**CellSociety Plan - Team 02** Greg Lyons, Kevin Rhyne, Chase Malik

**Introduction**

As a team we are working to develop a Java program that is capable of running various 2D Cellular Automata (CA) simulations. CA simulations are modeled by a grid of cells, with each cell having a certain state and updating its state based on defined rules regarding its current state and its neighbors’ states. CA simulations begin with each cell initialized to some given state, and they are capable of modeling many different types of phenomena. Perhaps the most well-known example in the programming world is Conway’s Game of Life, which updates each cell between being “alive” and “dead” based on the number of its neighbors that are alive or dead. Other well-known examples include Schelling’s model of segregation, Spreading of Fire, and the Wa-Tor World model of predator prey relationships. Our primary design goal for this project was not simply to implement these models, but also to allow for the addition of many other CA simulations.

An ideal CA simulation system has a flexible design so it is as easy as possible to add new simulations to the program. We focused on the principle of making design open to extension but closed to modification. In this sense, the program is made so that its capabilities can be easily extended for the rules of a new simulation, but the mechanisms of the program itself do not need to be modified. We utilized a system of abstraction, so that adding a new CA model to the system only requires implementing one abstract method with the model’s logic rules for updating cell states. We also employed inheritance, extending JavaFX classes to simplify the process of creating our visual display. In our program, the cells are set to initial states based on input from an XML file. The XML file also specifies the type of simulation to be modeled.

**Overview**

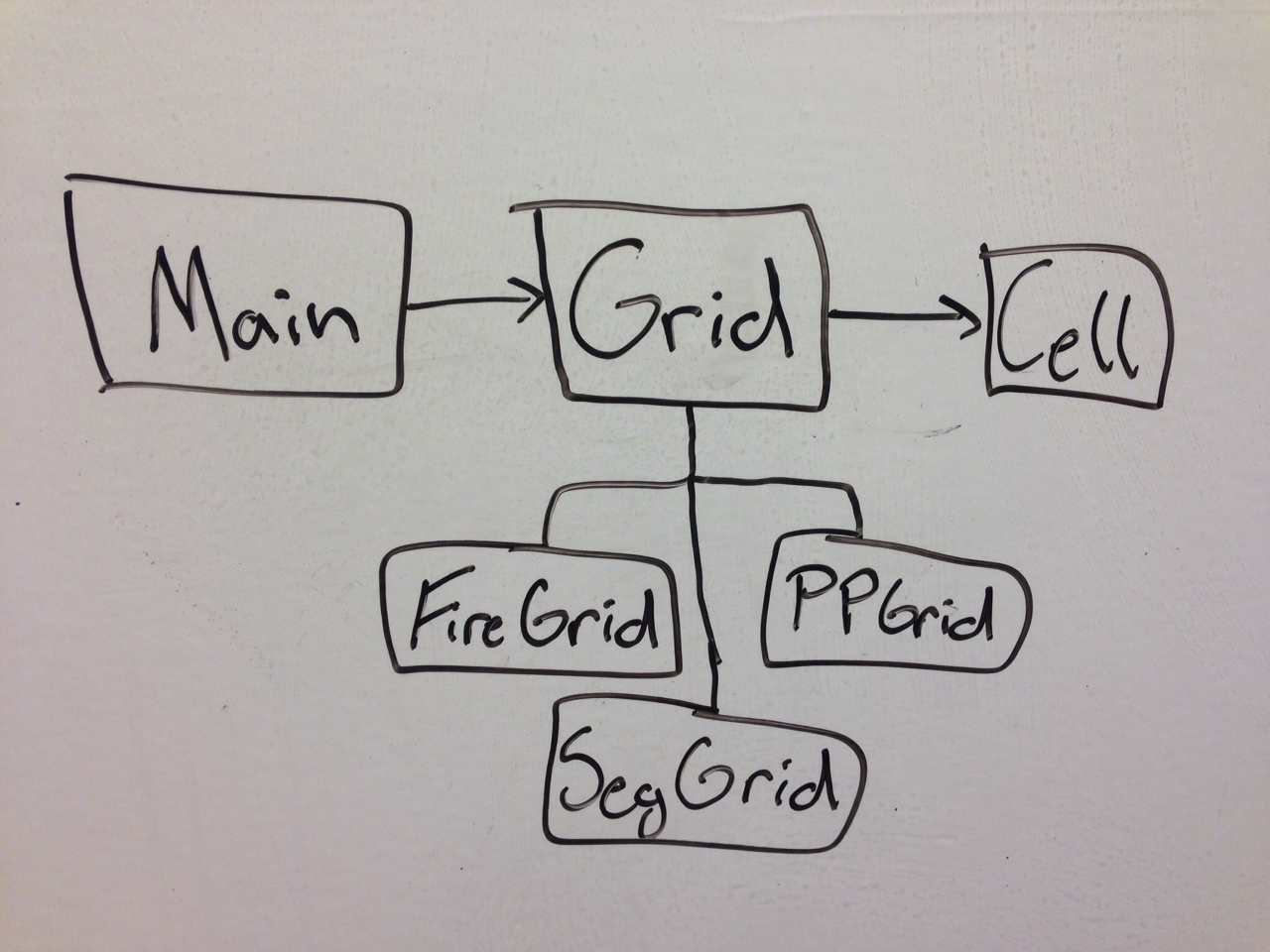
In order to solve the CA problem, we broke it up into the three separate issues - loading the data, creating the 2D grid based on the appropriate rules, and displaying the grid. The first issue is addressed by the first module. The Main class loads the XML data into Java and parses it into two pieces of information. It creates a variable that represents the type of CA model being tested (fire, segregation, etc.) and a 2-D array of integers to store the current state of each cell. This module then calls on the second module to create a grid based on the current array.

