Multi-Agent Project

EENG3

You will work in group of 2 to 3 people, from the same Lab Group.
You must submit one project per group.

The objective of this Project is to implement a Multi-Agent Simulation illustrating emergent behaviour. We will start with a famous Cellular Automata called the “Game of Life”, initially imagined by John Conway in 1970, explore variants, and another

Create a new Java project in eclipse. Unzip the zip file on moodle next to this subject, and then from the resulting folder:

- copy everything from « sources » in the src folder of your project
- copy everything from « examples » at the root of your project.

The project files provided contain a main class and a graphical interface. You can already run the project and see the interface, but for the buttons to work, you will need to implement the required methods. Once the loading method is done, you may use the given csv files as examples with known expected behaviours.

All the java files you create should be in the “backend” package (create classes by right clicking on the package to get the new>Class option)
You should modify neither the Main class nor the classes inside the interface package, but do not hesitate to read them to understand their behaviours.

Do not hesitate to use the internet to get information on the game of Life, its known structures and variants. Do consider that the present implementation limits the world to the visible grid, meaning huge structures are out of the question here.

To submit the labwork, you must have pushed all your code and documentation on gitarero.ecam.fr in a repository named “OOP_<groupName>_Project”

The deadline for submission is Friday, May 31, 23h59.

Useful informations:

- to get the square root:

Math.sqrt (value: double**)**: double

- to transform a double into an integer:

int numberInt = **(int) Math.round**(numberReel);
for the int part, with no rounding, use the Math.floor method instead

- to transform a long into int:

int numberInt = **(int)** numberLong

- to obtain an array of Strings, subdivision of a String at a certain character:

String[] myArray = stringToSplit**.split(**characterToSplitFrom**)**
Traditionaly, characterToSplitFrom = **“;”** or **“,”**

- to obtain an integer from its representation as a String :

int numberInt = **Integer.parseInt(**theStringContainingTheNumber**)**

- to get a random value :

Random randGenerator = **new Random();**
//generate a random float between 0 and 1 (uniform distribution)
float floatVal = randGenerator**.nextFloat()**;
//generates a random integer between minVal and maxVal (uniform distribution)
int intValue = randGenerator**.nextInteger(**minVal, maxVal**)**;

Multi-Agent Project

EENG3

Part 1: The resources: GUI and Class Simulator

The Simulator is the main class you will work with. (Meaning central and most important, not "Main" in the java sense. That one is in the default package, and you won't have to modify it. But do take a look and see what it does).

Core concepts of any Simulator are steps and its "world" W.
The world is the space the Simulator… simulates. The steps (also called ticks) are the discretisation of the simulated passage of time.
The simulator will hold data in its attributes representing the state of the World at some point in time, and from that point and rules of evolution, simulates the future states.
This means that to simulate the evolution of the world, the Simulator runs a loop, and at each iteration("step") of the loop, will apply the rules to make the world evolve from a state W(t) to a state W(t+1).

In the case of Conway's Game of Life, the world is a grid with cells that have two possible states: dead or alive. And the 3 rules of evolution are that at each step, for each cell:

1. Any living cell with strictly fewer than two living neighbors dies (referred to as **underpopulation** or **exposure**).
2. Any living cell with strictly more than three living neighbors dies (referred to as **overpopulation** or **overcrowding**).
3. Any dead cell with exactly three living neighbors will come to life. (Referred to as **spreading** or **growth**)

With the implied additional rules:

4. Any living cell with two or three living neighbors continues to live, unchanged.
5. Any dead cell who doesn't have exactly 3 living neighbors stays dead, unchanged.

These are the basic rules, they may evolve later, but the basic underlying concepts will remain the same.

