CompSci 308: CellSociety Analysis

===================

Project Journal

=======

###Time Review

•Start Date: 09/12/15

•Finish Date: 09/27/15

•Hours: about 65 to 70 hours. The team spent about 4-5 hours planning for the first phase of the project and about 2-3 hours planning for the second phase.

After planning the project with the team, I started designing and coding features pertaining to my part of the project. I made use of the resources provided on the course website which saved the time I needed to code new features considerably. I spent most of my time on this project refactoring code, and adding new layers of abstraction. At the beginning of phase one, I had five classes for all the code for GUI, but after refactoring I had around 15 classes. This however, made adding new features in phase much easier. It also made debugging easier. I did not spend a lot of time debugging this project because I fixed most issues by checking a few (usually two or three) small methods directly related to the error.

###Teamwork:

The team spent about 4 hours planning the first phase, and assigning tasks. After that we had another meeting were we made changes to the design. For phase two, we also met again and made changes to the design before started coding new features.

• Roles

At the beginning the team agreed to divide work as following:

\*Elsie: xml section

\*Abhi: Simulations

\*Sally: GUI

Knowing that implementing simulations can take a considerable time, Elsie and I decided to help Abhi with his section by brain storming together for Game of Life simulation. I also fixed a bug in the update method for this simulation By the end of phase one, we were falling behind on the number of simulations that we had ready, so I coded Segregation, and Elsie coded Fire.

• Communication

The team used face-to-face meeting to discuss project design. Elsie and I did peer-programming when we first tried to implement Segregation simulation, and later when we re-factored the project to no longer use subclasses to represent states in different simulations. We used gitter to share code when we did not work together. We also used facebook messages to notify each other about new features or to set meeting times.

I think that we needed to communicate more often toward the end than what we actually did because there were incidents where one member of the team did not know that a feature was implemented and ready to use. We were able to clarify that during meetings, but we could have saved time if we notified each other often towards the end.

We benefited however from sharing feedback from meetings with TAs. Before any of us had a meeting with a TA, we prepared a list of questions from all members of the team. This was very helpful as we were able to redesign and re-factor parts of the project and get feedback during different parts of the project.

###Conclusions:

I did not over-estimate the project,but I did not know that I would spend as much time refactoring code after merge pull requests. this is something that i will allow more time for in the future. I did over-estimate the amount of time i would need to debug in this project. surprisingly, I spent the least amount of time debugging unlike almost all other projects that I have worked on before.

###Team Plan:

After familiarizing ourselves with the project specifications, the team met to design the project components. We used a method similar to CRC cards that we used during recitation. Each of Elsie and I had a list of candidate classes and methods. We put them on a white board and made comparisons. At the end, we created one list of classes, their type (discrete or concrete), and their methods. The three of us found connection between the classes, and created a scenario to make sure that the classes communicated with each other logically.

We also divided the project into three main components: GUI, xml, and simulations. Then each of us chose one. I think that we made an error here by inadvertently neglecting the configuration component. For example, we did not specify who should be responsible for writing the methods that would let the user choose random or deterministic locations of the cells. I coded deterministic initialization method because I needed the code to test GUI after checking with other members of the team if either one of them had done so. In phase two Elsie coded random the initialization method. Additionally, I implemented all "Manager" classes that send configuration results from GUI to differnt simulationas regarding the shape of cells to use, the type of esges, and the number of neighbors to check. I also fixed a file path bug in the xmlreader class.

We adhered to our tasks throughout most of the duration of the project. In addition to implementing GUI, I helped creating Game of Life simulation update method. I also implemented Segregation simulation for the first phase of the project, and I created torodiral edge method. As mentioned before, I created deterministic initialization method.

As for updates, we notified each other before making any changes to master branch on github. We would usually take turns integrating our changes to minimize merge time conflicts. As for testing, we pushed our test cases along with the code to each other. Personally, I tested all files that were marked ready every time I added a new feature to GUI or re-factored my code because I did not want to lose track or overlook a bug.

