Introduction

The ideal goal of our team for the Cell Society project is to write a Java program that is flexible enough to cope with any rule of CA that it might encounter by taking advantage of the commonalities that exist between the different types of CA that would exist. We also want the code to be extendable with minimum effort on the part of the extender. We hope to achieve all this while using a clean coding style that adheres to modern coding standards.

We intend that all the code we write will be closed by the end of our coding sessions. If all goes according to our current plan, all that an extender would need to do to add on new features is to extend one class and add the name of that extension to a file.

We have also decided to write our code using a Model-View-Controller paradigm. We believe that separating the concerns each piece of our code has to deal with, will maximise the extendability and readability of our code should further maintenance be needed.

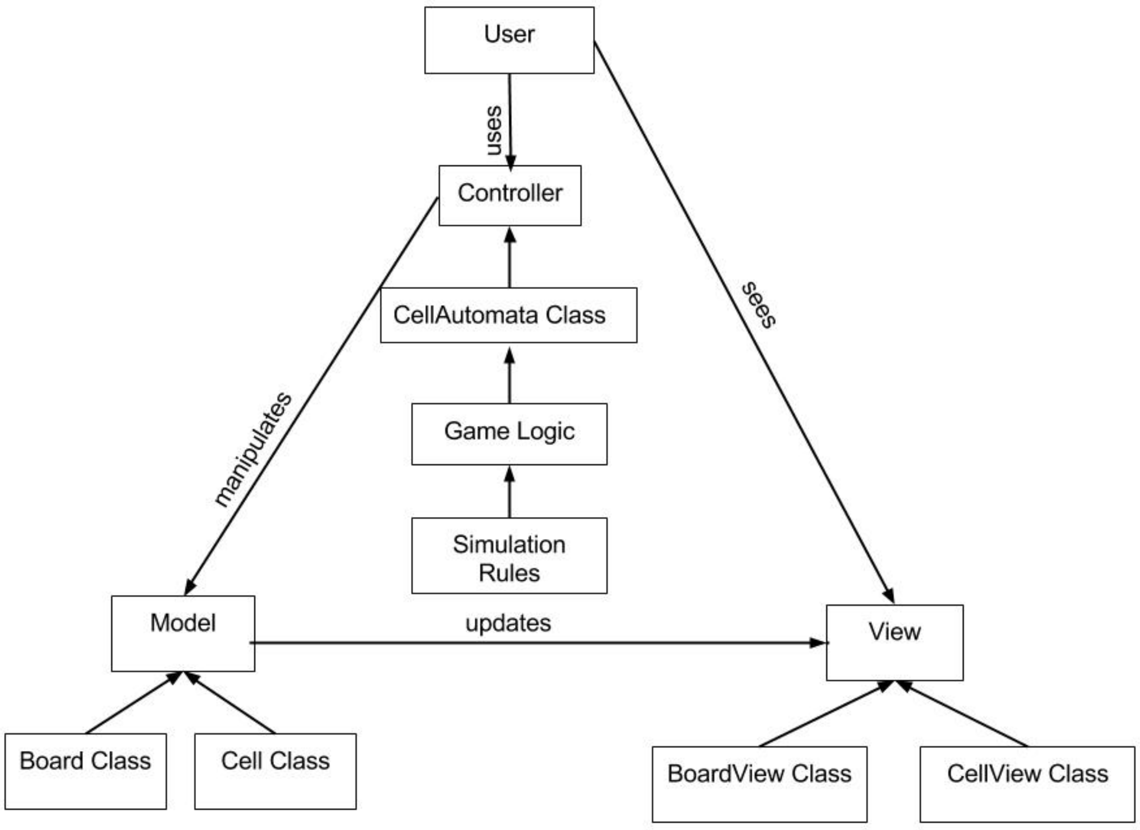
Overview

We opted to divide the program into three main parts: the model, view, and controller. The model will include all of the Cell and Board objects in the game, while the controller will control how each Cell’s state changes as the simulation progresses. The view will display the grid and images/colors for each cell.

