Aufgabe 5: Bauernopfer

Team: Hier könnte ihre Werbung stehen
Team-ID: 00093

19. November 2017

Inhaltsverzeichnis

1 Lösungsansätze

1.1 Aufgabe 1

Ausgehend von einem normalen 8x8-Schachbrett mit acht Bauern und nur einem Turm, welcher sich nur horizontal und vertikal bewegen kann, positioniert man die Bauern einmal diagonal über das komplette Feld. Dadurch wird das Brett in zwei voneinander abgeschnittene Bereiche getrennt, in denen sich der Turm befindet und allmählich in die Enge getrieben wird. Dies geschieht durch ein systematisches Vorgehen der Bauern, indem der äußerste Bauer nach unten, oben, rechts oder links, abhängig von der Position des Turms und der gezogenen Diagonale, bewegt, dass sich keine Lücke zwischen den separierten Feldern bildet. Daraufhin zieht sich dieser Schritt durch alle Bauern der Reihe nach durch, bis man zum Bauern angelangt ist, der vor sich bereits das Ende des Spielfeldes hat. Dieser Vorgang wiederholt sich anschließend so lange, bis der Turm gefangen wurde.

1.2 Aufgabe 2

Bei der zweiten Teilaufgabe ist es den Bauern nicht möglich den Turm zu fangen, da er immer die Möglichkeit hat sich so zu positionieren, dass er im nächsten Zug durch eine Lücke entkommen könnte. Und um dies zu verhindern muss ein Bauer immer nachziehen. Daraufhin sucht sich der Turm die neue entstandene Lücke aus und positioniert sich wieder so, dass er im nächsten Zug entwischen würde. Dieses Szenario wiederholt sich nun also endlos und der Turm wird nie gefangen.

2 Umsetzung

2.1 Datenstrukturen

Als Programmiersprache zur Implementierung haben wir hier Java gewählt und uns zudem jeweils für eine eigene Klasse zum Zwischenspeichern der Daten für die Position, Figur und dass Brett entschieden, welche in Kapitel 3.1 aufgelistet sind.

2.1.1 Figur

Die Klasse der Figur speichert lediglich, ob es ein Turm oder ein Bauer ist.

2.1.2 Position

Ein Tupel aus X- und Y-Koordinaten wird zum vereinfachten Bearbeiten des Prozesses in einer eigenen Hilfs-Datenstruktur gespeichert. Die X- und Y-Werte entsprechen hierbei den Indizes des zweidimensionalen Arrays. Sie nehmen also Werte zwischen inklusive 0 und 7 an.

2.1.3 Zug

Die einzelnen Züge werden mit Start- und Zielkoordinaten gespeichert, welche jeweils dem Index der Matrix des Feldes entspricht. Zur Veranschaulichung wurde die *toString()* Methode mit einer eigenen Ausgabe überschrieben, wie in Kapitel 3 nachzulesen ist.

2.1.4 Map

Als Feld haben wir einen zweidimensionalen Array genommen, welcher die Matrix des Feldes repräsentiert. In diesem Array werden die Figuren gepeichert, wobei (0 | 0) die linke obere Ecke des Feldes darstellt. Zudem wird dieses Feld zweifach gespeichert. Einmal um den initalen Stand für die Reproduktion zu haben und nicht alle Züge rekursiv zurück gehen zu müssen und zweitens den aktuellen Stand, mit dem der Algorithmus gearbeitet hat. Neben dem Feld selbst speichert es zudem chronologisch den Verlauf jeden Schrittes, welche vom Algorithmus übergeben wurde, in einer eindimensionalen ArrayList. Dies geschieht zur einfachen Verwendung durch die GUI.

2.2 Simuliertes Schachbrett

Um die Anforderung der Visualisierung der Aufgabe zu erfüllen, entschieden wir uns für eine simulierte Ausführung der einzelnen Züge der Figuren, welche anschließend auf einer grafischen Oberflache durch das erneute Durchspielen aller gespeicherten Schritte visualisiert werden. Der Algorithmus tätigt also zuerst hypothetische Züge auf dem Spielfeld und anschließend wird der Verlauf dieser Züge hintereinander auf der GUI erneut durchgespielt.

