**DUKE UNIVERSITY**

CS 308: Software Design and Implementation

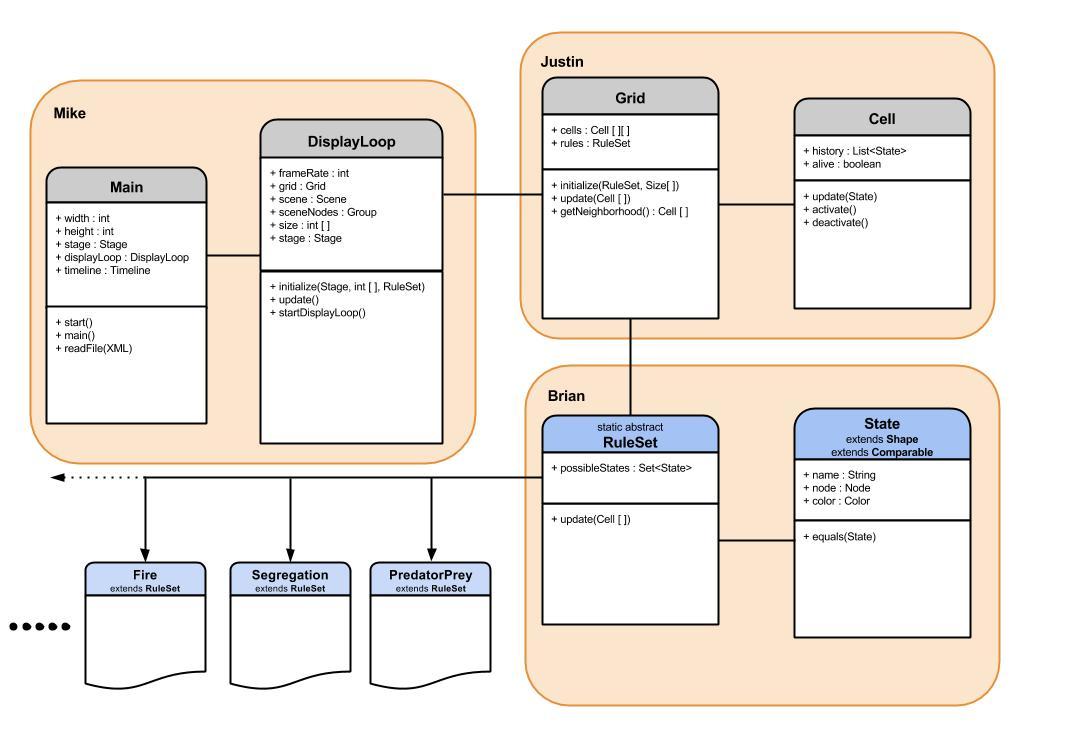
Cellular Automata : Cell Society

Group 8: Brian Bolze, Justin Carrao, Mike Zhu

**Introduction**

Our team is writing a program using JavaFX to animate a wide variety of 2D grid Cellular Automata simulations. Cellular Automata model natural phenomena as a grid of interacting cells - each cell represents an individual in the model (e.g., a tree in a forest fire simulation, an animal in a predator-prey simulation) and contains a set of finite states (e.g., alive, sick, dead). Cells change their states in response to its neighbouring cells, according to the rules of the simulation. For example, in a forest fire simulation, a healthy tree will switch to the “burning” state when it surrounded by two or more burning trees. Though the rules underlying their function are relatively simple, Cellular Automata can be used to accurately model high complex systems.

We are designing a highly modular architecture that allows simulation of many types of CA. The program will handle CA of different sizes by implementing a resizable grid structure. By abstracting the rules behind the cells states and state changes conditions we make the program highly extendable - to accommodate new types of CA, one simply needs to write new rules classes that extend the abstract superclass. The game will then incorporate the states and algorithms of the new ruleset used to run the specific simulation. By shifting all but the core methods to these abstract rulesets, we allow easy extension of the program’s capabilities without requiring modification of source code.

**Overview**

Our project design currently consists of six classes. It starts with an abstract cell class, which will consist of a few information variables (animation type, color, on/off, etc.). Our current plan is to have the Cell class be as generic as we can possibly make it, with all state changes and Cell behavior (which vary from CA to CA) shifted to Ruleset. This allows implementation of new CA simulations by just adding a single subclass. The behavior of the subclasses of our Cell class will be determined by two other classes: a State class and a Rules Class. In trying to keep the Cell as simple as possible, we have external methods (specified by a given Ruleset) modify Cell properties, namely its State.

The State class will contain information such as if a cell is currently active (animated), what it’s currently doing (color, for example), and the neighbors of a given cell. This state class will be passed to the Cell, which will be continuously storing its state history in a list. We aren’t sure in which cases state history will be important for updating cells, but we came across some examples of cellular automata for which history was important.

