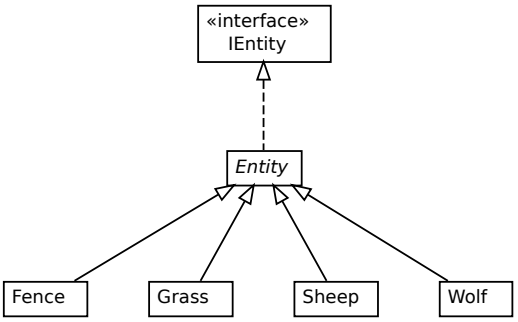
Ovanstående gör att får och vargar flyttar sig efter 4 resp. 5 tick, och att de kan se 3 resp. 4 rutor långt. Mer information och standardvärden finns i klassen *Main*.

3 Systemdesign

Här diskuterar jag min lösning i stora drag.

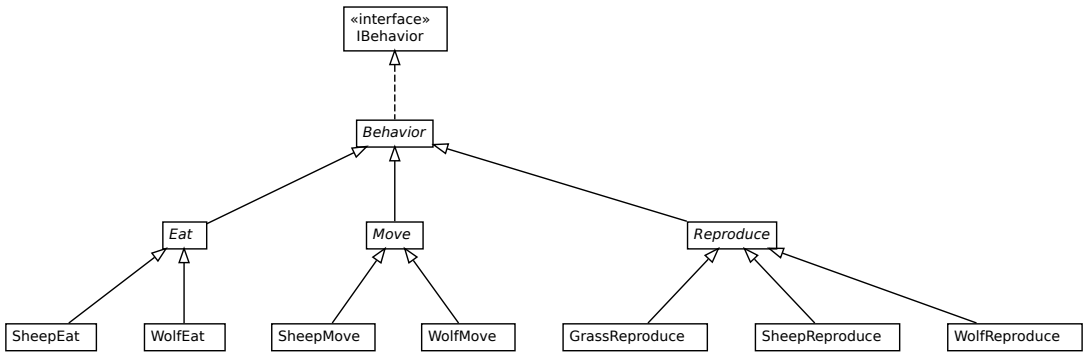
3.1 Entiteter

Initialt försökte jag modellera entiteter genom ett komplext arv, så som levande djur, stationära djur, förökande djur o.s.v. Men jag insåg snabbt att arvet blir onödigt krångligt och inflexibelt. I stället nöjde jag mig med ett mycket enkelt och ”platt” arv. Se figur:



Figur 1: Klassdiagram över entiteter.

För att möjliggöra varierande beteenden mellan olika entiteter skapade jag beteendeklasser, separerade från logiken i entititerna. Se figur:



Figur 2: Klassdiagram över entiteters beteenden.

Varje entitet har en samling beteenden som vid varje tick bestämmer hur entiteten agerar. Tack vare denna design är det enkelt att skapa fler beteenden, samt låta entiteter använda hur många eller hur få beteenden som helst, utan att påverka själva entitetsklasserna.

Man kan exempelvis skapa en entitet som äter och flyttar, en annan som flyttar och förökar sig, och en tredje som äter och förökar sig, utan att komplicera entitetsarvet. Varför inte skapa en entitet som inte gör något alls? Eller allt möjligt?

3.2 Ramverk

Det givna ramverket har använts och är i stort sett intakt. De största förändringarna jag gjorde var att från *Pasture* flytta skapandet av de första entiteterna till en ny klass, *Seeder*, och flytta logik som hanterar rutnätet till en ny klass, *Grid*. På så sätt blev *Pasture* mer överskådlig.

Några saker fick byta namn, varav det viktigaste är *Entity* som fick byta till *IEntity*.

4 Klasser

Här beskrivs de viktigaste klasserna i systemet.

4.1 Entiteter

Alla entitetsklasser är samlade i paketet ”pasture.entity”. Entiteter måste implementera interfacet *IEntity* som kräver metoder som är gemensamma för alla entiteter. Dessa metoder inkluderar ”getters” för t.ex. hagen och entitetens position i den, metoder för

att ta bort entiteten ur hagen, samt metoder för att undersöka om entiteten får stå på vissa positioner i hagen.

Den viktigaste metoden som interfacet kräver är *tick()*, vilken anropas av simuleringsmotorn vid varje tidssteg. Metoden är tänkt att låta entiteten agera ”ett snäpp”, t.ex. genom att förflytta sig ett steg och försöka äta det som finns på den nya positionen. Exakt vad som skall hända vid varje tick bestäms av entitetens beteenden.

För enkelhetens skull finns en abstrakt basklass *Entity* som implementerar det mesta av interfacet. Konkreta klasser behöver därmed bara ärva av *Entity*, skapa de beteenden som önskas och definiera metoden *maySharePositionWith()* som bestämmer om entiteten får stå på samma position som given annan entitet.

