Gruppe P24  
Matthias Ridder, Dominic von Zielinski

Domino

Projektdokumentation

Inhaltsverzeichnis

[1 Pflichtenheft 2](#_Toc423029481)

[Pflichtenheft 2](#_Toc423029482)

[1. Management- und Dokumentationsattribute 2](#_Toc423029483)

[2. Visionen und Ziele 2](#_Toc423029484)

[3. Rahmenbedingungen 4](#_Toc423029485)

[4. Kontext und Überblick 4](#_Toc423029486)

[5. Funktionale Anforderungen 4](#_Toc423029487)

[6. Qualitätsanforderungen 5](#_Toc423029488)

[7. Abnahmekriterien 5](#_Toc423029489)

[8. Quellen 5](#_Toc423029490)

[9. Hinweis zu dieser Vorlage 6](#_Toc423029491)

[9.1 Literaturliste 6](#_Toc423029492)

[2 Projektmanagement 7](#_Toc423029493)

[2.1 Rollen 7](#_Toc423029494)

[3 Projektplanung 7](#_Toc423029495)

[3.1 Zu Beginn 7](#_Toc423029496)

[3.2 Am Ende 7](#_Toc423029497)

[3.3 Erklärung zu den Abweichungen 7](#_Toc423029498)

[4 UML-Diagramme 8](#_Toc423029499)

[4.1 Use-Case 8](#_Toc423029500)

[4.2 Klassendiagramm 1 8](#_Toc423029501)

[4.3 Klassendiagramm 2 9](#_Toc423029502)

[5 Akzeptanztests 10](#_Toc423029503)

[6 Quellcode 11](#_Toc423029504)

# Pflichtenheft

Pflichtenheft

*Domino*

*P24*

*24.06.2015*

# Management- und Dokumentationsattribute

|  |  |
| --- | --- |
| Dokumentationsattribute | |
| Autor | Dominic von Zielinski, Matthias Ridder |
| Eindeutige Teamnummer | P24 |
| Quelle |  |
| Version | 1.0 |
| Bearbeitungsstatus | Vollständig |

# Visionen und Ziele

/PV10/ **Mehrspielermodus**

Standardmäßig spielen entweder zwei menschliche Spieler/ ein menschlicher Spieler und ein Computerspieler gegeneinander.

Die Vision des Mehrspielermodus soll das Ganze auf eine Spieleranzahl von 3/4 bringen. Die Erweiterung der Spieleranzahl gibt einer Gruppe die meist aus mehr als 2 Menschen besteht die Möglichkeit sich gleichzeitig in einer Runde zu messen

/PV20/ **KI**

Die Realisierung einer künstlichen Intelligenz ist eine weitere Vision für unser Programm. Die künstliche Intelligenz soll das Spielerlebnis des Spielers verbessern und ihm das Gefühl geben gegen einen richtigen Spieler zu spielen.

Neben dem üblichen Regelwerk muss die KI folgendes wissen:

1. Welche Steine sie auf der Hand hat
2. Welche Steine sie an welche Steine anlegen kann
3. Wie die Punkte für eine Runde und die Maximalpunktzahl errechnet wird
4. Sie muss die Sonderregeln kennen
5. Sie muss wissen, dass sie nach einer verlorenen Runde ihre restlichen Steine zusammenzählt
6. Sie muss wissen, dass sie Steine vom Talon nehmen muss wenn sie keine passenden Steine auf der Hand hat
7. Sie muss wissen wann sie gewonnen hat
8. Sie muss wissen wer das Spiel beginnt/den ersten Zug machen darf

/PV30/ **Verschiedene Schwierigkeitsmodi**

Zusätzlich zur KI gehört die Implementierung von verschiedenen Schwierigkeitsmodi ebenfalls zu unseren Visionen. Angesetzt ist fürs erste eine feste Schwierigkeitsstufe. Bei Zeiten kann dies auf bis zu 3 (leicht, mittel und schwer) erweitert werden.

Sinn der verschiedenen Schwierigkeitsmodi ist ein abwechslungsreicheres Spielerlebnis. Dem einen fällt es leichter/macht es mehr Spaß gegen einen schwierigen Computergegner zu spielen während der andere lieber auf ein leichtes Schwierigkeitslevel zurückgreifen will, da er noch nicht so erfahren ist.

/PZ10/ **Aufbau/Implementierung der Logik**

Die Implementierung der Logik ist das Herzstück des Spiels. Die Logik beinhaltet die

Korrekte Handhabung der Spielregeln, den in einer korrekten Reihenfolge ablaufenden Spielablauf und alle relevanten Informationen die einen korrekten und vollständig richtigen Spielverlauf beeinflussen.

/PZ20/ **GUI**

Ebenso wichtig wie die Logik ist die GUI. Die GUI bietet dem Spieler eine visuelle Plattform auf der er den Verlauf des Spiels, die Punkte, die Steine und die Züge in visueller Form aufnehmen und verarbeiten kann. Außerdem vereinfacht die GUI dem Spieler das spielen. Er kann sich ganz auf das Spiel konzentrieren und wird nicht von diversen Störfaktoren abgelenkt.

Die GUI soll folgende Punkte beinhalten:

1. Die Darstellung des Spielfelds in Form eines Fensters
2. Die Darstellung der 28 Spielsteine (Eigene Hand, gegnerische Hand, Talon und Spielfeld)
3. Die Darstellung der Punkte (Punktezähler)
4. Die Aufteilung des Spielfelds (z.B. Anzeige von eigenen Steinen getrennt von denen des Gegners) um dem Spieler eine klare Übersicht des Spielfelds zu gewährleisten.

# Rahmenbedingungen

/PR10/ **Anwendungsbereich**

Videospiel

/PR20/ **Zielgruppe**

Menschen zwischen 6-99 Jahren, die Gesellschaftsspiele mögen

/PR30/ **Betriebsbedingungen**

Hardware: PC, Bildschirm mit einer Mindestauflösung von 1600x900

Software: Windows und mindestens Java Version 7

Schnittstellen: Maus

# Kontext und Überblick

/PK10/ **Windows und Java**

Das Spiel wird auf jedem javafähigen Windows-PC, Mac oder Linux-System funktionieren. Optimiert wird es jedoch für Windows-Systeme.

Die eingesetzte Java-Version ist Version 7

/PÜ10/ **Domino**

Programmiert wird das Legespiel Domino. Es wird nach den Regeln des Muggins bzw. All-Fives Spiels gespielt.

# Funktionale Anforderungen

/PF10/ **Intuitive Bedienung mit Maus**

/PF20/ **Leicht zu lernen, schwer zu meistern**

/PF30/ **Flüssiger Spielablauf**

/PF40/ **Fehlerfreies Spiel**

# Qualitätsanforderungen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Systemqualität | Sehr gut | Gut | Normal | Nicht relevant |
| Funktionalität |  | X |  |  |
| Zuverlässigkeit | X |  |  |  |
| Benutzbarkeit |  | X |  |  |
| Effizienz |  |  |  | X |
| Wartbarkeit |  |  | X |  |
| Portabilität |  |  |  | X |

Tabelle 1: Qualitätsanforderungen

/PQF10/ **Funktionalität**

Das Spiel muss flüssig funktionieren, ohne dass es abstürzt.

/PQZ10/ **Zuverlässigkeit**

Zuverlässigkeit der Logik. Diese darf keine falschen Spielzüge zulassen und muss die Punkte immer richtig berechnen.

/PQB10/ **Benutzbarkeit**

Intuitive Benutzbarkeit und leicht verständliche GUI.

/PQE10/ **Effizienz**

Da es sich um ein kleines Spiel handelt, ist die Effizienz des Programms nicht relevant.

/PQW10/ **Wartbarkeit**

Fehlerhafter Code (Bugs) sollen im Nachhinein beseitigt werden können. Die Funktionalität des Quellcodes soll jedoch im Vordergrund stehen.

Der Quellcode kann im Nachhinein übersichtlicher gestaltet werden.

/PQP10/ **Portabilität**

Die Portabilität des Programms ist in keiner Hinsicht relevant.

# Abnahmekriterien

* Funktionalität der Spielmechanik
* Keine deutlich sichtbaren Fehler
* Intuitive Bedienbarkeit

# Quellen

* <http://docs.oracle.com/javase/7/docs/api/overview-summary.html>
* <https://docs.oracle.com/javase/tutorial/2d/images/index.html>

# Hinweis zu dieser Vorlage

Die Vorlage für dieses Pflichtenheft wurde Balzert (2009), S. 492 ff. entnommen.

## Literaturliste

Balzert, Helmut (2009). Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering. 3. Auflage. Heidelberg: Spektrum, Seite 492 ff.

# Projektmanagement

## Rollen

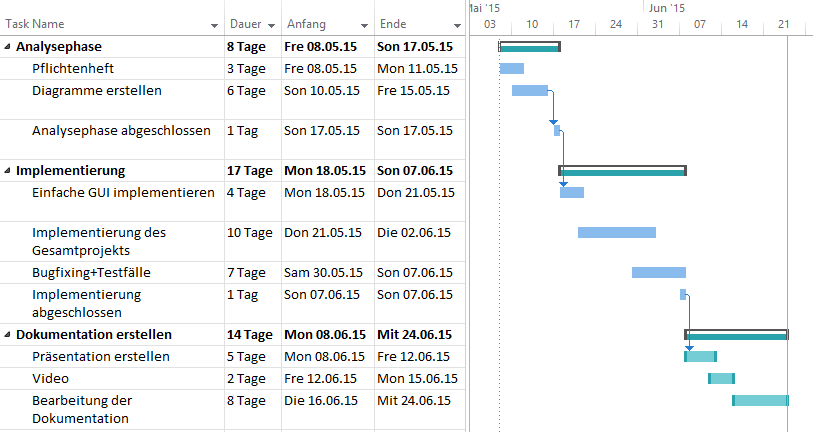
Die Teammitglieder übernahmen folgende Rollen:

Matthias Ridder: Dokumentation, Softwareentwicklung, Tester

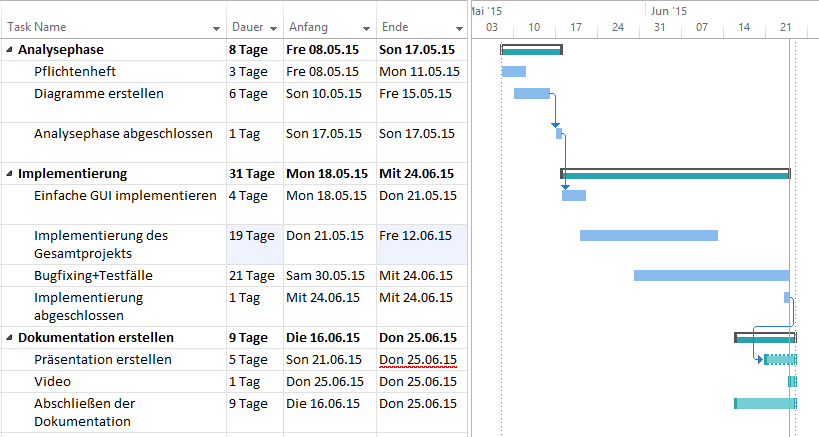
Dominic von Zielinski: Dokumentation, Softwareentwicklung, Tester

# Projektplanung

## Zu Beginn



## Am Ende

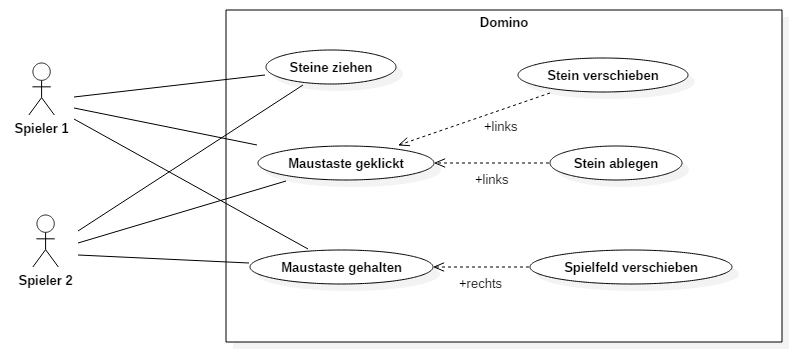


## Erklärung zu den Abweichungen

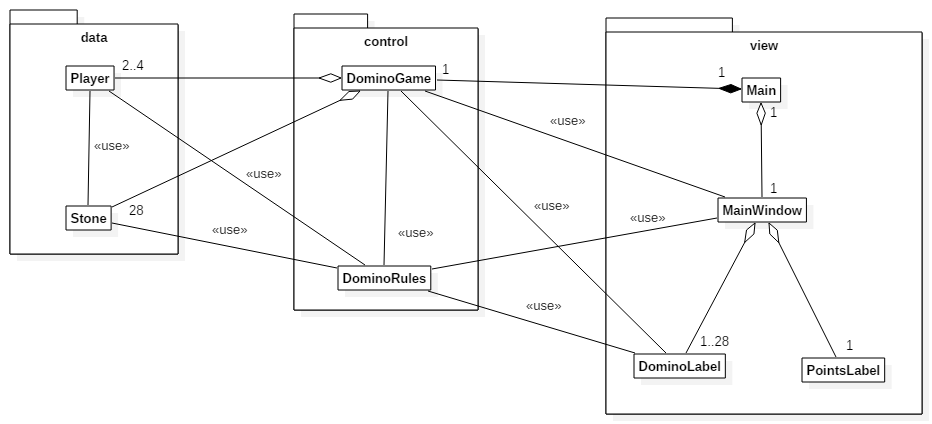
Die Abweichungen im Projektplan hingen von mehreren Faktoren ab. Ein Faktor war, dass zeitgleich zum Informatikprojekt noch andere Projekte gemacht werden mussten. Außerdem haben wir uns generell zu viel vorgenommen. Zusätzlich dazu, haben wir uns gleichzeitig immer mehr Features ausgedacht, die wir implementieren wollten. Je komplexer das Programm wurde, desto mehr kleine Fehler mit großen Auswirkungen sind aufgetaucht. Dadurch wurde das Bugfixing zu einer sehr zeitraubenden Aufgabe. Dadurch kam es zu Überschneidungen der Implementierung und der Erstellung der Dokumentation, wodurch wir erst zu spät mit der Vervollständigung der Dokumentation und dem Video beginnen konnten. Die Abgabe staute dadurch leider über den Abgabetermin hinaus.

