



Silas Hoffmann, inf103088  
3. Fachsemester  
3. Verwaltungssemester

25. Januar 2019

Dokumentation

## **Programmierpraktikum**

im Wintersemester 2018/19

Dozent: Prof. Dr. Andreas Häuslein

Fachbereich Informatik  
Fachhochschule Wedel

# Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>I. Allgemeine Problemstellung</b>           | <b>5</b>  |
| <b>1. Erklärung des Spiels</b>                 | <b>5</b>  |
| 1.1. Spielregeln . . . . .                     | 5         |
| 1.2. Anlegeregeln . . . . .                    | 5         |
| 1.3. Spielende . . . . .                       | 6         |
| <b>2. Implementierungsdetails</b>              | <b>6</b>  |
| 2.1. KI . . . . .                              | 6         |
| 2.2. Oberfläche . . . . .                      | 7         |
| 2.3. Karten . . . . .                          | 7         |
| 2.4. Log . . . . .                             | 8         |
| 2.5. Spielstand . . . . .                      | 8         |
| <b>II. Benutzerhandbuch</b>                    | <b>10</b> |
| <b>3. Ablaufbedingungen</b>                    | <b>10</b> |
| <b>4. Programminstallation / Programmstart</b> | <b>10</b> |
| <b>5. Bedienungsanleitung</b>                  | <b>11</b> |
| 5.1. Hintergrundinformationen . . . . .        | 11        |
| 5.2. Programmfunktionalität . . . . .          | 16        |
| <b>III. Programmierhandbuch</b>                | <b>21</b> |
| <b>6. Entwicklungskonfiguration</b>            | <b>21</b> |
| <b>7. Problemanalyse und Realisation</b>       | <b>22</b> |
| 7.1. Problemanalyse . . . . .                  | 22        |
| 7.2. Realisationsanalyse . . . . .             | 23        |
| <b>8. Beschreibung grundlegender Klassen</b>   | <b>26</b> |
| 8.1. gui . . . . .                             | 27        |
| 8.2. token . . . . .                           | 29        |
| 8.3. dataPreservation . . . . .                | 34        |
| 8.4. bankSelection . . . . .                   | 38        |
| 8.5. playerState . . . . .                     | 41        |
| 8.6. differentPlayerTypes . . . . .            | 49        |
| 8.7. logicTransfer . . . . .                   | 53        |

|                                     |           |
|-------------------------------------|-----------|
| <b>9. Programmorganisationsplan</b> | <b>65</b> |
| <b>10. Programmtests</b>            | <b>68</b> |
| <b>A. Anhang</b>                    | <b>ii</b> |
| A.1. Entwicklung der Gui . . . . .  | ii        |
| A.2. Bildquellen . . . . .          | vi        |

## Literatur

- [1] City Domino Aufgabenstellung. <http://intern.fh-wedel.de/mitarbeiter/klk/programmierpraktikum-java/aufgabetermine/ss18-citydomino/>. Aufgerufen am: 29-12-2018.
- [2] try-with-resources-Block Erklaerung. <https://www.baeldung.com/java-exceptions>. Unterpunkt 4.4, Aufgerufen am: 03-01-2019.
- [3] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Entwurfsmuster von Kopf bis Fuß* -. O'Reilly Germany, Köln, 1. aufl. edition, 2006.

## Abbildungsverzeichnis

|     |  |     |
|-----|--|-----|
| 1.  | Erstes Selektieren . . . . .   | 12  |
| 2.  | Schwebender Domino über gültiger Position . . . . .                  | 12  |
| 3.  | Schwebender Domino über ungültiger Position . . . . .                | 13  |
| 4.  | Menüoptionen . . . . .   | 14  |
| 5.  | Nachträgliches Abspeichern . . . . .                                 | 14  |
| 6.  | Filechooser . . . . .  | 15  |
| 7.  | Unterschiedliche Fehlermeldungen beim Einlesen einer Datei . . . . . | 16  |
| 8.  | Spielbeginn . . . . .  | 17  |
| 9.  | Initiales Selektieren - oben . . . . .                               | 17  |
| 10. | Initiales Selektieren - mittig . . . . .                             | 18  |
| 11. | Erstes Rotieren . . . . .  | 19  |
| 12. | Erste Ablage . . . . .   | 19  |
| 13. | Auswahlfenster . . . . .   | 25  |
| 14. | UML-Darstellung des token packages . . . . .                         | 29  |
| 15. | UML-Darstellung des dataPreservation packages . . . . .              | 34  |
| 16. | UML-Darstellung des token packages . . . . .                         | 39  |
| 17. | UML-Darstellung des playerState packages . . . . .                   | 42  |
| 18. | UML-Darstellung des differentPlayerTypes packages . . . . .          | 49  |
| 19. | Vereinfachte UML-Darstellung des gesamten Projekts . . . . .         | 65  |
| 20. | Testcoverage . . . . .   | 71  |
| 21. | Gui-Entwicklung 1 . . . . .  | ii  |
| 22. | Gui-Entwicklung 2 . . . . .  | iii |
| 23. | Gui-Entwicklung 3 . . . . .  | iii |
| 24. | Gui-Entwicklung 4 . . . . .  | iv  |
| 25. | Gui-Entwicklung 5 . . . . .  | iv  |
| 26. | Gui-Entwicklung 6 . . . . .  | v   |
| 27. | Gui-Entwicklung 7 . . . . .  | v   |
| 28. | Gui-Entwicklung 8 . . . . .  | vi  |

# Teil I.

## Allgemeine Problemstellung

Zu implementieren ist ein Dominospiel, bei dem vier Spieler jeweils ihre eigene Stadt gestalten. Ziel des Spiels ist es, möglichst viele Stadtteile mit Prestige zu gestalten. [1]

### 1. Erklärung des Spiels

#### 1.1. Spielregeln

Jeder Spieler besitzt ein eigenes 5\*5-Zellen großes Spielfeld und legt zu Beginn sein Stadtzentrum mittig ab. Ein Spielbeutel für alle Spieler enthält 48 Spielkarten in der Größe von zwei Zellen, die auf ihren zwei Hälften jeweils einen (evtl. auch den gleichen) Stadtteiltyp anzeigen. Die Stadtteiltypen unterscheiden sich durch Bild und Hintergrundfarbe voneinander. Jede Spielkarte besitzt eine definierte Wertigkeit. Auf manchen Stadtteilen sind zusätzlich ein bis drei Prestigesymbole abgebildet. Es werden vier Karten gezogen und im ersten Auswahlbereich angezeigt. Dabei wird die niederwertigste Karte zuoberst, die höchstwertigste zuunterst einsortiert. Der erste Spieler markiert die Karte im Auswahlbereich, die er gerne nehmen würde, die anderen Spieler treffen ihre Auswahl der Reihe nach ebenfalls und markieren die jeweils gewünschte Karte. Wurden alle Karten markiert, dann werden wieder vier Karten gezogen und ebenso sortiert im zweiten Auswahlbereich angezeigt.

**Spielablauf** Derjenige, der die oberste Karte im ersten Auswahlbereich markiert hat, beginnt eine Runde, es folgen der Reihe nach die Spieler, die die jeweils darunterliegende Karte markiert haben. In einer Runde wird zunächst eine Karte aus dem zweiten Auswahlbereich markiert und dadurch für die kommende Runde gewählt. Je wertvoller also seine markierte Karte in dieser Runde ist, desto später ist der Spieler am Zug und desto weniger Auswahl hat er für die kommende Runde.

#### 1.2. Anlegeregeln

Die erste Karte muss an das Stadtzentrum angrenzen. An das Stadtzentrum darf jeder Stadtteil angrenzen. Legt man eine Karte an eine andere Karte an, so muss mindestens eine Hälfte mit einer Seite an einen identischen Stadtteiltyp einer liegenden Karte angrenzen. Passt die abzulegende Karte weder an das Stadtzentrum noch an eine bereits ausliegende Karte, so wird sie verworfen. Alle Spielkarten müssen in das 5\*5-Feld passen, keine Hälfte darf hinausragen. Das Stadtzentrum muss aber nicht in der Mitte liegen, sondern kann im Spielverlauf verschoben werden, wodurch sich alle bereits gelegten Karten mit verschieben. Eine abgelegte Karte kann nicht verschoben werden.

### 1.3. Spielende

Wurden alle Spielkarten aus dem Beutel gezogen und von den Auswahlbereichen auf die Spielfelder platziert bzw. verworfen, werden die Punkte ermittelt.

- Jede Stadt besteht aus mehreren Stadtteilen. Ein Stadtteil setzt sich aus waagerecht und/oder senkrecht verbundenen Zellen desselben Stadtteiltyps zusammen. Das Stadtzentrum zählt zu keinem Stadtteil dazu.
- Die Punkte eines Stadtteils ergeben sich aus der Anzahl seiner Zellen multipliziert mit der Anzahl darin enthaltener Prestigesymbole.
- Innerhalb einer Stadt kann es mehrere voneinander getrennte Stadtteile desselben Typs geben. Jeder Stadtteil ist einzeln auszuwerten.
- Stadtteile ohne Prestigesymbole bringen keine Punkte.

Für die Auswertung wird für jeden Spieler die Summe der Punkte seiner Gebiete ermittelt. Gewonnen hat der Spieler mit den meisten Punkten. Bei einem Gleichstand gewinnt der Spieler mit dem größten einzelnen Gebiet. Besteht auch hier Gleichstand, so siegen beide Spieler gleichermaßen.

## 2. Implementierungsdetails

### 2.1. KI

Außer dem menschlichen Spieler, der im Spiel stets beginnt, existieren 3 computergesteuerte Spieler. Diese sollen einer sehr primitiven Logik folgen:

- bei der Auswahl wird die Karte markiert, mit der bei Auslage im eigenen Feld aktuell am meisten Punkte erzielt werden könnten
- dafür wird für jede freie Karte der Auswahlbank auf jeder freien Position des Spielfeldes und in jeder Rotation der Punktgewinn ermittelt
- bei Punktgleichheit mehrerer Positionen wird darauf geachtet, dass keine leeren Einzelzellen erzeugt werden bei der Ablage wird die so ermittelte Position genutzt
- die mögliche Verschiebung des Stadtzentrums wird nicht durchgeführt

Wer möchte, kann zusätzlich intelligentere KIs implementieren, die z.B. das Stadtzentrum verschieben, die Kartenwahl der kommenden Runde einbeziehen, verhindern, dass andere Spieler viele Punkte erhalten, oder Schlüsse aus den bereits abgelegten Karten ziehen.

