4CCS1PPA Programming Practice and Applications

Coursework 3: Predator/Prey Simulation

Students:

Alexandru Matei K20054925

Ejaz Karim K20059213

Our simulation describes the behaviour of 5 acting species: cats, snakes, toads, rats and wild chickens, as they interact in the context of the woods nearby a lake. They all act in a similar manner: they move, they eat, they mate when possible, and they spread disease when sick. All these different species behave based on different, specific attributes and there are established predator-prey relationships between them. The two predators: Cats and Snakes, will share a common prey of Rats, meaning the two compete for resources. The prey Wild Chickens will have the sole predator of Cats, and the prey Toad will have the sole predator of Snakes. Totaling to two predators and three preys in the simulation. All prey type species subsist on plants: Toads and Wild Chickens eat Fern, and Rats eat Grass. Plant species are not visually represented in the simulation, but they do exist in the field and they can only be eaten from adjacent cells.

Animals are only able to mate when they meet (are adjacent to) an individual of the same species and opposite sex. Mating is not certain to result in offspring. The number of offspring, the probability of giving birth and the time that animals must wait between mating varies for different species. These are examples of some attributes determined for every animal belonging to a certain species. Other such attributes determine how long animals from each of these species may live for (for example, snakes and cats may live for up to 170 simulation steps, whereas rats have a maximum age of 50 steps), how much sustenance their food offers (whether it is another animal or a plant), what the minimum age for breeding is and how easily they spread disease.

The simulation keeps track of the time of day, as different species act at different times. Currently, the daytime and the nighttime both last 5 steps each. Toads and cats are nocturnal creatures, as such they move, eat and mate during the night. Snakes, rats and wild chickens act during the day.

All animals get sick with the same illness, which is transmissible between any two animals, however, the degree to which it affects different species varies. Still, sickness makes an animal get less value out of food (each animal/plant eaten guarantees survival for less time than it did before), have a lower life expectancy, and it decreases the chance of breeding successfully along with the number of offspring the animals may produce. Animals spread sickness between them with a given chance that depends on their species, for example toads are the most likely to spread disease, whereas cats are the least likely. Each step, for each animal there is a small chance it may get sick (as animals get sick randomly in real life as well). The simulation tracks how much of the overall animal population gets sick, and if the sick population passes a certain threshold, immunization takes place, meaning that until a lower percentage of sick animals is achieved, there is a high chance that some random animals will be cured at each step of the simulation. When an animal is not active e.g., not currently moving/eating/mating it will still spread disease if it is sick.

Plants (ferns and grass) are very simple organisms, and the simulation only keeps track of their age. They do not act; they only grow older and eventually die of old age or get eaten.

Every 25 steps new plants emerge, to simulate the time it took to sprout from the seeds of the last generation of plants. Grass and ferns are evenly distributed throughout the field, as there is an equal chance for either of the species coming to existence in any cell.

Color designations for visualisation: Cat - cyan, Snake – gray, Toad – magenta, Wild Chicken – green, Rat – orange. The darker shade versions of the colours identify females of the species.

Base tasks:

- Your simulation should have at least five different kinds of acting species.

For this task we decided to boil down species as much as possible, and make sure that the subclasses of Animal that will define these species are as specific as possible. We believe the only differences between the species are the "when?" (when do they die of old age, when do they mate), "who?" (who is the species of each animal's partner and prey) and "what?" (what they eat) of their various actions, not the actions in themselves. In order to properly model this and to increase the maintainability and extendibility of our code we moved the action defining methods to the "Animal" superclass and made them call upon the subclasses in order to receive the answers to the aforementioned questions. Superclasses don't really have access to their subclass's methods, so we used a trick: all accessor methods in the species subclasses had to first be defined as abstract methods in Animal. We consider this to be the best of both worlds. The actions are general, the behaviour methods in Animal don't need to refer to any specific species, but the values used to determine the results of these actions and the objects that are manipulated through them are consistent with the acting animal's species. (because each animal will "use" their species' variant of the implementation of the accessor/checker methods that regulate their actions.)

Thus, for all animals, the same findFood() method will execute, but because it calls upon checkFoodType(), which is individually implemented for each species, each animal will only eat their respective food, without the need of it being explicitly specified. Similarly, the giveBirth() method is common for all species, as it is inherited from Animal, but the method that creates the objects representing the offspring is implemented within the subclasses, but we can still call it from Animal, as it is defined as an abstract method. Thus, we can get animal objects of the same subtype without ever naming that specific type. This way of implementing species is highly flexible and extendable.

