

Genetische Algorithmen  
am Beispiel des  
„Game of Life“  
visualisiert mit einer  
Java-GUI

Informatik LK 11

Fachlehrer: Herr Fassbender

Jan-Lukas Spilles

## Inhaltsverzeichnis:

<b>Einleitung:</b> .....	<b>3</b>
<b>Genetische Algorithmen:</b> .....	<b>4</b>
Definition: .....	4
Anwendung:.....	4
<b>Das Game of Life:</b> .....	<b>4</b>
Herkunft:.....	4
Begriffserklärung: .....	4
Funktionsweise: .....	5
<b>Implementation:</b> .....	<b>6</b>
<b>Darstellung der genetischen Algorithmen am Game of Life:</b> .....	<b>10</b>
<b>Probleme:</b> .....	<b>12</b>
<b>Literaturverzeichnis:</b> .....	<b>13</b>
Quellenverzeichnis: .....	13
Abbildungsverzeichnis: .....	13
Hilfsmittel: .....	13
<b>Anhang:</b> .....	<b>14</b>
Klassendokumentation:.....	14
Quelltext: .....	15
Die Klasse Zelle: .....	15
Die Klasse Frame: .....	17
Die Klasse Simulation: .....	19
Die Klasse GameOfLife: .....	27
<b>Eigenständigkeitserklärung:</b> .....	<b>28</b>

## **Einleitung:**

Man nimmt 200 Kaninchen und steckt sie in einen 30m<sup>2</sup> großen Stall. Nach einem Tag sind vier Kaninchen aufgrund der Überbevölkerung gestorben, nach drei weiteren Tagen sind weitere zehn Kaninchen verstorben, aber auch vier neue Tiere geboren.

Wenn man nun zum Beispiel die erste und die zweite Generation miteinander vergleicht fällt einem auf, dass sie nicht identisch sind, zum Beispiel sind in der zweiten Generation weniger Kaninchen als in der ersten, das bedeutet, dass einige Kaninchen aus Generation eins nicht fit genug waren, um Nachfolger zu zeugen.

In der Biologie wird dieses Geschehnis die „natürliche Selektion“ genannt und auch in der Informatik findet man diese Vorgänge in Form von genetischen Algorithmen wieder, diese sind Algorithmen, welche auf der Basis von Genetik und zum Beispiel der natürlichen Selektion Zustände zurückgeben.

In dieser Facharbeit werde ich die Funktionsweise von diesen Algorithmen anschaulich erklären und erläutern. Dazu werde ich die Computersimulation Game of Life mit einer GUI (Graphical-User-Interface) in Java implementieren.

Grund für die Wahl ist das Interesse an der theoretischen Informatik, in welcher Genetische Algorithmen zum Beispiel in Form des Game of Life auftreten.

Ziel dieser Facharbeit wird es sein, zuerst ein funktionierendes Game of Life mit GUI zu schreiben und anschließend damit Genetische Algorithmen zu erklären.

## **Genetische Algorithmen:**

### **Definition:**

Genetische Algorithmen oder auch evolutionäre Algorithmen sind Algorithmen, die in ihrer Funktionsweise von der Evolution natürlicher Lebewesen inspiriert sind.

Es wird eine künstliche Population erzeugt, welche sich in jedem Schritt verändert. Bei der Veränderung der Population werden dann im Algorithmus festgelegte „Regeln“, wie zum Beispiel Genetik – wie sah die Population vorher aus und Bereiche der natürlichen Selektion angewendet.

### **Anwendung:**

Genetische Algorithmen werden in vielen Gebieten zur Problemlösung bzw. als Optimierungsverfahren eingesetzt. Sie werden in der Regel bei größeren oder komplexeren Datenmengen verwendet, bei denen herkömmliche Algorithmen zu viel Zeit beanspruchen. Zum Beispiel beim Erstellen von Stundenplänen an großen Schulen oder zur Platzierung von Containern auf einem Schiff, um beispielsweise eine größtmögliche Stabilität zu gewährleisten. Ein anderer Aspekt ist die Biologie, man kann beobachten, wie eine Population in Generationen aussieht.

## **Das Game of Life:**

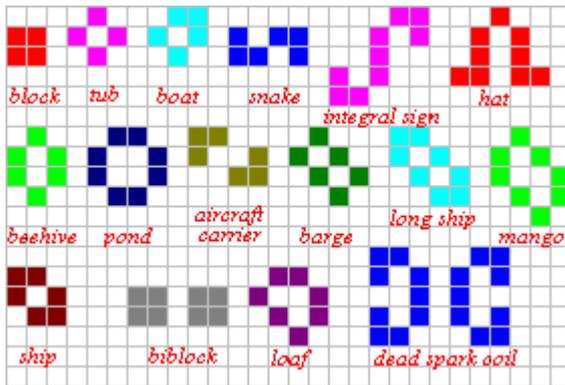
### **Herkunft:**

Das Game of Life ist eine Computersimulation, welche im Jahre 1970 von dem englischen Mathematikprofessor John Conway entwickelt wurde. Mit ihr kann man das Verhalten von Populationen über mehrere Intervalle simulieren und beobachten. Es gibt allerdings nicht nur Anwendungsmöglichkeiten in der theoretischen Informatik, es kann auch aus Sicht der Biologie, Chemie, Physik und Automatentheorie verwendet werden.

