**Plan:**

**Title:**
Cellular Automata

**Names:**
Michael Deng
Pranava Raparla
David Zhang

**Introduction:**

The problem we have to solve is to try to create a game that, for each frame, updates an entire matrix of cells based off of the states and the values of its neighbors. We would have to read in an XML file with a bunch of different parameters and values detailing how each cell would react to the states of each one of its neighbors and under what conditions would that particular cell react. We would have to design an application that can effectively take a wide variety of values and goals and adapt those different guidelines to one method of updating the grid. We also have the issue of needing to create the grid ourselves and then dynamically populating the grid per frame based on how the cells are updated.

Our ultimate goal is to design an application that allows us to adapt our game to any situation with the minimal amount of effort and additions to the code. For example, for each new scenario that comes along, ideally we would want to have an architectural design that allows us to only need to implement a new cell subclass in order to run that new scenario correctly. In order to do this, we believe that the best way to design our project would be to have a game loop that constantly updates a grid. The grid would then iterate through each of the cells and depending on the state of its neighbors, that cell would either change its state or not change at all. We would have a superclass for a cell and then create new subclasses for the cell for each different scenario.

We would ultimately want our most flexible class to be the grid because it is that class that ultimately interacts the most with each individual cell. We would want to design the grid to be extremely shy, where all it has to do is call a method and change the state of the current cell. Each individual cell subclass is where we want the majority of the work to be done, because each type of cell for each type of scenario is going to have different values with different goals. Cell subclasses will be open to the grid and vice versa, but cells will not be open to any other class. The game loop should only work with the grid and the main class should only initialize the game loop. Cell subclasses would extend an abstract cell class.

**Overview:**

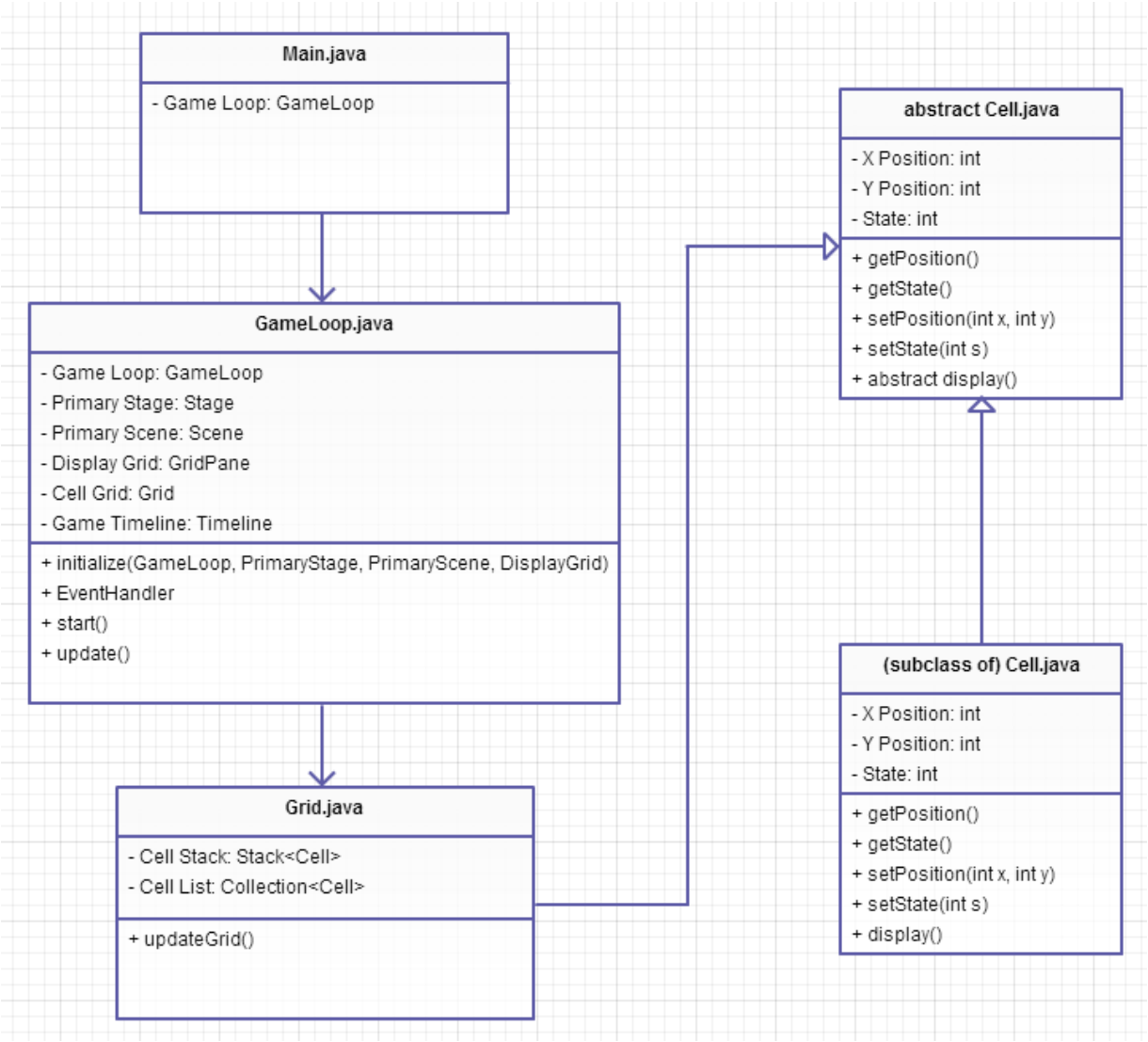
The Cell Society design follows the basic template of the first game in this class. A main driver sets up a game loop that updates its properties after each frame change. The primary difference here is that instead of having game actors and a collision handler, we have a grid and cells on the grid that are updated every frame. Three modules compose the Cell Society program: the main driver, the game loop, and the grid. The UML diagram with the class breakdown is shown below.

