

LAB REPORT NO. 8

The Game Of Life

PROF. DR. WEBER-WULFF / INFORMATIK 1 GROUP 1 / ROOM C579 / 12:15 - 13:45

Introduction

This weeks laboratory we learned how to handle two-dimensional arrays using the cellular automats Brian's Brain devised by Brian Silverman and Game Of Life devised by John Horton Conway.

Index

Introduction	1
Assignments	1
1 Modify the Cell class of game-of-life so it implements the rules for the Game of Life.	2
2 Modify Environment class so that the environment is not toroidal and the neighbors outside the bounds are always dead.	4
3 Try the initial patterns that oscillate and move.	8
Evaluation	15
Sophia:	15
Pavel:	15
Appendix	16
Pre-Lab Exercises	16
Code	17
Cell	17
Environment	19
EnvironmentView	23
References	29

Assignments

1 Modify the Cell class of game-of-life so it implements the rules for the Game of Life.

First we saved a copy of the downloaded project *Brain* as *game-of-life* and examined the behavior of the cells in the Environment Viewer and took a look at the three classes.

From the first task we could deduce that the following rules prevailed for the *Game Of Life*:

1. A cell has two states: alive or dead. Its next state depends on:
2. The number of alive neighbors which is counted.
3. If the cell is dead and has exactly three live neighbors its next state will be alive, otherwise it stays dead.
4. If the cell is alive and has fewer than two or more than three live neighbors it stays alive, otherwise dead.



Since there were three states *ALIVE*, *DYING* and *DEAD* in Brian's Brain, we deleted *DYING* and changed *NUM_STATES* to two so that there were only two states left.

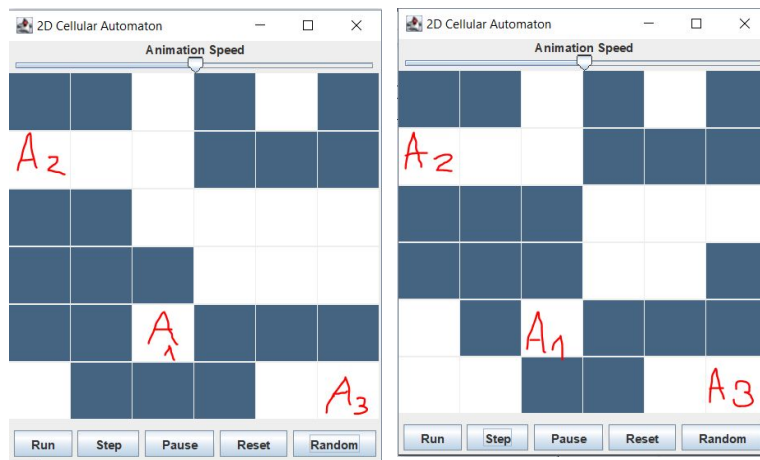
```
// The possible states.  
public static final int ALIVE = 0, DEAD = 1;  
// The number of possible states.  
public static final int NUM_STATES = 2;
```

We quickly found the method in Cell class where we should make the alterations: `getNextState()`

First we deleted all the if statements which include *DYING*. Since the switch from *DEAD* to *ALIVE* also depends on the number of live neighbors we moved the `aliveCount` to the top of the method. We changed the condition for the dead cells so it switches from *DEAD* to *ALIVE* if there are three live neighbors instead of the previous two.

```
public int getNextState()  
{  
    int aliveCount = 0;  
    // Count the number of neighbors that are alive.  
    if(state == DEAD) {  
        for(Cell n : neighbors) {  
            if(n.getState() == ALIVE) {  
                aliveCount++;  
            }  
        }  
        return aliveCount == 3 ? ALIVE : DEAD;  
    }  
    else { //(state==ALIVE)  
        for(Cell n : neighbors) {  
            if(n.getState() == ALIVE) {  
                aliveCount++;  
            }  
        }  
        else if(aliveCount<2||aliveCount>3) {  
            return ALIVE;  
        }  
    }  
    return DEAD;  
}
```

Then we had to think about the changes for the cells that are currently alive. Because we also have to count the live neighbors for the next state we added an if statement.



Testing:

Current State to Next State

A1: ALIVE (1 live, 7 dead)
should be ALIVE (✓)

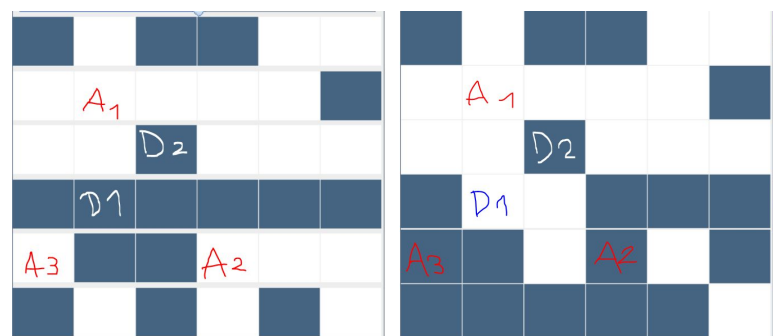
A2: ALIVE (2 live, 6 dead)
should be DEAD (✗)

A3: ALIVE (3 live, 5 dead)
should be DEAD (✗)

But we discovered that it doesn't work since the number of aliveCount is always less than 2 (0). So live cells always stay alive.

Since the if statement for the dead cells seemed to be correct we copied it for the alive cells in else. In this way the live neighbors are getting counted first and then checked with the ternary operator in the return statement how many are in aliveCount.

```
public int getNextState()
{
    int aliveCount = 0;
    // Count the number of neighbors that are alive.
    if(state == DEAD) {
        for(Cell n : neighbors) {
            if(n.getState() == ALIVE) {
                aliveCount++;
            }
        }
        return aliveCount == 3 ? ALIVE : DEAD;
    }
    else {
        for(Cell n : neighbors) {
            if(n.getState() == ALIVE) {
                aliveCount++;
            }
        }
        return aliveCount < 2 || aliveCount > 3 ? ALIVE : DEAD;
    }
}
```



A1: ALIVE (5 live, 3 dead)
should be ALIVE (✓)

D1: DEAD (3 live, 5 dead)
should be ALIVE (✓)

A2: ALIVE (2 live, 6 dead)
should be DEAD (✓)

D2: DEAD (5 live, 3 dead)
should be DEAD (✓)

