**The Game of Life**

CMSC 495 – Section 7982

Current Trends and Projects in Computer Science

Damien Manier

Aaron Moyer

Kimberly DeMarr

November 8, 2015

Table of Contents

[**Overview** 3](#_Toc433961679)

[Individual Contributions 4](#_Toc433961680)

[**Project** **Plan** 6](#_Toc433961681)

[Methodology 6](#_Toc433961682)

[**Requirements** **Specification** 9](#_Toc433961683)

[**System Specification** 12](#_Toc433961684)

[**User’s Guide** 13](#_Toc433961685)

[**Test Plan and Results** 16](#_Toc433961686)

[Test 1: A Clickable Grid 16](#_Toc433961682)

[Test 2: Implement the Three Rules 18](#_Toc433961682)

[Test 3: Random Patterns 20](#_Toc433961682)

[**Design and Alternate Designs** 23](#_Toc433961687)

Architecture for the Game of Life [23](#_Toc433961682)

[Entity Relational Diagram 32](#_Toc433961682)

[Component Diagram 33](#_Toc433961682)

[**Development History** 36](#_Toc433961688)

[**Conclusions** 37](#_Toc433961689)

[**References** 38](#_Toc433961693)

# Overview

In the late 1960s, mathematician, John Conway, created the no well-known cellular automaton, The Game of Life (Roberts, 2015). Not only did this game gain popularity with mathematicians and computer scientists but also with physicists, economists, philosophers and many others who use the Game of Life as an analogy to prove their own studies. Multiple rows and columns of cells make up a two-dimensional grid and there are only two possible states that cells can be in; either dead or alive. Once the simulation has started, each cell relies on three rules that determine if a cell lives, survives, or dies.

1. Death: A live cell dies if it has less than two or more than three live neighbors.
2. Survival: If a live cell has two or three live neighbors, it lives on to the next generation.
3. Birth: If a dead cell has exactly three live neighbors, that cell becomes alive.

(Weisstein, n.d.).

Various patterns are formed as the game progresses and this is all dependent on both the initial state of live and dead cells, and the rules. The rules are applied to every cell simultaneously, so birth, survival and death occur concurrently as well. A group of cells can be in three pattern types known as still life, oscillator, and spaceship. Understanding these terms will be essential during test planning. Still lives are patterns that are static and do not change through each generation. The picture below displays an example of a still life.

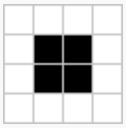
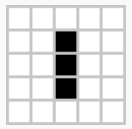
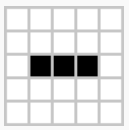
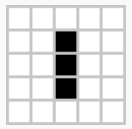
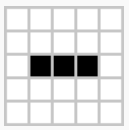


Figure 1 - Weisstein, n.d.

Unlike a still life, an oscillator is when a pattern “cycles through a set of configurations”, in the same general location on the grid (Weisstein, n.d.). The following pictures display an oscillator that cycle through two patterns in a 4 generation time period.

**Figure 2 - Weisstein, n.d.**

Lastly, spaceships are a series of patterns that glide across the board. Spaceships are generally made up of oscillators.

## *Individual Contributions*

In order to successfully implement all of these aspects into a working program, each group member in Automatons are contributing their knowledge and skills to a specific part of the project. Aaron Moyer will focus on constructing the front-end graphical user interface (GUI) as well as creating the User Guide and Project Design. The GUI will provide various buttons and menus that allow the user to fully interact with the grid in different ways. Damien Manier will work on the back-end programming which will comprise of the rules, user input, functions and calculations. Damien has also collaborated with Aaron on the Project Design and constructed the Entity Relational Diagram that is presented in that section. Kimberly DeMarr will work on developing all other documentation and plans as well as project management. Kimberly will also demonstrate her knowledge in programming when help is needed throughout the entirety of project development. In order to easily collaborate on the program, the coding repository, GitHub is being utilized. This allows each member to easily review, share and manage our program.

**Project Plan**

In order for a user to implement the Game of Life, the version JDK 7 or higher will be required, along with any Java IDE of the users’ choice. There are no data files needed to be read into the Game of Life. The cellular automaton will receive its information through user input on the clickable grid and provided buttons. This will allow the input to vary at the beginning of each game.

## *Methodology*

Software engineering methodology is a common practice when developing an information system. Utilizing software engineering methodology can ease the system development process by providing steps on evaluation and finding defects within a system. A basic framework of a system development is provided below.

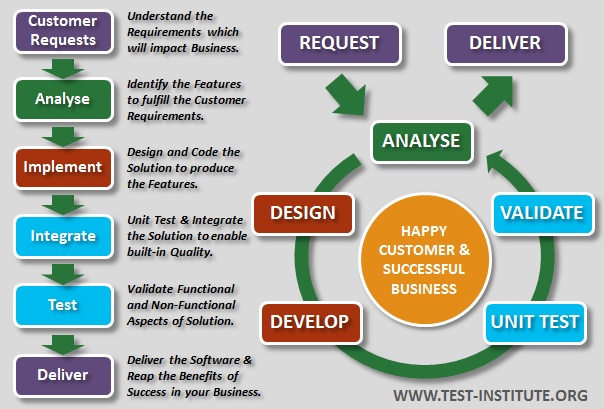


Figure 2 - International Software Test Institute, n.d.

The areas that would be affected by the implementation of the above framework would be analysis, implementation, testing, and validation. For the analysis stage, it is essential that we include all basic components that make up a simple Game of Life. From this, we are able to add more features to make our own unique version of the game. To name a few of our features, we will allow the user to adjust grid size, provide a random pattern generator and a button to view a specific generation in the game life.

The implement phase is made up of both design and development of the code. Design relates closely to GUI by providing the necessary buttons, drop down menus and displays that are necessary to implement all features provided in the analysis stage. Our goal is have a user friendly GUI that is easy to understand. By providing the user with instructions and having a clean and sleek layout is essential. Then the development of the code will make the design a reality.

In order to determine if the coding is a success, testing and validating is essential. We will focus on the following three scenarios that will guide us for any improvements.

1. Demonstrate known patterns.
2. Allow user to experiment with customized patterns.
3. Discover patterns through random generation.

The first scenario will be tested through the pre-generated patterns provided to the user. Expected outcomes will be compared to the outcomes of the given patterns. The second scenario will be tested through the clickable grid. We will be looking for any errors that may occur. Lastly, a random button will be available for the user to choose and this will be utilized in the last scenario. Testing on all other provided buttons and menus will be accomplished as well. With continuous tests, we will be able to find any defects that may be lingering within our code.

