**CS 308: Cell Society Plan**

by Steven Pierre, Minkwon Lee, Allan Kiplagat

09/10/2014

* **Introduction**

In this assignment, our goal is to provide a Cellular Automata model, or CA, that can perform simulations based on the set of rules provided by an XML file.The XML file will provide the location of a set of cells situated inside a large grid composed of patches, each with its own rules. The program must accept any form of rule, apply them accordingly to the grid, and show the simulation with the use of JavaFX. The simulation starts with all cells in an initial state and is run by updating each cell by a set of fixed rules as provided by the XML file that the program reads.

    The program is structured with two major classes, one that handles calculation and one that handles sprites. One of the two will manage the updates applied to our grid based on the sets of rules given to us by the XML file, in other words it takes care of the calculation for the rules in the model. Whereas, the other one, will take care of creating the stage and setting the scene in which our calculations are displayed. The other one, in other words, creates the visual representation of our calculation for us to see; it updates the previous scene to match our newest calculations. These two classes work in tandem by dividing up the calculation and the update of our sprites independently. These two classes are closed as they are the foundation of our code.

    The rest of the code consists of modules that create the cell in the grid visually, and classes that assign properties to each cell. These remaining classes have a hierarchy as they are the ones that set what can be done with our code and its limitations. These classes are open and can be change or improved at any time without creating a roadblock in the code.  These classes can be extended further with more subclasses, and more can be added to them as the user sees fit.

* **Overview**

Our project is divided up into three big modules: one that handles graphic user interface and what is shown on the screen, another that oversees the simulation and generally manages and updates what should on the grid, and the final one that defines individual units of cells and patches that will eventually have its own rules.

The main purpose of the first module, which consists of two classes, is to provide everything on the screen that users can see and possibly interact with. For example, Simulation class, as it launches the whole simulation, will provide the stage and some features on the screen that are relatively constant, such as a button for the menu and a console that shows a message to users. The other part of this module is CellScene. The CellScene class plays a role of showing the important portion of the screen, as it creates the 2-D grid and shows how it changes over time. This change over time is the part where this module comes into contact with the next module.

|  |
| --- |
| Simulation extends Application |
| -void main(String[] args)  Launch(args)  @override  -void start(Stage s)  //set stage properties  buildScene(s) |

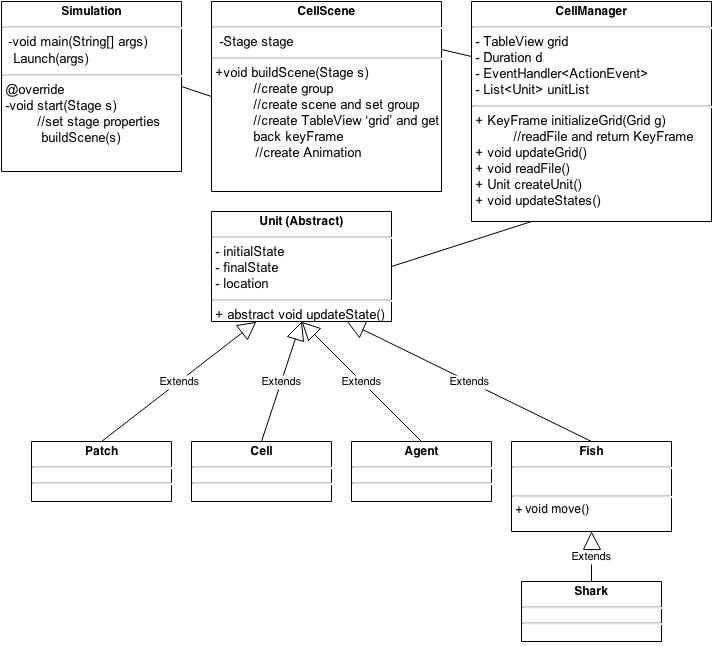
|  |
| --- |
| CellScene |
| -Stage stage  +void buildScene(Stage s)  //create group  //create scene and set group  //create TableView ‘grid’ and get back keyFrame  //create Animation |

