Assignment 3: Predator-Prey Simulation - Report

Mahdi Razzaque — K24033150
Ozgur Dorunay — K24027042

February 18, 2025

1 Simulation Description

My simulation models a basic ecosystem within a rectangular field. It contains several acting species, including predators (wolves and bobcats) and prey (squirrels and grouse), as well as plant life (seeds and berries). These entities interact based on rules governing breeding, feeding, and survival. The simulation is visualised through a graphical interface that displays the current state of the field, weather, and time, and provides statistical information.

2 Core Tasks

2.1 Species and Predation

I introduced five distinct acting species into the simulation: wolves, bobcats, squirrels, grouse, and plants (seeds and berries). Instead of simply copying and pasting code from the original template project, I attempted a significant refactoring effort. Realising that most of the code for predator and prey species was fundamentally similar, I identified common behaviours and combined them into the superclass `Animal`. The primary difference between predators and prey was their feeding behaviour, with predators actively hunting whilst prey primarily graze or forage. By abstracting common logic to the `Animal` class, I streamlined the creation of new animal subclasses and enhanced code reusability. This also simplified the act method, allowing the plant and animals to be placed using the same method. The simplification of animal subclass generation will decrease the time and complexity of adding new subclasses in the future. The wolf and bobcat species were designated as predators, while squirrels and grouse were categorised as non-predatory prey. Wolves are designed to hunt both squirrels and grouse, while bobcats specialise in preying exclusively on squirrels. These relationships establish a clear food web within the simulation. The addition of multiple predators competing for the same prey species was implemented as a core task. For each species, each species would be more or less likely to breed, and the more likely each species was to reproduce. Both wolf and bobcat species compete for squirrels as a food source. This competition adds a layer of complexity to the ecosystem dynamics. The population sizes of both predator species are influenced by the availability of squirrels, leading to fluctuations in their numbers based on hunting success and squirrel population levels.

2.2 Male and Female Individuals

I added an enum class called `Gender` to distinguish between male and female individuals within the animal populations. The `getRandomGender()` function within the `Gender` class utilises the `Random` module to randomly assign either a male or female gender to each newly created animal. This function is called during the field population process, ensuring a roughly even distribution of genders across all animal species. The existence of the `Gender` class has been helpful when determining if the animals can breed or not. During breeding attempts, the `giveBirth` method checks the gender of adjacent animals of the same species. Breeding is only successful if a mate of the opposite gender is found. Furthermore, there are a similar number of males and females. Since they both must exist for the breeding function to execute. Then a function, using the Random module, is executed which decided the size of the litter for the next breeding. The Gender class has enabled a more realistic way to create the next generation, with the need for 2 individuals.

2.3 Time of Day

The simulation incorporates the concept of time by using steps to represent minutes. During the `Time` object initialisation, the `timeStepIncrement` parameter dictates how many minutes elapse with each simulation step (default: 60 minutes). The `Grouse` class's `act` method was overridden to incorporate time-based behaviour. During the night it ignores other activities within act and will not move (to make it more realistic). They are able to breed at night, but at a much reduced rate than normal.

3 Challenge Tasks

3.1 Weather Implementation

I implemented a weather system that influences the behaviour of simulated agents. The system retrieves real-time weather conditions from the external weather API, weatherapi.com.

- **Implementation:**
  - Developed a `Weather` class to handle API requests and data parsing. The API key is hardcoded for simplicity, though ideally, this would be read from a system environment variable for security reasons.
  - Implemented the `getWeatherEmoji` method to retrieve the current weather condition and map it to corresponding emoji representation.
  - The `getWeatherDescription` method employs `String` manipulation instead of an external JSON library due to uncertainties about how BlueJ handled external libraries.
  - Implemented `isEnoughStepsPassed` to regulate API calls, checking if 100 simulation steps have elapsed before updating the weather, thus preventing excessive API requests.
- **Challenges:**
  - The weather API code was written and tested in IntelliJ, as the approach to external library handling within BlueJ was unknown.
  - Integrating an external API required learning new networking and data handling concepts in Java.
  - Due to the inability to utilise external libraries for JSON parsing, I decided to use `String` manipulation techniques instead. This presented challenges due to the inherent lack of robust error handling and the complexities associated with navigating the structure of JSON data using rudimentary string operations.
- **Agent Influence:**
  - Berries only grow if the weather is sunny/cloudy.

3.2 Plant Implementation

I added seeds and berries as food sources for prey animals, enabling their survival and adding another layer of complexity to the ecosystem. To facilitate the management of both plant and animal entities, I introduced an `Entity` superclass. This abstraction simplifies various aspects of the simulation, as methods can now be written to handle `Entity` objects rather than needing separate implementations for animals and plants. For example, the field's `populate` method, responsible for initialising the simulation, can now add both animals and plants using the same logic, treating them uniformly as `Entity` objects.

- **Implementation:**
  - Introduced `Seeds` and `Berries` classes, each with attributes defining their growth, reproduction, and lifespan. The `Animal` class, `Plant` class, and the classes of the animals and plants have been modified to inherit from the `Entity` class. This also required each subclass to make an act method that takes a 'Field' object and implements the subclass changes in the environment. Now, methods can be written to handle a single 'Entity' object that implements all the subclass differences. This reduces code and makes the project easier to maintain and scale.