After all of these steps are completed, the game will be distributed out to various test users for peer reviews. Based off of the user feedback, the testing areas will be revisited until the final product is error free.

# Requirements Specification

This project is based off of an already existing application, the Game of Life, which is designed to present cellular automata. We plan to recreate the application but incorporate different aspects of multiple online versions we’ve seen in order to make our project unique. In order to gather all of the requirements for the project, online research needed to be done. There is a great amount of information about John Conway’s the Game of Life, so it is essential that we determine what the most important aspects were. After visiting multiple websites that explain and present the Game of Life, we have noticed basic requirements that appear in every example so they have been included in our project design. These requirements are the 3 game rules, interactive grid, start and stop button. These basic requirements can be considered the most important as they form the essential foundation or starting point for every online example of the Game of Life our group has come across.

Along with those basic requirements, other necessary conditions have been determined that will allow our application to fully function as it is supposed to. The ten requirements are

* **3 rules of the game –** The rules are one of the basic requirements and it is the core of the entire game. Implementing the rules is essential for the program to provide correct output.
* **Interactive grid –** This is another basic requirement as it allows users to create their own initial design.
* **Random pattern generator –** This requirement is presented to the user through a button. When the user chooses this button, a random pattern is presented on the grid. This is not considered an essential aspect of the program, but it is helpful for the user if time constraints are an issue.
* **Start button –** This is essential to the game of life as it starts the process of applying the rules to each cell.
* **Stop button –** This is another essential requirement to the project because it halts the game in the current generation. Freezing the game in its current state allows the user to examine the pattern changes in depth.
* **Clear button –** This button clears the grid of the current pattern. Providing this button will mean that users do not have to exit out of the window and restart the program in order to test another pattern.
* **Generation lapse timer –** This is presented in a ticker which allows the user to choose how many seconds each generation lasts. This is useful if the user wants more time to study the pattern change at each generation. Our group decided to make this a requirement so the user does not have to constantly click the start and stop button at each generation.
* **Generation counter –** This is not one of the core requirements for the program, but including it will be helpful when doing comparisons just like the tests performed in Test Planning.
* **Straight forward design of the GUI –** Having a straight forward GUI design will allow user to easily understand and interact with the program.
* **Drop down menu with pre-determined patterns –** This allows the user to test against expected outcomes with specific initial designs.

All of the requirements are user functional except for the straight forward design of the GUI. The purpose of the straight forward GUI design is to allow the user to easily interact with the functions of the other requirements.

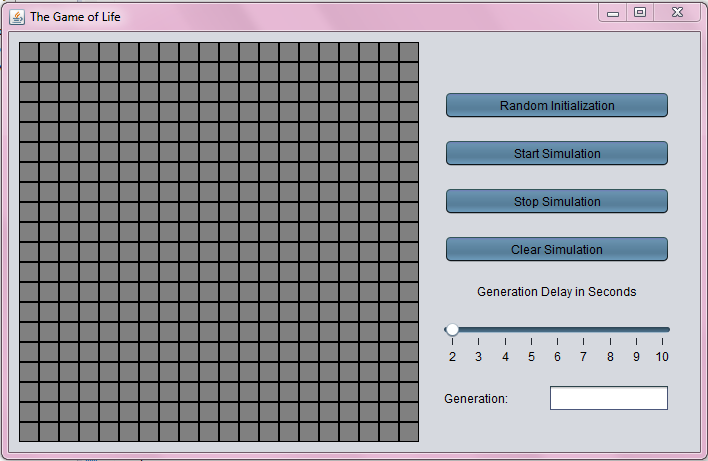
From the large amount of information found online about the Game of Life, a list of requirements was created to help guide the creation of our project. The requirements are made up of core features for the game as well as other characteristics from multiple versions that our group thought would be beneficial in our project. All requirements were successfully implemented and many were displayed in our sleek and user friendly GUI.

# System Specification

TBD

# User’s Guide

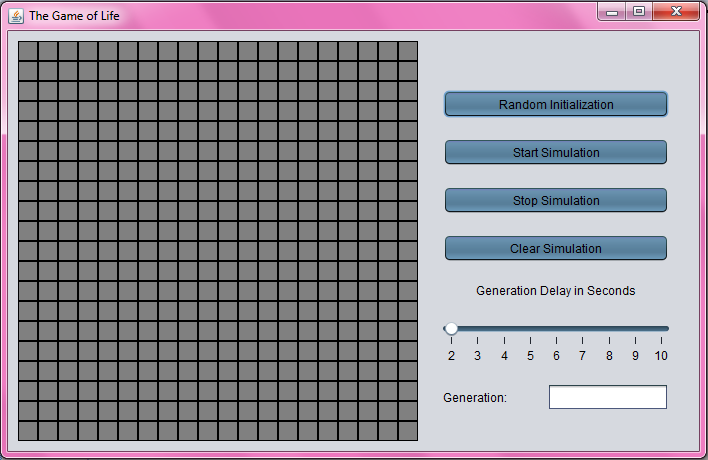
The Game of Life can be started by using Java Virtual Machine (JVM) 7 or higher and the user’s choice of a Java IDE. Once the program has been started, the following screen will appear.



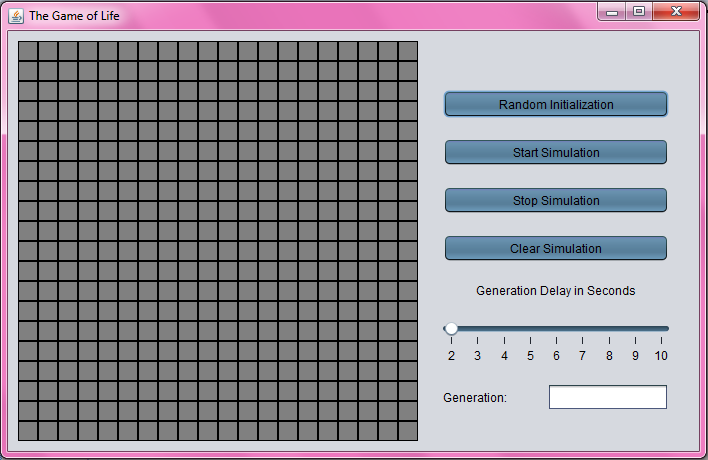
The user can then begin trying out different patterns to see how the application operates. The user can click on the grid square of their choice and make any kind of pattern that they want to experiment with in the Game of Life.

