# Einführung in die Informatik II: Programmierbeleg

*Martin Schuster --- Seminargruppe: Mi17w2*




Martin Schuster
27.7.2018

# Inhaltsverzeichnis

# 1. Aufgabenstellung

Diese Belegarbeit befasst sich mit dem Computergestützten lösen von Problemstellungen. Das Ziel war es ein Programm auf Java-Basis zu entwickeln, welches in der Lage ist ein Labyrinth unter Verwendung von wahlweise einem von zwei Algorithmen zu erstellen und dies anschließend zu lösen.

Für die beiden Algorithmen, die zur Erstellung verwendet werden sollten, wurde der „Recursive Backtracking Algorithmus“ und die „Recursive division method“ gewählt.

Zum Lösen des Labyrinthes wurde der „A-Stern Algorithmus“ gewählt.

Die Darstellung sollte über ein einfaches grafisches Menü dargestellt werden, zwei Canvas sind für die Anzeige des Labyrinthes und die Anzeige des Lösungsweges vorgesehen. Sowie zwei Buttons zum Starten der Erstellung und dem Starten des Lösungsalgorithmus.

Optional war die Möglichkeit gegeben das Labyrinth aus Hex Feldern oder Zylindrisch zu gestalten.

# 2. Spezifikation, Problemanalyse, Lösungskonzeption

## 2.1 Spezifikation

Die Anforderungen an das erstellte Programm sind die Möglichkeit ein Labyrinth zu erstellen und dieses auch lösen zu können.

Für die Erstellung stehen dem Nutzer zwei Algorithmen zur Verfügung. Zum einen der rekursive Backtrack Algorithmus und als zweites der rekursive Division Algorithmus. Die Größe des Labyrinthes kann von dem Nutzer mit einer Zellgröße von 2 mal 2 bis 64 mal 64 Pixel eingestellt werden.

Mittels des A-Stern Algorithmus erfolgt die Lösung des Labyrinthes, welcher den bzw. einen der optimalen Wege vom oberen linken zum unteren rechten Punkt durch das Labyrinth findet.

Zur Bedienung des Programmes steht eine einfache grafische Benutzeroberfläche zur Verfügung. Die Ansicht ist zweigeteilt, die linke Seite beinhaltet alles für das Erstellen des Labyrinthes, ein Canvas zur Darstellung des erstellten Labyrinthes, einem Slider zur Einstellung der Zellgröße, 2 Radio Buttons zur Auswahl des Erstellung Algorithmus und einen Button der das Generieren des Labyrinths auslöst. Die rechte Seite verfügt über ein Canvas welches den Pfad vom Start zum Endpunkt anzeigt, eine Legende zur Verdeutlichung was die Farbigen Punkte des Pfades Bedeuten, einen Label welches den Nutzer darauf hinweist ob ein Pfad gefunden wurde oder nicht und einen Lösen Button welcher den Start des A-Stern Algorithmus auslöst.

## 2.2 Problemanalyse

Durch die Spezifikation der Aufgabenstellung, ergaben sich die folgende Problemstellungen welche zum erfolgreichen Abschließen der Aufgabe nötig waren zu bewältigen.

Es musste sich mit der Erstellung eines Labyrinths auseinandergesetzt werden, dies bedeutete erstmal, dass ein allgemeines Verständnis was ein Labyrinth ausmacht erarbeitet werden musste. Damit das Erstellen eines Labyrinthes Dynamisch (wahlweise Größe des Labyrinthes) möglich war musste eine Methode bzw. Algorithmus gefunden werden mit dem dieses Vorhaben möglich war.

Da jedes Labyrinth möglichst einzigartig ist, durch die Erstellung durch einen Algorithmus und die Verwendung von zufällig gewählten Feldern, musste auch um das Lösen des Labyrinthes zu realisieren ein Algorithmus verwendet werden. Um diese Aufgabe zu bewältigen war ein einarbeiten in die Thematik der Wegfindung notwendig.

Für die Darstellung des Programmes mussten entsprechend weitere Kenntnisse in der JavaFX Programmierung vertieft werden.

## 2.3 Lösungskonzeption

Die Hauptproblemfelder, die drei Algorithmen zur Erstellung und Lösung des Algorithmus, konnten durch das wählen des rekursiven Backtrack Algorithmus und des rekursiven Division Algorithmus für die Erstellung und den A-Stern Algorithmus für die Lösung des Labyrinths bewältigt werden diese verfahren werden im Folgenden genauer vorgestellt.

### 2.3.1 Rekursiv Backtracker Algorithmus

Der rekursive Backtracker (Rückverfolgung) Algorithmus ist eine Problemlösungsmethode aus dem Gebiet der Algorithmik. Das Vorgehen des Algorithmus erfolgt nach dem „Versuch und Irrtum Prinzip“, es werden solange zulässige Lösungsmöglichkeiten versucht bis die gewünschte Lösung gefunden wurde, sollte bei der Durchführung der Algorithmus an einen Punkt kommen wo es nicht weitergeht wird werden die Schritte solange zurückverfolgt

(Backtrack) bis ein weiterführen möglich ist oder es definitiv gesagt werden kann das keine Lösung für dieses Problem existiert. Die Laufzeit für diesen Algorithmus ist im schlimmsten falle $O(z^N)$ $z$ mögliche Verzweigungen $N$ maximale Tiefe.

Der grobe Ablauf des Algorithmus zur Erstellung eines Labyrinthes erfolgt wie folgt. Es wird eine erste Zelle Ausgewählt (zufällig oder festgelegt) und diese wird als besucht markiert. Solange es noch unbesuchte Zellen gibt wird jeweils von der aktuellen Zelle aus geprüft welche Nachbar Zelle, jeweils die 4 Richtungen oben, rechts, unten, links noch nicht besucht worden. Wenn es Nachbarn gibt, auf welche diese Bedingung zutrifft existieren, wird zufällig einer dieser Ausgewählt. Nachdem auswählen wird die Aktuelle Zelle auf ein Stack verschoben und die Wand zwischen der Aktuellen Zelle und der neuen Zelle entfernt, daher kommt auch die Bezeichnung im Kontext der Labyrinth Erstellung des „Wall-Carver“ Algorithmus, zum Schluss wird die neue Zelle als aktuelle Zelle markiert und wiederum als besucht markiert. Dies wird solange wiederholt bis die Aktuelle Zelle keine unbesuchten Nachbarn mehr hat. Dann erfolgt der zweite Schritt des Algorithmus, der Stack wird solange Rückverfolgt (Backtrack) und nach neuen möglichen Zellen (mit unbesuchten Nachbarn) gesucht bis der Stack leer ist so das jedes Feld im Grunde genommen 2-mal abgefragt wird.

### 2.3.2 Rekursiv Division Algorithmus

Im Gegensatz zum rekursiv Backtrack Algorithmus, Arbeitet der rekursiv Division Algorithmus als ein sogenannter „Wall-adder“. Dies bedeutet zu Begin ist das Feld noch leer und die Wände des Labyrinthes werden nach und nach hinzugefügt.

Mit dem ersten Schritt wird das leere Feld, entweder zufällig oder gewichtet ob eine Seite des Feldes länger ist als die andere, horizontal oder vertikal geteilt. Danach wird zufällig in der teilenden Wand eine Lücke gelassen. Diese zwei Schritte werden solange auf jeweils den nun beiden Feldseiten ausgeführt bis die Zellen die gewünschte Größe erreicht haben. Der Vorteil dieses Algorithmus gegenüber dem Backtrack Algorithmus ist, dass zu jedem Zeitpunkt ein gültiges Labyrinth entsteht und mit jedem Schritt mehr Details und Komplexität hinzugefügt werden, theoretisch könnte dies unendlich lang fortgesetzt werden.

### 2.3.3 A-Stern Algorithmus

Damit der Pfad durch das Labyrinth ermittelt werden kann musste ein Algorithmus gewählt werden der dies Bewerkstelligen kann. Für dieses Problem wurde sich für den A-Stern Algorithmus aus der Klasse der informierten Suchalgorithmen entschieden, das erste Mal von Peter Hart, Nils J. Nilsson und Bertram Raphael 1968 beschrieben. Die Wahl für diesen Suchalgorithmus wurde aufgrund der Tatsache getroffen das der Algorithmus vollständig optimal ist was bedeutet, wenn das Labyrinth mit dem gewählten Start und Endpunkt eine Lösung hat wird eine optimale Lösung gefunden.

Die Funktionsweise des Algorithmus ist wie folgt. Es wird am Startknoten begonnen allen umliegenden Feldern einen Kostenwert, welcher die Kosten bzw. Reichweite bis zum Ziel enthält, zuzuordnen. Dies wird mittels einer Heuristik berechnet, für dieses Programm wurde die Manhattan-Heuristik verwendet. Die Distanz zum Ziel wird aus der Summe der absoluten Differenzen der Einzelkoordinaten berechnet.

$$d(a,b) = \sum_{i} |a_i - b_i|$$

Bei dieser Heuristik werden bewusst etwaige Hindernisse ignoriert und nur der kürzeste Weg zum Ziel genommen. Wenn alle kosten der umliegenden Felder bestimmt wurden wird zum nächsten Feld fortgeschritten welches die niedrigsten kosten zum Ziel hat, dies geschieht solange bis man das Zielfeld erreicht hat oder es feststeht das kein Weg zum Ziel existiert.

