Design Manual

Predator Prey Simulation

Brooks Burt, Connor Vucovich, Liam Stott

Introduction

This program uses the general concepts of agile development and classes to create a simulation for a predator and prey populations as they move around a graphical simulation in the GUI. The environment that the animals move on is a canvas from the JavaFX API. The prey is the basic animal class and it has attributes that include its x and y coordinates on the canvas, as well as its overall energy and its speed that it is able to move around the environment with. The animal then has certain abilities in that it is able to move around in random directions, as well as bounce off of the walls of the environment, and once the animal comes into contact with food on the canvas, it then has the ability to reproduce and create another animal. Eating food also increases the energy of the animal. Once an animal's energy runs out, it “dies” and is removed from the simulation. Thus, the ability of the animal is very related to its position and movement on the canvas. The predator class is inherited from the animal class since it is an animal, and so it has all the same abilities such as moving, eating and reproducing. The only difference being the actual size of the predator on the GUI as well as it would look to eat one of the prey and not one of the pieces of food on the canvas.

In the actual implementation of both the predator and prey, the animals were created as runnable objects so that they could each individually run as a process in the form of a thread[1]. The thread for each animal when it is started just moves the animal to a new location using the move method for the animal, and then checks to see if it has come into contact with any of its specific type of food that it eats, and if it does it then performs the eat method for that specific animal. Lastly, the animal then checks to see if it has run out of energy, and if it has the thread then ends and the animal is removed from the simulation.

Before the simulation is run, all of the food, predators and prey have to be created. When the generate button is pressed, depending on the value of the slider bars, that number of predator and prey are each created as objects and stored in predator and prey array lists. The food is also created as a list of objects that store x and y coordinates of where they are, and this is also an array list. In order to be able for each thread to individually access and possibly modify these

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#:~:text=A%20thread%20is%20a%20thread,to%20threads%20with%20lower%20priority.

lists when either predators or prey are eaten, in the implementation of the eat methods, the lists an iterable object had to be created to loop through the lists in order to avoid the concurrent modification runtime exception[2].

There is also a World class that handles all of the interactions between these animals and the food, just like the real world would do. The World is where all of the lists are stored. When the start button is pressed, all of the threads of the predators and prey are started so that they can start moving and interacting around the world. The World also handles the implementation of the GUI. A separate thread is started within the world that loops through the lists of food, predators and prey and then updates their positions on the canvas so that the user can see the positions of these objects on the screen. The thread then sleeps for a specified amount of time to allow the animals to move again, before it clears the GUI and starts again. Thus the World acts as a constant refresh at constant intervals almost undetected to the human eye, which then results in the user being able to see the constant change in positions and the interactions between the animals and the food. Once all of the interactions have taken place and the animals have all died, the canvas will reset and the simulation will have ended.

User Stories:

- Our first user story is of a University Professor that wants to use the visual aspect of our program to demonstrate to her students the effects of natural selection on the population of an animal. Having a good user interface where people could clearly see what the food, predators and prey was, was then a very important aspect. It is also important for the professor to be able to show the interactions between the animals and so they would have to be clearly distinguishable. We then chose for the food to be green, the prey to be black and the prey to be red so it would be easy for someone to be able to tell the differences in the objects as well as what their purpose in the simulation is.

- Our second user story is about an average person who is just scrolling through the internet for simulations and wants easy and interesting programs to run. This is why having an interface that was simple and easy to use for anyone was important for our project. A regular user with limited Java or simulation knowledge would be too overwhelmed with a complicated program, and so we opted to put in a minimalistic GUI that would be easily navigated and had clear labels so that the user would understand exactly what its functionality is.

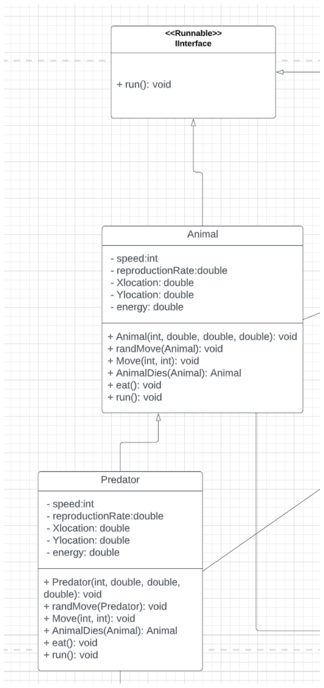
2
https://stackoverflow.com/questions/18448671/how-to-avoid-concurrentmodificationexception-while-removing-elements-from-arr

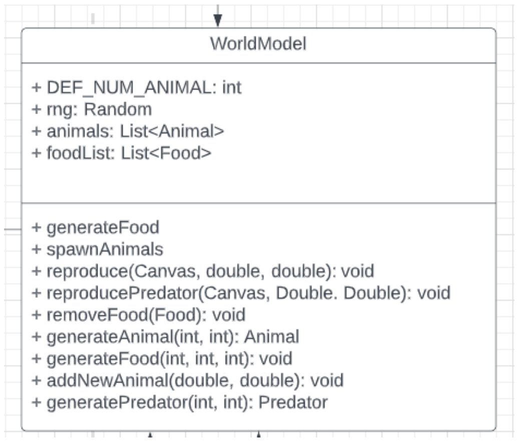
- Our next user story is from a graphical designer who is looking for simulations that are visually appealing so that she can use it as inspiration for her own work. This is why we have to create a simulation with graphics that are accurate so that people with experience in graphical design would be impressed with the end result. The World class is the result of this since the regular refresh rate of the canvas on the GUI creates a realistic simulation of the user being able to clearly see the objects moving around. Using the concepts of threads and independent processes would also be valuable to a graphic designer to implement in their own graphics for objects that might be moving around independently like our animals do.