The grid module is comprised of one abstract class and as many subclasses are required (currently three). Making this class abstract allows for other types of CA models to easily be added as extensions of it without any modifications to the current code. The abstract class will have the following methods - updateDisplay, updateStates, updateCell, initialize, start, pause. The constructor for the class will take the initial array and set it to the instance variable currentArray. The initialize method will deal with setting up the user interface and displaying the initial state of the 2D grid on the screen. When the start method is called on a key press, it sets up an Event Handler that continuously calls updateStates and then updateDisplay. These two methods will use the two private instance variables (arrays) called currentArray and futureArray. currentArray stores the states of the cells before they are each “simultaneously” updated, and futureArray stores the new states of each cell when they are updated.

The updateStates method loops over all of the cells in the currentArray. It then runs the method updateCell on each individual cell. This method is an abstract method, whose implementation varies based on the type of CA model being run. Thus, each subclass of Grid (FireGrid, SegregationGrid, etc.) will need to override this method with its own method. These model dependent methods will vary in many different ways, as some will have more than two states, while others will depend on fewer neighbors. However, regardless of the complexity in calculating the future state, the method updateCell edits the appropriate cell in futureArray to reflect its future state. This future cell is usually the same cell, but for some CA models may differ (Segregation). Thus, the final result of the updateStates call is to have a completed futureArray.

