KING'S COLLEGE LONDON

4CCS1PPA PROGRAMMING PRACTICE AND APPLICATIONS

Third "Simulation" Coursework (Feb 2025)

Project Name: The Simulation

Student Name:	Mehmet Kutay Bozkurt
Student ID:	23162628

Student Name:	Anas Ahmed
Student ID:	23171444

1 Introduction

This simulation project integrates multiple components to create a dynamic ecosystem that a researcher can use to study predator-prey-plant interactions, with the ability to add or remove any species or environmental factors easily by editing the given JSON file. Infact, almost every aspect of the simulation can be controlled in the JSON file. The simulation is designed to simulate without any grid-based restrictions, allowing entities to move freely in a continuous space. Specifically, the coordinates of the entities are stored as doubles from zero to the width and height of the field. The entities also have their own genetics, which are inherited from parent(s) and may mutate, all of which are initialised in the JSON file, where it is intended for all users of the program to modify it to play with the simulation parameters.

The simulation smoothly runs at 60 frames per second by utilising a `QuadTree` to store entities, which allows the entity searching and collision detection to be highly optimised. In each simulation "step," every entity is updated by calling its `update()` method, which handles movement, reproduction, and hunger. Entities make decisions based on other entities in their vicinity (that is, entities that are located inside their `sight` radius) and the current state of the environment (the weather and the time of day). Additionally, there is a method to handle overcrowding in the simulation, for limiting the growth of entities from being unnaturally rapid.

2 Directions for Use

Since the final code base is quite different from what was given as the base code for the project, it is necessary to provide a new set of instructions for running the simulation. Firstly, the `Engine` class is responsible for starting and setting up the simulation. A new `Engine` object can be created as

```
Engine engine
    = new Engine(WIDTH, HEIGHT, FPS);
```

where `WIDTH` and `HEIGHT` are the dimensions of the display window and `FPS` is the frame rate of the simulation. The simulation can then be started by calling the `start()` method on the `Engine` object.

Additionally, these have been already set up in the `Main` class, so the simulation can be run by simply running the `main()` method under it. Since the parameters of the simulation is read from the JSON file, it is necessary to have the `simulation_data.json` file in the root directory of the project (the place where the `Main.java` file can be found), but this is already provided in the Jar file in the submission. Finally, the simulation uses an external package called Gson to read the JSON file, and which is why the package is bundled in the Jar file in the submission.

3 Tasks Lists and Implementation Details

3.1 Base Tasks

Diverse Species: With how the simulation is implemented, adding new species is as easy as adding the species' behavioural data into a JSON file. Each entity species can be of type `Prey`, `Predator`, or `Plant`, and the data is added accordingly. For example, for the predator `Fox`, the following definitions are used:

```
{
  "name": "Fox",
  "multiplyingRate": [0.05, 0.15],
  "maxLitterSize": [1, 4],
  "maxAge": [80, 120],
  "matureAge": [40, 40],
  "mutationRate": [0.01, 0.05],
  "maxSpeed": [4, 6.5],
  "sight": [30, 50],
  "numberOfEntitiesAtStart": 12,
  "eats": ["Rabbit"],
  "size": [3, 6],
  "colour": [230, 20, 40],
  "overcrowdingThreshold": [8, 25],
  "overcrowdingRadius": [10, 15],
  "maxOffspringSpawnDistance": [3, 5]
}
```

The values that are arrays and contain two values (such as `sight`, `size`, or `maxSpeed`) represent the minimum and the maximum values that the entity can have for that genetic trait. In addition, these values can mutate when the entity breeds/multiplies. However, `numberOfEntitiesAtStart` and `eats` are fixed values that are not subject to mutation, as they are not genetic traits and they define what the entity is in the context of the simulation. Finally, the following entities are considered in the simulation:

- `Grass` — Plant.
- `Rabbit` — Prey, eats grass.
- `Squirrel` — Prey, eats grass.
- `Wolf` — Predator, eats rabbit and squirrel.
- `Fox` — Predator, eats rabbit.
- `Bear` — Predator, eats wolf and fox.

Two Predators Competes for the Same Food Source: With the JSON configuration file, it is quite easy to add multiple species for any type of entity. In this case, only two predators (`Wolf` and `Fox`) compete for the same food source (`Rabbit`), which is a prey species. The `Bear` is also added as a predator that eats both wolves and foxes.