# UML-Diagramme

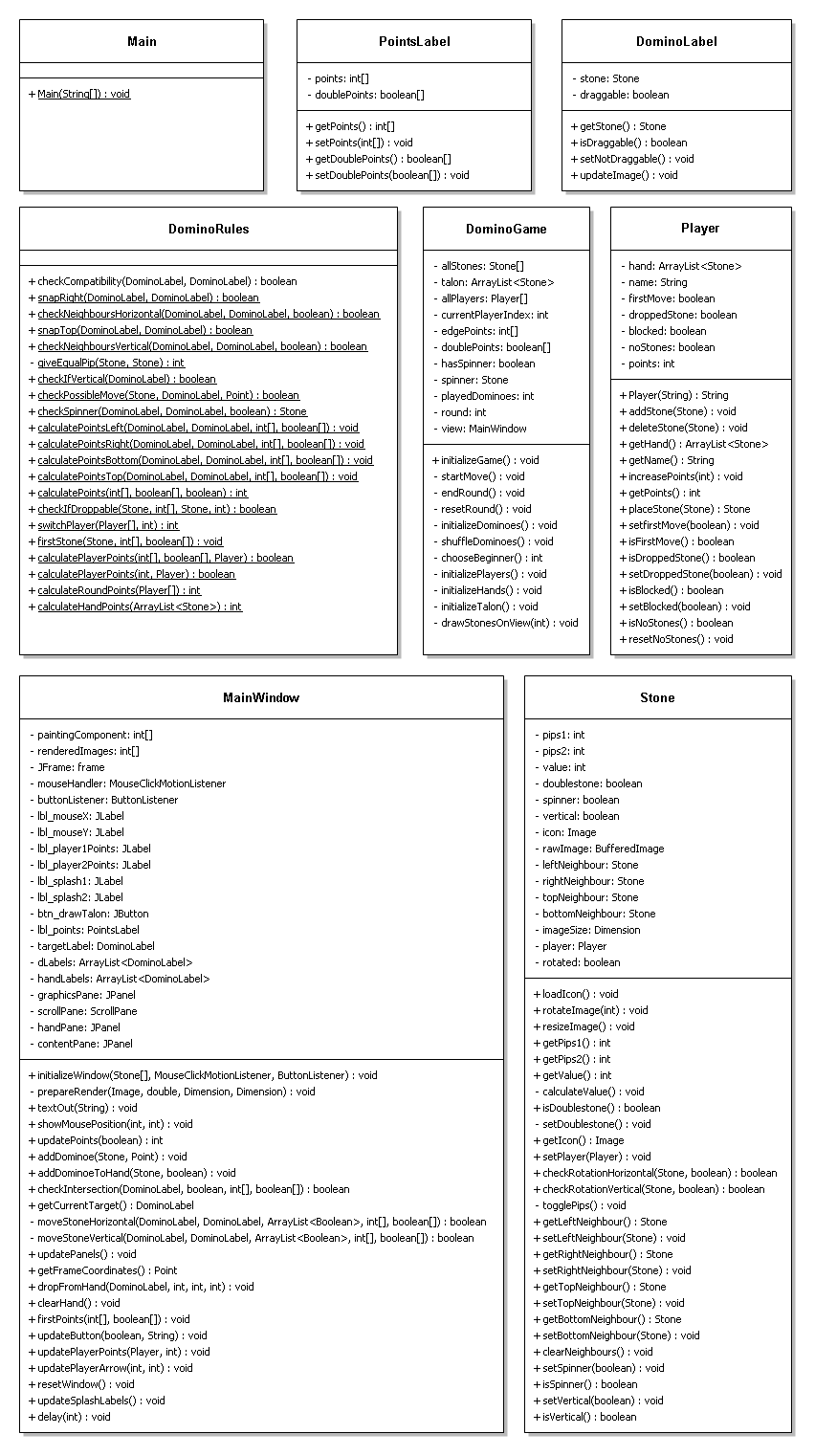
## Use-Case



## Klassendiagramm 1



## Klassendiagramm 2



# Akzeptanztests

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Test 6 | Test 7 | Test 8 | Test 9 | Test 10 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

*(Legende: 1 = bestanden, 0 = nicht bestanden)*

Test 1: Das Programm startet und erzeugt 28 Dominosteine

Test 2: Die Steine werden gemischt

Test 3: Jeder Spieler erhält 7 Steine

Test 4: Die Steine werden auf der Hand richtig gedreht dargestellt

Test 5: Die Drehung der Steine beim Anlegen ist korrekt

Test 6: Die Kantenpunkte werden korrekt berechnet (im Spiel blau hinterlegt)

Test 7: Die Spieler werden nach jedem abgeschlossenen Zug gewechselt

Test 8: Die Spielerpunkte werden richtig berechnet

Test 9: Das Spielfeld lässt sich wie gewünscht verschieben (ohne sprunghaften Versatz)

Test 10: Das Spiel zeigt nach Erreichen der Zielpunktzahl den Endscreen

# Quellcode

package control;

import java.awt.Cursor;

import java.awt.Dimension;

import java.awt.MouseInfo;

import java.awt.Point;

import java.awt.PointerInfo;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.awt.event.MouseMotionListener;

import java.util.ArrayList;

import java.util.Collections;

import javax.swing.JPanel;

import view.DominoLabel;

import view.MainWindow;

import data.Player;

import data.Stone;

/\*\*

\* Die Hauptklasse des Spiels

\*/

public class DominoGame

{

/\*\* Haelt alle Steine im Spiel \*/

private Stone[] allStones = new Stone[28];

/\*\* Der Pool aus dem die Spieler Steine ziehen\*/

private ArrayList<Stone> talon = new ArrayList<Stone>();

/\*\* Haelt alle Spieler\*/

private Player[] allPlayers;

/\*\* Der Index des aktuellen Spielers\*/

private int currentPlayerIndex;

/\*\* Ein Array, in dem die Punkte der offenen Enden gespeichert werden. <br>

\* [0]: links<br>[1]: rechts<br>[2]: oben<br>[3]: unten \*/

private int[] edgePoints = new int[4];

/\*\* Ein Array, in dem gespeichert wird, ob eine offene Kante doppelte Punkte bringt <br>

\* [0]: links<br>[1]: rechts<br>[2]: oben<br>[3]: unten \*/

private boolean[] doublePoints = new boolean[4];

/\*\* Gibt an, ob ein Spinner auf dem Feld liegt \*/

private boolean hasSpinner = false;

/\*\* Speichert den Spinner \*/

private Stone spinner = null;

/\*\* Zaehlt, wieviele Steine bereits gelegt worden sind \*/

private int playedDominoes = 0;

/\*\* Gibt die aktuelle Runde an \*/

private int round = 1;

/\*\* Das Hauptfenster, in dem das Spiel dargestellt wird \*/

private MainWindow view;

/\*\* Der Listener fuer alle Mauseingaben \*/

private MouseClickMotionListener mouseHandler = new MouseClickMotionListener();

/\*\* Die Anzahl der benoetigten Punkte zum gewinnen des Spiels \*/

private final int winningPoints = 250;

/\*\*

\* Erzeugt das Spiel

\* @param view - Die View

\*/

public DominoGame(MainWindow view, int numPlayers)

{

allPlayers = new Player[numPlayers];

// Alle Kantenpunkte werden zu Anfang mit 7 initialisiert,

// denn das ist fuer die Funktionalität des Punktesystems wichtig

for (int i = 0; i < edgePoints.length; i++)

edgePoints[i] = 7;

// das Hauptfenster wird im Spiel gespeichert

this.view = view;

initializeGame();

}

/\*\*

\* Initialisiert das Spiel

\*/

public void initializeGame()

{

initializeDominoes();

initializePlayers();

initializeHands();

initializeTalon();

// Initialisierung und anzeigen des Hauptfensters

view.initializeWindow(allStones, mouseHandler, new ButtonListener());

// Auswahl des Spielers der beginnen soll

currentPlayerIndex = chooseBeginner();

view.showStartSplash();

showRoundInfo("", "", 80f, 0);

startMove();

}

/\*\*

\* Zeigt die aktuelle Rundenzahl und den beginnenden Spieler

\* oder einen beliebigen Text an und blendet ihn nach

\* einer bestimmten Zeit wieder aus

\* @param text1 - der Text der im oberen Label stehen soll

\* @param text2 - der Text der im unteren Label stehen soll

\* @param textSize - die Textgroesse

\* @param delay - Verzoegerung in Millisekunden nach der der Text angezeigt werden soll

\*/

private void showRoundInfo(String text1, String text2, float textSize, int delay)

{

String labelText1, labelText2;

if (text1 == "" && text2 == "")

{

labelText1 = "Runde " + round;

labelText2 = "Spieler " + (currentPlayerIndex+1) + " beginnt!";

}

else

{

labelText1 = text1;

labelText2 = text2;

}

view.showGameInfo(labelText1, labelText2, textSize, delay, true);

}

/\*\*

\* Initialisierung des Spielzuges

\*/

private void startMove()

{

Player player = allPlayers[currentPlayerIndex];

boolean blocked = false;

int failCounter = 0;

boolean showDominoes = false;

boolean endRound = false;

player.setDroppedStone(false);

view.textOut("Runde " + round + "!");

view.textOut("gespielte Steine: " + playedDominoes);

view.updatePlayerArrow(currentPlayerIndex, playedDominoes);

player.setBlocked(false);

// Schleife die prueft, ob der Spieler einen legbaren Stein hat

for (Stone s: player.getHand())

{

if (!DominoRules.checkIfDroppable(s, edgePoints, spinner, playedDominoes))

failCounter++;

}

// Wenn der Spieler keinen Stein legen kann ist er blockiert

if (failCounter == player.getHand().size())

{

blocked = true;

player.setBlocked(true);

}

// Wenn beide Spieler blockiert sind endet die Runde

if (allPlayers[0].isBlocked() && allPlayers[1].isBlocked())

endRound = true;

// Wenn der Talon nicht leer ist, aber der Spieler geblockt ist darf er einen Stein ziehen

if (!talon.isEmpty() && blocked == true)

view.updateButton(true, "");

else

view.updateButton(false, "");

// wenn der Talon leer ist oder der Spieler trotzdem legen kann...

if (!talon.isEmpty() || (talon.isEmpty() && blocked == false))

showDominoes = true; // ...werden die Steine im Fenster angezeigt

// wenn der Talon leer ist und der Spieler nicht legen kann...

else if (talon.isEmpty() && blocked == true)

showDominoes = false; // ...werden die Steine nicht angezeigt

System.err.println("Stand showDominoes: " + showDominoes);

// wenn die Runde nicht blockiert ist

if (!endRound)

makeMove(showDominoes, player);

else

endRound();

}

/\*\*

\* Es werden entweder die Steine des aktuellen Spielers angezeigt

\* oder wenn der Spieler nicht legen ist der naechste Spieler an der Reihe

\* @param showDominoes - wenn true sollen die Steine angezeigt werden<br>

\* wenn false ist der naechste Spieler an der Reihe

\* @param player - der aktuelle Spieler

\*/

private void makeMove(boolean showDominoes, Player player)

{

if (showDominoes)

{

view.textOut("aktueller Spieler: " + player.getName());

drawStonesOnView(currentPlayerIndex);

}

else

{

int otherPlayer = DominoRules.switchPlayer(allPlayers, currentPlayerIndex);

String text1 = "Spieler " + (currentPlayerIndex+1) + " kann nicht legen";

String text2 = "Spieler " + (otherPlayer+1) + " ist am Zug";

view.showGameInfo(text1, text2, 40f, 0, true);

currentPlayerIndex = DominoRules.switchPlayer(allPlayers, currentPlayerIndex);

allPlayers[currentPlayerIndex].setDroppedStone(false);

startMove();

}

}

/\*\*

\* Beendet eine Runde

\*/

private void endRound()

{

// Berechnung des Gewinners und dessen Punkte

int winnerPoints = DominoRules.calculateRoundPoints(allPlayers, true);

int winner = DominoRules.calculateRoundPoints(allPlayers, false);

String winnerString = "Spieler " + (winner+1) + " gewinnt die Runde";

String pointsString = "und erhält " + winnerPoints + " Punkte";

// Anzeige des Gewinners und seiner Punkte auf dem Bildschirm

view.showGameInfo(winnerString, pointsString, 40f, 0, true);

view.updatePlayerPoints(allPlayers[winner], winner);

// wenn der Rundengewinner weniger Punkte hat als die zum

// Spielgewinn benoetigten, wird die naechste Runde gestartet

if (allPlayers[winner].getPoints() < winningPoints)

{

resetRound();

initializeDominoes();

initializeHands();

initializeTalon();

// Spieler auswaehlen, der die naechste Runde beginnt

currentPlayerIndex = chooseBeginner();

// Runde um 1 erhoehen

round++;

startMove();

}

// sonst steht der Gewinner des Spiels fest und das Spiel endet

else

endGame(winner+1);

}

/\*\*

\* Das Spiel ist beendet und der Gewinner wird bekannt gegeben

\* @param winner - Index des Gewinners + 1

\*/

private void endGame(int winner)

{

// Das Fenster zuruecksetzen, damit der Text besser lesbar ist

view.resetWindow();

String text1 = "Herzlichen Glückwunsch Spieler " + winner;

String text2 = "Sie haben gewonnen!";

view.updateButton(false, "Spiel beendet");

// Bekanntgabe des Gewinners

view.showGameInfo(text1, text2, 60f, 0, true);

view.showGameInfo("Toll gemacht", "Vielen Dank für's Spielen!", 80f, 2100, false);

}

/\*\*

\* Setzt alle relevanten Instanzvariablen zurueck, bevor die naechste Runde

\* gestartet wird

\*/

private void resetRound()

{

for (int i = 0; i < edgePoints.length; i++)

{

edgePoints[i] = 7;

doublePoints[i] = false;

}

for (Player p: allPlayers)

{

p.getHand().clear();

p.setBlocked(false);

p.setfirstMove(true);

p.setDroppedStone(false);

p.resetNoStones();

}

hasSpinner = false;

spinner = null;

playedDominoes = 0;

mouseHandler.panelOffsetX = 0;

view.resetWindow();

}

/\*\*

\* Erstellt die 28 Dominosteine

\*/

private void initializeDominoes()

{

int w = 100;

int h= 50;

int current = 0;

// Aeussere Schleife (linke Augenzahl)

for (int x = 0; x <= 6; x++)

{ // Innere Schleife (rechte Augenzahl)

for (int y = 0; y <= 6; y++)