4.2 Beteenden

Alla beteendeklasser är samlade i paketet ”pasture.behavior”. Beteenden måste implementera interfacet *IBehavior* och därmed metoden *act()*. Vid varje tick anropar varje entitet sina beteendens *act()*-metoder (förutsatt att entiteten fortfarande lever).

Den abstrakta basklassen *Behavior* tillhandahåller en timer som utlöses efter ett visst intervall. Äta gör man så fort tillfälle ges, alltså vid varje tick, medan förflyttning och förökning sker med jämna mellanrum. Detta konfigureras i subklasserna.

Eftersom olika varelser exempelvis förflyttar sig på likartade sätt finns ytterligare abstrakta klasser som ärver av *Behavior*. Dessa klasser är *Eat*, *Move* och *Reproduce*.

Eat har ytterligare en timer, nämligen för att hålla koll på om entiteten svälter. Den implementerar även själva ätandet, men låter informationen om vad som faktiskt kan ätas definieras av subklasser genom metoden *mayEat()*. Vargar kan exempelvis äta får, medan får kan äta grästuvor.

Move är den största beteendeklassen då den implementerar omfattande intelligens för att optimera nästa förflyttning. Subklasser till *Move* behöver endast definiera metoden *evaluatePosition()* som viktar en given position. Vargar föredrar positioner nära får, medan får föredrar positioner så långt bort ifrån vargar som möjligt men annars nära grästuvor.

Reproduce implementerar ett enkelt beteende för att föröka sig: finns det någon angränsande position som är ledig så skapas en ny entitet där. Vilken entitet som skapas bestäms av metoden *spawnChild()* som definieras i en subklass.

På detta vis blir de konkreta beteendeklasserna, t.ex. *SheepMove* och *WolfReproduce*, oerhört korta och koncisa.

4.3 Ramverk

Alla ramverksklasser är samlade i paketet ”pasture”. Här går jag igenom de viktigaste förändringarna och tilläggen jag gjort.

Config är en abstrakt klass med statiska metoder som tillhandahåller ett enkelt sätt att hantera parametrar som behövs på olika ställen i systemet.

Main startar programmet och skapar alla standardinställningar. Om argument har givits till programmet kommer dessa argument att skriva över standardinställningarna.

Pasture är själva hagen. Den hanterar alla entiteter som skall finnas i hagen, samt vidarebefordrar metodanrop som har att göra med själva rutnätet till klassen *Grid*.

Grid tar hand om rutnätet i hagen och håller reda på vilka entiteter som står på vilka positioner.

GUI tar hand om alla grafiska gränssnittskomponenter och ser till att hagen visas upp på skärmen.

Engine driver simuleringen och ber alla entiteter att agera vid jämna mellanrum.

5 Artificiell intelligens

Jag har följt uppgiftens tips på hur man kan implementera AI, nämligen att varje position inom en entitets synhåll får en vikt, där hög vikt är eftersträvansvärt. Vikten baseras på vad entiteten föredrar och aktar sig för, och vissa önskemål påverkar vikten mer än andra.

En entitet bestämmer sin nästa förflyttning genom följande procedur:

1. Hitta alla positioner inom synhåll,
2. vikta positionerna enligt entitetens viktningsfunktion,
3. behåll endast positioner som angränsar till entiteten,
4. behåll endast positioner som har högst vikt,
5. slumpmässigt välj en av de återstående positionerna.

5.1 Får

Får skall bete sig på följande vis, i sjunkande prioritetsordning:

1. Om varg inom synhåll, fly,
2. annars, om grästuva inom synhåll, närma sig,

3. annars, gå i ungefär samma riktning som tidigare,

4. annars, gå i slumpmässig riktning.

Viktfunktionen definieras av formeln:

$$(100 \cdot wolf) + (-1 \cdot food) + (0.1 \cdot prev)$$

Varje term innehåller en variabel, nämligen ett avstånd: $wolf$ är avståndet till närmaste varg inom synhåll, $food$ är avståndet till närmaste grästuva inom synhåll, och $prev$ är avståndet till entitetens föregående position.

Varje term multipliceras med en konstant faktor, där faktorns storlek bestämmer hur viktigt avståndet anses vara, och faktorns tecken om avståndet skall maximeras eller minimeras.

Sammanfattningsvis kommer denna formel att först och främst maximera avstånd till varg (fly från varg), sedan minimera avstånd till gräs (närma sig gräs), och till sist maximera avstånd till föregående position (bibehålla riktning).

5.2 Varg

Vargar skall bete sig på följande vis, i sjunkande prioritetsordning:

1. Om får inom synhåll, närma sig,

2. annars, gå i ungefär samma riktning som tidigare,

3. annars, gå i slumpmässig riktning.

Vargars beteende fungerar som för får, fast med en annan viktningsformel:

$$(-1 \cdot food) + (0.1 \cdot prev)$$

Formeln kommer först och främst att minimera avstånd till får (närma sig får), och till sist maximera avstånd till föregående position (bibehålla riktning).