Simuleringer, abstrakte klasser og interfaces

BGPP 2014

Oversigt

- Eksempel: Computersimulering af rovdyr-byttedyr verden
- Tekniske begreber:
  - Abstrakte klasser
  - Interfaces

Computersimuleringer

- Anvendes ofte når eksperimentet er for dyrt eller umuligt at udføre

- Eksempler
  - Vejrsimulationer
  - Atomprøvesprængninger
  - Økonomiske simulationer
  - Trafiksimuleringer
  - Biologi og medicin
  - Ingeniørprojekter

Modeller

- Simuleringer bygger altid på modeller
- Modeller baseres på abstraktioner og forsimplinger
- Overvejelser: Hvad er rimelige forsimplinger?
- Verden er for kompliceret til computeren
- Resultat af simulation afspejler kun virkeligheden i den grad modellen gør det

Foxes-and-rabbits projekt

- Simulation af kanin- og rævepopulationer
- Rovdyr-byttedyr simulation
- Simulation kan f.eks. svare på spørgsmål som
  - Hvorledes afhænger bæredygtighed af størrelsen af området?
  - Hvad er forholdet mellem antallet af ræve og antallet af kaniner i et bæredygtigt (stabilt) system?
  - Hvad sker der hvis en sygdom udrydder 90% af kaninerne?

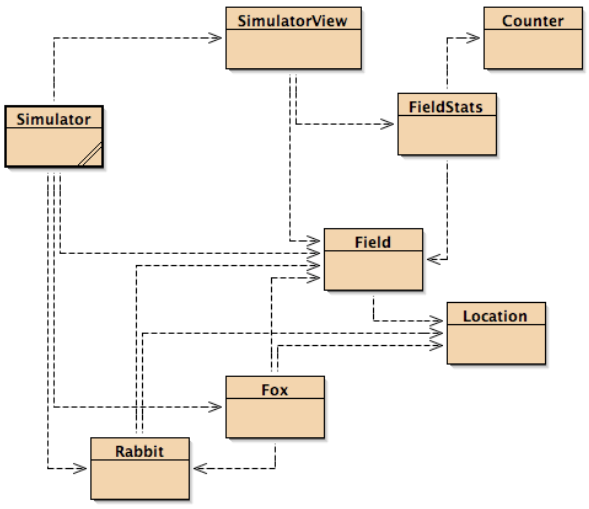
Simulationen

- Ræve og kaniner befinder sig på felter i rektangulært lukket landområde
- Hvert felt indeholder max ét dyr
- Simulationen foregår i skridt
- I hvert skridt formerer dyrene sig og nogle kaniner bliver spist af rævene
- Kaniner og ræve kan også dø af aldring eller overbefolkning

Designvalg

- Valgte forsimplinger:
  - Ingen forskel på hanner og hunner
  - Ignorer forskelle i terræn
  - Ingen andre dyr
  - etc
- Sådanne valg træffes *før* der skrives kode!
- Evt. i samarbejde med biolog

Klassediagram



Klassen Rabbit

Felter

```
// The age at which a rabbit can start to breed.
private static final int BREEDING_AGE = 5;
// The age to which a rabbit can live.
private static final int MAX_AGE = 50;
// The likelihood of a rabbit breeding.
private static final double BREEDING_PROBABILITY = 0.15;
// The maximum number of births.
private static final int MAX_LITTER_SIZE = 5;
// A shared random number generator to control breeding.
private static final Random rand = new Random();

// Individual characteristics (instance fields).

// The rabbit's age.
private int age;
// Whether the rabbit is alive or not.
private boolean alive;
// The rabbit's position
private Location location;
```


Klassen Rabbit

Metoden *run*:

```
/**
 * This is what the rabbit does most of the time - it runs
 * around. Sometimes it will breed or die of old age.
 * @param updatedField The field to transfer to.
 * @param newRabbits A list to add newly born rabbits to.
 */
public void run(Field updatedField, List<Rabbit> newRabbits)
{
```

Klassen Fox

Instansfelter

```
// Individual characteristics (instance fields).

// The fox's age.
private int age;
// Whether the fox is alive or not.
private boolean alive;
// The fox's position
private Location location;
// The fox's food level, which is increased by eating rabbits.
private int foodLevel;
```