# 3. Software-Architektur

## 3.1 Klassendiagramm Konzept

Die Erstellung der ersten Klassendiagramm Konzeption, wurde mittels der erarbeitetet Anforderungen für die einzelnen Algorithmen gestaltet und für die Elemente der grafischen Oberfläche wurde eine einfaches Desing Mocup erstellt, um die Anzahl der benötigten Elemente besser abschätzen zu können.

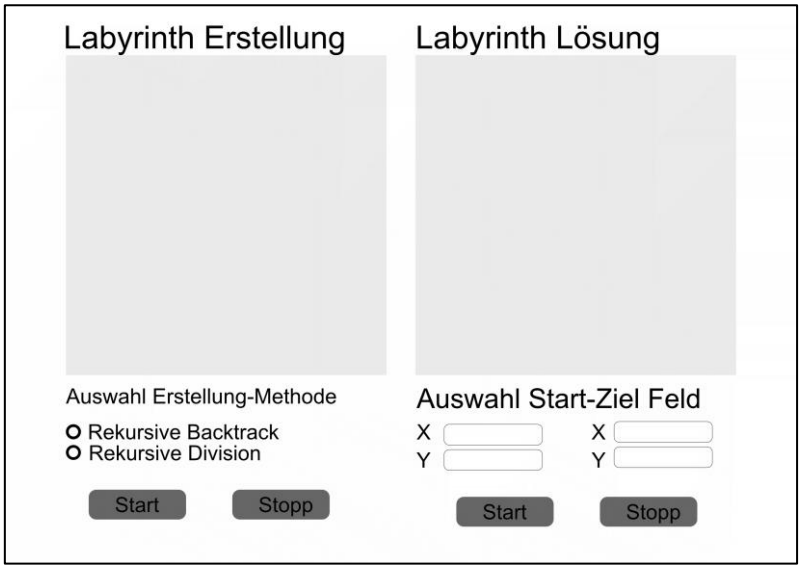



*Abbildung 1 Design Mocup*

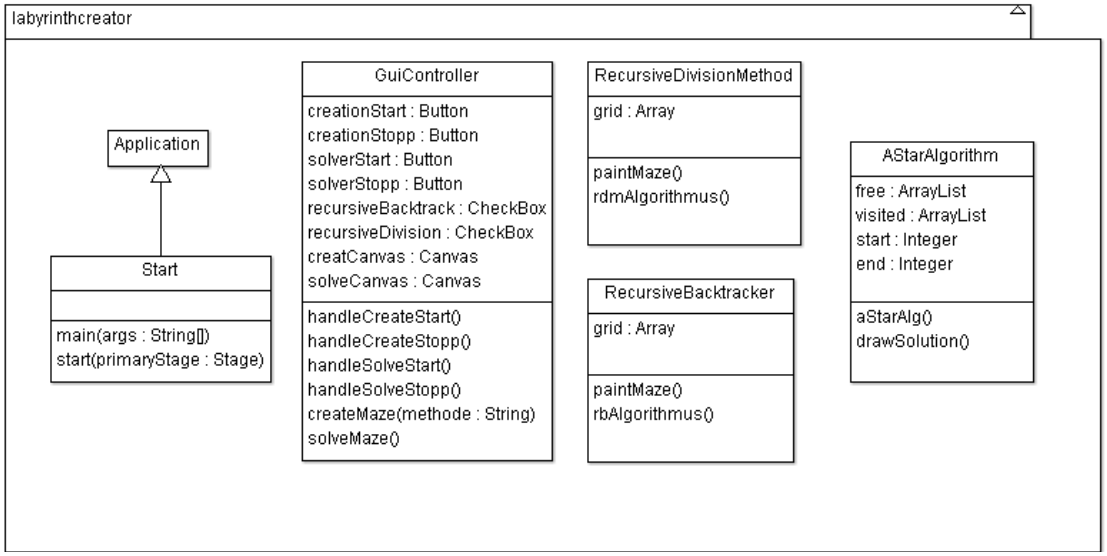
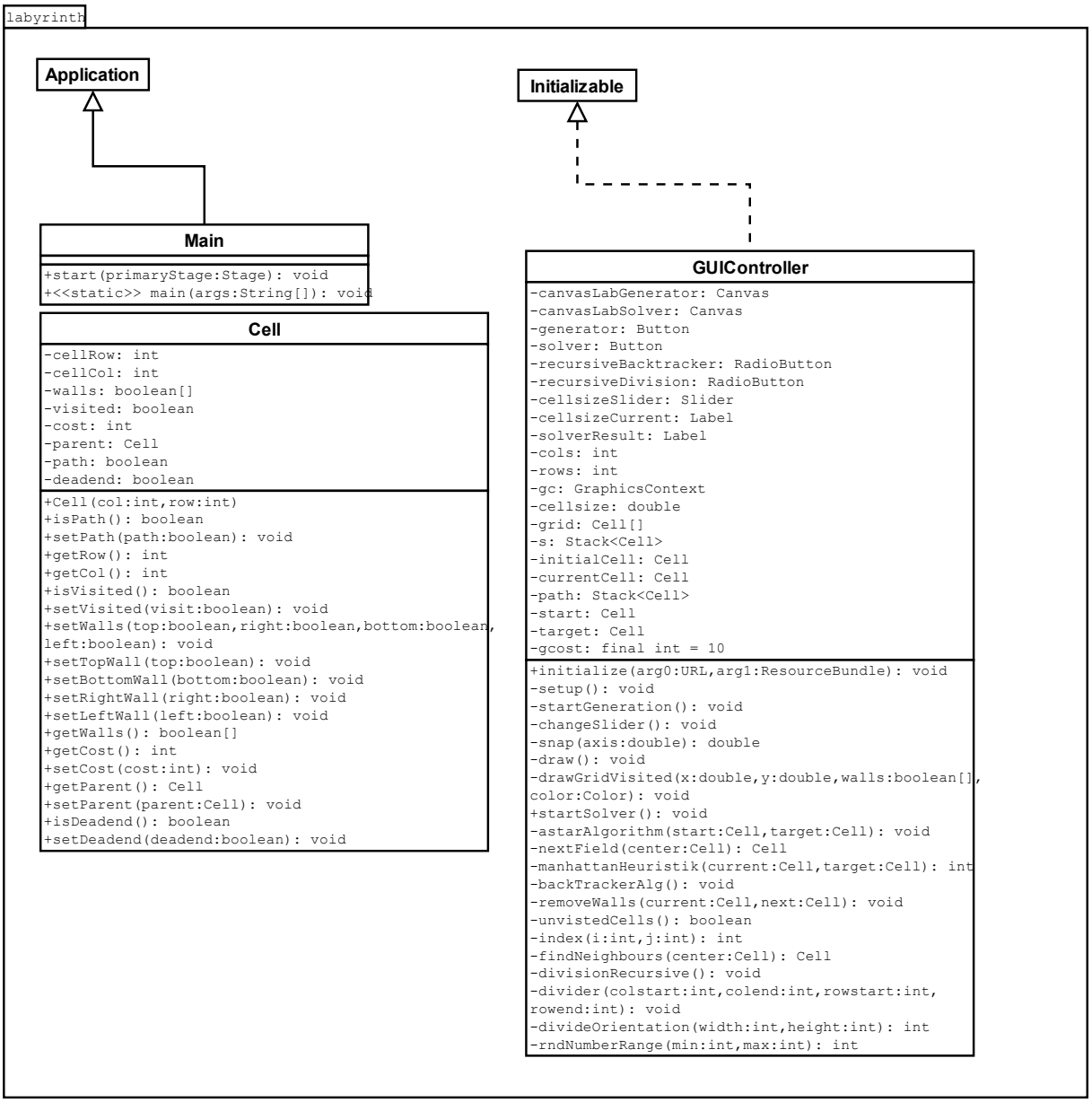



*Abbildung 2 Klassendiagramm Konzept*

## 3.2 Klassendiagramm Final

Das finale Klassendiagramm hat sich dahingehend verändert das die Auslagerung der 3 Algorithmen weggefallen ist alle 3 befinden sich nun in der GUIController Klasse. Dies hat hauptsächlich Zeitmanagement Gründe gehabt. Dies wäre für eine etwaige Weiterverwendung von Teilen des Programmes nützlich gewesen aber so ist es nur für die Lesbarkeit des Codes hinderlich. Ansonsten wurde das grobe Konzept beibehalten aber es sind wesentlich mehr Attribute und Methoden zusammengekommen als ursprünglich geplant.