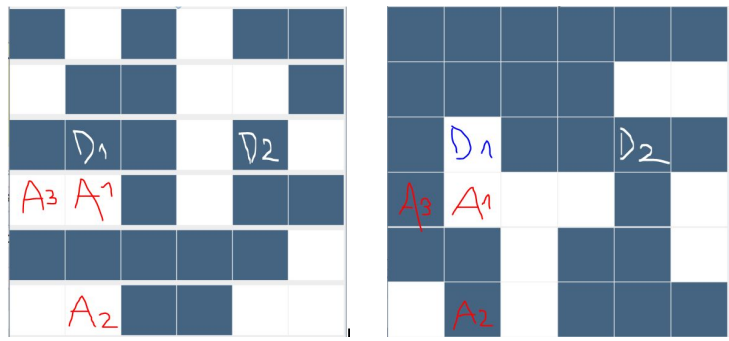
A3: ALIVE (3 live, 5 dead)
should be DEAD (✓)

```
public int getNextState()
{
    int aliveCount = 0;
    // Count the number of neighbors that are alive.
    for(Cell n : neighbors) {
        if(n.getState() == ALIVE) {
            aliveCount++;
        }
    }
    if(state == DEAD) {
        return aliveCount == 3 ? ALIVE : DEAD;
    }
    else {
        return aliveCount < 2 || aliveCount > 3 ? ALIVE : DEAD;
    }
}
```

In the end, we shortened the method a bit, leaving only one for-loop. The result shouldn't be different.

Commented out are some further ideas for the return statement, which are all the same.

Again we tested the alteration with an environment 6x6.



A1: ALIVE (1 live, 7 dead)
should be ALIVE (✓)

A2: ALIVE (2 live, 6 dead)
should be DEAD (✓)

A3: ALIVE (3 live, 5 dead)
should be DEAD (✓)

D1: DEAD (3 live, 5 dead)
should be ALIVE (✓)

D2: DEAD (5 live, 3 dead)
should be DEAD (✓)

estimated time used: 120 minutes

2 Modify Environment class so that the environment is not toroidal and the neighbors outside the bounds are always dead.

Is there any difference between this behavior and the toroidal one?

We look through the *Environment* class and decided that we have to change the method *setupNeighbors*. Because we remembered from lecture that the toroidal form arises through “something with modulo” we took a look into the nested for-loop.

```
private void setupNeighbors()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    // Allow for 8 neighbors plus the cell.
    ArrayList<Cell> neighbors = new ArrayList<>(9);
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            Cell cell = cells[row][col];
            // This process will also include the cell.
            for(int dr = -1; dr <= 1; dr++) {
                for(int dc = -1; dc <= 1; dc++) {
                    int nr = (numRows + row + dr) % numRows;
                    int nc = (numCols + col + dc) % numCols;
                    neighbors.add(cells[nr][nc]);
                }
            }
            // The neighbours should not include the cell at
            // (row,col) so remove it.
            neighbors.remove(cell);
            cell.setNeighbors(neighbors);
            neighbors.clear();
        }
    }
}
```

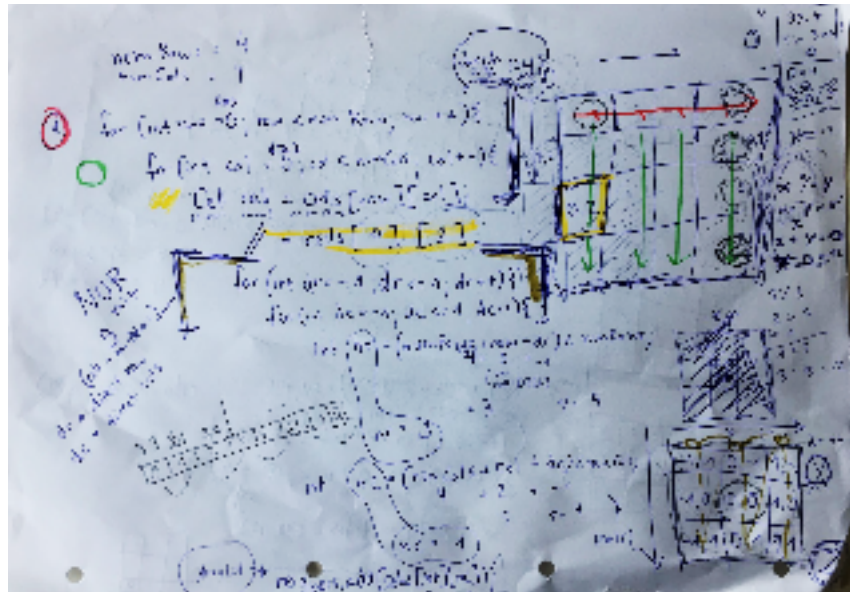
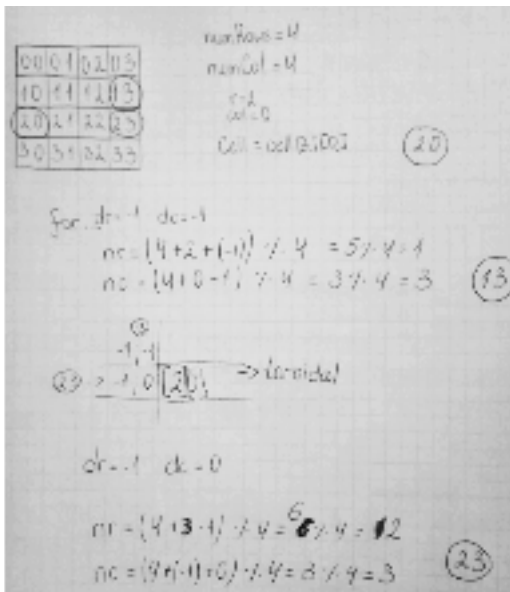
In lab we tried to remove the two modulo operations, but the Environment wouldn't even run and was giving out an error message.

at [Environment.setup\(Environment.java:129\)](#)
at [Environment.<init>\(Environment.java:36\)](#)
at [Environment.<init>\(Environment.java:26\)](#)

After that we tried removing the last two nested for-loops. It compiled but when we start running it the entire board was filled with dead cells.

We decided to go through the method step by step to understand what happens when. In this way we could find the solution easier instead of haphazardly try something that doesn't make sense.

For this reason we both made some sketches on paper and finally understood how the method works and why the 2D array board is toroidal.



So we tried to see how exactly the cells behave so that the board is toroidal. As soon as we drew it on paper and explained our thought process, it was a lot clearer what exactly happens and we gained a better understanding about the representation of cells with a 2D array.

We could clearly see here that when we add the number of rows(numRows) and the numCols to the whole sum and then use the modulo operator we were able to get the cells that are out of bounds.

The last Pre-Lab exercise helped a lot here, because we already had an idea about finding the neighbors of a cell.

