Assignment 2b

COMP 2526 Object-Oriented Programming with Java

Milestone 1 due in lab during the week of March 7th - 11th
Complete assignment due on Friday March 25th at 11:59 PM

1 Purpose

Modify and enhance assignment A2a in order to examine your choices in OOP design.

2 Description

You are going to modify the "Game of Life" simulation you created for A2a. The modifications will include new lifeforms (Carnivores and Omnivores) and behaviours as described in the Requirements section below.

3 Requirements

This assignment will demonstrate how expanding and maintaining a software system can be easier with a well planned design. You must adhere to these rules:

Plant

1. Colour GREEN
2. Eats nothing
3. Can 'give birth' to a seedling if there are:
   (a) 3 or more other Plant neighbours
   (b) 2 or more empty neighbouring Cells
4. Wilts away and dies after 10 moves.

Herbivore

1. Colour YELLOW
2. Eats Plants
3. Can 'give birth' to a baby herbivore if there are:
   (a) 1 or more other Herbivore neighbours
   (b) 1 or more empty neighbouring Cells
   (c) 2 or more neighbouring Cells with food (Plants)
4. Must eat within 10 moves or dies.

Carnivore

1. Colour MAGENTA
2. Eats Herbivores
3. Can 'give birth' to a baby Carnivore if there are:

   (a) 1 or more other Carnivore neighbours
   (b) 3 or more empty neighbouring Cells
   (c) 3 or more neighbouring Cells with food (Herbivores)
4. Must eat within 3 moves or dies.

Omnivore

1. Colour BLUE
2. Eats Herbivores, Carnivores, Plants
3. Can 'give birth' to a baby Omnivore if there are:
   (a) 1 or more other Omnivore neighbours
   (b) 3 or more empty neighbouring Cells
   (c) 3 or more neighbouring Cells with food (Herbivores, Carnivores, and Plants)
4. Must eat within 2 moves or dies.

Optional: change (darken) a Lifeform's colour over time until they eat again. Do this by reducing all their colour channels (RGB) by 80% each turn.

Things to consider when implementing your solution's design:

1. Create an **abstract Lifeform** superclass that contains these methods:
   (a) getNeighbours() fetches and returns a list of the neighbouring Cells (perhaps as an array, or maybe an ArrayList)
   (b) choosePosition() chooses a Cell to move to based on the rules for its type of Lifeform
   (c) move() looks for an adjacent Cell to move to that is either empty or contains something it can eat. If it finds a neighbouring Cell that contains something edible, it eats that edible Lifeform in the Cell and then moves into that Cell (removing itself from its old location in the process, of course).
   (d) eat() eats the edible Lifeform found at the (possibly specified) position
   (e) a method to track how many moves have passed since the Lifeform last ate.
   (f) reproduce() 'gives birth' to a new Lifeform based on the rules for its particular type of Lifeform.
2. Don't forget to ensure a Lifeform doesn't move or eat once it has been eaten or starved.
3. It might be helpful to add an update() method to the **World** class that:
   (a) Creates a list of all the Lifeforms (you can store them in an ArrayList)
   (b) For each Lifeform:
      i. if it's alive, move() it and giveBirth()
      ii. if it's dead, remove it from the board.
4. For **generating Lifeforms**, consider code similar to this:

```
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);
            // gives a number bwetween 0 and 99
            int val = RandomGenerator.nextNumber(99);

            if(val >= 80) { // 20% chance of creating an Herbivore
                cell[i][j].addLife(new Herbivore(cell[i][j]));
            }
```

```
                else if(val >= 60) { // 20% chance of creating a Plant
                    cell[i][j].addLife(new Plant(cell[i][j]));
                }

                else if(val >= 50) {  // 10% chance of creating a Carnivore
                    cell[i][j].addLife(new Carnivore(cell[i][j]));
                }

                else if(val >= 40) {  // 10% chance of creating an Omnivore
                    cell[i][j].addLife(new Omnivore(cell[i][j]));
                }
            }
        }
    }
```

5. **Eating:** To handle eating yet allow for future changes, a suggested technique is to create an interface to represent what a type can eat, i.e., HerbEdible for something an Herbivore can eat. No methods need to be defined in the interface, but whatever a Herbivore can eat then implements this type. The Herbivore can implement methods such as countFood() or choosePosition() that can use the instanceof operator to determine of another Lifeform is HerbEdible. This means that all Lifeforms that an Herbivore can eat just need to implement HerbEdible. Similarly, consider interfaces like CarnEdible and OmniEdible.
6. **Reproduction:** All Lifeforms reproduce based on a set of rules specific to that type. Note, however, that the rules are basically the same and just the specifics are different, i.e., how many empty neighbouring Cells are needed, how much adjacent food is needed. 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. To reproduce, consider defining an abstract giveBirth() or createLife() method in the Lifeform superclass. All subclasses of Lifeform must implement this abstract method which simply created a Lifeform of the correct subclass. Note the return type of the method would still be Lifeform, the superclass.

We've included the source code for a suggested A2a solution. You may use this as your launching point.

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. However all rules should be applied in a logical way and your submission will be marked accordingly.

4 Marking Guidelines

You will demonstrate the milestone code to your lab instructor during the week of March 7th - 11th. Create the additional lifeform classes described in the Requirements section, and instantiate and display them on the worldboard according to the ratios described in the code snippet.

50%	Functionality (does your program function as described in this document)
20%	Good object oriented design (reduce/eliminate duplicated code, etc.)
20%	Comments and style (follow your lab instructor's guidelines)
10%	Milestone week of March 7th in lab

Good luck, and have fun!