{

/\* wenn die innere Schleife neu beginnt, wird

deren Schleifenzaehler auf den aktuellen

Wert des aeusseren Zaehlers gesetzt \*/

if (y == 0)

y = x;

// Erzeugt den Dominostein und legt Augenzahlen und Groesse fest

allStones[current] = new Stone(x, y, new Dimension(w, h));

// Laedt das passende Bild

allStones[current].loadIcon();

// Dieser Zaehler zaehlt durch die Schleifen bedingt von 0 bis 27

current++;

}

}

// Konsolenausgabe bevor gemischt wurde

for (Stone n : allStones)

{

System.out.println(n.getPips1() + "|" + n.getPips2() + " = " + n.getValue() + " | DS: " + n.isDoublestone());

}

shuffleDominoes();

// Ausgabe nachdem gemischt wurde

System.out.println("Gemischt!:");

for (Stone n : allStones)

{

System.out.println(n.getPips1() + "|" + n.getPips2() + " = " + n.getValue() + " | DS: " + n.isDoublestone());

}

}

/\*\*

\* Mischt die Dominosteine

\*/

private void shuffleDominoes()

{

// Anlegen einer ArrayList, die spaeter gemischt wird

ArrayList<Stone> stoneShuffler = new ArrayList<Stone>();

// Laedt alle Dominosteine in die ArrayList

for(Stone s : allStones)

{

stoneShuffler.add(s);

}

// Mischen der Steine mithilfe der shuffle Methode fuer ArrayLists

Collections.shuffle(stoneShuffler);

// Zurueckschieben der Steine in das Array

int z = 0;

for (Stone s : stoneShuffler)

{

allStones[z] = s;

z++;

}

}

/\*\*

\* Auswahl des Spielers, der die Runde beginnt

\* Der Spieler mit dem hoechsten Doppelstein beginnt

\* Hat kein Spieler einen Doppelstein, beginnt der

\* Spieler der ansonsten den hoechstwertigsten Stein hat

\* @return - Der Index des Beginners

\*/

private int chooseBeginner()

{

// Anlegen einer Variable fuer den hoechsten Stein

Stone highest = null;

// Schleife durch alle Spieler

for (Player p: allPlayers)

{

// Schleife durch alle Steine auf der Hand

for (Stone s: p.getHand())

{

// wenn der Stein ein Doppelstein ist

if (s.isDoublestone() == true)

{

// wenn es bereits einen hoechsten Stein gibt

if (highest != null)

{

/\* wird ersetzt, wenn sein Wert hoeher ist

\* als der Wert des aktuell hoechsten Steins \*/

if (s.getValue() > highest.getValue())

highest = s;

}

// wenn noch kein Stein der hoechstwertigste ist

else

// wird der aktuelle Stein der hoechstwertigste

highest = s;

}

}

}

/\* wenn kein Spieler einen Doppelstein auf der Hand hat

werden die normalen Steine ueberprueft\*/

if (highest == null)

{

for (Player p: allPlayers)

{

for (Stone s: p.getHand())

{

if (highest != null)

{

if (s.getValue() > highest.getValue())

highest = s;

}

else

highest = s;

}

}

}

// auswerten der vorrangegangenen Pruefung und bestimmen des Beginners

int beginnerIndex = 0;

for (Player p: allPlayers)

{

if (p == highest.getPlayer())

break;

else

beginnerIndex++;

}

return beginnerIndex;

}

/\*\*

\* Erzeugt die Spieler

\*/

private void initializePlayers()

{

for (int i = 0; i < allPlayers.length; i++)

{

allPlayers[i] = new Player("Player\_" + (i+1));

view.textOut("Neuer Spieler: " + allPlayers[i].getName());

}

}

/\*\*

\* Initialisiert die Haende der Spieler

\*/

private void initializeHands()

{

int numStones;

int numPlayers = allPlayers.length;

int playerID = 0;

// wenn 2 Spieler spielen bekommt jeder zu Anfang 7 Steine

if (numPlayers == 2)

numStones = 7;

// sonst bekommt jeder 5 Steine

else

numStones = 5;

// Schleife durch alle Spieler

for (Player p: allPlayers)

{

view.textOut("");

// Schleife durch alle Spieler

for (int i = 0; i < numStones; i++)

{

/\* um zu vermeiden, dass mehrere Spieler identische Steine

bekommen koennen bekommt zum Beispiel bei 2 Spielern jeder

Spieler jeden zweiten Stein aus allStones bei 3 Spielern

jeden dritten usw.\*/

int pos = playerID + (numPlayers\*i);

p.addStone(allStones[pos]);

allStones[pos].setPlayer(p);

view.textOut(p.getName() + " Stein " + (i+1) + ": "

+ allStones[pos].getPips1() + "|" + allStones[pos].getPips2());

}

playerID++;

}

}

/\*\*

\* Initialisierung des Pools aus dem die Spieler ziehen koennen,

\* wenn sie nicht legen koennen

\*/

private void initializeTalon()

{

/\* den Talon zu beginn leeren, weil zu Beginn einer neuen

Runde noch Steine enthalten sein koennten \*/

talon.clear();

// Schleife durch alle Steine

for (Stone s: allStones)

{

// wenn ein Stein keinem Spieler gehoert kann er in den Talon

if (s.getPlayer() == null)

talon.add(s);

}

view.textOut("");

for (Stone s: talon)

{

view.textOut("Talon " + (talon.indexOf(s)+1) + ": "

+ s.getPips1() + "|" + s.getPips2());

}

}

/\*\*

\* Uebergibt die aktuelle Hand des aktiven Spielers zum<br>

\* darstellen an das Hauptfenster

\* @param PlayerIndex - Der Index des Spielers, dessen Steine gezeichnet werden sollen

\*/

private void drawStonesOnView(int PlayerIndex)

{

// geht durch alle Steine, die der Spieler auf der Hand hat

for (Stone s: allPlayers[PlayerIndex].getHand())

{

// uebergibt den aktuellen Stein an eine Methode des Hauptfensters,

// die die Steine auf die handPane zeichnet

view.addDominoeToHand(s, allPlayers[PlayerIndex].isFirstMove());

}

}

/\*\*

\* Von diesem Listener werden alle Mausgesten ausgewertet und verarbeitet

\*/

public class MouseClickMotionListener implements MouseListener, MouseMotionListener

{

/\*\* Speichert die Mausposition \*/

private PointerInfo mousePos;

/\*\* Der Stein, den der Spieler gerade legt \*/

private DominoLabel draggedStone;

/\*\* Der Stein, an den der Spieler anlegen will \*/

private DominoLabel target;

/\*\* Die derzeit gedrueckte Maustaste \*/

private int pressedButton;

/\*\* Die Koordinaten bei denen begonnen wurde das Spielfeld zu verschieben \*/

private int draggedAtX, draggedAtY;

/\*\* Die Versatz des Ursprungs des Spielfeldes zum Ursprung des Haupfensters \*/

private int panelOffsetX = 0, panelOffsetY = 0;

@Override

public void mouseDragged(MouseEvent e)

{

// erhalten der aktuellen Koordinaten des Cursors

mousePos = MouseInfo.getPointerInfo();

int mouseX = mousePos.getLocation().x;

int mouseY = mousePos.getLocation().y;

int offsetX, offsetY;

view.showMousePosition(mouseX, mouseY);

// wenn die linke Maustaste gedrueckt gehalten wird

if (pressedButton == 1)

{

/\* wenn die Maustaste ueber einem Stein gedrueckt gehalten und

die Maus verschoben wird \*/

if (e.getSource() instanceof DominoLabel)

{

DominoLabel dLabel = (DominoLabel) e.getSource();

// wenn der Stein verschiebbar ist

if (dLabel.isDraggable())

{

// Ursprungskoordinaten des Hauptfensters holen

Point origin = view.getFrameCoordinates();

draggedStone = dLabel;

// Offset berechnen, wenn der Stein horizontal gedreht ist

if (!dLabel.getStone().isDoublestone() && !dLabel.getStone().isVertical())

{

offsetX = panelOffsetX + dLabel.getWidth()/2 - (origin.x + 102);

offsetY = panelOffsetY + dLabel.getHeight() - (origin.y + 100);

}

// Offset berechnen, wenn der Stein vertikal gedreht ist

else

{

offsetX = panelOffsetX + dLabel.getWidth()/2 - (origin.x + 52);

offsetY = panelOffsetY + dLabel.getHeight() - (origin.y + 175);

}

// Stein an die Position des Mauszeigers verschieben

// (Stein "klebt" am Cursor)

dLabel.setLocation(offsetX + mouseX, offsetY + mouseY);

// pruefen, ob sich Steine ueberschneiden

view.checkIntersection(draggedStone, false, edgePoints, doublePoints);

/\* laesst sich den Stein zurueckgeben mit dem sich der Stein,

der "an der Maus klebt" gerade ueberschneidet \*/

target = view.getCurrentTarget();

}

else

view.textOut("Dieser Stein laesst sich nicht mehr verschieben");

}

}

// wenn die rechte Maustaste gedrueckt gehalten wird

else if (pressedButton == 3)

{

/\* wenn die rechte Maustaste auf einer freien Flaeche des Spielfeldes

gedrueckt gehalten wird \*/

if (e.getSource() instanceof JPanel)

{

view.textOut("JPanel gedraggt");

JPanel p = (JPanel) e.getSource();

Point origin = view.getFrameCoordinates();

view.textOut("OnScreen: " + p.getLocationOnScreen());

view.textOut("MausPos: " + mousePos.getLocation().x + "|" + mousePos.getLocation().y);

/\* den Ursprungsort der Verschiebung auf den Punkt setzen, an dem man angefangen hat

die rechte Maustaste gedrueckt zu halten \*/

e.translatePoint(draggedAtX, draggedAtY);

// hier "klebt" das Spielfeld am Cursor und kann verschoben werden

p.setLocation(mouseX - (draggedAtX + origin.x) + 320, mouseY - (draggedAtY + origin.y) + 178);

/\* das Offset wird auch mit verschoben, damit es bei weiterem Verschieben des Spielfeldes

nicht zu Sprüngen kommt \*/

panelOffsetX = - p.getLocation().x;

panelOffsetY = - p.getLocation().y;

view.textOut("panelOffset: " + panelOffsetX + "|" + panelOffsetY);

}

}

view.updatePanels(); // Die Grafik des Fensters auffrischen, um Anzeigefehlern vorzubeugen

e.consume();

}

@Override

public void mouseMoved(MouseEvent e)

{

mousePos = MouseInfo.getPointerInfo();

view.showMousePosition(mousePos.getLocation().x, mousePos.getLocation().y);

// das Offset des Spielfeldes neu berechnen, wenn es noch nicht berechnet wurde

if (e.getSource() instanceof JPanel && panelOffsetX == 0)

{

JPanel p = (JPanel) e.getSource();

panelOffsetX = - p.getLocation().x;

panelOffsetY = - p.getLocation().y;

view.textOut("panelOffset gesetzt");

}

}

@Override

public void mouseClicked(MouseEvent e)

{

// wenn die rechte Maustaste geklickt wurde

if (e.getButton() == 1)

{

Object c = e.getSource();

// wenn ein Spielstein angeklickt wurde

if (c instanceof DominoLabel)

{

DominoLabel clickedStoneLabel = (DominoLabel) c;

Stone clickedStone = clickedStoneLabel.getStone();

Player player = allPlayers[currentPlayerIndex];

// Es wird geprueft, ob der angeklickte Stein noch auf der Hand des Spielers ist

if (clickedStoneLabel.getParent().getName() == "Hand")

{

// wenn der angeklickte Stein aufs Feld gelegt werden kann

if (DominoRules.checkIfDroppable(clickedStone, edgePoints, spinner, playedDominoes) &&

!player.isDroppedStone())

{

view.textOut("Kann gelegt werden");

// der Stein wird von der Hand aufs Spielfeld gelegt

view.dropFromHand(clickedStoneLabel, panelOffsetX, panelOffsetY, playedDominoes);

// der Stein wird von der Hand des Spielers geloescht

clickedStone.getPlayer().deleteStone(clickedStone);

// der Spieler hat einen Stein gelegt und kann in diesem Zug keinen weiteren mehr legen

player.setDroppedStone(true);

// wird nur beim ersten Stein der gelegt wird aufgerufen

if (playedDominoes == 0)

{

// gelegte Steine um 1 erhoehen

playedDominoes++;

// der Stein laesst sich nicht mehr verschieben

clickedStoneLabel.setNotDraggable();

// Punktzahlen setzen

DominoRules.firstStone(clickedStone, edgePoints, doublePoints);

// wenn der gelegte Stein ein Spinner ist wird hasSpinner auf true gesetzt

hasSpinner = clickedStone.isSpinner();

// ist ein Spinner auf dem Feld wird dieser im Spiel gespeichert

if (hasSpinner)

spinner = clickedStone;

// Punkte im Fenster anzeigen

view.firstPoints(edgePoints, doublePoints);

view.updatePoints(true);

// Spielerpunkte berechnen

if (!DominoRules.calculatePlayerPoints(edgePoints, doublePoints, player, true))

System.err.println("Schade, leider diesmal keine Punkte");

// wenn der Spieler genug Punkte hat, um das Spiel zu gewinnen endet das Spiel

if (player.getPoints() >= winningPoints)

endGame(currentPlayerIndex+1);

// Spielerpunkte im Fenster anzeigen

view.updatePlayerPoints(player, currentPlayerIndex);

// Hand im Fenster loeschen, Spieler wechseln und naechsten Zug starten

view.clearHand();

currentPlayerIndex = DominoRules.switchPlayer(allPlayers, currentPlayerIndex);

startMove();

}

}

}

}

if (c instanceof JPanel)

{

JPanel p = (JPanel) c;

if (p.getName() != "Hand")

{

// Nur für Debugging

/\*

int index = allPlayers[0].getHand().size() - 1;

if (index >= 0)

{

view.addDominoe(allPlayers[0].getHand().get(index), e.getPoint());

allPlayers[0].getHand().remove(index);

}

else

view.textOut("Dieser Spieler hat keine Steine mehr");

\*/

}

}

}

view.textOut(edgePoints[0] + ", " + edgePoints[1]

+ ", " + edgePoints[2] + ", " + edgePoints[3]);

view.textOut(doublePoints[0] + ", " + doublePoints[1]

+ ", " + doublePoints[2] + ", " + doublePoints[3]);

e.consume();