We removed the numRows and numCols from the sums and the modulo operations in both lines. Then we added an if statement and as a condition first we check if nr is bigger than zero and less than numRows to ensure we don't go over the border on the left and right. Then we did the same for nc for the top and bottom borders.

```
ArrayList<Cell> neighbors = new ArrayList<>(9);
for(int row = 0; row < numRows; row++) {
    for(int col = 0; col < numCols; col++) {
        Cell cell = cells[row][col];
        // This process will also include the cell.
        for(int dr = -1; dr <= 1; dr++) {
            for(int dc = -1; dc <= 1; dc++) {
                //int nr = (numRows + row + dr) % numRows;
                //int nc = (numCols + col + dc) % numCols;
                int nr = (row + dr);
                int nc = (col + dc);
                /*nr muss größer gleich 0
                 * muss kleiner NumRows
                 *nc muss größer gleich 0
                 * MUSS KLEINER NUMcols
                 */
                if(nr >= 0 && nr < numRows && nc >= 0 && nc < numCols){
                    neighbors.add(cells[nr][nc]);
                }
            }
        }
        // The neighbours should not include the cell at
        // (row,col) so remove it.
    }
}
```


In order to set the cells outside on DEAD, we added an else statement

```
ArrayList<Cell> neighbors = new ArrayList<>(9);
for(int row = 0; row < numRows; row++) {
    for(int col = 0; col < numCols; col++) {
        Cell cell = cells[row][col];
        // This process will also include the cell.
        for(int dr = -1; dr <= 1; dr++) {
            for(int dc = -1; dc <= 1; dc++) {
                //int nr = (numRows + row + dr) % numRows;
                //int nc = (numCols + col + dc) % numCols;
                int nr = (row + dr);
                int nc = (col + dc);
                /*nr muss größer gleich 0
                * muss kleiner NumRows
                *nc muss größer gleich 0
                * MUSS KLEINER NUMcols
                */
                if(nr>=0 && nr<numRows && nc>=0 && nc<numCols){
                    neighbors.add(cells[nr][nc]);
                } else {
                    //add DEAD neighbor (siehe constructorcell)
                    neighbors.add(new Cell());
                }
            }
        }
        // The neighbours should not include the cell at
        // (row,col) so remove it.
        neighbors.remove(cell);
        cell.setNeighbors(neighbors);
    }
}
```

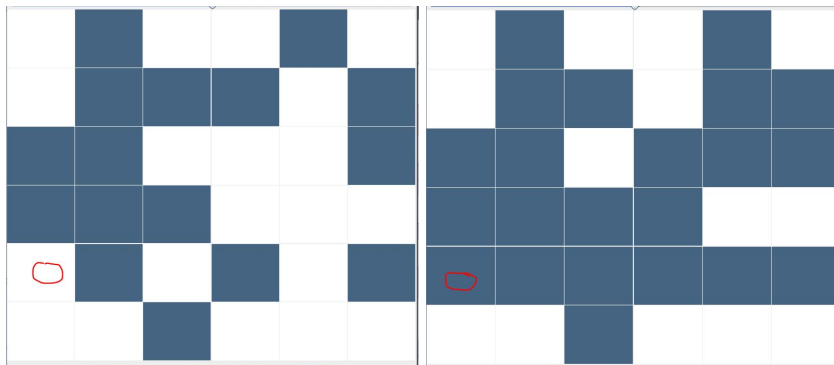
The final, cleaned up method:

```
private void setupNeighbors()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    // Allow for 8 neighbors plus the cell.
    ArrayList<Cell> neighbors = new ArrayList<>(9);
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            Cell cell = cells[row][col];
            // This process will also include the cell.
            for(int dr = -1; dr <= 1; dr++) {
                for(int dc = -1; dc <= 1; dc++) {
                    int nr = (row + dr);
                    int nc = (col + dc);
                    if(nr>=0 && nr<numRows && nc>=0 && nc<numCols){
                        //add neighbors INSIDE the bounds
                        neighbors.add(cells[nr][nc]);
                    } else {
                        //add neighbors OUTSIDE the bounds as DEAD
                        neighbors.add(new Cell());
                    }
                }
            }
            // The neighbours should not include the cell at
            // (row,col) so remove it.
            neighbors.remove(cell);
            cell.setNeighbors(neighbors);
            neighbors.clear();
        }
    }
}
```

Again we tried out the 6x6 environment.

On the toroidal board the cell (marked with a red circle) should still be alive since it has four live neighbors.

If our new code works the cell should be DEAD in the next state, which really happened here.



Difference in behaviour between toroidal and not-toroidal environment:

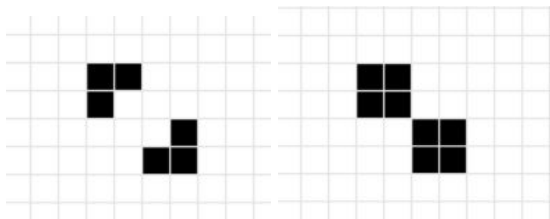
If the board was toroidal in this example, the marked cell on the left picture would stay alive, because the board would take in consideration the cells out of bound as well. After we took a step it changed its state to dead, which further proves that the board here is not toroidal. On a not-toroidal environment the cells that are out of bound are all assumed to be in a dead state.

estimated time used: 160 minutes

3 Try the initial patterns that oscillate and move.

We looked up different oscillate patterns on *wikipedia* and decided to try out the BEACON.

In hope for further informations we also visited the *conwaylife* page where we could find different kinds of oscillators including the beacon.



First we tried to create a new method in cell class but had a better idea for which reason we stopped working on this method.

```
public void beacon()
{
    int aliveCount = 0;
    for(Cell n: neighbors) {
        if(n.getState() == ALIVE) {
            aliveCount++;
        }
    }
    if(state == DEAD) {
        return aliveCount == 5 ? ALIVE : DEAD;
    } else {
        return aliveCount == 5 ? DEAD : ALIVE;
    }
}
```

We changed to the *environment* class and thought about what we need in order to create the patterns.

Since there was already a reset function, we tried to create something similar, an alive, blank board and only set the cells which we need for the patterns on *DEAD*.

```
/**
 * Reset the state of the automaton to all ALIVE.
 */
public void alive()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.ALIVE);
        }
    }
}
```

In order to set up a new button *Alive* on the 2D Cellular Automaton we searched through the *EnvironmentView* class and found the method *setupControls* and just added a similar line of code. We received an error message and searched in the same class for the position where the other symbols, variables were.

Luckily it was in the same method. We realized that we saw that before in lab. It was a lambda function!

