Assignment 2: an Ecological Simulation System

2AD512 / 2AD510
Object-oriented programming Fall 2004 (4.0 / 5.0 points)

Uppsala University
Department of Information Technology
Division of Computing Science

Version 0.1
Assistants: Jesper Wilhelmsson and Ruslan Fomkin

1 Introduction

In this exercise You are asked to develop an application that models and simulates an ecological system. The system is a simplified model of a piece of the reality. To simulate the ecological system you first have to develop a general simulation system that can run different simple models of some reality. Then you should implement the model of the ecological system and you will use the general simulation system to simulate it.

To be able to run the implemented simulation we provide a Graphical User Interface (GUI). You need to combine the given GUI with your implementation of the general simulation system and the ecological system. Once this is done you will be able to run and see the simulation.

To develop this application you have to make a design. Only once your design is finished (and accepted by the assistants) you should proceed to the implementation. This is important so we’ll say it once again: **DO NOT WRITE ANY SOURCE CODE UNTIL YOU HAVE AN ACCEPTED DESIGN!!** You present the design in UML notation. You implement the application in Java. It is required that the design and the implementation of the application are object-oriented.

The next section describes the application. The tasks of the design part and implementation part of the assignment are described in the last section.

2 The Application

2.1 The General Simulation System

The general simulation system is a component which provides all general functionality for development of any simple event-driven simulation system. To develop a specific event-driven simulation system you need just to extend the general simulation system with specific implementation of your simulation system. It is therefore very important to design the general simulation system with the ability to easily extend it.

The general simulation system is event-driven. This means that time only determines the order of event execution (activation). The actual (real) time between events is constant. To show that real time and simulation time is completely unrelated, we use the time unit $\text{glan}$ to measure time in
the simulation. One glan is the time between each note sang by a bird in any given forest.

The general simulation system contains three main components: the simulation engine, entities, and events. Entities are the objects of a simulation system. Events are actions that happens in the simulation system. Each event affect one or several entities of the simulation system. The simulation engine contains all events and entities of the particular simulation system. It runs the simulation by activating events.

2.1.1 The simulation engine

The simulation engine is the core of an event-driven simulation. It contains and manages entities and events. It activates events that will change the status of the simulation system, for example, a new entity might appear in the system, or an entity disappears from the system, or an entity changes its position.

The simulation engine is designed for driving event-driven simulations. It activates events in the order of their activation time. The event with the smallest activation time is activated first. The activated event can invoke some methods of an entity or several entities, or add or remove entities or events in the simulation system. The simulation engine activates the next event after a constant delay which is independent of difference between activation time of this event and the next event.

To be able to chose the event with the smallest activation time events should be ordered, (otherwise you need to search for the smallest event). To do so, each event have to implement the interface Comparable from the java.lang package\(^1\). It is mandatory to use already implemented ordering or search algorithms from the Java Collection Framework\(^2\)\(^3\). The chosen algorithm should use your implementation of the interface Comparable.

The simulation engine should remember the current time which is the time of the last activated event. It should not activate any event in the past (the activation time must always be in the future).

You should also provide possibility to see, or observe, how the simulation is going. To do so the simulation engine should be observable and should notify observers, for example the GUI, whenever the state in the simulation has changed.

The simulation should run in its own thread so that the GUI is able to interact with it. A new thread is started each time a simulation is started. Please note that all methods in java.lang.Thread used to pause a thread are deprecated and You are not supposed to do this.

The GUI, events and entities need to communicate with the simulation engine. We provide an interface that the simulation engine have to implement. The interface is called Simulation, see Figure 1, and could be downloaded from the course web page \(^4\).

Method description:

- **start()** - Starts the simulation. It activates events in the order of their activation time with a constant delay between activations. The simulation stops when there is no more events or if it was requested (see the methods **pause** and **reset**.

- **start(int maxEvents)** - Starts the simulation. Only a limited number of events will be activated. The number is given by the argument **maxEvents**. The simulation can also stop when there are no more events left or if it was requested.

- **pause()** - Temporarily stops the on-going simulation.

- **reset()** - Stops the simulation and reset it to initial (or random) state.