# 4. Implementierungsdetails

## 4.1 Main Klasse

Die Main Klasse ist der Startpunkt für die Anwendung. Hier wird mittels der FXMLLoader Methode, die durch den Scene Builder erstellte Fxml Datei geladen, diese Enthält die Formatierungsinformationen für alle Elemente die in der grafischen Oberfläche verwendet werden.

```
1. package labyrinth;
2. import javafx.fxml.FXMLLoader;
3. import javafx.application.Application;
4. import javafx.stage.Stage;
5. import javafx.scene.Scene;
6. import javafx.scene.layout.AnchorPane;
7. /** * Main Klasse Startpunkt der Anwendung * Laden der GUI.fxml erstellt aus dem S
   cene Builder *  * @author Martin Schuster * @version 1.0 */
8. public class Main extends Application {@
9.     Override public void start(Stage primaryStage) {
10.             try {
11.                 /*           * Einbinden der GUI.fxml datein aus SceneBuilder
          */
12.                 FXMLLoader loader = new FXMLLoader();
13.                 loader.setLocation(Main.class.getResource("GUI.fxml"));
14.                 AnchorPane root = loader.load();
15.                 Scene scene = new Scene(root, 700, 600);
16.                 primaryStage.setTitle("Labyrinth Generator & Löser");
17.                 primaryStage.setScene(scene);
18.                 primaryStage.show();
19.             } catch (Exception e) {
20.                 e.printStackTrace();
21.             }
22.         }
23.         /**  * Startpunkt    * @param args Kommandozeilen Parameter  */
24.     public static void main(String[] args) {
25.         launch(args);
26.     }
27. }
```

## 4.2 Cell Klasse

Für die Organisation des Labyrinths, wurde eine eigene Klasse gewählt, die Einzelnen Felder werden in einem Array vom Typ Cell gespeichert. Jedes Feld hat damit eine Spalte und Zeile und kann mehrere Eigenschaften besitzen, diese sind zum Teil für das Erstellen wichtig (visited, walls), aber hauptsächlich dienen diese dazu die die Arbeit des A-Stern Algorithmus zu leiten (cost, parent, path, deadend).

```
1.  package labyrinth;
2.  /** *  * Cell Klasse beinhaltet alle r
    elevanten Informationen zum Erstellen
    * und Navigieren des Labyrinths *  * @
    author Martin Schuster  * @version 1.0
     */
3.  public class Cell {
4.      /*   * Relevante Variablen für die
     Erstellung des Labyrinths     *   */

5.      private int cellRow; //Zeile der Z
    elle
6.      private int cellCol; //Spalte der
    Zelle
7.      private boolean[] walls = new bool
    ean[4]; //Stelle an der die Zelle eine
     Wand hat oben, rechts, unten, links
8.      private boolean visited; //wurde d
    ie Zelle besucht relevant für die Erst
    ellung des Labyrinths
9.      /*   * Relevante Variablen für die
     Lösung des Labyrinths     *   */
10.     private int cost; //Kosten der Zel
    le um zum Zielpunkt zu gelangen
11.     private Cell parent; //Zelle von d
    er die Aktuelle Zelle besucht wird
12.     private boolean path; //gehört die
     Aktuelle Zelle zum ziel Pfad
13.     private boolean deadend; //Ist die
     Aktuelle Zelle eine Sackgasse kosten
    sind geringer als andere Zelle aber ke
    in erreichen des Zieles möglich
14.     /**  * Konstruktor Neue Celle mit
    standard werten Initialisieren     *
     * @param col Spalte der Zelle als int
       * @param row Reihe der Zelle als in
    t    */
15.     public Cell(int col, int row) {
16.             cellCol = col;
17.             cellRow = row;
18.             visited = false;
19.             cost = 0;
20.             parent = null;
21.             path = false;
22.             deadend = false;
23.             for (int i = 0; i < walls.
    length; i++) {
24.                 walls[i] = true;
25.             }
26.         }
27.         /*   * Getter und Setter für d
    ie Variablen   */
28.         /**  * Gibt an ob Zelle zum Zi
    elpfad gehört  * @return true oder fal
    se   */
29.     public boolean isPath() {
30.             return path;
31.         }
32.         /**  * Setzt den Status ob die
     Zelle zum Zielpfad gehört     * @para
    m path bekommt einen true oder false W
    ert    */
33.     public void setPath(boolean path)
    {
34.             this.path = path;
35.         }
36.         /**  * Gibt die Reihe der Zell
    e zurück   * @return  Zeile als int We
    rt   */
37.     public int getRow() {
38.             return cellRow;
39.         }
40.         /**  * Gibt die Spalte der Zel
    le zurück  * @return Spalte als int We
    rt   */
41.     public int getCol() {
42.             return cellCol;
43.         }
44.         /**  * Gibt an ob die Zelle be
    sucht wurde    * @return true oder fal
    se   */
45.     public boolean isVisited() {
46.             return visited;
47.         }
48.         /**  * Setzt den Besucht zusta
    nd der Zelle   * @param visit bekommt
    true oder false übergeben    */
49.     public void setVisited(boolean vis
    it) {
50.             visited = visit;
```

```
51.         }
52.         /**  * Setzt die Wänder der Ze
    lle    * @param top obere Wand true od
    er false     * @param right rechte Wan
    d true oder false  * @param bottom unt
    ere Wand true oder false     * @param
    left linke Wand true oder false    */

53.     public void setWalls(boolean top,
    boolean right, boolean bottom, boolean
     left) {
54.             walls[0] = top;
55.             walls[1] = right;
56.             walls[2] = bottom;
57.             walls[3] = left;
58.         }
59.         /**  * Setzt die obere Wand de
    r Zelle    * @param top true oder fals
    e    */
60.     public void setTopWall(boolean top
    ) {
61.             walls[0] = top;
62.         }
63.         /**  * Setzt die untere Wand e
    iner Zelle     * @param bottom true od
    er false     */
64.     public void setBottomWall(boolean
    bottom) {
65.             walls[2] = bottom;
66.         }
67.         /**  * Setzt die rechte Wand e
    iner Zelle     * @param right true ode
    r false  */
68.     public void setRightWall(boolean r
    ight) {
69.             walls[1] = right;
70.         }
71.         /**  * Setzt die linke Wand ei
    ner Zelle  * @param left true oder fal
    se   */
72.     public void setLeftWall(boolean le
    ft) {
73.             walls[3] = left;
74.         }
```

```
75.         /**  * Gibt die Wände einer Ze
    lle zurück     * @return boolean Array
      mit true oder false Werten     */
76.     public boolean[] getwalls() {
77.             return walls;
78.         }
79.         /**  * Gibt die Kosten der Zel
    le zum Zielpunkt zurück    * @return K
    osten als int Wert   */
80.     public int getCost() {
81.             return cost;
82.         }
83.         /**  * Setzt die Kosten der Ze
    lle zum Zielpunkt  * @param cost Koste
    n als int Wert   */
84.     public void setCost(int cost) {
85.             this.cost = cost;
86.         }
87.         /**  * Gibt die vorgänger Zell
    e der aktuellen Zelle zurück   * @retu
    rn Vorgänger Zelle typ Cell  */
88.     public Cell getParent() {
89.             return parent;
90.         }
91.         /**  * Setzt die vorgänger Zel
    le der aktuellen Zelle     * @param pa
    rent Vorgänger Zelle typ Cell    */
92.     public void setParent(Cell parent)
     {
93.             this.parent = parent;
94.         }
95.         /**  * Gibt an ob die aktuelle
     Zelle eine Sackgasse ist  * @return t
    rue oder false   */
96.     public boolean isDeadend() {
97.             return deadend;
98.         }
99.         /**  * Setzt die aktuelle Zell
    e als Sackgasse    * @param deadend tr
    ue oder false    */
100.          public void setDeadend(bool
    ean deadend) {
101.              this.deadend = deadend;

102.          }
103.      }
```

## 4.3 Rekrusive Backtrack Algotihmus

Der rekursive Backtrack Algorithmus arbeitet als „Wall-Carver“, sprich das Grid für das Labyrinth ist zunächst mit Wänden gefüllt und diese werde nach und nach solange reduziert bis das Labyrinth entstanden ist. Die Umsetzung für diesen Algorithmus erfolgte über folgenden Pseudocode aus der Wikipedia Quelle zum Thema „Maze generation algorithm“:

1. Make the initial cell the current cell and mark it as visited
2. While there are unvisited cells
    1. If the current cell has any neighbours which have not been visited
        1. Choose randomly one of the unvisited neighbours
        2. Push the current cell to the stack
        3. Remove the wall between the current cell and the chosen cell
        4. Make the chosen cell the current cell and mark it as visited
    2. Else if stack is not empty
        1. Pop a cell from the stack
        2. Make it the current cell

Im Programm sieht das Ganze dann wie folgt aus.

```
1.  private void backTrackerAlg() {
2.      initialCell = grid[0];
3.      s.clear();
4.      currentCell = initialCell;
5.      Cell next;
6.      currentCell.setVisited(true);
7.      while (unvisitedCells()) {
8.          next = findNeighbours(currentCell);
9.          if (next != null) {
10.             next.setVisited(true);
11.             s.push(currentCell);
12.             removeWalls(currentCell, next);
13.             currentCell = next;
14.         } else if (s.size() >= 0) {
15.             currentCell = s.pop();
16.         }
17.     }
```

Die folgenden Methoden waren zusätzlich nötig bzw. wurden ausgelagert um den Backtrack Algorithmus erfolgreich umsetzen zu können.

