CENG 443 - Intr. to Object Oriented Programming Languages and Systems

Fall 2015 - Homework 1

Agar.oo : AutoCells - Evolution v1.0

Selim Temizer

Feedback : Between November 16th and November 20th, 2015
Due date : November 29th, 2015 (Submission through COW by 23:55)

Agar

Noun \ˈä-gər\

1. A gelatinous colloidal extractive of a red alga (as of the genera Gelidium, Gracilaria, and Eucheuma) used especially in culture media or as a gelling and stabilizing agent in foods
2. A culture medium containing agar

In this homework, we will build an application based on the famous ***Agar.io*** gameplay scenario. We will use object-oriented design principles to build the application, and we will also slightly modify and advance the original game scenario such that the cells are now autonomous, and they evolve and learn new behavior types as they gain mass.

Basically, we will need the following classes to implement our application:

- ***Sugar***: Represents pieces of sugar particles in the petri dish. Sugar particles are non-living things and *cannot move*. Sugar particles can be drawn as randomly oriented squares in random colors. The length of each side of the squares might be randomly picked between 8 and 12 pixels.
- ***Organism***: Represents hypothetical microorganisms that are much smaller than cells. Organisms sometimes *move around slowly in random directions*, and sometimes *stand still* to rest. Organisms can be drawn as filled circles in random colors. The radii of organisms might be randomly picked between 5 and 7 pixels.
- ***Cell***: Represents the entities that we are most interested in, in our application. Cells eat sugar particles, organisms and other smaller cells when they come into close contact. The application starts with a certain number of young cells in the petri dish. Initially the cells only know how to *feed on the sugar particles and organisms* around them, and they also *stand still* to rest from time to time. As young cells gain mass, they gain additional abilities by learning how to *move randomly* and also how to *move linearly*, evolving into what we call **Roamer** cells. Roamers then evolve into **Evader** cells by gaining additional mass and learning how to *move away from the closest cell* that is larger than them, and also how to *lose some mass* in order to move faster. Finally, evader cells learn how to *chase the closest cell* that is smaller than them, and complete the evolution by becoming **Hunter** cells. The speed of a cell is inversely proportional to its mass. Cells can be drawn as circles with

transparent membranes in random colors, and in sizes that are proportional to their masses. Additionally, a small line can be drawn inside each cell to indicate the facing direction of that cell. Cells should also be marked with text indicating the name of the cell, current mass of the cell, number of food (sugar particles and organisms) eaten, number of other smaller cells swallowed, and an indication of the current behavior the cell is exhibiting.

- ***Display***: Extends *JPanel* and consists of three logical parts. On one of the three parts, dynamic textual statistics about the application and cells should be displayed. On the second part, there should be a logo image (any logo that you may design for your application works). And the third part should display the state of the petri dish in real time.
- ***Environment***: Brings together application parameters, instances of entities (sugar, organism and cell), and other utility fields / methods that are required for running the application.

However, we will also use the following software design patterns that will require us to extend our basic design:

- ***Factory Method / Abstract Factory***: A food factory will create either *sugar* or *organism* type entities (other additional types are also possible). Make sure the part of the application that creates the food is unaware of the type of available food (i.e., demonstrate proper use of the factory method / abstract factory design pattern in your implementation).
- ***Strategy***: Will provide a behavior to each of the entities in the application. As a minimum, 7 types of strategies that are mentioned above should be prepared: *StandStill*, *GrabFood*, *MoveRandom*, *MoveLinear*, *AvoidLarger*, *LoseMass* and *ChaseSmaller*. At any given time during the run, each entity will be executing one strategy picked from this list, according to their abilities. Entities should not be aware of the types and numbers of available strategies. It should be possible to easily extend the system with additional strategies in the future.
- ***Decorator***: Each cell will be marked with colors to denote whether it has gained Roamer, Evader and Hunter abilities. Make sure that the code fragment that decorates the cells is outside the *Cell* class implementation, for good object oriented design.

Using the above design patterns, we should be able to:

- Prepare a petri dish with some predefined number of young cells and food particles (each food particle has an equal chance to be created as a sugar or as an organism instance). Sugar particles should be assigned the *StandStill* strategy. Organisms can be assigned either of *StandStill* and *MoveRandom* strategies at random. Young cells can only use *StandStill* or *GrabFood* strategies at the beginning. Later on as they evolve, they will be capable of demonstrating other behavior as described above. Each strategy takes effect for a random number of steps (say, between 10 and 100 turns), and then finishes. Once a strategy finishes, another one from the set of strategies available for the entity will be assigned to that entity. When grading, we need to clearly see the changing strategies during the flow of the application. Also make sure that the code fragment that assigns strategies to entities is outside the source code of that entity (so the entities are not aware of the number and the names of available strategies, demonstrating good object oriented design when using the strategy pattern).
- Decorate cells at run time: As a cell's mass reaches the predefined thresholds for becoming a roamer, an evader and a hunter, rings of colors green, blue and red should be added when drawing a cell. Here, for correct usage of the decorator design pattern, make sure that the

Cell class is not aware of the fact that it will be decorated (once or more) somewhere else in the application source code.