Distinguishing Gender: Each animal has their own gender, represented in their genetics, which affects reproduction mechanics. Only animals of opposite genders can reproduce. Specifically, the gender genetic trait is implemented as an Enum with two values: `MALE` and `FEMALE`. Plants do not have gender in the genetics system, meaning that they reproduce asexually.

Tracking Time of Day: An `Environment` class is used to track the time of day and the weather, which governs how both cycles impact entity behaviour. During the night, entities will not move unless they are hungry or there is a predator nearby. Additionally, when sleeping, food consumption is reduced. The day-night cycle can be controlled by the JSON file, and the time of day is displayed on the screen. Additionally, as the day progresses into the night, the screen darkens to represent the time of day, without affecting the text on the screen.

3.2 Challenge Tasks

Adding Plants: Plants have been added, featuring growth and reproduction dynamics. Plants die when they detect too many plants of the same species nearby (as determined by their overcrowding genetics: `overcrowdingThreshold` and `overcrowdingRadius`), which results in natural looking patches of grass.

Adding Weather: As mentioned, weather is added under the `Environment` class, wherein weather conditions influence behavior and visibility, increasing the realism in the simulation environment. There are 4 weather conditions:

- `Clear` — No effect on entities.
- `Raining` — Plants grow faster (by a defined factor in the plants' genetics).
- `Windy` — Pushes entities in the wind direction, even when they are sleeping. Wind direction is also visualised for the ease of the user.
- `Storm` — Slows down entities by some factor and has the effect of windy. Different to windy condition, stormy condition is more severe, in the sense that the wind changes directions much more rapidly.

Genetics System: As one of the self-admitted challenges, a genetics system for all of the entities was implemented. As mentioned earlier in the first base task, when reproducing, animals combine their parents' genetics to form their own, with a chance to mutate certain attributes by some mutation factor. Specifically, if $r \in [0, 1]$ is a random number, then the new genetic trait is calculated as:

$$\text{value} = \text{fatherTrait} \times r + \text{motherTrait} \times (1 - r),$$

allowing for a smooth transition between the parents' traits. Then, if we define $s \in \{-1, 1\}$ to be a random value, the mutation factor is applied to the new trait as follows:

$$\text{newTrait} = \text{value} + \text{value} \times \text{mutationFactor} \times s, \quad (1)$$

where the mutation factor is a value, in the range $[0, 1]$, that determines how drastic the mutation is. This system allows for a wide range of genetic diversity in the simulation, which is easily observable when the `mutationFactor` is increased. Lastly, plants reproduce asexually, so they inherit their parent's genetics directly, but these can also mutate according to Equation 1.

JSON Configuration File: Almost every single aspect of the simulation is controlled from this file, including the entities' genetics, the environment, and the simulation parameters. The JSON file is loaded at the start of the simulation, by utilising the external package Gson. Then, the simulation is run according to the parameters defined in the file. As well as the entity genetic intervals mentioned above, the following are the parameters that can be controlled in the JSON file:

- `foodValueForAnimals` — Scales the food value when an entity eats an animal.
- `foodValueForPlants` — Scales the food value when entity eats a plant.
- `animalHungerDrain` — Controls the rate of hunger drain over time.
- `animalBreedingCost` — Scales how much food is consumed during breeding (note that food is a value in the range 0 to 1).
- `mutationFactor` — How drastic the mutation changes are.
- `entityAgeRate` — How fast entities age.
- `fieldScaleFactor` — The size of the field, smaller value means more zoomed in.
- `weatherChangeProbability` — The probability of the weather changing at the end of the day.
- `windStrength` — How strong the wind pushes entities.
- `stormMovementSpeedFactor` — How much to slow entities during a storm.
- `dayNightCycleSpeed` — How fast the time passes in the simulation.
- `doDayNightCycle` — Whether the day-night cycle is enabled.
- `doWeatherCycle` — Whether the weather is enabled.
- `showQuadTrees` — Whether to show the debug effect of quadtrees. It just looks really cool.
- `animalHungerThreshold` — The level of food level when an animal is considered "hungry."
- `animalDyingOfHungerThreshold` — The level of food level when an animal is considered to be "dying of hunger."

Quadtree Optimisation: While not a visible feature, due to its complexity, it is worth discussing. A quadtree is utilised to store entities, instead of a list. This is done to efficiently handle entity proximity checks. Each frame, entities are added to the quadtree, and when they need to find nearby entities, they query the tree. This is is a major upgrade to the naïve approach of $O(n^2)$ time-complexity, where every entity checks its distance to every other entity (1000 entities means 1,000,000 calls, 60 times a second). Using the quadtree for every entity has an average complexity of $O(n \log n)$, since the quadtree organises entities by proximity. Overall, this greatly improves performance, making the experience of the simulation much smoother.