\(^1\)http://java.sun.com/j2se/1.4.2/docs/api/index.html
\(^2\)http://java.sun.com/j2se/1.4.2/docs/guide/collections/index.html
\(^3\)http://java.sun.com/developer/onlineTraining/collections/Collection.html
\(^4\)http://www.it.uu.se/edu/course/homepage/oop/ht04/assignments/
public interface Simulation
{
    public void start();
    public void start(int maxEvents);
    public void pause();
    public void reset();
    public void register(Entity en) throws RuntimeException,
            AlreadyRegisteredException;
    public void remove(Entity en) throws NoSuchEntityException;
    public void schedule(Event ev, double activationTime) throws
            AlreadyScheduledException,
            IllegalActivationTimeException;
    public void unSchedule(Event ev) throws NoSuchEventException;
    public Collection getEvents();
    public Collection getEntities();
    public double getTime();
    public int getSimulationDelay();
    public void setSimulationDelay(int milliSeconds);
    public void addObserver(Observer o);
}

Figure 1: Interface Simulation

• register(Entity en) - Registers the entity en in the simulation. This method invokes an
  initialization method of the entity (described later) and if an exception occurred in the
  entity method it passes up the exception to the calling object. If the entity is already in the
  simulation an exception AlreadyRegisteredException should be thrown.

• remove(Entity en) - Removes the entity en from the simulation. If the entity does not exist
  in the simulation an exception NoSuchEntityException should be thrown.

• schedule(Event ev, double ActivationTime) - Adds the event ev to the simulation. The
  event can be added correctly into the event collection since all events are ordered by their
  activation time. Note! Implementation of sorting or inserting in right place algorithms from
  the Java Collection Framework should be used. If the event is already in the simulation this
  method should throw an exception, AlreadyScheduledException. If the event is in the past
  this method should throw the exception IllegalActivationTimeException.

• unSchedule(Event ev) - Removes the event ev from the simulation. If the event does not
  exist in the simulation this method should throw an exception NoSuchEventException

• Collection getEvents() - Returns a collection of all events in the simulation.

• Collection getEntities() - Returns a collection of all entities in the simulation.

• double getTime() - Returns the current time of the simulation. It is equal to the activation
  time of the last activated event.

• int getSimulationDelay() - Returns the current simulation delay. The delay say how
  much time the simulation engine should wait until the next event is activated. During this
  wait nothing happens in the simulation

• setSimulationDelay(int milliSeconds) - Sets new simulation delay. It should throw an
  exception OutOfRangeException if the new delay is not within suitable range, for example,
  if the delay is negative.
• `addObserver(Observer o)` - Registers the observer `o`. All registered observers should be notified by the simulation engine when any change has happened in the simulation. The simulation engine itself is observable.

2.1.2 The entity

Each entity of any specific simulation system has some general properties or behaviors that are independent of the specific implementation. These properties are part of the general simulation system. You need to develop an abstract (general) entity which implements the general properties. This will include an implementation of a given interface that the simulation engine, events, and the GUI package can use to be able to communicate with the entities. The given interface is called `Entity`, see Figure 2, and could be downloaded from the course web page.

```java
public interface Entity {
    public void init(Simulation hostSystem) throws RuntimeException;
    public void remove();
    public SimulationEntityType getEntityType();
    public Point getPosition();
    public void setPosition(Point newPosition) throws IllegalPositionException;
}
```

Figure 2: Interface Entity

Method description:

• `init(Simulation hostSystem)` - Initializes the entity. This method is invoked by the simulation engine when it adds (registers) the entity in the simulation. The argument of the method is the simulation engine. The interface `Simulation` is described in Figure 1. We assume that the entity need to know the host simulation system. A specific implementation of this method can create initial events. This method should also make sure that the position in the entity is valid. If not it should throw an extension of `RuntimeException`.

• `remove()` - This method is invoked by the simulation engine when the entity is removed from the simulation. There is not much to implement in the method during implementation of the general system. In specific cases this method can, for example, remove or update all events that affect this entity.

• `SimulationEntityType getEntityType()` - Returns a description of a type of this entity. The interface `SimulationEntityType` is described in Figure 3. This method is invoked by external systems, for example, by a GUI package, and can also be invoked by another entity or by events.

• `Position getPosition()` - Returns the current position of this entity. We assumed that all entities exist in a 2-dimensional simulation. This method could be invoked by objects inside the simulation and by external systems, for example, by a GUI.

• `setPosition(Point newPosition)` - Set a new position for the entity. If the position is illegal, because, for example, it is occupied, this method should throw an exception `IllegalPositionException`.

The interface `SimulationEntityType` is empty, see Figure 3. It needs to be extended and implemented for a specific simulation.
public interface SimulationEntityType
{
}

Figure 3: Interface SimulationEntityType

2.1.3 The event

All changes in the simulation happen because of activation of events. Events affect entities, for example, some attribute of an entity is changed, or some entity is removed or added to the simulation.