2.3 Visualisierung

Mithilfe der bereits in Java integegrierten Klassen JFrame und JPanel visualisiert die GUI den Verlauf, welcher in der Map abgespeichert ist. Hierbei wird eine statische Fenstergröße benutzt und das Grid-Layout verwendet. Mithilfe eines Timer, der alle 200ms ein Event auslöst, um den Verlauf schrittweise zu reproduzieren. Dieses Event überprüft den aktuellen Stand der Reproduktion und zeichnet das Interface entsprechend neu. Neben der graphischen Zeichnung des Bretts mit schwarzen und weißen Figuren wurde eine Textausgabe noch mit einprogrammiert. Diese dient als Log für die schrittweise Reproduktion. Zur Darstellung von Endlosschleifen wurde ebenfalls eine Loop-Option einprogrammiert. Die GUI sieht dann wie folgt aus:

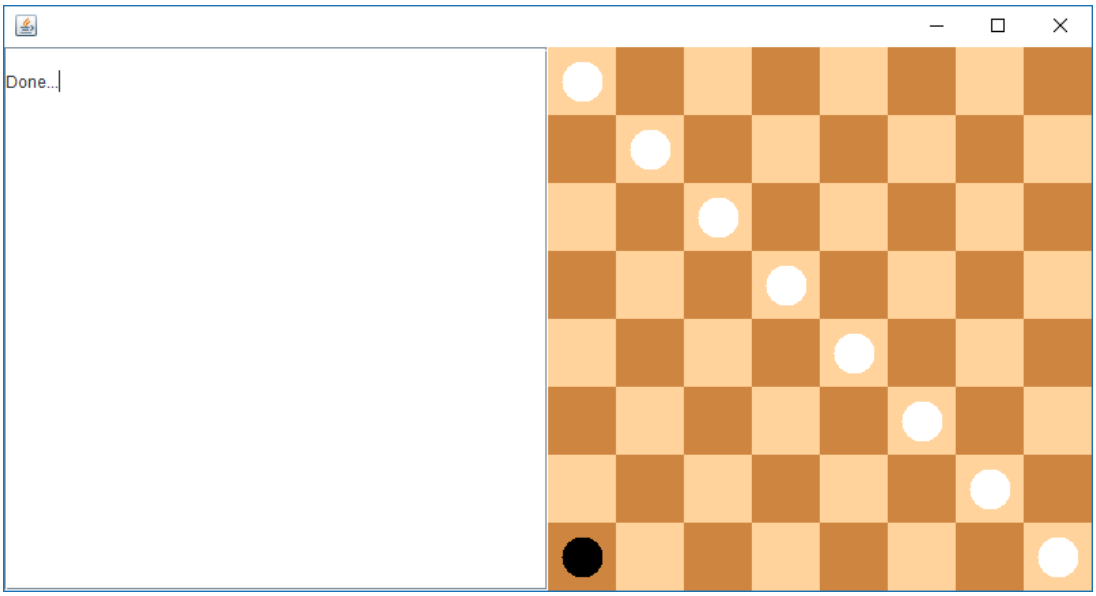

Abbildung 1: Log links, Schachbrett rechts

2.4 Hilfs-Methoden

Zum Vereinfachen wurde die Datenklasse der Position erstellt und der Map noch Hilfs-Methoden hinzugefügt. Diese zwei Methoden geben die Positionen der Bauern bzw. des Turms zurück, indem sie alle Felder der Matrix überprüfen:

$$public\ Position\ getTurmPosition();$$

$$public\ ArrayList < Position >\ getBauerPositions();$$

Und auch zwei weitere Methoden zum Fangen des Turmes:

$$public\ boolean\ TurmIsCatchAble();$$

$$public\ Position\ getCatchPosition();$$

Wobei Ersteres nur überprüft, ob der Turm in einem Zug gefangen werden kann und Zweiteres die Position des Bauern zurückgibt, für den diese Möglichkeit besteht.

3 Teilaufgaben

3.1 Aufgabe 1

Es werden nun, wie in Kapitel 1.1 beschrieben, die acht Bauern auf einer Diagonalen des Feldes instanziiert, wobei die Position des Turmes bei der Instanziierung keine Rolle spielt und zur Veranschaulichung haben wir ihn einfachheitshalber in die linke untere Ecke gesetzt, wie in Abbildung 1 zu sehen ist. Nun tätigen die Bauern und der Turm abwechselnd jeweils einen Zug, wobei es wie gesagt keine Rolle spielt, was für einen Zug der Turm tätigt, da die Bauern ihn in einen Teil der Feldes eingeschlossen haben und keine Lücke öffnen um ihn hindurchzulassen. Die Bauern rücken nun einer nach dem Anderen stetig näher an die Kante und schränken so den Bereich des Turms ein ohne eine Lücke entstehen zu lassen. Wie in Abbildung 2 zu sehen, verkleinert sich dadurch der mögliche Handlungsbereich des Turms bis er in Abbildung 3 letztendlich gefangen ist.