The neighbours of a cell in the grid are the 8 cells surrounding it…
But what happens when the cell is adjacent to a border of the world, meaning there aren't 8 cells around it?
There are two ways to handle this, which shall both be implemented, with the ability of switching between both:

1. Closed border:
   - The cells adjacent to the border are considered to have less neighbours OR the missing neighbours are considered always dead.
   - This means these cells behave differently than the rest, in a way.
2. Looping border:
   - the cells adjacent to the border are considered adjacent to the cells on the opposite border. Meaning the cells of the first and last rows/columns are adjacent.
   - This way, all cells have 8 neighbours, wherever they are.

This is the starting point, but the provided framework will enable you to also add Agents to your world, change the rules for the evolution of the world, and how Agents interact with it.

Part 2: the Mandatory methods

The methods of the class Simulator that you must implement are called directly by the GUI (Graphical User Interface).

List of the expected public methods of class Simulator with missing content:

- void **toggleLoopingBorder**()
  - switches the state of the border, between looping and not.

- boolean **isLoopingBorder**()
  - returns a boolean indicating the state of the border rule (true if the border is looping, false otherwise)

- void **togglePause**()
  - switches the value of the pauseFlag provided attribute of Simulator. While this flag is true, the Simulator only waits and doesn't advance of any step until the flag is back to false.

- void **setLoopDelay**(int delay)
  - setter for the corresponding provided attribute. The higher the delay, the slower the simulation runs. (This delay is the time the Simulator waits between steps)

- int **getWidth**()
  - getter for the size of the world along the x axis
- int **getHeight**()
  - getter for the size of the world along the y axis

- int **getCell**(int x, int y)
  - gets value of Cell at coordinates x,y.
- void **setCell**(int x, int y, int val)
  - sets value of Cell at coordinates x,y to val.

- void **clickCell**(int x, int y)
  - Called when clicking on the GUI at the coordinates provided as arguments.
  - Behaviour will vary depending on the clickActionFlagValue flag:
    - Cell case (default) : toggles the state of the cell at coordinates x,y.
    - Agent case : create an Agent (Sheep provided for example) at provided coordinates.
    - You might want to implement more behaviors for the click depending on different values of the flag (its type can be changed to tolerate more values)
- void **toggleClickAction**()
  - when called (by the corresponding button click in the GUI), toggles the value of the flag clickActionFlagValue.
  - A different value of the flag means clickCell will then behave differently.

- void **generateRandom**(float chanceOfLife)
  - chanceOfLife is the probability any given cell will be alive in a randomly generated "world" (expressed as a value between 0 and 1) this method uses to replace the previous world of the simulator.

- ArrayList<String> **getSaveState**()
  - Returns an arrayList of strings destined to make up a multi-line text file representing the "world" of your simulation in its present state.
  - The returned list of Strings contains one item per line of the file, each String being the content of that line (which you probably want to represent as **semicolon(;)** separated concatenated values).
- void **loadSaveState**(ArrayList<String> lines)
  - Does a reverse process of previous method.
  - Sets the world content to values corresponding to the content of the strings, decoding the encoding of getSaveState()
- ArrayList<String> **getRule**():
  - Works similarly to getSavedState(), but makes a list of lines describing the rules of evolution for the world
- void **loadRule**(ArrayList<String> lines)
  - similar to loadSaveState but for rules
- ArrayList<String> **getAgentsSave**():
  - Works similarly to getSavedState(), but makes a list of lines describing the agents present in the simulation
- void **loadAgents**(ArrayList<String> lines)
  - similar to loadSaveState but for agents
- void **makeStep**()
  - the method that will be called for every step of the simulation. Agent part is provided. Evolution of the world should be organized from here.
- The constructor: **Simulator**(MyInterface mjfParam)
  - Working as is, but you should modify this to add the values to attributes you add to the class, and to add any behavior you wish at the very beginning of the Simulation.

These methods are already declared, but most are essentially empty and non-functional.
You must fill them in. Comments have been left in the code to help.

You SHOULD create freely other methods and/or attributes for the Simulator class, and any number of additional classes (probably one for the World of your simulation, the grid… which contains cells, which themselves might be represented as instances of a Cell class? Some other concepts that should be classes might appear during your thought process....)