}

@Override

public void mouseEntered(MouseEvent e)

{

// wenn der Cursor ueber einem Stein ist

if (e.getSource() instanceof DominoLabel)

{

DominoLabel label = (DominoLabel) e.getSource();

// Auf der Konsole ausgeben, ob ein Stein legbar ist oder nicht (fuer Debugging)

if (label.getParent().getName() == "Hand")

{

view.textOut("" + DominoRules.checkIfDroppable(label.getStone(),

edgePoints, spinner, playedDominoes));

view.textOut("" + label.getStone().toString());

}

}

}

@Override

public void mouseExited(MouseEvent e)

{

// TODO Auto-generated method stub

}

@Override

public void mousePressed(MouseEvent e)

{

/\* wenn eine Maustaste auf einer frei Flaeche

des Spielfeldes grueckt gehalten wird, aber die Maus

nicht verschoben wird \*/

if (e.getSource() instanceof JPanel)

{

draggedAtX = e.getX() + 323;

draggedAtY = e.getY() + 206;

}

view.textOut("draggedAt: " + draggedAtX + "|"+ draggedAtY);

// gedrueckte Maustaste speichern fuer spaetere Abfragen

pressedButton = e.getButton();

/\* Cursor aendern, wenn die rechte Maustaste ueber dem

Spielfeld gedrueckt gehalten wird \*/

if (pressedButton == 3 && e.getSource() instanceof JPanel)

{

JPanel p = (JPanel) e.getSource();

p.setCursor(Cursor.getPredefinedCursor(Cursor.MOVE\_CURSOR));

}

}

@Override

public void mouseReleased(MouseEvent e)

{

boolean endRound = false;

boolean endGame = false;

/\* wenn die Maustaste ueber dem Spielfeld losgelassen wurde

wird der Standardcursor wiederhergestellt \*/

if (e.getSource() instanceof JPanel)

{

JPanel p = (JPanel) e.getSource();

p.setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT\_CURSOR));

}

// wenn die linke Maustaste losgelassen wurde und bereits Steine gelegt wurden

if (pressedButton == 1 && playedDominoes > 0)

{

// bekommt von checkIntersections zurueck, ob der Stein gesnappt hat

boolean hasSnapped = false;

Player player = allPlayers[currentPlayerIndex];

if (draggedStone != null)

view.textOut("DraggedStone: " + draggedStone.getStone().getPips1()

+ "|" + draggedStone.getStone().getPips2());

// wenn ein Stein ueber einem anderen Stein auf dem Spielfeld losgelassen wurde

if (target != null && draggedStone != null

&& draggedStone.getParent().getName() != "Hand")

{

view.textOut("Es gibt ein target");

// wenn der Zug moeglich ist

if (DominoRules.checkPossibleMove(target.getStone(), draggedStone))

{

// wenn noch kein Spinner auf dem Feld liegt

if (spinner == null)

{

/\* wenn der gelegte Stein ein Spinner ist, wird er

dem Spiel hinzugefuegt \*/

spinner = DominoRules.checkSpinner(draggedStone, target, hasSpinner);

if (spinner != null)

hasSpinner = true;

}

// wenn ein Stein angelegt wurde, wird hasSnapped auf true gesetzt

hasSnapped = view.checkIntersection(draggedStone, true,

edgePoints, doublePoints);

view.textOut("Target: " + target.getStone().getPips1()

+ "|" + target.getStone().getPips2());

}

// wenn ein Stein angelegt wurde

if (hasSnapped)

{

playedDominoes++; // gelegte Steine um 1 erhoehen

// Berechnung der Spielerpunkte und Debugausgabe

if (!DominoRules.calculatePlayerPoints(edgePoints, doublePoints,

player, false))

System.err.println("Schade, leider diesmal keine Punkte");

System.err.println("Punkte " + player.getName() + ": "

+ player.getPoints());

/\* wenn ein Spieler genug Punkte hat, um das Spiel zu

gewinnen endet das Spiel\*/

if (player.getPoints() >= winningPoints)

endGame = true;

// Punkte im Fenster anzeigen

view.updatePlayerPoints(player, currentPlayerIndex);

view.clearHand();

// wenn ein Spieler keine Steine mehr hat, endet die Runde

if (player.isNoStones())

{

endRound = true;

}

// sonst ist der naechste Spieler am Zug

else

{

currentPlayerIndex = DominoRules.switchPlayer(allPlayers,

currentPlayerIndex);

startMove();

}

}

// wenn der Stein nicht angelegt wurde springt er ein Stueck zur Seite

else

if (target != null)

{

draggedStone.setLocation(target.getX() + 150, target.getY() + 150);

view.checkIntersection(draggedStone, false, edgePoints, doublePoints);

}

}

// die Variable fuer den zu legenden Stein wieder auf null setzen

draggedStone = null;

}

// Debugausgaben

view.textOut(edgePoints[0] + ", " + edgePoints[1] + ", "

+ edgePoints[2] + ", " + edgePoints[3]);

view.textOut(doublePoints[0] + ", " + doublePoints[1] + ", "

+ doublePoints[2] + ", " + doublePoints[3]);

// wenn das Spiel beendet werden soll

if (endGame == true)

endGame(currentPlayerIndex+1);

// wenn die Runde beendet ist

if (endRound == true)

{

endRound();

showRoundInfo("", "", 80f, 2100);

}

}

}

/\*\*

\* Der Listener fuer den Button zum ziehen der Steine aus dem Talon

\*/

public class ButtonListener implements ActionListener

{

@Override

public void actionPerformed(ActionEvent e)

{

// es wird der erste Stein aus dem Talon geholt

Stone stone = talon.get(0);

Player player = allPlayers[currentPlayerIndex];

view.textOut(player.getName() + player.getHand().size());

// der Stein wird aus dem Talon geloescht

talon.remove(stone);

// der Stein wird dem Spieler hinzugefuegt

player.addStone(stone);

stone.setPlayer(player);

// der Stein wird im Fenster angezeigt

view.addDominoeToHand(stone, true);

view.textOut(player.getName() + player.getHand().size());

// wenn der Stein auf dem Feld angelegt werden kann

if (DominoRules.checkIfDroppable(stone, edgePoints,

spinner, playedDominoes))

{

// der Button wird deaktiviert, damit man nicht mehr ziehen kann

view.updateButton(false, "");

// der Spieler ist nicht mehr blockiert

player.setBlocked(false);

}

// wenn der Talon leer ist...

if (talon.isEmpty())

{

// ...wird das auf dem Button angezeigt...

view.updateButton(false, "Der Talon ist leer");

// ...und der naechste Spieler ist am Zug

if (player.isBlocked())

{

int otherPlayer = DominoRules.switchPlayer(allPlayers, currentPlayerIndex);

String text1 = "Spieler " + (currentPlayerIndex+1) + " kann nicht legen";

String text2 = "Spieler " + (otherPlayer+1) + " ist am Zug";

view.showGameInfo(text1, text2, 40f, 0, true);

DominoRules.switchPlayer(allPlayers, currentPlayerIndex);

startMove();

}

}

}

}

}package control;

import java.util.ArrayList;

import data.Player;

import data.Stone;

import view.DominoLabel;

public final class DominoRules

{

/\*\* Ueberprueft die grundlegende Kompatibilitaet zweier Steine

\*

\* @param dragged - Der Stein, der geprueft werden soll

\* @param target - Der Stein, der bereits auf dem Feld liegt

\* @return <b>true</b> - wenn kompatibel <br>

\* <b>false</b> - wenn nicht kompatibel

\*/

public static boolean checkCompatibility(DominoLabel dragged, DominoLabel target)

{

int d\_p1 = dragged.getStone().getPips1();

int d\_p2 = dragged.getStone().getPips2();

int t\_p1 = target.getStone().getPips1();

int t\_p2 = target.getStone().getPips2();

// wenn einer Seite des einen Steins mit einer Seite des anderen uebereinstimmt

if (d\_p1 == t\_p1 || d\_p1 == t\_p2 || d\_p2 == t\_p1 || d\_p2 == t\_p2)

return true; // sind die Steine grundlegend kompatibel

else

return false;

}

/\*\*

\* Prueft, ob an der rechten Seite des Steins angelegt werden kann

\* @param draggedStone - Der Stein, der gelegt werden soll

\* @param target - Der Stein, an den angelegt werden soll

\* @return <b>true</b> - Der Stein soll rechts angelegt werden<br>

\* <b>false</b> - Der Stein soll links angelegt werden

\*/

public static boolean snapRight(DominoLabel draggedStone, DominoLabel target)

{

// Steine aus den DominoLabeln holen

Stone dStone = draggedStone.getStone();

Stone tStone = target.getStone();

// Es wird geprueft, welche Seite des Zielsteins mit dem zu legenden Stein uebereinstimmt

int equalPip = giveEqualPip(dStone, tStone);

// wenn das Ziel ein Doppelstein ist

if (equalPip == 0)

{

// wenn das Ziel rechts keinen Nachbarn hat

if (tStone.getRightNeighbour() == null)

{

// wird geprueft, ob der Stein gedreht werden muss

dStone.checkRotationHorizontal(tStone, true);

draggedStone.updateImage(); // das Bild des Labels wird aktualisiert

return true; // der Stein kann rechts angelegt werden

}

// das Ziel hat rechts einen Nachbarn

else

{

dStone.checkRotationHorizontal(tStone, false);

draggedStone.updateImage();

return false; // der Stein kann links angelegt werden

}

}

// wenn das Ziel rechts keinen Nachbarn hat und die uebereinstimmende Augenzahl

// auf rechten Seite des Zielsteins ist

if(tStone.getRightNeighbour() == null && equalPip == 2)

{

// wenn der anzulegende Stein falsch gedreht ist, muss er gedreht werden

if (tStone.getPips2() != dStone.getPips1())

{

dStone.checkRotationHorizontal(target.getStone(), true);

draggedStone.updateImage();

}

return true; // der Stein kann rechts angelegt werden

}

// sonst soll links angelegt werden

else

{

if (tStone.getPips1() != dStone.getPips2())

{

dStone.checkRotationHorizontal(target.getStone(), false);

draggedStone.updateImage();

}

return false; // der Stein kann links angelegt werden

}

}

/\*\*

\* Ueberprueft fuer das horizontale anlegen, wo ein Stein Nachbarn hat

\* @param dragged - Der anzulegende Stein

\* @param target - Der Zielstein

\* @param snapRight - boolean, ob rechts angelegt werden soll

\* @return <b>true</b> - der Stein hat an der Zielseite keinen Nachbarn <br>

\* <b>false</b> - sonst

\*/

public static boolean checkNeighboursHorizontal(DominoLabel dragged, DominoLabel target, boolean snapRight)

{

// wenn rechts angelegt werden soll

if (snapRight)

{

// wenn der Stein rechts keinen Nachbarn hat

if (target.getStone().getRightNeighbour() == null)

return true; // kann rechts angelegt werden

else

return false; // es kann nicht rechts angelegt werden

}

else

{

if (target.getStone().getLeftNeighbour() == null)

return true;

else

return false;

}

}

/\*\*

\* Prueft, ob an der oberen Seite des Steins angelegt werden kann

\* @param draggedStone - Der Stein, der gelegt werden soll

\* @param target - Der Stein, an den angelegt werden soll

\* @return <b>true</b> - Der Stein soll oben angelegt werden<br>

\* <b>false</b> - Der Stein soll unten angelegt werden

\*/

public static boolean snapTop(DominoLabel draggedStone, DominoLabel target)

{

// Steine aus den Labels holen

Stone dStone = draggedStone.getStone();

Stone tStone = target.getStone();

// uebereinstimmende Seite berechnen

int equalPip = giveEqualPip(dStone, tStone);

// wenn der Stein ein Doppelstein ist

if (equalPip == 0)

{

// wenn der Stein oben keinen Nachbarn hat

if (tStone.getTopNeighbour() == null)

{

// Stein richtig drehen und Bild im Label aktualisieren

dStone.checkRotationVertical(tStone, true);

draggedStone.updateImage();

return true; // Der Stein soll oben angelegt werden

}

// wenn der Stein obenen einen Nachbarn hat

else

{

dStone.checkRotationVertical(tStone, false);

draggedStone.updateImage();

return false; // Der Stein soll unten angelegt werden

}

}

// wenn der Stein oben keinen Nachbarn hat und die

// obere Seite des Zielsteins uebereinstimmt

if (tStone.getTopNeighbour() == null && equalPip == 1)

{

dStone.checkRotationVertical(tStone, true);

draggedStone.updateImage();

return true; // Der Stein soll oben angelegt werden

}

else

{

dStone.checkRotationVertical(tStone, false);

draggedStone.updateImage();

return false; // Der Stein soll oben angelegt werden

}

}

/\*\*

\* Ueberprueft fuer das vertikale anlegen, wo ein Stein Nachbarn hat

\* @param dragged - Der anzulegende Stein

\* @param target - Der Zielstein

\* @param snapRight - boolean, ob rechts angelegt werden soll

\* @return <b>true</b> - der Stein hat an der Zielseite keinen Nachbarn <br>

\* <b>false</b> - sonst

\*/

public static boolean checkNeighboursVertical(DominoLabel draggedStone, DominoLabel target, boolean snapTop)

{

// wenn oben angelegt werden soll

if (snapTop)

{

// wenn der Zielstein oben keinen Nachbarn hat

if (target.getStone().getTopNeighbour() == null)

return true; // kann oben angelegt werden

else

return false; // sonst kann nicht oben angelegt werden

}

else

{

// wenn der Zielstein unten keinen Nachbarn hat

if (target.getStone().getBottomNeighbour() == null)

return true; // kann unten angelegt werden

else

return false; // sonst kann nicht unten angelegt werden

}

}

/\*\*

\* Prueft, welche Seite des Zielsteins mit dem anzulegenden Stein kompatibel ist

\* @param dragged - Der Stein, der gelegt werden soll

\* @param target - Der Stein, an den angelegt werden soll

\* @return <b>0</b> - wenn das Ziel ein Doppelstein ist <br>

\* <b>1</b> - wenn die linke Seite des Zielsteins mit <br>

\* einer Seite des zu legenden Steins uebereinstimmt <br>

\* <b>2</b> - sonst

\*/

private static int giveEqualPip(Stone dragged, Stone target)

{

if (target.isDoublestone())

return 0;

if (target.getPips1() == dragged.getPips1() || target.getPips1() == dragged.getPips2())

return 1;

else

return 2;