## 2.2. Oberfläche

Existieren müssen folgende Elemente:

- ein Spielfeld für den menschlichen Spieler
- ein erster Auswahlbereich für die aktuelle Runde
- ein zweiter Auswahlbereich für die kommende Runde
- das Legen einer Karte auf das Spielfeld per Drag and Drop, gültige Ablegepositionen werden beim DragOver sichtbar markiert
- eine Möglichkeit, um die Karte zu verwerfen. Das kann z.B. ein Button sein, der zum Verwerfen der aktuellen Karte betätigt wird, oder auf den die aktuelle Karte gezogen wird. Verworfene Karten müssen nicht angezeigt werden.
- die Spielfelder der drei KI-Spieler, so dass die dort abgelegten Karten jederzeit erkennbar sind.

Mögliche Lösungen für das Legen einer Karte:

- Die aktuell zu legende Karte kann in einer Drehbox erscheinen, in der die Karte durch einfaches Anklicken gedreht werden kann. Hat sie die gewünschte Orientierung erreicht, so kann die Karte per Drag and Drop auf das eigene Spielfeld gelegt oder verworfen werden.
- Die aktuell gedragte Karte wird durch Tastendruck unter dem Mauscursor gedreht.

Die Reihenfolge der Spieler muss erkennbar sein. Es muss also zugeordnet werden können, welches der angezeigten Spielfelder zu welcher Kartenauswahl im Auswahlbereich gehört (z.B. durch gleiche Symbole oder farbliche Markierungen an beiden Stellen). Das Stadtzentrum eines Spielfeldes muss zusammen mit allen bereits gelegten Karten verschoben werden können, entweder per Drag and Drop oder beispielsweise durch Buttons am Spielfeldrand. Dabei dürfen keine Stadtteile aus dem Spielfeld geschoben werden. Die Auswertung eines Spiels muss für jeden Spieler die erreichten Punkte pro Stadtteiltyp darstellen. Die Bedienung des Spiels muss intuitiv möglich sein für jemanden, der die Spielregeln kennt. Die Größe des Fensters darf zu Spielbeginn höchstens 1600 \* 900 Pixel betragen.

## 2.3. Karten

Die für ein Spiel vorhandenen Karten sind in dieser Datei definiert. Die zu den Stadtteiltypen gehörigen Bilder findet man hier. Pro Zeile wird eine Karte mit ihren beiden Hälften und ihrem Wert festgelegt: <Art><Symbolanzahl>,<Art><Symbolanzahl>,<Wert> Art ist dabei der Anfangsbuchstabe eines Stadtteiltyps (Amusement, Industry, Office, Park, Shopping, Home), die Symbolanzahl eine Ziffer von 0 bis 3. Eine mögliche Zeile wäre also *H1,P0,24* für eine Karte mit einem Symbol auf einem Haus und ohne Symbol in einem Park und einem Wert von 24 Punkten.

## 2.4. Log

In einer Datei (gleichzeitig auch auf System.out) sind durchgeführte Aktionen zu protokollieren. Der zuerst angegebene Stadtteil einer Karte ist dabei immer der an der angegebenen Position, bei horizontaler Ausrichtung liegt der zweite Stadtteil rechts davon, bei vertikaler darunter. Beispiel:

```
BOT1 chose [H1, P0] at index 1 for next round
BOT1 put [A0, P2] horizontally to (1, 2)

HUMAN chose [P0, S0] at index 0 for next round
HUMAN dragged center to (2, 3)
HUMAN put [A0, A0] vertically to (0, 0)

BOT3 chose [O0,I2] at index 3 for next round
BOT3 did not use [O0, A1]
```

## 2.5. Spielstand

Der aktuelle Spielstand soll gespeichert und geladen werden können. Laden/Speichern soll nur möglich sein, wenn der menschliche Spieler am Zug ist. Eine Spielstandsdatei enthält die 4 Spielfelder der Spieler in ihrer Reihenfolge (das erste Feld gehört immer dem menschlichen Spieler 0), die zwei Auswahlbereiche und die im Beutel verbliebenen Karten mit folgenden Bereichen:

```
<Spielfeld>
<Spielfeld>
<Spielfeld>
<Spielfeld>
<Bänke>
<Beutel>
```

Die einzelnen Bereiche enthalten jeweils eine Einführungszeile (einen Kommentar) gefolgt von Inhaltsangaben:

- Ein Spielfeld enthält einen Kommentar, zu wem es gehört, und in Folge für jede Zelle eine Inhaltsangabe:
  - ‘-’ für eine nicht belegte Zelle,
  - ‘CC’ für das Stadtzentrum und
  - zwei Buchstaben für eine Hälftenbeschreibung.
- Die Zellen sind durch Leerzeichen separiert.

Beispiel:

```
<Spielfeld>
-- -- -- -- --
-- -- H1 P0 --
-- -- CC -- --
-- -- -- -- --
-- -- -- -- --
```

- Die Bänke enthalten Angaben für die aufliegenden Karten und von wem diese bereits gewählt wurde. Die erste Bank kann weniger als vier Karten enthalten, in entsprechender Anzahl enthält die zweite Bank dann bereits Markierungen (sonst '-' für fehlende Markierung). Die erste Bank wird als erste Markierung immer Spieler 0 (den menschlichen Spieler) aufweisen. Beispiel:

```
<Bänke>
0 H1P0,2 P001,3 I1P0
- POPO,- AOA0,1 HOAO,- P1HO
```

- Der Beutel enthält alle im Beutel befindlichen Karten kommasepariert. Im folgenden Beispiel befinden sich nur noch 4 Karten im Beutel

```
<Beutel>
POPO,POPO,A1H0,I2P0
```

# Teil II.

# Benutzerhandbuch

## 3. Ablaufbedingungen

Überblick über die benötigten Hardware und Softwarekomponenten, die für die Ausführung des kompilierten Programms benötigt werden.

| Softwarekomponenten      |              |
|--------------------------|--------------|
| Name                     | Version      |
| Java Runtime Environment | jdk1.8.0_131 |

Um den selben Font nutzen zu können wie er in den Screenshots zu sehen ist, muss ein Font namens „Papyrus“ installiert werden (kann hier heruntergeladen werden: <https://www.wfonts.com/font/papyrus>). Wird dies nicht getan, wird im Regelfall der normale Standardfont sämtlicher FXML-Dokumente bzw. Programme verwendet.

## 4. Programminstallation / Programmstart

Die gegebene *.jar*-Datei muss nicht entpackt werden. Bei Programmstart wird allerdings ein Ordner im selben Verzeichnis wie die Datei angelegt. Dieser Ordner heißt “dataOutput“ und beinhaltet die Log-Datei. Dieser Ordner dient außerdem als Startpunkt beim Öffnen eines Filechoosers beim Ein-/Auslesen einer Datei. Zum Starten reicht es per doppeltem Mausklick die *.jar*-Datei auszuwählen.

## 5. Bedienungsanleitung

### 5.1. Hintergrundinformationen

#### Spielinteraktion

**Selektierungsvorgang** Um einen gewünschten Domino zu selektieren, muss der Spieler auf einen der schwarzen Kästen rechts neben dem angezeigten Domino per Mausklick auswählen (siehe Abbildung 1). Anschließend erscheint die Zahl 1 in diesem Feld. Diese Zahl repräsentiert den menschlichen Spieler. Um dem Benutzer deutlich zu machen von welcher Bank, oder ob er überhaupt in seinem aktuellen Zug einen Domino selektieren darf, kann dieser nur auf der Bank, welche nicht verschwommen ist, einen Domino auswählen. Um jederzeit ablesen zu können welcher Spieler gerade am Zug ist, gibt es hierfür ein Feld oberhalb der Darstellung der beiden Bänke eine Anzeige.

**Justierung des Dominos** Nachdem der Spieler erfolgreich sämtliche Selektierungsschritte auf den beiden Bänken absolviert hat, kann er seinen zuvor ausgewählten Domino in dem dafür vorgesehenen Kasten drehen. Um den Domino um 90 Grad zu drehen muss der Spieler lediglich einen Mausklick auf dem Domino ausführen.

**Positionierung auf dem Spielfeld** Um den justierten Domino nun auf dem Feld zu platzieren, zieht der Benutzer den Domino an die gewünschte Stelle auf dem Spielfeld. Während dem Ziehen färben sich die zugrunde liegenden Felder jeweils grün, falls es möglich sein sollte den Domino an der gewünschten Stelle anzulegen (siehe Abbildung 12), beziehungsweise rot, falls dies nicht der Fall sein sollte (siehe Abbildung 3. Für genauere Informationen siehe Abschnitt 5.2). Falls der Domino an der gewünschten Stelle nicht passen sollte und dennoch versucht wird ihn dort zu platzieren, passiert nichts. Der Domino befindet sich weiterhin in dem Kasten zum Justieren der Ausrichtung und es kann ein neuer Versuch unternommen werden.

**Verwerfen des Dominos** Um den Domino zu verwerfen reicht es per Mausklick einmal auf das Müllimer-Symbol rechts neben dem Domino zu klicken. Der Domino wird anschließend aus dem Rotationsfeld entfernt.

