Chris Bernt, Scotty Shaw, Marcus Cain - Team 19

Design Document for Cell Society Project

## Introduction

In this project our team's goal is to develop software that can run any kind of cellular automata simulation. This involves a grid of cells in which every cell has a state, and the grid updates the cells based on pre-decided rules. The user should be able to choose which simulation they want to run, and enter an XML file with the starting conditions for the given simulation. In addition, the user should be able to easily add a new simulation by programming a new Java class.

One of the primary design considerations for this project is therefore flexibility in how different simulations are handled. The general running of the simulation should be general enough that the main code stays the same for all simulations; only user input determines which simulation should be run. The type of cell that is used is therefore open for extension, and the code that actually runs the actions and updates the grid is closed to modification. Everything that is different about each simulation is taken care of within the various Cell subclasses.

## Overview

The general design of the program is split up into three categories: the main method that asks the user for input and parses through the XML file; the simulation loop that handles the animation and runs the simulation at a given frames per second; and the types of Cells which include rules for updating specific to each Cell's simulation.

The program was split up in this way because each one of these sections can do almost all of its own work within its own classes, with as little an amount of data as possible passing to other classes that do more unrelated tasks. For example, the user’s input for the grid size and the parsing of the XML file for the initial conditions of the simulation will be in a similar section as they both serve the same purpose of setting up the simulation, as well as the grid size, because it checks the validity of cell positions in the XML file.

One important feature of the main class will be reading the XML file, which we may create a new class for. One design consideration is to have the new XML parser class parse through the information and overriding the cells states that are initially created. This will ensure that all of the cells in the MxN grid (which the user decides in the UI) will have a state. All of the cells that the user decides to define in the XML file will be given their corresponding state (according to the XML file) and all other cells, not specified by the XML file, will be given a default state.

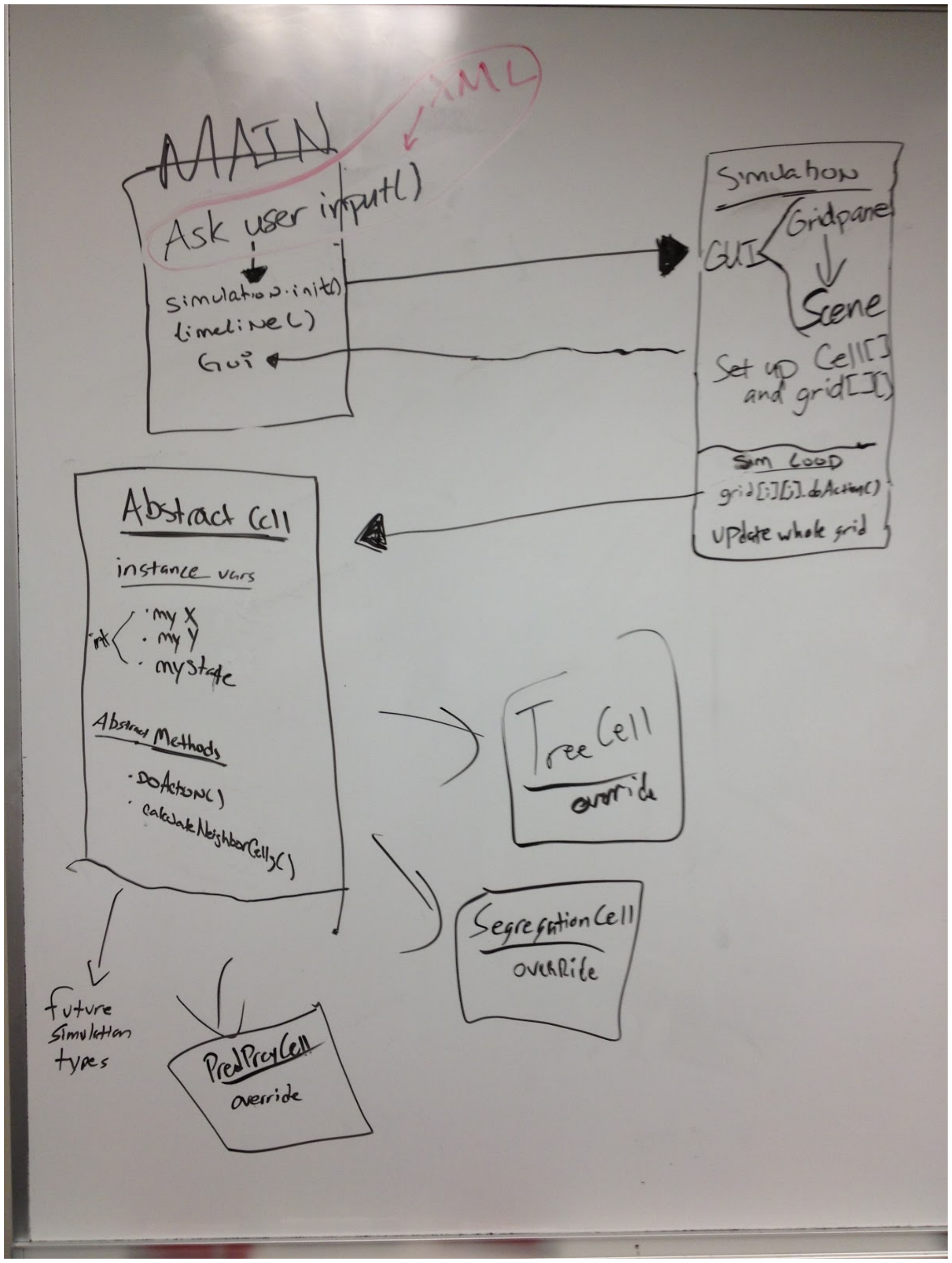
The simulation loop handles the important task of continually updating the Cell[ ][ ] (back end) and the corresponding animation of the GridPane (front end), which is the JavaFX class we chose to use to represent the cellular automata grid. We chose to represent the different possible simulations as subclasses of an abstract superclass Cell. We chose this design as Cells in the grid are really the only elements actually doing anything during the simulation loop; very little information needs to pass to Cell in order for the elements to make decisions about how to behave to their situation with their neighbors, and little information needs to pass from Cell to the grid to update the animation.

