Homework #2: Virtual World at FaceDuck

Due on Monday, October 16 at 10:30am (before the lecture)

The FaceDuck CEO has come up with a fun concept for the FaceDuck website: applications that allow FaceDuck users to interact in fun, virtual environments. Since this concept was discussed, the FaceDuck framework design committee has developed a simulation framework for simple 2-dimensional virtual worlds that evolve in real-time. Your manager promoted you to enhance a prototype virtual environment that your co-workers have been working on.

The prototype environment is designed to simulate nature. Grass will grow, and rabbits hop playfully. Of course, there will be some conflict; thus, there will be foxes constantly plotting to catch a rabbit for dinner.

Your goals for this assignment are:

- Effectively use objects and inheritance for encapsulation and code reuse
- Program implementations for existing interfaces
- Experience working with a medium-sized application
- Start thinking about code from a design point-of-view

Instructions

Luckily, your manager has already developed a virtual world environment. Your job is to build the objects that will fill the world. You will need to build rabbits and foxes, as well as the AI for these objects. Furthermore, you must create an animal of your choice, whose behavior you must define.

You need to implement the following classes:

- actors.Gnat, actors.FoxImpl, actors.RabbitImpl
- ai.FoxAI, ai.RabbitAI, ai.GnatAI,
- (and an animal and its AI of your imagination)

Documentation

The FaceDuck design team has provided the following documentation for you. Please read this carefully.

(1) Types of Objects in the World – Specifications and Definitions

The world contains the following object types, with each type having different properties and specifications described here.

Actor: An Actor is a participant in the world. Actors are objects which are able to act in the world, and they have a cool down property and a view range.

An Actor's view range is defined as the number of spaces away from the actor that is visible. This means that if an Actor is currently at a location (i, j) in our world and has a view range of k (k is a nonnegative integer), then the Actor can see all spaces in the square defined by the corners $(i - k, j - k)$, $(i + k, j - k)$, $(i + k, j + k)$, $(i - k, j + k)$, provided that all of the corners of that squares is within the boundary of the world (if not, then set of spaces that the Actor can see is the largest subset of spaces of that square such that the subset is within the boundary of the world). Note that we define (0, 0) to be the top left corner of the world.

Time in the virtual world progresses in discrete time steps and actors also have a cool down property, an integer that determines how often an actor acts. The actor's cooldown determines how many steps must pass before the actor can act again. For example, if an actor has a cooldown of 0 and it acts on step 0, it may act again on step 1. If an actor had a cooldown of 1 and it acts on step 0, however, then it may not act again until step 2. The world implements these cooldown properties for you; you only need to understand them so you know how long an actor must wait before acting again, to help you build a better AI.

Animal: An Animal is an Actor that has an added property of energy, and is able to eat, breed, and move. Energy is used to act in the world; every time an Animal acts, it loses one unit of energy. More specifically, any time an Animal uses the AI to select an action (please read the later section on AIs), the Animal loses one unit of energy, even if the AI returns a nullresponse. If an Animal's energy reaches zero, the Animaldies and it removes itself from the world. Each kind of Animal will have a different initial level of energy, and a different maximum energy. The latter is the maximum energy permitted at the end of a step, after considering the energy used to ask the AI what to do, and considering any energy gained by eating.

Edible: An Edible object is an object which an Animal can eat. Edible objects have an energy value that specify how much energy the eater gains when the object is eaten. For example, suppose grass has an energy value of 5. Each time a Rabbit eats grass it would then gain 5 units of energy. An Edible's energy value should be constant. Note that the energy value of an Edible is different from (though related to) the current energy and maximum energy of an Animal.

(2) Specific Objects in the World

You will create some of the following classes that implement the interfaces described above. Many of the following have been created for you.

Grass: Grass is an Edible Actor. The Grass class has been implemented for you.

Gardener: A Gardener is a simple Actor which stays in place and adds Grass to the world at random locations. The Gardener class has been implemented for you.