|  |  |
| --- | --- |
| Example of user clicking on grid squares to make a custom pattern: |  |
| Example of user clicking all grid squares: |  |

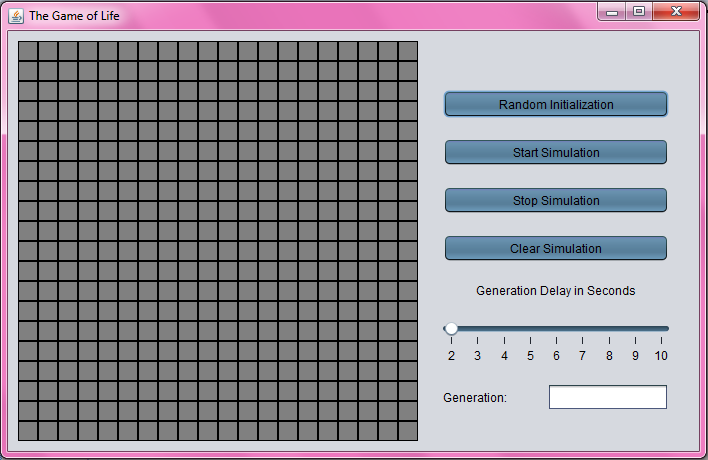
Another way to acquire an initial pattern is by choosing to select the Random Initialization button located on the right side of the screen. Below are explanations of each buttons function.

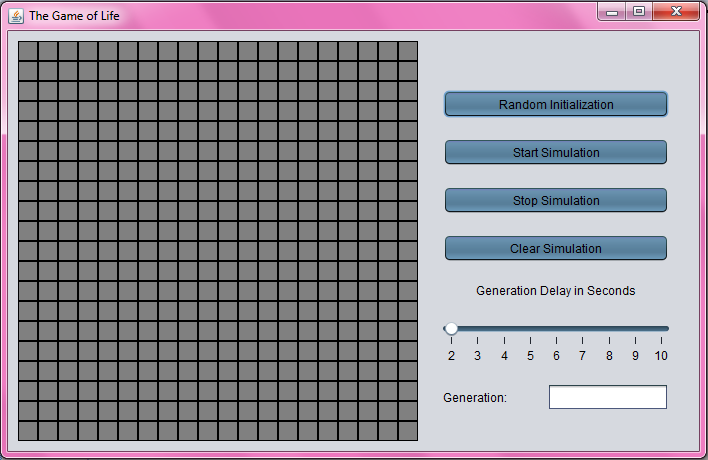
 **Random Initialization** can be used to populate the grid with a random number of cells that will be filled the color Green.

|  |  |
| --- | --- |
| Example Output of Random Initialization: |  |

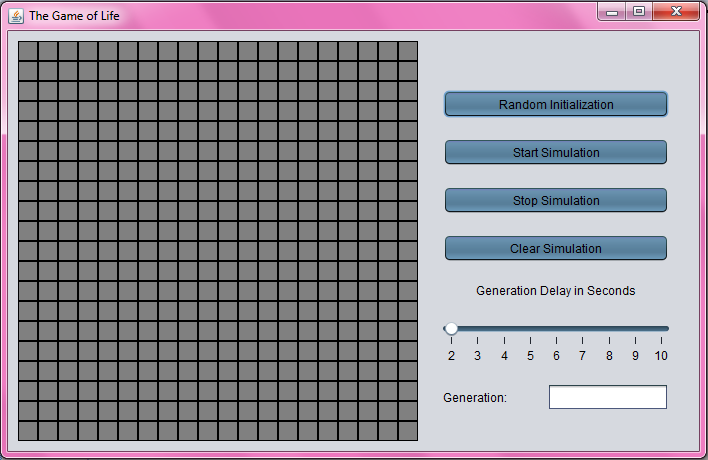
 **Start Simulation** will start the simulation of the Game of Life and the following rules will be applied to each cell:

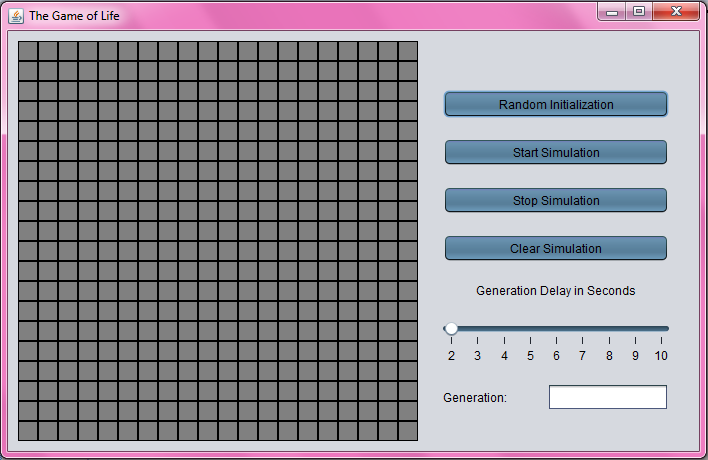
* A live cell dies if it has less than two or more than three live neighbors.
* If a live cell has two or three live neighbors, it lives on to the next generation.
* If a dead cell has exactly three live neighbors, that cell becomes alive.

 **Stop Simulation** will make the simulation halt in its current generation, and what is populated and unpopulated on the grid will stay until started or cleared.

 **Clear Simulation** will clear out all of the cells on the grid and turn all of the cells gray.

|  |  |
| --- | --- |
| Example Output of Cell Simulation Cleared: |  |

 The **Generation Delay** Slider can be used to toggle between how long it takes the Game of Life to complete a generation and move on to the next. This means that the user can make the patterns change every 2 seconds to every 10 seconds per generation.

 The **Generation Count** field is used to count how many generations the Game of Life has gone through. This will be a number such as 3 or 27 for that many generations that has passed.