Each event of any specific simulation system also has some general properties or behaviors that are independent of the specific implementation. These properties are part of the general simulation system. You need to develop an abstract (general) event which implements the general properties. This will include an implementation of a given interface that the simulation engine and entities can use to communicate with the entities. The given interface is called Event, see Figure 4, and can be downloaded from the course web page.

public interface Event extends Comparable
{
    public void activate();
    public double getActivationTime();
    public void setActivationTime(double activationTime);
}

Figure 4: Interface Event

Method description:

- `activate()` - Activates this event. This method usually invokes some methods of an entity or several entities, or some methods of the simulation engine. The simulation engine invokes this method when the simulation is running (was started). This method could only be implemented in a specific system. There is no implementation that can be done in the general simulation system.

- `double getActivationTime()` - Returns the activation time of this event.

- `setActivationTime(double activationTime)` - Sets new activation time of this event.

2.2 The Ecological System

The specific application which you should develop is an event driven simulation of the ecosystem in a pasture. There are animals and plants of different kind in this pasture. The pasture is surrounded by a fence, and there can also be fences scattered inside the pasture.

Compared to the real life several strong simplifications have been made for constructing the dynamic model of a pasture's ecological system which should be simulated in this assignment.

The number of species included in the model are few, but the behavior of each species in the model represents a collection of behaviors gathered from several species with similar behaviors in real life. The same is true for other objects in the pasture as well.

Each being and object included in the model represents a population of beings or objects of the same kind. That is why the model allows beings to reproduce itself without having another being
of the same kind nearby. To model populations instead of single beings is also more statistically correct and can give better prognosis from the simulations of the model.

2.2.1 The Pasture

The pasture is built around a two dimensional delimited area with a squared shape. We look at the pasture from above, and disregard any hills and slopes.

The squared area is divided into a grid with $W \times H$ squares, where $W$ and $H$ is the width and height (number of squares) of the grid. The position of each square is given with a coordinate pair $(x, y)$. $(0, 0)$ is in the bottom-left corner and $(W - 1, H - 1)$ is in the top-right corner. See figure 5.

![Figure 5: Coordinates in the pasture](image)

2.2.2 Objects and Beings in the Pasture

The Pasture contains 4 kinds of objects (entities): fences, grass tufts, sheep, and wolves. To be able to differentiate each kind of entity they need to implement the method `getEntityType`. The method should return an object of some class that implements an extension of the interface `SimulationEntityType` called `PastureEntityType`, see Figure 6.

```java
public interface PastureEntityType extends SimulationEntityType {
    public boolean isSolid();
    public boolean isPlant();
    public boolean isHerbivorous();
    public boolean isPredator();
}
```

![Figure 6: interface PastureEntityType](image)

Entities in the pasture

Fences are objects that occupy space. They are solid (see the interface `PastureEntityType`). No objects of any kind can be at a position which is occupied by a fence. Fence objects are used to
surround the pasture so that nothing can move out of the pasture. Fence objects could be placed inside the pasture as well. Animals can see through fences, but should preferably not be limited by this in their hunt for food or friends.

*Grass tufts* are living objects that cannot move but they can reproduce. They are plants (see the interface `PastureEntityType`). Grass tufts do not die from old age, but they can be eaten. Each grass tuft reproduces in a nearby square, see Figure 7, if the square is not occupied by another grass tuft or by a fence. The time between two reproductions of the same grass tuft is constant and is called *reproduction time*. A new grass tuft has a slightly different reproduction time compared to the parent grass tuft (mutation). Reproduction time of any grass tuft should be between minimum and maximum reproduction time. The limits should be set by you before the simulation is started.

![Figure 7: Squares nearby in the pasture and possible movements to them (in the general case)](image)

*Sheep* are living beings that can move, run, eat food, reproduce and die. They are herbivorous (see the interface `PastureEntityType`). Sheep sees other objects in the pasture within a limited range which is called *sight range*. The sight range is a number of squares. A sheep can only see objects that are within the sight range. A sheep usually moves but if it needs to flee from an enemy it runs. *Speed* is the time that it takes for a sheep to move to the next square. Therefore the speed of running is lower than the speed of moving. A sheep can move only to the nearby square see Figure 7 if it is not occupied by fence, another sheep, or a wolf.

Sheep eat grass tufts and the enemies of sheep are predators. When a sheep is to move, it do this with the following priorities:

1. It tries to flee from enemies by running in the opposite direction. (Given some sort of intelligence if there are several enemies within the sight range.)
2. If it see food, it moves towards the closest food.
3. If it is not already close to other sheep and it sees some it tries to move toward them (*herd behavior*).
4. If none of the above is true it moves in a random direction, probably close to the direction it was already moving in.

To eat food does not take time for sheep. If a sheep is in the same position as a grass tuft the grass tuft should be removed from the simulation. This means that it was eaten. A sheep cannot eat when it is running.
If a sheep have not eaten for a long time it dies from starvation. The time that it can live without food until it dies is called *starvation time*. A fleeing sheep is in panic, it will not stop to eat even if it is about to starve to death.