### **Begriffserklärung:**

#### **Stabile Population:**

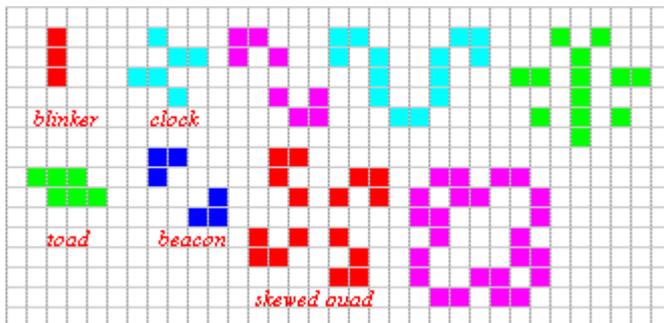
Stabile Populationen oder auch Stilleben bestehen nur aus Zellen mit zwei oder drei Nachbarn, sie bleiben somit unverändert, solange kein Oszillator auf sie trifft.



1. Bild: Verschiedene Stilleben

### Oszillator:

Oszillatoren sind periodische Muster von Zellen, welche immer zum Ursprungsmuster zurückkehren. Die Periode bezeichnet in diesem Fall, wie lange es dauert, bis das Ursprungsmuster wieder erreicht ist. Solange keine anderen Zellen auf einen Oszillator treffen, oszillieren sie immer weiter. Des Weiteren gibt es noch „spezielle“ Oszillatoren, wie zum Beispiel die „Gleiter“ und „Raumschiffe“, diese oszillieren nicht wie gewöhnlich um einen Punkt, sondern bewegen sich über das Feld.



2. Bild: Verschiedene Oszillatoren

### Erstarrung:

Man spricht von einer Erstarrung der Simulation, wenn das Feld nur noch aus Oszillatoren und stabilen Populationen besteht, sich das Feld also optisch nichtmehr verändert.

### Funktionsweise:

Beim Game of Life gibt es ein unbegrenztes, kariertes Feld, auf welchem einige dieser Zellen „tot“ und einige „lebendig“ sind. Je nach Zustand der umliegenden Zellen können Zellen „sterben“, sich nicht verändern oder „geboren“ werden. Was genau geschieht, wird durch Regeln festgelegt. Die Regeln lauten: Wenn um eine Zelle mehr als vier Zellen leben stirbt diese an Überbevölkerung, sind um eine Zelle nur eine oder null Zellen stirbt diese an Einsamkeit. Bei zwei oder drei Nachbarzellen bleibt die Zelle

unverändert, wenn um eine tote Zelle genau drei Zellen sind wird dort eine neue geboren. Diese Regeln werden in jeder Runde beziehungsweise bei jeder Generation auf jede Zelle auf diesem Feld angewandt. Ein richtiges Ende gibt es dabei nicht, es kann lediglich passieren, dass sich das Feld nicht mehr verändert, dies ist der Fall, wenn nur noch stabile Populationen oder Oszillatoren auf dem Feld sind, man sagt auch, das Feld ist „erstarrt“.

## Implementation:

Für die Simulation des Game of Life wurden zunächst die vier Klassen Frame, Simulation, GameOfLife und Zelle erstellt.

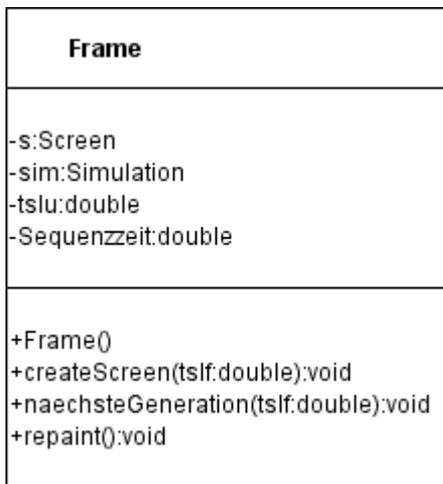
### Die Klasse Zelle:

Zelle
-x: int -y: int -lebens: boolean -nextround: boolean -size: int -grid: boolean
+Zelle(x: int, y:int) +isLebens(): boolean +setLebens(lebens:boolean): void +setNextRound(nextround:boolean): void +nextround(): void +draw(g:Graphics): void