Die Methode „removeWalls“ löscht die Wand zwischen den übergebenen Zellen. Da jede Zelle seine eigenen Wände besitzt mussten auch jeweils 2 Wände entfernt werden.

```
1.  private void removeWalls(Cell current, Cell next) {
2.      int x = current.getCol() - next.getCol();
3.      if (x == 1) {
4.          current.setLeftWall(false);
5.          next.setRightWall(false);
6.      } else if (x == -1) {
7.          current.setRightWall(false);
8.          next.setLeftWall(false);
9.      }
10.     int y = current.getRow() - next.getRow();
11.     if (y == 1) {
12.         current.setTopWall(false);
13.         next.setBottomWall(false);
14.     } else if (y == -1) {
15.         current.setBottomWall(false);
16.         next.setTopWall(false);
17.     }
18. }
```

Mittels der „unvisitedCells“ Methode wird nach jedem durchlauf im Algorithmus geprüft ob es noch weitere unbesuchte Zellen im Grid-Array vorhanden sind.

```
1.  private boolean unvisitedCells() {
2.      boolean tmp = false;
3.      for (int i = 0; i < grid.length; i++) {
4.          if (grid[i].isVisited() == false) {
5.              tmp = true;
6.          }
7.      }
8.      return tmp;
9.  }
```

Die „index“ Methode war notwendig um anhand der Spalte und Zeile einer Zelle den Index im Grid-Array ermitteln zu können. Da diese der reihen nach von oben links bis unten rechts gespeichert wurden.

```
1.  private int index(int x, int y) {
2.      return x + y * cols;
3.  }
```

Mit der „findNeighbours“ Methode wird anhand der übergebenen Zelle alle unbesuchten Nachbarn ermittelt. Aus diesen wird dann zufällig eine ausgewählt und dem Algorithmus zurück übergeben.

```
1.  private Cell findNeighbours(Cell center) {
2.      ArrayList < Cell > neighbours = new ArrayList < Cell > ();
3.      if (center.getRow() == 0 && center.getCol() == 0) {
4.          Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
5.          Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
6.          if (!right.isVisited()) {
7.              neighbours.add(right);
8.          }
9.          if (!bottom.isVisited()) {
10.             neighbours.add(bottom);
11.         }
12.     } else if (center.getCol() == 0 && center.getRow() != 0 && center.getRow() !=
    rows - 1) {
13.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
14.         Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
15.         Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
16.         if (!top.isVisited()) {
17.             neighbours.add(top);
18.         }
19.         if (!right.isVisited()) {
20.             neighbours.add(right);
21.         }
22.         if (!bottom.isVisited()) {
23.             neighbours.add(bottom);
24.         }
25.     } else if (center.getCol() == 0 && center.getRow() == rows - 1) {
26.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
27.         Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
28.         if (!top.isVisited()) {
29.             neighbours.add(top);
30.         }
31.         if (!right.isVisited()) {
32.             neighbours.add(right);
33.         }
34.     } else if (center.getRow() == 0 && center.getCol() != 0 && center.getCol() !=
    cols - 1) {
35.         Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
36.         Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
37.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
38.         if (!left.isVisited()) {
39.             neighbours.add(left);
40.         }
41.         if (!right.isVisited()) {
42.             neighbours.add(right);
43.         }
44.         if (!bottom.isVisited()) {
45.             neighbours.add(bottom);
46.         }
47.     } else if (center.getRow() == 0 && center.getCol() == cols - 1) {
48.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
49.         Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
50.         if (!left.isVisited()) {
51.             neighbours.add(left);
52.         }
53.         if (!bottom.isVisited()) {
54.             neighbours.add(bottom);
```

```
55.         }
56.     } else if (center.getCol() == cols - 1 && center.getRow() != 0 && center.getRo
    w() != rows - 1) {
57.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
58.         Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
59.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
60.         if (!top.isVisited()) {
61.             neighbours.add(top);
62.         }
63.         if (!left.isVisited()) {
64.             neighbours.add(left);
65.         }
66.         if (!bottom.isVisited()) {
67.             neighbours.add(bottom);
68.         }
69.     } else if (center.getCol() == cols - 1 && center.getRow() == rows - 1) {
70.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
71.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
72.         if (!top.isVisited()) {
73.             neighbours.add(top);
74.         }
75.         if (!left.isVisited()) {
76.             neighbours.add(left);
77.         }
78.     } else if (center.getRow() == rows - 1 && center.getCol() != 0 && center.getCo
    l() != cols - 1) {
79.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
80.         Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
81.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
82.         if (top != null && !top.isVisited()) {
83.             neighbours.add(top);
84.         }
85.         if (right != null && !right.isVisited()) {
86.             neighbours.add(right);
87.         }
88.         if (left != null && !left.isVisited()) {
89.             neighbours.add(left);
90.         }
91.     } else if (center.getRow() != 0 && center.getRow() != rows - 1 && center.getCo
    l() != 0 && center.getCol() != cols - 1) {
92.         Cell top = grid[index(center.getCol(), ((center.getRow()) - 1))];
93.         Cell right = grid[index(((center.getCol()) + 1), center.getRow())];
94.         Cell bottom = grid[index(center.getCol(), ((center.getRow()) + 1))];
95.         Cell left = grid[index(((center.getCol()) - 1), center.getRow())];
96.         if (!top.isVisited()) {
97.             neighbours.add(top);
98.         }
99.         if (!right.isVisited()) {
100.                neighbours.add(right);
101.            }
102.            if (!bottom.isVisited()) {
103.                neighbours.add(bottom);
104.            }
105.            if (!left.isVisited()) {
106.                neighbours.add(left);
107.            }
108.        }
109.        if (neighbours.size() > 0) {
```

```
110.                int rnd = (int)((Math.random() * neighbours.size()));
111.                return neighbours.get(rnd);
112.            } else {
113.                return null;
114.            }
115.        }
```

## 4.4 Rekursiv Division Algorithmus

Der Rekursive Division Algorithmus beginnt mit einem leeren Feld. Dieses wird solange zweigeteilt bis die einzelnen Zellen die festgelegte Größe erreicht haben. Als Pseudocode wurde aus der Quelle http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm folgendes verwendet:

1. Begin with an empty field.
2. Bisect the field with a wall, either horizontally or vertically. Add a single passage through the wall.
3. Repeat step #2 with the areas on either side of the wall.
4. Continue, recursively, until the maze reaches the desired resolution.

Der Pseudocode wurde folgendermaßen Umgesetzt.

```
1.  private void divider(int colstart, int colend, int rowstart, int rowend) {
2.      int tempcols = colend - colstart; //ermitteln der Breite des Feldes
3.      int temprows = rowend - rowstart; // ermitteln der höhe des Feldes
4.      int divideori = divideOrientation(tempcols, temprows); //festlegen ob horizont
    al oder vertikal geteilt wird
5.      if (divideori == 1) {
6.          int rndrow = rndNumberRange(rowstart + 1, rowend - 1);
7.          int rndcolwindow = rndNumberRange(colstart, colend - 1);
8.          for (Cell c: grid) {
9.              if (c.getRow() == rndrow) {
10.                 if (c.getCol() != rndcolwindow) {
11.                     if (c.getCol() >= colstart && c.getCol() <= colend) {
12.                         c.setTopWall(true);
13.                         grid[index(c.getCol(), (c.getRow() - 1))].setBottomWall(tr
    ue);
14.                     }
15.                 }
16.             }
17.         }
18.         if ((rndrow - rowstart) > 2 && colend - colstart > 2) {
19.             divider(colstart, colend, rowstart, rndrow);
20.         }
21.         if ((rowend - rndrow > 2 && colend - colstart > 2)) {
22.             divider(colstart, colend, rndrow, rowend);
```

```
        }
    } else {
        int rndcol = rndNumberRange(colstart + 1, colend - 1);
        int rndrowwindow = rndNumberRange(rowstart, rowend - 1);
        for (Cell c: grid) {
            if (c.getCol() == rndcol) {
                if (c.getRow() != rndrowwindow) {
                    if (c.getRow() >= rowstart && c.getRow() <= rowend) {
                        c.setLeftWall(true);
                        grid[index((c.getCol() - 1), c.getRow())].setRightWall(tru
e);
                    }
                }
            }
        }
        if ((rndcol - colstart) > 2 && rowend - rowstart > 2) {
            divider(colstart, rndcol, rowstart, rowend);
        }
        if ((colend - rndcol > 2 && rowend - rowstart > 2)) {
            divider(rndcol, colend, rowstart, rowend);
        }
    }
}
```

Mit Hilfe der „divideOrientation“ Methode wurde bestimmt wie das vorliegende Feld zu teilen. Horizontal oder vertikal wurden zufällig bestimmt, wenn das Feld Quadratisch war, hatte das Feld eine höhere Höhe als breite wurde horizontal geteilt und wenn es breiter als hoch war wurde es vertikal geteilt dies sollte zu einem besseren Aussehen des Labyrinths führen.

```
private int divideOrientation(int width, int height) {
    if (width < height) {
        return 1; //horizontal teilen
    } else if (height < width) {
        return 2; //vertikal teilen
    } else { //Beide gleich zufällige auswahl
        int decider = rndNumberRange(1, 2);
        if (decider <= 1) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

Die „rndNumberRange“ Methode liefert einen zufallswert zwischen einem übergebenen minimal und maximal Wert. Java hat keine eigene Standard Funktion um einen Zufallswert aus einem übergebenen Bereich zu bestimmen. Diese Methode wurde von der aus der Quelle

angegeben Seite https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java übernommen.

```
1.  private int rndNumberRange(int min, int max) {
2.      if (min >= max) {
3.          throw new IllegalArgumentException("Minimalwert ist größer als Maximalwert");
4.      }
5.      return ThreadLocalRandom.current().nextInt(min, max + 1); //Max+1 damit der Maximalwert
     inclusive ist
6.  }
```

## 4.5 A-Stern Algorithmus

Die „astarAlgorithm“ Methode bekommt die Start Zelle und Ziel Zelle übergeben. Von dem Start punkt aus werden von den Umliegenden erreichbaren Zellen die Kosten um zum Ziel zu gelangen ermittelt. Die Zelle mit den geringsten Kosten wird als nächste Zelle gewählt und die Ursprüngliche Zelle wird auf ein Stack gesetzt. Dies wird solange weitergeführt bis die nächste gleich der Ziel Zelle ist oder der Stack leer ist und somit keine Pfad vom Start zum Ziel Punkt existiert.