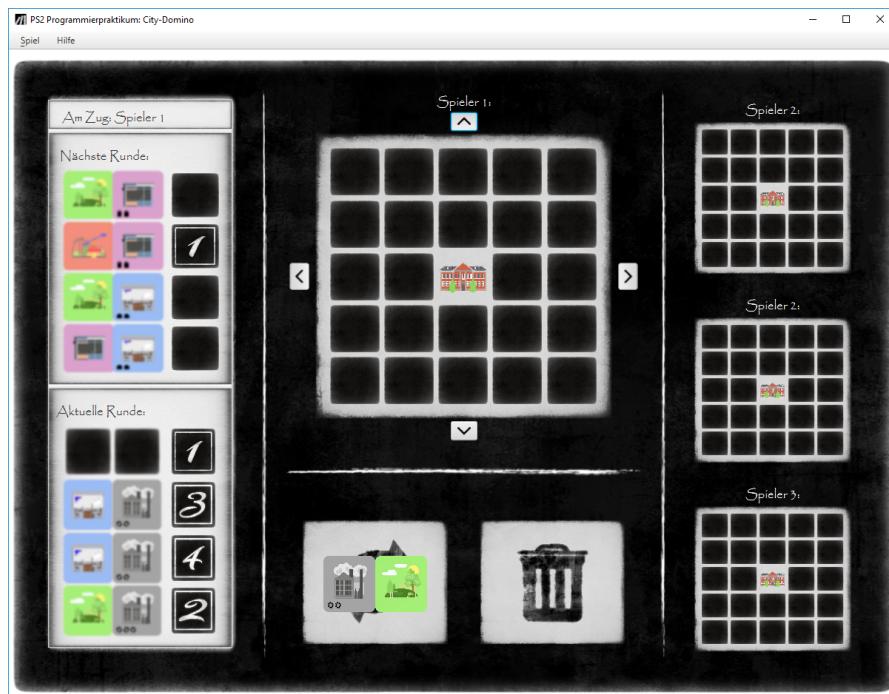


Abbildung 1: Erstes Selektieren auf der Bank für die nächste Runde

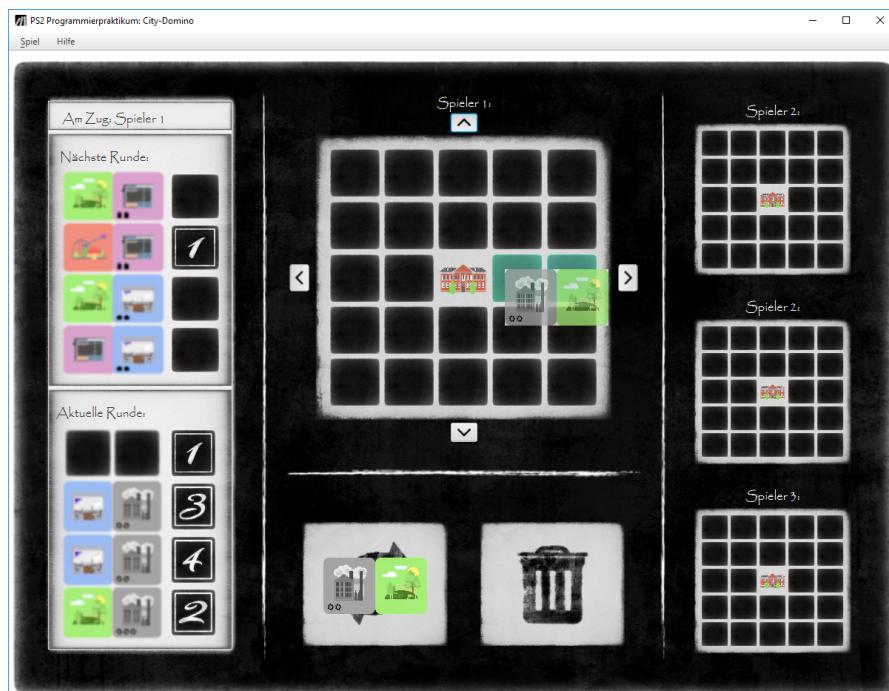


Abbildung 2: Schwebender Domino über gültiger Position

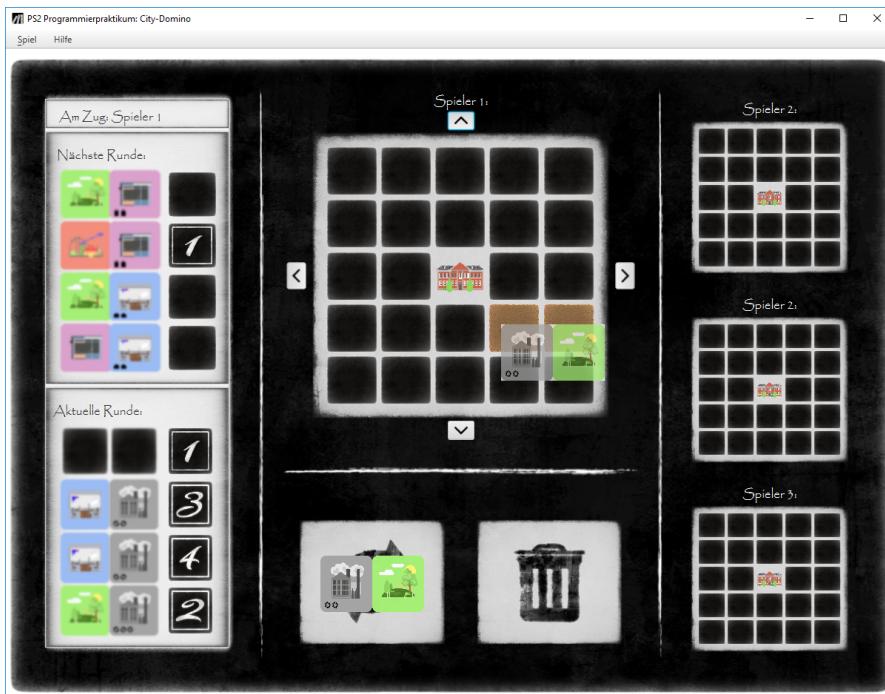


Abbildung 3: Schwebender Domino über ungültiger Position

## Menüinteraktion

**Starten bzw. Schließen** Um ein neues Spiel zu starten wählt der Benutzer den Menüpunkt "Neues Spiel". Alternativ ist dies auch per Tastenkombination **strg + N** möglich. Um das geöffnete Fenster zu schließen und das bestehende Spiel zu verwerfen, wählt der Benutzer den Reiter *Beenden* (Tastenkombination **strg + E**). Falls der Benutzer das Spiel nicht vorher gespeichert hat erscheint hierbei ein weiteres Fenster, welches den Benutzer darauf hinweist und ihm die Möglichkeit gibt dies nachzuholen (siehe folgendem Abschnitt). Möchte er fortfahren, ohne den aktuellen Spielstand zu speichern, muss der Button mit der Aufschrift *Abbrechen* betätigt werden (siehe Abbildung 5).

**Spielstand abspeichern** Um einen Spielstand zu speichern gibt es zwei Reiter mit folgenden Möglichkeiten:

1. Speichern: **Strg + S**
2. Speichern unter: **Strg + Shift + S**

„*Speichern unter*“ gibt dem Benutzer die Möglichkeit, ausgehend vom Ablageverzeichnis, den gewünschten Speicherort anzugeben. Hierzu navigiert er mit dem gegebenen Filechooser an den gewünschten Speicherort und gibt der abzuspeichernden Datei einen Namen. Hierbei wurde die benötigte Dateiendung *.txt* bereits ausgewählt, sodass der

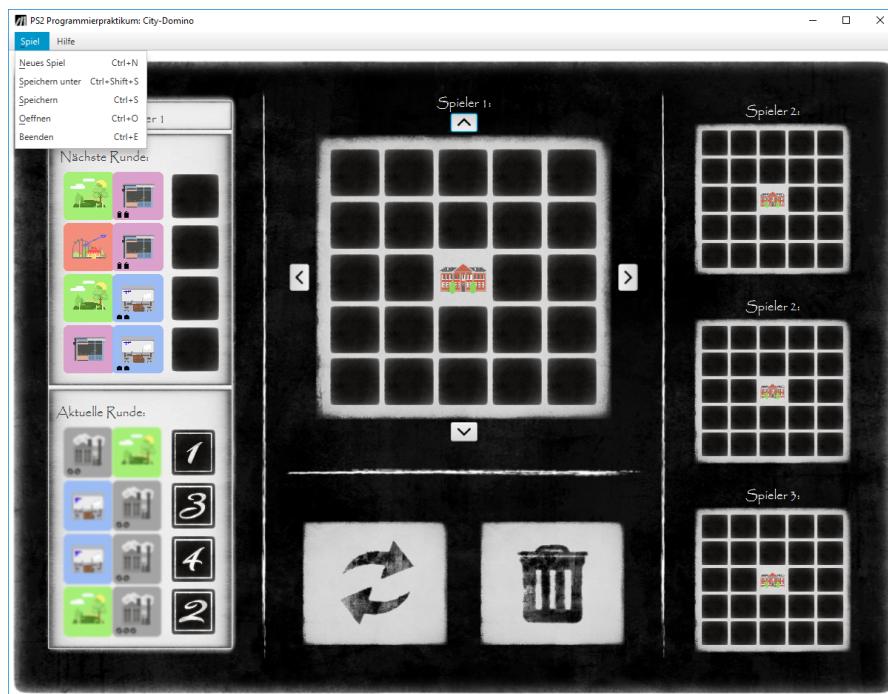


Abbildung 4: Menüoptionen

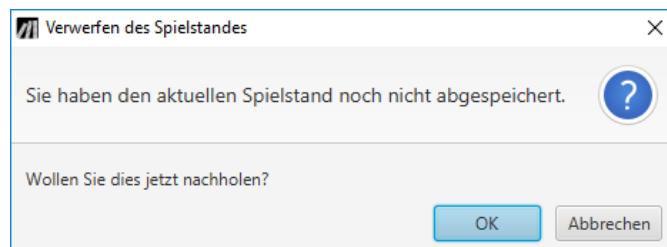


Abbildung 5: Nachträgliches Abspeichern

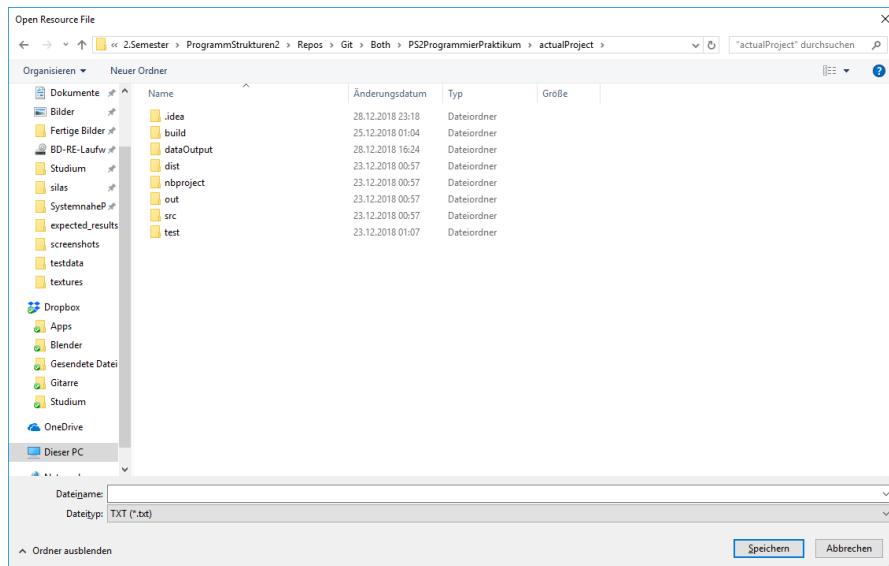


Abbildung 6: Filechooser zum Abspeichern eines Spielstandes

Benutzer seine Auswahl lediglich auf dem Feld *Speichern* per Mausklick zu bestätigen braucht (siehe Abbildung 6).

