#### **Team 16: Cell Society Plan**

*Team Members:*

Rahul Harikrishnan (rh151@duke.edu)

Kevin Li (sl260@duke.edu)

Abhishek Balakrishnan (anb17@duke.edu)

#### **Specification**

Cell class - will contain pointers to the adjacent cells (instances of Cell); these pointers will be stored dynamically as a list so that we are not having an explicit # of Cell pointers that constitute neighbors. This class would include state of the cell, list of neighbors, current cell location, and whether the current cell has been updated in the current round)

Cell Manager - will initialize the 2-D grid of cells, and then set parameters of each of the cells Cell manager will loop through all possible modes

Extensions to the Cell class will have:

* Update which factors in the logic for that specific mode
* State List, which represents the possible states for that mode

**Introduction**

The program we are developing is a framework for running cellular automata simulations. Cellular automata (CA) is a collection of cells, with different states, on a grid that evolve over time following given rules based on the states of neighboring cells. Cellular automation can be used as a computational tool that helps us model different real-life systems. Our program will read in an XML file as input. This file will specify what type of simulation should be run as well as the initial parameters for that scenario. Once the mode has been determined, an appropriate class will be run in order to apply the correct turn-based logic to the new dynamic system.

One of the major design goals is flexibility - our design will determine what type of cellular automata to run based off of a keyword in the XML file. We are also using inheritance to craft the various simulation modes from a parent cell class. This makes it a lot easier for a new simulation to be programmed to deal with new input data. Lastly, each individual cell will be aware of its neighbors, which will make interactive logic easier to implement. This implies that from a broader software architecture standpoint, attributes such as states and positions of cells will need to be publicly exposed, while the ability for cells to update themselves can be left as a closed method.

**Overview**

Our cell automata program will be able to model three distinct systems – the spread of wildfire, predator/prey relationships, and segregation in populations. The basic design of our program consists of four major classes – CellWorld, CellLoop, CellManager, and Cell (see Figure 1).

The Cell Class and subclasses

At the core of the program is the class Cell. The Cell class is an abstract class that defines all the shared attributes of cells. The Cell class consists of four different instance variables – myNeighbor, myX, myY, and myUpdated.

* myNeighbor is an ArrayList that contains a list of all of the cells that constitutes as neighbors to the existing cell. We chose to implement this variable as an ArrayList in order to give flexibility in cases where the amount of neighbors a cell has differs.
* myX is an integer value that holds the x coordinates of our cell in the grid.
* myY is an integer value that holds the y coordinates of our cell in the grid.
* myUpdated – is a boolean that tests for whether this cell was updated this turn and should therefore be left alone. This is to prevent updated cells from updating again in the same turn.

The cell class consists of only one unimplemented abstract method, update. The update method differs depending on what type of cell is used. It contains the so-called “rules” that determine the evolution of the cell and nearby cells.

The Cell class is implemented in three different ways, that each corresponds with the three model systems – FireCell, EcoCell, and SegCell. These three implementations all have one additional instance variable, myState. myState is a list of states that the cell can be in. We decided to leave this in the subclass instead of the Cell Class because each subclass has different states that are not necessarily shared amongst classes. In addition, some subclasses will have additional instance variables. For example, EcoCell will have myLifeSpan to signify how long the cell has stayed in a certain state. Each class also implements the update method. Here is the general logic that each subclass’s update class follows:

InitialCell

The InitialCell class is a structure that contains he following instance variables myState, myXPos, myYPos. It stores data parsed in the readAndParsed method for use in the CellManager initialize method (classes explained later).

CellManager

The CellManager class is the class that keeps track of and initializes the initial state of the grid. It has one instance variables – myGrid.

* myGrid is an ArrayList of ArrayLists of Cells that contains all of the cells in the model. We chose to store the grid as an ArrayList of ArrayLists because it allows for flexibility of grid size and easy access to Cells based on coordinates.

The cell manager also contains two methods – initialize and setNeighbors.

* The method initialize will take in parameters to generate a grid of blank grid of cells that it will then fill in with initial state information. This method takes in four parameters, modelType, xDimension, yDimension, and initialState.
* Using the string ModelType, initialize will use compare using if statements to determine which type implementation of Cell to generate.
* Using xDimension and yDimension, the initialize function will know which size to make myGrid.
* initialState consists of an Arraylist of InitialCell. Initial Cell contains the state of a Cell as well as its x and y position. Initialize will use this data to make changes to Cells in myGrid. For example, if an InitialCell contains the instance variables ON, 5, and 5, the initialize method will set the element at (5,5) in myGrid to on.
* The method setNeighbors will traverse myGrid and update the myNeighbor ArrayList in each Cell. It takes in a single parameter, modelType. ModelType is a string that contains what model is being followed. Using if statements, the setNeighbor method will generate neighbors. This was done because each model has different rules for what constitutes as neighbors and we wanted to incorporate that into our design.