Part 2: The first objectives

The primary part of this project is to make the whole interface work, and to have the simulator play the standard Game of Life.

This is the intended behaviour:

- When clicking on the start/pause button:
  - If the world is not loaded, a default world is created, and the simulation starts running, meaning the number of past steps should start to increment, and the world bellow evolve following the rules of Life.
  - If the world is loaded, the pause should be toggled between inactive and active. While pause is activated, the world is maintained in its present state and the step stays the same.

- When clicking on the stop button:
    - The simulation stops, the world is unloaded, and everything on the interface resets to the default value (the same one as on launch)
        - Note: the simulation STOPS at the beginning of a step. So, if paused then stopped, the simulation stays in the paused state until un-paused, and will only stop after.

- When using the speed slider in the interface:
    - The delay between simulation steps will be changed, so that the simulation goes faster the more the slider is to the right.

- When clicking inside the visible world space
    - Does an action depending on the active click action
        - he Simulator will toggle the cell corresponding to the clicked position on the screen.
    - This behaviour enables the user to "draw" a starting state for the world by clicking inside the new world, and then launch the simulation from that state. Or to edit an existing state, live during the simulation, or while pausing it.

- When clicking on the "Toggle Border" button
    - If the looping border has already been set to looping, it is set to closed.
    - If the border is set to closed, it is set to looping.

- When clicking the "Random" button
    - Fills the world with randomly living or dead cells. The probability for each cell to be living is set by the random density slider bellow.

- When clicking the "Save File" or "Load File", the world state as is will be set by the content of a file or written to the content of a file. The interface opens a file selection window similar to the traditional file explorer, to select the file to load from or save to.
    - The simulation state is represented as a text file, where each line of the file represents a part of the simulation world. You are free to use your own format in these constraints, but if you want to open the examples given, you must:
        - Use the lines of the file as representation of either the columns or the rows of the world (if you don't do the same choice as done for the examples, the loaded structures will be rotated, but behave the same… at least with the basic set of rules of Life)
        - Represent the row/column as a string of concatenated values separated by a semi-colon (";")
        - Be able to load a structure of 100 rows and 100 columns (so 100 lines of 100 values separated by semicolons) (this does not force you to have your worlds be limited to this size)
    - Note that there will be no penalty for using a different format in your final version (it can even be an improvement), as long as your project is able to load files generated when saving its own world states…

- You should also be able to load agents and rules in a similar way with the corresponding buttons.
    - Warning: Loading a new world will reset the rules and agents, and pause the simulation

Multi-Agent Project

Part 3: Instructions

Your goal here is to provide a code and save files for rules, world states, and agent populations that develop into interesting patterns, which you should describe how to make appear (and maybe why they are interesting if it isn't self-evident) in the documentation of your code (a Readme.txt in the project will be perfect)

Other than respecting the coding rules already seen during the tutorials and available on Moodle, here are some guidelines:

- If you are not careful, it is very easy to miss some subtleties in the implementation of even the simple rules of the game of Life. So be sure to test the behaviour of well-known structures. Some are given as example files.
  Even if your loading function does not work yet, you might want to draw these structures by clicking once you have this function working (the names glider, r-pentomino, and gosper glider gun should be enough to find these structures on the internet)
  - If these structures don't behave as expected, you probably aren't being careful with the fact that the state of all the neighbours of a cell, when applying the rules to it, should still be that of the step before the one you are currently computing…
- Reminder: this is an OBJECT-ORIENTED Project.
  - This means that it is not because you CAN fit all your data and methods inside the Simulator class that you should, that it would be good reusable code for the industry, or that you would get a good grade doing so.
  - For your information: I expect **AT LEAST** 3 additional classes.
    - More could be better depending on how you structure your code.
  - Do remember that every time you have a named concept that isn't simply a String or a number, it could probably be a class. And that you can (and probably should) make a class for "components" of your code, parts that deal with specific tasks. (such as, for example, loading and reading files… or in your case, encoding and decoding the Stringification destined to read and write to files…)
    - Do not hesitate to move existing methods if they seem to fit better in a logical component you added.
  - This project will be graded depending on several parameters, by order of importance:
    - Structure and Readability
      - Does the code follow the guidelines?
        - No staticity
        - CamlCase
      - Is it understandable and maintainable?
    - Functionality and complexity of the implemented behaviours.
      - Does the code work to do what it tries to do?
      - Does it do complicated stuff well?
      - Is it bug-prone or not?
    - Originality of the added material
      - especially amongst the other groups
    - computational efficiency (memory and time)
      - Choice of the proper data structures and avoiding