Der Reiter „*Speichern*“ ermöglicht es einen bereits gespeicherten Spielstand ohne Öffnen eines Filechoosers zu überschreiben. Falls der Benutzer diesen Reiter betätigt, ohne dass zuvor ein Spielstand des aktuellen Spiels abgespeichert worden ist, öffnet sich bei der Auswahl dieses Reiters dennoch ein Filechooser und es wird nach der Funktionsweise von „*Speichern unter*“ vorgegangen.

**Öffnen** Ähnlich wie beim Reiter „*Speichern unter*“ wird hier ein Filechooser geöffnet. Dieser wird jedoch dazu verwendet eine Datei auszuwählen um aus dieser einen Spielstand zu lesen. Falls die Datei nicht der geforderten Syntax entspricht, erscheint eine Fehlermeldung in Form eines Popup-Fensters mit einer groben Fehlermeldung wo der Fehler liegt (siehe Listing 7, S. 16, Unterschiedliche Fehlermeldungen beim Einlesen einer Datei). Falls der Benutzer vor dem Öffnen eines neuen Spielstands den alten nicht gespeichert hat, wird er ähnlich wie beim Beenden darauf hingewiesen und es wird per Filechooser eine Möglichkeit bereitgestellt dies nachzuholen. Falls eine fehlerhafte Datei geöffnet wurde, wird dies dem Benutzer wieder per Pop-Up-Fenster mitgeteilt. Allerdings muss anschließend ein valides Spiel geöffnet oder ein neues Spiel gestartet werden. Der Benutzer kann also nicht einfach das gerade laufende Spiel weiterspielen.

**Hilfestellung** Unter dem Reiter „*Hilfe*“ ist der Link zur Website der Aufgabenstellung zu finden. Beim Auswählen des Menüpunktes *Aufgabenstellung* öffnet sich diese im, vom Benutzer standardmäßig genutzten, Browser.

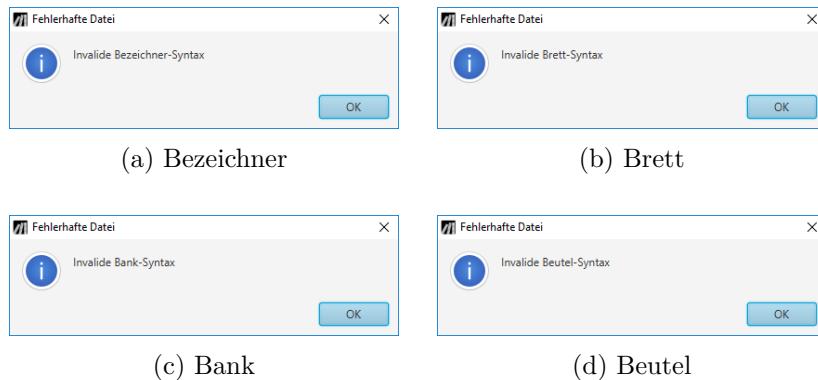


Abbildung 7: Unterschiedliche Fehlermeldungen beim Einlesen einer Datei

## 5.2. Programmfunktionalität

Generell gilt es zwischen einem standardmäßig ausgeführten Spiel und einem aus einer Datei eingelesenen Spiel zu unterscheiden.

### Spiel ohne Einlesen einer Datei

**Spielbeginn** Ein Spielbeutel für alle Spieler enthält 48 Spielkarten in der Größe von zwei Zellen, die auf ihren zwei Hälften jeweils einen (evtl. auch den gleichen) Stadtteiltyp anzeigen. Die Stadtteiltypen unterscheiden sich durch Bild und Hintergrundfarbe voneinander. Jede Spielkarte besitzt eine definierte Wertigkeit. Auf manchen Stadtteilen sind zusätzlich ein bis drei Prestigesymbole abgebildet. Jeder Spieler besitzt ein eigenes 5\*5-Zellen großes Spielfeld und legt zu Beginn sein Stadtzentrum mittig ab [1]. (siehe Abbildung 8). Dies wird bereits vom Spiel übernommen, sodass der erste Spielzug des Benutzers das initiale Selektieren vom ersten Auswahlbereich (hier mit *Aktuelle Runde* gekennzeichnet) darstellt. Um einen Domino selektieren zu können werden vier Karten gezogen und im ersten Auswahlbereich angezeigt. Dabei wird die niedrigwertigste Karte zuoberst, die höchswertigste zuunterst einsortiert. Der erste Spieler markiert die Karte im Auswahlbereich, die er gerne nehmen würde, die anderen Spieler treffen ihre Auswahl der Reihe nach ebenfalls und markieren die jeweils gewünschte Karte. Wurden alle Karten markiert, dann werden wieder vier Karten gezogen und ebenso sortiert im zweiten Auswahlbereich angezeigt. [1] (Siehe Abbildung 9)

**Spielablauf** Derjenige, der die oberste Karte im ersten Auswahlbereich markiert hat, beginnt eine Runde, es folgen der Reihe nach die Spieler, die die jeweils darunterliegende Karte markiert haben. In einer Runde wird zunächst eine Karte aus dem zweiten Auswahlbereich markiert und dadurch für die kommende Runde gewählt. Je wertvoller also seine markierte Karte in dieser Runde ist, desto später ist der Spieler am Zug und desto weniger Auswahl hat er für die kommende Runde. [1] Zwischen dem initialen Selektieren und dem beschriebenen Spielablauf gibt es keine Pause. Wie man in Abbildung 10 sehen kann, ziehen die Gegner bereits ihre ersten Dominos auf ihr Feld, obwohl der Benutzer