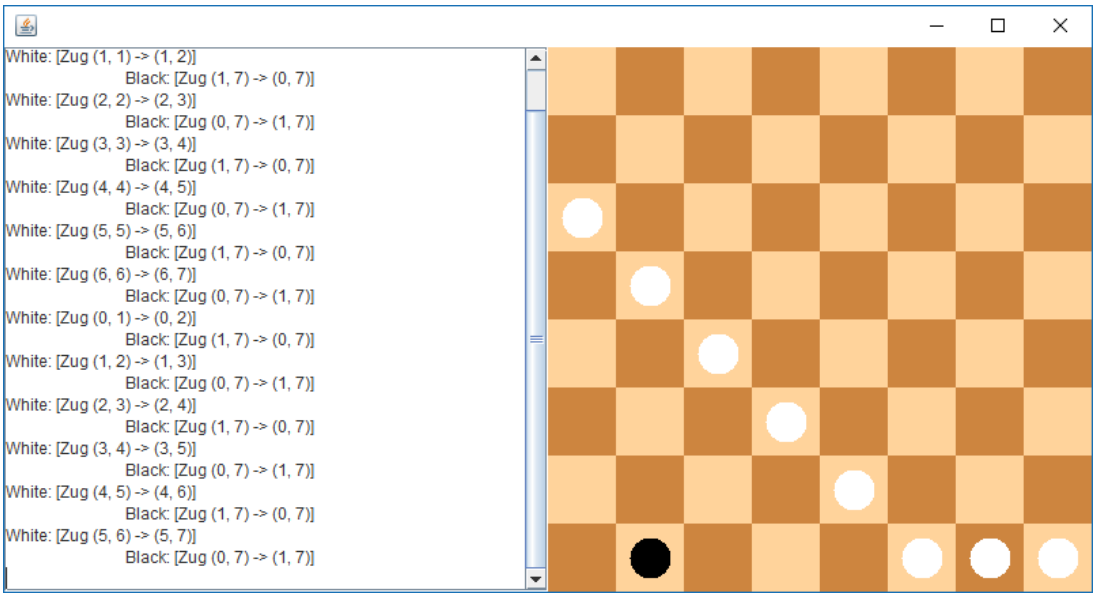

Abbildung 2: Foranschreiten bei der Reproduktion

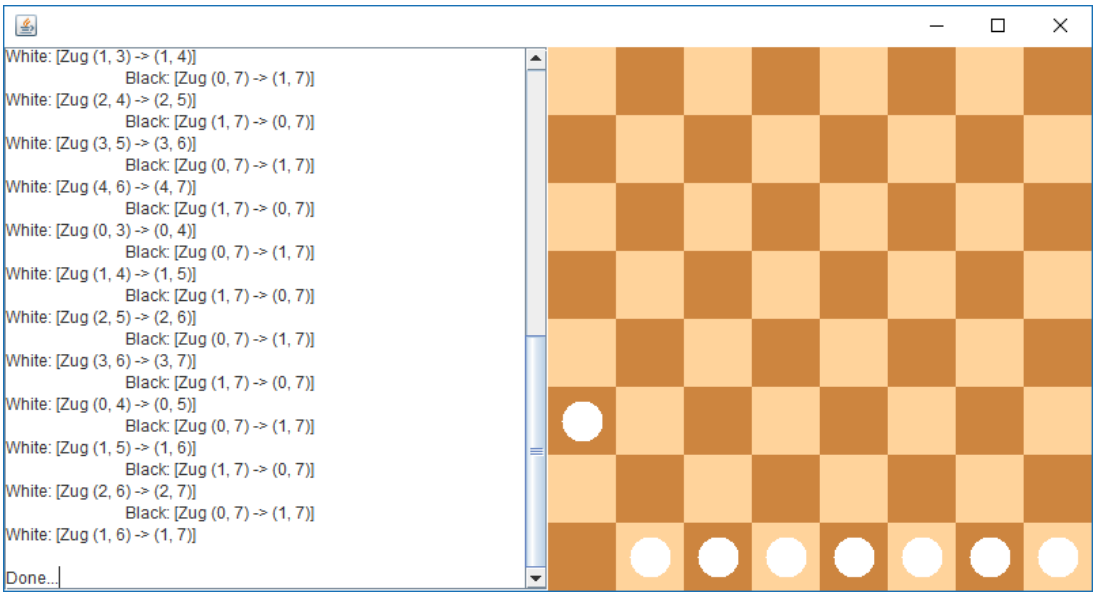

Abbildung 3: Der Turm wurde gefangen

3.2 Aufgabe 2

Aufgrund der in Kapitel 1.2 beschriebenen Patt-Situation, fehlt nur noch die graphische Aufarbeitung zur Erfüllung der Aufgabenanforderung. Dazu benutzen wir die in Kapitel 2.3 erwähnte Loop-Funktion der GUI, sodass letztendlich die GUI, wie in Abbildung 4 dargestellt, endlos diese vier Züge wiederholt.

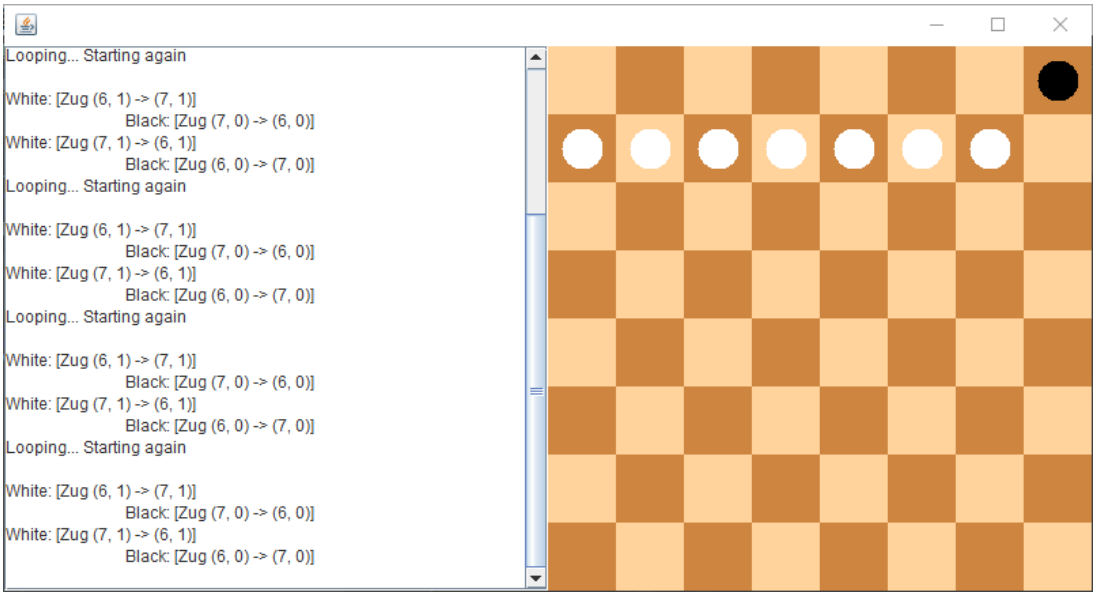

Abbildung 4: Der Turm kann nicht gefangen werden

4 Quellcode

4.1 Datenstrukturen

4.1.1 Figur

Listing 1: Figur mit Speicherung, ob es ein Turm oder Bauer ist

```
public class Figur {
    public boolean bauer;

    /**
     * @param color, true = white Bauer; false = black Turm
     */
    public Figur(boolean color) {
        this.bauer = color;
    }
}
```

4.1.2 Position

Listing 2: Aktuelle Position einer Figur auf dem Brett

```
public class Position {
    public int x;
    public int y;

    public Position(int posX, int posY) {
        this.x = posX;
        this.y = posY;
    }
}
```

4.1.3 Zug

Listing 3: Datenspeicherung der einzelnen Bewegungen von den Figuren

```
public class Zug {
    public int x;
    public int y;
    public int toX;
    public int toY;

    public Zug(int x, int y, int toX, int toY) {
        this.x = x;
        this.y = y;
        this.toX = toX;
        this.toY = toY;
    }

    @Override
    public String toString() {
        return "[Zug (" + x + ", " + y + ") -> (" + toX + ", " + toY + ")]";
    }
}
```