- As a suggestion, ideally a *stepAll* method in the *Environment* class might ask each entity to apply their strategies for one step, then might check which food particles (and maybe cells) are consumed, spawns additional food to replace the consumed food, then reassigns strategies to entities whose strategies have been completed, and decorates the cells that need to be decorated. This *stepAll* method might be called over and over (with a screen refresh and sleeping for a few milliseconds in between) to create the animation.

A screenshot of a sample implementation is provided in the following part of this document as a reference. We are also required to professionally **design** and briefly **document** our application. For this purpose, we will need to use a specific UML tool (StarUML) to first design the class diagram for our application, generate stub code from it, and then fill out the methods and missing implementation details. (You need to submit the UML document from StarUML application together with your source code). A detailed (but not complete) class diagram is given in the following pages to serve as a starting point and a guideline for you.

What to submit? (Use ***only ASCII characters*** when naming all of your files and folders)

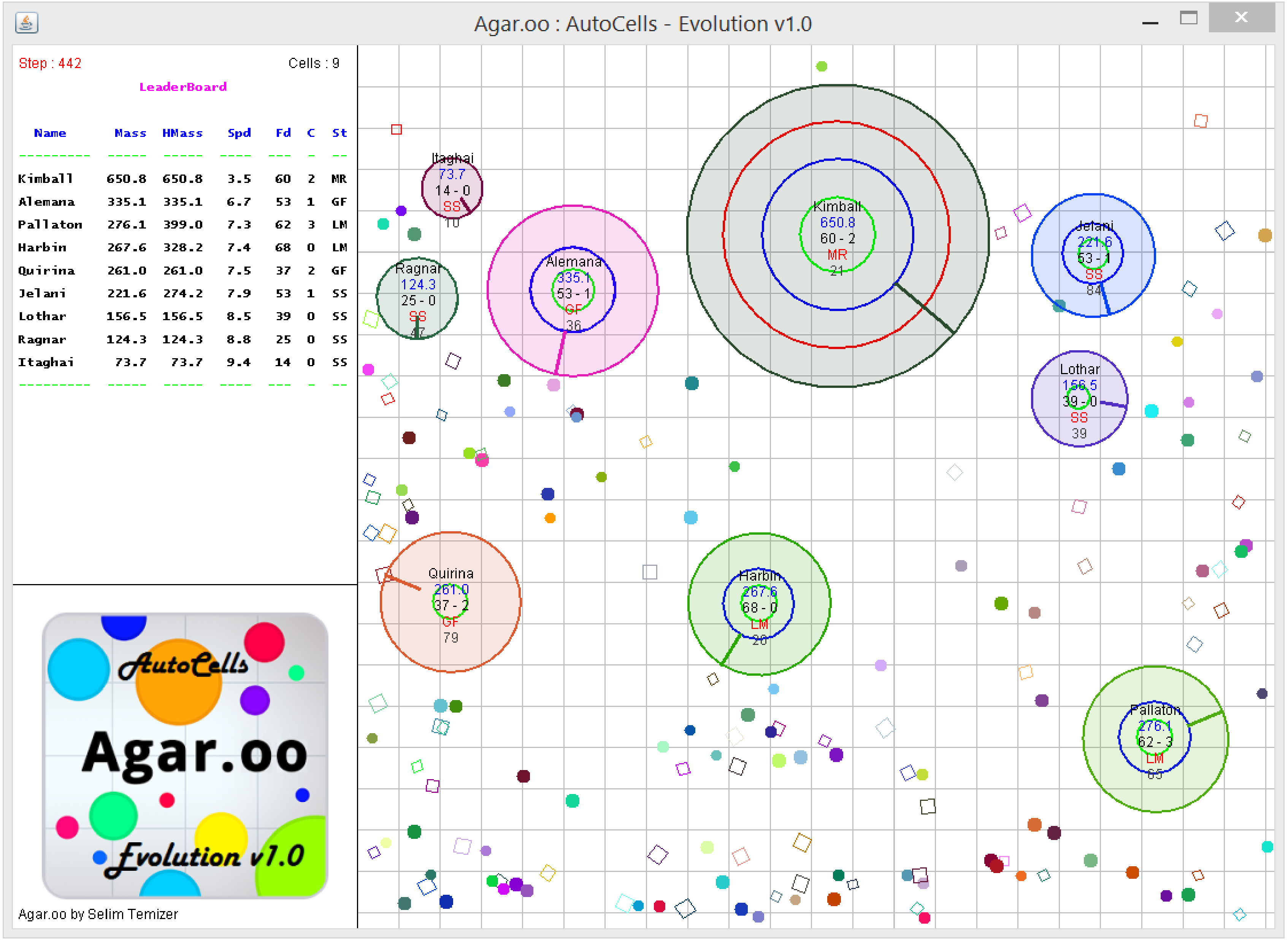
1. A single ".uml" file from StarUML v1, or an ".mdj" file from StarUML v2. Name it as "***Design.uml***" (or "***Design.mdj***"). Start with an EMPTY project. Just have a single class diagram (and maybe optionally a statechart diagram and/or an activity diagram for bonuses). Don't first write the code and then reverse engineer it to get the UML. This homework aims to teach you how to go from a carefully designed UML to the code (this is what you will be hired to do for the rest of your software design careers).
2. Any other documentation that you would like to add (in a directory named "***Docs***").
3. Your source code (all in a single directory named "***Source***"). Do NOT use packages. Do NOT submit binary code or IDE created project files. Just submit your ".java" files. I should be able to compile and run your code simply by typing the following:

```
C:\...\Source>  javac  *.java
C:\...\Source>  java  AgarRunner  <any documented parameters that you might need>
```

