***Specification***

***Describe the primary classes (in packages if possible) you envision are needed to read in and represent a CA model and its simulation. Focus your design on how to represent a model in a general way and specifically on what* behavior *(i.e., methods) your classes would have. If describing an inheritance hierarchy, clearly identify what behaviors the classes have in common, the superclass, and what are different, the subclasses, rather than the instance variables.***

*Include the following sections:*

***Introduction***

*Describe the problem your team is trying to solve by writing this program, the primary design goals of the project (i.e., where it is most flexible), and the primary architecture of the design (i.e., what is closed and what is open). Discuss the program at a high-level (i.e., without referencing specific classes, data structures, or code).*

The objective of this project is to write a program that will implement several different Cell Automata (CA) models. The configuration details of the model will be defined in an XML file (written in the format determined by our team), and this file will be read in by the program. Depending on the information read in, the program will generate a simulation of a certain type (i.e. Game of Life, Spreading of Fire). A simulation type is identified by the set of rules it follows for updating the state of each cell in the grid for the next generation. The program will also allow visualization of this simulation by some interaction via user inputs through mouse and key. The goal of this program is that modifications and additions can be made to certain key features of the simulation. For example, the program should be flexible enough that either an entire new simulation type can be added to the code or individual elements such as new rules or new states can be added to existing simulation types. In terms of the primary architecture of the design, the architecture will be open to modifications so that a new CA model can be readily added by adding a new XML file and new subclass(es) implementing the abstract class that already exists for all simulation types. Behavior can be extended by creating these new subclasses, but the structure and scope of these new subclasses will be limited by the closed abstract superclass. This abstract class, as well as the main program involving visualizing the simulations, should stay unchanged as much as possible when adding a new type of simulation to the program.

***Overview***

*This serves as a map for your design for other programmers to gain a general understanding of how and why the program was divided up, and how the individual parts work together to provide the desired functionality. As such, it should describe specific components you intend to create, their purpose with regards to the program's functionality, and how they collaborate with each other. It should also include a picture of how the components are related. This section should discuss specific classes, methods, and data structures, but not individual lines of code.*

The classes and their relationships are illustrated in the picture below.

*<Insert picture>*

**CA (main class)**

Interpretation of the XML file, initialization and visualisation of the cells, handling user inputs (maybe separate into multiple classes). It extends Application class from JavaFx and implements the start method. It is responsible for creating the stage, scene and root for the program.

It has the following methods:

* start()
  + Creates splash scene with UI class
  + Reads in XML file
    - Contains type of simulation, dimensions of grid, constants like thresholds and initial states of the cells
  + Calls initialize method
  + Creates simulation scene
  + Sets transitions between scenes
  + Creates animation
* initialize()
  + Initializes grid as AbstractCell[][]
  + Initializes each cell inside the grid
  + Adds ImageView object of each cell to the root
  + Initialize Rule
* step( time )
  + Calls Rule.apply() to update all the elements in grid AbstractCell[][]
* handleUserInput()
* main()
  + Launches the start method

**AbstractCell**

AbstractCell has field parameters such as and int representing the state of the cell, an ImageView object to appear in JavaFx scene, and two int to represent the index of the cell in the cell grid, and a AbstractCell[][] (the grid), a double variable width and a double variable height, a Group JavaFx node root, an int called nextState representing the next state of the cell (this variable is necessary because we want to change the states of all cells in one batch).

It has the following methods:

* public List<AbstractCell> getNeighbors ()
* public void updateNextState ()
* private void updateState ()
* private void changeImageView (int state)
* public int getState ()
* public ImageView getImageView ()

Each simulation will require the following 3 subclasses:

**SpecificCell extends AbstractCell**

Have specific int state variables. (such as FIRE = 1)

Have other specific variables such as energy of a shark.

Constants from XML need to be passed as parameters for the constructors (i.e. *probCatch*).

Implements the four abstract methods.

**EdgeSpecificCell extends SpecificCell**

Overrides the getNeighbors () method.

**CornerSpecificCell extends SpecificCell**

Overrides the getNeighbors () method.