# Test Plan and Results

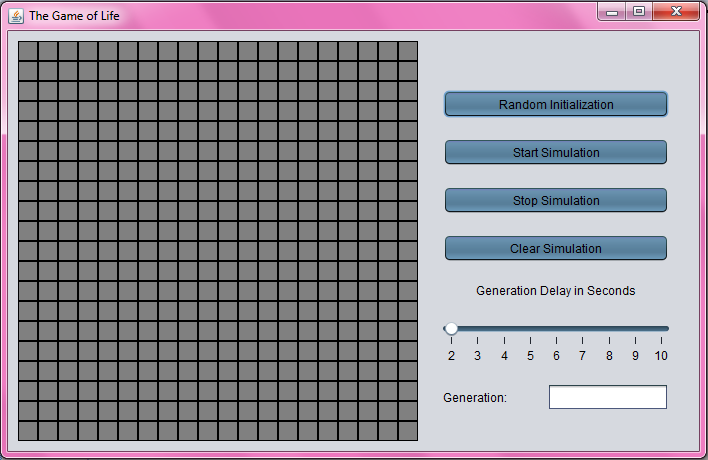
Previously mentioned in this document were the following project requirements.

* Acquire a clickable grid
* Implement the 3 rules
* Provide random patterns

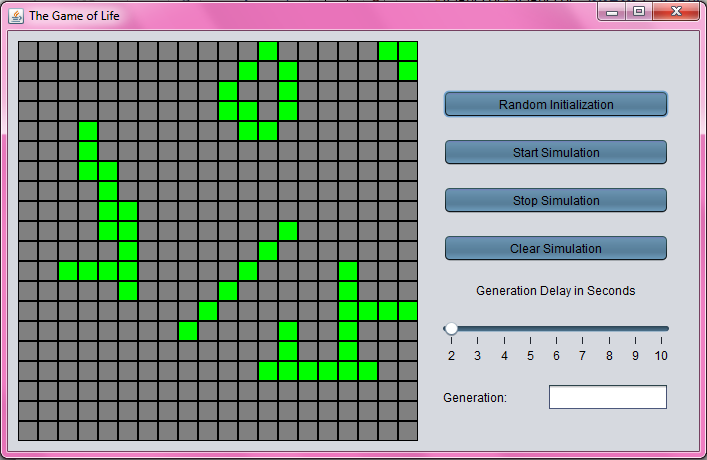
These requirements are guidelines for our group to follow in order to successfully design the program. Specific tests will be performed in order to thoroughly ensure that we completed all requirements. Damien and Aaron created the program thus far and Kimberly will be completing the following tests.

*Test 1: A Clickable Grid*

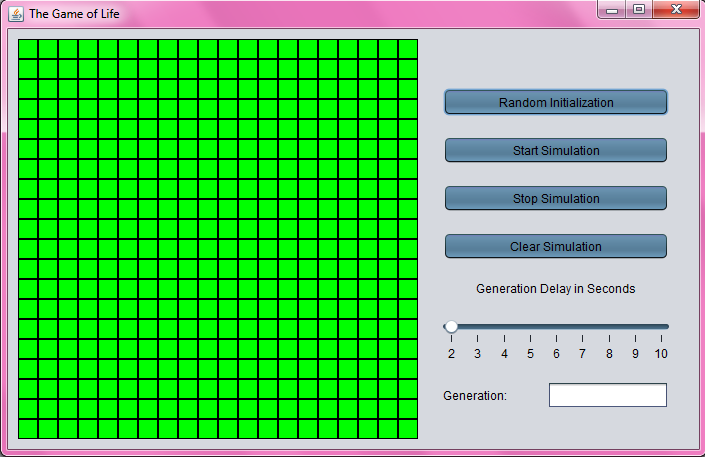
The clickable grid could be considered one of the most important aspects of our project. Without having a complete functional grid, users will not be able to interact with the program in its entirety. In order to show the user that there is interaction occurring within the grid, each cell will turn green if clicked on. Once the user opens and runs the program, the following screen will be displayed.



As you can see, the initial state of the grid is currently empty as all cells are gray. To see if the user is able to interact with the grid, random cells will be clicked on.



To further test this requirement, every cell will be clicked on to prove that the entire grid is interactive. In order to pass this test, each cell must be green.



In both tests, the selected cells did turn from gray to green which shows that the requirement of acquiring a clickable grid has been met.

*Test 2: Implement the Three Rules*

The three rules that were previously mentioned in the overview are the backbone of the whole application. Without correctly implementing each rule, the output patterns provided to the user will be incorrect. In order to test this, known initial patterns from an online implementation of the game will be used for this testing. These patterns will be entered into our grid, will run through the program, and then our final output will be compared to the expected outcomes. Since it is unknown how long each pattern will continue to change, the program will be stopped at generation 20. The pattern provided at generation 20 is the expected outcome that we will be testing against. The following screenshot provides the known “Acorn” pattern along with the expected outcome on generation 20.

|  |  |
| --- | --- |
| **Initial Pattern** | **Expected Outcome** |
|  |  |

Table 1 – Conway’s Game of Life, 2010

The initial input provided above is then recreated within our own grid.

|  |  |
| --- | --- |
| **Our Input** | **Our** **Outcome** |
|  |  |

When our results are compared the expected outcome, the patterns are identical. For more validation, another known pattern will be tested. Below is the “R-pentomino” initial pattern with its expected outcome at generation 20.

|  |  |
| --- | --- |
| **Initial Pattern** | **Expected Outcome** |
|  |  |

Table 2 - Conway's Game of Life, 2010

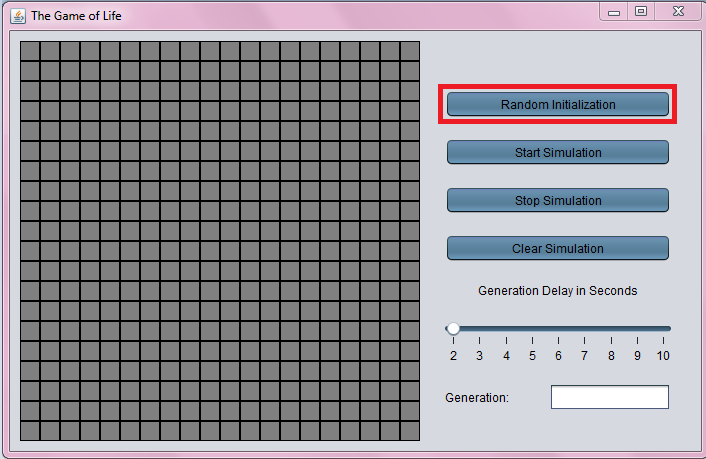
The R-pentomino pattern is then recreated within our program and is ran until generation 20. As we can see below, our outcome matches perfectly with the expected outcome once again

|  |  |
| --- | --- |
| **Our Input** | **Our Outcome** |
|  |  |

Running two known patterns through our program have successfully proven that our outcomes match up to the expected output. This is another test that our application passes.