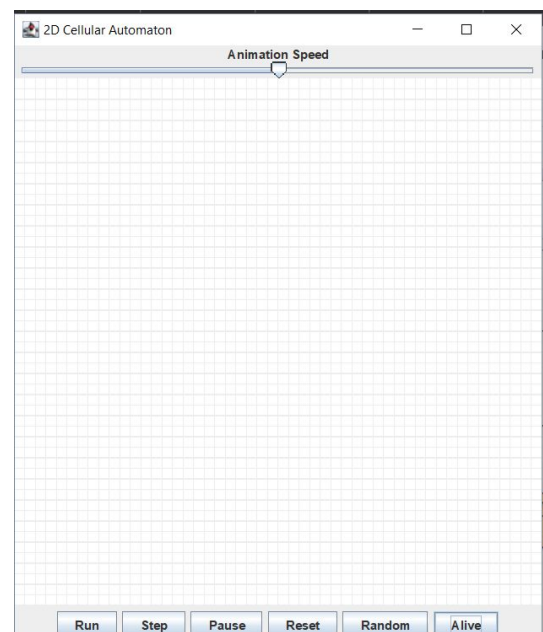
```
// Place the button controls.
JPanel controls = new JPanel();
controls.add(run);
controls.add(step);
controls.add(pause);
controls.add(reset);
controls.add(randomize);
controls.add(alive);

//Sets the entire environment to alive
final JButton alive = new JButton("Alive");
alive.addActionListener(e -> {
    running = false;
    env.alive();
    showCells();
});
```

cannot find symbol - variable alive

Now we created a similar function like the *random* one.

We could compile it and when we opened a new *Environment* the *Alive* Button was added to the bar and also worked when clicking on the button.



Now the next step was to add the dead cells to create a pattern which looks like a beacon and hopefully behave like one.

```
int midRows = numRows/2;  
int midCols = numCols/2;
```

For this we thought about calculating the middle of the cells and add the remaining dead cells around it.

This was simply done by creating two new variable of type int which contains the mid cell as value.

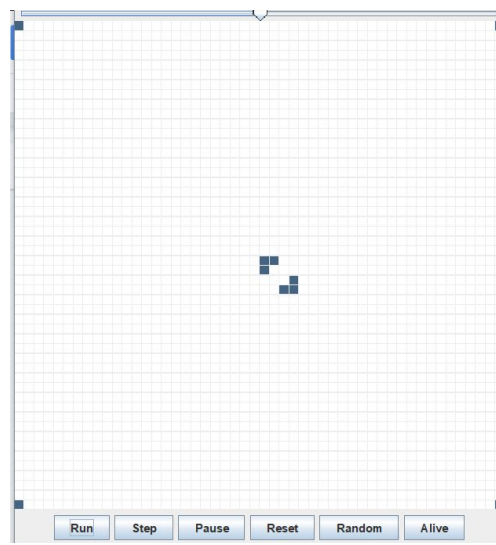
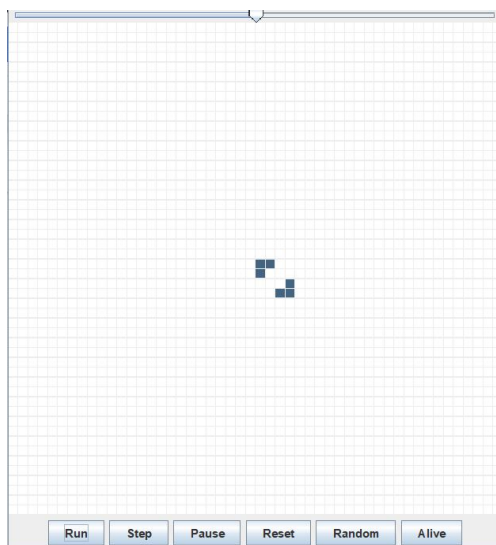
Now we calculated from this cell in the direction in which the next five dead cells should be.

Because we first wanted to try out our idea we were a bit lazy and stayed in the alive method so we don't have to add a new button.

After we were sure that we added all the needed dead cells for beacon we opened the environment and clicked on *Alive*.

The beacon was there but when we *Run* the automaton only the cells in the corner blinked. The beacon stood still and did nothing.

```
/**  
 * Reset the state of the automaton to all ALIVE.  
 */  
public void alive()  
{  
    int numRows = cells.length;  
    int numCols = cells[0].length;  
    for(int row = 0; row < numRows; row++) {  
        for(int col = 0; col < numCols; col++) {  
            setCellState(row, col, Cell.ALIVE);  
        }  
    }  
    int midRows = numRows/2;  
    int midCols = numCols/2;  
    setCellState(midRows, midCols, Cell.DEAD);  
    setCellState(midRows-1, midCols, Cell.DEAD);  
    setCellState(midRows-1, midCols+1, Cell.DEAD);  
  
    setCellState(midRows+2, midCols+2, Cell.DEAD);  
    setCellState(midRows+2, midCols+3, Cell.DEAD);  
    setCellState(midRows+1, midCols+3, Cell.DEAD);  
}
```



Of course because in the corners there are always three live neighbors but looking at the beacon we couldn't find three live but three dead neighbors.

So we tried to do the other way around, reset everything to *DEAD* and only leave the needed cells for the pattern *ALIVE*.

But that didn't work so we tried to change the conditions so *ALIVE* cells need 4 live neighbors to change their state to *DEAD*.

```
/**
 * Reset the state of the automaton to all ALIVE.
 */
public void alive()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.DEAD);
        }
    }

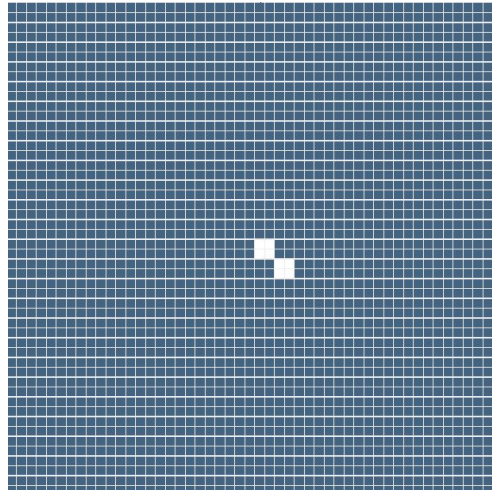
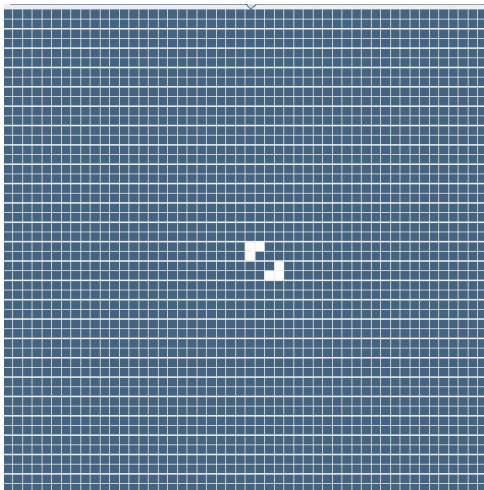
    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows, midCols, Cell.ALIVE);
    setCellState(midRows-1, midCols, Cell.ALIVE);
    setCellState(midRows+1, midCols, Cell.ALIVE);

    setCellState(midRows, midCols-1, Cell.ALIVE);
    setCellState(midRows, midCols+1, Cell.ALIVE);
    setCellState(midRows-1, midCols+1, Cell.ALIVE);
    setCellState(midRows+1, midCols+1, Cell.ALIVE);
}

public int getNextState()
{
    int aliveCount = 0;
    // Count the number of neighbors that are alive.
    for(Cell n : neighbors) {
        if(n.getState() == ALIVE) {
            aliveCount++;
        }
    }

    if(state == DEAD) {
        return aliveCount == 3 ? ALIVE : DEAD;
    } else { //state == ALIVE
        return aliveCount != 4 ? ALIVE : DEAD;
        //return aliveCount <2||aliveCount>3 ? ALIVE : DEAD;
        //return aliveCount == 2|| aliveCount ==3 ? DEAD: ALIVE;
    }
}
```

That did work on the board but we had to change it back to the original conditions in *getNextState*.