**Rule**

Rule is constructed after AbstractCell[][] is initialized. Has AbstractCell[][] as the field parameter. In method public void apply(), Iterate through every cell and call cell.updateState() on each one.

**ReadXML**

This class will have a method to interact with the XML file and additional helper methods called getCAtype(), getCAparams() which will return the related values parsed from the XML file. Its constructor will require an XML file.

**UI**

This class will have a splash page method which will be setting up the splash page with the 4 options(Game of Life, Segregation, Predator-Prey and Fire). There will also be a class handling the transition between the splash page and the simulation screen. In other ways, after the user click on one of the four buttons, the simulation screen will be displayed accordingly. There will also be a method setting up the simulation screen. There will also be a class handling user input in the simulation screen.

//XML file example

<?xml version="1.0" encoding="UTF-8"?>

<cell\_society>

<simulation>

<name>Segregation</name>

<tile>Segregation\_1</title>

<author>Team13</author>

</simulation>

<params>

</params>

<dim&config>

<width>w</width>

<height>h</height>

<length>l<length>//length of the side of the cell

<agentX>x</agentX>

<agentO>o</agentO>

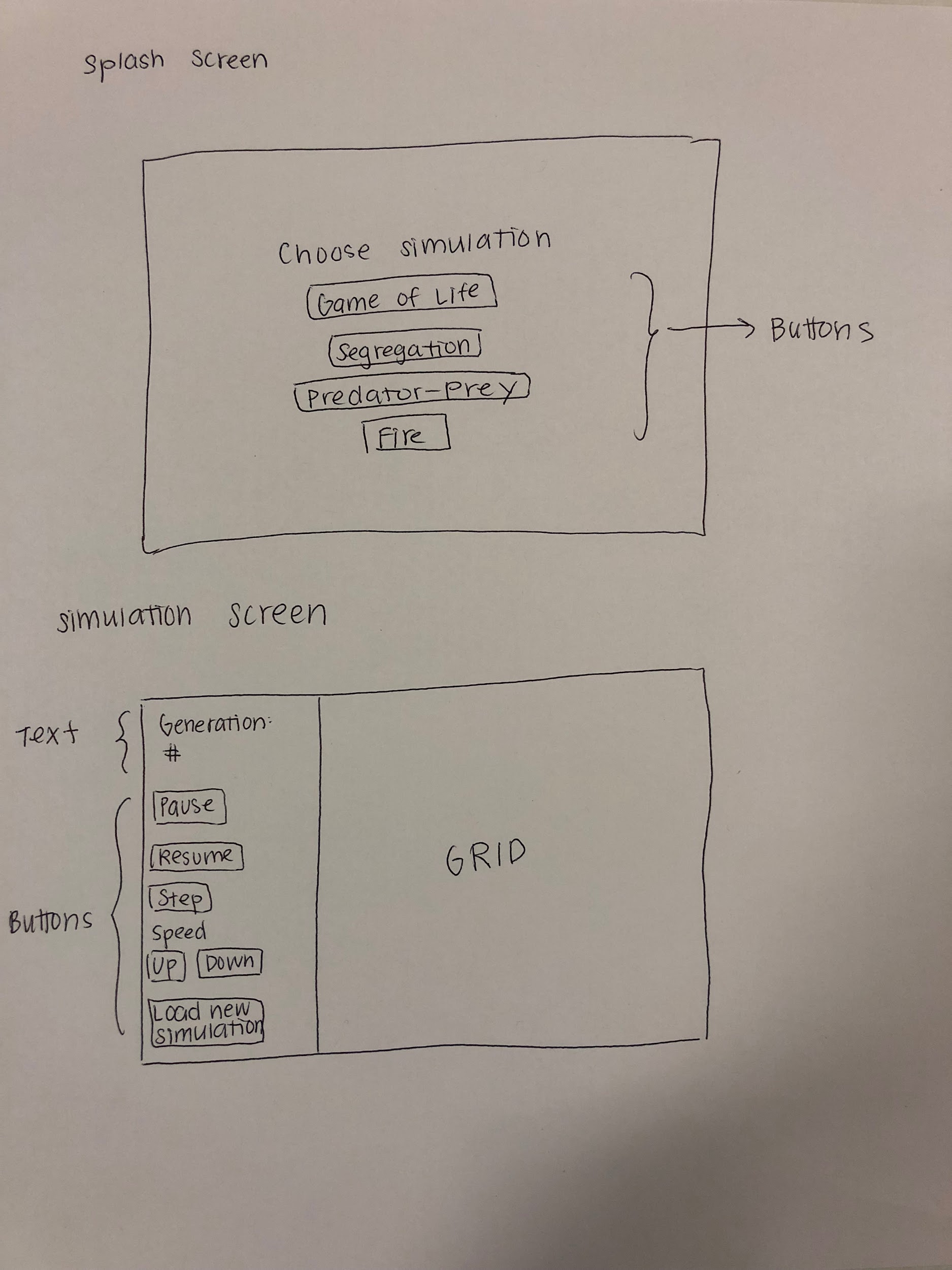
<empty>e</empty> //percentage x+o+e = 100