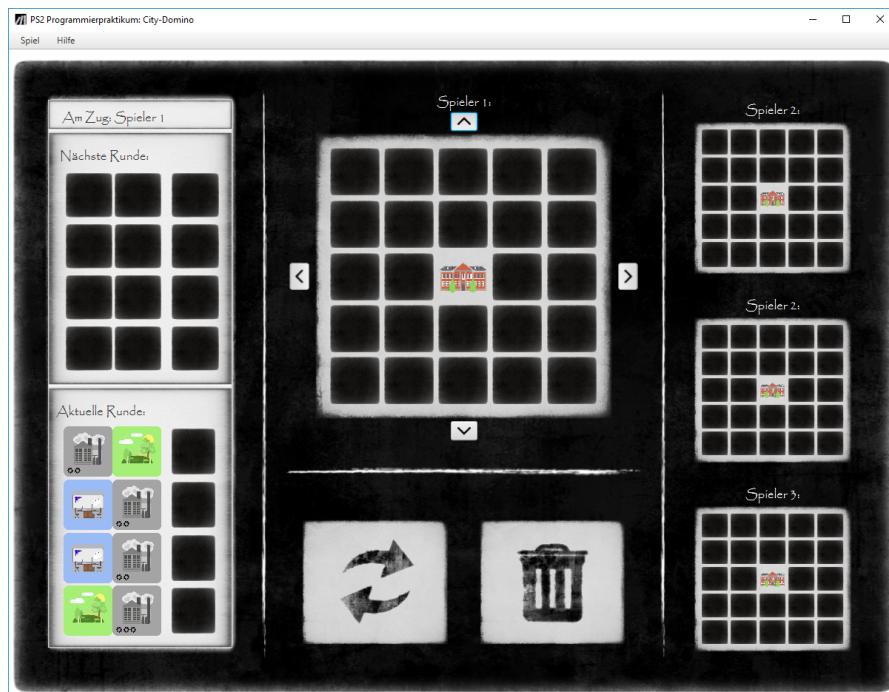


Abbildung 8: Spielbeginn nach Programmstart

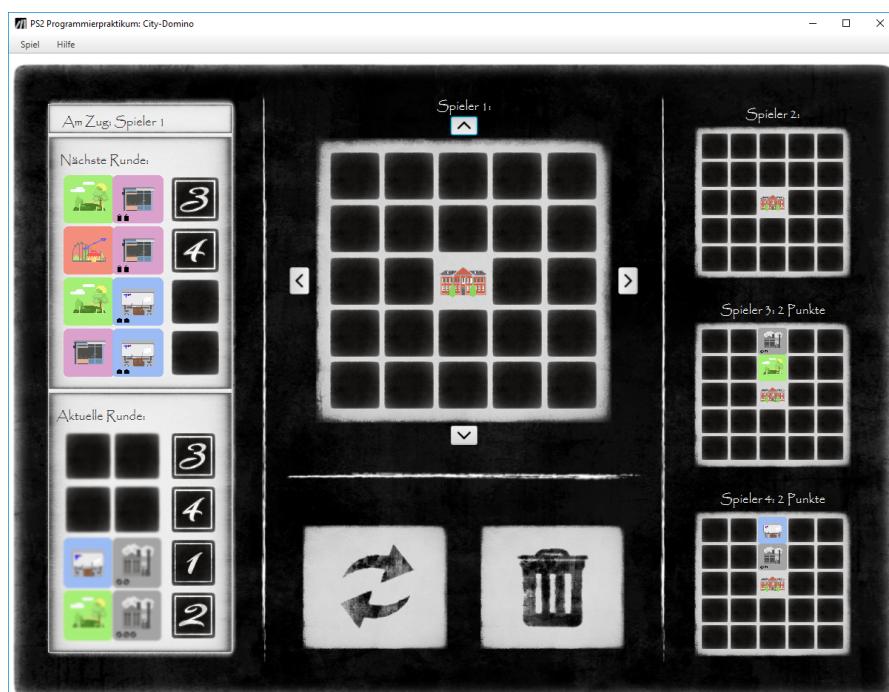


Abbildung 9: Initiales Selektieren (oben) nach Programmstart

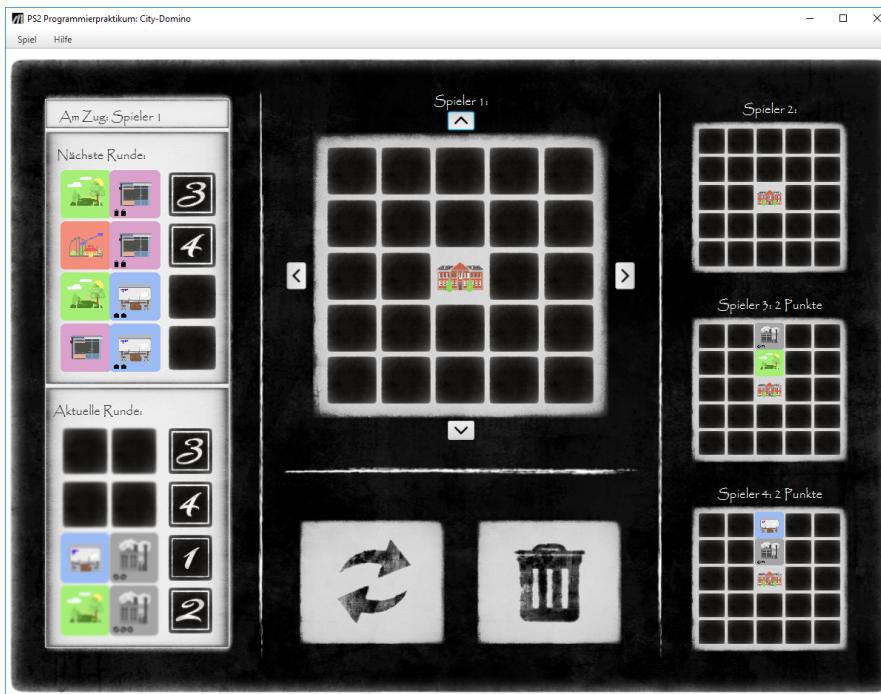


Abbildung 10: Initiales Selektieren (mittig) nach Programmstart

noch gar nicht an der Reihe war. Nun kann der Benutzer einen Domino auf dem nächsten Auswahlstapel beziehungsweise der nächsten Bank einen der übrig gebliebenen Dominoe auswählen. Nun wird der Domino, welcher als erstes selektiert wurde, in den Kasten zum Rotieren geladen (siehe Abbildung 11). Der Benutzer kann nun den Stein auf sein Brett legen (Abbildung 12). Nach dieser Aktion selektiert der Benutzer einen Domino auf der Bank für die nächste Runde. Danach muss er wieder "warten" bis er an der Reihe ist um einen ausgewählten Domino auf seinem Brett zu platzieren. Alternativ kann er seinen Domino aber auch verwerfen.

**Einlesen einer Datei** Nachdem der Benutzer eine Datei eingelesen hat, ist dieser auch gleichzeitig am Zug. Je nachdem, ob der Spielstand während des initialen Selektierens oder während einer standardmäßigen Runde abgespeichert wurde, muss der Benutzer entweder von der Bank der aktuellen Runde oder von der Bank der nächsten Runde selektieren. Generell gelten hier die gleichen Regeln zum Spielablauf wie beim Spielen ohne gespeicherten Spielstand, allerdings kann hierbei der Schritt des initialen Selektierens übersprungen werden.

**Anlegeregeln** Die erste Karte muss an das Stadtzentrum angrenzen. An das Stadtzentrum darf jeder Stadtteil angrenzen. Legt man eine Karte an eine andere Karte an, so muss mindestens eine Hälfte mit einer Seite an einen identischen Stadtteiltyp einer liegenden Karte angrenzen. Passt die abzulegende Karte weder an das Stadtzentrum noch an eine bereits ausliegende Karte, so wird sie verworfen. Alle Spielkarten müssen in das

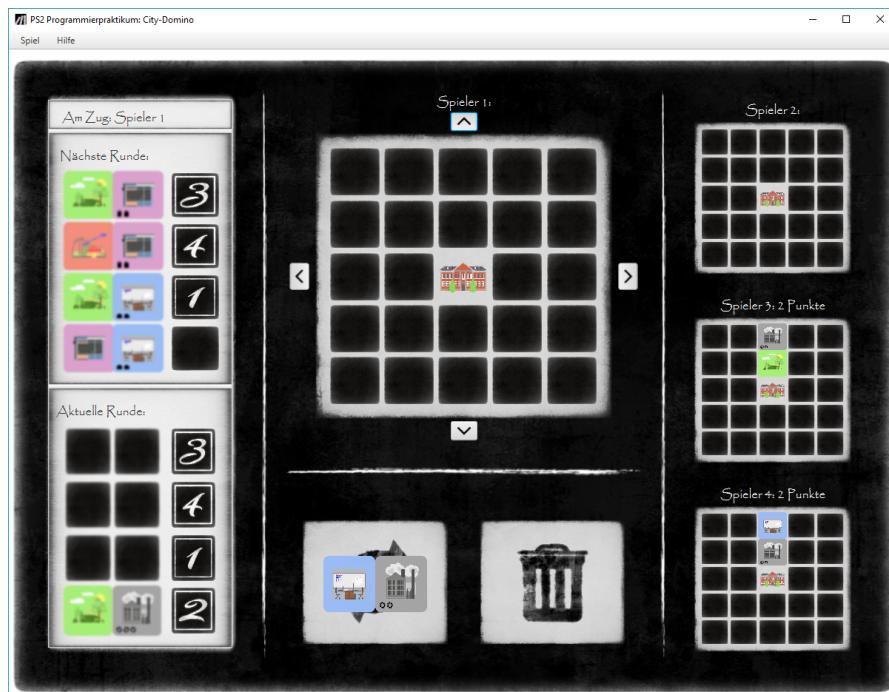


Abbildung 11: Erstes Rotieren

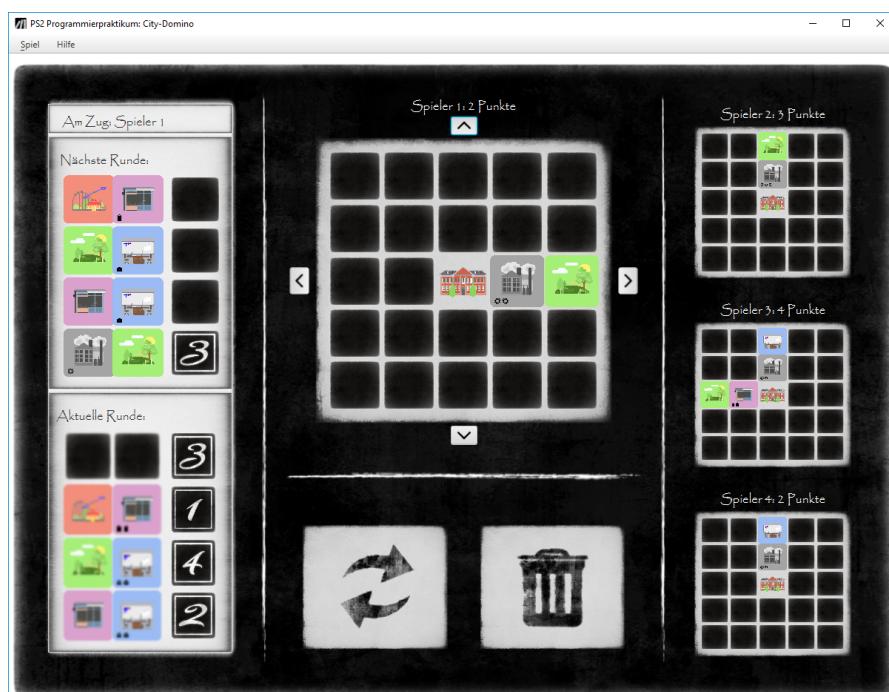


Abbildung 12: Erste Ablage auf dem Spielfeld

*5\*5-Feld passen, keine Hälfte darf hinausragen. Das Stadtzentrum muss aber nicht in der Mitte liegen, sondern kann im Spielverlauf verschoben werden, wodurch sich alle bereits gelegten Karten mit verschieben. Eine abgelegte Karte kann nicht verschoben werden. [1]*

**Spielende** *Wurden alle Spielkarten aus dem Beutel gezogen und von den Auswahlbereichen auf die Spielfelder platziert bzw. verworfen, werden die Punkte ermittelt.*

- *Jede Stadt besteht aus mehreren Stadtteilen. Ein Stadtteil setzt sich aus waagerecht und/oder senkrecht verbundenen Zellen desselben Stadtteiltyps zusammen. Das Stadtzentrum zählt zu keinem Stadtteil dazu.*
- *Die Punkte eines Stadtteils ergeben sich aus der Anzahl seiner Zellen multipliziert mit der Anzahl darin enthaltener Prestigesymbole.*
- *Innerhalb einer Stadt kann es mehrere voneinander getrennte Stadtteile desselben Typs geben. Jeder Stadtteil ist einzeln auszuwerten. Stadtteile ohne Prestigesymbole bringen keine Punkte.*