4.2 Map

Listing 4: Implementierung des Spielbretts

```
import java.util.ArrayList;

public class Map {
    private Figur[][] map; // 1. Dim ist A-H; 2. Dim ist 1-8
    public Figur[][] initalMap; // wird nur zur Visualsierung, um den Initalzustand zu haben,
        benutzt
    public ArrayList<Zug> verlauf; // zur Visualisierung
    public boolean isTurmGefangen;

    public Map() {
        this.reset();
    }

    /**
     *  resettet alles
     */
    public void reset() {
        map = new Figur[8][8];
        initalMap = new Figur[8][8];
        verlauf = new ArrayList<>();
    }

    /**
     * @return Position des Turm auf dem Brett
     */
    public Position getTurmPosition() {
        for (int i = 0; i < 8; i++) {
            for (int k = 0; k < 8; k++) {
                if (map[i][k] != null && !map[i][k].bauer) {
                    return new Position(i, k);
                }
            }
        }
        System.exit(2);
        return null;
    }

    /**
     * @return Liste der Positionen der Bauern auf dem Brett
     */
    public ArrayList<Position> getBauerPositions() {
        ArrayList<Position> positions = new ArrayList<>();
        for (int i = 0; i < 8; i++) {
            for (int k = 0; k < 8; k++) {
                if (map[i][k] != null && map[i][k].bauer) {
                    positions.add(new Position(i, k));
                }
            }
        }
        return positions;
    }

    /**
     * @param x X-Position des Bauern
     * @param y Y-Position des Bauern
     */
    public void spawnBauer(int x, int y) {
        if (x <= 7 && x >= 0 && y <= 7 && y >= 0) {
            if (map[x][y] == null) {
                map[x][y] = new Figur(true);
                initalMap[x][y] = map[x][y];
                return;
```

```
        }
    }
    System.exit(1);
}

/**
 * @param x X-Position des Turm
 * @param y Y-Position des Turm
 */
public void spawnTurm(int x, int y) {
    if (x <= 8 && x >= 0 && y <= 8 && y >= 0) {
        if (map[x][y] == null) {
            map[x][y] = new Figur(false);
            initalMap[x][y] = map[x][y];
            return;
        }
    }
    System.exit(1);
}

public boolean TurmIsCatchAble() {
    Position p = this.getTurmPosition();
    ArrayList<Position> positions = this.getBauerPositions();
    for (Position px : positions) {
        if ((px.x + 1 == p.x && p.y == px.y) || (px.x - 1 == p.x && p.y == px.y) || (px.x ==
            p.x && p.y + 1 == px.y) || (px.x == p.x && p.y - 1 == px.y)) {
            return true;
        }
    }
    return false;
}

public Position getCatchPosition() {
    Position p = this.getTurmPosition();
    ArrayList<Position> positions = this.getBauerPositions();
    for (Position px : positions) {
        if ((px.x + 1 == p.x && p.y == px.y) || (px.x - 1 == p.x && p.y == px.y) || (px.x ==
            p.x && p.y + 1 == px.y) || (px.x == p.x && p.y - 1 == px.y)) {
            return px;
        }
    }
    return null;
}

 */
public void doMove(Zug move) {
    System.out.println("Fuehre Zug aus" + move.toString());
    if (map[move.x][move.y] == null) {
        System.out.println("Error: Da ist keine Figur, die bewegbar ist");
        System.exit(0);
    } else if (map[move.toX][move.toY] != null && map[move.toX][move.toY].bauer) {
        System.out.println("Error: Auf dem Zielfeld ist bereits eine Figur (Bauer)
            vorhanden");
        System.exit(0);
    } else {
        if (map[move.toX][move.toY] != null && !map[move.toX][move.toY].bauer) {
            isTurmGefangen = true;
        }
        map[move.toX][move.toY] = map[move.x][move.y];
        map[move.x][move.y] = null;
        verlauf.add(move);
    }
}
```

```
}
```

4.3 GUI

Listing 5: Implementierung der Grafischen Ausgabe

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class GUI extends JPanel {
    private Map map;
    private Figur[][] brett; // auf diesem Brett wird der ganze Verlauf erneut durchsimuliert
    private Timer t; // schrittweise Reproduktion der Schritte
    private int atMove;
    private JTextArea log;
    private boolean loop;

    public GUI(Map m, JTextArea tOutput, boolean loop) {
        this.map = m;
        this.log = tOutput;
        this.loop = loop;
        brett = m.initalMap;
        this.atMove = 0;