- At least two predators should compete for the same food source.

Snakes and cats both eat rats. In order to implement this, inside the checkFoodType() methods in each subclass respectively, the passed object is declared as a valid food source in the case that it is either Rat or the unique respective prey type specific to each of the predators. The species of the animal passed to this method is checked using instanceof.

- Some or all species should distinguish male and female individuals.

The Animal class has a String type instance field named “sex” which contains either the literal “female” or “male”, to establish the sex of each individual animal. When any animal tries to mate, it will search through its neighbouring cells until it finds a valid mate. As far as implementation goes, the animal that is trying to mate will pass objects that it finds nearby to a checkMate() method that establishes whether that passed object represents an animal of the same species and the opposite sex, and if it is old enough and its mating cooldown is down to 0. If all is as it should be, the initiating animal will take on the job of creating the offspring, then both animals will have their cooldown set to the species-specific value. The cooldown is needed both as a means of emulating real-life mating conditions and also so that the animals don’t mate twice for a single pair, or to prevent a single animal from mating multiple times inside the same step.

- You should keep track of the time of day.

The main Simulator object has a Boolean instance field that is set to true when it is day, and false when it is night. It is used by all animals to determine what their behaviour should be. Although this creates a small dependency from animals to the simulator object, we think it to be the better choice logically (the simulator holds all the general information, so it should know about the time of day too) and as dictated by responsibility-driven design principles (the simulator class is also tasked with updating the day-night flag every 5 steps). All species have constants named “MATE_DURING_DAY” and “MOVE_OR_EAT_DURING_DAY” which are accessed and checked against the current time of day every time an animal attempts to undertake an action. These constants may be set to true or false according to the species specifics. For example, Cats act during the night, so they are set to false.

Challenge tasks:

- Simulate plants

Because we wanted an animal and a plant to be able to be contained within the same space, but also thought that modifying the field class to accommodate a flexible number of objects in the same space would be too time-consuming, we opted to have plants in a different instance of the Field class from the animals. Because of this, herbivorous animals need to look for food in the plant field instead of the normal field, as such, there is a isHerbivore() method on which the field which gets searched in the findFood() method depends.

- Simulate disease

We decided for the disease to affect attributes relating to the health of the animal: max age, food value, breeding probability and litter size. New constants were made to hold the values of these attributes when the animal is sick. The initial, healthy attributes are now dubbed as "default". Then, we decided to use instance fields, to either store the default or diseased attributes depending on the current illness status of the animal so that they can be universally accessed and provide the right values according to the animal's condition at the current time, without having to check the health status from the animal class each time in order to choose between one accessor method (for a default value) or the other (for a sick value).

Code Limitations:

There is code duplication in the animal species subclasses. Unfortunately, this was needed because each class implements abstract methods from Animal so that the main action methods in Animal can be generalised. Maybe this can be solved by storing attribute values in Animal instance fields, populated on construction depending on a parameter that specifies the species. Although this would eliminate some code duplication, our implementation may be clearer than this option.

Because we do not know how to create new objects that are typed dynamically, without naming their type explicitly, a similar sequence of code is added for the creation of each species.

Also, the current layout of the populate() and createNewPlants() method, which puts the creations of different species one after another, linked with else if's, modifies the actual creation probabilities from the values we give them. Wild Chickens are marginally less likely to appear than their creation probability constant specifies just because they are the last species to be named. We could change the implementation of this by allocating a pair that defines a range of int values between 0 and 1000 to each species, while making sure any two value ranges do not overlap. Then for each creation of an organism, we create a new random number in the 0-1000 range, and whichever range it falls into determines which species should be created. Then, the larger the range for a species, the more likely it is to spawn.

Also, generalising the whole of plants and animals to organisms might have been premature, but it could be useful for later additions to the simulation (plant sickness, propagation, etc.).

In the simulation, the bottom panel shows the population of all the organisms. When a species goes extinct, instead of showing the population being 0, it will instead disappear from the panel. This is not ideal as it can be confusing to interpret. Also, the order of the species listed in the bottom panel is different each time the simulation is ran. This is because the simulation gets the species in the order they appear in the fields, which is random each time, so the listing order in the bottom panel will be random as well.