After receiving feedback from my TA, we redesigned the code so that it does not create new instances for each cell everytime the update method is called. We achieved this by changing states of simulations from subclasses to attributes in simulation class. Elsie and I peer-programmed to achieve that. This was one of the times where we did not only do our assigned tasks. Team roles held by more in phase two than in phase one especially after re-factoring our code. The code was very well logically separated; we did not have a lot of merge conflicts. This also minimized the dependency of the code on other sections.

###Commits

I committed my code every time I added a new tested feature, fixed a bug, or re-factored my code considerably. I usually packaged my commits to include a combination of these three areas if the bug or the feature I added was small.

I worked from Sally branch throughout the first phase of the project, and from SallyTemp branch throughout the second phase. My Commit messages are in general descriptive of the task completed, and they are represented of the work I did.

Commit #1: Trig grid complete

https://github.com/duke-compsci308-fall2015/cellsociety\_team14/commit/d95b8b9d730282ee032ec323aa1e363d45df54ac

As the title suggests, in this commit I added the triangle grid to GUI. I also added a choice box for users to select between the default square grid, and the triangular. After adding this commit and the following one, I merged with master, and fixed a merge conflict because our temporary branches where not synchronized with master. Adding the feature did not hinder other member's work because setting it only required adding the word "Triangle" in the corresponding .xml file or hitting a button on gui that will automatically change the grid shape.

Commit#2: "f"

https://github.com/duke-compsci308-fall2015/cellsociety\_team14/commit/11a569ca0744d197989aadfbe928f89b1adf50e0

The commit's title is was not intended. I had trouble pushing my code to github at this point. I tried to push my code several times but it did not show. I typed "f" as a commit message in one of my attempts, and it was the only one that went through.

I refactored grid.java and specificInitialStates.java classes in this commit. This commit was neccessary to fix a design issue and be able to work on phase two. before re-factoring the code, intilizeGrid() in specificInitialStates.java used to create a new grid shape every time the grid is updated. this made the program slow down and glitch. In this commit I changed intilizeGrid() to become the method that creates the array of states of type simulation. then in Grid.java I created two methods. the first one displayGrid() is responsible for visulizeing the grid the first time the program is loaded, reseted, or when the graid shape has changed. the second one displayGrid2() is responsible for updating the grid through out the program. only after seperating the methods, the program has become flexibile to support different behavior related to gird shape or different button clicked.

Design Review

=======

###Status

We followed naming convention by using Camel style of naming. We used descriptive method name. In the future though, I will suffeix classes names but the world class. I used it because many of GUI classes that I created are already classes in java. For example I named a class SqaureClass because there is already a class in java named Square. I personally created manager classes that include functionality and "regular" classes when needed that include states. We also made sure that we did not use any abbreviation in the code. As a result the program is comment free.

Before refactoring the code, I had long parameter list passed from a class to an abstract class to all subclasses that may or may not use them. To avoid that, I noticed that the same parameters are being passed to many methods. I decided to create a class called GUIVariables. This class contains variables that are usually used by other classes. I used getters and setters in this class to get the private variables. This helped make the code more readable, and the reduced dependency made it easier to call methods from classes that are not necessarily connected directly.

Features like buttons, checkboxes, Choice boxes, shape of grid, type of edge, number of neighbors are very easy to extend. They are all subclasses of their perspective abstract hierarchy. Adding extensions in phase two was very easy, as I just extended a new class within a package, and added only methods and attributes related to that object.

###Two classes: Wator.java and Kelp.java

These two classes are part of Wator simulation. Wator class does all the logic to create and update the simulation. It is a subclass of simulation, therefore it inherits findNeighbors(), needUpdate(), and update() from Simulation.java superclass. It also inherts attributes common to all simulations such as myState, myNextState, and myColor. It also has atrributes specific to Wator simulation. Kelp.java has four methods only; two of them are setters the other two are getters. It does not have any other behavior.

These classes are very closely related. I would make them one class. I would most probably make them one class and change kelp into a state rather than a class. First, all the getters and setters do is that they return a list of kelp locations. This can be done by using an if statement that will add to a list of all cells with kelp state. If this design was implemented, the classes can get rid of setting and getting the same information multiple times. At the moment a new instance of class Kelp.java is created every time a cell's state is kelp only to record its location

Another think that is interesting about these two classes is the use of public methods strictly. Many of the methods that count states, and set booleans are not used outside these two classes. In fact it is not necessary to have getters and setters for all of these variables if they are used in the same class.