3. Bild: Klassendokumentation Zelle

Als erstes wurde die Klasse Zelle benötigt, welche Objekte erzeugt, die später in das Raster eingefügt werden können. Das Raster ist zweidimensional angelegt, daher muss jede Zelle eine X und eine Y Position besitzen, außerdem muss festgelegt werden, ob die Zelle lebt oder tot ist und wie es in der nächsten Generation aussieht. Dann wird noch ein int für die Größe in Bezug auf die Simulation benötigt und ein boolean, welcher Entscheidet, ob das Raster angezeigt wird. Wird der Konstruktor dieser Klasse aufgerufen, werden nur X und Y Koordinate festgelegt. Abgesehen von der draw()-Methode besteht die Klasse nur aus Getter und Setter Methoden für die booleans lebens und nextround. Die draw()-Methode ist dafür zuständig auf der Simulation später das Raster mit ihren lebenden und toten Zellen zu zeichnen.

## Die Klasse Frame:



4. Bild: Klassendokumentation Frame

Als nächstes wurde eine Klasse für die Darstellung benötigt, die Klasse Frame. Sie ist eine Unterklasse der Klasse JFrame und ist für das Fenster beziehungsweise den Rahmen zuständig, in welchem die Simulation stattfinden soll.

Wenn der Konstruktor aufgerufen wird, werden zuerst zwei Dialogfenster geöffnet, in welchen die gewünschte Sequenzzeit und die Zellengröße abgefragt werden. Danach wird ein leerer JFrame erzeugt.

Die Klasse besteht aus drei wichtigen Methoden, einmal die createScreen()-Methode, welche einmal ein neues Objekt der Klasse Simulation und eins der Klasse Screen erstellt – Die Klasse Screen ist eine innere Klasse der Klasse Frame und eine Unterklasse der Klasse JLabel, sie ist das was auf dem Frame angezeigt wird –, anschließend wird mit der Methode add() der Klasse JFrame, diesem noch das JLabel hinzugefügt.

Die nächste wichtige Methode ist naechsteGeneration(), mit dieser Methode wird der nächste Frameinhalt erzeugt. Wenn diese Methode also aufgerufen wird, geht die Darstellung eine Generation weiter.

Die Dritte ist die repaint()-Methode, sie ist eine Methode der abstrakten Klasse Graphics und muss somit überschrieben werden. Sie aktualisiert die Simulation in der graphischen Benutzeroberfläche.

## Die Klasse Simulation:

<b>Simulation</b>
-zellen: Zelle[] [] -random: Random -width: int -height: int -generation: int -go: boolean -button: int
+Simulation() +naechsteGeneration(): void +draw(g: Grpahics): void +keyPressed(e: KeyEvent): void +keyReleased(e: KeyEvent): void +keyTyped(e: KeyEvent): void +mouseDragged(e: MouseEvent): void +mouseMoved(e: MouseEvent): void +mouseClicked(e: MouseEvent): void +mouseEntered(e: MouseEvent): void +mouseExited(e: MouseEvent): void +mousePressed(e: MouseEvent): void +mouseReleased(e: MouseEvent): void

5. Bild: Klassendokumentation Simulation

Die Klasse Simulation ist der Kern des Programms, in ihr ist der Algorithmus des Ganzen zu finden, außerdem wird in dieser Klasse das gespeichert, was später von den Klassen Frame und Screen in die graphische Benutzeroberfläche umgewandelt wird.

Im Konstruktor wird ein neues Zelle-Array erzeugt, welches dann mit weißen (toten) Zellen gefüllt wird.

Die wichtigste Methode dieser Klasse ist die naechsteGeneration()-Methode, durch sie wird festgelegt, wie die Population in der nächsten Generation aussieht, bzw. welche Individuen überleben oder Nachfahren gezeugt haben. Zuerst wird geschaut, wie viele Nachbarn eine Zelle hat, dafür wird ein Zähler vom Typ int genommen. Anschließend folgen drei if-abfragen. Einmal, ob eine Zelle weniger als zwei oder mehr als drei Nachbarn hat, falls dies der Fall ist, wird die Zelle in der nächsten Runde tot sein. Dann wird geschaut, ob eine Zelle genau zwei Nachbarn hat, wenn dies der Fall ist, wird die Zelle in der nächsten Generation genau wie in der vorherigen sein. Sollten um eine

Zelle genau drei Nachbarn sein, wird die Zelle in der nächsten Generation geboren werden bzw., wenn sie schon lebt, am Leben bleiben. Am Ende der Methode wird das Zelle-Array mithilfe von zwei for-Schleifen mit den neuen Informationen über die Zellen gefüllt.