*Test 3: Random Patterns*

Lastly, the availability of random patterns to the user will be tested. Clicking on individual cells and creating a pattern can be time consuming. So giving the option to provide random patterns can eliminate that step for users who are strictly interested in seeing the program operate. Our group has provided this option through the “Random Initialization” button located on the right side of the window.



In order to test if each pattern given is random and unique, the button will be pressed consecutively 3 times.

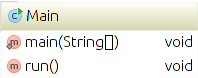
|  |  |
| --- | --- |
| **First Random Initialization** |  |
| **Second Random Initialization** |  |
| **Third Random Initialization** |  |

As can be seen above, when the Random Initialization button is pressed consecutively, the patterns are not identical. The test to see if random patterns are provided has been successfully passed.

# Design and Alternate Designs

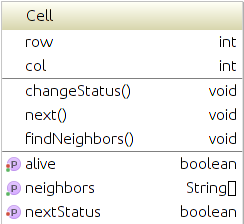
***Architecture for the Game of Life***

1. Main.java



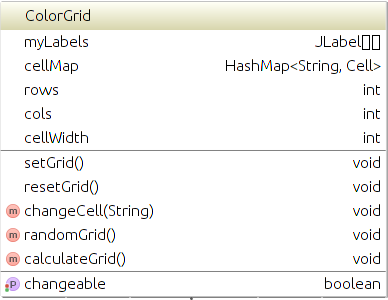
* The main class for the program to run from that implements Runnable() method
  + Utilizes the Nimbus look and feel to make the GUI more aesthetically  
    pleasing
  + Uses the run() method to run the thread with TopFrame()

2. Cell.java



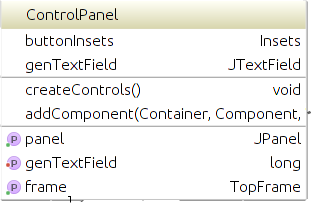
* Java class that makes up attributes of the cells
  + Stores its location and the location of its neighbors in the grid
  + Uses isAlive() method to retrieve the status of a cell
  + Sets the status of a cell to a specified value with setAlive() method
  + Makes use of changeStatus() method to change the status of a cell to alive or not
  + Uses next() and setNextStatus() methods to check and make sure all cells are alive or not
  + Uses the String getNeighbors() method to return neighbors for other methods to use
  + The findNeighbors() method algorithimically determines the position of all potential neighbors based on the position of the cell

3. ColorGrid.java



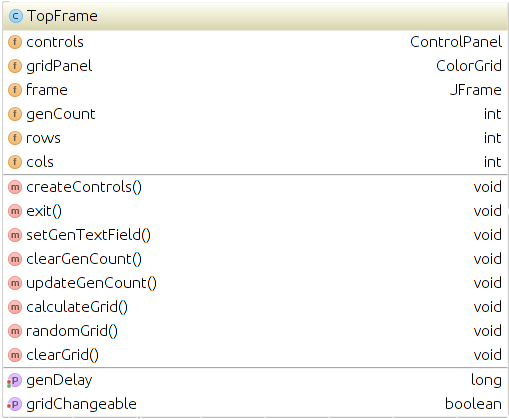
* Java class that creates a grid of opaque JLabels that are either grey or green based on their status and can be changed by clicking on them
  + Constructs a color grid that uses rows and columns and sets them to changeable
  + Uses methods isChangeable() and setChangeable() to make the grid clickable by the mouse
  + Method setGrid() used for the initial creation of the grid
    - Sets labels for the grid with JLabel
    - Uses the GridLayout for the rows and columns
    - Uses a HashMap called cellMap for cell position
    - Next give the grid lines by using a border
    - Use the cell width to make a square on the grid
    - Uses a for loop to make the JLabel opaque for every spot and then make the cells gray and respond to mouse clicks
  + Uses a method to reset the grid to a clean state
    - Uses a for loop to make a cells gray again
    - Uses another for loop to make cells not alive
  + Method used to make the cells set to alive and make them green when clicked
  + Method to calculate which neighbor cells are alive or not according to the game of life rules

4. ControlPanel.java



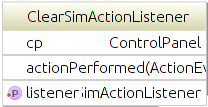
* Java class that creates the controls and places them on a panel. The control panel is the means by which all of the listeners and buttons interact with the TopFrame.
  + Variable that specifies the amount of space between buttons
  + Variable to make use of TopFrame
  + Variable to define a JPanel for the controls
  + Create a text field for the generation count
  + Construct the Control Panel
  + Method called createControls() that create the buttons and controls to be added to the panel
    - Adds the Listeners for all of the buttons that are on the panel
    - A new panel for the buttons is added
    - The layout is set to GridBagLayout
    - The layout is also started at the top row address of the panel
    - The Random button is added to the panel
    - The Start button is added to the panel
    - The Stop button is added to the panel
    - The Clear button is added to the panel
    - The slider is added to the panel with its generation delay functionality
    - The generation label is added to the panel
    - The generation text field is added to the panel
  + Method to add all of the components using GridBagLayout
  + Method that makes use of TopFrame
  + Method to set the generation text field
  + Method to get the panel and return it for use by other classes

5. TopFrame.java



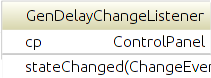
* Java class that creates the frame to place all of the panels into. The TopFrame is the means by which the ControlPanel and its child processes interact with the ColorGrid and vice versa.
  + Variables for controls, panels, and rows/columns
  + Constructor TopFrame that has createControls in it
  + Method used to create the controls for the frame
    - Sets rows, columns, generation count, and generation delay
    - Instantiates a control panel
    - Instantiates a ColorGrid
    - Sets up the top frame with JFrame
    - Creates a top level panel with JPanel and uses FlowLayout
    - Adds the grid and the control panel to the frame
    - Sets the frame settings such as the layout, size, and making it visible
  + Method to be able to exit the frame
  + Wrapper over control panel setGenTextField
  + Method to reset the genCount to 0
  + Method to increment the generation count
  + Method to return how long the delay between calculations is
  + Method to tell if the grid is in the changeable state
  + Wrapper for the ColorGrid called calculateGrid() method
  + Wrapper for the ColorGrid called randomGrid() method
  + Wrapper for the ColorGrid called resetGrid() method