Furthermore, method countStates in Wator uses two for loops and three if statements to count different states. This method does not take advantage of the already implemented method (xml.getStates) that returns all possible states in any given simulator. Instead the code relies on hard coded state names in the if statements. The method could also use a map to set the count of states to get rid of the if statements since a hashmap will automatically put a new entry for a new state. In this case the code only needs to check if a state already exists, and in this case increase the value by 1, or if it is a new state then set its value to 1. The code uses hard coded states throughout the class. For this reason it is not possible to use it as is in a different application. If the code had used the method (getmystate()) instead then it would be easier to generalize the code.

###Design

The Simulation component includes the logic to implement the simulations as well as their attributes. The different simulations are subclasses that extend one super-class which has all common attributes and common abstract methods. Additionally each simulation implements a number of methods that are unique to it. Even though the simulations have states, they do not have any reference to their visual representation. Simulations get their initial data from xml file or are modified by the user on the GUI. There are a number of "Manager" classes that hand the simulation class the shape of cells to use, and consequently the number and the direction of neighbors a cell can have, as well as the type of edges to check.

A simulation returns a list of the new states to the GUI component of the program. The GUI also gets shape of cell parameter from GridShapeManager class. The first time the program is loaded the grid is created with the shape of cells, images, and all buttons, sliders, and boxes, and the graph. Afterwards these components are called and updated periodically. If the gird shape is changed then the grid dimensions are recalculated and the grid is recreated. all other GUI components have listeners that notify the GUI with the feature that is changed and the grid is updated accordingly

#### Adding a new simulation:

To add a new simulation, first create an xml file that has the simulation name, and its states, and a list of states that determine the initial state of the simulation. The state can be left empty, and in this case the simulation will be initialized randomly. Additionally, the xml file should have the shape of cells on the grid, the number of neighbors to check for the simulation, the type of edges, and probability if applies.

Furthermore, a class called PossibleSpecifiStates should be updated to include an instance of the new simulation, and a mapping of the simulation's states to images to show from Images folder.

Finally, a new class with the simulation name is created. It should extend Simulation.java to inherit attributes such as state, color, and location, as well as update(), and checkNeighbors(). The new simulation can have any number of other methods and attributes specific to it. The class should eventually return a simulation array to GUI, which in turns passes it to displayGrid2().

###My code:

The main components of the GUI are the grid, the graph, the control panel which includes: buttons, sliders, checkboxes, and choice boxes. The GUI class acts as a manager that coordinates between different the components mentioned above at any moment during the program life. When GUI.java first gets data from the xml component, it saves them in a class called GUIVariables to add flexibility and reduces dependency on parameters between different classes.

I created a shape package that contains implementation of different types of grid shapes. For each shape there is a different method that calculates dimensions of cells depending on the number of rows and the number of columns specified in the xml file. There is also a method that sets attributes specific to that shape. There is also a class called ShapeManger that currently has the number of neighbors to consider for each shape. At the moment this class has a shared method that returns delta of neighbors in cardinal, diagonal, and all directions for a square, and a triangle grid. The method is implemented in the super-class because we assumed that a triangle cell can have 8 neighbors maximum like a square cell. In the future this can be extended easily to implement different numbers of neighbors to check in different directions. For example, a triangle cell can have 3 or 12 neighbors instead of 8.

The grid class has methods that create the grid the first time the program loaded, display it, and update it throughout the life of the program. When the program first starts the grid receives information about the states of a simulation from the xml file. It also receives information regarding the shape of the grid from GridShape super-class. With this information, a method in the grid class creates the attributes of the grid and saves them in a map. Then this method calls another method specialized only in the visual rendering of the graph. In the beginning these two methods were combined together, but I decided to separate them since they have completely two functionalities. This made it easier to debug the code, and pass less parameter. The separation of the two methods proved valuable later when I needed to recreate the grid when the shape is changed by the user from the GUI. Once the grid is created, it is updated every cycle depending on the new states.

