CmpE 160 – Introduction to Object Oriented Programming

Project#1 - Nature Simulation

Due Date: April 8, 2018 23:55 PM

1 Introduction

In this project, you are going to implement a simple nature simulation/game composed of pixel-like creatures interacting with each other. In summary, you will implement two types of creatures: *Plants* will grow and reproduce, while *herbivores* will move around to eat them and will also reproduce when they accumulate enough health. The game will not be played by a human, but will rather play itself. The objective of this project is learning the fundamentals of object oriented programming, class/object structures and is-a relationship (inheritance). We provide you with the drawing-related functionality, so you can focus only on the game logic.

Before we present you the details of the project, let's show you what the game will look like in Figure 1 (the colors are optional and may be different for your implementation).

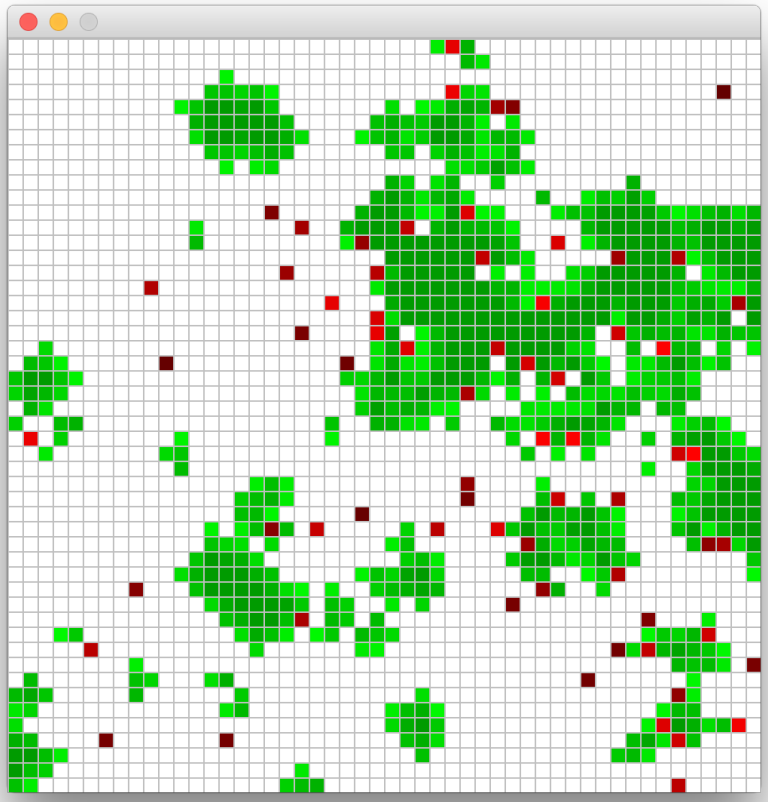

Figure 1: UI of the game.

2 Class Hierarchy

Within the content of the project, you are provided with 5 packages:

- `ui`: *do not modify*
- `game`: *do not modify*
- `main`: *you can modify to test your project, but it will be discarded*
- `naturesimulator`: *do not modify*
- `project`: *currently empty, you will put the classes you create in this package*

Before we go on to briefly explain the classes each package contains, let us say just one thing: *Don't panic!* We provide you with many classes only to make your life easier, not to complicate things. First of all, we give you all the drawing and user interface related functionality, so you don't have to worry about them. We even give you the base game logic, so all you'll have to do will be to implement different types of creatures, and if you do it right, the game will automatically run just fine. You don't have to fully understand the internals of most classes we give you; just being able to interact with them via their public methods will be enough. All the public methods are commented, so you can refer to them for any questions.

Having said that, we suggest that you at least take a peek at the inheritance hierarchy of the given classes, as we believe it is a good example. The given game system is designed to be generic enough to be used for any kind of grid-based game, not just this project. An important purpose of object oriented programming is being able to do just that.

The `ui` package includes the classes that are necessary for creating the main interface of the game, which will enable you to visualize the world you create. You can read the comments in the classes for further information, if you are curious. You should not modify the classes in this package in any way, as they will be reset when grading your project.