Out of curiosity we also tried to set the entire board to ALIVE again and changed the conditions like shown in the screenshot.

```
public int getNextState()
{
    int aliveCount = 0;
    // Count the number of neighbors that are alive.
    for(Cell n : neighbors) {
        if(n.getState() == ALIVE) {
            aliveCount++;
        }
    }

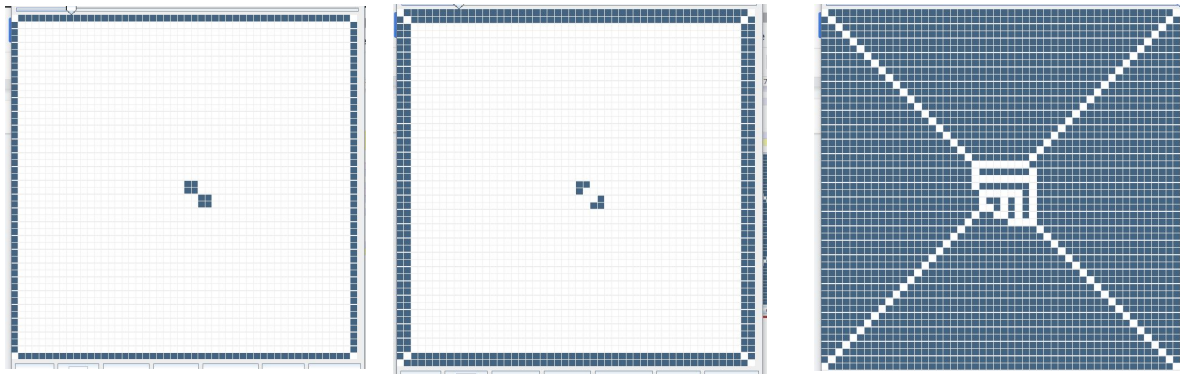
    if(state == DEAD) {
        return aliveCount == 4 ? ALIVE : DEAD;
    } else { //state == ALIVE
        return aliveCount == 5 ? DEAD : ALIVE;
        //return aliveCount <2||aliveCount>3 ? ALIVE : DEAD;
        //return aliveCount == 4 ? ALIVE : DEAD;
    }
}
```

ALIVE → 5 → DEAD

DEAD → 4 → ALIVE

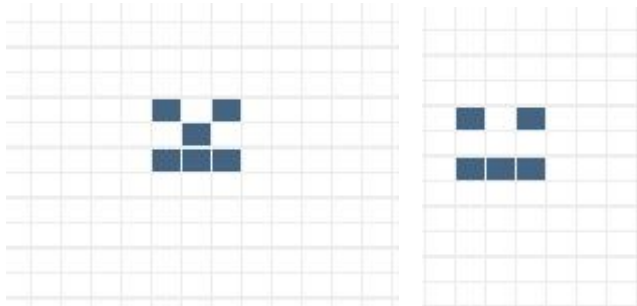
The beacon blinked in the middle but we could see that because of the dead cells outside bounds the environment became

darker and darker and finally had a cool pattern in the end.



We tried searching for oscillators by clicking the *Random* button and filtered out some pattern which looked like a face.

Of course still the corner blinked but also the pattern in the middle.

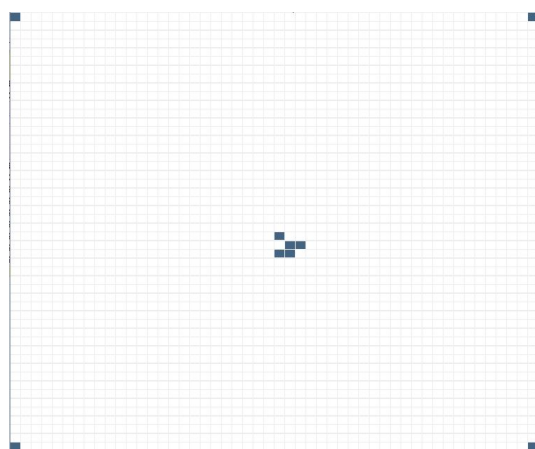


After we created the oscillator we searched for moving patterns.

We tried out a pattern which we saw in Fan's code for both versions (DEAD and ALIVE ship)

```
/**
 * Creates a oscillate pattern
 */
public void oscillator()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.ALIVE);
        }
    }

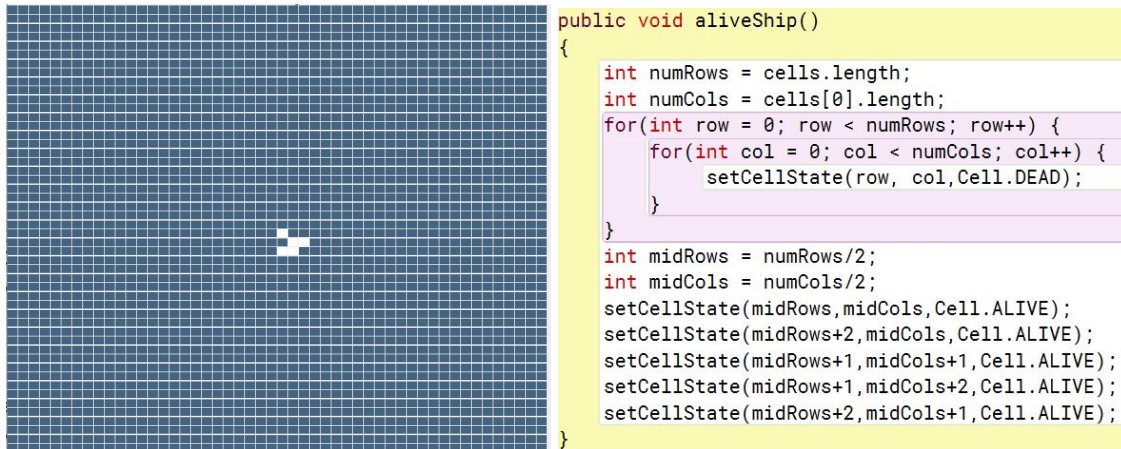
    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows, midCols, Cell.DEAD);
    setCellState(midRows+1, midCols-1, Cell.DEAD);
    setCellState(midRows-1, midCols-1, Cell.DEAD);
    setCellState(midRows+1, midCols, Cell.DEAD);
    setCellState(midRows-1, midCols+1, Cell.DEAD);
    setCellState(midRows+1, midCols+1, Cell.DEAD);
    setCellState(midRows-1, midCols+1, Cell.DEAD);
}
```