Klassen Fox

Statiske felter

```
// Characteristics shared by all foxes (static fields).

// The age at which a fox can start to breed.
private static final int BREEDING_AGE = 10;
// The age to which a fox can live.
private static final int MAX_AGE = 150;
// The likelihood of a fox breeding.
private static final double BREEDING_PROBABILITY = 0.09;
// The maximum number of births.
private static final int MAX_LITTER_SIZE = 3;
// The food value of a single rabbit. In effect, this is the
// number of steps a fox can go before it has to eat again.
private static final int RABBIT_FOOD_VALUE = 4;
// A shared random number generator to control breeding.
private static final Random rand = new Random();
```

Klassen `Fox`

Metoden `Hunt`

```
/**
 * This is what the fox does most of the time: it hunts for
 * rabbits. In the process, it might breed, die of hunger,
 * or die of old age.
 * @param currentField The field currently occupied.
 * @param updatedField The field to transfer to.
 * @param newFoxes A list to add newly born foxes to.
 */
public void hunt(Field currentField, Field updatedField, List<Fox> newFoxes)
{
```


Klassen `Simulator`

- Holder styr på kaniner, ræve og deres placering i felten
- Etablerer starttilstanden
- Simulerer hvert skridt ved at kalde kaniners run metode og ræves hunt metode
- Bemærk: `field` holder styr på aktuelle tilstand (før næste skridt) og `updatedField` bruges til at etablere et nyt landområde

Klassen `Simulator`

```
// Constants representing configuration information for the simulation.
// The default width for the grid.
private static final int DEFAULT_WIDTH = 50;
// The default depth of the grid.
private static final int DEFAULT_DEPTH = 50;
// The probability that a fox will be created in any given grid position.
private static final double FOX_CREATION_PROBABILITY = 0.02;
// The probability that a rabbit will be created in any given grid position.
private static final double RABBIT_CREATION_PROBABILITY = 0.08;

// Lists of animals in the field. Separate lists are kept for ease of iteration.
private List<Rabbit> rabbits;
private List<Fox> foxes;
// The current state of the field.
private Field field;
// A second field, used to build the next stage of the simulation.
private Field updatedField;
// The current step of the simulation.
private int step;
// A graphical view of the simulation.
private SimulatorView view;
```


Problemer med design

- Klasserne `Fox` og `Rabbit` ligner hinanden for meget uden at have fælles superklasse
- Kodeduplikering i `Simulator`

Kodeduplikering i `SimulateOneStep`

```
// Provide space for newborn rabbits.
List<Rabbit> newRabbits = new ArrayList<Rabbit>();
// Let all rabbits act.
for(Iterator<Rabbit> it = rabbits.iterator(); it.hasNext(); ) {
    Rabbit rabbit = it.next();
    rabbit.run(updatedField, newRabbits);
    if(!rabbit.isAlive()) {
        it.remove();
    }
}
// Add new born rabbits to the main list of rabbits.
rabbits.addAll(newRabbits);

// Provide space for newborn foxes.
List<Fox> newFoxes = new ArrayList<Fox>();
// Let all foxes act.
for(Iterator<Fox> it = foxes.iterator(); it.hasNext(); ) {
    Fox fox = it.next();
    fox.hunt(field, updatedField, newFoxes);
    if(!fox.isAlive()) {
        it.remove();
    }
}
// Add new born foxes to the main list of foxes.
foxes.addAll(newFoxes);
```


`Animal` superklasse

- Vi opretter en superklasse `Animal` af `Fox` og `Rabbit`
- Felter som `Fox` og `Rabbit` har tilfældes kan flyttes til `Animal`
- I `Simulator` erstat lister af ræve og kaniner med liste af dyr

SimulateOneStep

```
// let all animals act
for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    animal.act(field, updatedField, newAnimals);
    // Remove dead animals from the simulation.
    if(! animal.isAlive()) {
        it.remove();
    }
}
// add new born animals to the list of animals
animals.addAll(newAnimals);
```