The `game` package includes the classes that are responsible for constructing the bridge between the backend of the game and the visual part. It provides a generic game system that can be used for any grid-based game. Similar to the `ui` package, you should not modify the classes provided in this package.

The `main` package includes the `Main` class that triggers the start of the game with the necessary parameters. The `Main` class, that is provided to you, contains example code to create a UI with the specified grid world size and game speed, and adds several randomly positioned creatures to the

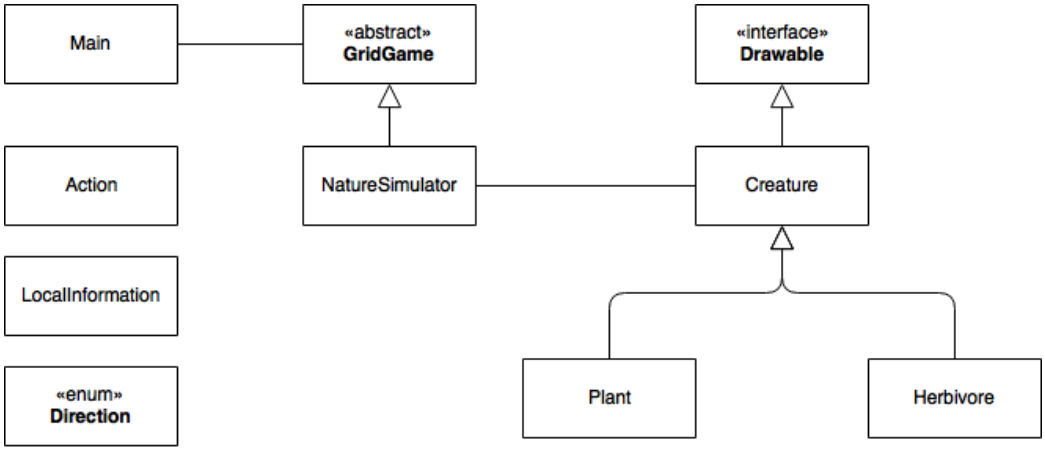

Figure 2: Class diagram.

game world. You are free to edit the main method to test your project in any way you like, but it will be discarded and replaced with our own method when grading your project. We expect your game to be functional with any set of parameters.

Lastly, the main logic and operations of the game are partially implemented in the classes that are provided by the `naturesimulator` package. You should not modify any of the classes in this package as well.

The main objective of this project is completing the implementation of the game logic by adding missing classes to the `project` package. The partial class hierarchy of the project is given in Figure 2. You need to create additional classes considering the necessary hierarchy:

- You **should implement** the `Creature`, `Plant` and `Herbivore` classes, and put them in project package.
- The signature of the methods and names of the fields to be implemented are actually specified within the available classes through function calls. In order to give an example case, `NatureSimulator` class has a method that triggers the `reproduce` function call that belongs to the `Creature` class. This means that, the `Creature` class should possess a `reproduce` method with the same signature. In a way, the given code will force you to correctly implement the missing methods and classes. So, please consider the available function calls while implementing the classes and necessary methods/fields.

3 Gameplay

The game is started through a call within the Main class (you should run this class). By creating a new `NatureSimulator` object and sending the `start` message to that object, the game automatically creates the UI and executes the game. Then, it proceeds with the logic implemented in `NatureSimulator`.

The main loop of the game, where each iteration is implemented by the `timerTick` method in `NatureSimulator` class, is executed via a periodic timer within the `GridGame` superclass. The `frameRate` parameter provided to the `NatureSimulator` constructor determines the speed of the game, i.e. how many times per second `timerTick` will be called. The game continues to run by itself until the application window is closed. You don't need to worry about stopping the game by any other means, even though we also provide a `stop` method.

Within each `timerTick` iteration, the game executes one of the possible actions for every single creature available in the game. The requirements and conditions of the actions are specified by the following section.

4 Implementation Details

Each turn forces the creatures in the game to take an action to progress the game. There are some rules that determine the actions to be taken by the creatures. This section provides the details about the implementation and rules of the game.