</dim&config>

</cell\_society>

***User Interface***

*Describe how the user will interact with your program (keep it very simple to start). It should describe the overall appearance of program’s user interface components and how users interact with these components (especially those specific to your program, i.e., means of input other than menus or toolbars). It should also include one or more pictures of the user interface. Finally, it should describe any erroneous situations that are reported to the user (i.e., bad input data, empty data, etc.). This section should go into as much detail as necessary to cover all your team wants to say.*



**Appearance & User Interaction**

*Splash page:*

There will be a splash page with program options when the program is run. The user will have choices of simulation types which will appear as buttons. Clicking on one of buttons will start the animation with the corresponding XML file at a default speed.

*During simulation:*

The simulation screen will be created based on the dimension determined in the XML file. These dimensions will not be changeable once the simulation has started. On the simulation screen, as the simulation proceeds, there will also be buttons to *Pause, Resume,* and *Step* through the simulation. Pressing the *Step* button will advance the simulation to the next generation and remain paused there until another button is pressed. There will also be speed *Up* and *Down* buttons*,* which will alter the speed of the animation by incrementing or decrementing the current value of frames per second each time either button is pressed. There will be another button *Load New Simulation* which will stop the current simulation, and bring the user back to the splash page. Additionally, this screen will display the number of generations that the simulation has passed through since starting.

*Error Handling*

If the selected simulation prompts the program to try to read a file that is empty or has bad data, the user will be alerted with an error message that pops up on the simulation screen. Because the user input is limited to clicking different buttons, there are no other issues with bad input data that we have to consider. If the user tries to increase or decrease the speed of the animation past certain bounds that have been decided within the program, the speed will no longer change in that direction when the buttons are pressed.

***Design Details***

*This section describes each component introduced in the* ***Overview*** *in detail (as well as any other sub-components that may be needed but are not significant to include in a high-level description of the program). It should describe how each component handles specific features given in the assignment specification, what resources it might use, how it collaborates with other components, and how each could be extended to include additional requirements (from the assignment specification or discussed by your team). Include the steps needed to complete the*