This also allows new simulations to be added extremely easily; one would simply need to create a new Cell subclass and write in the rules of their simulation, then add it to an Array in the main class, which would then add it to the list of possible simulations in the user input screen. The loop of the simulation will also be able to handle any subclass of Cell, since it will be calling an abstract Cell method to update the grid. The three general functionalities will therefore work together in the following way: the main input getter will set up the conditions for the simulation loop to begin, which will then call the update method for each Cell in the grid which is uniquely defined for each type of simulation. In terms of class organization, all of the different Cell subclasses will be put in one package so that they will be easy to find and edit.

The way we are choosing to represent the grid on the back end is through an Array of Arrays, or a 2D Array. The type of this Array will be the type Cell. This will be the primary data that is passed to Cell which allows them to make their decisions about how to update their states for the next generation. We chose a 2D Array based on the assumption that the ground for a cellular automata simulation will be a rectangle, but felt that the strengths of a 2D Array in representing the grid abstractly outweighed the potential drawbacks from our assumption.

One data structure that came up in conversation is a graph. If the program specifications were to include non-rectangular grids or shapes, a graph would allow us some flexibility to move in non-cardinal directions when our program is calculating the number of neighbors each cell has. Of course, the difficulty with this data structure is whether our program uses breadth-first search or depth-first search to iterate through each Cell when calculating the number of the neighbors and updating the state of each Cell. It seems to us that the runtimes would be the same for a 2D array and a graph because we have to iterate through every single Cell, but a 2D array would be simpler because it allows a double for loop.

The following image represents graphically the series of calls and how the main sections of the program are organized.

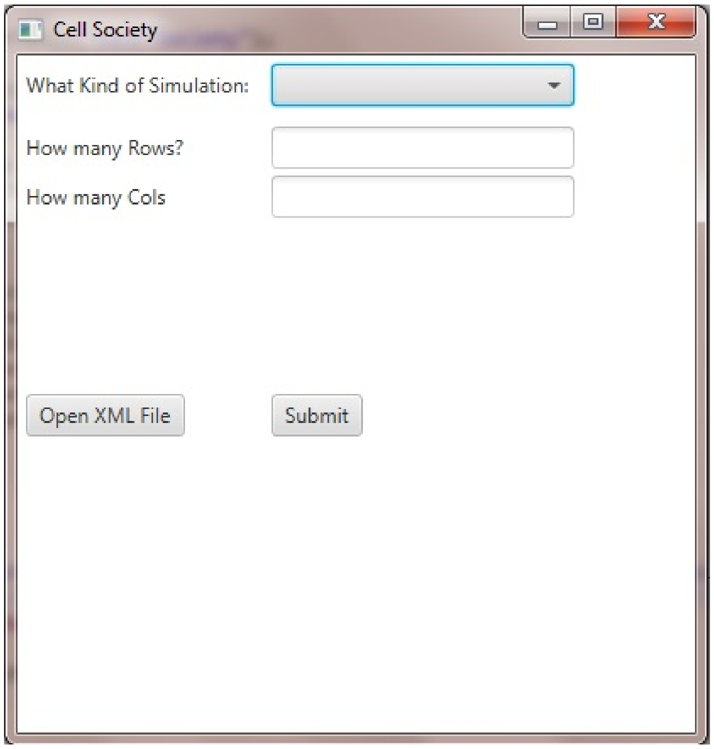


## User Interface

The user will be able to interact with our program through a pop-up window that asks which simulation (Burning Trees, Segregation, Predator/Prey) the user wants to run, how large they want the grid in use to be, and for an XML file that sets starting conditions. We designed the structure of the user interface to automatically display the total options to run each type of simulation. For example, if another developer/user wanted to implement a new simulation type; one would just need to add this information is one place (adding a new cell in the array of types of simulations); the user interface will pull the necessary information from this array, thereby always ensuring that the total options of game simulation types are shown in the user interface for  choosing by the program user.