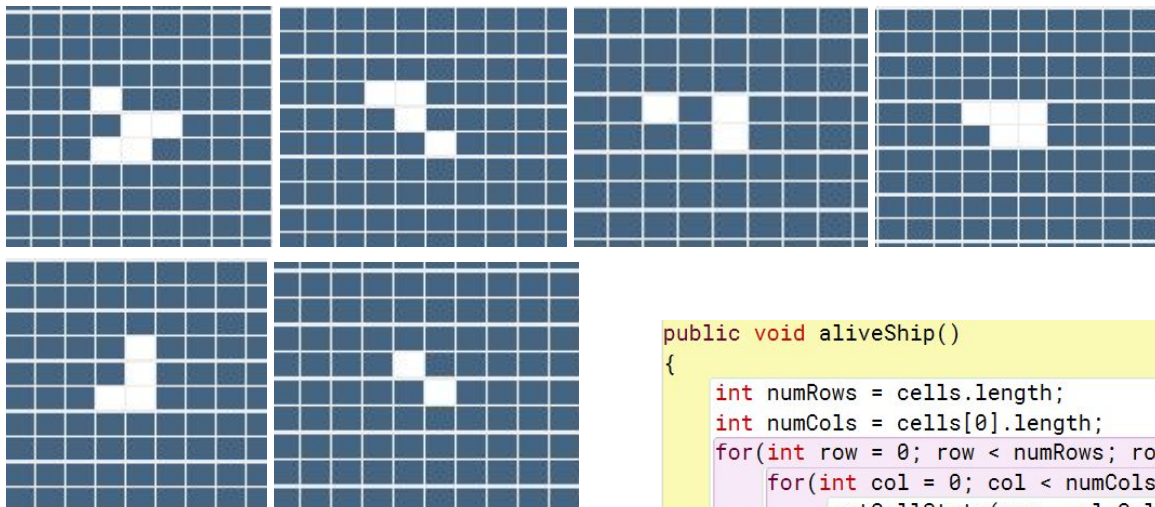
```
public void deathShip()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.ALIVE);
        }
    }

    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows, midCols, Cell.DEAD);
    setCellState(midRows+2, midCols, Cell.DEAD);
    setCellState(midRows+1, midCols+1, Cell.DEAD);
    setCellState(midRows+1, midCols+2, Cell.DEAD);
    setCellState(midRows+2, midCols+1, Cell.DEAD);
}
```

As though the ship made out of *DEAD* cells wouldn't move.



We also created a ship made of ALIVE cells which changed his form five time and then remains in the last form.



It doesn't exactly move like we saw on wikipedia.
So the patterns we found in Wikipedia didn't work unless we changed the conditions or changed the "background" to DEAD.

```
public void aliveShip()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.DEAD);
        }
    }
    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows, midCols, Cell.ALIVE);
    setCellState(midRows+2, midCols, Cell.ALIVE);
    setCellState(midRows+1, midCols+1, Cell.ALIVE);
    setCellState(midRows+1, midCols+2, Cell.ALIVE);
    setCellState(midRows+2, midCols+1, Cell.ALIVE);
}
```

estimated time used: **230** minutes

Evaluation

Sophia:

Since I did not look at the code before lab, I could only guess where I would do the changes. Going through the methods step by step and looking at how the cells behave out the bounds showed me that it is very important to record the changes before removing toroidal and after. The third task was a bit tricky so I looked at Fan's Code for inspiration. Still with the given conditions for aliveCount we couldn't figure out what was wrong. It was funny jumping around the classes see and understand the connections between them and where to change methods in order to create a field for the oscillate and moving form, even if they didn't work!

Pavel:

The exercises in this lab were more interesting, because we were working on a game and we had to make it simpler. I learned more about the use of 2D arrays and how to think about implementing them in code. In this exercise I had to read code and understand it, so that I can apply changes to it. I also learned how I can create my own methods that fill the board in a certain way that i want, with either dead or alive cells even though the end result wasn't exactly as expected. I also learned how to properly check if a given cell is in inside the bounds of the board.

estimated time used for report: **2** hours

Appendix

Pre-Lab Exercises

P1. Download the Brain project, create an Environment object and use the GUI controls to create a random initial setup for the automaton. Then either singl-step it or run/pause it to obtain a feel for how it behaves. Summarize your findings.

- We can create a random generated Environment or we can input the number of rows and columns
- There are white(alive), black(dead) and yellow(dying) cells
- We have a navigation bar where we can reset, randomize, start the program

- After letting the program run for a while, most of the cells die and in the end there are certain shapes left that keep repeating a certain movement

P3. Given a 2D array board and a cell board $[i][j]$. List all of the cells that are its neighbors.

$i \rightarrow$			
j ↓	$i-1; j-1$	$i; j-1$	$i+1; j-1$
	$i-1; j$	$i; j$	$i+1; j$
	$i-1; j+1$	$i; j+1$	$i+1; j+1$

Code

Cell

```
import java.util.*;
/**
 * A cell in a 2D cellular automaton.
 * The cell has multiple possible states.
 * This is an implementation of the rules for Brian's Brain.
 * @see https://en.wikipedia.org/wiki/Brian%27s\_Brain
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Cell
{
    // The possible states.
    public static final int ALIVE = 0, DEAD = 1;
    // The number of possible states.
    public static final int NUM_STATES = 2;

    // The cell's state.
    private int state;
    // The cell's neighbors.
    private Cell[] neighbors;

    /**
     * Set the initial state to be DEAD.
     */
    public Cell()
    {
        this(DEAD);
    }

    /**
     * Set the initial state.
     * @param initialState The initial state
     */
    public Cell(int initialState)
    {
        state = initialState;
        neighbors = new Cell[0];
    }

    /**
     * Determine this cell's next state, based on the
     * state of its neighbors.
     * This is an implementation of the rules for Brian's Brain.
     * @return The next state.
     */
    public int getNextState()
    {
        int aliveCount = 0;
        // Count the number of neighbors that are alive.
    }
}
```

```
        for(Cell n : neighbors) {
            if(n.getState() == ALIVE) {
                aliveCount++;
            }
        }
        if(state == DEAD) {
            return aliveCount == 3 ? ALIVE : DEAD;
        } else { //state ==ALIVE
            return aliveCount == 2 || aliveCount== 3 ? DEAD : ALIVE;
        }
    }

    /**
     * Receive the list of neighboring cells and take
     * a copy.
     * @param neighborList Neighboring cells.
     */
    public void setNeighbors(ArrayList<Cell> neighborList)
    {
        neighbors = new Cell[neighborList.size()];
        neighborList.toArray(neighbors);
    }

    /**
     * Get the state of this cell.
     * @return The state.
     */
    public int getState()
    {
        return state;
    }

    /**
     * Set the state of this cell.
     * @param The state.
     */
    public void setState(int state)
    {
        this.state = state;
    }
}
```

