Assignment 2b

COMP 2526 Object-Oriented Programming with Java

Due no later than 23:59 PM on Sunday 19 November 2017

1 Purpose

Modify assignment 2a to examine (and possibly change) your choices in OOP design.

2 Description

You will modify your solution to assignment 2a so that lifeforms can mate, eat, and die of starvation. You will add two kinds of lifeforms, a Carnivore and an Omnivore, to the existing Herbivore and Plant.

3 Requirements

You will add two kinds of lifeforms, a Carnivore and an Omnivore.

1. Carnivore
   (a) displayed as a Magenta Cell
   (b) eats Herbivores and Omnivores
   (c) can give birth if there is/are:
      i. at least 1 Carnivore neighbour
      ii. at least 2 free neighbouring cells
      iii. at least 2 neighbouring cells with food (Herbivores or Omnivores).
   (d) Must find something to eat before 7 "turns" have passed or it dies
   (e) Initial placement is random; each Cell has a 10% chance of generating a Carnivore when the game starts
2. Omnivore
   (a) displayed as a Blue Cell
   (b) eats Plants, Herbivores, and Carnivores
   (c) can give birth if there is/are:
      i. at least 1 Omnivore neighbour
      ii. at least 3 free neighbouring cells
      iii. at least 3 neighbouring cells with food (Herbivores or Carnivore or Plants).
   (d) Must find something to eat before 2 "turns" have passed or it dies
   (e) Initial placement is random; each Cell has an 8% chance of generating an Omnivore when the game starts

   Herbivores can reproduce now, too. An Herbivore can give birth if there are at least 2 Herbivore neighbours, at least one free neighbouring cell, and 2 neighbouring cells with food (Plants). An Herbivore cannot be born and give birth during the same turn.

NOTE: some interpretation of the rules is likely to occur so your program's behaviour may not be *exactly* the same as the example provided. That's okay! Note, however, that all rules should be applied in a logical way and we will mark your assignment accordingly

To generate lifeforms, you will probably implement code in your World class that looks like this (without the magic numbers, of course):

```
public void init() {
    RandomGenerator.reset();
    for (int i = 0; i < w; i++)
        for (int j = 0; j < h; j++) {
            cell[i][j] = new Cell(this, i, j);
            int val = RandomGenerator.nextNumber(100);
            if (val >= 80) {
                cell[i][j].addLife(new Herbivore(cell[i][j]));
            } else if (val >= 50) {
                cell[i][j].addLife(new Plant(cell[i][j]));
            } else if (val >= 40) {
                cell[i][j].addLife(new Carnivore(cell[i][j]));
            } else if (val >= 32) {
                cell[i][j].addLife(new Omnivore(cell[i][j]));
            }

        }
}
```

Things to consider

1. An update() method in World that:
   (a) acquires all the lifeforms (you can store them in an ArrayList)
   (b) for each lifeform
      i. if alive, move()
      ii. if alive, reproduce() // could die during movement
      iii. if dead, remove from the board.
2. A Lifeform class that does most of the work and contains methods like:
   (a) move() Looks for a new Cell that is either empty or contains something it can eat. Lifeforms prefer a spot with food. Eats the food-lifeform in the desired randomly chosen cell (if not empty) and then moves into that cell (removing itself from its old location)
   (b) updateHealth() Updates status of health based on last time eaten
   (c) reproduce() Gives birth to a new lifeform based on rules for the lifeform
   (d) choosePosition() Chooses cell to move to based on rules for lifeform
   (e) getNeighhours() Returns an array containing the neighbouring Cells
   (f) eat() Consumes the lifeform at the destination move location.
   (g) NOTE: make sure a Lifeform cannot eat or move if it is dead or has been eaten!
3. Eating. A clever approach to handle eating yet allow for future changes is to create an interface to represent what a type can eat e.g. HerbivoreEdible. No methods are provided but whatever an Herbivore can eat then implements this type. The methods countFood() and choosePosition() each then test using instanceof if HerbivoreEdible. All types that a Herbivore can eat would simply need to implement HerbivoreEdible. Similarly interfaces CarnivoreEdible and OmnivoreEdible can be created.
4. Reproduction. All things give birth based on a set of rules specific to that class however note that the FORM of the rules is always the same, just the specifics are different, i.e., how many empty cells

are needed, how much food is needed, etc. These aspects can be obtained from parameters passed in to a single method found in Lifeform. The only other difference is the type of lifeform. For this, the reproduce() method can invoke an abstract method createLife() that all lifeforms must have overridden which simply creates a lifeform of the desired type and returns Lifeform.

5. Here's a helpful algorithm you might use or adapt for reproduction:

```
giveBirth(numberMates, numberEmptyCells, numberFoodCells, lifeRemaining)
if all conditions passed
find cell for new lifeform to occupy
Lifeform l = giveBirth()
cell.setLife(l)
```

4 Milestone 1 (10% of mark): Create the additional lifeform classes, instantiate and display on the board correctly during first lab next week

5 Marking Guidelines

50% Functionality (does it work: eat birth, dead, movement, etc)
10% Milestone 1 (described above)
20% Good design (elimination of duplicate code, data and methods are encapsulated, inheritance used correctly)
20% Comments and style (use Checkstyle)