A drop down menu will ask the user which simulation they want to run, and there will be two text fields (one for the number of rows, and one for the number of columns) for them to enter their desired grid dimensions. Finally, there will be a text field to display the file the user has selected to run, which can be located with a "Open XML File" button. The "Open XML File" button will open a secondary pop-up window that lets the user locate and choose a file to upload, and a "Submit" button will let the user send the information once they feel ready for the program to operate.

An nearly completed pop-up window below depicts most of what has been described.

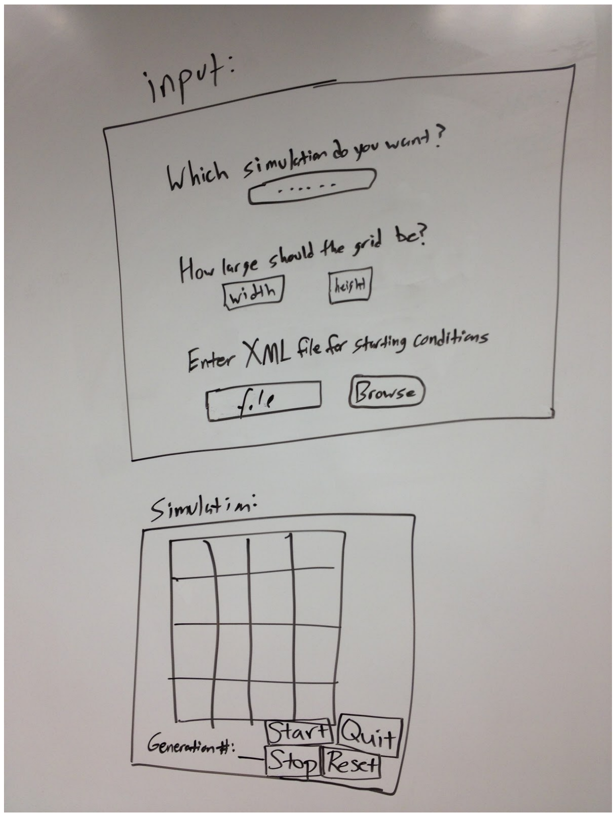


If the user does not select a simulation, the program will ask them to do so. If they don't enter an integer between 1 inclusive and some upper bound, they will be notified of the error and asked to enter an integer within that range. This can be conducted by a try/catch exception, and we will have to test and find that upper bound. As for the XML file, our program will use an XML parser that loops through and only takes in information from objects tagged as <Cell>. That will eliminate any non-Cell entries that may exist within the file.

As for the information itself, our program will also check and see if any Cells are placed outside of the bounds of the indicated size, have been set to invalid states, or are duplicating positions. If that is the case, the user will be notified of where the error is in the XML file so that they can edit it. If the file is corrupted or empty, the user will be notified and asked to try again. Of course, anytime the user does not submit an XML file for our program to process, they will be asked to do so.

During the simulation, the user will see a new pop-up window that displays the grid with the specified dimensions and the cells in their specified states and locations. The window will display information, such as the number of generations the simulation has processed, and buttons (Play, Pause, Reset, Quit).

A drawing of the input and simulation pop-up windows is shown below.


input:
Which simulation do you want?
How large should the grid be?
width
height
Enter XML file for starting conditions
file
Browse
Simulation:
Generation#:
Start
Quit
Stop
Reset

## Design Details

The first section of the general program design is that which takes in input from the user and sets up the initial simulation conditions. The first thing this module does is to introduce a graphics window which prompts the user for the simulation type, grid rows, grid columns, and an XML file which contains the initial conditions. One consideration was to handle the size of the grid based on the initial inputs of the XML file. However, this idea was quickly thrown out considering that the user may only specify states for a subset of the total gridspace, and thus the gridpane would only show the subsets, rather than the entire grid space. These variables are used to initialize both the back end 2D Array of the Cell types specified by the user and the front end GridPane. The XML file will be handled by a separate XML file reader class which takes in the XML file, reads it, and returns back a List of Cell Objects with their desired x positions, y positions, and initial states. All of the error handling in user input is also handled in this section of the program. For example, a user must enter a valid row number (any integer within the range of 1 inclusive and an upper bound, which will be determined later), or it will be rejected by this module.