Environment

```
import java.security.SecureRandom;
import java.util.*;
/**
 * Maintain the environment for a 2D cellular automaton.
 *
 * @author David J. Barnes
 * @version 2016.02.29
 */
public class Environment
{
    // Default size for the environment.
    private static final int DEFAULT_ROWS = 50;
    private static final int DEFAULT_COLS = 50;

    // The grid of cells.
    private Cell[][] cells;
    // Visualization of the environment.
    private final EnvironmentView view;

    /**
     * Create an environment with the default size.
     */
    public Environment()
    {
        this(DEFAULT_ROWS, DEFAULT_COLS);
    }

    /**
     * Create an environment with the given size.
     * @param numRows The number of rows.
     * @param numCols The number of cols;
     */
    public Environment(int numRows, int numCols)
    {
        setup(numRows, numCols);
        randomize();
        view = new EnvironmentView(this, numRows, numCols);
        view.showCells();
    }

    /**
     * Run the automaton for one step.
```

```
*/
public void step()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    // Build a record of the next state of each cell.
    int[][] nextStates = new int[numRows][numCols];
    // Ask each cell to determine its next state.
    for(int row = 0; row < numRows; row++) {
        int[] rowOfStates = nextStates[row];
        for(int col = 0; col < numCols; col++) {
            rowOfStates[col] = cells[row][col].getNextState();
        }
    }
    // Update the cells' states.
    for(int row = 0; row < numRows; row++) {
        int[] rowOfStates = nextStates[row];
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, rowOfStates[col]);
        }
    }
}

/**
 * Reset the state of the automaton to all DEAD.
 */
public void reset()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.DEAD);
        }
    }
}

/**
 * Creates a oscillate pattern
 */
public void oscillator()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.ALIVE);
        }
    }
    int midRows = numRows/2;
```



```
int midCols = numCols/2;
setCellState(midRows ,midCols, Cell.DEAD);
setCellState(midRows+1,midCols-1, Cell.DEAD);
setCellState(midRows-1,midCols-1, Cell.DEAD);
setCellState(midRows+1,midCols, Cell.DEAD);
setCellState(midRows-1,midCols+1, Cell.DEAD);
setCellState(midRows+1,midCols+1, Cell.DEAD);
setCellState(midRows-1,midCols+1, Cell.DEAD);
}

/**
 * Creates an dead ship surrounded by live cells.
 */
public void deathShip()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.ALIVE);
        }
    }
    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows,midCols, Cell.DEAD);
    setCellState(midRows+2,midCols, Cell.DEAD);
    setCellState(midRows+1,midCols+1, Cell.DEAD);
    setCellState(midRows+1,midCols+2, Cell.DEAD);
    setCellState(midRows+2,midCols+1, Cell.DEAD);
}

/**
 * Creates an ship full of live in a sea of dead cells.
 */
public void aliveShip()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, Cell.DEAD);
        }
    }
    int midRows = numRows/2;
    int midCols = numCols/2;
    setCellState(midRows,midCols, Cell.ALIVE);
    setCellState(midRows+2,midCols, Cell.ALIVE);
    setCellState(midRows+1,midCols+1, Cell.ALIVE);
    setCellState(midRows+1,midCols+2, Cell.ALIVE);
    setCellState(midRows+2,midCols+1, Cell.ALIVE);
}
```

```
}

/**
 * Generate a random setup.
 */
public void randomize()
{
    int numRows = cells.length;
    int numCols = cells[0].length;
    SecureRandom rand = new SecureRandom();
    for(int row = 0; row < numRows; row++) {
        for(int col = 0; col < numCols; col++) {
            setCellState(row, col, rand.nextInt(Cell.NUM_STATES));
        }
    }
}

/**
 * Set the state of one cell.
 * @param row The cell's row.
 * @param col The cell's col.
 * @param state The cell's state.
 */
public void setCellState(int row, int col, int state)
{
    cells[row][col].setState(state);
}

/**
 * Return the grid of cells.
 * @return The grid of cells.
 */
public Cell[][] getCells()
{
    return cells;
}

/**
 * Setup a new environment of the given size.
 * @param numRows The number of rows.
 * @param numCols The number of cols;
 */
private void setup(int numRows, int numCols)
{
    cells = new Cell[numRows][numCols];
    for(int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
            cells[row][col] = new Cell();
        }
    }
}
```

```
        setupNeighbors();
    }

    /**
     * Give to a cell a list of its neighbors.
     */
    private void setupNeighbors()
    {
        int numRows = cells.length;
        int numCols = cells[0].length;
        // Allow for 8 neighbors plus the cell.
        ArrayList<Cell> neighbors = new ArrayList<>(9);
        for(int row = 0; row < numRows; row++) {
            for(int col = 0; col < numCols; col++) {
                Cell cell = cells[row][col];
                // This process will also include the cell.
                for(int dr = -1; dr <= 1; dr++) {
                    for(int dc = -1; dc <= 1; dc++) {
                        int nr = (row + dr);
                        int nc = (col + dc);
                        if(nr>=0 && nr<numRows && nc>=0 && nc<numCols){
                            //add neighbors INSIDE the bounds
                            neighbors.add(cells[nr][nc]);
                        } else {
                            //add neighbors OUTSIDE the bounds as DEAD
                            neighbors.add(new Cell());
                        }
                    }
                }
            }
        }
        // The neighbours should not include the cell at
        // (row,col) so remove it.
        neighbors.remove(cell);
        cell.setNeighbors(neighbors);
        neighbors.clear();
    }
}
```

EnvironmentView

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
/**
 * A GUI for the environment, with runtime controls.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class EnvironmentView extends JFrame
{
    // The longest delay for the animation, in milliseconds.
    private static final int LONGEST_DELAY = 1000;
    // Colors for the different cell states.
    private static final Color[] colors = {
        Color.WHITE, // Alive
        new Color(68, 100, 129), // Dead
        new Color(204, 196, 72), // Dying
    };

    private GridView view;
    private final Environment env;
    private boolean running;
    private int delay;