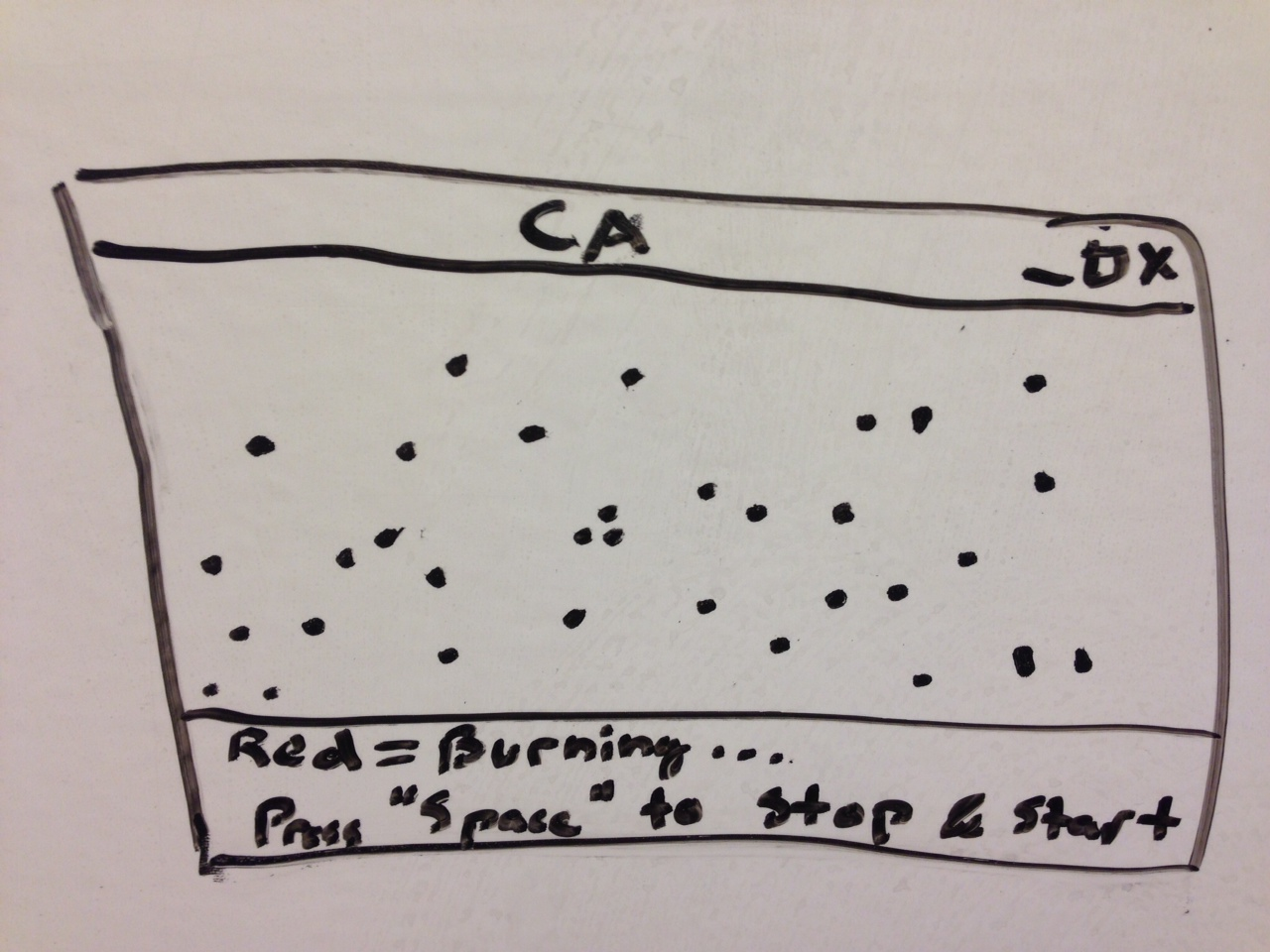
The updateDisplay method will then loop over the futureArray and create new Cells at the appropriate location with the appropriate color based on its state (entry). This method will rely on the third module of our design - the Cell class. The Cell, which extends the JavaFX Rectangle class, will create a new rectangle with a color based on the current state of the cell. The state is passed as a parameter when constructing a new Cell. The updateDisplay method then adds these new shapes to the display. After this addition, the method sets the currentArray equal to the futureArray, as all of the pending changes will have been displayed at this point.

The final method in the Grid class is pause. The pause method stops the EventHandler from updating the states and updating the display. Thus, the display does not change, and the actions occurring in the CA model pauses. This method will be called based on a key listener.



**User Interface**

For now, our interface will be on the more simplistic side. A window will pop up, prompting the user for the XML document. After that is submitted, the pop-up disappears, and the CA simulation is initialized. There will not be any visible buttons to control the simulation. However, there will be a text field that explains the significance of the colors in the simulation (e.g. that red means the tree is burning) and details the keyboard controls (e.g. “Press space to start & stop”). If there is an error with the XML document, or if a non-XML file is submitted, this text field will instead notify the user that something has gone wrong.



**Design Details**

The first module of our program, the Main class, has the primary purpose of loading data from the XML file and using said data to determine which CA model to simulate. As of right now, the parsing of XML data is a bit of a black box for us, but we know that Java has tools built in for such tasks and we have faith in our ability to employ them in a way that will give us the data we need. From the XML file, the two pieces of information extracted are the type of simulation to be run (parsed and stored as a String), and the initial states of the cells, stored in an integer array. From there we will use a switch statement, using each of the simulation types as cases. Each of these cases will contain a statement that creates a new Grid instance using the implementation for that type (new FireGrid, new SegregationGrid, etc). The Main method will also set up a Scene, Stage, and Timeline for the game and will call the initialize() method from the Grid class.