Zip the 3 items above together, give the name <ID>_<FullName>.zip to your zip file (tar also works, but I prefer Windows zip format if possible), and submit it through COW. For example:

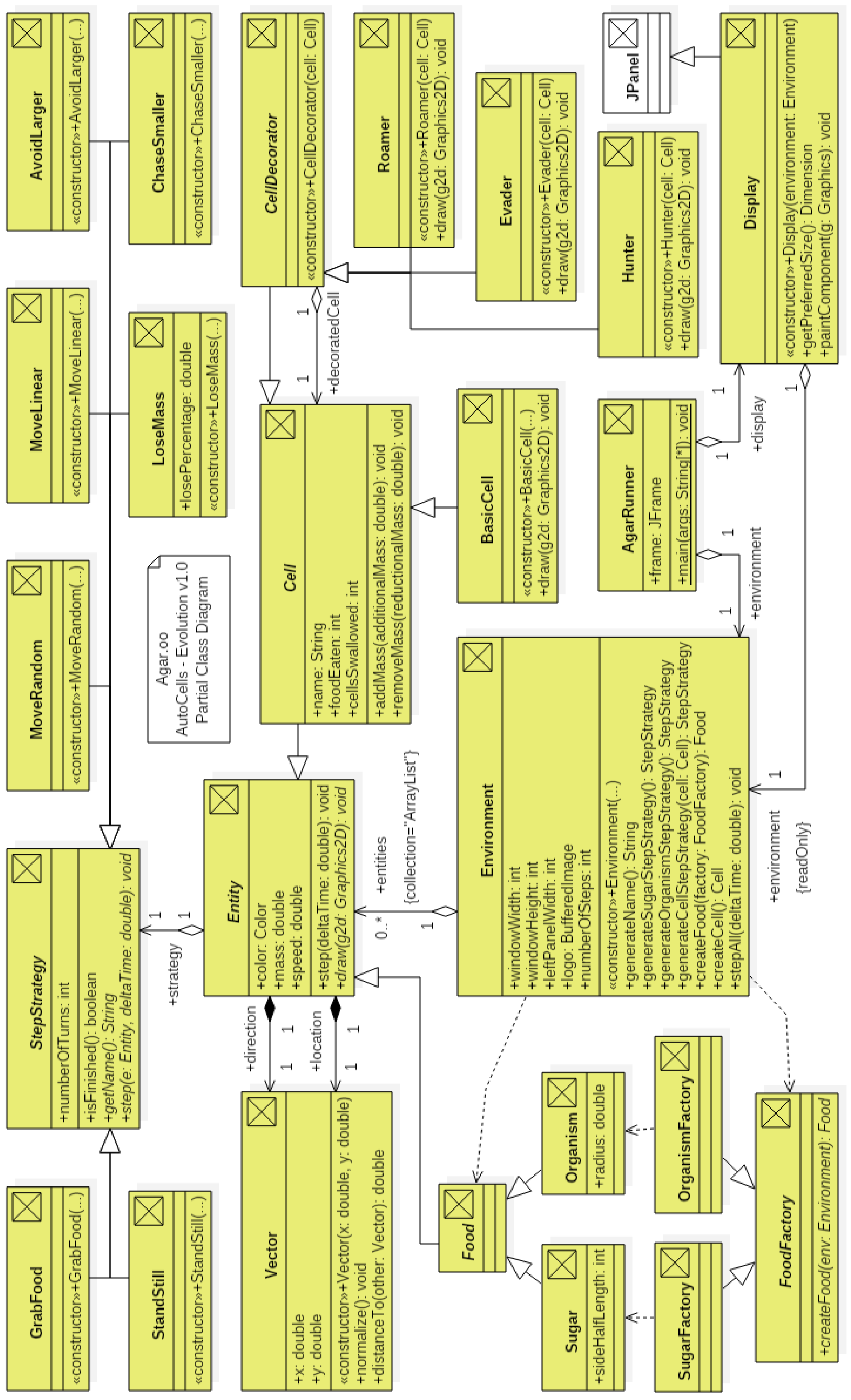
e1234567_SelimTemizer.zip

There are a number of design decisions (example: read all parameter values from editable configuration files) and opportunities for visual improvements (example: draw much nicer figures) and creative extensions (example: additional strategies) that are deliberately left open-ended in this homework specification. We have enough time until the deadline to discuss your suggestions and make further clarifications as necessary. There will be bonuses awarded for all types of extra effort. Late submissions will NOT be accepted, therefore, try to have at least a working baseline system by the deadline. Good luck.


Agar.oo : AutoCells - Evolution v1.0
Itaghai 73.7 14 - 0 SS 10
Ragnar 124.3 25 - 0 SS 47
Alemana 335.1 53 - 1 GF 36
Kimball 650.8 60 - 2 MR 21
Jelani 221.6 53 - 1 SS 84
Lothar 156.5 39 - 0 SS 39
Quirina 261.0 37 - 2 GF 79
Harbin 267.6 68 - 0 LM 20
Pallaton 276.1 62 - 3 LM 65
AutoCells
Agar.oo
Evolution v1.0
Agar.oo by Selim Temizer

```
Step : 442                          Cells : 9
                 LeaderBoard

  Name        Mass   HMass   Spd   Fd  C  St
---------    -----   -----   ----  ---  -  --
Kimball      650.8   650.8   3.5   60  2  MR
Alemana      335.1   335.1   6.7   53  1  GF
Pallaton     276.1   399.0   7.3   62  3  LM
Harbin       267.6   328.2   7.4   68  0  LM
Quirina      261.0   261.0   7.5   37  2  GF
Jelani       221.6   274.2   7.9   53  1  SS
Lothar       156.5   156.5   8.5   39  0  SS
Ragnar       124.3   124.3   8.8   25  0  SS
Itaghai       73.7    73.7   9.4   14  0  SS
---------    -----   -----   ----  ---  -  --
```


Agar.oo
AutoCells - Evolution v1.0
Partial Class Diagram
AvoidLarger
«constructor»+AvoidLarger(...)