6. ClearSimActionListener.java



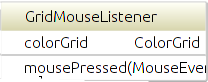
* Java class to add an action listener to the clear simulation button
  + Variable that maintains relationship with ControlPanel class
  + Variable to be able to use the start listener
  + Constructor for clear sim action listener
  + Constructor called setListener to make use of listener variable
  + Method for the action performed by the clear button
    - Clears the grid
    - Makes the grid editable
    - Can clear the generation count
    - Can stop the simulation from running

7. GenDelayChangeListener.java



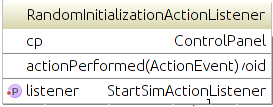
* Java class that makes use of ChangeListener for a generation delay
  + Variable that maintains relationship with ControlPanel class
  + Constructs a generation delay change listener that uses ControlPanel to communicate to TopFrame
  + Method for when the state is changed and actions associated with it
    - Creates a source variable to get actions from the slider
    - If the slider is moved it gets the source and does a change calculation

8. GridMouseListener.java



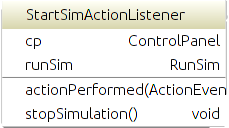
* Java class that makes use of MouseAdapter to change the grid status
  + Makes use of ColorGrid
  + Constructs a listener called GridMouseListener
  + Method used to change the grid with mouse clicks
    - Clicks are acknowledged if grid is in changeable state
    - Uses methods to change the grid from gray to green and vice versa

9. RandomInitializationActionListener.java



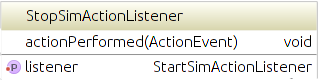
* Java class that makes use of ActionListener to know when Randomize actions need to be performed. Communicates to the TopFrame via ControlPanel which then tells the ColorGrid to randomize.
  + Variable that maintains relationship with ControlPanel class
  + Uses a variable called listener to make use of the start button
  + Constructor for the Random button
  + Constructor for the setListener that uses listener from StartSimActionListener

10. StartSimActionListener.java



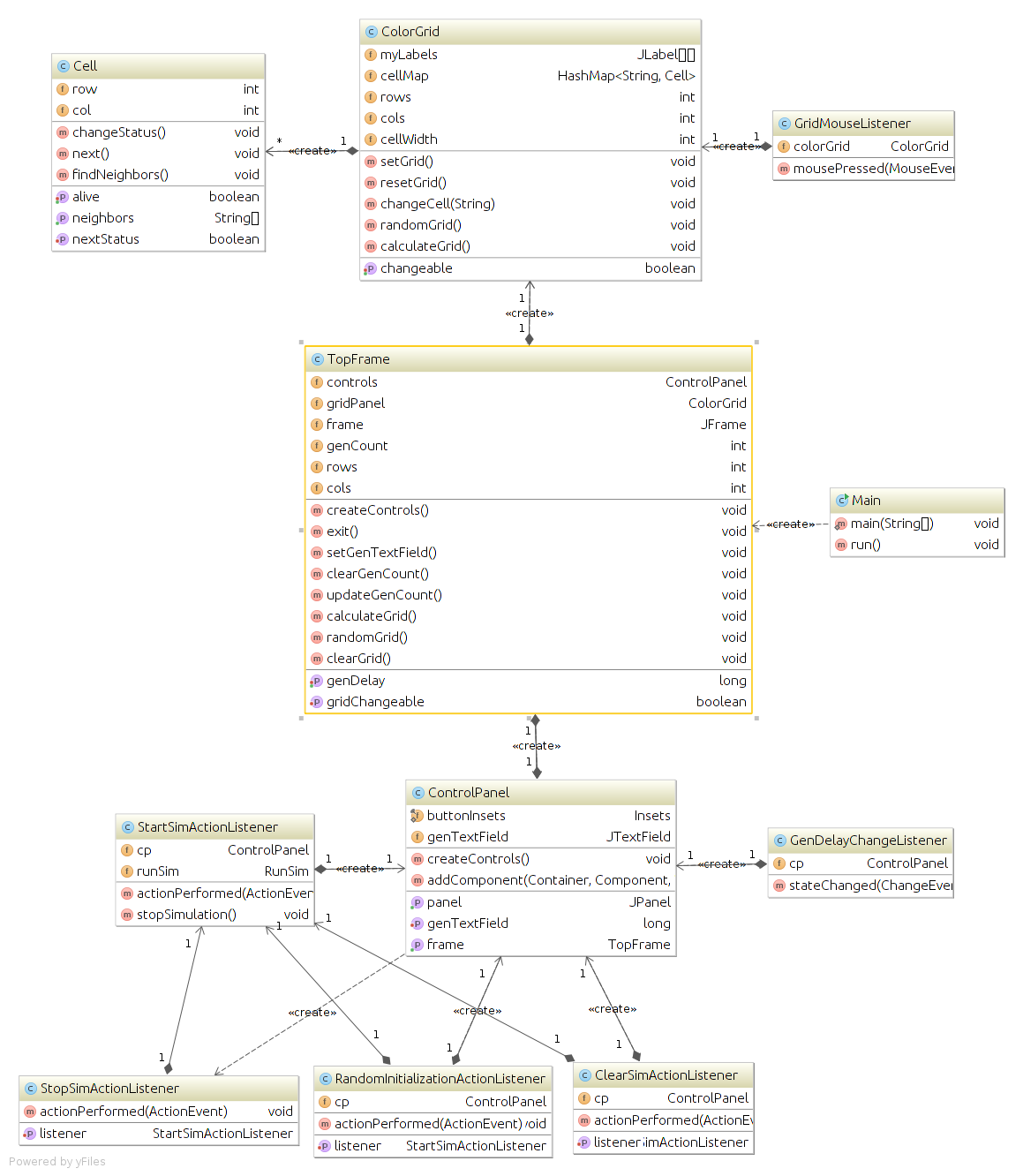
* Java class that makes use of ActionListener to create actions for the Start button
  + Variable that maintains relationship with ControlPanel class
  + Variable called runSim to run the simulation
  + Constructor for the start listener that uses ControlPanel to communicate to TopFrame
  + Method to define what start button does
    - Uses the runSim method
    - Starts a thread running of runSim
  + Method to stop the simulation from running
  + Class that implements Runnable
    - Variable that sets running to true or false
    - Run method
      * Sets thread running to true
      * Communicates with TopFrame via ControlPanel to update generation counter and to start the simulation with the ColorGrid
    - Method to put the thread to sleep
    - Method to stop the simulation