***Use Cases*** *below to help make your descriptions more concrete:*

* *Apply the rules to a middle cell: set the next state of a cell to dead by counting its number of neighbors using the Game of Life rules for a cell in the middle (i.e., with all its neighbors)*
  + Call updateNextState() method on a middle cell, which will update the instance variable nextState based on states from the list of neighbors returned by getNeighbors().
* *Apply the rules to an edge cell: set the next state of a cell to live by counting its number of neighbors using the Game of Life rules for a cell on the edge (i.e., with some of its neighbors)*
  + Call updateNextState() method on an edge cell, which will update the instance variable nextState based on the states from the list of neighbors returned by getNeighbors()
* *Move to the next generation: update all cells in a simulation from their current state to their next state and display the result graphically* 
  + Rule.apply() will iterate through the whole grid and call cell.updateNextState() on each cell. Rule.apply() will iterate through the grid again, and call cell.updateState() on each cell. Each cell.update() besides changing the state int variable, will implicitly call cell.changeImageView(int myState) to update (or add or remove) their ImageView objects in root.
* *Set a simulation parameter: set the value of a parameter, probCatch, for a simulation, Fire, based on the value given in an XML fire*
  + We have a class for reading XML file. After the value is read, it will be passed into the specific class for cell and rule initialisation.
* *Switch simulations: use the GUI to change the current simulation from Game of Life to Wator*
  + Press Load New Simulation to go back to splash page. If the user press the Water button on the screen, a new screen and a new root will be created and Water will be initialised and simulated. Each time a new root is created, a new AbstractCell[][] grid is created.

***Design Considerations***

*Describe any issues which need to be addressed or resolved before attempting to devise a complete design solution. It should include any design decisions that the group discussed at length (include pros and cons from all sides of the discussion) as well as any assumptions or dependencies regarding the program that impact the overall design. This section should go into as much detail as necessary to cover all your team wants to say.*

Several issues were discussed during the planning of this project’s design. These are explained below.

*Creating a class for cell state*

We noticed that within each simulation type and its specific set of rules, there were many dependencies between data and behavior, and the type of state. For example, in the Wa-Tor simulation model, the rules that are applied to a cell in the Shark state are different than the rules applied to a cell in the Fish state, but it’s possible to view one set of rules a subset of the other. Therefore, if there was some abstract superclass state that had a subclass for each type of state with a different set of rules within each simulation, this might enable us to determine cell state updates in an easier way. However, we decided that there were more important ways to categorize cells. We ultimately decided to have a subclass per cell position (middle, corner, edge) instead for each type of simulation. Since the state can be represented in a very simple way (with an integer representation), it makes more sense for the state to be an instance variable for each cell type. Then, the behavior that is defined by the rules can be determined in other methods within the cell subclasses of each simulation model.

*Whether to include neighbours of a cell inside the Cell Class or calculate a cell’s neighbours when they are needed from the Rule Class.*

If we write the getNeighbours method inside the Cell Class, it is easier for us to call cell.getNeighbours() in the Rule Class for updates. However, this will require the Cell Class to know where itself is in the grid. This can be achieved by either using location variables or passing into the Cell Class the whole grid. The latter one might be very troublesome and waste memory. We also considered the option of calculating the cell’s neighbours when we need them in the Rule Class. For this one, we have the whole grid so we know where each cell is.

*Putting Rule Logic in SpecificRule Subclass vs. in SpecificCell Subclass*

Once we had determined the inheritance hierarchy of the AbstractCell superclass, we faced a problem with redundancy in terms of how the rules would be applied. If the simulation-specific rules were applied in the method Rule.apply(), then this would require several subclasses within the Rule class for each type of simulation. A benefit of this would be that simulation-specific parameters (such as fish reproduction cycle in Wa-Tor) would only have to be stored in one Rule subclass, instead of storing them in every instance of a SpecificCell subclass. However, to reduce this idea of redundancy and reduce the amount of information that the Rule class needs to know about how to apply the rules, we decided that it was a better design choice to keep the simulation-specific data and behavior in the subclasses of AbstractCell.

*How to Read in Simulation-Specific Variables and Constants from the XML File*

There are several variables that we might need to parse from the XML file depending on the type of simulation. For example, there is the threshold *t* in the Segregation simulation, the probability *probCatch* in the Fire simulation, and several reproduction-energy related variables in the Wator simulation. We have to decide whether these variables can be read in at runtime in order to create static variables within the WatorCell. This would prevent the WatorCell from having repetitive field variables. For example, static DYINGTHRESHOLD for WatorCell instead of dyingThreshold for new WatorCell().

*Instantiate a Class by Name with Reflection*

We considered using reflection so that, for example, when we read the XML file and decide the simulation type is Wator, we can let CA instantiate specifically a WatorCell instead of other kinds of cells. Although it might be a challenge to figure out how to use the unfamiliar concept of reflection here, it may be a worthwhile decision in order to allow for this flexibility in the code. <https://stackoverflow.com/questions/9886266/is-there-a-way-to-instantiate-a-class-by-name-in-java>.