ChaseSmaller
«constructor»+ChaseSmaller(...)
MoveLinear
«constructor»+MoveLinear(...)
LoseMass
+losePercentage: double
«constructor»+LoseMass(...)
MoveRandom
«constructor»+MoveRandom(...)
StepStrategy
+numberOfTurns: int
+isFinished(): boolean
+getName(): String
+step(e: Entity, deltaTime: double): void
GrabFood
«constructor»+GrabFood(...)
StandStill
«constructor»+StandStill(...)
CellDecorator
«constructor»+CellDecorator(cell: Cell)
Roamer
«constructor»+Roamer(cell: Cell)
+draw(g2d: Graphics2D): void
Evader
«constructor»+Evader(cell: Cell)
+draw(g2d: Graphics2D): void
Hunter
«constructor»+Hunter(cell: Cell)
+draw(g2d: Graphics2D): void
JPanel
Display
«constructor»+Display(environment: Environment)
+getPreferredSize(): Dimension
+paintComponent(g: Graphics): void
Cell
+name: String
+foodEaten: int
+cellsSwallowed: int
+addMass(additionalMass: double): void
+removeMass(reductionalMass: double): void
+decoratedCell
BasicCell
«constructor»+BasicCell(...)
+draw(g2d: Graphics2D): void
AgarRunner
+frame: JFrame
+main(args: String[*]): void
+display
+environment
Entity
+color: Color
+mass: double
+speed: double
+step(deltaTime: double): void
+draw(g2d: Graphics2D): void
+strategy
+entities
{collection="ArrayList"}
0..*
Environment
+windowWidth: int
+windowHeight: int
+leftPanelWidth: int
+logo: BufferedImage
+numberOfSteps: int
«constructor»+Environment(...)
+generateName(): String
+generateSugarStepStrategy(): StepStrategy
+generateOrganismStepStrategy(): StepStrategy
+generateCellStepStrategy(cell: Cell): StepStrategy
+createFood(factory: FoodFactory): Food
+createCell(): Cell
+stepAll(deltaTime: double): void
+environment
{readOnly}
+direction
+location
Vector
+x: double
+y: double
«constructor»+Vector(x: double, y: double)
+normalize(): void
+distanceTo(other: Vector): double
Food
Organism
+radius: double
Sugar
+sideHalfLength: int
OrganismFactory
SugarFactory
FoodFactory
+createFood(env: Environment): Food
1