Die nächste wichtige Methode ist die draw()-Methode. Sie wendet, mithilfe von zwei for-Schleifen, auf jede Zelle im Array die draw()-Methode der Klasse Zelle, welche die einzelnen Zellen zeichnet. Zusätzlich ist sie für den Generationszähler oben links in der Mitte zuständig.

Zur Steuerung der graphischen Benutzeroberfläche wurden Key-, Mouse- und MouseMotionListener verwendet. Das erste KeyEvent geschieht, wenn die Taste „G“ gedrückt wird, sie macht das Raster unsichtbar bzw. sichtbar. Die Taste „Z“ sorgt für eine zufällige Population, dies geschieht mithilfe von zwei for-Schleifen und der Klasse Random, das ganze Array wird durchlaufen und jede Zelle wird per Zufall auf lebend oder tot gesetzt. Außerdem wird der Generationenzähler zurückgesetzt.

Die Taste „R“ dient zum zurücksetzen, sie durchläuft das Array und setzt alle Zellen auf tot. Dann wird noch der Generationenzähler zurückgesetzt. Die Leertaste dient als Start/Stopp Taste, sie setzt den boolean „go“ auf true bzw. false.

Ein weiteres Steuerelement ist der Mouse- bzw. der MouseMotionListener, mit ihnen kann man später auf dem Fenster Zellen belegen. Dafür müssen zuerst alle abstrakten Methoden übernommen werden, da es sich um eine abstrakte Klasse handelt. Die Methode mouseDragged() sorgt dafür, dass man die linke oder rechte Maustaste gedrückt halten kann und damit direkt mehrere Zellen belegen oder löschen kann. Wenn die linke Taste gedrückt wird, wird der int button auf eins gesetzt und alle Zellen, über die man nun mit dem Mauszeiger geht, werden auf lebend gesetzt. Ist der int button nicht eins, also wenn die rechte Maustaste gedrückt wird, dann werden die Zellen auf tot gesetzt. Die Methode mousePressed() funktioniert ähnlich, nur das sie dazu da ist, um einzelne Zellen per Klick zu belegen oder zu löschen. Die letzte Steuermethode ist mouseReleased(), sie ist dafür zuständig, dass, wenn man die Taste loslässt nicht weitergezeichnet oder gelöscht wird, dafür wird der int button auf -1 gesetzt, so dass nichts mehr passiert.

## Die Klasse GameOfLife:

<b>GameOfLife</b>
-width: int -height: int
+main(String[] args): void

6.Bild: Klassendokumentation GameOfLife

Die Klasse GameOfLife ist die Mainklasse des Projekts, sie beinhaltet die Main-Methode und keinen Konstruktor. Wird die Main-Methode aufgerufen wird zunächst ein neues Objekt der Klasse Frame erzeugt, welches sich automatisch dem Bildschirm anpasst. Danach werden Länge und Breite des Frames in den beiden Attributen gespeichert und die Methode createScreen() der Klasse Frame wird aufgerufen.

Nun beginnt eine while(true)-Schleife, also eine Endlosschleife, in der in jedem Durchgang der double tsf erneuert wird und mit diesem die Methode naechsteGeneration() der Klasse Frame aufgerufen wird, zum Aktualisieren der graphischen Benutzeroberfläche wird nun noch die repaint()-Methode aufgerufen.

### **Klassenerklärung:**

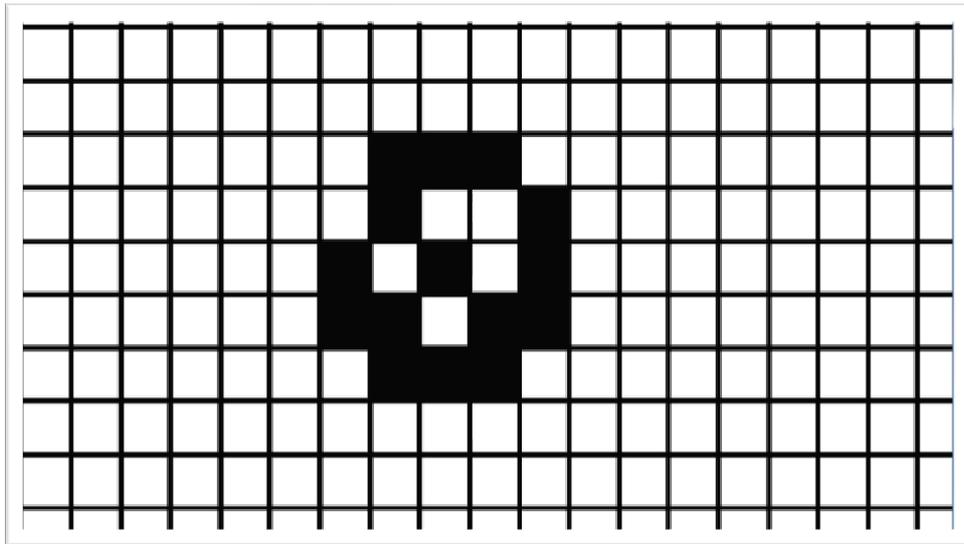
Es wird die Main-Methode der GameOfLife Klasse ausgeführt, die erzeugt sich dann ein Objekt der Klasse Frame, welches wiederum ein Objekt der Klasse Screen und der Klasse Simulation erzeugt. Die Simulation erstellt sich dann ein Zelle-Array, welches auf den Screen, welcher im Frame ist, übertragen wird.