Gnat: A Gnat is an Actor. Gnats do not lose energy when they act and therefore live forever. When Gnat acts, it randomly moves to a new (empty) location that is adjacent to its current location.

Rabbit: A Rabbit is an Animalwhich is also Edible. Furthermore, Rabbitsneed to breed as often as possible and only eat Grass. You must write your own RabbitImpl class which implements the Rabbitinterface.

Fox: A Fox is an Animal which eats Rabbits. Foxs also breeds at a lower rate than Rabbits. You must write your own FoxImpl class which implements the Fox interface.

Your Third Creature: The FaceDuck team is giving you full creative license to design and implement a third creature of your choice. Your creature does not need to be an animal but it should be interesting–more interesting than grass! It should be an actor that has an AI, like the Fox and the Rabbit. Feel free to have fun with this one; for example, you could implement an alien creature that randomly teleports around the world.

We have also provided some constants for you which we think will make a stable world. In our opinion, reasonably smart Rabbits and Foxes inhabiting this world will allow it to run indefinitely without either species going extinct. You may change the existing constant values to make your world more stable.

(3) Commands and Behavior – Communication

The AI uses Command in order to tell Actors what to do. Animals have three types of Commands: BreedCommand, EatCommand, and MoveCommand, each of which call the Animal's breed, eat, and move methods respectively:

Breeding: When an Animal breeds, it makes a copy of itself on an adjacent tile. The Animal can only breed when it has enough energy to do so. Breeding occurs alone; there is no mating between multiple Animal. When it breeds, an Animal's energy is reduced to 50% of its former energy (rounded down), and the newly-bred Animal also starts at 50% of its parent's energy. Finally, newly bred Animal must be placed in an empty location that is adjacent to the parent.

Eating: An Animal is only able to eat something that is adjacent to it. When eating an item, the Animal removes the item from the world and remains in place. The eaten item must be Edible, and the Animalgains its energy value.

Moving: An Animal can only move one space at a time and movement is restricted to only adjacent spaces. An Animal must move only to empty spaces.

Dying: Dying is also something you must implement. An Animal dies when its energy reaches 0 and you must remove the Animal from the world when it reaches that energy level. No dead Animals should be left rotting in the world and an Actor is expected to remove itself when it dies; it should remove itself if, after its last action, it has 0 energy. This means that if an Animal had 1 energy and used that last energy to eat something (and successfully gained energy from the Edible), the Animal would not die because it would have more than 0 energy at the end of the action.

(4) The AI – Decision Making

How you implement your AI determines how Rabbits and Foxs behave in the world.

We describe how the AI works using the Gnat code as an example. When a Gnat is instan- tiated, it stores a GnatAI object in a data field. When a Gnat acts, its act method returns a Command object (or null), with which the AI will use to execute the gnat's next action.

Your AI implementations must implement the AI interface and have a zero-argument constructor (or no constructor, in which case Java implicitly defines a zero-argument constructor for you). Our code that evaluates your submission requires that all AIs have zero-argument constructors.
The AI will determine what command to provide based on the Actor and the state of the world; the visible portion of the world depends on the Actor. For example, if a Rabbit has a view range of 1 and is looking for Grass, it will only be able to see locations that are within 1 adjacent square (including diagonals). The AI should be able to see all these locations and then determine which Commandit should create depending on the situation.

If the AI cannot determine a proper command to execute, it may return a null value, which commands the Actor to do nothing. All actors lose energy each time act is called, even if the actor chooses to do nothing.

(5) The World–Putting it all together

The World is an $n \times m$ grid of Actors, with the top left corner being (0, 0). Each space in the grid can hold a single object. Empty spaces on the grid are represented by a null value and each coordinate in the World is represented by a Location object. Location objects contain some helpful methods you should be aware of. Furthermore, the World provides the notion of direction that will aide interaction with the Location object.