*Für die Auswertung wird für jeden Spieler die Summe der Punkte seiner Gebiete ermittelt. Gewonnen hat der Spieler mit den meisten Punkten. Bei einem Gleichstand gewinnt der Spieler mit dem größten einzelnen Gebiet. Besteht auch hier Gleichstand, so siegen beide Spieler gleichermaßen. [1]*

# Teil III.

# Programmierhandbuch

## 6. Entwicklungskonfiguration

| Softwarekomponenten  |                      |                             |
|----------------------|----------------------|-----------------------------|
| Art                  | Name                 | Version                     |
| Betriebssystem       | Windows              | 10 Professional             |
| Compiler             | Java development kit | 1.8.0_131                   |
| Entwicklungsumgebung | IntelliJ IDEA        | 2018.2.3 (Ultimate Edition) |
| Textbearbeitung      | Texmaker             | 5.0.3                       |
| Bildbearbeitung      | GIMP                 | 2.8                         |
| 3d Modellierung      | Blender              | 2.79b                       |
| UML-Editor           | Umletino             | 14.3.0                      |

Der angegebene UML-Editor wurde direkt im Browser verwendet <sup>1</sup>. Außerdem wurde ein Tool namens Checkstyle verwendet. Checkstyle gibt dem Programmierer einige Richtlinien wie der Code am Ende auszusehen hat. Es wird zum Beispiel eine vollständige Javadoc-Dokumentation, oder Konstanten statt Zahlen ohne Kontext, verlangt. Hierzu wurden die Vorgaben der Übung *Algorithmen und Datenstrukturen* verwendet. Allerdings wurde die maximale Zeichenanzahl von 100 auf 120 erhöht um den Code lesbare zu gestalten. Die entsprechende XML-Datei findet sich hier im Anhangs- bzw. Lesezeichenverzeichnis dieses Kapitels. Ansonsten ist es auch im Anhangsverzeichnis unter dem Ordner „Entwicklungskonfiguration“ zu finden <sup>2</sup>.

<sup>1</sup><http://www.umlet.com/umletino/umletino.html>

<sup>2</sup><https://www.acrobat-tutorials.de/2013/03/26/dateianlagen-und-seitenanlagen-in-pdf-dokumenten/>

## 7. Problemanalyse und Realisation

### 7.1. Problemanalyse

Hierbei habe ich mich einer Technik bedient in der man versucht sich in eine jeweilige Teilkomponente des Problems hineinzuversetzen und anzugeben für welchen Teilbereich diese Komponente verantwortlich ist. Anschließend ist es etwas einfacher sowie übersichtlicher sich auf die Struktur festzulegen, da man sämtliche Nomen als Klassen ansehen kann (hier einmal orange dargestellt). Verben spiegeln die benötigten Methoden wieder (hier grün dargestellt).

#### 1. Benutzer

- a) Als Benutzer möchte ich den aktuellen Spielstand in eine Datei mit Auswahl des Dateinamens speichern . Das Logging wird mit gespeichert .
- b) Als Benutzer möchte ich eine Datei mit einem Spielstand auswählen und öffnen können.
- c) Als Benutzer möchte ich ein Spiel initialisieren und neu starten.
- d) Als Benutzer möchte ich ein Spiel beenden (mit/ohne zu speichern ).
- e) Als Benutzer möchte ich das Laden oder Speichern loggen können.

#### 2. Spieler

- a) Als Spieler möchte ich einen Domino auswählen .
- b) Als Spieler möchte ich einen Domino drehen .
- c) Als Spieler möchte ich einen Domino setzen .
- d) Als Spieler möchte ich das Stadtzentrum bewegen .
- e) Als Spieler möchte ich meine Aktionen loggen .

#### 3. Spiel

- a) Als Spiel möchte ich die Spielfelder der Spieler visualisieren .
- b) Als Spiel möchte ich die Auswahlfelder visualisieren .
- c) Als Spiel möchte ich den aktuellen Spielstand der Spieler anzeigen .
- d) Als Spiel möchte ich den Gewinner anzeigen .
- e) Als Spiel möchte ich den Gewinner loggen .

## 7.2. Realisationsanalyse

**Grundsätzlich benötigte Datenstrukturen** Um eine Partie spielen zu können werden erst einmal Spielsteine benötigt. Hierbei wurden diverse Klassen eingeführt, die im Kapitel Domino 8.2 auf Seite 32 genauer beschrieben werden. Letztendlich bieten diese allerdings sämtliche benötigte Schnittstellen um einen Domino mit einer bestimmten Aufschrift sowie Position zu versehen.

Diese Dominos können auf Spielbrettern positioniert werden. Jeder Spieler besitzt hierbei eins, und die Gui ist in der Lage diese ordnungsgemäß darzustellen.

Um einen Domino wählen zu können ist es essentiell einen Konstrukt für eine Bank zu implementieren. Ich habe dabei eine Struktur gewählt, in der sämtliche Spieler nicht anhand irgendeines Indices, sondern anhand ihrer Referenz gespeichert werden. Dies ermöglicht es mit dem Rückgabewert einer entsprechenden Getter-Methode direkt arbeiten zu können, ohne Umwege beim Konvertieren nutzen zu müssen.

**Benutzerschnittstelle** Der Punkt Benutzer entspricht in diesem Projekt sämtlichen Anfragen, die ein Benutzer dem Programm stellen kann. Es bietet sich an eine Struktur zu wählen in der eine Klasse oder Schnittstelle als Anlaufstelle dient, über die sämtliche Anfragen bearbeitet werden können. Hierbei ist nur abzuwählen ob man eventuell die „Antwort“ des Programms gleich in diese Schnittstelle mit aufnimmt. Dies führt allerdings zu unübersichtlichem Code.

**Spielerverhalten** Der Punkt Spieler beschreibt die wirklichen Spielteilnehmer. Er muss in der Lage sein selbstständig oder durch die Interaktion mit der Gui einen Zug vollziehen zu können. Hierzu gehört das Auswählen, Drehen und Setzen eines Dominos. Es gilt abzuwählen ob dies durch eine gemeinsame Klasse geschehen soll. Ich habe mich allerdings für eine Unterteilung entschieden, da der menschliche Spieler auf die Eingabe des Benutzers reagiert, während die Bots diese mehr oder weniger ignorieren und isoliert ihre eigenen Züge vollziehen.

Des Weiteren benötigt der Spieler ein Spielbrett, auf dem er Dominos setzen kann. Man könnte das Spielbrett auch der Verwaltung (also der Spiel-Klasse) übergeben, dann wäre ein künstlicher Spieler aber nicht mehr derartig isoliert, wie es in diesem Entwurf der Fall ist. Indem die künstlichen Spieler selbst alle wichtigen Schritte ausführen, ist so möglich eine minimale Schnittstelle dem Spiel (der Verwaltung) gegenüber bereitzustellen.

Die Kapselung des Spielerverhaltens ermöglicht es außerdem wartbaren Code zu erzeugen. Es ist einfacher Fehler zu beheben oder bestimmte Prozesse auszutauschen ohne das komplette Programm umstrukturieren zu müssen. Am wichtigsten hierbei ist die Möglichkeit der Erweiterbarkeit. Durch die Kapselung ist es möglich, sämtliche Schritte eines Spielers polymorph zu gestalten. Jede Spielerart reagiert also anders auf eine Anfrage und es ist nicht nötig den Aufruf hierfür zu verändern. All diese Möglichkeiten würden entfallen, wenn man das Spielverhalten der künstlichen Spieler in der Spiel-Klasse aufrufen würde.

**Spielerinstantiierung** Um die Bindung zum Spiel so gering wie möglich zu halten (wie im letzten Absatz erklärt), habe ich versucht auch bei der Instantiierung möglichst nachhaltigen Code zu schreiben indem ich eine statische Fabrikmethode verwende um die einzelnen Spieler zu instantiieren. Letztendlich verschiebt man mit solch einer Fabrik lediglich die Instantiierung in eine gemeinsame Methode, allerdings muss man im späteren Verlauf der Entwicklung auch nur hier Dinge ändern. Dies ist besonders hilfreich beim Hinzufügen weiterer Spielertypen. Die Fabrik-Methode befindet sich in einem Enum namens *Playertypes*. Diese Enum-Werte können in gewisser Weise aber auch als Platzhalter für die „richtigen“ Spieler herhalten. Wie das funktioniert, wird im folgendem Abschnitt beschrieben.

**Alternative Spielerinitialisierung** Der Vorgang des Instantiierens in der Fabrik-Methode ist relativ aufwendig. Man hätte im Prinzip auch komplett auf die Fabrik verzichten können, auf lange Sicht ist es aber effektiver sie hier einzuführen da man sonst nur sehr umständlich ein Auswahlfeld für mögliche Spieler starten kann. Hierzu folgendes Szenario: Es gibt N Spielertypen, unter denen der Benutzer seine Gegner wählen kann. Dafür gibt es ein eigenes Auswahlfenster, welches vor dem eigentlichen Spiel auftauchen soll (siehe Abbildung 13, S. 25). Der Controller dieses Fensters übernimmt somit die Aufgabe der Main-Methode das Spiel zu starten. Diese tut genau dasselbe wie die ursprüngliche Main-Methode, indem sie das FXML-Dokument aufruft. Der Controller wird dabei automatisch gestartet und die Initialize-Methode der Controller-Klasse des GameFXML-Dokuments durchlaufen. Hierbei habe ich bei meinen Überlegungen keinen Weg gefunden Argumente der Initialize-Methode zu übergeben, da diese ja das Interface Initializable implementiert und somit die Signatur nicht verändert werden kann. Daher gibt es eine eine Methode, welche losgelöst vom Konstruktor das Spiel mit übergebenen Spielertypen starten kann. Dieser Ansatz funktioniert soweit und ist in der Main-Klasse als auskommentierter Block mit dem Schlüsselwort „Alternative“ gekennzeichnet. Falls dieser Block einkommentiert und der Rest der Main-Methode auskommentiert wird, startet ein Auswahlfenster mit einer Möglichkeit der Spielerselektierung. Dies ist allerdings noch nicht ausgereift da es lediglich zur Darstellung des Problems dienen soll, sodass bei invalider Selektierung eine Exception geworfen wird.

