Mohammed Rahmat Azam **2nd assignment/6. Task** 11th May 2024
FRONMX
azamrahmat9@gmail.com
Group 11

Task

Create a program to simulate colonies of different kinds of animals. The simulation will manage various species (lemmings, hares, gophers, foxes, wolves, owls) within a tundra environment. Each colony has a name (string) and population size (natural number). The simulation will run over multiple turns, during which predators will affect the populations of prey, and prey populations will grow periodically. If a prey colony's population reaches zero or below or it quadruples compared to starting value, we stop the simulation. The simulation should identify the final state of each colony at the end..

- ***Lemmings:*** *On every second turn, their population doubles. If the population exceeds 200, it is reduced to 30. When preyed upon by a predator colony, the number of lemmings decreases by four times the number of predators.*
- ***Hare:*** *On every second turn, their population grows by 50%. If the population exceeds 100, it is reduced to 20. When preyed upon by a predator colony, the number of lemmings decreases by two times the number of predators.*
- ***Gopher:*** *On every fourth turn, their population doubles. If the population exceeds 200, it is reduced to 40. When preyed upon by a predator colony, the number of lemmings decreases by two times the number of predators.*

Every data is stored in a text file. The first line contains the number of creatures. Each of the following lines contain the data of one creature: A string for the colony name, one character for the species (l for lemming, h for hare, g for gopher, f for fox, w for wolf, o for owl), name of the creature (one word), and an integer for the initial population size.

Analysis[1]

Independent objects in the task are the colonies, which can be divided into predators and prey. Each colony has a name and population size that can be accessed and modified. The simulation proceeds turn by turn, and each turn affects the populations of the colonies based on predefined rules.

- **Prey Colonies (Lemmings, Hares, Gophers)**: Their populations increase periodically and are reduced by predator interactions.

- **Predator Colonies (Foxes, Wolves, Owls)**: Their populations are sustained by preying on other colonies.

[1] This part may be skipped. It is enough to show the tables of traverse in the Planning section.

Plan[2]

To describe the colonies, several classes are introduced: a base class `Colony` to describe the general properties and six derived classes for the specific species: `Lemming`, `Hare`, `Gopher`, `Fox`, `Wolf`, and `Owl`. Regardless of the type of colony, they have several common properties, such as the name (`name`) and the population (`num`). These properties can be accessed through getter methods. Additionally, the colonies have a method to update their population (`UpdatePopulation()`) based on specific rules for each species. The general class `Colony` will be abstract because the method `UpdatePopulation()` is abstract and we do not want to instantiate this class directly.

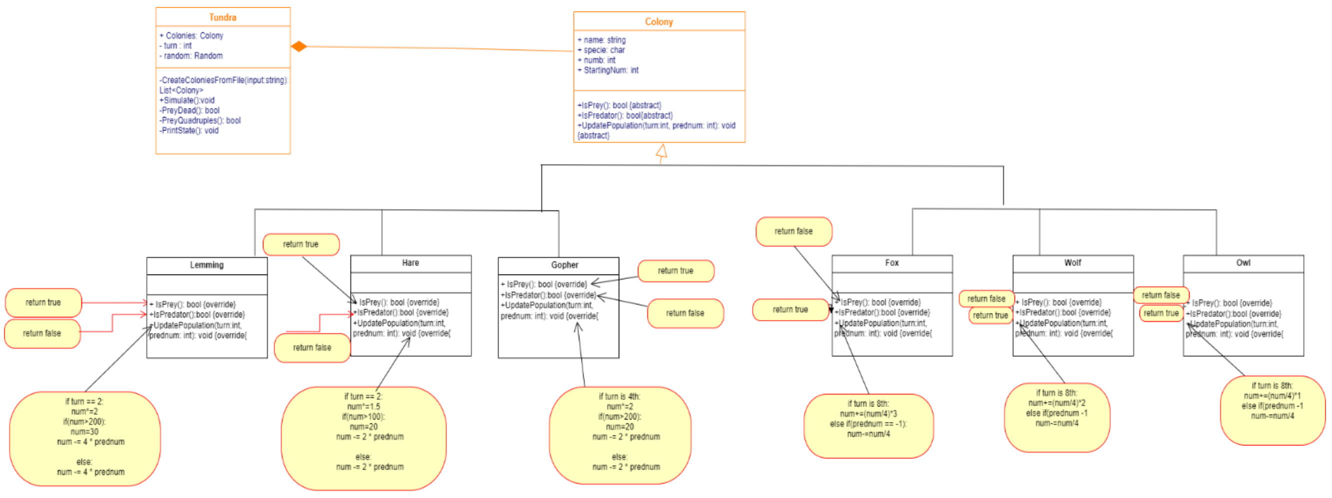
Each specific colony class will initialize the name and the population through the constructor of the base class and override the method `UpdatePopulation()` in a unique way. The rules for updating the population are specific to each species and are implemented in the respective classes. For example, a `Lemming` doubles its population every second turn and reduces its population by four times the number of predators.