The World has several methods. Given a World object, anything is able to add, remove, search, and view objects in the world. Furthermore, the World provides a step function that you will not use personally (other than for debugging) but should understand how it works. Each step forces every Actor in the Worldto act once, provided their cooldown allows them to act.

We have also provided a GUI you may use to visualize the world and see your Animal. The GUI has a simple interface containing two buttons: a "Step" to execute a single step; and a "Start/Stop" toggle button to run indefinitely until the toggle button is pressed again. For completing the homework, it is not necessary to understand the implementation of the GUI. You may run the code by running the main method in the WorldUIclass.

(6) Customizing the World

We have provided an initial version of a class named WorldLoader. This class defines how the world should be set up, adding the initial elements into the world such as the starting number of rabbits, foxes, grass, initial energy of creatures, etc. You get to decide how many of each object are added. Choices here are not extremely important; it's just a way for you to see test your AIs in different situations.

For this homework, we understand that testing code in this virtual environment is very difficult, and we rather you devote your time to writing good AIs and understanding inheritance. You should try to run your implementations in the GUI, you can experiment with different values for the constants, and you write can write test code. However, **we do not have any testing-related requirements**. While we encourage you to write test code, it is fine not to provide any test code in your submission.

Evaluation

To earn full credit, you must do the following:

- Design your Rabbit and Fox to reuse as much code as possible. Avoid copying-and-pasting code; instead, design your code with useful abstractions and class hierarchies.
- Your AIs must implement the provided AI interface and must be independent of the rest of your implementation. The AI may rely on the Fox and Rabbit interfaces we provide with the assignment, but they should not depend on your specific implementation of those interfaces.
- You must name your Fox AI class FoxAI and your Rabbit AI class RabbitAI, including the exact capitalization given here. Both classes must either have no constructor or a public constructor without parameters. Your AI implementations must be in the faceduck.ai package.
- You must name your Fox class FoxImpl and your Rabbit class RabbitImpl. They must be in the faceduck.actors package.
- In general, adhere to the code organization. You may add classes, abstract classes, and interfaces that you desire to the code base. Place any new files in the appropriate package. For example, interfaces go in interfaces, FoxImpl goes in actors, etc.
- You must not delete, add, or modify any files in the skeleton directories.
- Use the most restrictive access level that makes sense for each of your fields and methods (i.e. use private unless you have a good reason not to). Instead of manipulating class fields directly, make them private and implement getter and setter methods to manipulate them from outside of the class.
- Provide reasonable documentation. Document at least each public class and public method with a short javadoc comment.
- Do not use magic numbers in your code. Magic numbers are unnamed constant values that do not have a well-defined meaning. Instead of using magic numbers, define static final constant variables with good names.
- Make sure your code is readable. Use proper indentation and whitespace, abide by standard Java naming conventions, and add additional comments as neces- sary to document your code. Hint: using **Ctrl + Shift + F** to auto-format your code!
- Most importantly, write a report of up to 3 pages (in Korean) that describes your design and implementation. Make sure the report explains all the important aspects that we discuss above. Your report must describe what is implemented and also what is not implemented – if your implementation is different from what you describe in the report, we will deduct up to 50% from your grade.
- Your code should include an ant build file (build.xml); when ant is run it should create a jar file (vw.jar) in the project root directory.

Additional comments:

- The tasks may be underspecified. In case of doubt use your judgment. If you want to communicate your assumptions, use comments in the source code.
- Do not use the java.lang.Class class or the java.lang.reflect package. You do not need – and should not use – those advanced techniques for this homework.

We focus on the followings for the evaluation:

- Correct working Gnat, Rabbit and Fox implementations with working AIs.
- Completion of your third animal implementation.
- Good design, including well designed class hierarchies and avoiding code replication.
- Documentation and style.
- Report!

For the submission, please submit it to your gitlab repository under project name "virtualworld". For the report, you should submit it at the class.

Have fun!