The first module, the main driver, is composed of a single class, Main.java. In Main.java, we will create the stage, the scene, and the grid pane, after which we will pass it to the game loop in the second module. The purpose of main driver is to set up the GUI, initialize and start the game loop. This separation enables better compartmentalization for testing and debugging.

The second module consists of a single class as well, GameLoop.java. This module takes in the stage, scene, and grid pane parameters passed in from the main driver module. GameLoop also has a Timeline and event handlers that manage updates from frame to frame. Methods include initialize(), start(), and update(). Initialize and start are both called from Main.java, while update is called from within after frame changes. Initialize() stores the parameters passed in from the first module, start() starts the timeline, and update() manages changes to the grid and the cell society models. GameLoop also holds variables for the Grid, including a GridPane (javafx standard class), Grid, and Cell.

The third module includes all the classes related to the setting up the grid and the cell society model. The first class, GridPane, is a javafx class that sets up a grid on the scene. A grid pane is instantiated in the main driver and passed to the game loop. The Grid.java class stores all the cells in the grid, which are added to the instantiated GridPane object. Grid.java has a list of all cells and a collection (stack or queue) for cells that are to be moved to empty locations on the grid. It also has a method, updateGrid(), which handles updates for individual cell properties and arrangements of cells on the grid pane. Cell.java is the abstract superclass that manages all the actions and information of a cell in the grid of the cell society. It is an abstract class because depending on the type of model, different state variables, display actions, and update methods will be appropriate. However, regardless of how cells update and interact with each other, they are always updated in the same manner in Grid.java. Cell outlines some initial variables like the position and state, along with basic get, set, and display methods. Each cell is added to its respective position on the GridPane object, where it is then displayed. The subclasses need to explicitly define the states and visualizations. This Grid architecture is highly compartmentalized and flexible, making extensions of Cell Society easy to implement. All that would be required is the creation of a new cell subclass that the Grid.java class would hold a list of.

The basic flow of the program is as follows: 1) the main driver sets up the initial visualization and start frame for cell society model, 2) the game loop updates the cell society grid in each frame, and 3) the grid updates cell states and positions on the grid when appropriate. We believe that this architecture will make division of tasks simple, improve debugging, enhance the readability of our code, and enable quick extensions of our code.