The second module, perhaps the most crucial and important among the three, consists of only one class, CellManager. Generally speaking, its main role is to oversee the simulation and manage all the updates, so that the grid created by CellScene class can be updated constantly and accordingly. CellManager class, using initializeGrid(), first reads in the given XML file and creates cells with its respective rules and positions. Once the grid is initialized, it will constantly and automatically call for EventHandler, in which updateGrid() method is called. Using updateGrid() as this method calls for updateStates(), it constantly updates individual cells and patches on the grid. The updated unitList is used in updateGrid() to show its new states and locations as the result of respective time step.

|  |
| --- |
| CellManager |
| -TableView grid  -Duration d  -EventHandler<ActionEvent>  -List<Unit> unitList  +KeyFrame initializeGrid(Grid g)  //readFile and return KeyFrame  +void updateGrid()  +void readFile()  +Unit createUnit()  +void updateStates() |

The final module is used to make our project’s building blocks, cells and patches. Patch and Cell class extend abstract super Unit class, which contains a few common variables and methods, such as initialState, finalState, location, and updateState(). Cell objects and Patch objects will be initialized with its own position and interaction rules, according to what’s given from XML file. Once initialized, these cells and patches will be used in CellManager class to update the grid. There will be a few subclasses of Cell class, as each will specialize in different, specific CA simulation.

|  |
| --- |
| Unit |
| -initialState  -finalState  -location  +abstract void updateState() |



Overall inheritance hierarchy diagram

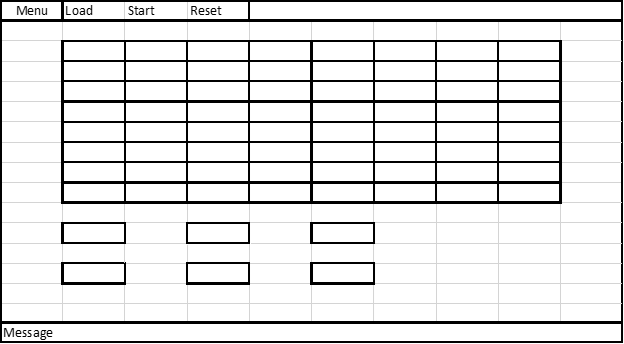
* **User Interface**

The graphical interface is expected to look like the one depicted below. It will be created through JavaFx and will be updated by the means of the class CellManager. It has a console at the bottom, which will inform the users of any errors or missing information. The console will constantly update the user on the state of the code in ways such as, which steps you are on , if it is done processing or not, and what kind of errors you are encountering due to faulty or missing data.

    Above the console are the buttons, which will provide the user an interface through which he can select or modify is simulation. Three of the buttons are dedicated to select which model the user wants to apply. One of the buttons is dedicated to Schelling’s model of segregation, one to Wa-Tor World model of predator-prey relationships, and one to Spreading of fire. One or more of the buttons will be reserved for any new models the user would feel like introducing. And a couple of buttons are saved to allow the user to set grid size, or make the simulation run faster or slower if they are not pleased with the default speed of the simulation.These buttons in other words will determine what kind of simulation we will run, how fast we will run it, and how we will run it.

    The third part in the interface will be the grid in which the simulation will be displayed. This part of the code will not react to user input as it is just a display of the simulation. It will display how the simulation spreads as function of time or steps.The user will be capable of setting how many steps he wants displayed per second through the buttons at the bottom but he won’t be able to modify the code from the grid by clicking on it or anything of that nature. The picture will be located in the center of the GUI and will be the most visual component of the interface.

    The last part of the graphical user interface will be the toolbar and the menu located at the very top of the GUI. The toolbar will contain a load, start, and reset button which will do as they say. The Load button will allow the user to load an xml file through a user interface. The Start button will tell the simulation to start and display the resulting simulation on the grid depicted below. The Reset button will allow the user to reset the simulation if he wants to review all the steps again for analysis purposes or further inspection.

****

* **Design Details**

As mentioned in the Overview, this project is divided into three modules: the graphical user interface module (consists of the Simulation and CellScene classes), the simulation handling module (consists of the CellManager class), and the cell simulation elements module (consists of the Unit class and its inheritance hierarchy).

The Simulation class extends the Application Java class and is where the application is launched. The overridden start() method inherited from the Application class sets up the program’s Stage i.e. the stage size, resizability etc, then passes the stage to the CellScene buildScene() method. The CellScene class stores the pointer to the Stage as a private instance variable in case the Stage’s scene needs to be changed later. The buildScene() method creates a Group for organizing the elements in the scene (grid, buttons, etc) then creates a Scene object and sets its Group. A TableView node is then created and added to the group. A KeyFrame is obtained by calling initializeGrid(TableView) in CellManager. An Animation object is then created, assigned the KeyFrame, cycle count set, then the animation started by invoking play().

