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
```

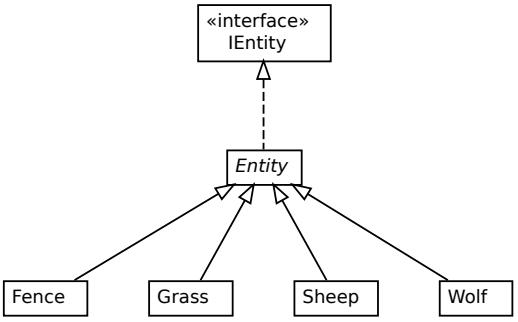
Programmet går att konfigurera genom att ge parvisa argument:

```
$ 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*.

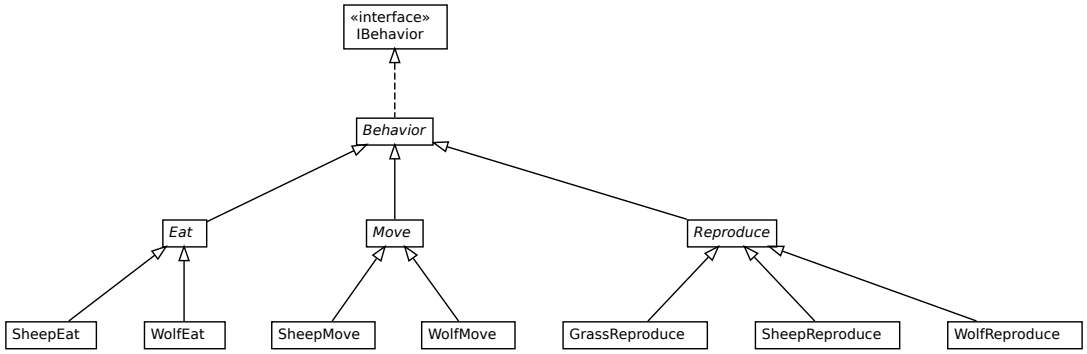
3 Systemdesign

Lösningen jag till slut valde förlitar sig mer på komposition än arv. 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 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 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. Eller varför inte en entitet som inte gör något alls? Eller allt möjligt?

4 Klasser

Här beskrivs de viktigaste klasserna i systemet.

4.1 Spelbordet

Spelbordet implementeras i klassen *Board*. Den skapar alla kort som skall visas, samt har en metod för att underrättas om något kort har ändrats.

Publika metoder:

void notifyChangedCard(Card *card*) Placerar kortet *card* överst, så att det inte skyms av några andra kort.

4.2 Kort

Kort implementeras i klassen *Card* som ärver av *JPanel*. Den skapar, utifrån två sökvägar, två bilder som används som kortets fram- och baksida. Varje kort skapar en egen instans

av en klass som prenumererar på mushändelser. Kortet får även en referens tillbaka till bordet som skapar det.

Publika metoder:

void moveByDelta(Integer *deltaX*, Integer *deltaY*) Flyttar kortet *deltaX* pixlar i x-led och *deltaY* pixlar i y-led. Meddelar sedan bordet att det har uppdaterats.

void flip() Vänder på kortet samt berättar för bordet att det har uppdaterats.

Kort överskuggar *JPanel*-metoden *paintComponent* så att när kortet skall ritas upp på skärmen så ritas bilden som associerats med kortets synliga sida.

4.3 Muspekaren

Varje kort har en egen muspekarlyssnare som implementeras i klassen *CardMouseListener* som implementerar två interface för att hantera mushändelser: *MouseListener* och *MouseMotionListener*. Lyssnaren får meddelanden om musen försöker manipulera kortet.

Publika metoder av intresse:

void mouseClicked(MouseEvent *mouse*) Anropas när musen har klickat (tryckts och släppts utan rörelse däremellan) på kortet. Kortet ombeds vända på sig.

void mousePressed(MouseEvent *mouse*) Anropas när musen trycks ned på ett kort. Sparar musens nuvarande tillstånd så att avstånd kan beräknas när kortet börjar flyttas.

void mouseDragged(MouseEvent *mouse*) Anropas när en musknapp redan är nedtryckt och muspekaren rör sig. Avståndet från föregående position beräknas och kortet ombeds flytta på sig lämplig sträcka.

5 Programflöde

Programmet börjar i *main*-metoden som återfinns i klassen *Main*. Den skapar ett fönster och placerar ett spelbord inuti detta. Fönstret ritas sedan upp.

När bordet skapas konstruerar det i sin tur ett antal kort. Varje kort skapar sina bildikoner och sätter upp en egen lyssnare för mushändelser. På så sätt kan Swing se till att mushändelser rapporteras till rätt kort, d.v.s. kortet som ligger överst under muspekaren.

När en mushändelse fångas upp av ett kort uppdaterar kortet sitt eget tillstånd och avslutar med att meddela bordet att kortet har uppdaterats. Bordet ser då till att kortet hamnar överst.

Eftersom bordet är en *JLayeredPane* är det mycket lätt att flytta en komponent så att det ritas ut överst, nämligen genom anropa metoden *moveToFront*. Vill man i stället implementera bordet som en *JPanel* går det lika bra, men då måste man lösa problemet på ett annat sätt, exempelvis genom att plocka bort kortet från den egna ”containern”, lägga tillbaka det igen, men då överst, och sedan låta kortet rita om sig själv.