- Our last user story is about a Biologist who wants to be able to use our simulation to simulate the interactions between species over a long period of time, in a very short timespan. He also wants to be able to manipulate various factors about the environment of the simulation. This is why we had to add in the features that allow the user to set the population of both the predator and prey at the very beginning of the simulation and thus someone like a biologist would be able to study these effects. Being able to restart the program and see the differences in results for population sizes was also important for the Biologist to then be able to note this. Likewise in order to make this realistic and something that a Biologist would want to use, we had to add in a certain amount of randomness into the simulation in the animals so that the results were never predictable and would more resemble the real world.

Object Oriented Design:

As far as our object oriented design, we first worked to subdivide all of our required tasks into various modules that tried to maintain a high degree of cohesion, while maintaining minimal coupling. We decided to first work to encapsulate the objects that would populate our actual simulation. We would create three main objects, a predator, a prey (animal), and food. Considering these were the lowest hanging fruit, we decided to code them first and adjust the UML diagrams as we necessitated further methods and variables. As each object would be placed on the Canvas in the GUI, we gave each object an X and Y coordinate value stored as a double. For the food object, this was sufficient as it required no further capabilities to be encapsulated within it; however, the animal required the ability to reproduce, eat, move, and die. We made the decision to introduce a variable to store the energy of the Animal, which was inherited in the predator class as well, as illustrated in the UML diagram to the left. Similarly, we added methods for movement,




WorldModel
+ DEF_NUM_ANIMAL: int
+ rng: Random
+ animals: List<Animal>
+ foodList: List<Food>
+ generateFood
+ spawnAnimals
+ reproduce(Canvas, double, double): void
+ reproducePredator(Canvas, Double. Double): void
+ removeFood(Food): void
+ generateAnimal(int, int): Animal
+ generateFood(int, int, int): void
+ addNewAnimal(double, double): void
+ generatePredator(int, int): Predator

eating, dying, and running the objects as a thread. We wanted our objects to be able to make adjustments to their location in a non-static matter. As illustrated in the diagram, the Animal object and therefore, the predator object both implement the Runnable interface for use in multithreading. These threads are created within the class of each object so that they can encapsulate the functionality of each animal type, while still being able to be used in the GUI. The only method absent from the Animal objects was the reproduction methods, as these were implemented in the WorldModel class. The reason for this lack of cohesion was simply due to the fact that reproduction as a method was affecting the list variables that were implemented in the WorldModel method, so implementing these methods in WorldModel, made that aspect of change much more efficient. Similarly, since it is the class that populates the GUI with all of its objects, the WorldModel class contains the generated predator and animal methods. As reproducing an animal is more akin to populating the GUI with a new animal, than it is adjusting an actual animal itself, we made the decision to encapsulate the functionality of reproduction into the WorldModel class, which itself is associated with the Animal and Predator objects themselves as it implements them into the Model itself. In general, we really worked to try and subdivide the objects within the simulation and the GUI of the simulation itself. The WorldModel serves as the bridge of all of the methods required to populate the Model with the required objects, while the GUI that implements those methods is handled in the Model and the Simulation classes.

Continuing with the implementation of our GUI goes, we created it utilizing the MVC design approach. As previously stated, our main model class was called WorldModel and dealt with things like generating animals and dealing with reproduction. Our view class was World which communicated with scenebuilder and dealt with displaying the actual visual representation of animal objects onto the screen. The setModel class in the view sets up all of the visual aspects to later be called by our controller. Finally, our controller class was Simulator which is the program that is actually called when the simulator is run and is in charge of setting up all of the correct classes to display the view.

Citations:

"A Computer Science Portal for Geeks." *GeeksforGeeks*, https://www.geeksforgeeks.org/.

"Where Developers Learn, Share, & Build Careers." *Stack Overflow*, https://stackoverflow.com/.

Java Threads Tutorial | Multithreading in Java Tutorial - YouTube. https://www.youtube.com/watch?v=TCd8QIS-2KI.

“Evolution Simulator: MinuteLabs.io.” *Labs.minutelabs.io*, https://labs.minutelabs.io/evolution-simulator/#/s/1/viewer?intro=1.

“Synchronizing Threads.” *Synchronizing Threads (Multithreaded Programming Guide)*, https://docs.oracle.com/cd/E19455-01/806-5257/6je9h0346/index.html.