Ein Implementationsdiagramm dazu findet sich im Anhang.

## **Darstellung der genetischen Algorithmen am Game of Life:**

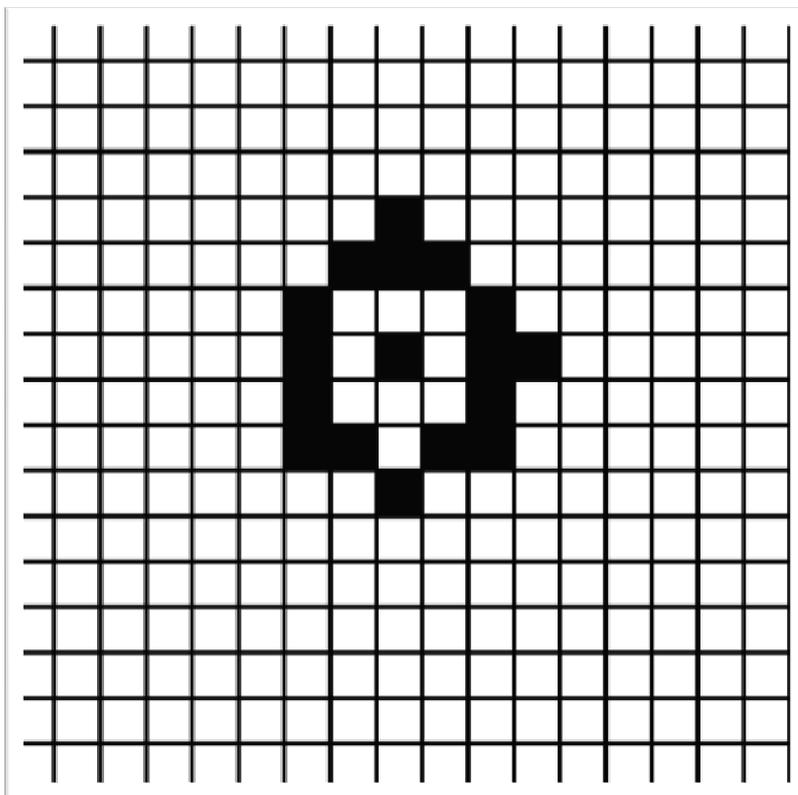
Die Struktur eines genetischen Algorithmus lässt sich am Game of Life sehr gut erklären. Die oben genannten biologischen Aspekte bezüglich Genetik und der natürlichen Selektion finden sich in der Simulation wieder. Es müssen noch die Begriffe Population und Individuum festgelegt werden. Unter einer Population versteht man die Ansammlung von mehreren Zellen, ein Individuum hingegen ist nur eine einzelne Zelle.

Wenn nun eine Population wie folgt aussieht:



*7. Bild: 1. Generation, eigene Darstellung*

Wird sie in der nächsten Generation so aussehen:



*8. Bild: 2. Generation, eigene Darstellung*

Bezüglich der Genetik kann man die Population als ein Ganzes betrachten. Die erste Generation übergibt somit durch ihre Gegebenheiten der zweiten Generation, das äußere Erscheinungsbild. Würde in der ersten Generation eine Zelle mehr lebend oder eine Zelle mehr tot sein, könnte die Population in der zweiten Generation schon ein ganz anderes Erscheinungsbild haben. Um die natürliche Selektion hieran zu erklären, muss man die Zellen als einzelne Individuen betrachten. Es gilt das Überleben der Stärksten, wobei hier die stärkeren Individuen die sind, welche eine bestimmte Anzahl an Nachbarn besitzen. So zum Beispiel Zelle Z, welche in der nächsten Generation noch lebt. Neben dem Überleben der Stärksten ist ein anderer wichtiger Begriff noch die sogenannte Fitness, welche daran gemessen wird, wie viele Nachkommen ein Individuum bekommt. Somit ist Zelle Z nicht nur „stark“, sondern auch fit, da durch sie die Zelle N entstanden ist. Wie die Population in die nächste Generation übergeht, ist hier das für einen genetischen Algorithmus typische, auf Genetik makrokosmisch und auf Selektion mikrokosmisch bezogen.