The RuleSet class will implement the rules for animating the grid of Cells. RuleSet will be an abstract class with subclasses that will be called and used. Presumably the set of rules will be read in from the XML file, which determines which specific subclass of RuleSet that our Grid class will use. It will probably have a **createStates()** abstract method that is implemented in each subclass of RuleSet. Each subclass will determine the behavior of cells, specifically by giving them start states, describing how they transition between states and based on what factors they do this (neighbors, history, probability, etc.). Ruleset will also have an abstract **update()** to be implemented by a Ruleset specific to a given CA. Update handles changing the states of each Cell object in the Grid.

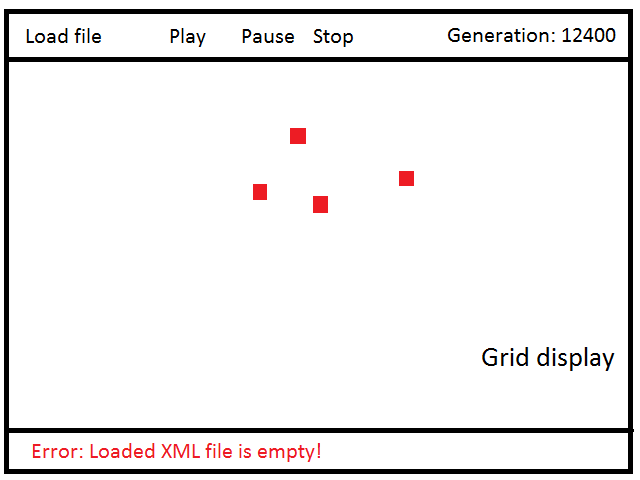
The Grid class will contain an **initialize()** method which initializes instances of Cell objects and places them into a two-dimensional array, using RuleSet to initialize start states. It will contain an **updateCells()** method for looping through all the cells in the grid and changing their state based on the factors defined in whichever subclass of RuleSet was instantiated in the Grid class. We are planning to use various fields to keep track of statistics relevant to the grid **(cellsOn, cellsOff, cellsOfCertainColor, cellsSatisfied, cellsUnsatisfied, etc.).**

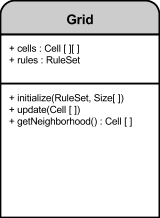
Next we plan to use a DisplayLoop class that basically (on a high level) has the same function as the GameLoop class that was given to us for the last project: Game. It will make a KeyFrame and determine the frame rate. It will initialize an instance of the Grid class. We might implement KeyEvent handlers for Pause and Play functionality in the DisplayLoop class, but at the moment we are unsure if this is a better option than implementing a toolbar for Play/Pause/Stop instead.

Our main class is where we plan to initialize some DisplayLoop object and manage the animations. At a high level, think of the BallWorld class given to us from the Game Project. However there will be more implementation in terms of User Interface than there was in the original BallWorld class.

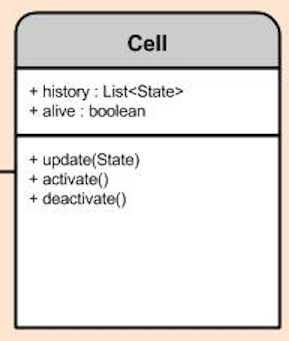
**User Interface**

The UI will display the 2D grid containing all interacting cells. Cells will be basic rectangles that can change color with changes in state. The size of the window will depend on the dimensions of the CA as specified in an XML file. Users will have simple ways to interact with the program through the UI: a load button to create a new simulation from an XML file, and controls to play and pause the simulation. All such controls will be implemented using the JavaFX Button class. Basic simulation output will be reported on the top right corner (namely the generation number of the CA) small output window on the bottom will report input errors(e.g., loading an empty XML file, loading a non-implemented CA simulation, etc.).

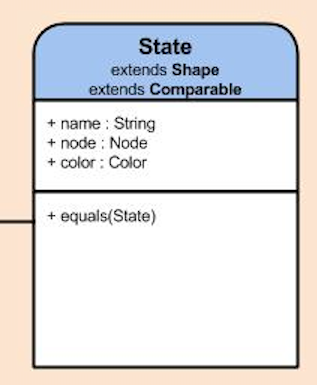
*Simple representation of the planned GUI.*