CellLoop

The CellLoop class is very similar to GameLoop class in our first assignment . It has seven instance variables myCellManager, myCellType, myInitialState.

* myXDim, myYDim, myInitialState, myCellType are all instance variables that contains the initial state of the grid. These variables are the variables that are inputted as parameters into the initialize function of CellManager whenever the program starts or resets. We chose to make these instance variables in order to avoid parsing every time we reset the program.
* myCellManager is an instance of the CellManager class that is used to call methods within that class.
* myTurns contains the number of turns needed for fish and shark to breed. It is public static final and can be used by the EcoCell class to determine whether fish and shark can breed.

The CellLoop class has eight methods, generateScene, start, updateCells, updateDisplay readInAndParse, pause, continue, and restart.

* The generateScene method creates a scene to display. It creates a grid pane, which can be used to display colors based on the information in myGrid. It will generate buttons that will pause, continue, or restart the model by triggering the appropriate method (pause, continue, restart – see below). It will also generate a space for inputting an XML file that will trigger the method for reading and parsing the file (readInAndParse – see below). Potentially, if we wanted to add buttons to control certain parameters such as days required for fishes to breed, etc. We can add those buttons too.
* The updateCells method, iterates through all the cells in myGrid and calls the update function (defined in the Cell), on them.
* The updateDisplay method, uses the information in myGrid to recreate the display.
* The startFrame method returns a Keyframe that can be used in the running timeline generated in CellWorld (we will go over this class later). It uses an EventHandler<ActionEvent> called oneRound that will call the updateCell and updateDisplay function at certain intervals.
* The readInAndParse will take in a XML file submitted and converts in into the instance variables, myXDim, myYDim, myInitialStates, myCellType, myTurns.
* The pause function will stop the timeline and stop the model.
* The continue function will restart the timeline and resume the model.
* The restart function will recall initialize and set the timeline to 0, in order to restart the model.

CellWorld

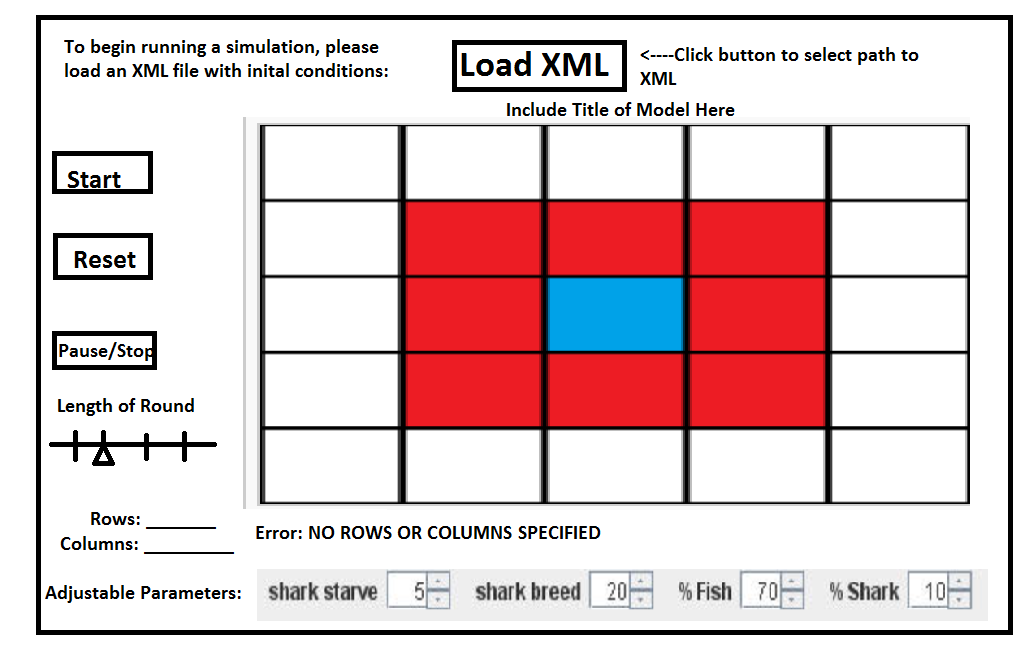
The CellWorld class serves as the main method that is used to create the GUI for the program. It has three instance variables, myCellLoop, WIDTH, and HEIGHT.

* myCellLoop is an instance of the CellLoop class. It is used to call the generateScene method in order to create the scene to be displayed. It is also used in order to call the start() method, which is used in the Timeline for the scene.
* WIDTH and HEIGHT are final static variables that adjust the display size of the GUI.

The CellWorld Class has two simple methods, a main method for starting the program and a start() method. The start method will set the stage using the scene generated by the generateScene method in CellLoop and create a timeline that takes in the Keyframe generated using the startFrame method in CellLoop.