### **Probleme:**

Trotz der ordentlichen Implementation, hat sich bei dem Ganzen ein Problem eingeschlichen, dies betrifft den Mouse- und den MouseMotion-Listener. Diese funktionieren nur bei Zellengröße 10, ich vermute, dass der Fehler bei der Größe des JLabels ist, konnte ihn aber leider noch nicht finden.

Allerdings ist dieses Problem nur ein „Schönheitsfehler“ und verändert nicht die Funktionalität der Simulation.

## **Literaturverzeichnis:**

## **Quellenverzeichnis:**

- 1) <http://www.mathematische-basteleien.de/gameoflife.htm>
- 2) [http://de.wikipedia.org/wiki/Conways\\_Spiel\\_des\\_Lebens](http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens)

## **Abbildungsverzeichnis:**

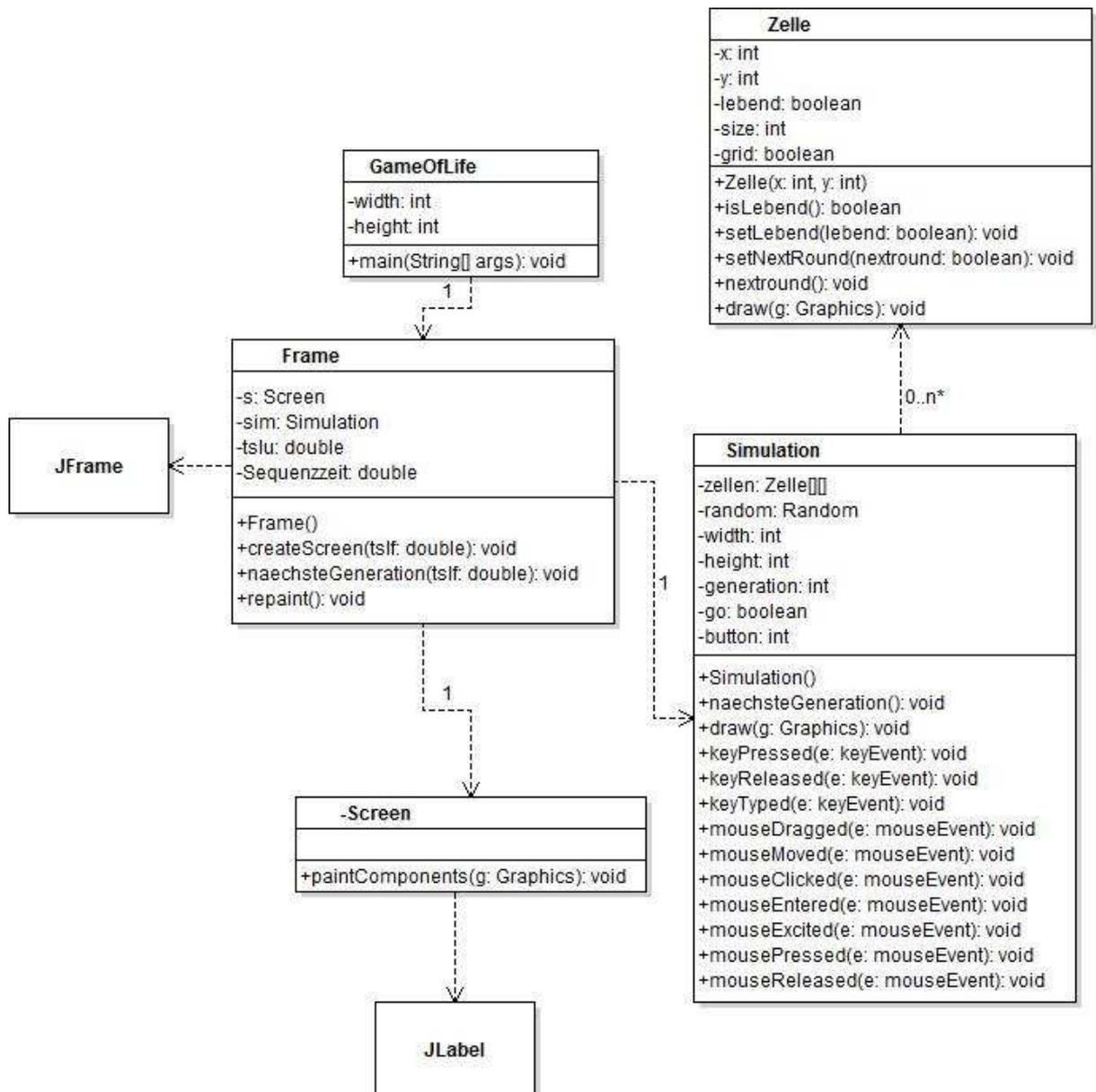
- 1) Beispiele für stabile Populationen, <http://www.mathematische-basteleien.de/game05.gif>
- 2) Beispiele für Oszillatoren, <http://www.mathematische-basteleien.de/game06.gif>
- 3) Klassendokumentation der Klasse Zelle, eigene Darstellung
- 4) Klassendokumentation der Klasse Frame, eigene Darstellung
- 5) Klassendokumentation der Klasse Simulation, eigene Darstellung
- 6) Klassendokumentation der Klasse GameOfLife, eigene Darstellung
- 7) Darstellung einer Population in der ersten Generation, eigene Darstellung
- 8) Darstellung der Population aus Bild 7 in der zweiten Generation, eigene Darstellung