11. StopSimActionListener.java

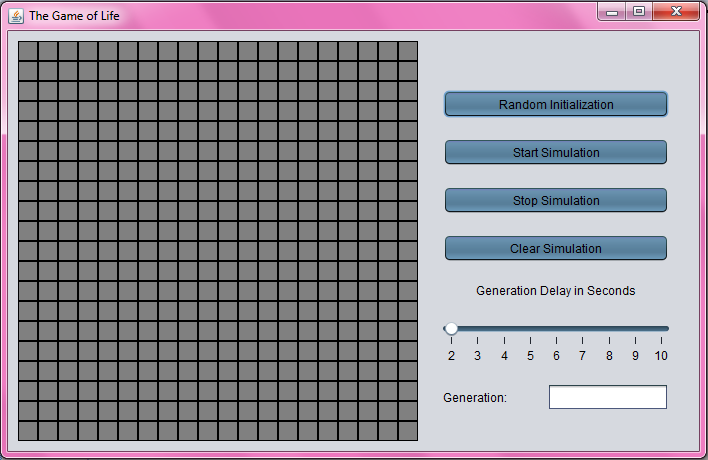


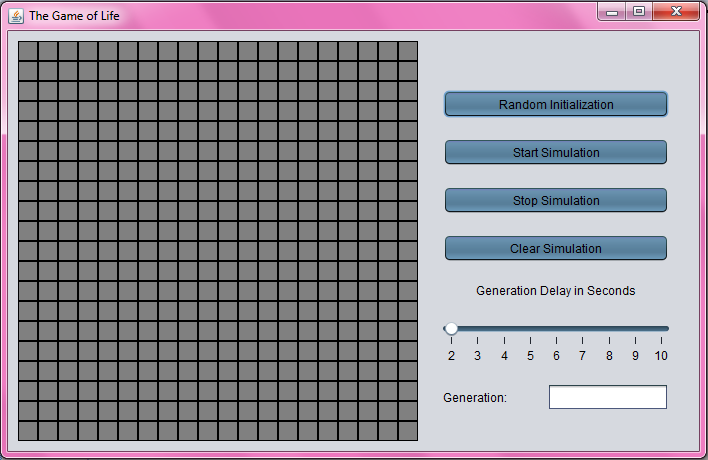
* Java class that uses ActionListener to stop the simulation with the Stop button
  + Uses a variable listener that uses the start listener
  + Constructor setListener that uses listener from start listener
  + Method that defines what the Stop button does
    - Stops the simulation

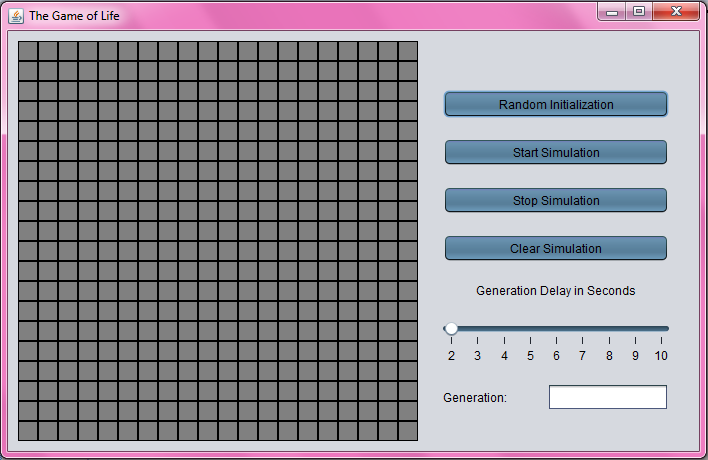
***Entity Relational Diagram***

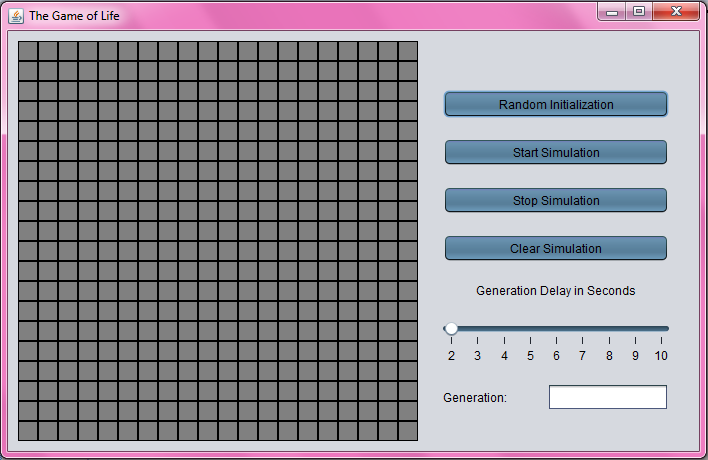


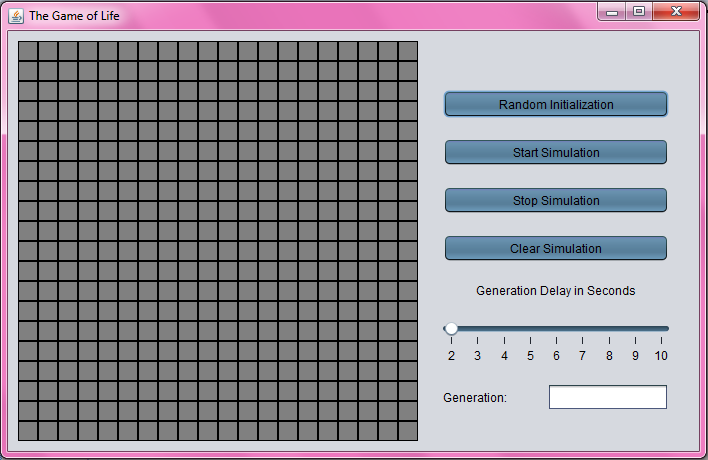
***Component Diagram***

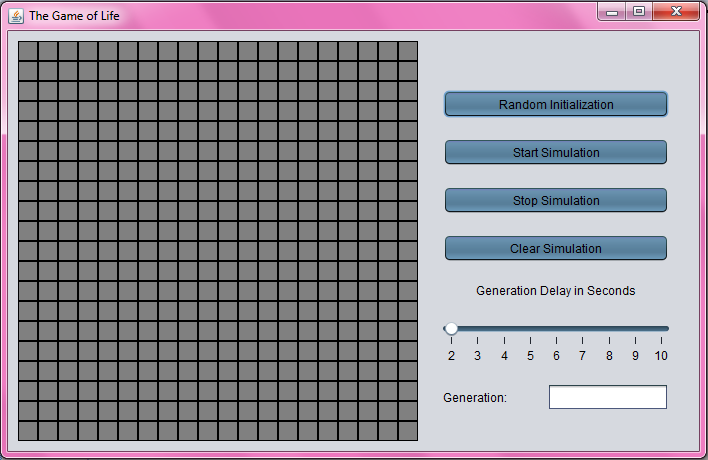
 : Button that starts a random initialization of the game of life, it populates the color grid with a random number of green cells. This component will make cells alive or not alive randomly on the grid.