```
1.  private void astarAlgorithm(Cell start, Cell target) {
2.      Cell next = null;
3.      next = nextField(start);
4.      start.setPath(true);
5.      path.push(start);
6.      while (!path.isEmpty()) {
7.          path.push(next);
8.          next = nextField(next);
9.          if (next == target) {
10.             solverResult.setText("Pfad gefunden!");
11.             break;
12.         }
13.     }
14.     if (next == null) {
15.         solverResult.setText("Kein Pfad gefunden!");
16.         return;
17.     }
18.     gc.setFill(Color.GOLD);
19.     for (int i = 0; i < grid.length; i++) {
20.         if (grid[i].isPath() && !grid[i].isDeadend()) {
21.             gc.fillOval(snap((grid[i].getCol() * cellsize) + ((cellsize / 2) / 2))
    , snap((grid[i].getRow() * cellsize) + ((cellsize / 2) / 2)), cellsize / 2, cellsi
    ze / 2);
22.         }
23.     }
24. }
```

Die „nextField“ Methode suchte um ein übergebenes Feld die Zelle mit dem geringsten Kosten um zum Ziel zu kommen. Konnte keine Zelle gefunden Werden wurde der Pfad solange zurück verfolgt bis ein weiteres fortführen des Algorithmus möglich war oder es eindeutig bestimmt werden konnte das kein Weg zum Zielpunkt existiert.

```
1.  private Cell nextField(Cell center) {
2.      ArrayList < Cell > nextfields = ne
    w ArrayList < > ();
3.      boolean walls[] = center.getwalls(
    );
4.      for (int i = 0; i < walls.length;
    i++) {
5.          if (walls[i] == false) {
6.              if (i == 0 && grid[index(c
    enter.getCol(), center.getRow() - 1)].
    getParent() == null) {
7.                  nextfields.add(grid[in
    dex(center.getCol(), center.getRow() -
     1)]);
8.              }
9.              if (i == 1 && grid[index(c
    enter.getCol() + 1, center.getRow())].
    getParent() == null) {
10.                 nextfields.add(grid[in
    dex(center.getCol() + 1, center.getRow
    ())]);
11.             }
12.             if (i == 2 && grid[index(c
    enter.getCol(), center.getRow() + 1)].
    getParent() == null) {
13.                 nextfields.add(grid[in
    dex(center.getCol(), center.getRow() +
     1)]);
14.             }
15.             if (i == 3 && grid[index(c
    enter.getCol() - 1, center.getRow())].
    getParent() == null) {
16.                 nextfields.add(grid[in
    dex(center.getCol() - 1, center.getRow
    ())]);
17.             }
18.         }
19.     }
20.     if (nextfields.isEmpty()) {
21.         center.setPath(false);
22.         center.setDeadend(true);
23.         Cell alt;
24.         alt = path.pop();
25.         center = alt;
26.         walls = alt.getwalls();
27.         while (nextfields.isEmpty()) {
28.             for (int i = 0; i < walls.
    length; i++) {
29.                 if (walls[i] == false)
     {
30.                     if (i == 0 && grid
    [index(alt.getCol(), alt.getRow() - 1)
    ].getParent() == null) {
31.                         nextfields.add
    (grid[index(alt.getCol(), alt.getRow()
     - 1)]);
32.                     }
33.                     if (i == 1 && grid
    [index(alt.getCol() + 1, alt.getRow())
    ].getParent() == null) {
34.                         nextfields.add
    (grid[index(center.getCol() + 1, alt.g
    etRow())]);
35.                     }
36.                     if (i == 2 && grid
    [index(alt.getCol(), alt.getRow() + 1)
    ].getParent() == null) {
37.                         nextfields.add
    (grid[index(alt.getCol(), alt.getRow()
     + 1)]);
38.                     }
39.                     if (i == 3 && grid
    [index(alt.getCol() - 1, alt.getRow())
    ].getParent() == null) {
40.                         nextfields.add
    (grid[index(alt.getCol() - 1, alt.getR
    ow())]);
41.                     }
42.                 }
43.             }
44.             if (nextfields.isEmpty())
    {
45.                 alt.setDeadend(true);
46.                 alt.setPath(false);
47.                 if (!path.isEmpty()) {
48.                     alt = path.pop();
49.                     walls = alt.getwal
    ls();
50.                     center = alt;
51.                 } else {
52.                     return null;
53.                 }
54.             }
55.         }
56.     }
```

```
57.     for (int i = 0; i < nextfields.siz
    e(); i++) {
58.         nextfields.get(i).setCost(manh
    attanHeuristik(nextfields.get(i), targ
    et));
59.     }
60.     int cost = Integer.MAX_VALUE;
61.     int lowestCostItem = 0;
62.     for (int i = 0; i < nextfields.siz
    e(); i++) {
63.         if (nextfields.get(i).getCost(
    ) < cost) {
64.             cost = nextfields.get(i).g
    etCost();
65.             lowestCostItem = i;
66.         }
67.     }
68.     nextfields.get(lowestCostItem).set
    Path(true);
69.     nextfields.get(lowestCostItem).set
    Parent(center);
70.     gc.setFill(Color.SLATEGREY);
71.     gc.fillOval(snap((nextfields.get(l
    owestCostItem).getCol() * cellsize) +
    ((cellsize / 2) / 2)), snap((nextfield
    s.get(lowestCostItem).getRow() * cells
    ize) + ((cellsize / 2) / 2)), cellsize
     / 2, cellsize / 2);
72.     return nextfields.get(lowestCostIt
    em);
73. }
```

Mittels der „manhattanHeuristik“ werden die Kosten der umliegenden Zellen zum Ziel berechnet.

```
1.  private int manhattanHeuristik(Cell current, Cell target) {
2.      return (((Math.abs(current.getCol() - target.getCol()) + Math.abs(current.getR
    ow() - target.getRow())) * gcost) + gcost);
3.  }
```

## 4.6 Weiter Funktionen

Die „initialize“ Methode war notwendig damit, wenn das Programm gestartet wurde die beiden Canvas Elemente sichtbar waren, ohne die Methode wären die beiden Elemente vor dem ersten Ausführen der Generierungsfunktion nicht sichtbar gewesen.

```
1.  @Override
2.  public void initialize(URL arg0, ResourceBundle arg1) {
3.      gc = canvasLabGenerator.getGraphicsContext2D();
4.      gc.setFill(Color.WHITE);
5.      gc.fillRect(snap(0), snap(0), snap(canvasLabGenerator.getWidth()), snap(canvasLabGenera
    tor.getHeight()));
6.      gc.strokeRect(snap(0), snap(0), snap(canvasLabGenerator.getWidth()), snap(canvasLabGene
    rator.getHeight()));
7.      gc = canvasLabSolver.getGraphicsContext2D();
8.      gc.setFill(Color.WHITE);
9.      gc.fillRect(snap(0), snap(0), snap(canvasLabSolver.getWidth()), snap(canvasLabSolver.ge
    tHeight()));
10.     gc.strokeRect(snap(0), snap(0), snap(canvasLabSolver.getWidth()), snap(canvasLabSolver.
    getHeight()));
11.     solver.setDisable(true);
12. }
```

Die „setup“ Methode legt nach anklicken des Generieren Buttons, die Werte für die Höhe und Breite des Labyrinths mittels der eingestellten Zellengröße fest. Wenn der Rekursive Division Algorithmus gewählt wurde werden nur die äußeren Wände der Zellen die sich jeweils am Rand befinden behalten die restlichen werden auf false gesetzt.

```
1.  @FXML
2.  private void setup() {
3.      int width = (int) canvasLabGenerator.getWidth();
4.      int heigth = (int) canvasLabGenerator.getHeight();
5.      this.cellsize = (int) cellsizeSlider.getValue();
6.      solverResult.setText("");
7.      cellsizeCurrent.setText(String.valueOf(((int) cellsizeSlider.getValue())));
8.      rows = (int)(heigth / cellsize);
9.      cols = (int)(width / cellsize);
10.     grid = new Cell[rows * cols]; //größe des grid Arrays
11.     int count = 0; //befülle Cell grid Array
12.     for (int y = 0; y < rows; y++) {
13.         for (int x = 0; x < cols; x++) {
14.             grid[count] = new Cell(x, y);
15.             count++;
16.         }
17.     }
18.     if (recursiveDivision.isSelected()) {
19.         for (Cell c: grid) {
20.             c.setWalls(false, false, false, false);
21.         }
22.         for (Cell c: grid) {
23.             if (c.getRow() == 0) {
24.                 c.setTopWall(true);
```

```
            }
            if (c.getCol() == 0) {
                c.setLeftWall(true);
            }
            if (c.getRow() == rows - 1) {
                c.setBottomWall(true);
            }
            if (c.getCol() == cols - 1) {
                c.setRightWall(true);
            }
        }
    }
}
```

Die „startGeneration“ Methode leert die Canvas und ruft entsprechend der Auswahl für die Generierung den entsprechenden Algorithmus auf. Nach dem Ablauf des Algorithmus wird auf die beiden Canvas das Labyrinth gezeichnet auf der rechten Seite werden zusätzlich der Startpunkt und Endpunkt für das Lösen eingezeichnet.

