BasicCell Society Design Final

TEAM 1

Ken Kalin (kdk33), Ethan Horowitz (ejh55), Tim Geissler (tdg24)

Team Roles and Responsibilities

- Ken Kalin (kdk33):
    - Cell Subclasses for various simulations including
        - Conway's Game of Life
        - Spreading of Fire
        - Schelling's Model of Segregation
        - Wa-Tor (Predator Prey Model)
        - Percolation
        - Falling Water/Sand
        - Foraging Ant Simulation
    - Documentation
- Ethan Horowitz (ejh55):
    - UI Design
        - Simulation Controls (Pause, Play, Speed, Step Fwd)
        - Histogram viewer of simulation progress
        - Support for separate windows for UI controls and Grid
- Tim Geissler (tdg24):

    - Grid Class
        - Supports simulation and initializes various simulation types based on XML data
    - XML Loader
        - Parses XML file that contains
            - Initial states and positions of cells
            - General simulation paramters (e.g. probabilty of tree catching fire)
- XML Generator Tool
    - Additional support program that enables team to generate XML files for testing/verification of project.

Design goals

What Features are Easy to Add

In our implementation of the simulation, a `Grid` class is responsible for supporting a simulation in which cells each understand the rules of their simulation and change states based on internal and external factors such as the number of neighbors surronding them. The `Grid` class is responsible for creating the simulation based on paramters defined in an XML file and then instanciating the correct types of `Cell` objects. The `UI` controls the speed at which the simulation progresses and allows a user to load different XML files with various simulation paramters. ## High-level Design

Core Classes

Cell

The `Cell()` class contains the overarching information upon which different subclasses can be created for each simulation type. In our implementaiton, the rules for a given simulation are primarily contained within a Cell (the Grid is responsible for instantiating the correct cells and assigning them the correct neighbors based on the paramters).

`Cell` objects are constructed from the `Grid` class by assigning them an `x` and `y` position as well as a list of `COLORS` and possible `STATES` that the cell can take on.

At each time slice througout the simulation, the grid interacts with each cell by calling `calcNextState()` on every cell in the simualtion. Calculating the next state of a cell before updating it on the grid prevents

undefined behaviors such as some cells updating first therby changing the results for their neighbors.

Once every cell has had a chance to calculate its next state, the `Grid` calls `update()` which sets `state = nextState` and then calls `setColor()` which displays the new states color on the grid.

Neighbor

Each cell maintains a list of its neighbors which can be used as part of the simulation to calcualte a cell's next state. The neighbor class provides a container for this information since the number of neighbors a cell has depends on the rules of the simulation as well as the size/shape of the grid itself. Subclasses of the `Neighbor` class allow for different configurations of neighbors such as `SurroundNeighbors` which has 8 neighbors (all adjacent grid cells in a rectangular grid) or `CrossNeighbors` which is only the left,right, top, and bottom adjacent square. (This is used in the Fire simuation for example). Adding neighbors on the other side of the grid allows the simulation to wrap around and become periodic.

Grid

In keeping with Model View Seperation the Grid is actually broken up into two parts. `CellGrid` contains the simualtion logic and `Grid` contains the UI functionality.

Cell Grid

`CellGrid` contains methods which setup and run the simulation. `populateGrid()` populates the grid with cells, given the cell class and cellsize. This method constructs new cell objects at each location in the grid that is occupied based on the XML file.

`getCellParams()` take in the Cell class and the XML loader and reads the params field from the `Cell` class

`linkCells()` takes in a class of cells and gets the `Neighbotype` which is a string that determines the configuraiton of neighbors for a simulation. For each cell in the grid, it gets the neighbor and passes them to each cell in the form of a `Neighbor` object.

The simulation runs in a two part cycle for each "frame". First the future states of cells are calculated by calling ` `calcNextState()` which calls the corresponding method (of the same name) in each cell. Then on the next part of the cyle, the cells are updated to match their next state by running `update()` which runs the method on each cell.

UI

The UI consists of three main components each with their own window. The `ControlBar` contains buttons

that trigger the simulation to execute different tasks such as play, pause, step, speed, info card, loading file and making a new simualtion window. (Closing the control bar closes the entire simluation).