## **Hilfsmittel:**

Zum Erstellen der UML-Klassendiagramme habe ich das Programm „Violet UML Editor“ verwendet, es ist eine open Source und auf der Seite:  
<http://alexdp.free.fr/violetumleditor/page.php> zu finden.

## Anhang:

### Klassendokumentation:



## Quelltext:

### Die Klasse Zelle:

```
import java.awt.Color;
import java.awt.Graphics;
public class Zelle
{
    private int x;
    private int y;
    private boolean lebend;
    private boolean nextround;
    static int size;
    static boolean grid=true;

    public Zelle(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public boolean isLebend()
    {
        return lebend;
    }

    public void setLebend(boolean lebend)
    {
```

```

    this.lebend = lebend;
}

public void setNextRound(boolean nextround)
{
    this.nextround = nextround;
}

public void nextRound()
{
    lebend = nextround;
}

public void draw(Graphics g)
{
    if(grid == true)
    {
        g.setColor(Color.BLACK);
        g.drawRect(x * size, y * size, size, size);
        if(lebend == true)
        {
            g.setColor(Color.BLACK);
        }
        else
        {
            g.setColor(Color.WHITE);
        }
        g.fillRect(x * size + 1, y * size + 1, size - 1, size - 1);
    }
    else

```

```

    {
        if(lebend == true)
        {
            g.setColor(Color.BLACK);
        }
        else
        {
            g.setColor(Color.WHITE);
        }
        g.fillRect(x * size, y * size, size, size);
    }
}
}

```

### **Die Klasse Frame:**

```

import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;

public class Frame extends JFrame
{
    private Screen s;
    private Simulation sim;
    private double tslu;
    private double Sequenzzeit;

    public Frame()
    {
        String input = JOptionPane.showInputDialog(this,"Zellengröße?");
        try

```

```

    {
        Zelle.size = Integer.parseInt(input);
    } catch (Exception e)
    {
        System.exit(0);
    }

    input = JOptionPane.showInputDialog(this, "Sequenzzeit?");
    try
    {
        Sequenzzeit = Double.parseDouble(input);
    } catch (Exception e)
    {
        System.exit(0);
    }
}

```

```

public void createScreen()
{
    sim = new Simulation();

    addKeyListener(sim);
    addMouseListener(sim);
    addMouseMotionListener(sim);

    s = new Screen();
    s.setSize(this.getSize());
    add(s);
}

```

```

}
public void naechsteGeneration(double tslf)
{
    tslu += tslf;
    if(tslu >= Sequenzzeit)
    {
        sim.naechsteGeneration();
        tslu = 0;
    }
}
public void repaint()
{
    s.repaint();
}
private class Screen extends JLabel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        sim.draw(g);
    }
}
}

```

### **Die Klasse Simulation:**

```

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

```

```

import java.awt.event.MouseMotionListener;
import java.util.Random;

public class Simulation implements KeyListener, MouseMotionListener, MouseListener
{
    private Zelle[] [] zellen;
    private Random random;
    private int width = GameOfLife.width/Zelle.size;
    private int height = GameOfLife.height/Zelle.size;
    private int generation;
    private boolean go;
    private int button;
    public Simulation()
    {
        random = new Random();

        zellen = new Zelle[width] [height];
        for(int x = 0;x < width;x++)
        {
            for(int y = 0;y < height;y++)
            {
                zellen[x] [y] = new Zelle(x,y);
            }
        }
    }

    public void naechsteGeneration()
    {
        if(go==true)
        {
            generation++;