- Nu også nemmere at tilføje flere dyr

Abstrakte metoder og klasser

- Kaldet `animal.act` giver kun mening hvis `Animal` har en `act` metode
- Problem: Det giver ingen mening at implementere en metode `act` i klassen `Animal`
  - Alle dyr er enten kaniner eller ræve
- Løsning: Erklær klassen `Animal` som abstrakt og metoden `act` som abstrakt

Abstrakte metoder

Abstrakte metoder er metoder der ikke er implementeret i klassen, men som kan implementeres i subklasser

Syntaks

```
public abstract class Animal
{

    ...

    /**
     * Make this animal act - that is: make it do whatever
     * it wants/needs to do.
     * @param currentField The field currently occupied.
     * @param updatedField The field to transfer to.
     * @param newAnimals A list to add newly born animals to.
     */
    abstract public void act(Field currentField,
                             Field updatedField, List<Animal> newAnimals);
```

Implementation af subklasser

```
public class Rabbit extends Animal
{
    ...

    public void act(Field currentField,
                    Field updatedField,
                    List<Animal> newAnimals)
    {
        ...
    }

    ...
}
```


Regler for abstrakte klasser

- Kun abstrakte klasser kan have abstrakte metoder
- Man kan ikke have objekter af en abstrakt klasse
- Almindelige (konkrete) subklasser skal implementere abstrakte metoder, ellers skal de også være abstrakte

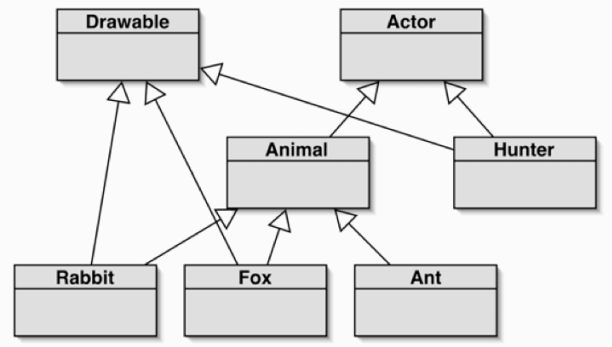
Fordele ved abstrakte klasser

- Selvom der ikke findes instanser af abstrakte klasser er de alligevel nyttige
  - Abstrakte klasser kan indeholde konkrete (ikke-abstrakte) metoder
  - De kan anvendes som typer
- I eksemplet brugte vi `Animal` som type til at undgå kodeduplikering

Overskrivning af felter

- I modsætning til metoder overskrives felter ikke i subklasser
- Hvis en subklasse har et felt med samme navn som en superklasse vil objekter af subklassen have to felter med samme navn
- Hvilket felt der refereres til i kald afgøres af den ***statiske*** type

Nedarvning fra flere klasser



- Man kan ikke nedarve fra flere klasser i Java!

Interfaces

- Interfaces er fuldstændigt abstrakte klasser, dvs. alle metoder er abstrakte
- Interfaces har ingen konstruktor
- Felter i interfaces skal være public, final og static
- Interfaces kan betragtes som primitive kontrakter

```
public interface Drawable
{
    public void draw(java.awt.Graphics g);
}
```

Interfaces

- Klasser kan *implementere* interfaces
- Implementation af interfaces i (konkrete) klasser skal implementere metoderne fra interfacet

```
public class Plant implements Drawable
{
    public void draw(java.awt.Graphics g);
    {
        ...
    }
}
```


Implementation af interfaces

Klasser kan implementere flere interfaces og samtidig nedarve fra superklasser

```
public class Rabbit extends Animal implements Drawable
{
    ...


public class Hunter implements Actor, Drawable
{
    ...
```

Interfaces i Java API

- En masse datastrukturer som f.eks. `List` og `Collection` er interfaces i Java API
- Her er også eksempler på interfaces der udvider interfaces
- Forskellige implementationer af samme interface har ofte forskellig effektivitet
- Brug af interfaces gør det nemt at skifte fra en implementation til en anden

Opsummering

- Abstrakte klasser er klasser hvor en eller flere metoder ikke er implementeret
- Der kan ikke erklæres objekter af abstrakte klasser
- Interfaces er fuldstændigt abstrakte klasser
- En klasse kan højst have en direkte superklasse, men kan implementere 0 eller flere interfaces