The Main class has an arrayList of simluations (`Sim` class objects) and 60 times a second it runs the update method of each object using a Java FX animation. The sim class updates each of the three windows for its simluation independently allowing for multiple simultaneous simulations to run at the same time.

Grid

Grid is a part of the Model View Separation which isolations the Java FX functionality from the logic of the simulation. It loads a window and displays the cells and can also display the info card.

`InfoCard` shows the title, author, cell states, and description for each type of simluation.

The `Historgram` class displays cell counts over time as the simulation elapeses.

XML Loader

The XML loader scans in fields from an XML file and then it stores the data in its own local variables which are accessible by the grid through getter methods. For example, `getTitle() getWidth() getHeight()` which setup the Grid window and `getCells()` returns an arrayList of cell representations which is a `record` class of `row` , `column` , and `state` data which are then instantiated into cells.

The XML Loader includes error checking cababilities to validate the XML file.

- Checks that values for each XML field correspond to their aassigned data types.
- Checks if there are null values that should be filled
- Checks for a valid grid size (600 pixels square or less)
- Checks for file integrity

XML Generator (Companion Program)

Tim (tdg24) wrote an additional companion program to generate XML files for testing and validation

The XML generator prompts the user for a `Grid Size` and generates a grid that user can populate with cells using mouse clicks. There is a control window that allows the user to select the simluation type, as well as fill simluation parameters to be saved. Selecting a different simulation type automatically updates the interface and repopulates the grid with different template cell types. **Clicking on a cell multiple times cyles through its possible states.**

Clickng the **Generate** button exports an XML file which can then be loaded into the main simulator.

Assumptions that Affect the Design

Features Affected by Assumptions

Our design assumes square Cells which each are encoded with the simulation rules and neighbor types. Since this information is not defined within the XML, adding a new type of simulation to our program requires adding a correspodning Cell class with the simulation rules *the XML file only specifies what kind of cell to load based on the existing library in the program*.

The grid can be a maximum of 600x600 cells because that is the number of pixel in the window. Any few cells and the cells will automatically increase their `cellSize` to fill the screen. Scrolling was not implemnted so a greater number will result in some cells not being visible.

All cells start with the same paramters. So for simluations that operate based on probabalities, there is no way to have different settings for each cell, but only to specificy the rules for ALL cells of a given type.

Grid squares are not indpendant from their cell. For example, when cells move in a simluation they call `swapStates()` swaps the states of the cells rather than the cell objects themselves. This makes implenting certain simulations more complicated such as the Ant problem which assumes that grid squares can have data about them independant from the cell occupying then.

All colors and states are predefined and hard coded for each cell. Colors, States, Params, and Neighbor type must all be `public static final` fields in each cell so that the Grid can instantiate them correctly

Significant differences from Original Plan

- The original plan had no Model View Separation (MVC)
- Everything was designed in one window
- In addition, there was no encapsulation for the neighbors of a cell. Each cell was originallty designed with an array of neighbors, this has been replaced with a seperate object to contain this data
- Changed XML format to only include cells with non-zero states **decreasing the file size significantly**

New Features HowTo

Easy to Add Features

- Scrolling on the grid
- Additional simultion types:
  - Create a new sublass of `Cell` with the following information
    - `COLORS`
    - `STATES`
    - `NEIGHBOR_TYPE` (Assumes `SurrondNeighbors` if not defined)
    - `PARAMS` (Assumes empty)
  - Needs to override `calcNextState()` method and implement a constructor
  - Modify `update()` method if cell needs to move in additon to change color to reflect new state
  - Update `XML Generator` to be able to create files for new simluation type

Other Features not yet Done

- Different grid shapes and layouts
  - Currently the `x` and `y` coordinates are passed to each cell. For hexagonal grid this would be the same but there would be extra math involved to calculate neighbiors and positions.
  - Would also need to implement MVC for cells since currently the Java `Rectangle` class is used in each cell to display on the screen.
- Multithreading to allow each simulation to run on a parallel thread. Drastically increasing efficiency
- Step back function in addition to step forward
  - Would need to cache last n frames as XML state files (stripped down to eliminate simulation informaton like title/initial config)