The initializeGrid(TableView) method in the CellManager class stores the TableView as a private instance variable through which CellManager can access the grid locations and update their appearance. The class has an EventHandler<ActionEvent> instance variable that defines the code that runs on each frame i.e. calls to state-updating methods. The Duration instance variable defines the simulation speed. initializeGrid(TableView) returns a KeyFrame to the CellManager, instantiated with the Duration and EventHandler parameters.

           The CellManager class a List<Unit> instance variable, unitList, that contain the unit instance variables read in from the XML file. The readFile() method reads in the XML file and within it, calls are made to createUnit() that instantiates the units and adds the elements to the list. The initializeGrid() method calls the readFile() method after which the unitList instance variable contains the unit objects with their initial states. initializeGrid then calls updateGrid() which loops through the list and updates the grid with a visual representation of the unit states. For every frame, the updateStates() method is called which loops through the list and executes the updateState() method in each of the units.

The Unit superclass has private initialState, finalState and location instance variables, which are attributes shared by all kinds of units. The superclass also has an abstract updateStates() method that each subclass implements with its own unique set of updating rules as defined by the particular CA model in question. Patch and Cell classes are direct subclasses of the Unit class. According to the program specs, these are the two main components being studied in the CA model: the grid locations (Patches) and elements on the grid locations (Cells). This design can be extended easily to accommodate other kinds of simulations by having the components in the simulations be subclasses of the Unit class. For example, Schellings’s segregation model will have an Agent subclass with its own implementation of updateStates(). The War-Tor World model will have a Fish subclass extend the Unit class, then also have its own subclass of Shark, since the model says that Sharks move like fish sometimes therefore they should inherit that functionality. Spreading of fire would just use the Patch class.

This 3-module design was selected with the main idea being that the CA simulations work the same way, with the difference being how the updating rules are implemented for the different CA components. Therefore any kind of new updating can be accommodated without changing the existing code by simply extending the Unit class. Isolating the different major functions of the program into modules also allows making upgrades easy to implement. For example, introducing a splash page would just require an extension of the CellScene and addition of a few lines of code.

* **Design Considerations**

    -Whether subclasses of Cell class should be initialized with its own interaction rules or should be separately given interaction rules after they are initialized, because separately giving interaction rules after being initialized may make those subclasses more flexible, but reading each version of simulation and extract rules may be more difficult and inefficient to implement

    - Whether or not we should take a brand new set of rules from the xml file, since it would be hard for the xml file to provide enough coding information to create a brand new simulation from that. It is hard to convert words to code. The pros is that it would make for an incredibly impressive code, the con is that there is no practical way to do it.

    - whether we should add more subclasses to our code in order to render it more flexible in areas where that is not obviously the case so far. The  positive is that it makes for a more versatile and flexible code, the con is that there is no clear indication that this would happen.

    -Whether we should use recursion to update all the cells in that time step or use a simple loop to go through all the cells and update each separately, because cells’ behavior seem like they can be updated using recursion, but recursion is much more likely to be inefficient to use. We could also run the whole thing as an animation.

    -

* **Team Responsibilities**

    In the beginning we created three main modules in order to divide up the work evenly across all of us. So we decided to each work on one of the module. Steven Pierre will work on the unit superclass and all the subclasses relating to it. His job consist of implementing the models that are given in the description of cell society. He will create those classes that will in return perform the corresponding model once called upon by the calculation class cell manager. He has to implement three different models as part of subclasses of unit, so that the cells inherit the appropriate methodology to do the simulation needed.

    Minkwon will take care of the cell manager. His task consist of reading the xml file and calling on the appropriate method to run the proper simulation. He will perform the simulation on the cells and return the appropriate results to the Cell Scene class to be displayed on the graphical user interface. He will also take care of the update of the grid and pass it back to the Cell Scene which will take care of displaying the updated cells.

    Allan will take care of all the visuals associated with the simulation. He will create the GUI and will take care of creating the buttons and the menu. He will also create the grid by means of javafx built on grid function tableview. Allan is essentially responsible of creating all of the visuals and assigning the proper functions to all the buttons as suggested by their respective name.