The `Tundra` class will manage the simulation. It will read colony data from an input file, create the appropriate colony objects, and store them in a list. The `Tundra` class will also contain a method to simulate the turns (`Simulate()`), updating the population of each colony based on the rules defined in their respective classes. The simulation continues until all prey are extinct or a prey population quadruples.

To ensure flexibility and adherence to SOLID principles, the `UpdatePopulation()` method in each specific colony class is implemented without conditionals that depend on other classes. This design avoids the need to modify multiple classes when extending the program with new species or rules, thereby maintaining the open/closed principle.

Unit tests will be created to verify the behavior of each colony type and the overall simulation. These tests will check various scenarios, such as population updates with and without predators, edge cases with maximum population limits, and the correct handling of input files.

[2] Plain text explanation is not necessary for the student documentations


Tundra
+ Colonies: Colony
- turn : int
- random: Random
-CreateColoniesFromFile(input:string): List<Colony>
+Simulate():void
-PreyDead(): bool
-PreyQuadruples(): bool
-PrintState(): void
Colony
+ name: string
+ specie: char
+ numb: int
+ StartingNum: int
+IsPrey(): bool {abstract}
+IsPredator(): bool{abstract}
+UpdatePopulation(turn:int, prednum: int): void {abstract}
Lemming
+ IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return true
return false
if turn == 2: num*=2 if(num>200): num=30 num -= 4 * prednum else: num -= 4 * prednum
Hare
IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return true
return false
if turn == 2: num*=1.5 if(num>100): num=20 num -= 2 * prednum else: num -= 2 * prednum
Gopher
+ IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return true
return false
if turn is 4th: num*=2 if(num>200): num=20 num -= 2 * prednum else: num -= 2 * prednum
Fox
+ IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return false
return true
if turn is 8th: num+=(num/4)*3 else if(prednum == -1): num-=num/4
Wolf
+ IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return false
return true
if turn is 8th: num+=(num/4)*2 else if(prednum -1 num-=num/4
Owl
+ IsPrey(): bool {override}
+IsPredator():bool {override}
+UpdatePopulation(turn:int, prednum: int): void {override{
return false
return true
if turn is 8th: num+=(num/4)*1 else if(prednum -1 num-=num/4

Example Logic in the Simulation

- **Initialization**:
  - Colonies are created from the input file, specifying their name, species, and initial population.
  - Example input: "Lemming l 10", "Fox f 5" would create a `Lemming` colony with 10 individuals and a `Fox` colony with 5 individuals.
- **Simulation Turn**:
  - Each turn, prey populations are updated based on predator interactions.
  - Predators are randomly selected to interact with prey, affecting prey populations.
  - Predator populations are updated based on internal logic (e.g., natural growth or decline).
- **End Conditions**:
  - The simulation checks if all prey are dead or if any prey population has quadrupled from its starting number.
  - If either condition is met, the simulation ends.

Detailed Breakdown of Key Methods

- `CreateColoniesFromFile(string inputFilePath)`:
  - Opens the input file and reads the number of prey and predator colonies.
  - Initializes colonies based on the species and population specified in the file.
  - Validates the data to ensure consistency.
- `Simulate()`:
  - Iterates through turns, updating populations and printing the state.
  - Uses nested loops to process prey and predator interactions.
  - Randomly selects predators to affect prey populations.
  - Checks end conditions after each turn.
- `UpdatePopulation(int turn, int predatorNum)`:
  - Abstract method in the `Colony` base class, implemented by each subclass to define specific population update logic.

Testing

Grey box test cases:

Explanation of Test Cases

1. **TestLemmingPopulationUpdate**:
    - Tests the population update logic for lemmings on the second turn without any predators. Ensures the population doubles as expected.
2. **TestLemmingPopulationWithPredators**:
    - Tests the population update logic for lemmings on the second turn with predators present. Ensures the population doubles and then decreases by four times the number of predators.
3. **TestTundraSimulation**:
    - Runs the full simulation from an input file and checks that the population of lemmings is not negative after the simulation.
4. **TestEdgeCaseEmptyFile**:
    - Tests the scenario where the input file is empty. Ensures no colonies are created in this case.
5. **TestEdgeCaseMaxPopulation**:
    - Tests the scenario where the lemming population exceeds 200. Ensures the population resets to 30 as per the rules.

These tests cover various aspects of the simulation, including initialization, population updates, edge cases, and overall simulation behavior, ensuring the correctness and robustness of the Tundra simulation implementation