**User Interface**

The user will interact with the program by choosing the path to the XML file on their computer. As indicated in the specifications, the XML will contain “the start state of the simulation as well as any initial parameter settings.” Selecting the path to the XML will enable extracting the various parameter and state properties, which would be provided as input to classes and methods in our program. The user will also have the option to reset, start, stop/pause, and specify the duration of each round of updates (enabling speed of cell changes from a visualization standpoint), number of cells in the grid. By extracting the initial parameters, we expect to be able to generate the necessary set of buttons that allows the user to change the values of parameters. In this way, we hope to keep the code flexible. The erroneous situations that we expect to report the user include bad XML data (i.e., corrupt XML file), be it an empty file or otherwise. The user can type into the GUI the total number of cells they want in the model, and this will be error-checked to make sure that rows and columns provided are natural numbers (>0) within permittible bounds (limit total number of cells to 900 or something - TBD later). We expect to black-out the Start button, until a valid XML file has been provided. We also plan on informing the user that they need to load an XML file before starting the simulation.

 *Preliminary Sketch of User Interface*

**Design Details**

The Cell Manager object will loop through the grid of cells, mark the current cell it reaches as visited, and skip any cell it encounters that is marked as visited. Each of the simulation modules has different update logic. They are represented in pseudo codes below:

For Schelling’s Model of Segregation:

Mark the current cell as visited

Look at all the 8 neighbors

Calculate percentage of neighbors satisfied with current cell

If percentage is below the defined satisfaction threshold,

Then find an empty cell

Change that empty cell’s state to current cell’s state

Current cell’s state is empty

For Wa-Tor World:

Mark the current cell as visited

Increment the number of turns for this cell

If the current cell’s state is FISH

If there is an empty neighbor

Then set the empty neighbor to the current state

If the number of turns > the number of turns required for breeding

Then set the current state to FISH, number of turns = 0

If the current cell’s state is SHARK

If there is a neighbor whose state is FISH

Then set the neighbor to empty

Else if there is an empty neighbor

Then set the empty neighbor to the current state

If the number of turns > the number of turns required for breeding

Then set the current state to SHARK, number of turns = 0

For Spreading of Fire:

Mark the current cell as visited

If the current cell’s state is BURNING

Then change current cell’s state to EMPTY, return and end the round

If the current cell’s state is TREE and if any of the neighbors’ states are BURNING

Generate a random # between 0 and 1.0

If that random # < some defined threshold

Then myState = BURNING

This logic will be implemented in the specific extension of the Cell class (update method).

InitialCell class is included in order to store the data from the XML file regarding the initial parameters that are set in the grid. That way, the user of this program does not have to go back and reload the XML file in order to restart the simulation.

CellLoop is meant to serve the cyclic updates. It will call the CellManager on each cycle to run updates through the grid of Cells. Then, it will pass the updated grid data to the view to be refreshed. The idea of the CellLoop is to have a control that calls both the function and the view separately but consecutively. CellLoop will serve as the interface between the CellManager and the GUI, especially after the user has selected an input file.

Lastly, CellWorld serves as the overarching main class, which will spawn the initial GUI and call CellLoop to update the displayed information each cycle.

**Design Considerations**

A design consideration that need to be addressed before attempting to devise a complete design solution is determining the most suitable/optimal format for the XML file holding the specific initial conditions. At present, we have set the XML that is read-in by the simulation to include the name of the model, the dimensions of the grid, and the initial states of cells. We may have to modify this later, but these were the initial key pieces of data that we felt was required at a minimum. The main design decision that we discussed was the class hierarchy for this project. At our in-class meeting, we decided that the best way to proceed was to split the 3 cell automata examples and make note of important characteristics of them. We came to the consensus that a Cell class would be the fundamental unit at hand, a collection of which would make up the grid. We listed down attributes common to the three examples, and features that we could reasonably expect to be extend and modify in future iterations. We decided that it would be best for the Cell class to be abstract given that the different simulations updated cells in different ways. Perhaps, the most important design consideration that will likely be worked out once we start writing the code, is how the data parsed from the XML is fed into classes as arguments. Also, figuring out the interaction between the Cell, which is responsible for displaying the cells, and the CellWorld which updates the scene. Our default scene for the GUI includes all the parameters that could possibly be tweaked during a simulation (in addition to the Start, Reset, Pause/Stop), but only relevant buttons will be enabled. We weren’t sure if the user should be able to change parameters such as “Shark starve” or “Shark breed” during the simulation, or whether all that information comes from the XML. In order to be flexible with our code, we decided that having a super GUI with only certain buttons enabled would help account for this possibility.

XML FORMAT:

Line 1: Name of Simulation Mode (i.e. “FireCell”)

Line 2: Grid X Dimension

Line 3: Grid Y Dimension

Line 4: Logic threshold for specific simulation mode

For every initial cell that is occupied: <cell state=“” x=“” y=“”>

**Team Responsibilities**

GUI Elements - Rahul

Parent class - Kevin

Logic for each game - Abhishek

XML Parsing - Abhishek

See next document for class diagram!