For our game of life project, we decided to use 4 classes. Our classes were named LifeFrame, LifeView, LifeModel, and LifeController. Life Model deals with the actual rules of the game. Our project also includes a GUI that has graphics, which is dealt with by the LifeView class. First, two 2D arrays are created, and they deal with the cells' current and next lives. The width and height are also created as integers, confirming the arrays' sizes. The constructor initializes the width and the length as x and y respectively, and sets the size of the two 2D arrays with the width and height variables. Then, the initializeGrid() function is called. The initializeGrid() function contains a nested for loop in which x is the row and y is the column and fills the 2D array with random values and controls if the cells are born alive or dead. It sets the cells to a 20% chance of being alive, meaning that if the Math. If the random () number that is generated is less than 0.2, then the cell is alive.

The next function is the update life () function, which deals with the actual rules of the game and whether the cell is alive or dead in the next life. This is within a nested for loop that also loops through each cell in the 2D array nextLife. First, an integer variable is created that represents the number of neighbors that are alive, and it takes into account each cell in the form of x and y. If the cell is currently alive, by checking if the life array is equal to 1(because 1 means alive and 0 means dead), then if the number of alive neighbors is less than 2 or greater than 3, the cell dies in its next life, so the next life array is equal to 0. If this is not the case, then the cell stays alive, so the next life array is equal to 1. This follows the rules of the game in that if the cell has 2 or 3 neighbors, then it stays alive. If the cell is dead, then if the amount of neighbors is exactly 3, it becomes alive, otherwise, it stays dead. After these rules are applied, the updated states of the cells are copied from the next life 2D array to the life 2D array, and this is repeated.

The next function is countAliveNeighbors, it checks the number of neighbors that are alive for each cell and also takes into account if the cell has neighbors on the other side of the board. The count variable deals with the actual number of neighbors that are alive per cell. The loop variables deal with the neighbors of the cells, and if they both equal 0, then the cell is skipped. The dx and dy statements deal with whether the neighbors are on the opposite side of the board. It checks if the neighboring cell is alive, and if it is, it’s added to the count variable. The state is also checked through the life array, and if it is, then it is added to the count variable, and the count variable is then returned. It also accounts for the top left and right, as well as the bottom left and right parts of the board. The copyNextLifeToLife() function copies the information from the next life array to the life array, to signify the next state that the cell will be in. This

function is called at the very end of the update function and is integral to confirming the next state of each cell. The getLife() function returns the life 2D array. Finally, the last function is the matchesPattern function. This goes through the grid and checks to see if certain parts of it match an aforementioned pattern. First, a 2D array called the current grid is created, and it has the life array. Then, there are two variables for the rows and columns, which have a number for each. The number of matching patterns is put into the count variable. The matches are counted when, in the first two loops, parts of the grid are checked against each other. Then, the patterns are compared to the matching element in the specific part of the grid, and if there is a match, then the count variable is incremented. The count is then returned.

The LifeView class deals with the display, including how the board and cells look. In this class, we used graphics components to create the board. The constructor uses an object of the Life model class called model, and the cell size with a variable called cell Size. The constructor initialized both of these variables and also sets the background color of the JPanel to pink using the setBackground method and RGB values. The paint component method is also from the JPanel class and called draw grid and drawers to create the grid and cells respectively. The drawGrid method has a graphic object as a parameter and creates a 2D array called a grid that matches the life 2D array. It also declares the rows and columns. After setting the lines to be black on the board, it uses a for loop to loop through the grid and draw both vertical and horizontal lines. The drawLine function takes two sets of points with an x and y for each. The first drawLine deals with the vertical lines and starts from 0 and goes to the end of the grid. The second drawLine does the same with horizontal lines. The function drawCells deals with creating each cell for the board, with an object from Graphics. It also created a 2D array called grid that takes its information from the life 2D array. Each square between the horizontal and vertical lines is considered a cell, so the grid is looped through, and if a cell is alive (this is known through if the life array is equal to 1), then that square is colored back to be considered alive used the setColor and fillRect methods from Graphics. The first two parameters in the fillRect method are where the cell is, and the second two indicate the width and height of the cell.

The LifeController class connects both the model and the view classes. It creates model and view objects, as well as a timer and integer variables for the number of generations. In the constructor, all of the objects and variables are initialized, and the timer is created and started. The actionPerformed function first prints out the current generation number, then checks to see if the number of generations is less than one,

which would mean that the program would have to end as there would be no more generations. If the number of generations is less than 0, then the timer will be stopped. Otherwise, the update lie method is called for the model, which means that each cell's life status will be checked again based on its neighbors, and the view will also be refreshed, and there will be one less number of generations. The thread method is also used as thread.sleep and the program is stopped for 300 milliseconds. After this, the patterns are compared using the matchesPattern method. The glider, ten-cell, spaceship, and tumbler patterns are all used. Separately, each one is compared to the model using the method, and if there is a match, then it is incremented as a respective increment variable and printed.

Finally, the LifeFrame class deals with creating the graphics that the user will use in the program. The title is set, and the user can set the speed and number of generations through pop-up boxes. The speed and number of generations are taken from these and used. The model is set to a size of 700x700, the view sets the cell size to 20, and a new controller object is also made. Then, the size of the frame is set to 800x800 and the view is added to this, the program ends when it is closed, and setLocationRelativeTo(null) centers the program. It is also set to visible, which means that it will be seen. The main method makes sure that the graphical elements are set up and run first, and then the program is run. Our program follows all of the rules from the rubric and runs perfectly.