}

/\*\*

\* Prueft, ob vertikal angelegt werden muss

\* @param target - Der Zielstein

\* @return <b>true</b> - wenn vertikal angelegt werden muss<br>

\* <b>false</b> - sonst

\*/

public static boolean checkIfVertical(DominoLabel target)

{

// Stein aus dem Label holen

Stone tStone = target.getStone();

// wenn der Zielstein ein Spinner ist und horizontal nicht angelegt werden kann

if (tStone.isSpinner() && tStone.getLeftNeighbour() != null && tStone.getRightNeighbour() != null)

return true;

// wenn der Zielstein bereits vertikal liegt

else if (tStone.isVertical())

return true;

else

return false;

}

/\*\*

\* Prueft, ob ein Stein der von der Hand aufs Feld gebracht wurde an den Stein angelegt<br>

\* werden kann, über den er gelegt wurde

\* @param target - Der Zielstein

\* @param draggedStone - Der Stein der angelegt werden soll

\* @return - <b>true</b> - wenn angelegt werden kann<br>

\* <b>false</b> - wenn nicht angelegt werden kann

\*/

public static boolean checkPossibleMove(Stone target, DominoLabel draggedStone)

{

String error = "Dieser Zug ist leider nicht moeglich";

// Nachbarn des Zielsteins zum leichteren Zugriff aus Stein holen

Stone left = target.getLeftNeighbour();

Stone right = target.getRightNeighbour();

Stone top = target.getTopNeighbour();

Stone bottom = target.getBottomNeighbour();

// wenn der Zielstein horizontal liegt

if (!target.isVertical())

if (!target.isSpinner()) // wenn der Zielstein kein Spinner ist

{

// wenn der Zielstein links oder rechts keinen Nachbarn hat

if (left == null || right == null)

{

return true;

}

else

{

System.out.println(error);

return false;

}

}

else

{

// wenn der Spinner links, rechts, oben oder unten keinen Nachbarn hat

if (left == null || right == null || top == null || bottom == null)

{

return true;

}

else

{

System.out.println(error);

return false;

}

}

// wenn der Zielstein vertikal liegt

else

{

// wenn der Zielstein oben und unten keinen Nachbarn hat

if (top == null || bottom == null)

{

return true;

}

else

{

System.out.println(error);

return false;

}

}

}

/\*\*

\* Prueft, ob der angelegte Stein oder der Zielstein ein Spinner werden soll

\* @param draggedStone - Der angelegte Stein

\* @param target - Der Stein, an den angelegt wurde

\* @param hasSpinner - Gibt an, ob bereits ein Spinner auf dem Feld liegt

\* @return - Den Stein, der der neue Spinner ist oder null

\*/

public static Stone checkSpinner(DominoLabel draggedStone, DominoLabel target, boolean hasSpinner)

{

// wenn das Spiel keinen Spinner hat und der angelegte Stein ein Doppelstein ist

if (!hasSpinner && draggedStone.getStone().isDoublestone())

{

// Im Steinobjekt den Stein zum Spinner machen

draggedStone.getStone().setSpinner(true);

return draggedStone.getStone();

}

else if (!hasSpinner && target.getStone().isDoublestone())

{

target.getStone().setSpinner(true);

return target.getStone();

}

// wenn es schon einen Spinner gibt, oder der keiner der Steine ein Doppelstein ist

else

return null;

}

/\*\*

\* Berechnet die Punkte, wenn ein Stein links angelegt wird

\* @param draggedStone - Der Stein, der angelegt wird

\* @param target - Der Stein, an den angelegt wird

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - Gibt an, an welchen Kanten Doppelsteine liegen

\*/

public static void calculatePointsLeft(DominoLabel draggedStone, DominoLabel target,

int[] edgePoints, boolean[] doublePoints)

{

// holt sich die Punkte der linken Seite aus dem Stein und schreibt

// sie das Punkte-Array an den Index 0 fuer die linke Seite

edgePoints[0] = draggedStone.getStone().getPips1();

// wenn der angelegte Stein ein Doppelstein ist

if (draggedStone.getStone().isDoublestone())

doublePoints[0] = true; // wird gespeichert, dass links ein Doppelstein ist

// ist der angelegte Stein kein Doppelstein

else

doublePoints[0] = false; // wird gespeichert, dass dort kein Doppelstein (mehr) ist

}

/\*\*

\* Berechnet die Punkte, wenn ein Stein rechts angelegt wird

\* @param draggedStone - Der Stein, der angelegt wird

\* @param target - Der Stein, an den angelegt wird

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - Gibt an, an welchen Kanten Doppelsteine liegen

\*/

public static void calculatePointsRight(DominoLabel draggedStone, DominoLabel target,

int[] edgePoints, boolean[] doublePoints)

{

// Kommentare aehnlich calculatePointsLeft

edgePoints[1] = draggedStone.getStone().getPips2();

if (draggedStone.getStone().isDoublestone())

doublePoints[1] = true;

else

doublePoints[1] = false;

}

/\*\*

\* Berechnet die Punkte, wenn ein Stein unten angelegt wird

\* @param draggedStone - Der Stein, der angelegt wird

\* @param target - Der Stein, an den angelegt wird

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - Gibt an, an welchen Kanten Doppelsteine liegen

\*/

public static void calculatePointsBottom(DominoLabel draggedStone, DominoLabel target,

int[] edgePoints, boolean[] doublePoints)

{

//Kommentare aehnlich calculatePointsLeft

edgePoints[3] = draggedStone.getStone().getPips2();

if (draggedStone.getStone().isDoublestone())

doublePoints[3] = true;

else

doublePoints[3] = false;

}

/\*\*

\* Berechnet die Punkte, wenn ein Stein oben angelegt wird

\* @param draggedStone - Der Stein, der angelegt wird

\* @param target - Der Stein, an den angelegt wird

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - Gibt an, an welchen Kanten Doppelsteine liegen

\*/

public static void calculatePointsTop(DominoLabel draggedStone, DominoLabel target,

int[] edgePoints, boolean[] doublePoints)

{

// Kommentare aehnlich calculatePointsLeft

edgePoints[2] = draggedStone.getStone().getPips1();

if (draggedStone.getStone().isDoublestone())

doublePoints[2] = true;

else

doublePoints[2] = false;

}

/\*\*

\* Berechnet die Punkte der offenen Kanten mit Beruecksichtigung von Doppelsteinen

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - Gibt an, an welchen Kanten Doppelsteine liegen

\* @param firstStone - true uebergeben, wenn diese Methode fuer den ersten Stein

\* der Runde aufgerufen wird

\* @return - Die berechnete Punktzahl

\*/

public static int calculatePoints(int[] edgePoints, boolean[] doublePoints, boolean firstStone)

{

// Deklaration und Initialisierung der Rueckgabevariable sowie des Schleifenzaehlers

int points = 0;

int i = 0;

// wenn die Methode nicht fuer den ersten Zug aufgerufen wurde

if (!firstStone)

{

// Schleife durch alle Kantenpunkte

for (int p: edgePoints)

{

// Wenn p eine 7 ist, wurde an diese Kante noch nicht angelegt. Meist oben oder unten

if (p != 7)

{

if (doublePoints[i] == false)

// Einfache Addition, wenn kein Doppelstein an der Kante liegt

points += p;

else

// Verdoppelung der Punkte, wenn an der Kante ein Doppelstein liegt

points += p\*2;

}

i++; // Inkrementieren des Schleifenzaehlers

}

}

// wenn die Methode fuer den ersten Zug einer Runde aufgerufen wurde

else

// beim ersten Zug muessen nur die linke und rechte Kante addiert werden;

return edgePoints[0]+edgePoints[1];

return points; // Rueckgabe der Punkte

}

/\*\*

\* Prueft, ob ein Stein von der Hand des Spielers auf dem Feld angelegt werden kann

\* @param stone - Der Stein der gelegt werden soll

\* @param edgePoints - Das Array, das die Punkte der offenen Enden beinhaltet

\* @param spinner - Der Spinner, der auf dem Feld liegt. Kann auch <i>null</i> sein

\* @param playedDominoes - Die Anzahl der bereits gelegten Steine

\* @return <b>true</b> - wenn der Stein gelegt werden kann <br>

\* <b>false</b> - wenn der Stein nicht gelegt werden kann

\*/

public static boolean checkIfDroppable(Stone stone, int[] edgePoints, Stone spinner, int playedDominoes)

{

// Deklaration und Initialisierung der Zaehlvariable

int i = 0;

// wenn noch keine Steine auf dem Feld liegen kann der Stein auf jeden Fall gelegt werden

if (playedDominoes == 0)

return true;

// Schleife durch alle Kantenpunkte

for (int points: edgePoints)

{

// wenn es keinen Spinner gibt muessen die vertikalen Kanten nicht geprueft werden

// wenn i groesser wird als 1 bedeutet das, dass die vertikalen Kanten geprueft werden

if (i > 1 && spinner == null)

break;

// wenn die Punktzahl mit einer der Augenzahlen uebereinstimmt

if (points == stone.getPips1() || points == stone.getPips2())

return true; // der Stein kann gelegt werden

i++; // Schleifenzaehler inkrementieren

}

// wenn eine der vertikalen Kanten belegt ist und ein Spinner auf dem Feld liegt

if ((edgePoints[2] == 7 || edgePoints[3] == 7) && spinner != null)

{

// wenn an eine der beiden Kanten des spinners angelegt werden kann (null)

if (spinner.getTopNeighbour() == null || spinner.getBottomNeighbour() == null)

{

// wenn eine der Augenzahlen des zu legenden Steins mit der Augenzahl des Spinners uebereinstimmt

if (stone.getPips1() == spinner.getPips1() || stone.getPips2() == spinner.getPips1())

return true; // der Stein kann gelegt werden

}

}

return false; // der Stein kann nicht gelegt werden

}

/\*\*

\* Wechselt den aktiven Spieler

\* @param allPlayers - Array, dass alle Spieler haelt

\* @param currentPlayerIndex - Index des aktuellen Spielers

\* @return Gibt den Index des neuen aktuellen Spieler zurueck

\*/

public static int switchPlayer(Player[] allPlayers, int currentPlayerIndex)

{

// wenn die Spieler gewechselt werden hat der aktuelle Spieler seinen ersten Zug gemacht,

// daher kann die Variable firstMove im Spieler auf false gesetzt werden

allPlayers[currentPlayerIndex].setfirstMove(false);

// wenn der letzte Spieler am Zug war, ist der erste Spieler an der Reihe

if ((allPlayers.length - 1) == currentPlayerIndex)

return 0;

// sonst ist der naechste Spieler dran

else

return ++currentPlayerIndex;

}

/\*\*

\* Setzt die Punktzahlen fuer den ersten Stein, der gelegt wurde

\* @param stone - Der Stein, der gelegt wurde

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - speichert, an welcher offenen Kante ein Doppelstein liegt

\*/

public static void firstStone(Stone stone, int[] edgePoints, boolean[] doublePoints)

{

// Punktzahlen fuer linke und rechte Kante setzen

edgePoints[0] = stone.getPips1();

edgePoints[1] = stone.getPips2();

// wenn der erste Stein ein Doppelstein ist wird gespeichert, dass an der linken

// Kante ein Doppelstein liegt, weil der zweite Stein rechts angelegt wird

if (stone.isDoublestone())

{

doublePoints[0] = true;

// da es der erste Doppelstein ist der gelegt wurde ist dieser der Spinner

stone.setSpinner(true);

}

}

/\*\*

\*

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - speichert, an welcher offenen Kante ein Doppelstein liegt

\* @param player - der Spieler fuer den die Punkte berechnet werden sollen

\* @param firstMove - true, wenn die Methode fuer den ersten Zug aufgerufen wird

\* @return <b>true</b> - wenn der Spieler Punkte bekommen hat<br>

\* <b>false</b> - wenn der Spieler keine Punkte bekommen hat

\*/

public static boolean calculatePlayerPoints(int[] edgePoints, boolean[] doublePoints

, Player player, boolean firstMove)

{

int points = calculatePoints(edgePoints, doublePoints, firstMove);

// wenn die Punkte glatt durch 5 teilbar sind...

if (points % 5 == 0)

{

// ...werden die Punkte des Spielers um diesen Wert erhoeht

player.increasePoints(points);

return true;

}

return false;

}

/\*\*

\* Prueft welcher Spieler eine Runde gewinnt

\* @param allPlayers - Array, das alle Spieler haelt

\* @param returnPoints - wenn true uebergeben wird, werden die Punkte zurueckgegeben

\* @return wenn returnPoints true - Der Index des Gewinners<br>

\* wenn returnPoints false - Die Punkte, die der Gewinner gutgeschrieben bekommt

\*/

public static int calculateRoundPoints(Player[] allPlayers, boolean returnPoints)

{

// vor der Pruefung gewinnt Spieler 1

int winner = 0;

int loser = 1;

// wenn Spieler 1 mehr Steine auf der Hand hat als Spieler 2...

if (allPlayers[0].getHand().size() > allPlayers[1].getHand().size())

{

// gewinnt Spieler 2 die Runde

winner = 1;

loser = 0;

}

// die Werte aller Steine auf der Hand des Verlierers werden aufsummiert

int winnerPoints = calculateHandPoints(allPlayers[loser].getHand());

// Rueckgabe des jeweiligen Wertes abhaengig von returnPoints

if (returnPoints)

return winnerPoints; // Punkte des Gewinners

else

{

// nur wenn returnPoints false ist werden dem Spieler die Punkte gutgeschrieben

allPlayers[winner].increasePoints(winnerPoints);

return winner; // der Index des Gewinners

}

}

/\*\*

\* Berechnet die Summe der Werte aller Steine auf der Hand eines Spielers

\* @param hand - Die Hand eines Spielers

\* @return Die berechnete Summe

\*/

private static int calculateHandPoints(ArrayList<Stone> hand)

{

int points = 0;

for (Stone s: hand)

points += s.getValue();

while (points % 5 != 0)

points += 1;

return points;

}

}package data;

import java.util.ArrayList;

/\*\*

\* Der Spieler

\*/

public class Player

{

/\*\* Die Hand des Spielers \*/

private ArrayList<Stone> hand = new ArrayList<Stone>();

/\*\* Der Name des Spielers \*/

private String name;

/\*\* Erster Zug des Spielers (true/false) \*/

private boolean firstMove = true;

/\*\* Wird benoetigt um festzustellen, ob ein Spieler

bereits einen Stein aufs Feld gelegt hat\*/

private boolean droppedStone = false;