```
@FXML
private void startGeneration() {
    gc = canvasLabGenerator.getGraphicsContext2D();
    gc.clearRect(snap(0), snap(0), snap(canvasLabGenerator.getWidth()), snap(canvasLabGenerator.getHeight()));
    gc.setFill(Color.WHITE);
    gc.fillRect(snap(0), snap(0), snap(canvasLabGenerator.getWidth()), snap(canvasLabGenerator.getHeight()));
    gc.strokeRect(snap(0), snap(0), snap(canvasLabGenerator.getWidth()), snap(canvasLabGenerator.getHeight()));
    setup();
    if (recursiveBacktracker.isSelected()) {
        backTrackerAlg();
    } else {
        divisionRecursive();
    }
    draw();
    gc = canvasLabSolver.getGraphicsContext2D();
    gc.clearRect(snap(0), snap(0), snap(canvasLabSolver.getWidth()), snap(canvasLabSolver.getHeight()));
    draw();
    gc.setFill(Color.AQUAMARINE);
    gc.fillOval(snap(0 * cellsize), snap(0 * cellsize), cellsize, cellsize);
    gc.setFill(Color.CORAL);
    gc.fillOval(snap((cols - 1) * cellsize), snap((rows - 1) * cellsize), cellsize, cellsize);
    solver.setDisable(false);
}
```

Mittels der „changeSlider“ Methode wird die Anzeige für die Zellgröße verändert dafür wurde ein Listener verwendet welcher auf die Veränderung des Zellgröße Wertes achtet.

```
@FXML
private void changeSlider() {
    cellsizeSlider.valueProperty().addListener((observable, oldValue, newValue) -
> {
        cellsizeCurrent.setText(Integer.toString(newValue.intValue()));
    });
}
```

Die „snap“ Methode war notwendig, da der JavaFX unter Umständen beim Zeichnen von Linien die zwischen zwei Pixeln sind eine Art Doppellinie zeichnet, dies wird behoben indem der zu zeichnenden Achse einen kleinen Offset von .5 Pixel verpasst so wird sicher gestellt das eine scharfe Linie gezeichnet wird.

```
private double snap(double axis) {
    return ((int) axis) + .5;
}
```

Die „draw“ Methode übergibt der „drawGridVisited Methode“ die einzelnen Spalten und Reihen der Zellen sowie die Wände damit die Zellen an die richtige Position gezeichnet werden können.

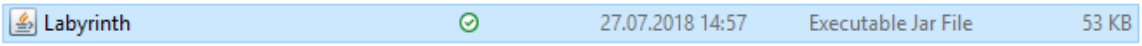
```
private void draw() {
    for (int i = 0; i < grid.length; i++) {
        drawGridVisited(grid[i].getCol(), grid[i].getRow(), grid[i].getwalls(), Co
lor.WHITE);
    }
}
```