Es gibt zwar bereits einen weiteren Spielertypen, dieser ist allerdings lediglich durch den Umstand entstanden, dass ich die Aufgabenstellung zuerst falsch verstanden habe. In der Aufgabenstellung steht, dass der künstliche Spieler bei Punktegleichheit mehrerer potentieller Positionen für einen Domino diejenige wählen soll, welche die geringste Anzahl an Einzelzellen produziert. Hierbei habe ich allerdings fälschlicher Weise angenommen, dass damit einzelne Distrikte gemeint sind. In der Version des Bots, welche sich im endgültigen Spiel wiederfindet, wurde dies aber behoben und es wird auf die Einzelzellen außerhalb der Distrikte geachtet.

**Spiel** Dieser Begriff beschreibt koordinierte Abarbeitung der Spielerzüge. Es wird der Benutzeroberfläche mitgeteilt, was angezeigt werden soll. Außerdem werden sämtliche Stapel von Dominos bereitgestellt die für ein vollwertiges Spiel genutzt werden sollen (Beutel, Bänke). Alle benötigten Dateioperationen werden hier eingeleitet, aber nicht

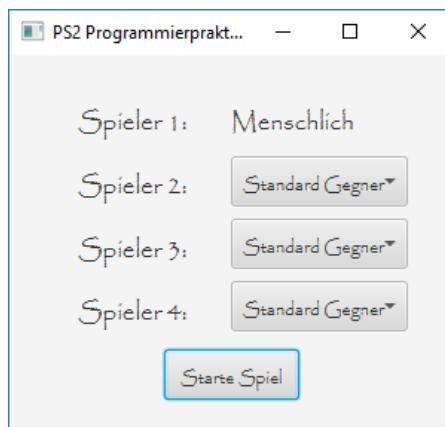


Abbildung 13: Auswahlfenster

direkt in dieser Klasse bearbeitet. Durch das ganze Exceptionhandling wird es ziemlich unübersichtlich die gesamte Funktionalität in das Spiel zu verlagern, da dieses in erster Linie für die übergeordnete Organisation des ganzen Programms gedacht ist.

**Log** Da man vermehrt, und vor allem an vielen verschiedenen Stellen im Code, eine neue Zeile in die Logdatei schreiben, beziehungsweise auf der Konsole ausgeben möchte, bietet sich für die Implementierung des Loggers das *Singleton-Muster* an. Dieses Muster verwaltet eine einzige globale Instanz auf die immer wieder zugegriffen wird. Das Muster eignet sich besonders gut für das Loggen von Daten, da man alles in dieselbe Datei schreiben möchte und diese nicht jedesmal neu *suchen / anlegen* muss. Im Logger kann man zum Beispiel einfach ein entsprechendes Feld verwalten.

**Speichern und Laden** Auch beim Speichern und Laden wird in diesem Entwurf ein *Singleton-Muster* verwendet. Da man beim Speichern jeweils den Dateipfad, nach erstmaliger Eingabe, nicht erneut eingeben möchte, wenn dies nicht unbedingt erforderlich ist (siehe Abschnitt *Speichern als* gegenüber *Speichern*, 5.1 auf Seite 13). Und auch das Speichern und Laden erstreckt sich vermehrt über das gesamte Projekt. Alternativ könnte man auch eine klassische Klasse verwenden, wegen den genannten Punkten empfiehlt sich aber gerade für diese beiden Anwendungsfälle das Muster. Für eine genauere Beschreibung des Musters siehe 8.3 Singleton-Muster auf Seite 34.

## 8. Beschreibung grundlegender Klassen

Alle Klassen wurden in folgende Packages unterteilt. Im folgenden werden diese vom Gesamtprogramm isoliert betrachtet. Eine globale Übersicht über sämtliche Zusammenhänge findet sich erst im Kapitel 9 Programmorganisationsplan, auf Seite 65.

### 8.1 gui-Package

- Main
- Controller
- JavaFXGUI

### Logic-Package

- 8.2 token
  - Pos
  - DistrictType
  - Tiles
  - SingleTile
  - Domino
- 8.3 dataPreservation
  - Loader
  - Logger
- 8.4 bankSelection
  - Bank
  - Choose
  - Entry
- 8.5 playerState
  - District
  - Board
  - Botbehavior
  - Result
  - ResultRanking
- 8.6 differentPlayerTypes
  - DefaultAIPlayer
  - HumanPlayer
  - playerType
- 8.7 logicTransfer
  - Exceptions
  - Converter
  - GUI2Game
  - GUIConnector
  - PossibleField
  - Game

## 8.1. gui

**Main** Diese Klasse dient als Startpunkt des gesamten Projekts. In der Hauptmethode wird das FXML-Dokument mit seinem Controller geladen. Die einkommentierte *start*-Methode startet zusätzlich noch ein neues Spiel, indem sie der *startGame*-Methode die nötigen Parameter übergibt.

**Controller** Hiermit sind die beiden Klassen namens *FXMLDocumentController* sowie *FXMLIntroController* gemeint. Beide Klassen bilden die direkte Schnittstelle mit ihrem jeweiligen FXML-Dokument und reichen im Kern alle Befehle an die betreffende Klasse weiter. Im *FXMLDokumentController* findet sich allerdings außerdem die Modifikation der Gui. Um den Hintergrund mit einem eigenen Bild zu versehen, wurde eine Methode namens *setPnWithImage* implementiert (siehe Listing 1, S. 27, *FXMLDocumentController* - *setPnWithImage*). Um diese Methode effektiv nutzen zu können wurde im FXML-Dokument selbst für jedes Bild eine leere Pane eingefügt. Diese dient der Methode als Anker. Das Flag *addAsForeground* gibt an, ob das Bild im Vorder- oder Hintergrund eingefügt werden soll.

Die Klasse *FXMLIntroController* dient als Schnittstelle für das Intro-Fenster, falls mehr Spielertypen implementiert werden sollen und es eine Möglichkeit geben soll. Dem Benutzer wird so eine Auswahlmöglichkeit seiner Gegnertypen bereitgestellt. Wie diese aussehen könnte wurde bereits in Abbildung 13 auf Seite 25 gezeigt.

## JavaFXGUI

**Einleitung** Diese Klasse implementiert das Interface *GUIConector* und somit die Logik, welche für die Darstellung einer Partie benötigt wird. Sie hält (ähnlich wie die Game-Klasse selbst) sämtliche Teilkomponenten in einer Datenstruktur. Die hierbei verwendeten Datenstrukturen beziehen sich allerdings direkt auf die Darstellungsebene. Bänke werden hier beispielsweise nicht vom Datentyp Bank beschrieben, sondern besitzen den Typ *ImageView*]. Zu großen Teilen war auch diese Klasse bereits aus der

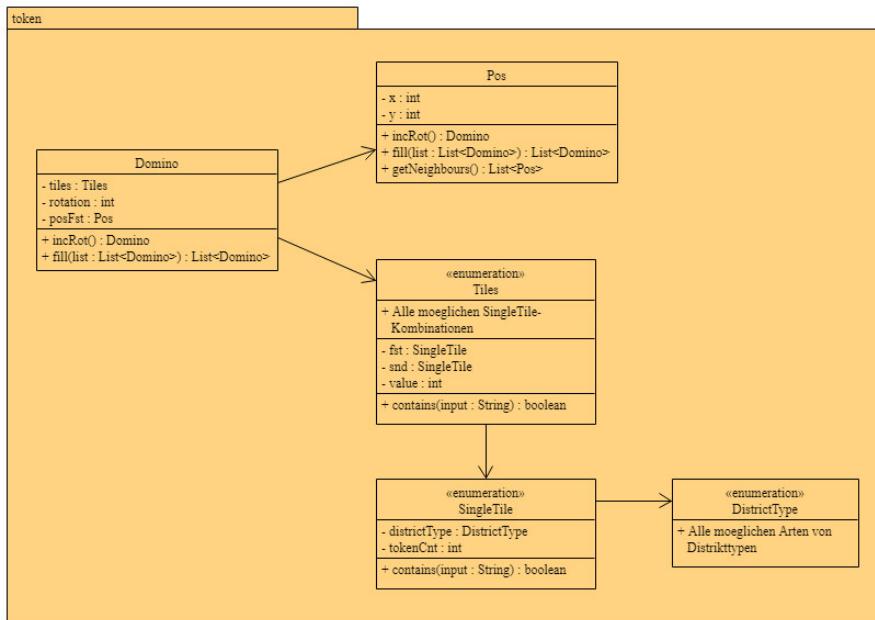
Listing 1: *FXMLDocumentController* - *setPnWithImage*

```

1 private void setPnWithImage(Pane pane, Image image, boolean addAsForeground) {
2     ImageView imgVW = new ImageView(image);
3     imgVW.setPreserveRatio(false);
4     imgVW.fitHeightProperty().bind(pane.heightProperty());
5     imgVW.fitWidthProperty().bind(pane.widthProperty());
6     if (addAsForeground) {
7         pane.getChildren().add(imgVW);
8     } else {
9         pane.getChildren().add(0, imgVW);
10    }
11 }
```

Bonusaufgabe vorgegeben und wurde entsprechend übernommen.

**Methoden** Sämtliche Daten werden hier in Form von Bildern gespeichert. Ein Aufruf von zum Beispiel *setToBank* tut hierbei nichts anderes, als die Bilder der einzelnen *Tiles* des gegebenen Dominos auf der gewünschten Bank zu setzen. Ähnlich arbeitet die Methode *showCellOnGrid*. Um derartige Methoden nutzen zu können muss die Klasse in der Lage sein, diese zu „übersetzen“. Hierzu wird im Konstruktor ein entsprechendes Array an Bildern initialisiert, um nun das gewünschte Bild zu erhalten reicht es den Ordinal-Wert des jeweiligen *Tiles* zu ermitteln und als Index in das Array zu reichen.

Abbildung 14: UML-Darstellung des `token` packages

## 8.2. token

Das package `token` (siehe Abbildung 14) vereint sämtliche Klassen, die benötigt werden um einen Domino erstellen zu können.