/\*\* true, wenn der Spieler nicht legen kann \*/

private boolean blocked = false;

/\*\* true, wenn der Spieler keine Steine mehr auf der Hand hat \*/

private boolean noStones = false;

/\*\* Die Punkte des Spielers \*/

private int points = 0;

/\*\*

\* Der Konstruktor mit Uebergabe des Namens

\* @param name - Der Name des Spielers

\*/

public Player(String name)

{

if (name != null && name != "")

this.name = name;

}

/\*\*

\* Fuegt dem Spieler einen Stein zur Hand hinzu

\* @param stone - Der Stein der hinzugefuegt wird

\*/

public void addStone(Stone stone)

{

hand.add(stone);

}

/\*\*

\* Loescht einen Stein von der Hand

\* @param stone - Der Stein der geloescht werden soll

\*/

public void deleteStone(Stone stone)

{

hand.remove(stone);

if (hand.isEmpty())

noStones = true;

}

public ArrayList<Stone> getHand()

{

return hand;

}

public String getName()

{

return name;

}

public void increasePoints(int points)

{

this.points += points;

}

public int getPoints()

{

return points;

}

public Stone placeStone(Stone s)

{

return s;

}

public void setfirstMove(boolean isFirstMove)

{

this.firstMove = isFirstMove;

}

public boolean isFirstMove()

{

return firstMove;

}

public boolean isDroppedStone()

{

return droppedStone;

}

public void setDroppedStone(boolean droppedStone)

{

this.droppedStone = droppedStone;

}

public boolean isBlocked()

{

return blocked;

}

public void setBlocked(boolean blocked)

{

this.blocked = blocked;

}

public boolean isNoStones()

{

return noStones;

}

public void resetNoStones()

{

noStones = false;

}

}package data;

import java.awt.Dimension;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.Toolkit;

import java.awt.geom.AffineTransform;

import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

/\*\*

\* Der Dominostein

\*/

public class Stone

{

/\*\* linke bzw. obere Augenzahl \*/

private int pips1 = 36;

/\*\* rechte bzw. untere Augenzahl \*/

private int pips2 = 36;

/\*\* Wert des Steins \*/

private int value;

/\*\* Doppelstein ja/nein \*/

private boolean doublestone;

/\*\* Spinner ja/nein \*/

private boolean spinner;

/\*\* Vertikal gelegt ja/nein \*/

private boolean vertical;

/\*\* Das Bild des Dominosteins \*/

private Image icon;

/\*\* Das Originalbild \*/

private BufferedImage rawImage;

/\*\* linker Nachbar \*/

private Stone leftNeighbour;

/\*\* rechter Nachbar \*/

private Stone rightNeighbour;

/\*\* oberer Nachbar \*/

private Stone topNeighbour;

/\*\* unterer Nachbar \*/

private Stone bottomNeighbour;

/\*\* Groesse des Bildes auf dem Stein \*/

private Dimension imageSize;

/\*\* Der Spieler, der den Stein zuerst auf der Hand haelt \*/

private Player player;

/\*\*

\* Konstruktor Dominostein

\* @param pips1 - Augenzahl 1

\* @param pips2 - Augenzahl 2

\* @param imageSize - Bildgroesse

\*/

public Stone(int pips1, int pips2, Dimension imageSize)

{

this.pips1 = pips1;

this.pips2 = pips2;

this.imageSize = imageSize;

// Wert des Steins berechnen

calculateValue();

// Pruefen, ob der Stein ein Doppelstein ist

setDoublestone();

}

public void loadIcon()

{

BufferedImage image;

// Es wird ein String erzeugt, der auf die passende Bilddatei verweist

String fileName = new String("ImageSrc/" + "new" + pips1 + "\_" + pips2 + ".png");

System.out.println("Dateiname: " + fileName);

// Das Bild laden und eventuellen Fehler ausgeben

try {

image = ImageIO.read(new File(fileName));

rawImage = image;

icon = Toolkit.getDefaultToolkit().createImage(image.getSource());

} catch (IOException e) {

System.err.println("Bild: " + fileName + " konnte nicht geladen werden");

e.printStackTrace();

}

}

public void rotateImage(int angle)

{

/\* Das Bild wird bei jedem Schleifendurchlauf um

90 Grad gedreht \*/

int z = angle / 90;

for (int i = 1; i<=z; i++)

{

double degrees = 90;

double radians = Math.toRadians(degrees);

double sin = Math.abs(Math.sin(radians));

double cos = Math.abs(Math.cos(radians));

// Berechnungen zur neuen Breite und Hoehe des Steins

int newWidth = (int)Math.round(icon.getWidth(null)

\* cos + icon.getHeight(null) \* sin);

int newHeight = (int)Math.round(icon.getWidth(null)

\* sin + icon.getHeight(null) \* cos);

// Bild auf Stein mittig ausrichten

int x = (newWidth - icon.getWidth(null)) / 2;

int y = (newHeight - icon.getHeight(null)) / 2;

System.out.println("Breite: " + newWidth + ", Hoehe: " + newHeight);

/\* neues BufferedImage erstellen, auf das das rotierte Bild

gezeichnet werden soll \*/

BufferedImage rotatedImage = new BufferedImage(newWidth, newHeight,

BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g2d = rotatedImage.createGraphics();

AffineTransform at = new AffineTransform();

// Bild rotieren

at.setToRotation(radians, x + (icon.getWidth(null) / 2), y

+ (icon.getHeight(null) / 2));

// Bild verschieben

at.translate(x, y);

// Drehung und Verschiebung ausfuehren

g2d.setTransform(at);

// Transformiertes Bild zeichnen

g2d.drawImage(rawImage, 0, 0, null);

g2d.dispose();

// Originalbild speichern (zum erneuten rotieren)

rawImage = rotatedImage;

// Das Icon fuer das DominLabel aus dem rotierten Bild erstellen

icon = Toolkit.getDefaultToolkit().createImage(rotatedImage.getSource());

}

}

public int getPips1()

{

return pips1;

}

public int getPips2()

{

return pips2;

}

public int getValue()

{

return value;

}

private void calculateValue()

{

value = pips1+pips2;

}

public boolean isDoublestone()

{

return doublestone;

}

private void setDoublestone()

{

if (pips1 == pips2)

this.doublestone = true;

}

public Image getIcon()

{

return icon;

}

public void setPlayer(Player p)

{

if (p != null)

this.player = p;

}

public Player getPlayer()

{

return player;

}

/\*\*

\* Ueberprueft, ob ein Stein horizontal gespiegelt werden muss

\* @param target - der Stein, an den angelegt werden soll

\* @param snapRight - ob der Stein rechts angelegt werden soll

\* @return <b>true</b> - wenn der Stein gedreht wurde

\*/

public boolean checkRotationHorizontal(Stone target, boolean snapRight)

{

// wenn der Stein rechts angelegt werden soll

if (snapRight)

{

/\* wenn die Augenzahlen nicht uebereinstimmen

muss der Stein gespiegelt werden\*/

if (pips1 != target.getPips2())

{

// Drehung um 180 Grad

rotateImage(180);

// Augenzahlen tauschen

togglePips();

return true;

}

else

return false;

}

// wenn der Stein links angelegt werden soll

else

{

if (pips2 != target.getPips1())

{

rotateImage(180);

togglePips();

return true;

}

else

return false;

}

}

/\*\*

\* Ueberprueft, ob ein Stein vertikal gespiegelt werden muss

\* @param target - der Stein, an den angelegt werden soll

\* @param snapRight - ob der Stein rechts angelegt werden soll

\* @return <b>true</b> - wenn der Stein gedreht wurde

\*/

public boolean checkRotationVertical(Stone target, boolean snapTop)

{

setVertical(true);

rotateImage(90);

if (snapTop)

{

if (pips2 != target.getPips1())

{

rotateImage(180);

togglePips();

return true;

}

else

return false;

}

else

{

if (pips1 != target.getPips2())

{

rotateImage(180);

togglePips();

return true;

}

else

return false;

}

}

/\*\*

\* Augenzahlen tauschen

\*/

private void togglePips()

{

int temp = pips1;

pips1 = pips2;

pips2 = temp;

}

public Stone getLeftNeighbour()

{

return leftNeighbour;

}

public void setLeftNeighbour(final Stone s)

{

leftNeighbour = s;

}

public Stone getRightNeighbour()

{

return rightNeighbour;

}

public void setRightNeighbour(final Stone s)

{

this.rightNeighbour = s;

}

public Stone getTopNeighbour()

{

return topNeighbour;

}

public void setTopNeighbour(final Stone s)

{

this.topNeighbour = s;

}

public Stone getBottomNeighbour()

{

return bottomNeighbour;

}

public void setBottomNeighbour(final Stone s)

{

this.bottomNeighbour = s;

}

public void setSpinner(boolean isSpinner)

{

this.spinner = isSpinner;

}

public boolean isSpinner()

{

return spinner;

}

public void setVertical (boolean isVertical)

{

this.vertical = isVertical;

}

public boolean isVertical()

{

return vertical;

}

}package view;

import javax.swing.ImageIcon;

import javax.swing.JLabel;

import data.Stone;

/\*\*

\* Ein JLabel, dass die Dominosteine des Spiels halten kann

\*/

public class DominoLabel extends JLabel

{

private static final long serialVersionUID = 1L;

private Stone stone;

private boolean draggable = true;

public DominoLabel(Stone stone)

{

// System.out.println("Konstruktor Domino");

this.stone = stone;

updateImage();

}

public Stone getStone()

{

return stone;

}

public boolean isDraggable()

{

return draggable;

}

public void setNotDraggable()

{

this.draggable = false;

}

public void updateImage()

{

this.setIcon(new ImageIcon(this.stone.getIcon()));

this.setSize(this.getIcon().getIconWidth(), this.getIcon().getIconHeight());

}

}package view;

import control.DominoGame;

/\*\*

\* Main Domino

\*/

public class Main

{

public static void main(String[] args)

{

MainWindow view = new MainWindow();

new DominoGame(view, 2);

}

}package view;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.FlowLayout;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.Point;

import java.awt.Rectangle;

import java.awt.ScrollPane;

import java.awt.Shape;

import java.awt.Toolkit;

import java.awt.geom.AffineTransform;

import java.awt.geom.Path2D;

import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Properties;

import java.util.Timer;

import java.util.TimerTask;

import java.util.concurrent.TimeUnit;

import javax.imageio.ImageIO;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JComponent;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import data.Player;

import data.Stone;

import control.DominoGame.ButtonListener;

import control.DominoGame.MouseClickMotionListener;

import control.DominoRules;

/\*\*

\* Das Hauptfenster des Spiels

\*/

public class MainWindow

{

/\*\* Die paintingComponent rendert jeden Frame \*/

private PaintingComponent paintingComponent = new PaintingComponent();

/\*\* Eine Liste der Bilder der Dominosteine, die gerendert werden sollen \*/

private ArrayList<RenderImage> renderedImages = new ArrayList<RenderImage>();

/\*\* Das Fenster selbst \*/

private JFrame frame;

private MouseClickMotionListener mouseHandler;

private ButtonListener buttonListener;

private JLabel lbl\_mouseX = new JLabel("Position X: 0");

private JLabel lbl\_mouseY = new JLabel("Position Y: 0");

private JLabel lbl\_player1Points = new JLabel("Spieler 1: 0");

private JLabel lbl\_player2Points = new JLabel("Spieler 2: 0");

private JLabel lbl\_splash1 = new JLabel();

private JLabel lbl\_splash2 = new JLabel();

private JButton btn\_drawTalon = new JButton("Stein ziehen");

/\*\* Zeigt die Punkte des Spiels an \*/

private PointsLabel lbl\_points = new PointsLabel("Punkte: 0");

private DominoLabel targetLabel = null;

private ArrayList<DominoLabel> dLabels = new ArrayList<DominoLabel>();

private ArrayList<DominoLabel> handLabels = new ArrayList<DominoLabel>();

private JPanel graphicsPane = new JPanel();

private ScrollPane scrollPane = new ScrollPane();

private JPanel handPane = new JPanel();

private JPanel contentPane;

/\*\*

\* Initialisiert alle Komponenten

\* @param allStones

\* @param mouseHandler

\* @param buttonListener

\*/

public void initializeWindow(Stone[] allStones,

MouseClickMotionListener mouseHandler, ButtonListener buttonListener)

{

this.mouseHandler = mouseHandler;

this.buttonListener = buttonListener;

frame = new JFrame("Domino");

contentPane = (JPanel) frame.getContentPane();

scrollPane.setBounds(-2, 734, 1600, 141);

JLabel lbl\_help1 = new JLabel("Rechte Maustaste "

+ "gedrueckt halten, um das Spielfeld zu verschieben");

JLabel lbl\_help2 = new JLabel("Mit linker Maustaste "

+ "auf einen Stein im unteren Feld klicken, um ihn zu legen");

//Initialisierung aller Kompontenten

contentPane.setLayout(null);

contentPane.setBounds(0, 0, 1600, 900);

contentPane.add(lbl\_mouseX);

contentPane.add(lbl\_mouseY);

contentPane.add(lbl\_points);

contentPane.add(lbl\_splash1);

contentPane.add(lbl\_splash2);

contentPane.add(lbl\_player1Points);

contentPane.add(lbl\_player2Points);

contentPane.add(btn\_drawTalon);

contentPane.add(lbl\_help1);

contentPane.add(lbl\_help2);

contentPane.add(scrollPane);

contentPane.add(graphicsPane);

scrollPane.add(handPane);

lbl\_help1.setBounds(600, 5, 400, 20);

lbl\_help1.setHorizontalAlignment(JLabel.CENTER);

lbl\_help2.setBounds(575, 25, 450, 20);

lbl\_help2.setHorizontalAlignment(JLabel.CENTER);

lbl\_splash1.setBounds(0, (contentPane.getHeight()/2) - 200, contentPane.getWidth(), 100);

lbl\_splash1.setHorizontalAlignment(JLabel.CENTER);

lbl\_splash1.setFont(lbl\_splash1.getFont().deriveFont(100.0f));

lbl\_splash2.setBounds(0, (contentPane.getHeight()/2) -100, contentPane.getWidth(), 150);

lbl\_splash2.setHorizontalAlignment(JLabel.CENTER);

lbl\_splash2.setFont(lbl\_splash1.getFont().deriveFont(100.0f));

lbl\_mouseX.setBounds(0, 0, 100, 20);

lbl\_mouseY.setBounds(0, 20, 100, 20);

lbl\_points.setBounds(contentPane.getWidth() - 89, 13, 70, 21);

lbl\_points.setIcon(new ImageIcon("ImageSrc/BG\_Points.png"));

lbl\_points.setIconTextGap(-65);

lbl\_player1Points.setBounds(contentPane.getWidth() - 93, 45, 105, 20);

lbl\_player2Points.setBounds(contentPane.getWidth() - 93, 65, 105, 20);

btn\_drawTalon.setBounds((contentPane.getWidth()/2) - 77, scrollPane.getY() - 40, 155, 40);

btn\_drawTalon.addActionListener(buttonListener);