    /**
     * Constructor for objects of class EnvironmentView
     * @param env
     */
    public EnvironmentView(Environment env, int rows, int cols)
    {
        super("2D Cellular Automaton");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(20, 20);
        this.env = env;
        this.running = false;
        setDelay(50);
        setupControls();
        setupGrid(rows, cols);
        pack();
        setVisible(true);
    }

    /**
     * Setup a new environment of the given size.
     * @param rows The number of rows.
     * @param cols The number of cols;
     */
    private void setupGrid(int rows, int cols)
```

```
{
    Container contents = getContentPane();
    view = new GridView(rows, cols);
    contents.add(view, BorderLayout.CENTER);
}

/**
 * Show the states of the cells.
 */
public void showCells()
{
    Cell[][] cells = env.getCells();
    if(!isVisible()) {
        setVisible(true);
    }

    view.preparePaint();
    for(int row = 0; row < cells.length; row++) {
        Cell[] cellRow = cells[row];
        int numCols = cellRow.length;
        for(int col = 0; col < numCols; col++) {
            int state = cellRow[col].getState();
            view.drawMark(col, row, colors[state]);
        }
    }
    view.repaint();
}

/**
 * Set up the animation controls.
 */
private void setupControls()
{
    // Continuous running.
    final JButton run = new JButton("Run");
    run.addActionListener(e -> {
        if(!running) {
            running = true;
            try {
                new Runner().execute();
            }
            catch(Exception ex) {
            }
        }
    });

    // Single stepping.
    final JButton step = new JButton("Step");
    step.addActionListener(e -> {
        running = false;
    });
}
```



```
        env.step();
        showCells();
    });

    // Pause the animation.
    final JButton pause = new JButton("Pause");
    pause.addActionListener(e -> running = false);

    // Reset of the environment
    final JButton reset = new JButton("Reset");
    reset.addActionListener(e -> {
        running = false;
        env.reset();
        showCells();
    });

    //Create an oscillate pattern
    final JButton oscillator = new JButton("Oscillator");
    oscillator.addActionListener(e -> {
        running = false;
        env.oscillator();
        showCells();
    });

    //Create an move pattern.
    final JButton deathShip = new JButton("Deathship");
    deathShip.addActionListener(e -> {
        running = false;
        env.deathShip();
        showCells();
    });

    //Create an move pattern.
    final JButton aliveShip = new JButton("Aliveship");
    aliveShip.addActionListener(e -> {
        running = false;
        env.aliveShip();
        showCells();
    });

    // Randomize the environment.
    final JButton randomize = new JButton("Random");
    randomize.addActionListener(e -> {
        running = false;
        env.randomize();
        showCells();
    });

    Container contents = getContentPane();
```

```
// A speed controller.
final JSlider speedSlider = new JSlider(0, 100);
speedSlider.addChangeListener(e -> {
    setDelay(speedSlider.getValue());
});
Container speedPanel = new JPanel();
speedPanel.setLayout(new GridLayout(2, 1));
speedPanel.add(new JLabel("Animation Speed", SwingConstants.CENTER));
speedPanel.add(speedSlider);
contents.add(speedPanel, BorderLayout.NORTH);

// Place the button controls.
JPanel controls = new JPanel();
controls.add(run);
controls.add(step);
controls.add(pause);
controls.add(reset);
controls.add(randomize);
controls.add(aliveShip);
controls.add(deathShip);
controls.add(oscillator);

contents.add(controls, BorderLayout.SOUTH);
}

/**
 * Set the animation delay.
 * @param speedPercentage (100-speedPercentage) as a percentage of the
 * LONGEST_DELAY.
 */
private void setDelay(int speedPercentage)
{
    delay = (int) ((100.0 - speedPercentage) * LONGEST_DELAY / 100);
}

/**
 * Provide stepping of the animation.
 */
private class Runner extends SwingWorker<Boolean, Void>
{
    @Override
    /**
     * Repeatedly single-step the environment as long
     * as the animation is running.
     */
    public Boolean doInBackground()
    {
        while(running) {
            env.step();
        }
    }
}
```

```
        showCells();
        try {
            Thread.sleep(delay);
        }
        catch(InterruptedException e) {
        }
    }
    return true;
}
}

/**
 * Provide a graphical view of a rectangular grid.
 */
@SuppressWarnings("serial")
private class GridView extends JPanel
{
    private final int GRID_VIEW_SCALING_FACTOR = 10;

    private final int gridWidth, gridHeight;
    private int xScale, yScale;
    private Dimension size;
    private Graphics g;
    private Image fieldImage;

    /**
     * Create a new GridView component.
     */
    public GridView(int height, int width)
    {
        gridHeight = height;
        gridWidth = width;
        size = new Dimension(0, 0);
    }

    /**
     * Tell the GUI manager how big we would like to be.
     */
    @Override
    public Dimension getPreferredSize()
    {
        return new Dimension(gridWidth * GRID_VIEW_SCALING_FACTOR,
                               gridHeight * GRID_VIEW_SCALING_FACTOR);
    }

    /**
     * Prepare for a new round of painting. Since the component
     * may be resized, compute the scaling factor again.
     */
}
```

```

public void preparePaint()
{
    if(! size.equals(getSize())) {
        size = getSize();
        fieldImage = view.createImage(size.width, size.height);
        g = fieldImage.getGraphics();

        xScale = size.width / gridWidth;
        if(xScale < 1) {
            xScale = GRID_VIEW_SCALING_FACTOR;
        }
        yScale = size.height / gridHeight;
        if(yScale < 1) {
            yScale = GRID_VIEW_SCALING_FACTOR;
        }
    }
}

/**
 * Paint on grid location on this field in a given color.
 */
public void drawMark(int x, int y, Color color)
{
    g.setColor(color);
    g.fillRect(x * xScale, y * yScale, xScale-1, yScale-1);
}

/**
 * The field view component needs to be redisplayed. Copy the
 * internal image to screen.
 */
@Override
public void paintComponent(Graphics g)
{
    if(fieldImage != null) {
        Dimension currentSize = getSize();
        if(size.equals(currentSize)) {
            g.drawImage(fieldImage, 0, 0, null);
        }
        else {
            // Rescale the previous image.
            g.drawImage(fieldImage, 0, 0, currentSize.width, currentSize.height,
null);
        }
    }
}
}
}
}
}

```

References

Conwaylife(20.03.2019) - Oscillator <http://www.conwaylife.com/wiki/Oscillator> (retrieved on 15.06.2019)

Wikipedia(12.06.2009) - Beacon.gif
https://en.wikipedia.org/wiki/File:Game_of_life_beacon.gif (retrieved on 15.06.2019)

Conwaylife(26.01.2019) - Beacon <http://www.conwaylife.com/wiki/Beacon> (retrieved on 15.06.2019)