There are 4 possible actions that a creature can perform (see enum `Type` in class `Action`):

- **Move**: Creatures can move to an adjacent empty cell.
- **Reproduce**: Creatures can reproduce, i.e. create a new creature of the same type on an adjacent empty space.

- **Stay**: Creatures can stay at the same space.
- **Attack**: Creatures can attack another adjacent creature, in which case it will kill it and move to its position.

At each game loop iteration, the game will call the `chooseAction(LocalInformation information)` method of each creature, which is expected to return the action chosen by the creature. You can think of this method as the brain of a creature: Via this method, the creature will decide on which action to perform depending on the information about its environment. This information is contained in an instance of `LocalInformation` class, provided automatically as an input to this method. Please refer to the comments in the public methods of `LocalInformation` class for details on what kind of information is available for a creature.

The return type of `chooseAction(...)` is `Action` which contains the type of the action and the direction of it, if applicable. You should use one of the two constructors of `Action` to create and return the action you want to choose. Actions *move*, *reproduce* and *attack* require a direction, while action *stay* should not have a direction.

After `chooseAction(...)` method returns the chosen action, the game logic implemented in `NatureSimulator` checks whether the chosen action is valid, and if so, calls the respective action method of the creature. The possible action methods are:

- `void move(Direction direction)`
- `Creature reproduce(Direction direction)`
- `void attack(Creature creature)`
- `void stay()`

Each type of creature should implement the action methods that it is able to perform. How you should implement them for each creature type are detailed in Table 1. An important remark: The `chooseAction(...)` method should not make any changes to the state of this creature or any other creature, i.e. you should only choose the action you wish to execute and not execute it immediately. The actual action should be implemented in the respective action method, which will be automatically called by the game logic, if your action is determined to be valid.

An example: A Herbivore wants to move one square up (which is empty), therefore it creates an `Action` object with type `Action.Type.MOVE` and direction `Direction.UP` and returns it. It does not yet change its position. The game logic in turn determines that this action is valid, and therefore calls the `move(...)` function of the Herbivore with the argument `Direction.UP`. The Herbivore then takes the actual action by modifying its position to move up in `move(...)` method; i.e. its decrements its y coordinate by 1.

Table 1 depicts the action types and the creatures that are capable of taking the corresponding action. The two types of creatures, Plant and Herbivore, will have different rules for each of these actions, which are also presented in the table. You should follow these rules exactly, as you game will be graded according to them. In addition to this, here are some details that will help you to implement the game accordingly:

- Each creature should have a health property (type double) that will also affect its decision making (see Table 1 for the specific rules). This property should also have a getter and a setter. The initial and maximum health values should be:
  - A Plant should have an initial health of 0.5 and a maximum health of 1.0.
  - A Herbivore should have an initial health of 10.0 and a maximum health of 20.0.
- Creatures with zero health will automatically be removed by the game logic we provide, so you don't have to worry about getting rid of them.