**Design Details**

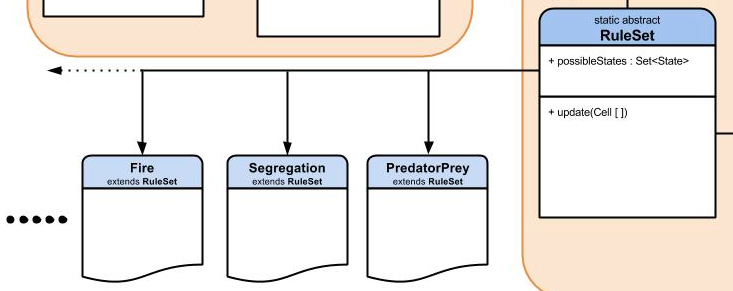
In the Grid class we are going to implement a 2D array filled in by cell objects. the method for getNeighbors() will return an array of the current cell and the 9 surrounding cells, and which neighbors to take into account will be determined by RuleSet. Grid also contains a Ruleset object that dictates the rules for Cell state change; different implementations of Ruleset allow simulation of different CA algorithms. In order to update Cell states, Grid contains a method to iterate through all stored Cells and apply its Ruleset object’s update() function to each Cell. Ultimately, Grid must also return the 2D grid to the DisplayLoop to enable animation.



In the Cell class we are currently planning to use a Stack to keep track of the state history of each individual cell. This is necessary for certain CA simulations that track a Cell’s previous state. We want Cell to extend Shape so that we can easily use square/rectangles/etc. to populate our grid class and easily change their color for animation.

The State class extends Shape, and acts as storage for both a Cell’s state (the exact number and name of possible states varies with each CA) and its appearance for each state. Each state has a variable storing its name (alive, dead, sick, etc.) and a method for changing it’s fill color. A list of all states possible for a given Cell is initialized by the program’s current Ruleset and stored in a list in the Grid. This allows easy access to the list of possible states and setting those states onto a Cell. 

The Ruleset class contains two abstract methods to be implemented by Rulesets for specific CA simulations (e.g., FireRuleset, SegregationRuleset, etc.). One, createStates() creates all the possible states for a Cell in a particular simulation, storing it in a List. Since different simulations have different numbers of possible Cell states, this design choice allows our program for function for any given CA. The second method, update(), is the meat of a Ruleset subclass. update() contains the rules that govern how a Cell responds to external stimulus and changes its state. Since different CA simulations contain different definitions of “neighbors” (e.g., the four above, below, left, and right, or all eight adjacent cells), each update() implementation will need to specify which adjacent cell to access. This method will contain conditions to handle state change for each state that a Cell is in. The use of abstraction in Ruleset allows easy extension of our program: to implement a new CA, simply write a new Ruleset implementation, specify that Ruleset in the XML, and load it into the Grid.



**Design Considerations**

The team deliberated on whether individual Cell objects or the Grid that holds all Cells should be responsible for updating Cell states. Having the Grid handle all updates would simplify the program and remove dependencies between Cells and the Ruleset. Since the Ruleset is instantiated in Grid from specifications stored in the XML file, having the Grid run the Ruleset’s specific update() function on each Cell stored in Grid does not require passing the Ruleset into Cells. However, this method effectively turns Cells into passive data containers, holding only their state, a variable modified by external methods. Ultimately we decided on having the Grid handle all updating as it removes dependencies and keeps the code modular.

Another discussion on reducing the role of Cells involved whether or not to shift all state initiation and behavior to the Ruleset class. The argument against this decision is again that it turns Cells into little more than passive data containers. However, shifting all behavior to the Rulesets makes extending the program incredibly easy. Instead of making new Cells and Rulesets for each new CA simulation, one simply needs to make a new Ruleset.

The team also considered several ways to implement Cell states. One option was to store states as simple strings, and have several update methods, one for each state (e.g., if a Cell is dead, update will act on it differently than if it is alive). This makes updating Cells very simple, as comparing strings is easy (within update, a conditional statement if(Cell.state.equals(”alive”)). A separate method would then check a Cell’s state each frame and change its color accordingly. We instead opted for a second approach in which states are implemented as a separate class, State, which extends Shape. This class stores both a Cell’s State and its appearance at a given time, removing the need for a separate update method. We do need to have State implement comparator here to use conditional statements to implement unique update() methods for each state, but the combination of state and appearance made this the more attractive option.

**Team Responsibilities**

Brian Bolze - RuleSet Class and State Class

* Secondary Responsibility is to learn about interacting with XML Files
* Secondary Responsibility is also implementation of some specific RuleSet subclasses

Mike Zhu - Main Class and DisplayLoop Class

* Secondary Responsibility is also to learn about interacting with XML Files
* Secondary Responsibility is also implementation of some specific RuleSet subclasses

Justin Carrao - Grid Class and Cell Class

* Secondary Responsibility is to learn and collaborate with Mike and Brian about interacting with XML Files
* Secondary Responsibility is also implementation of some specific RuleSet subclasses

We will all collaborate on implementing subclasses of Ruleset to incorporate new Cellular Automata simulations to our program. To start with, we will divide up the Fire, Segregation, and Predator-Prey models, with each team member implementing one of the three. We all want to learn how to use XML files, and none of us know how yet, so we would like to all learn and collaborate on that aspect of the project.