The Grid class is an abstract class that has one abstract method to be implemented. The rest of its methods are defined, which makes implementation of new CA models easy (only one method must be implemented in the new class that extends Grid). The Grid class has several instance variables. These include variables for the display such as a Scene and Group, integer arrays currentArray and futureArray to hold the states of the cells, and a Boolean to keep track of whether or not the simulation is running (for pausing functionality). The Grid class has an initialize() method that sets up the display, declares instance variables, and displays the initial state before calling the start() method when a key is pressed. The start() method will create an Event Handler that takes care of looping the simulation by repeatedly calling updateStates() and updateDisplay().

The updateStates() method works by iterating through currentArray (using a nested for loop for the 2D array). At each element of the array, it calls the abstract method updateCell(int i, int j), where i and j represent the position in the array (and on the grid). The reason for calling the abstract method within another defined method is that all of the Grid subclasses will need to iterate through a nested for-loop, so that loop might as well be already implemented in the superclass.

The abstract updateCell method is where the heart of the simulation behavior lies. Each simulation will be a subclass of Grid that implements this updateCell method in a different way. Based on its parameters and its access to the instance variable currentArray, the updateCell method can access each of the current cell’s neighbors’ states. Based on its own state and these neighbor states (neighbor is defined differently for some simulations), the cell’s value in futureArray will be set to whatever its new state must be. The purpose of the two separate arrays currentArray and futureArray is so that when the cells are updated to new states in futureArray, the old states still exist in currentArray so that the changes can be “simultaneous”.

After updateCell has been called on every cell in currentArray, and all of the updated states have been entered into futureArray, we re-assign currentArray to be a copy of futureArray. Then, the updateStates() method is completed, and the handler will call updateDisplay(). updateDisplay() first clears the current grid display and then adds the new one. It adds each of the new rectangles individually as instances of the Cell class, which extends Rectangle. When these Cells are constructed, they take the grid position coordinates (i and j) as parameters along with the current state. In the Cell class, this state is interpreted as a fill color for the Rectangle (based on a switch statement using state as the case). The Cell’s position is set using the coordinates and multiplying by appropriate scaling based on the number of Cells in the grid (the size of either array). Each of these new Cells is added to the Group, which displays it on the screen.

The handler method keeps running through updateStates() and updateDisplay(), unless it comes across the condition that futureArray and currentArray are the same after all updates have been made (this condition would be checked as a Boolean right before making currentArray to be a copy of futureArray). If it comes across this condition, the simulation has reached a point where no more state changes are possible, and it will end.

**Design Considerations**

Our group had a lengthy discussion about whether to make the Grid or Cell class abstract. We decided early on that we would need something to be abstract in order to make our program flexible and accommodate for the possibility of adding more, unknown CA simulations. Initially, we chose to focus on Cell. We decided each cell would need to store it’s X and Y coordinates as well as its state/value within the simulation. However, this decision led to some discussion about getters and setters because with these variables it seemed very likely that we would need Grid to call a getState method in cell. And thanks to our discussion in class, we know to avoid getters and setters. Despite this initial setback, we continued our discussion. We would also need cells to update themselves based off of their neighbors. To accomplish this we considered having the grid class loop through all of its cells, check the cell’s neighbors, and pass that to the cell for it to update. However, we realized this design would not work as desired, because cells would change based on its neighbors, but before the other cells in the loop. Therefore, the grid would update based on the position in the array instead of “simultaneously”. In order to address this issue, we would need Cell to have a parameter where we could feed it the current grid. Thus, to try to avoid overcomplicating Cell, we decided to switch to an abstract Grid class.

With an abstract Grid class, the various CA simulations will manifest themselves as subclasses that extend Grid (FireGrid, SegGrid, etc.). The Grid class has every method that the simulation will need in order to complete its duties. The simulation-specific grid classes override the updateCell method with the simulation-specific logic (e.g. in the FireGrid, updateCell will implement how fire spreads from cell to cell). Having Grid be abstract also allows for easier implementation of “simultaneous” cell updating (see **Overview** or **Design Details** and how we use currentArray & futureArray). We ultimately chose this design because we thought it allowed better flexibility.

**Team Responsibilities**

* Chase
  + Primary
    - SegGrid
    - Main
  + Secondary
    - Greg’s
    - Cell
* Greg
  + Primary
    - PPGrid
    - abstract Grid
  + Secondary
    - Kevin’s
    - Cell
* Kevin
  + Primary
    - FireGrid
    - Main
  + Secondary
    - Chase’s
    - Cell