Graphics: Some additional visual effects were also included, such as different shapes for different entity types (squares for predators, circles for preys, and triangles for plants), as well as rain and lightning, and a day-night darkening effect (just to make the weather more visual). There is also some text describing the time, day, weather, wind direction (when windy or stormy), and current entity count for each type of species.

4 Code Quality Considerations

4.1 Coupling and Responsibility-Driven Design

Coupling is minimised by separating all major functions into different classes. The class structure of the program starts with the `Engine` class. The `Engine` controls the main loop of the simulation, combining the actual simulation and the graphics together.

The `Engine` only stores the `Display` for graphics, the `Simulator` for updating the simulation and a `Clock` class for maintaining the frame rate. The `Engine` only coordinates these classes, it does not handle their internal logic, meaning it has minimal coupling.

Graphics are handled by the `Display` and the `RenderPanel` classes. The `RenderPanel` class extends the `JPanel` class and is responsible for rendering the simulation, and since it has low coupling, it is easy to change the `RenderPanel` class to render the simulation on different mediums — a web browser, for example. Furthermore, the simulation is controlled by the `Simulator` class. All entities are stored in a `Field` class, which is created by a `FieldBuilder` class to move the population of the `Field` outside of the `Field` class, improving Responsibility-Driven Design and minimising coupling.

All of the animals and plants in the simulation originate from an `Entity` class. This then has `Animal` and `Plant` as subclasses, and `Animal` has `Predator` and `Prey` as subclasses. It is ensured that any method or attribute shared by any class is stored in their respective parent class to reduce code duplication and enforce Responsibility-Driven Design for each subclass. The subclasses strictly only do things that they do differently from other sibling classes.

Also, as mentioned above, the `Entity` class has an `update()` method which is called in the `Simulator` class. This minimises coupling as all entity specific behavior remains encapsulated, while remaining easy to invoke. Additionally this `update()` method is overridden in the subclasses to allow for different behavior for each type of entity, while functionality that every entity shares is called from the parent class, such as the `incrementAge()` method.

Finally, only necessary dependencies are stored between classes, making the code base low in coupling. Naturally, as Responsibility-Driven Design was strictly followed for every class, the coupling is further reduced.

4.2 Cohesion

To improve cohesion, the `Animal` class has four main attributes:

- `AnimalMovementController`,
- `AnimalHungerController`,
- `AnimalBreedingController`, and
- `AnimalBehaviourController`.

Each of these controllers control their respective function of the `Animal` class. Similar to the controllers of the `Animal` class, controllers for the `Environment` class are also created, the `WeatherController` and the `TimeController`. This massively increases cohesion as the different operations are split into different relevant sections, making the code much easier to understand. This also has the added benefit of making the `Animal` and `Environment` classes quite small, and improves responsibility driven design and code readability.

The structure of `Entity` and its subclasses also lends itself to high cohesion — its extremely clear what an `Animal` should do, and what a `Predator` and `Prey` does differently while also inherintly being subclasses of `Animal`. This once again decreases code duplication and increases the cohesion of the code base.

4.3 Maintainability

The JSON file and genetics system is highly modular, allowing easy modifications by updating only the JSON file and either the `SimulationData`, `AnimalData`, or `PlantData` classes accordingly.

Code is structured into packages, improving organisation and making it easier to locate, modify, and extend functionality while maintaining encapsulation. The code base is written with low coupling and adheres to Responsibility-Driven Design, resulting in high modularity. This means adding functionality is quite straightforward, without needing to change other parts of the code. Modular code also makes unit testing very simple, as demonstrated by the unit test classes in the code base, which were highly effective in testing the code and ensuring it was working as expected as the code was developed.

One aspect that may prove difficult is making the simulation deterministic. The simulation is not deterministic and runs differently each time, as the random number generation system uses different instances of `Math.random()` throughout the code. This is one aspect that could be improved upon by utilising a single random number generator instance.

5 Final Remarks

This project implements a dynamic, modular and optimised simulation of an ecosystem. The system expands and goes beyond the base tasks, completely reworking and improving the original code. It is hoped that the simulation is both educational, and enjoyable to use and watch, while modifying the JSON file to see how different parameters affect which species thrive and which do not.