Part 4: First Extension: Rules and The Grid

This part is not optional, but you may take it into many different directions, and go either very far or not. (you might want not to go too far here, to focus on part 5)

Some ideas for alternate rules to the Conway ones:

- The most well-known basic variant of Life, HighLife, where the only difference is that dead cells not only become living with exactly 3 living neighbors, but also with exactly 6.
- A prey/predator model, with a prey spreading easily but being “eaten” to help spread the predator.
- Mixing populations: two populations that support and overcrowd each other as if they were one, but each cell “born” is of the dominant type in the 3 living neighbors that gave it life.
- An aging model, where a cell is born as in the standard Life, but rather than dying usually can “age”, changing type (and behavior?) with time.
- A blob that spreads and repairs when attacked (good to combine with Agents that remove items from the field)

Test with different ruleSets, saving and loading them to files thanks to the buttons and associated methods.

The GUI can handle printing cells with values different than 0 or 1. This is why the getCell does not return a Boolean signifying life or death, but an Integer: more states are possible.

Once everything works fine with rules for only life and death, you can choose a variant of the Game of Life with more than two states and implement that one.

- you will probably need to add list attributes to represent the additional rules for the other possible values of Cell
    - In any case, you should probably modify the structures describing the rules (the two ArrayList<Integer> with survive and birth values, in the initial implementation)
- You might also want to change your toggleCell to cycle through all possible states you are using.

Note: only the first 5 positive integers will have distinct colors in the GUI, but other cell values may still behave differently in your simulator and work just fine.

You may also try variants that compute the neighborhoods differently (like using the next layer of adjacency to get to 15 neighboors, or even 24… Maybe cells with different values use different neighborhood sizes?) and use different numbers.

Mix and match variants and make your own special game of Life.
Do not forget to put comments in your code to explain the rules chosen, and to provide example structures showing interesting behaviors of your chosen variant. You may add a text file “README.txt” at the root of your project to indicate what is to be seen in the structures you saved.
You might also want to rework your file loading/saving encryption for something more efficient than representing explicitly every cell… You CAN condense this information.

Part 5: Second Extension : Agents and further

An Interface and code in the Simulator is provided for Agents to be handled.
Agents are entities that exist in the world, on top of a certain cell, can move around, and interact with each other and with the cells.
They are printed as circles of the color provided by their attribute, thanks to the provided methods.
An example of Agent is given to you as the Sheep class.

By changing the rules of the world and the behavior of Agents, you can use the framework to represent easily more complex automatons such as the Langton Ant or more generally Turmites (You may research those examples)

But to start you could base yourself on the existing Sheep, make a Wolf, that kills neighboring Sheep (eating it, replenishing a hunger value?), and add a system of reproduction when Wolf and Sheep meet other Agents of the same species. This would give you a classical prey-predator model.

As Sheep eat the cells from the grid world, you can then simulate a whole environment, and might very well notice equilibriums or extinctions and dominations happening depending on how you tweak the rules and the starting populations.

Play Around!

Part 6: Going Further: The GUI and framework limitations:

For those of you who would like to change the interface and circumvent its limitations (for example to show an infinite grid in which one could move and zoom), you are totally allowed to do so.

Just keep in mind these enhancements are a bonus, not the expected work (which is to have a functioning cellular automata handling different rulesets and possibly interactions with moving agents)