A sheep can reproduce only when it reaches a grown-up age. The time that is needed to grow up is called *grown time*. The time between two reproductions of the same sheep is constant and is called *reproduction time*. A new sheep will produce first child after it has lived the grown time plus the reproduction time. A new sheep has slightly different (mutated) reproduction time, grown time, move speed, run speed, sight range, and starvation time. All these parameters should be between minimum and maximum values for any sheep. The limits should be set by you before the simulation is started.

Wolves are living beings that can hunt for food, eat food, reproduce and die. They are predators (see the interface *PastureEntity*). Wolves do not have enemies. They see other objects in the pasture in a limited range which is called *sight range*. The sight range of wolves are bigger then the sight range of sheep. Wolves will hunt for sheep as often as possible. A wolf hunts one individual sheep until it kills the sheep or the sheep escapes the wolf’s sight range. If there is no sheep in sight range of a wolf, the wolf moves in random direction, probably close to the direction it already was moving in.

A wolf can only move to a nearby square see Figure 7 if it is not occupied by a fence or another wolf.

To eat sheep does not take time for wolves. If a wolf is in the same position as a sheep the sheep should be removed from the simulation. It means that it was eaten. If a wolf have not eaten for a long time it dies from starvation.

A wolf can only reproduce when it reaches a grown-up age. The time between two reproductions of the same wolf is constant. A new wolf has slightly different (mutated) reproduction time, grown time, move speed, sight range, and starvation time. All these parameters should be between minimum and maximum values for any wolf. The limits should be set by you before the simulation is started.

### 2.3 Connection to the given GUI

You also need to run the application that you are developing. The ecological simulation system should be represented so that a user can interact with it and see what is happening in it. To do so you need to implement the main method and connect a GUI to your system. We provide a GUI package, `pasture.gui`, that will present the simulation and interact with a user.

The main method should create the simulation containing the entities and events, and then create an object of the class, `pastureGUI`, from the package `pasture.gui`. You need to pass a `Simulation` and a Dimension (containing the size of the pasture) to the `pastureGUI` object as arguments to the constructor. After you created a `pastureGUI` object you should add it to your simulation as observer. Then You can start the GUI by invoking the method `start` of the `pastureGUI`. See the description of the API of the class `pastureGUI` that is available on the course page. The description was generated by using the Javadoc Tool.

### 3 The Tasks

#### 3.1 The Design Part

First you should make an object-oriented design of the application. The design should contain a description of all necessary classes and all relationships of different kinds between them. All classes and dependencies between them should be presented as UML class diagram(s). During the
design it is very important to get the full picture of the needed class hierarchy and of how the communication between them are done.

The standard classes and interface from the java library and relationships to them should also be presented in the diagram. You do not need to describe attributes or methods of standard classes. You do not need to present exceptions in the diagram.

3.1.1 Hand-In

You will present your design orally. You need to bring a report which contains the diagram and a front page where you state the name of the course, the assignment number (2, design part) and names of both members of the group. The diagram should be presented in the UML notation, based on UML version 1.3. The diagram could be presented on several sheets of paper and could be hand written or by some drawing tool. It is very important that the diagram is readable and understandable. You can separate presentation of relationships among the classes and interfaces and presentation of the class/interface attributes and methods.

Please, book time for the presentation of your design on the door of "your" assistant. Booking schedule will be available one week before the presentation time. The presentation time is published on the course page.

3.2 The Implementation Part

Only after your design of the application was accepted you can start to implement it. Your implementation should fully correspond to the design. You should implement all classes, including all exception classes. Be aware that the given GUI classes do not catch any exceptions.

The implementation of the general simulation system should be in its own package. The package name should be simulation. The implementation of the ecological system should be in another package that is called pasture.

All source code should be commented. The packages, classes, interfaces, methods, and attributes should be commented according to requirements of the Javadoc Tool. The code inside methods should be commented to explain it.

You should set the number of initial entities and their initial parameters in such a way that the simulation of the ecological system is stable.

3.2.1 Hand-In

Before hand-in the implementation should be tested on a Unix machine in the University so that it does not have odd behavior.

You should hand in the report containing links to the source code, class files, and the generated documentation of the implementation, and the (already approved) UML diagram describing the implemented design in the mail box of the corresponding assistant. Do not forget to state the name of the course, the assignment number (2, implementation part) and names of both members of the group in the report. You should hand-in it before the 10th of December 10 o’clock.

The corrected assignment should be picked up from the corresponding assistant during his office time, see the course page for more information.