        // Initalisierung des Timers zur Visualisierung
        this.t = new Timer(200, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (atMove == map.verlauf.size() && loop) {
                    System.out.println("Looping... Starting again");
                    log.append("Looping... Starting again\n\n");
                    atMove = 0;
                }

                if (atMove == map.verlauf.size() || map.verlauf.size() == 0) {
                    System.out.println("Finished with redo");
                    log.append("\nDone...");
                    t.stop();
                } else {
                    Zug m = map.verlauf.get(atMove++);
                    System.out.println("Redo: " + m.toString());
                    if (brett[m.x][m.y].bauer) {
                        log.append("White: " + m.toString() + "\n");
                    } else {
                        log.append("\tBlack: " + m.toString() + "\n");
                    }
                    brett[m.toX][m.toY] = brett[m.x][m.y];
                    brett[m.x][m.y] = null;
                    repaint();
                    log.repaint();
                }
            }
        });
        t.start();
    }

    @Override
    public void paintComponent(Graphics g) {
        boolean color = true;
        for (int i = 0; i < 8; i++) {
```

```
            for (int n = 0; n < 8; n++) {
                if (color) {
                    g.setColor(new Color(255, 211, 155));
                } else {
                    g.setColor(new Color(205, 133, 63));
                }
                color = !color;
                g.fillRect(i * 50, n * 50, 50, 50);

                // Draw Figur
                if (brett[i][n] != null) {
                    if (brett[i][n].bauer) {
                        g.setColor(Color.WHITE); // Bauer
                    } else {
                        g.setColor(Color.BLACK); // Turm
                    }
                    g.fillOval(i * 50 + 10, n * 50 + 10, 30, 30);
                }
            }
            color = !color;
        }
    }
}
```

4.4 Aufgabe 1

Listing 6: Implementierung der Aufgabe 1 nach Lösungsansatz

```
public static void teilaufgabe1(Map m) {
   loop = false;
   //spawnen der Bauern auf der Diagonalen
   for (int i = 0; i < 8; i++) {
      m.spawnBauer(i, i);
   }

   m.spawnTurm(0, 7);
   boolean bauerZug = true;
   int turmGezogen = 0;
   int bauernGezogen = 0;
   int anzDurchgaenge = 0;
   System.out.println("Turm gefangen: " + m.isTurmGefangen());
   while (!m.isTurmGefangen()) {
      if (bauerZug) {
         if (m.TurmIsCatchAble()) {
            Position p = m.getCatchPosition();
            m.doMove(new Zug(p.x, p.y, m.getTurmPosition().x, m.getTurmPosition().y));
         } else {
            // Zumachen der Diagonalen zu einer Linie
            Position p = m.getTurmPosition();
            Position p2 = m.getBauerPositions().get(0);
            if (p2.x <= p.x && p2.y < p.y) {
               Position p3 = m.getBauerPositions().get(bauernGezogen++);
               m.doMove(new Zug(p3.x, p3.y, p3.x, p3.y + 1));

            } else {
               Position p3 = m.getBauerPositions().get(bauernGezogen++);
               m.doMove(new Zug(p3.x, p3.y, p3.x + 1, p3.y));
            }
            if (bauernGezogen + anzDurchgaenge == 7) {
               bauernGezogen = 0;
               anzDurchgaenge++;
            }
```

```
        }
      } else {
        // Der Turm bewegt sich abwechselnd nur um 1 Feld nach rechts oder links
        Position p = m.getTurmPosition();
        Position p2 = m.getBauerPositions().get(0);
        int modulo = 1;
        if (!(p2.x <= p.x && p2.y < p.y)) {
          modulo = 0;
        }
        if (turmGezogen % 2 == modulo) {
          m.doMove(new Zug(p.x, p.y, p.x - 1, p.y));
        } else {
          m.doMove(new Zug(p.x, p.y, p.x + 1, p.y));
        }
        turmGezogen++;
      }
      bauerZug = !bauerZug;
    }
    System.out.println("Turm gefangen!");
}
```

4.5 Aufgabe 2

Listing 7: Implementierung der Aufgabe 2 nach Lösungsansatz

```
public static void teilaufgabe2(Map m) {
    loop = true;
    //spawnen der Bauern auf einer Linie
    for (int i = 0; i < 7; i++) {
        m.spawnBauer(i, 1);
    }
    m.spawnTurm(7, 0);
    m.doMove(new Zug(6, 1, 7, 1));
    m.doMove(new Zug(7, 0, 6, 0));
    m.doMove(new Zug(7, 1, 6, 1));
    m.doMove(new Zug(6, 0, 7, 0));
}
```