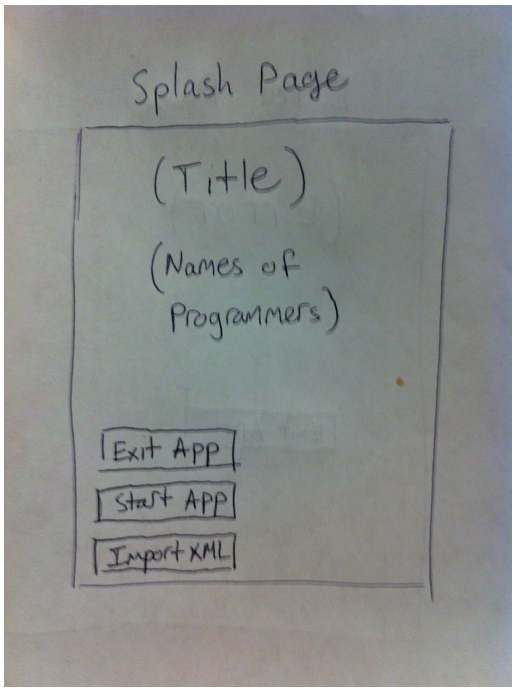


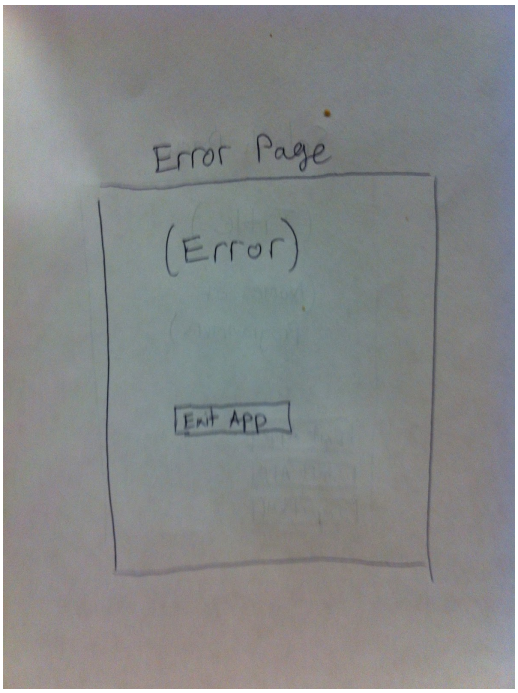

**User Interface:**

We want to create a three scene interface that is comprised of a splash title scene, the main game scene, and a pause screen that appears whenever an erroneous situation occurs. When the application is started, the first scene that is initialized is the splash screen, which will have the title of our application, all of our names, a button to exit the application, and a button to start the main game scene, and a button to load in the XML file. The user would have to first load in an XML file, or else clicking the start button would immediately take the user to the erroneous error page. Once a XML file is loaded in, that the user can click the start button which will take the user to the main game scene.

The main game scene is where the majority of the program would actually take place. We would used a gridpane to populate the scene and basically turn the entire staged into a gridded matrix with individual cells of different images, colors, number, etc. The scene would start automatically and have very little input from the user. The only button on the page would give the user the ability to quit the application altogether.

The erroneous pause page would appear automatically under certain conditions. Our ultimate goal with the erroneous page would be to prevent the start of the game scene by catching problems and discrepancies in the XML file. We would want the erroneous page to only appear before the main application starts so that it would not interrupt the application once it has started. Examples of things that would trigger the erroneous page would be an incorrect number of arguments, arguements that are not of the right type, a sum of values our application is not built to handle, if cases we did not prepare for, etc. I guess we would try to build in asynchronous error catches if we had the time and if we knew how to do it appropriately while the application was running. On the error page, we would have a button that allows us to exit the application.




**Design Details:**

As described in the overview, the program consists of three modules. The main driver sets up and starts the game loop module, which sets up the frame updates and holds the grid variables. The grid module contains the grid and cell classes that manage the updating of individual cells and their positions on the grid. In this section, we will go over the specifics of what these modules do and how they interact with each other.

The first module sets up the scenes and the game loop. In our program, there are three primary pages that we switch between: the start page, a pause page, and a main game page. Each of these pages are associated with a scene, all of which are set up in Main.java. The start page has buttons for which game page to go to, and the pause page is displayed when an error has occurred in the program. These pages are created in Main.java, and the event handler in GameLoop.java decides when to switch them, based on button clicks.