For the control panel, I implemented Four main super-classes: buttons, sliders, checkbox, and choice-box where each had a number of subclasses. The control panel class' main functionality is creating new instances of the features when the program is first initialized, reset them, and set their position on the stage. This made it easy to add new buttons or slider .etc to the panel because I did not have to pass a long list of unnecessary parameters to their constructors. Each component has a listener that reset, or updates variables in GUIVariables.java.

My main goal in addition to having required features was to make the code as flexible as possible for two reasons. First, I wanted the code to be able to transition from phase 1 to 2 as smoothly as possible. Second, I wanted to create as little dependency as possible between different components of the program. The code struggled with this before refactoring it. I had a meeting with my team and asked them to design code that passes the fewest number of parameters to and from GUI. Refactoring the code helped separate between simulation states, and their representation. These changes made the code flexible enough to support the new extensions without having to make heavy changes.

### feature1: XML reader:

This class takes in an xml file, parses it, and returns variables pertain for the initial parameters of the program, and specific simulations. The class implements error checking at the begging such as stack exceptions and IO exceptions. This class takes into account not all simulations will implement all fields defined. In this case null means random initialization for example, or probability does not apply. In case a new field is added to the xml file, a new corresponding field can be added without changing the original design in this class. Hence the class is closed in that sense. This class is limited in flexibility due to the fact that it deals with xml files. But it is easy to add new fields and parse them in this class.

###Feature 2: modifiable frame rate:

It was a design choice to use a slider to modify the frame rate instead of a text box or choice box for example. It does not require any type of input validation, it does not require the user to know the maximum supported rate, and it is visually appealing and easy to use. The frameRateSLider is a subclass within the slider super-class even though it is one slider. At some stage during this project, the program had three sliders to change the number of rows and columns in addition to frame rates. Having each in a sub-class made it easy to add or remove sliders. It also reduced the amount or duplicate code since sliders have similar create(). The only difference is minimum and maximum values for each. The play and reset buttons use the value from the slider. Each can get a copy of the slider and use the value to update the frame rate or reset it.

###Alternate Designs

The current design does not handle infinite number of states because it is implemented with a two-dimensional array list. From my discussion with my TA I realized that we need to change that to a more generic type of lists such as a list or a linked list instead to allow this feature. I discussed that with my teammates but they preferred to keep the current design because of lack of time. Using array list in our current design is enough to make all the current features work. It is also convenient for keeping track of locations of cells on the graph. A list can still be used as easily, and it also has the ability to return object by location.

The other design alternative that we actually implemented was creating the simulation array and the grid array once and update their cells instead of creating new ones every time a simulation is updated. This change was necessary because before that, the program speed gradually decreased even when the frame rate was high. We re-factored the code to address this limitation after I discussed the issue with my TA. This makes the program capable of supporting an infinite grid even though we did not implement it.

I also think that there is another design consideration that we need to make to optimize the program. I think that we should not keep methods (checkNeighbors(), needUpdate(), and update()) seperated and called in from gui. The main reason is that not all simulations can implement update in this order or use these three methods only. The second one is to create less dependency between GUI.java and Simulation.java. I think that a method update() should be called from gui.java, and this method implemented differently within each simulation. It could for example call check neighbors(), needupdate() or any other methods from within any simulation. Additionally, since simulation array is passed as parameter in the current implementation, I think that there is no need to create the double for loops in gui.java, instead create it in simulation class.

### best features of the current design:

1. The program ability to support any number or types of gui components easily

### issues:

1. inability to implement infinite grid

Code Masterpiece

================

refactored button class

for comparision with previous design, please check merge pull request #18

I worked on refactoring the code in this commit after the weekly meeting with my TA. Previously I defined all the buttons and their functionality in one class, even though i had a subclass for each button. Becuase the buttons were in one class and initialized by one constructor, I had to pass a long list of parameters to create them. the list included attributes common to all of them such as "description", but it also included attributes that were specific to one or two buttons.

In this commit I created an abstract class (buttonsClass) and extended it to all possible buttons the project needed. I also created a class that will later assist all other parts of the program in sharing variables called GUIVariables.java. in that class i put the parameters that the code used to pass around too almost every object, such as root, stage, rows,..etc. Then I created abstract methods in buttonsclass.java to reduce duke duplicates. Then adding new buttons became easier.