The second module is that which runs the loop that updates the grid and animation. After the first module has read in the XML file and created the starting grid, the second module will be ready to operate. Assuming a 2D array, a double for loop will iterate through each row (outer loop) and column (inner loop) and check how many neighbors of each Cell are in specific states. That Cell’s updated state will be stored temporarily until the double for loop is completed, and then every Cell will be updated to its new state. This will be considered one full generation, and the simulation loop will run through the 2D array to check on the neighbors of each Cell again.

The third module is the actual abstract Cell superclass and all of its subclasses that implement a specific simulation. Cell will have the instance variables of x position in the grid, y position, current state, and next state. The need for next state is that when one cell wants to change its state, it should wait until the end of the current generation to switch or else its change will affect the nearby cells, which is not what is wanted for these example of cellular automata. A Cell can thus change its next state to be equal to its current state when the whole grid is updated all at once.

Cell will have four abstract methods: doAction, getNeighbors, toString, and makeNew. doAction is the method that contains the rules for the simulation. In the example of Conway's Game of Life, this method will contain the rule "If a cell if surrounded by too few cells, it dies". Since this method is different for all simulations, it is abstract. getNeighbors returns an Array of Cells of the Cells surrounding this one, which may be different depending on the simulation. In the forest fire example, neighbors were only considered cells directly to the North, South, East, or West of a cell. In the segregation example, neighbors were all 8 cells that surround the one cell in a square. That is why this method is also abstract. toString is simply a method that returns a String description of the simulation type, for use in the drop down selection menu. It is abstract as each subclass needs it own description. makeNew is a method that is more specific: it returns a new Cell of the same type as the class it is defined in, with the given parameters. The need for this method arose when we thought about implementing the different Cell types in an Array that contained all the simulation types. These Cells needed to have no parameters, so we established a nullary constructor for Cell and all subclasses. When we instantiate the 2D Array that represents the grid, we grab the desired null instantiation out of the Array with the Cell type options, and call makeNew with the desired x, y, and state parameters on the Cell type, and add it to the 2D grid. In this way we can fill the Array with Cells with different parameters, without changing any code for different Cell subclasses. This method is abstract as it needs to return the type specific to the class it is in.

## Design Considerations

Our group decided relatively quickly that we wanted some sort of superclass of simulation types. Specific simulations would extend that superclass and an Array of those instantiations would be stored in a class that runs the animation. One of the biggest conversations our group had about this design was whether to have that superclass be of Simulation type (essentially a class that knows all of the rules of a simulation and carries them out), or of Cell type (each Cell subclass would contain the actions it should take during the simulation). Although we initially thought the first option would be the best, since all of the rules could be organized into separate classes and manipulate the one Cell class, we eventually opted for the latter version. This was because we believed that the cells are the only things that are really doing anything during the simulation; to have a separate class would require passing cell data to that class, it figuring out what the cell should do, and then passing that data back to the cell. Our thoughts are that it does not matter what type of cell automata simulation is run; in essence a simulation is a simulation - we can take care of each type of simulation by the subclasses of Cells. One downside of this approach is that Cell is no longer a standalone entity that exists by itself and can be manipulated freely. The types of Cell have to have specific information and methods that obey its particular simulation.

Our group additionally discussed how we should represent the grid's information on the back end of the program. The two options were essentially using the GridPane information (the class we are using to display the animation) or using a separate 2D Array to hold the grid's information that is the reflected by the GridPane. The advantage to the former is that the information can all be held in one location, but it would require passing a significant amount of data to the Cell classes for them to update their states. The advantage of the latter is that only essential information is presented and that it can do more complex operations like storing future positions. Its disadvantage is that it requires extra information in the main loop class. We decided upon the 2D Array design, since we felt that it would be easiest to pass through to the Cells. Of course, both of these solutions rest on the assumption that the CA platform will be a rectangular grid of some sort. If it were a hexagon, for example, we would have to change this implementation to one that was a Graph and connected all of the Cells' neighbors with instance variables.

## Team Responsibilities

Although our group plans to meet consistently and work together on the general coding of the program, we do have plans for dividing up the work to some extent. Chris will be in charge of the reading of user input and the XML parser and passing that data to the main loop class. Scotty will be in charge of developing that main loop class and ensuring it runs properly with all types of cells and that it updates the grid properly. Marcus will be in charge of developing the particular Cell simulation types. All of the roles assigned to each team member is subject to change. Since we are meeting daily to progress through this project, we may find that one role may be suited for one team member rather than another. This will ensure that progress will be made in a timely manner.