- Plants only grow during their specified growth start and end hours, adding time-based realism. This logic is within the `Plant` class, and the `Seeds` and `Berries` subclasses both implement this logic.
- The plants are placed using the populate method, the same as the animals, due to the existence of the `Entity` class. They also have random changes to spread, increasing the density of plant matter.

- **Code and Attributes:**
  - **Growth Rate**: How often they grow at their stage
  - **Reproduction Rate**: How often plants reproduce
  - **Life Span**: How long the plant lives, unless set to infinite (-1)
  - **Spread Rate**: How often the plant spread
  - **Growth Start and End Hours**: How often they start and end their time

3.3 Disease

I introduced a disease system to simulate infection dynamics in animals, allowing diseases to spread and affect the ecosystem in a more complex way. To manage the disease mechanics, I created a Disease superclass and derived specific disease types (e.g., Flu and Rabies) from it. This system provides flexibility, allowing different diseases to have unique behaviors while maintaining consistency in their core properties.

- **Implementation:**
  - Created the `Disease` superclass to define shared attributes for all diseases, including **infection duration**, **transmission probability**, and **lethality**.
  - Introduced subclasses for specific diseases, such as `Flu` (non-lethal) and `Rabies` (lethal). Each subclass overrides core methods to reflect the disease's specific behavior.
  - The `Disease` class includes methods like `spreads()` to determine if a disease should be transmitted to another animal and `isCured()` to check if the infection has ended.
  - Animals are randomly infected during the simulation's initialization (`populate` method), and infected animals spread diseases when they encounter others, based on the disease's transmission probability.
  - The system tracks infection progress, allowing infected animals to either recover or die based on the disease's duration and lethality.

- **Challenges:**
  1. **Tracking Infected Animals in `FieldStats`** Initially, the simulation failed to correctly track the number of infected animals because the `generateCounts()` method did not update the infected count properly. The issue was traced to the fact that the `infectedCount` was not being reset in the `reset()` method, leading to inaccurate statistics. To resolve this, I integrated the `incrementInfected()` method into the `Counter` class, ensuring that the infected count is updated correctly with each simulation cycle.
  2. **Displaying Infection Data in the GUI** The GUI initially failed to show the correct number of infected animals because the `getPopulationDetails()` method in the `FieldStats` class did not access the infection count correctly. This was corrected by modifying the `incrementCount()` method to handle infected and non-infected animals separately, ensuring that the correct statistics were displayed in the interface.

4 Code Quality

4.1 Coupling

The refactoring effort to move common animal behaviours to the `Animal` superclass significantly reduced coupling between specific animal subclasses (`Wolf`, `Bobcat`, `Squirrel`, `Grouse`). Instead of each subclass

containing redundant code for movement, breeding, and eating, these behaviours are now contained within the `Animal` class. This allows each subclass to focus solely on its unique attributes and behaviours, such as specific food preferences or time-based activity patterns.

4.2 Cohesion

The creation of separate `Seeds` and `Berries` classes with well-defined attributes (growth rate, reproduction rate, lifespan, spread rate) promotes high cohesion. Each class is responsible for managing its own data and behaviours, leading to modular and understandable code. Furthermore, there is the superclass `Plant`, that shares common functionalities of plants across the seed and berry classes, improving the code cohesion and readability.

4.3 Responsibility-Driven Design

The `Entity` abstract class is a prime example of responsibility-driven design. By extracting common attributes and behaviours shared by both `Animal` and `Plant` classes (e.g., location, `isAlive` status, `act` method), the `Entity` class assumes responsibility for managing these core aspects of all entities in the simulation. This reduces code duplication and simplifies the implementation of specific animal and plant behaviours in their respective subclasses. Furthermore, the Plant and Animal super classes are abstract, requiring the implementation of specific methods in the subclasses.

4.4 Maintainability

The `Weather` class promotes maintainability by including a fallback mechanism. If the external weather API is unavailable or returns invalid data, the simulation defaults to using a randomly generated weather condition. This ensures that the simulation can continue to function even in the absence of a reliable external data source. This design decision reduces the system's reliance on external factors and improves its overall robustness and long-term maintainability.

5 Known Bugs or Problems

- **Population Stability:** Achieving long-term population stability proved challenging. Initial simulations often resulted in the extinction of certain species or an overpopulation of others. This issue was addressed by carefully tuning the breeding probabilities, creation probabilities, and lifespan attributes of each species. Initially, the plants seemed to be growing too quickly and spreading too rapidly, leading to overcrowding and the eventual death of other species due to resource scarcity. Further adjustments to plant growth and reproduction rates were needed to establish a more balanced ecosystem.
- **Weather API Reliability:** The external weather API is subject to potential downtime or changes in its data format. While the simulation includes a fallback to randomly generated weather, changes to the API's data format could still disrupt the weather system. Future work could involve implementing more robust JSON parsing techniques to better handle potential format changes and ensure continued functionality.

6 Conclusion

This assignment provided a valuable opportunity to expand and refine the predator-prey simulation. The implementation of weather and plant growth added realism to the simulation, while the code quality improvements made the codebase more maintainable and scalable. The refactoring has decreased the amount of redudant code in the code, especially due to the creation of the Entity class and the Animal class, which combines a lot of different logic and attributes. The challenges encountered, particularly with population stability and API integration, highlighted areas for future exploration and improvement. The population size of the simulation has been significantly increased through changing the attribues, while the weather has added another challenge. This adds to the real world examples of the predator prey situation, with outside influences.