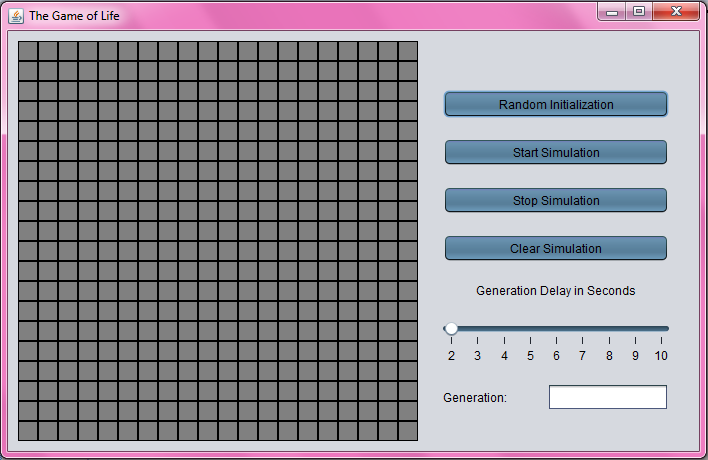
 : Button that starts the simulation for the game of life. The random button can be clicked first then this button to run the simulation. This component will start to make cells alive or not alive depending on neighbors and rules for the game of life.

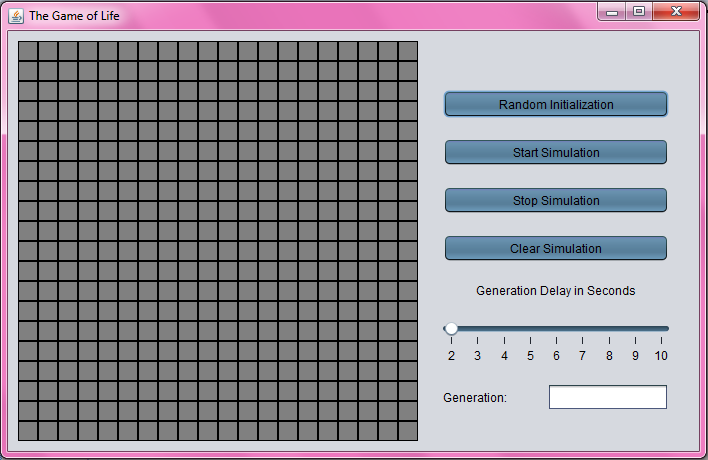
 : Button that stops the simulation for the game of life. It will make the initialization on the grid freeze and keep the cells alive or not alive. It will stop a simulation from running that was started with the start button. This component will freeze the grid simulation and make cells stay in alive or not alive states.

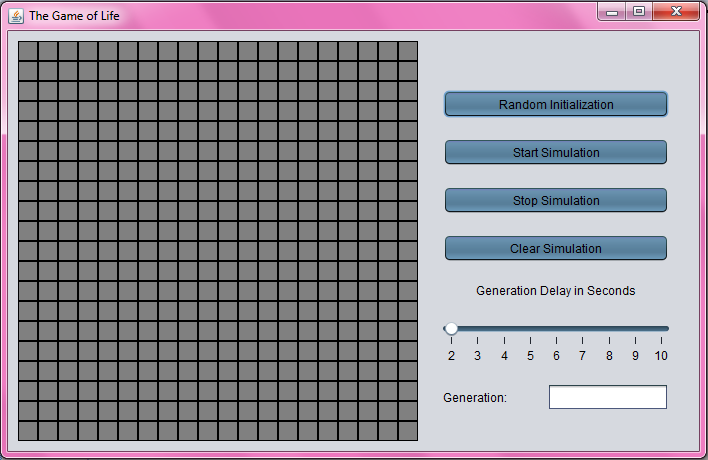
 : Button that clears the simulation that is present on the grid. It will reset the stop, start, and random buttons to their original states and the grid to its original state. It also clears the generation count. This component will clear grid cells and make them not alive again.

 : This part of the panel has the label that says “Generation Delay in Seconds” and the slider that sets the generation delay. The generation delay slider will make the generation change from 2 seconds to 10 seconds. This slider slows the generations on the color grid down to 10 seconds, or speeds it up to 2 seconds.

 : Label and Text Field for the generation count that is associated with how many generations have gone by on the color grid. This will continue to count generations until the stop button is clicked. Once the start button is clicked it will begin counting, when the clear button is clicked it will clear the text field. It is associated with counting the generations that are displayed on the color grid. This component is used to hold the generation count and count generations of color grid cells

: This panel holds all of the buttons, slider, labels, and text fields used by the game of life. It is then added to the frame. This component is used to lay out the buttons and components of the program and have an orderly appearance.

 : This is the grid that is created by ColorGrid and is editable by clicking the mouse on a cell and changing it to green. All of the buttons in the controls panel control what happens in this grid. This component is used to change the cells in the game of life from not alive to alive and vice versa.



This is the main frame of the game of life that holds all of the panels and components within those panels. This is also how the design on the game of life will look on the main screen of the application. This component is used to hold all of the panels and buttons on the program.

# Development History

TBD

# Conclusions

TBD

# References

Callahan, P. (n.d.). *What is the Game of Life?* Retrieved from

<http://www.math.com/students/wonders/life/life.html>

Conway’s Game of Life. (2010). *Conway’s Game of Life.* Retrieved from

<http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/>

International Software Test Institute. (n.d.). *Introduction to Software Testing.* Retrieved from

<http://www.test-institute.org/Introduction_To_Software_Testing.php>

Roberts, Siobhan. (2015, September 5). *A Life in Games: The Playful Genius of John Conway.*

Retrieved from <http://www.wired.com/2015/09/life-games-playful-genius-john-conway/>

Weisstein, E. (n.d.). *Game of Life.* Retrieved from

<http://mathworld.wolfram.com/GameofLife.html>