***Team Responsibilities***

*This section describes the program components each team member plans to take primary and secondary responsibility for and a high-level plan of how the team will complete the program.*

* CA: A main class to run simulation from -Haotian, Julia, Yunhao
* ReadXML: A class for reading XML file (DOM) -Yunhao
* AbstractCell: A class for individual cell that has parameter such as its states. - Julia, Haotian
* SpecificCellA and their edge, corner subclasses: -Haotian
* SpecificCellB and their edge, corner subclasses: -Julia
* SpecificCellC and their edge, corner subclasses: -Yunhao
* SpecificCellD and their edge, corner subclasses: -Haotian
* Rule: A class for rule that calls cells to update to next generation. -Julia, Yunhao
* UI: A class for interfaces -Haotian, Yunhao, Julia

Every evening, we would discuss what we have done. We will meet once before Friday and once on weekends. We will meet more often in person or online if we encounter problems. Priority will be to build abstract parent classes first.

*Rules of the simulation versions*

* name of the kind of simulation it represents, as well as a title for this simulation and this simulation's author
  + (Game of Life, Wator, Fire, Segregation)
* dimensions of the grid and the initial configuration of the states for the cells in the grid
* settings for global configuration parameters specific to the simulation
  + Segregation -- threshold t for similarity satisfication
  + Wator -- reproduction cycle of fish and shark, energy cycle of shark, energy gained per fish eaten
  + Fire -- probCatch

*How will we differentiate between Game of Life, Wator, Fire, Segregation in our classes? What is different between them?*

* *States*
  + *Game of Life has 2: dead, alive*
  + *Segregation has at least 3: agent X, agent O, empty*
  + *Wator has at least 3: Fish, shark, empty*
  + *Fire has at least 3: Normal tree, burning tree, empty*
* *Neighbors*
  + *Game of Life*
    - *Middle cell has 9 adjacent, Edge cell has 5, Corner cell has 3*
  + *Segregation* 
    - *Middle cell has 9 adjacent*
  + *Wator*
    - *N, E, S, W*
    - *Edges wrap around (tor)*
  + *Fire*
    - *N, E, S, W*
* *Rules*
  + *Game of Life: Depends on # of neighbors alive* 
    - *Alive → dead if:*
      * *# alive neighbors is >= 4 or <= 1*
    - *Dead → alive if:*
      * *# alive neighbors = 3*
    - *State doesn’t change if:*
      * *Alive & # alive neighbors is 2 or 3*
      * *Dead & # alive neighbors is not 3*
  + *Segregation:*
    - *Relocate if:*
      * *Less than t percent of neighbors are the same state*
      * *Freedom to define relocation, i.e.*
        + *Move to random empty cell*
        + *Move to nearest empty cell*
        + *\*Have to keep track of where disatisfied cells are relocating to*
  + *Wator: Depends on cell state, shark depends on energy, both depend on # generations*
    - *Fish: Check N, E, S, W* 
      * *If one is empty, move there (random if many)*
        + *Check # generations & compare to reproduction cycle, if reproduce then leave a fish in old position, reproduction resets*

***need to keep track of # generations***

* + - * *If none empty, don’t move*
    - *Shark: Check N, E, S, W*
      * *Deprive 1 unit of energy* 
        + *If energy is 0, set state to empty (shark died)*
      * *If one is fish, move there*
        + *Gain energy*
      * *If none are fish, move to empty neighbor* 
        + *Check # generations & compare to reproduction cycle, if reproduce then leave a shark in old position, reproduction resets*
      * *If none are empty, don’t move*
  + *Fire: Depends on probcatch*
    - *If empty:*
      * *Stay empty*
    - *If burning:*
      * *Change to empty*
    - *If normal tree:*
      * *Check N, E, S, W* 
        + *If one is burning, determine next state based on probCatch*