The second module consists of the the game loop (GameLoop.java), which is initialized from the main driver and started from the main driver. These act in the same way as the first assignment of this course. The event handler in the game loop controls action events for button clicks, which control switching between pages. A Grid.java object is instantiated when the GameLoop is initialized. When GameLoop calls update() after each frame change, a call to the Grid object's update method is made, thus compartmentalizing all actions related specifically to the Grid and the cells.

The third module, which contains all the grid classes, handles all the grid and cell related updates. the Grid.java class has a list of all the cells that should be displayed in the grid pane, along with a collection (stack or queue) of cells that should be moved around the grid. When the updateGrid() is called from Grid.java, Grid loops through its list of cells. Each cell is updated as appropriate, depending on its location in the grid, and cells that should be moved are removed and put on the other collection. As soon as an empty grid space is found, the cells queued up for position changes are removed from the collection and placed back into the grid. The Cell.java abstract class has a few public position and state variables, along with an abstract method for displaying itself. The cell states will be updated from Grid.java's updateGrid() method because it can handle changes between cells and use Cell's get/set methods to modify their states.

**Design Considerations:**

One thing well will have to consider are the different kind of stages that could potentially populate our cells. We don't know if the values we will be fed are going to be integers, colors, strings, images, etc. So what we need to do is create a flexible method within our cell classes that allows for the stage to be dynamically changed if need be. Or what we can do is ask for the format in the XML document and then change our variable type in the cell class appropriately.

One of the bigger issues we have to consider is how the cells are going to update from scenario to scenario. In the case of the Game of Life or a forest fire simulation, each cell that is affected by its neighbors need only update its one state. However, in the case of Schelling's Model of Segregation, whenever a cell is updated due to the conditions of the cells around it, another cell has to be updated as well. In this sense, our code has to be flexible and dynamic enough to autonomously handle either case correctly. Our currently plan of implementation is to use some sort of stack or queue into order to keep track of updated cells, but that idea is subject to change.

Our group also had a lot of trouble with how much power we wanted to give either the cell or the grid. One side of the argument is that cells have control over where they are placed and they are fed the entire grid. The other side of the argument says that the grid should have the ultimate control and should control where each, individual cell is placed. We have currently decided to go with the implementation where the grid has the ultimate power to manipulate the state of cells.

Another issue we need to consider is how much information we want the user to be able to input through the XML document. Depending on how much information the user can provide, the code for our application may change very drastically and very fundamentally. The type of information that user can input is also very important and an issue we will have to consider.

The number of if cases we want to handle for any particular scenario is also a very important consideration we as a group have to take in account. We will of course handle all the if cases that have anything to do with the boundaries of the grid, or anything else that is generic to other cellular automata problems, but there are some if cases that are unique to individual scenarios. For example, in the case of

Schelling's Model of Segregation, there is the slim possibility that there are not free spaces for a cell to jump too. In this case, our application will break if we do not handle it with a unique if case or if we just decide to let our erroneous scene to take care of the error. In the end we will have to deal with all the exceptions sooner or later.

Just as a coding consideration, we have to bear in mind that a gridpane may not work exactly the same way as a matrix in java. We have not looked into it, but just in case it works differently, we may need to consider minor design changes and implementations in order to handle those cases.

**Team Responsibilities:**

Michael Deng would be responsible for creating the different scenes, stage, and panes of the application. He would be responsible for populating the Main class, as well as the GameLoop class that is responsible for iterating through frames and updating the grid once per frame. Michael would also work with Pranava on the Grid class, which is our major class responsible for interacting with cells on an individual bases.

Pranava Raparla would be responsible for writing the class and the methods that read in and interpret XML files. He would also be the one responsible for designing the specific format of the XML files we would want to be read in. Pranava is also responsible for helping and working with Michael designing and code the Grid class, which is essential to our application.

David Zhang will be responsible for the creation of the abstract cell class, as well as the creation of all present and future cell sub classes. He will be responsible for understanding and interpreting new scenarios and figuring out how exactly to create new cell subclasses in order to interact with the grid class.

All three of us will work together to plan and code how the cell classes and the grid class will work together. We want to do this part together because the relationship between these two classes is the most important relationship in our application.