```
private void drawGridVisited(double x, double y, boolean walls[], Color color) {
    gc.setFill(color);
    x = x * cellsize;
    y = y * cellsize;
    gc.fillRect(x, y, cellsize, cellsize);
    gc.setStroke(Color.BLACK);
    if (walls[0]) {
        gc.strokeLine(snap(x), snap(y), snap(x + cellsize), snap(y));
    }
    if (walls[1]) {
        gc.strokeLine(snap(x + cellsize), snap(y), snap(x + cellsize), snap(y + ce
llsize));
    }
    if (walls[2]) {
        gc.strokeLine(snap(x), snap(y + cellsize), snap(x + cellsize), snap(y + ce
llsize));
    }
    if (walls[3]) {
        gc.strokeLine(snap(x), snap(y), snap(x), snap(y + cellsize));
    }
}
```

# 5. Benutzerleitpfaden

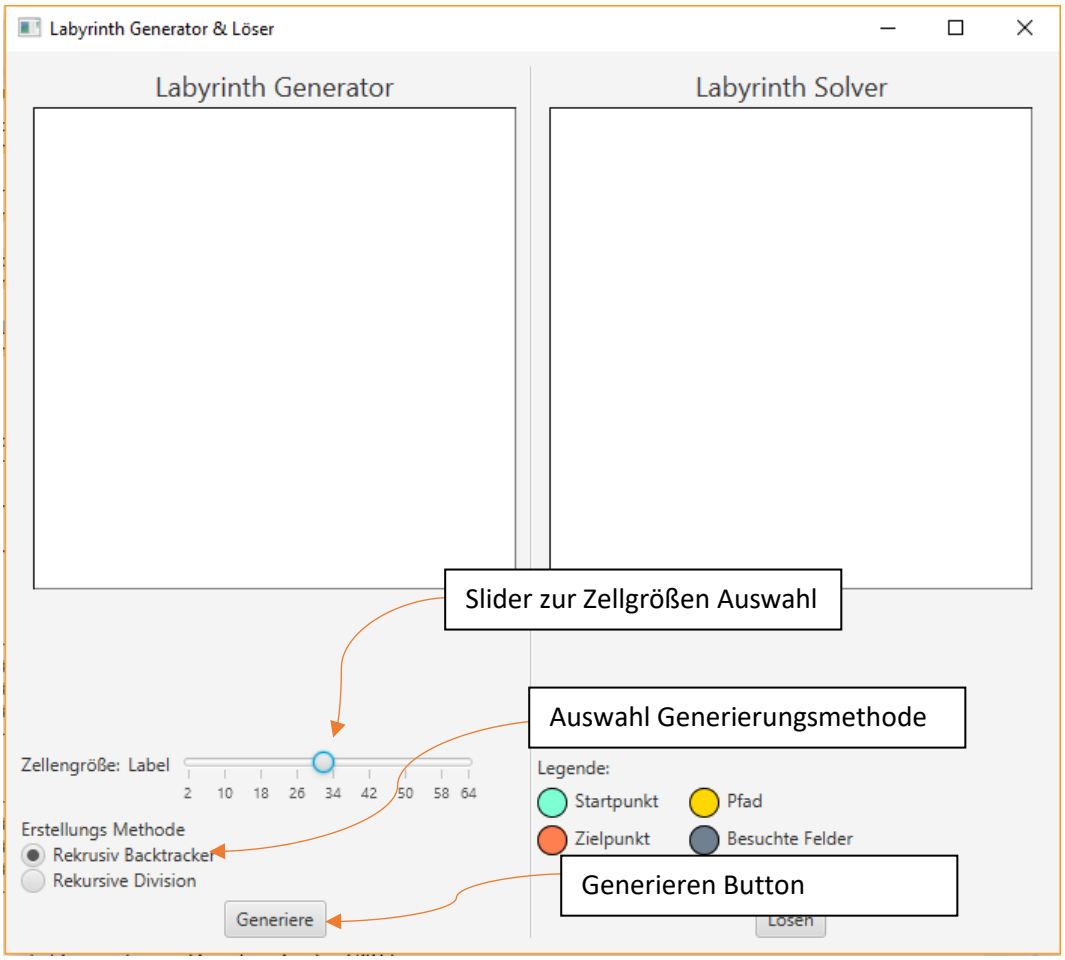
Schritt 1:

- Starten der Anwendung Labyrinth.jar



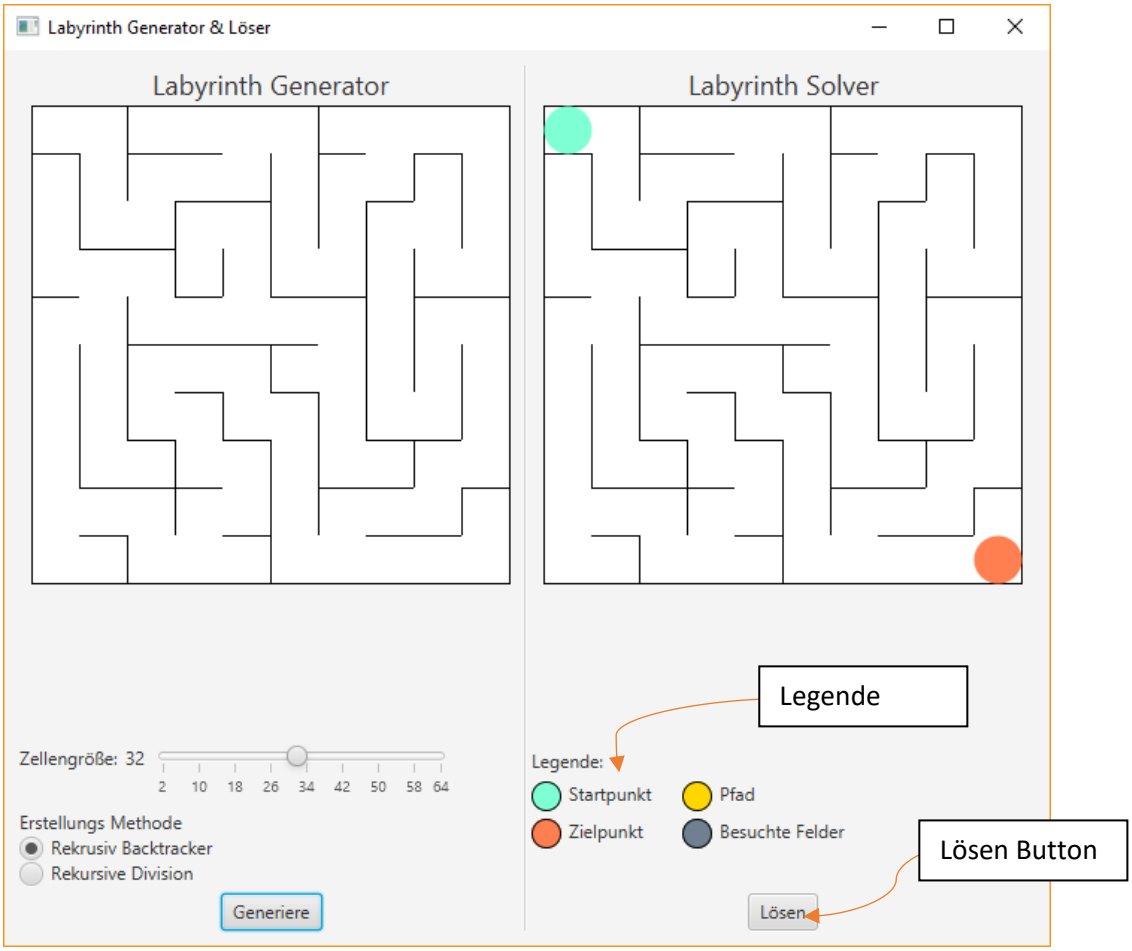

Schritt 2:

- Festlegen der Zellengröße mittels des Sliders
- Auswahl der gewünschten Methode zur Erstellung
- Betätigen des Generieren Buttons um Erstellung des Labyrinths zu Starten



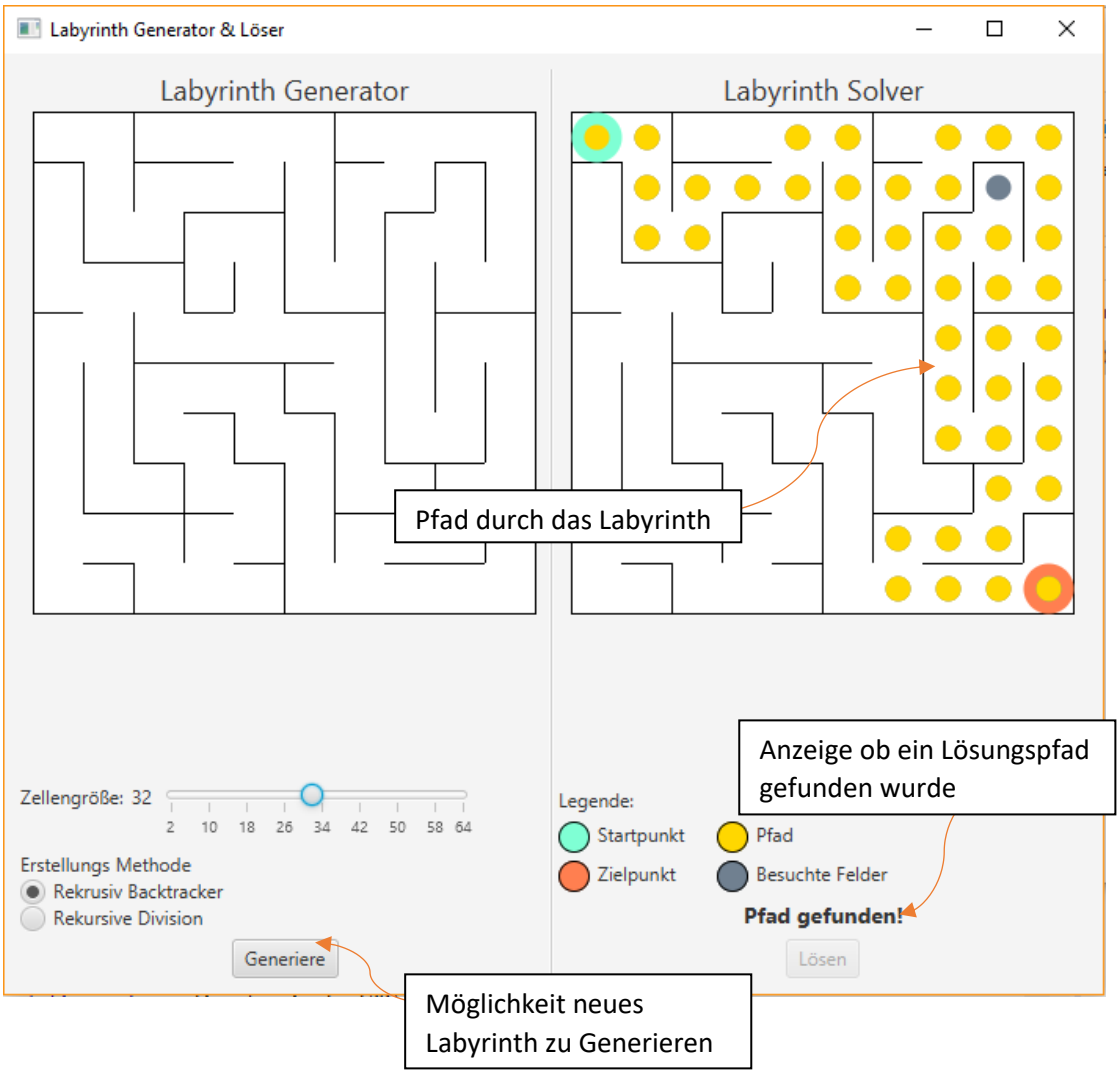
Schritt 3:

- Anklicken des Lösen Buttons



Schritt 4:

- Neue Generierung eines Labyrinths
- Schließen der Anwendung



# 6. Quellen, Hilfsmittel, Entwicklungstools

## 6.1 Quellen

Buck, Jamis: Maze Generation: Recursive Division, unter:
http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm
(abgerufen zuletzt am. 27.07.2018)

Buck, Jamis: Maze Generation: Recursive Backtracking, unter:
http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking
(abgerufen zuletzt am 27.07.2018

Wikipedia, the free encyclopedia: Maze generation algorithm, unter:
https://en.wikipedia.org/wiki/Maze_generation_algorithm (abgerufen zuletzt am 27.07.2018)

Wikipedia, the free encyclopedia: A*-Algorithmus, unter
https://de.wikipedia.org/wiki/A*-Algorithmus (abgerufen zuletzt am 27.07.2018)

Wikipedia, the free encyclopeda: Backtracking, unter
https://de.wikipedia.org/wiki/Backtracking (abgerufen zuletzt am 27.07.2018)

StackOverflow, developer community: How do I generate random integers within a specific range in Java? Lösung von Greg Case answered Dec 12 '08 at 18:25,
unter
https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java (abgerufen zuletzt am 27.07.208)

## 6.2 Hilfsmittel

Erstellung der grafischen Benutzerschnittstelle mit Scene Builder

Scene Builder 8.4.1: https://gluonhq.com/products/scene-builder/

Erstellung erste Version des Klassendiagramms mittels ArgoUML

ArgoUML 0.34: http://argouml.tigris.org/

Erstellung finale Version des Klassendiagramms mittels Dia

Dia 0.97.2: http://dia-installer.de/index.html.de

## 6.3 Entwicklungstools

Programmierung auf Basis von Java 8

Java SE Development Kit 8u181: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Entwicklungsumgebung Eclipse Photon

Eclipse Photon Release (4.8.0):

https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/photon/R/eclipse-java-photon-R-win32-x86_64.zip

# 7. Ausblick / Fazit

Mit der Belegarbeit wurde der Nachweis erbracht das eine selbst gewählte Aufgabe im Bereich der Programmierung Selbstständig gelöst werden kann.

Die Lösung für die 3 Hauptproblemstellungen der Aufgabenstellungen, dass Umsetzen der 3 Algorithmen zur Erstellung und Lösung eines Labyrinthes, konnte selbstständig unter Nutzung von Informationen zum Ablauf dieser Algorithmen, aus den angegebenen Quellen, erarbeitet werden und entsprechend auf das eigene Programm angewandt werden.

Aus Zeitmangel, bedingt teilweise durch schlechtes Zeitmanagement und mehr Belastung in anderen Fächern, konnten die optionalen Zusatzaufgaben nicht fertiggestellt werden. Zudem litt die grafische Darstellung dadurch, dass die Erstellung und Lösung Statisch dargestellt wird, geplant war das dem Algorithmus beim Erstellen und Lösen des Labyrinths zugesehen werden kann.

Das Programm stellt eine gute Grundlage, um vergleiche mit anderen Objektorientierten Programmiersprachen zu erstellen. Da keine zu komplexen Funktionen verwendet werden bzw. Java eigene Funktionen (außer Spezifische Teile der grafischen Darstellung) lässt sich das Vorgehen zur Lösung einfach auf andere Programmiersprachen anwenden. Wodurch ein einarbeiten in eine neue Programmiersprache einfacher gestaltet werden kann, wenn schon eine „Musterlösung“ existiert und so eine Einarbeitung in Eigenheiten einer Programmiersprache besser verinnerlicht werden können.

# 8. Reflexion

## 8.1 Themenwahl

In meiner Belegarbeit im Fach Weiterführende Programmierung im 2. Semester im Studiengang Medieninformatik, habe ich mich mit dem Thema der computergestützten Lösung von Problemen beschäftigt.

Mein Ziel war es ein Programm zu entwickeln, auf Basis der im 1. Semester im Fach Einführung in die Informatik und im 2. Semester im Fach Weiterführende Programmierung gelehrten Programmiersprache Java welches durch Nutzen von den gelehrten Vorgehen der Objektorientierten Programmieren, mittels zweier Algorithmen zum einen dem Rekursiven Backtracking und der Rekursiven Division Methode, dass Erstellen eines Labyrinthes zu realisieren und durch nutzen des A-Stern Algorithmus einen optimalen Lösungsweg darzustellen.

Die grafische Darstellung wurde durch Nutzung der JavaFX Bibliothek realisiert. Des Weiteren sollte mit diesem Beleg gezeigt werden das ein alleiniges erarbeiten und bearbeiten einer selbstgestellten größeren Aufgabe kein allzu großes Hindernis für mich darstellt.

Die Aufgabenstellung habe ich mir gewählt, da Sie einen Großteil der behandelten Themengebiete aus den bisherigen beiden Semestern in dem Java ein Schwerpunkt war behandelt. Zudem bildet das Thema eine gute Grundlage zur Einarbeitung in andere Programmiersprache, da keine Speziellen auf Java zugeschnittene Funktionen genutzt werden sondern Funktionen die eigentlich alle aktuellen Objektorientierten Programmiersprachen nutzen, kann die Aufgabenstellung an die Besonderheiten der jeweiligen Programmiersprache angepasst werden und ein schneller Vergleich zu bekannten Programmiersprachen gezogen werden was ein einarbeiten in eine neue Programmiersprache und deren Besonderheiten erleichtert.

## 8.2 Umsetzung

Der Ablauf dieser Belegarbeit begann damit ein passendes Thema zu finden und dieses entsprechend über die im Seminar und Praktikum genutzte Plattform „moodle“ einzureichen.

Meine erste Idee war es eine Funktion zu erstellen welche auf Basis von „Machine Learning“ anhand von frei Hand eingaben in ein Zeichenfeld (Zahlen oder Buchstaben) entsprechend erkennt welche Zahl oder Buchstaben geschrieben wurde und diese dann ausgibt. Nach genauerer Recherche erschien mir diese Problemstellung allerdings etwas zu Komplex für den gesteckten Zeitrahmen von 85 Stunden.