```

```

for(int x = 0;x < width;x++)
{
    for(int y = 0;y < height;y++)
    {
        int zaehler = 0;
        if(x-1>=0&&y-1>=0){
            if(zellen[x-1][y-1].isLebend()){
                zaehler++;
            }
        }
        if(x-1>=0){
            if(zellen[x-1][y].isLebend()){
                zaehler++;
            }
        }
        if(x-1>=0&&y+1<=height-1){
            if(zellen[x-1][y+1].isLebend()){
                zaehler++;
            }
        }
        if(y-1>=0){
            if(zellen[x][y-1].isLebend()){
                zaehler++;
            }
        }
        if(y+1<=height-1){
            if(zellen[x][y+1].isLebend()){
                zaehler++;
            }
        }
        if(x+1<=width-1&&y-1>=0){

```

```

        if(zellen[x+1][y-1].isLebend()){
            zaehler++;
        }
    }
    if(x+1<=width-1){
        if(zellen[x+1][y].isLebend()){
            zaehler++;
        }
    }
    if(x+1<=width-1&& y+1<=height-1){
        if(zellen[x+1][y+1].isLebend()){
            zaehler++;
        }
    }
    if(zaehler < 2 | | zaehler > 3)
    {
        zellen[x] [y].setNextRound(false);
    }
    else if(zaehler == 2)
    {
        zellen[x] [y].setNextRound(zellen[x] [y].isLebend());
    }
    else if(zaehler == 3)
    {
        zellen[x] [y].setNextRound(true);
    }
}
}
for(int x = 0;x < width;x++)
{
    for(int y = 0;y < height;y++)

```

```

        {
            zellen[x] [y].nextRound();
        }
    }
}
}

```

```

public void draw(Graphics g)
{
    for(int x = 0;x < width;x++)
    {
        for(int y = 0;y < height;y++)
        {
            zellen[x] [y].draw(g);
        }
    }
    g.setColor(Color.RED);
    g.setFont(new Font("TimesNewRoman", Font.BOLD, 25));
    g.drawString("" + generation, 10, 10 + g.getFont().getSize());
}

```

```

public void keyPressed(KeyEvent e)
{

}

```

```

public void keyReleased(KeyEvent e)
{
    if(e.getKeyCode() == KeyEvent.VK_G)
    {
        if(Zelle.grid == true)

```

```

{
    Zelle.grid = false;
}
else
{
    Zelle.grid = true;
}
}
if(e.getKeyCode() == KeyEvent.VK_Z)
{
    for(int x = 0;x < width;x++)
    {
        for(int y = 0;y < height;y++)
        {
            zellen[x] [y].setLebend(random.nextBoolean());
        }
    }
    generation = 0;
}
if(e.getKeyCode() == KeyEvent.VK_R)
{
    for(int x = 0;x < width;x++)
    {
        for(int y = 0;y < height;y++)
        {
            zellen[x] [y].setLebend(false);
        }
    }
    generation = 0;
}
if(e.getKeyCode() == KeyEvent.VK_SPACE)

```

```

    {
        if(go==true)
        {
            go = false;
        }
        else
        {
            go = true;
        }
    }

}

public void keyTyped(KeyEvent e)
{

}

public void mouseDragged(MouseEvent e)
{
    if(go == false)
    {
        int mx = e.getX()/Zelle.size;
        int my = e.getY()/Zelle.size;
        if(button == 1)
        {
            zellen[mx-1] [my-3].setLebend(true);
        }
        else
        {
            zellen[mx-1] [my-3].setLebend(false);
        }
    }
}

```

```

    }
}

public void mouseMoved(MouseEvent e)
{

}

public void mouseClicked(MouseEvent e)
{

}

public void mouseEntered(MouseEvent e)
{

}

public void mouseExited(MouseEvent e)
{

}

public void mousePressed(MouseEvent e)
{
    button = e.getButton();
    if(go == false)
    {
        int mx = e.getX()/Zelle.size;
        int my = e.getY()/Zelle.size;

```

```

    if(button == 1)
    {
        zellen[mx-1][my-3].setLebend(true);
    }
    else
    {
        zellen[mx-1][my-3].setLebend(false);
    }
}
}

```

```

public void mouseReleased(MouseEvent e)
{
    button = -1;
}
}

```

### **Die Klasse GameOfLife:**

```

import javax.swing.JFrame;

public class GameOfLife
{
    static int width;
    static int height;

    public static void main(String[] args)
    {
        JFrame f = new JFrame();
        f.setSize(java.awt.Toolkit.getDefaultToolkit().getScreenSize());
        f.setLocation(0,0);
        f.setDefaultCloseOperation(f.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}

```

```
width = f.getWidth();
height = f.getHeight();
f.createScreen();

long lastFrame = System.currentTimeMillis();
while(true)
{
    long thisFrame = System.currentTimeMillis();
    double tslf = (double) ((thisFrame - lastFrame) / 1000.0);
    lastFrame = thisFrame;
    f.naechsteGeneration(tslf);
    f.repaint();
}
}
```

### **Eigenständigkeitserklärung:**

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt habe und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.