// FlowLayout fuer die ScrollPane, damit die Steine nebeneinander angeordnet werden

FlowLayout flow = new FlowLayout();

flow.setVgap(10);

flow.setHgap(15);

// Dem JPanel fuer die Hand einen Namen geben fuer spaetere Pruefung

handPane.setName("Hand");

handPane.setBackground(new Color(140,100,40));

handPane.setBounds(0, 580, 1280, 120);

handPane.setLayout(flow);

handPane.addMouseListener(mouseHandler);

handPane.addMouseMotionListener(mouseHandler);

graphicsPane.setBackground(Color.LIGHT\_GRAY);

graphicsPane.setBounds(-5500, -5500, 10000, 10000);

graphicsPane.setLayout(null);

graphicsPane.addMouseListener(mouseHandler);

graphicsPane.addMouseMotionListener(mouseHandler);

lbl\_points.setIcon(new ImageIcon("ImageSrc/BG\_Points.png"));

graphicsPane.add(paintingComponent);

paintingComponent.setSize(new Dimension(graphicsPane.getWidth(), graphicsPane.getHeight()));

/\*

\* Fenster anhand Bildschirmaufloesung zentriert ausrichten

\*/

Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();

Properties p = System.getProperties();

String os = p.getProperty("os.name");

textOut("Aufloesung: " + (int) screen.getWidth() + "x" + (int) screen.getHeight());

int x = (screen.width / 2) - (contentPane.getWidth() / 2);

int y = (screen.height / 2) - (contentPane.getHeight() / 2);

textOut(os);

if(os.toLowerCase().contains("mac"))

{

x = contentPane.getWidth() - screen.width;

y = contentPane.getHeight() - screen.height;

}

frame.setBounds(x, y, contentPane.getWidth(), contentPane.getHeight());

frame.setResizable(false);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

/\*\*

\* Zeichnet Ueberschneidungen farbig auf das Spielfeld

\*/

class PaintingComponent extends JComponent

{

private static final long serialVersionUID = 1L;

private Object[] intersections;

private Object[] intersectionColors;

/\*\*

\* Darf nicht vom Programmierer aufgerufen werden! Java ruft diese Methode bei bedarf selbst auf.

\*/

@Override

protected void paintComponent(Graphics g)

{

super.paintComponent(g); // Bevor gezeichnet wird, wird die Zeichenflaeche geleert.

Rectangle[] rects = new Rectangle[renderedImages.size()];

Shape[] shapes = new Shape[renderedImages.size()];

Graphics2D g2d = (Graphics2D) g;

int i = 0;

for (RenderImage ri: renderedImages)

{

double x = ri.getPos().getWidth();

double y = ri.getPos().getHeight();

double w = ri.getSize().getWidth();

double h = ri.getSize().getHeight();

rects[i] = new Rectangle((int)x, (int)y, (int)w, (int)h);

Path2D.Double path = new Path2D.Double();

path.append(rects[i], false);

g2d.setColor(Color.GREEN);

g2d.draw(shapes[i]);

if (i >= 1)

{

if (shapes[i].intersects(shapes[i-1].getBounds2D()));

{

g2d.setColor(Color.RED);

g2d.draw(shapes[i].getBounds().intersection(shapes[i-1].getBounds()));

}

}

i++;

}

if (intersections != null)

{

int z = 0;

for (Object s: intersections)

{

if ((Boolean) intersectionColors[z] == true)

g2d.setColor(Color.GREEN);

else

g2d.setColor(Color.RED);

g2d.fill((Shape)s);

z++;

}

}

g2d.dispose();

}

public void setIntersectionShapes (Object[] shapes, Object[] colors)

{

if (shapes != null)

{

this.intersections = shapes;

this.intersectionColors = colors;

}

else

{

this.intersections = null;

this.intersectionColors = null;

}

}

}

/\*\*

\* Schreibt den uebergebenen Text auf die Konsole

\* @param s - Der Text der ausgegeben werden soll

\*/

public void textOut(String s)

{

System.out.println(s);

}

/\*\*

\* Zeigt die Mausposition im Fenster an

\* @param x - Mausposition X

\* @param y - Mausposition Y

\*/

public void showMousePosition(int x, int y)

{

lbl\_mouseX.setText("Position X: " + x);

lbl\_mouseY.setText("Position Y: " + y);

lbl\_mouseX.setSize(lbl\_mouseX.getText().length() \* 6, 20);

lbl\_mouseY.setSize(lbl\_mouseY.getText().length() \* 6, 20);

}

/\*\*

\* Aktualisiert die Punkteanzeige

\* @param firstStone - true wenn der erste Stein gelegt wird

\*/

public void updatePoints(boolean firstStone)

{

int points = DominoRules.calculatePoints(lbl\_points.getPoints()

, lbl\_points.getDoublePoints(), firstStone);

lbl\_points.setText("Punkte: " + points);

}

/\*\*

\* Fuegt auf der handPane einen Stein zur Hand des Spielers hinzu

\* @param s - Der Dominostein, der hinzugefuegt werden soll

\* @param firstMove - Gibt an, ob es der erste Spielzug der Runde ist

\*/

public void addDominoeToHand(Stone s, boolean firstMove)

{

boolean abort = false;

// Hiermit wird sichergestellt, dass nicht versehentlich

// zwei identische Steine auf der Hand liegen koennen

for (DominoLabel h: handLabels)

{

if (h.getStone() == s)

abort = true;

}

// wenn der Stein noch nicht auf der Hand ist

if (!abort)

{

/\* wenn der Stein noch nicht gedreht ist,

aber gedreht werden muss wird er hier gedreht \*/

if (s.isDoublestone() && firstMove)

s.rotateImage(90);

// Fuegt den Stein zu Liste der angezeigten Steine hinzu

handLabels.add(new DominoLabel(s));

DominoLabel d = handLabels.get(handLabels.size() -1);

textOut("HandLabels Groesse: " + handLabels.size());

// fuegt dem Stein den MouseListener hinzu

d.addMouseListener(mouseHandler);

d.addMouseMotionListener(mouseHandler);

// fuegt den Stein zur Anzeige der handPane hinzu

handPane.add(d);

// Grafik aktualisieren

updatePanels();

}

}

/\*\*

\* Prueft ueberlappende Steine auf Ueberschneidungen

\* @param draggedStone - der Stein, der gelegt werden soll

\* @param released - true, wenn die Maustaste losgelassen wurde

\* @param edgePoints - die Kantenpunkte

\* @param doublePoints - haelt die Positionen der Doppelsteine

\* @return true - wenn es Ueberschneidungen gibt <br>

\* false - wenn nicht

\*/

public boolean checkIntersection(DominoLabel draggedStone,

boolean released, int[] edgePoints, boolean[] doublePoints)

{

ArrayList<Shape> intersections = new ArrayList<Shape>();

ArrayList<Boolean> intersectionColors = new ArrayList<Boolean>();

int i = 0;

int lastIndex = dLabels.size() - 1;

for (DominoLabel d: dLabels)

{

if (d == null || (i+1) > lastIndex)

break;

else

{

/\* prueft fuer alle gelegten Steine,

ob es Ueberschneidungen gibt\*/

for (int j = i+1; j < dLabels.size(); j++)

{

if (dLabels.get(i).getBounds().intersects

(dLabels.get(j).getBounds()))

{

intersections.add(dLabels.get(i).getBounds().intersection

(dLabels.get(j).getBounds()));

if (dLabels.get(j) == draggedStone)

{

targetLabel = dLabels.get(i);

intersectionColors.add(DominoRules.checkCompatibility

(dLabels.get(j), targetLabel));

}

else if (dLabels.get(i) == draggedStone)

{

targetLabel = dLabels.get(j);

intersectionColors.add(DominoRules.checkCompatibility

(dLabels.get(i), targetLabel));

}

else

intersectionColors.add(DominoRules.checkCompatibility

(dLabels.get(i), dLabels.get(j)));

}

/\* wenn der nächste Wert der ArrayList leer ist

soll es keinen weiteren Durchlauf geben \*/

if ((j+1) > lastIndex)

break;

}

}

i++;

}

// wenn es mindestens eine Uberschneidung zwischen 2 Steinen gibt

if (!intersections.isEmpty())

{

/\* hier werden die Ueberschneidungen zum zeichnen an

die paintingComponent uebergeben \*/

paintingComponent.setIntersectionShapes(intersections.toArray(),

intersectionColors.toArray());

/\* wenn es mehrere Ueberschneidungen gibt, kann auch

kein Stein angelegt worden sein \*/

if (intersections.size() > 1)

return false;

// wenn die Maustaste losgelassen wurde

if (released == true)

{

if (intersectionColors.get(0))

{

// wenn der Stein horizontal angelegt werden kann

if(!DominoRules.checkIfVertical(targetLabel))

return moveStoneHorizontal(draggedStone, targetLabel,

intersectionColors, edgePoints, doublePoints);

else

return moveStoneVertical(draggedStone, targetLabel,

intersectionColors, edgePoints, doublePoints);

}

}

return false;

}

// wenn es keine Ueberschneidungen gibt

else

{

targetLabel = null;

paintingComponent.setIntersectionShapes(null, null);

return false;

}

}

/\*\*

\* Gibt den Stein zurueck, mit dem es aktuell

\* eine Ueberschneidung gibt

\* @return

\*/

public DominoLabel getCurrentTarget()

{

return targetLabel;

}

/\*\*

\* Legt einen Stein horizontal an

\* @param draggedStone - der Stein, der gelegt werden soll

\* @param target - der Stein, an den angelegt werden soll

\* @param intersectionColors - Auswertung der Ueberschneidungen

\* @param edgePoints - die Punkte der offenen Kanten

\* @param doublePoints - weiß ob Punkte doppelt gezaehlt werden sollen

\* @return true - wenn der Stein angelegt wurde<br>

\* false - wenn nicht angelegt wurde

\*/

private boolean moveStoneHorizontal(DominoLabel draggedStone, DominoLabel target,

ArrayList<Boolean> intersectionColors, int[] edgePoints, boolean[] doublePoints)

{

boolean snapRight = DominoRules.snapRight(draggedStone, target);

boolean noNeighbours = DominoRules.checkNeighboursHorizontal(draggedStone, target, snapRight);

// wenn die beiden Steine kompatibel sind

if (intersectionColors.get(0) == true && draggedStone.isDraggable() && noNeighbours)

{

// Steinpositionen und Groessen holen

int tPosX = target.getLocation().x;

int tPosY = target.getLocation().y;

int draggedWidth = draggedStone.getWidth();

int draggedHeight = draggedStone.getHeight();

int targetWidth = target.getWidth();

int targetHeight = target.getHeight();

// wenn der Stein rechts angelegt werden soll

if (snapRight)

{

// wenn der Stein kein Doppelstein ist

if (!draggedStone.getStone().isDoublestone())

{

// wenn der Zielstein kein Doppelstein ist

if (!target.getStone().isDoublestone())

// Stein positionieren

draggedStone.setLocation(tPosX+draggedWidth, tPosY);

else

draggedStone.setLocation(tPosX+(draggedWidth/2),

tPosY+(draggedHeight/2));

}

// wenn der Stein ein Doppelstein ist

else

{

draggedStone.setLocation(tPosX+targetWidth, tPosY-(targetHeight/2));

}

// Nachbarn festlegen

target.getStone().setRightNeighbour(draggedStone.getStone());

draggedStone.getStone().setLeftNeighbour(target.getStone());

// Punkte berechnen

DominoRules.calculatePointsRight(draggedStone, target, edgePoints, doublePoints);

}

// wenn links angelegt werden soll

else

{

if (!draggedStone.getStone().isDoublestone())

{

if (!target.getStone().isDoublestone())

draggedStone.setLocation(tPosX-draggedWidth, tPosY);

else

draggedStone.setLocation(tPosX-draggedWidth, tPosY+(draggedHeight/2));

}

else

{

draggedStone.setLocation(tPosX-draggedWidth, tPosY-(targetHeight/2));

}

target.getStone().setLeftNeighbour(draggedStone.getStone());

draggedStone.getStone().setRightNeighbour(target.getStone());

DominoRules.calculatePointsLeft(draggedStone, target, edgePoints, doublePoints);

}

// beide Steine nicht verschiebbar machen

draggedStone.setNotDraggable();

target.setNotDraggable();

// Ueberschneidungen neu berechnen, um Grafikfehler zu vermeiden

checkIntersection(draggedStone, false, edgePoints, doublePoints);

// Punkte im Fenster anzeigen

lbl\_points.setPoints(edgePoints);

lbl\_points.setDoublePoints(doublePoints);

updatePoints(false);

return true;

}

// wenn nicht angelegt werden konnte

else

{

textOut("Die Steine sind leider nicht kompatibel");

return false;

}

}

/\*\*

\* Legt einen Stein horizontal an

\* @param draggedStone - der Stein, der gelegt werden soll

\* @param target - der Stein, an den angelegt werden soll

\* @param intersectionColors - Auswertung der Ueberschneidungen

\* @param edgePoints - die Punkte der offenen Kanten

\* @param doublePoints - weiß ob Punkte doppelt gezaehlt werden sollen

\* @return true - wenn der Stein angelegt wurde<br>

\* false - wenn nicht angelegt wurde

\*/

private boolean moveStoneVertical(DominoLabel draggedStone, DominoLabel target,

ArrayList<Boolean> intersectionColors, int[] edgePoints, boolean[] doublePoints)

{

// Kommentare aehnlich moveStoneVertical

boolean snapTop = DominoRules.snapTop(draggedStone, target);

boolean noNeighbours = DominoRules.checkNeighboursVertical(draggedStone, target, snapTop);

if (intersectionColors.get(0) == true && draggedStone.isDraggable() && noNeighbours)