Table 1: List of Actions and Game Rules

Action Type	**Creature**	**Rule**
Reproduce	Plant, Herbivore	**Plant:** If a Plant's health is larger than or equal to 0.75 & there is an empty cell (direction) around, reproduce. The newly created Plant should have 10% of its parent's health. The parent Plant's health should be reduced to 70% of its original health (20% health is lost in the process). If there are multiple empty cells around, the new plant should be created randomly, at one of the empty cells. Example: A Plant with health 0.75 reproduces; the new Plant should have 0.075 health and the parent's health should be reduced to 0.525. **Herbivore:** If a Herbivore's health is equal to its maximum health (20.0) & there is an empty cell around, reproduce. The newly created Herbivore should have 20% of its parent's health. The parent's health should be reduced to 40% of its original health (Herbivore reproduction is less efficient than Plant reproduction as 40% of total health is lost in the process). If there are multiple empty cells around, the new herbivore should be created randomly, at one of the empty cells. Example: A Herbivore with 20.0 health reproduces; the new Herbivore should have 4.0 health and the parent's health should be reduced to 8.0.
Attack	Herbivore	**Herbivore:** If reproduction is not possible and if there is any Plant around, attack. All of the attacked Plant's health is added to the attacking Herbivore's health (no health lost in the process). The attacked Plant's health should be set to 0.0 afterwards (so that it can be automatically removed by the game logic later). The health of a Herbivore cannot exceed 20.0, as also stated before. Even if a Herbivore is already at maximum health before attacking, it should choose to attack anyway (Herbivores are a bit greedy in this game). Moreover, the Herbivore should move to the position of the attacked Plant. If there are more than one plants around, choose randomly. (Note: A Herbivore should only be able to attack Plants, not other Herbivores. Therefore, you need to check if a Creature is actually a Plant, before choosing to attack.)
Move	Herbivore	**Herbivore:** If there is no Plant around the Herbivore, so attacking is not possible, it should move to one of the 4 adjacent empty cells (up, down, left, right). You can get the available free directions via the `getFreeDirections()` method of `LocalInformation`. Moving makes the Herbivore lose 1.0 health. If moving will cause the Herbivore's health to reach 0.0 or less, it should choose to stay instead. So it should choose to stay if its current health is lower than or equal to 1.0.
Stay	Plant, Herbivore	**Plant:** If A Plant's health is not enough to reproduce or there is no space around, stay and photosynthesize. This action increases the Plant's health by 0.05. The health of a Plant cannot exceed 1.0, as also stated before. **Herbivore:** A Herbivore should choose to stay if no other action is possible due to the above-mentioned rules. This action causes the Herbivore to lose 0.1 health. (So, if it cannot find food soon it will die, but it is better than moving one more cell and facing certain death. After all, It is possible that a Plant will reproduce and a new Plant will appear nearby)

- The coordinate system of the world: Position (0,0) is the top-left square, x coordinate increases from left to right, y coordinate increases from top to bottom. (The example world created in `main` is 50 x 50)
- Each creature should have two properties (x and y of type int) to represent its position in the world. These properties should also have getters.
- The visibility of the creatures are limited to the adjacent cells. For instance, if there is a creature at cell (x,y), this means that the creature can only see the cells (x+1,y), (x,y+1), (x-1,y), (x,y-1). This limited information is automatically provided via `LocalInformation` argument to each creature. From this class instance, you can get the creatures at each direction (`null` if no creature exists at a direction) or get a `List` of free directions to which the creature can *move* or *reproduce*. `LocalInformation` class also contains a static utility method which you may use to randomly select a direction among a list of multiple directions.
- A Plant only supports the actions *reproduce* and *stay*, while a Herbivore can do all four actions. For unsupported actions you may leave the implementations of the respective methods empty (which should never be called anyway, as you are supposed to never choose action *move* for a Plant, for instance).

- The action choice order is as follows for Plants and Herbivores (Table 1 is also organized to reflect this order):
  - A Plant reproduces if it can, else it stays.
  - A Herbivore first checks if it can reproduce, if not it tries to attack, if not it tries to move, else it stays.

Finally, all creatures should implement the `Drawable` interface and consequently the `draw` method, so that they can be visualized on the UI. An example draw method implementation is as follows:

```
@Override
public void draw(GridPanel panel) {
panel.drawSquare(x, y, Color.GREEN);
}
```

This example draws a creature as a green square (it can be applicable for a Plant, for instance). For the example screenshot on page 1, we chose to vary the color of a creature according to its remaining health (healthier plants are darker green, while healthier herbivores are brighter red). This is completely optional, but if you wish to do something similar, you can use `Color(int r, int g, int b)` constructor of `Color` (imported via package java.awt.Color) to set red, green or blue components of a color, as you like.

5 Some Remarks

- You need to consider the usage of appropriate access modifiers (public, protected, private, package-private) for providing the necessary accessibility and visibility. For instance, if you implement everything as public, there will be some penalty. Therefore, be careful about the package structures and class hierarchy while configuring the accessibility issues. Use appropriate access modifiers, keywords (super, final, static etc.) whenever it is necessary.
- Shortly, there will be a partial credit that is specific to the software design.
- Code documentation is important. You need to document your code in Javadoc style including class implementations, method definitions and field declarations.
- The majority of your project will be automatically graded, therefore it is important that you follow the game rules specified in this document exactly.
- Your project is tested and graded with different environment settings. Therefore, you need to consider different scenarios while implementing the project.