Während der Recherche zu dem Thema kam ich dann auf die Idee das Thema Computergestützte Lösungen genauer zu verfolgen und entschied mich dann für die bereits in dem Kapitel zuvor beschriebene Aufgabenstellung zu wählen.

Nach der Freigabe der eingereichten Aufgabenstellung, sollte eine erste Version des Klassendiagrammes erstellt werden. Aus den genutzten Materialien zur Recherche ließ sich zwar schnell die Anforderungen an die Algorithmen ableiten aber das Umsetzen in ein wirklich praktikables Klassendiagramm fiel mir dann doch recht schwer, dies liegt sicherlich zum einen an der fehlenden Praxis, zum anderen wäre das sicherlich auch ein Punkt den man in dem Seminar oder Praktikum noch stärker einbauen könnte, es ist zwar oft ein Klassendiagramm gegeben und das lesen selbigen stellt auch kein allzu großes Problem dar, aber das aus einer Aufgabenstellung ableiten welche Teile das Programm haben muss bzw. dieses in einem Klassendiagramm konkretisieren kam meiner Meinung nach zu kurz.

Als nächstes war es erforderlich eine erste Alpha Version meines Programmes zu entwickeln. Die Basis dafür sollte das zuvor erarbeitete Klassendiagramm sein. Während der Umsetzung wurde mir schnell klar, dass das Klassendiagramm in der Ursprünglichen Form nur bedingt umgesetzt wird. Das Auslagern der Funktionen zum Erstellen und Lösen des Labyrinthes wurde spielte beim Erstellen erstmal keine Rolle, es wurde der Großteil des Programmkodes in die GUIController Klasse implementiert. Dies hatte zunächst hauptsächlich den Vorteil das schneller ein brauchbares Ergebnis erzielt werden konnte. Durch das Erstellen einer eigenen Klasse (Cell) für die Organisation konnten einfach alle Relevanten Informationen verwaltet werden, dass Lösen des selben Problems mittels einem Zweidimensionalen Arrays wäre

besonders bei der Verwendung der ganzen Status die eine Zelle haben kann wesentlich umständlicher umzusetzen gewesen. Durch die Nutzung der Cell Klasse hat sich jedoch das Navigieren innerhalb des Grid-Array schwieriger gestaltet, das Finden der Nachbarn hatte eine erhöhte Gefahr eine ArrayOutofBoundsEception hervor zu bringen, da nicht einfach ausgeschlossen werde konnte welche Zelle an welcher Position im Labyrinth ist.

Die grafische Oberfläche habe ich mittels Scene Builder erstellt, durch die Einbindung in Eclipse war die Umsetzung sehr einfach, einzig das genaue Ausrichten von Elementen kann durch die Tatsache erschwert werden, bzw. erfordert ein gewisses Umdenken, da man die einzelnen Objekte in extra Objekte verpacken muss welche dann das Ausrichten der Objekte übernehmen.

Für die Alpaversion hatte ich geplant einen der beiden Algorithmen zum Erstellen und den A-Stern Algorithmen zum Lösen umzusetzen. Der Rekursive Backtracking Algorithmus zum Erstellen des Labyrinthes, lies sich mithilfe der genutzten Quellen ohne größere Probleme umsetzen, lediglich das Erfassen der Nachbarschaft Felder wurde dadurch erschwert das alle Felder in einem eindimensionalen Array gespeichert war, was die Korrekte Erfassung der Nachbarfelder etwas erschwert hat da keine genaue Aussage getroffen werden konnte anhand der Position im Array an welcher Stelle sich das Feld befindet. Mittels des A-Stern Algorithmus habe ich die Lösung für das Labyrinth gestaltet, auch hier war die Umsetzung anhand von Pseudocode bzw. der Beschreibung wie der Algorithmus arbeitet ohne große Probleme möglich.

Als letztes blieb die Fertigstellung des Programmes und das Erstellen der Schriftlichen Ausarbeitung zu der Belegarbeit. Die meisten Probleme bereitete die Rekursive Division Methode, der Algorithmus hat teilweise Wände gezogen die eigentlich im Algorithmus nicht vorgesehen waren was zu allerhand Problemen geführt hat. Leider auch bis zum Ende der Belegarbeit konnte dies nicht komplett behoben werden, dies bedeutet es können Labyrinthe entstehen die eigentlich so nicht möglich sein sollten. Des Weiteren ist zum Ende hin die Möglichkeit weggefallen einen eigenen Start und Ziel Punkt für den A-Stern Algorithmus festzulegen, dies sorgte für mehr Probleme als das es nutzen hatte, da dies in diesem Sinne nicht Teil der ursprünglichen Aufgabenstellung war kann dies als weniger kritisch angesehen

werden. Der Aufwand für die Anfertigung der Schriftlichen Ausarbeitung habe ich am Ende doch etwas Unterschätzt gehabt.

Durch die Probleme mit dem Rekursiven Division Algorithmus und dem höheren als erwartetem Aufwand für die Schriftliche Ausarbeitung hat sich das Fertigstellen doch bis auf den letzten Abgabe Tag herausgezögert. Was sich definitiv in den nächsten Belegarbeiten verbessern muss.

## 8.3 Fazit

Für die Belegarbeit kann ich für mich ein positives Fazit ziehen. Da die Aufgabenstellung größtenteils erfolgreich bearbeitet wurde. Dadurch das die Aufgabenstellung selbst zu wählen war, war die Motivation diese erfolgreich zu bewältigen auch deutlich größer als wenn es eine vorgegebene Aufgabe gestellt wurden wäre. Wenn es auch zu Beginn etwas schwieriger war etwas zu finden, was nicht zu unterfordernd bzw. überfordernd war.

Der größte Problembereich bei dieser Belegarbeitet für mich war klar das Zeitmanagement, besonders durch die umfangreiche Semesterarbeit im Fach „Wissenschaft und Wirtschaft I“, wurde die Belegarbeit des Öfteren hintenangestellt. Dadurch sind besonders die grafische Gestaltung des Programmes sowie die Gestaltung des Quellcodes hinter meinen eigenen Ansprüchen zurückgeblieben auch die vollständige Lösung rum rekursiven Division Algorithmus konnte leider nicht erreicht werden. Dies werde ich in den nächsten Semestern versuchen zu reduzieren/beheben durch eine bessere Planung, vor allem eine bessere Unterteilung der Aufgabenbereiche einer Belegaufgabe und das Erstellen eines groben Ablaufdiagramms.

Rückblickend bin ich mit meiner Themenwahl sehr zufrieden, es war nicht zu schwer aber hatte doch genug Feinheiten bei der Umsetzung, so dass auch keine unter Forderung eingetreten ist. Dadurch hat sich mein Entschluss noch weiter gefestigt die IA (Informatics Applications) Vertiefung des Studiengangs Medieninformatik weiter zu verfolgen. Da besonders im Bereich des Maschinen gestütztem Lernen sich immer weiter Besonders interessante Aufgabengebiete erschließen.

# 9 Eigenständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegeben Literatur und Hilfsmittel angefertigt habe.

<Datum und Unterschrift>