{

int tPosX = target.getLocation().x;

int tPosY = target.getLocation().y;

int draggedWidth = draggedStone.getWidth();

int draggedHeight = draggedStone.getHeight();

int targetWidth = target.getWidth();

int targetHeight = target.getHeight();

if (snapTop)

{

if (!draggedStone.getStone().isDoublestone())

{

if(target.getStone().isSpinner() || !target.getStone().isDoublestone())

draggedStone.setLocation(tPosX, tPosY-targetHeight);

else

{

System.err.println("Normalen Stein vertikal oben an Doppelstein angelegt");

draggedStone.setLocation(tPosX+(draggedWidth/2), tPosY-draggedHeight);

}

}

else

{

System.err.println("Doppelstein oben angelegt");

draggedStone.setLocation(tPosX-(targetWidth/2), tPosY-targetWidth);

}

target.getStone().setTopNeighbour(draggedStone.getStone());

draggedStone.getStone().setBottomNeighbour(target.getStone());

DominoRules.calculatePointsTop(draggedStone, target, edgePoints, doublePoints);

}

else

{

textOut("Lege unten an");

if (!draggedStone.getStone().isDoublestone())

{

if(target.getStone().isSpinner() || !target.getStone().isDoublestone())

draggedStone.setLocation(tPosX, tPosY+targetHeight);

else

{

System.err.println("Normalen Stein vertikal unten an Doppelstein angelegt");

draggedStone.setLocation(tPosX+(draggedWidth/2), tPosY+targetHeight);

}

}

else

{

System.err.println("Doppelstein unten angelegt");

draggedStone.setLocation(tPosX-(targetWidth/2), tPosY+targetHeight);

}

target.getStone().setBottomNeighbour(draggedStone.getStone());

draggedStone.getStone().setTopNeighbour(target.getStone());

DominoRules.calculatePointsBottom(draggedStone, target, edgePoints, doublePoints);

}

draggedStone.setNotDraggable();

target.setNotDraggable();

checkIntersection(draggedStone, false, edgePoints, doublePoints);

lbl\_points.setPoints(edgePoints);

lbl\_points.setDoublePoints(doublePoints);

updatePoints(false);

return true;

}

else

{

textOut("Die Steine sind leider nicht kompatibel");

return false;

}

}

/\*\*

\* Aktualisiert die JPanels

\*/

public void updatePanels()

{

contentPane.updateUI();

graphicsPane.updateUI();

handPane.updateUI();

}

/\*\*

\* Gibt die Position des Hauptfensters auf dem Bildschirm zurueck

\* @return - Die absoluten Koordinaten des Ursprungs des Hauptfensters

\*/

public Point getFrameCoordinates()

{

return frame.getLocationOnScreen();

}

/\*\*

\* Legt einen Stein von der Hand aufs Spielfeld

\* @param clickedStone - der angeklickte Stein

\* @param offsetX - das X-Offset des Spielfeldes

\* @param offsetY - das Y-Offset des Spielfeldes

\* @param playedDominoes - die Anzahl der gelegten Steine

\*/

public void dropFromHand(DominoLabel clickedStone, int offsetX,

int offsetY, int playedDominoes)

{

Stone s = clickedStone.getStone();

Player p = s.getPlayer();

textOut("Stein: " + s);

// den Stein von der Hand des Spielers loeschen

p.deleteStone(s);

// den Stein vom JPanel Hand loeschen

handPane.remove(clickedStone);

handLabels.remove(clickedStone);

// den Stein dem Spielfeld hinzufuegen

dLabels.add(clickedStone);

graphicsPane.add(clickedStone, 0);

/\*wenn bereits Steine gelegt wurden,

wird der Stein etwas ueber der Hand dargestellt \*/

if (playedDominoes != 0)

clickedStone.setLocation(offsetX + clickedStone.getX(),

offsetY + 680 - clickedStone.getHeight());

/\* wenn noch keine Steine gelegt wurden, wird der Stein

mittig auf dem Spielfeld dargestellt\*/

else

{

int halfSizeX = frame.getWidth()/2;

int halfSizeY = scrollPane.getY()/2;

int halfStoneWidth = clickedStone.getWidth()/2;

int halfStoneHeight = clickedStone.getHeight()/2;

clickedStone.setLocation(offsetX + halfSizeX - halfStoneWidth, offsetY + halfSizeY - halfStoneHeight);

}

updatePanels();

}

/\*\*

\* Loescht auf dem GUI alle Steine von der Hand

\*/

public void clearHand()

{

handPane.removeAll(); // Alle Labels vom JPanel "Hand" leoschen

handLabels.clear(); // Alle Labels aus der Hand-ArrayList loeschen

updatePanels(); // Grafik auffrischen, um Aenderungen wirksam zu machen

}

/\*\*

\* Uebergibt beim ersten Zug Referenzen von edgePoints

\* und doublePoints zur Punkteberechnung und -darstellung

\* an das PunkteLabel. Danach ist keine weitere Uebergabe dieser

\* Variablen mehr nötig, die diese noch mit der Hauptspielklasse

\* referenziert sind

\* @param edgePoints - haelt die Punktzahlen der offenen Kanten

\* @param doublePoints - speichert, an welcher offenen Kante ein Doppelstein liegt

\*/

public void firstPoints(int[] edgePoints, boolean[] doublePoints)

{

lbl\_points.setPoints(edgePoints);

lbl\_points.setDoublePoints(doublePoints);

}

/\*\*

\* Kann je nach Uebergabeparameter den Button aktiv/inaktiv

\* schalten und den Text des Buttons aendern

\* @param isActive - <i>true</i> - Button aktiv,

\* <i>false</i> - Button inaktiv

\* @param text - Text, der auf dem Button dargestellt wird

\* (Standardtext, wenn <b>""</b>)

\*/

public void updateButton(boolean isActive, String text)

{

if (text != "")

btn\_drawTalon.setText(text);

btn\_drawTalon.setEnabled(isActive);

btn\_drawTalon.setFocusPainted(isActive);

}

/\*\*

\* Aktualisiert das Punktelabel nach dem Zug eines Spielers mit

\* dessen aktuellem Punktestand

\* @param player - Der Spieler, dessen Punkte aktualisiert werden sollen

\* @param playerIndex - Der Spielerindex des Spielers

\*/

public void updatePlayerPoints(Player player, int playerIndex)

{

if (playerIndex == 0)

lbl\_player1Points.setText("Spieler 1: " + player.getPoints());

else

lbl\_player2Points.setText("Spieler 2: " + player.getPoints());

}

/\*\*

\* Stellt den Pfeil neben der Punktzahl des aktuellen Spielers dar

\* @param playerIndex - Index des aktuellen Spielers

\* @param playedDominoes - Die Anzahl der bereits gelegten Steine

\*/

public void updatePlayerArrow(int playerIndex, int playedDominoes)

{

// Variable fuer das Icon erzeugen

ImageIcon arrow = null;

// Positionen der Labels holen

int x1 = lbl\_player1Points.getLocation().x;

int y1 = lbl\_player1Points.getLocation().y;

int x2 = lbl\_player2Points.getLocation().x;

int y2 = lbl\_player2Points.getLocation().y;

// das Bild laden

try

{

arrow = new ImageIcon("ImageSrc/Arrow.png");

} catch (Exception e) {

e.printStackTrace();

}

// wenn Spieler 1 der aktive Spieler ist

if (playerIndex == 0)

{

// wenn das Label kein Icon hat

if (lbl\_player1Points.getIcon() == null)

{

// Icon setzen und Position anpassen, da das

// Label durch das Icon nach rechts geschoben wird

lbl\_player1Points.setIcon(arrow);

lbl\_player1Points.setLocation(x1 - 24, y1);

// das Icon aus dem Label des andern Spielers entfernen

lbl\_player2Points.setIcon(null);

// wenn bereits Steine gelegt worden sind, muss das Label

// des anderen Spielers zurueck nach rechts geschoben werden

if (playedDominoes != 0)

lbl\_player2Points.setLocation(x2 + 24, y2);

}

}

// wenn Spieler 2 der aktive Spieler ist

else

{

if (lbl\_player2Points.getIcon() == null)

{

lbl\_player2Points.setIcon(arrow);

lbl\_player2Points.setLocation(x2 - 24, y2);

lbl\_player1Points.setIcon(null);

if (playedDominoes != 0)

lbl\_player1Points.setLocation(x1 + 24, y1);

}

}

}

/\*\*

\* Setzt alle relaventen Variablen im Fenster zurueck,

\* bevor die Runde neu gestartet wird

\*/

public void resetWindow()

{

// Punktelabel zuruecksetzen

lbl\_points.setText("Punkte: 0");

btn\_drawTalon.setText("Stein ziehen");

// Punktelabel der Spieler auf Ursprungskoordinaten zurueck schieben

lbl\_player1Points.setBounds(contentPane.getWidth() - 93, 45, 105, 20);

lbl\_player2Points.setBounds(contentPane.getWidth() - 93, 65, 105, 20);

// Icon aus Spielerlabels entfernen

lbl\_player1Points.setIcon(null);

lbl\_player2Points.setIcon(null);

// Alle DominoLabels aus der Steinliste loeschen

dLabels.clear();

clearHand();

// Spielfeld wieder an die Ursprungsposition schieben

graphicsPane.setLocation(-5500, -5500);

// Alle Labels vom Spielfeld loeschen

graphicsPane.removeAll();

// paintingComponent wieder hinzufuegen

graphicsPane.add(paintingComponent);

updatePanels(); // Grafik auffrischen, um Aenderungen wirksam zu machen

}

/\*\*

\* Zeigt den Dominoschriftzug an der zu sehen ist, wenn das Programm startet

\*/

public void showStartSplash()

{

// Button inaktiv machen

updateButton(false, "");

int fontSize1 = 0; // Anfangsschriftgroesse ist 0

lbl\_splash1.setVisible(true); // Label sichtbar machen

// Labeltext setzen

lbl\_splash1.setText("Let's play Domino");

// Schriftgroesse des Labeltextes auf Anfangsschriftgroesse setzen

lbl\_splash1.setFont(lbl\_splash1.getFont().deriveFont((float)fontSize1));

// Schleife von 0 bis 255 (RGB- und Alpha-Werte)

for (int i = 0; i <= 255; i++)

{

// Schriftfarbe des Labels anhand des Schleifenzaehlers aendern

lbl\_splash1.setForeground(new Color(255 - i,(i/2),(i/4),i));

// Variable fuer Schriftgroesse erhoehen, bis maximal 100

if (i % 3 == 0 && fontSize1 < 100)

fontSize1++;

// Schrift vergroessern

lbl\_splash1.setFont(lbl\_splash1.getFont().deriveFont((float)fontSize1));

delay(5); // Verzoegerung vor dem naechsten Schleifendurchlauf

}

delay(1500); // Zeit, die der Schriftzug in voller Groesse dargestellt wird

// umgekehrte Schleife fuer verkleinerung der Schrift

for (int i = 255; i >= 0; i--)

{

lbl\_splash1.setForeground(new Color(255 - i,(i/2),(i/4),i));

if (i % 3 == 0 && fontSize1 >= 0)

{

fontSize1--;

}

lbl\_splash1.setFont(lbl\_splash1.getFont().deriveFont((float)fontSize1));

delay(1); // laeuft schneller ab, da Verzoegerung geringer

}

// zum Schluss wird das Label ausgeblendet

lbl\_splash1.setVisible(false);

delay(300); // Verzoegerung, bevor die Methode verlassen wird

}

/\*\*

\* Stellt diverse Meldungen auf dem Bildschirm dar

\* @param text1 - Der Text, der in dem oberen Label stehen soll

\* @param text2 - Der Text, der in dem unteren Label stehen soll

\* @param textSize - die Textgroesse

\* @param delay - Die Zeit, die verstreichen soll, bevor die Meldung angezeigt wird (in Millisekunden)

\* @param fadeOut - Gibt an, ob die dargestellte Meldung wieder ausgeblendet werden soll

\*/

public void showGameInfo(final String text1, final String text2,

final float textSize, int delay, boolean fadeOut)

{

Timer t = new Timer(true); // Der Timer fuer das ein- bzw. ausblenden

// Die Aufgabe, die der Timer nach der Verzoegerung ausfuehren soll

t.schedule(new TimerTask()

{

@Override

public void run()

{

// Textgroesse setzen

lbl\_splash1.setFont(lbl\_splash1.getFont().deriveFont(textSize));

lbl\_splash2.setFont(lbl\_splash2.getFont().deriveFont(textSize));

// Textfarbe setzen

lbl\_splash1.setForeground(Color.BLACK);

lbl\_splash2.setForeground(Color.BLACK);

// Text setzen

lbl\_splash1.setText(text1);

lbl\_splash2.setText(text2);

// Label anzeigen

lbl\_splash1.setVisible(true);

lbl\_splash2.setVisible(true);

}

}, delay);

// wenn die Meldung ausgeblendet werden soll

if (fadeOut)

{

t.schedule(new TimerTask()

{

@Override

public void run()

{

// Label nach Verzoegerung wieder ausblenden

lbl\_splash1.setVisible(false);

lbl\_splash2.setVisible(false);

}

}, delay + 2000);

}

}

/\*\*

\* Laesst den Thread fuer die angebene Zeit pausieren, um eine

\* Verzoegerung zu erreichen (nur fuer den Splashscreen zu Beginn des Spiels)

\* @param milliseconds - Die Zeit in Millisekunden, die der Thread schlafen soll

\*/

public void delay(int milliseconds)

{

try

{

TimeUnit.MILLISECONDS.sleep(milliseconds);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}package view;

import javax.swing.JLabel;

/\*\*

\* Ein JLabel, dass die Punkte haelt

\*/

public class PointsLabel extends JLabel

{

private static final long serialVersionUID = 1L;

/\*\* die KantenPunkte (Referenzen aus DominoGame) \*/

private int[] points = new int[4];

/\*\* Array dass speichert, an welchen

Kanten Doppelsteine liegen (Referenzen aus DominoGame) \*/

private boolean[] doublePoints = new boolean[4];

public PointsLabel(String string)

{

this.setText(string);

}

public int[] getPoints()

{

return points;

}

public void setPoints(int[] points)

{

if (points.length == 4)

{

this.points = points;

}

}

public boolean[] getDoublePoints()

{

return doublePoints;

}

public void setDoublePoints(boolean[] doublePoints)

{

if (doublePoints.length == 4)

{

this.doublePoints = doublePoints;

}

}

}