The model will contain the Board and Cell classes. The Board will contain a constructor to initialize the board, as well as a fillBoard method. The fillBoard method will take in an ArrayList of cells, and initialize the board based off of that list. The Cell will contain a constructor that takes two integers that indicate the Cell’s x and y coordinates on the board and an ArrayList indicating which of the multiple possible states it has. The ArrayList’s length will be determined by the number of states in the game. Each index will represent a possible state. The ArrayList will consist of all zeroes, except for a one at the index which will represent its current state.

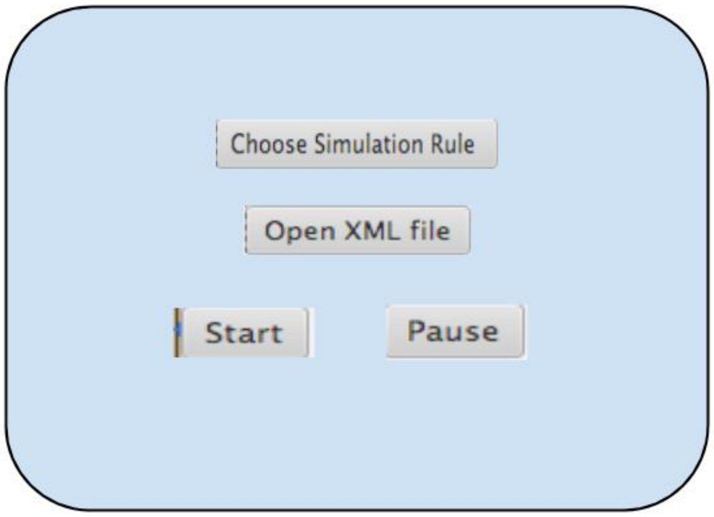
For controllers, we will have an abstract class CellAutomata which will initialize a set of rules for the game. The CellAutomata class will contain a method to parse the XML file and send the board information to Board’s fillBoard method. It will set up the animation and then run an instance of SimulationRules. SimulationRules will be an abstract class for all game modes. It will be up to the user to implement the required methods to run the automata. SimulationRules will contain one ArrayList to keep track of the next board’s state. There will be a method to update the states of cells by iterating through all of the cells and, based on the selected game's logic, decide the state for the next stage. The cell's state is changed, then it is added to the ArrayList with its new state, if its state does not change, the same cell is added. The SimulationRules will then reinitialize the board using the states from the ArrayList.

Lastly view will consist of images/colors for the board, and for each cell.



User Interface

The user interface will be a file selection box that gives the user the options of selecting the type of cellular automata they wish to run and one of several well named XML files that will provide initial values for all of the cells in the board, as well as a start button to begin the simulation, a pause button, and a stop button. The user interface will also return errors if the user tries to pass in a non-XML file, the XML file is empty, or doesn't initialize a valid grid.



Design Details

The Cell class will be initialized with its x and y coordinate in the grid, as well as its initial state in the form of an ArrayList. The Board will be initialized with its height and width, creating the 2d array.

The SimulationRules will be capable of changing the states of each cell depending on the logic of the simulation. As the controller it will have the ability to modify the Cells and Board (model) based on the logic specified in the game. It will also be able to update the view, based on the changes to the model.

Our design centered around maximizing flexibility. Our Cell class was designed to work for any type of simulation. Its ArrayList for states can work for any number of different types of states that a simulation might require. To add additional simulations, the user just needs to create a new class which extends SimulationRules and implement the methods in it; no alterations to the cell or board classes are needed.

Design Considerations

An initial consideration we had was how to store the cell state. At first we considered using a boolean to represent an alive and not alive state. We also considered using a counter to keep track of each cell's alive neighbors. However, this only worked for the Board of Life game which only involve two states of cells: alive and dead. Because we wished to maximize flexibility, we opted to store states in an ArrayList which would allow for more than just two states (i.e. Spreading fire). The trade-off of using the ArrayList is that it lowered the efficiency of simulations with only two states, because we could no longer store alive neighbors in an ArrayList. However, it opened up the Cell class to be usable in any simulation.

Team Responsibilities

Dimeji: Board model, and implementing spreading fire
Sandy: Cell model, and implement Schelling's Model of Segregation
Ethan: CellAutomata, SimulationRules, implement Predator-prey

Everyone: View, XML files