**Pos** Diese Klasse wurde von der Bonusaufgabe der PS2-Übung übernommen. Es wurde lediglich die `toString()`-Methode sowie `getNeighbours()` abgeändert und einige Konstanten hinzugefügt um den Checkstyle-Richtlinien (siehe Abschnitt 6 - Entwicklungskonfiguration) folge zu leisten. Dennoch folgt hier ein kurzer Überblick über diese Klasse:

Eine Position setzt sich aus einer X- und einer Y-Komponente zusammen. Diese sind als *final* deklariert und können somit nicht verändert werden. Neben dem Konstruktor und diversen Gettern gibt es eine Methode um festzustellen ob sich eine gegebene Position neben der Position des Objekts befindet. Dazu wird die Differenz der beiden jeweils gleichnamigen Komponenten gebildet und am Ende verglichen, ob nur jeweils eine der beiden Differenzen gleich Null ist (siehe Listing 2).

Die Methode `getNeighbours()` liefert eine ArrayList der vier Nachbarn. Hierzu wird eine Liste initialisiert und mit neuen Positionen gefüllt, deren X- und Y-Komponenten entsprechend modifiziert werden (siehe Listing 3).

Eine `equals`-Methode wurde ebenfalls implementiert, diese vergleicht allerdings nur die jeweiligen x- und y-Komponenten der beiden Positionen. Bei der `toString`-Methode werden die beiden Komponenten, zwischen einem Klammernpaar und mit Komma getrennt,

Listing 2: Pos - isNextTo()

```

1 public boolean isNextTo(Pos p) {
2     int xDiff = Math.abs(x - p.x());
3     int yDiff = Math.abs(y - p.y());
4     return (xDiff == 1 && yDiff == 0
5            || xDiff == 0 && yDiff == 1);
6 }
```

Listing 3: Pos - getNeighbours()

```

1 public List<Pos> getNeighbours() {
2     List<Pos> neighbours = new ArrayList<>();
3     neighbours.add(LEFT_ROT, new Pos(this.x - 1, this.y));
4     neighbours.add(DOWN_ROT, new Pos(this.x, this.y - 1));
5     neighbours.add(RIGHT_ROT, new Pos(this.x + 1, this.y));
6     neighbours.add(UP_ROT, new Pos(this.x, this.y + 1));
7     return neighbours;
8 }
```

ausgegeben.

**DistrictType** In diesem Aufzählungstyp werden die möglichen Kategorien der Distrikte aufgeführt. Wichtig hierbei, auch ein leeres Feld sowie das Stadtzentrum besitzen einen Typ (siehe listing 4). Dieser Aufzählungstyp spielt vor allem beim Auszählen der Punkte bzw. dem Verwalten der verschiedenen Distrikte eine wichtige Rolle.

**SingleTile** Dieser Aufzählungstyp repräsentiert einen Domino Aufdruck.

Ein Konstruktor verbindet die Enum-Darstellung mit einem Distrikttypen und einer Anzahl an Punkten, welche auf dem betrachteten Tile verfügbar sind (siehe Listing 5).

Außerdem verfügt der Aufzählungstyp über Getter für beide Felder und eine Methode, die überprüft, ob eine gegebene String-Repräsentation dem Wert eines der Enumobjekte entspricht. Hierzu wird eine Schleife durchlaufen, die beim Fund mit *true* und ansonsten mit *false* abbricht.

Listing 4: DistrictType

```

1 public enum DistrictType {
2     EMPTY_CELL, CENTER, AMUSEMENT, INDUSTRY, OFFICE, PARK, SHOPPING, HOME
3 }
```

Listing 5: singleTile

```

1 CC(CENTER, 0), EC(EMPTY_CELL, 0),
2 AO(AMUSEMENT, 0), A1(AMUSEMENT, 1), A2(AMUSEMENT, 2), A3(AMUSEMENT, 3),
3 IO(INDUSTRY, 0), I1(INDUSTRY, 1), I2(INDUSTRY, 2), I3(INDUSTRY, 3),
4 OO(OFFICE, 0), O1(OFFICE, 1), O2(OFFICE, 2), O3(OFFICE, 3),
5 PO(PARK, 0), P1(PARK, 1), P2(PARK, 2), P3(PARK, 3),
6 SO(SHOPPING, 0), S1(SHOPPING, 1), S2(SHOPPING, 2), S3(SHOPPING, 3),
7 HO(HOME, 0), H1(HOME, 1), H2(HOME, 2), H3(HOME, 3);
8
9 private DistrictType districtType;
10
11 private int tokenCnt;
12
13 SingleTile(DistrictType disctrictType, int tokenCnt) {
14     this.districtType = disctrictType;
15     this.tokenCnt = tokenCnt;
16 }
```

Listing 6: Tiles

```

1 POPO_Val1(P0, P0, 1),
2 POPO_Val2(P0, P0, 2),
3 ...
4 POI2_Val47(O0, I2, 47),
5 POI3_Val48(P0, I3, 48);
6
7 public static final int TILES_CNT = Tiles.values().length;
8
9 private final SingleTile fst;
10 private final SingleTile snd;
11
12 private final int value;
13
14 Tiles(SingleTile fst, SingleTile snd, int value) {
15     this.fst = fst;
16     this.snd = snd;
17     this.value = value;
18 }
```

Listing 7: Domino - fill

```

1 public static List<Domino> fill(List<Domino> list) {
2     if (null == list) {
3         list = new LinkedList<>();
4     } else {
5         list.clear();
6     }
7     for (Tiles tile : Tiles.values()) {
8         list.add(new Domino(tile, DEFAULT_POS));
9     }
10    return list;
11 }
```

**Tiles** Dieser Aufzählungstyp beschreibt alle möglichen *SingleTile*-Kombinationen, die im Stapel einer normalen Partie des Spiels möglich sind (siehe listing 6).

Auch hier gibt es wieder einen Konstruktor, der diverse Werte an die jeweiligen Enum-Werte bindet. Es werden neben den beiden SingleTile-Kombinationen auch der Wert dieser spezifischen Kombination gespeichert. Der Wert dient dem Sortieren der Bänke und wurde aus der Aufgabenstellung entnommen (wurde also nicht willkürlich festgelegt).

Um von außerhalb einfacher bestimmte Tile-Kombinationen zu erstellen und das Testen einfacher gestalten zu können, wurden mehrere Methoden eingeführt die dies erleichtern sollen. Diese sind allerdings im Hauptprogramm nicht von Bedeutung. Neben sonstigen Gettern besitzt diese Klasse lediglich eine *toString()*-Methode, wo nur die ersten 4 Buchstaben der Enum-Werte zurückgegeben werden, da diese die Tile-Kombination angeben. Der Wert der Kombination entfällt hierbei (ist aber in dieser Methode auch unbedeutend).

**Domino** Diese Klasse vereint sämtliche zuvor genannten Datenstrukturen.

Ein Domino besitzt eine bestimmte Kombination aus SingleTiles und einer Rotation sowie Position. Letztere beiden beziehen sich auf ein Board, nicht auf eine Bank oder dergleichen. Um eine gegebene Liste zu füllen wird die Methode *fill* bereitgestellt. Hierbei wird eine gegebene Liste geleert und mit allen möglichen unterschiedlichen Dominos gefüllt (siehe listing 7, Domino - fill). Um einen Domino auf einem Spielfeld zu platzieren muss es möglich sein seine Position sowie Rotation zu verändern, dies geschieht über die Methoden *setPosition()* und *incRot() / setRotation()*. *incRot()* inkrementiert hierbei die Rotation um neunzig Grad. Außerdem ist diese Klasse in der Lage mittels der *toFile()*-Methode einen Zeichenfolge zu generieren um die Tile-Kombination des Steins später abspeichern zu können. Hierbei wird allerdings die *toString()*-Methode der Tile-Kombination aufgerufen. Als Letztes implementiert die Domino Klasse noch das Interface *Comparable* um später Dominos per *Collections.sort(...)* einfacher Sortieren zu können. Hierbei wird jeweils auf den Wert der Tile-Kombination geschaut (siehe 8, Domino - *compareTo*).

Listing 8: Domino - compareTo

```
1 @Override
2 public int compareTo(Object o) {
3     assert null != o && (o instanceof Domino);
4     Domino other = (Domino) o;
5     return this.tiles.getValue() - other.tiles.getValue();
6 }
```

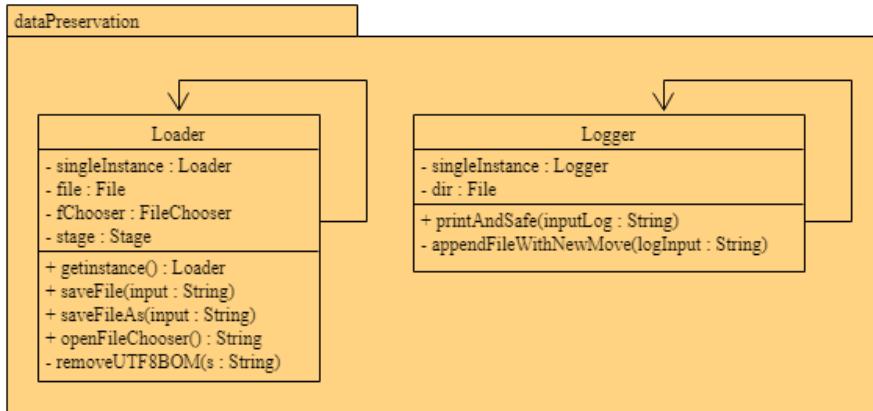


Abbildung 15: UML-Darstellung des dataPreservation packages

### 8.3. dataPreservation

Dieses Package beinhaltet die beiden Klassen dieses Projekts, welche sich mit dem Schreiben und Lesen von Dateien auseinandersetzen. Die Loader Klasse ist hierbei zur Speicherung / zum Laden des Spiels gedacht, während sich die Logger Klasse mit dem Schreiben in die Log-Datei beschäftigt.

**Singleton-Muster** *Das Singleton-Muster sichert, dass es nur eine Instanz einer Klasse gibt, und bietet einen globalen Zugriffspunkt für diese Instanz.* [3] Der Konstruktor einer Singleton-Klasse ist als private deklariert und kann somit nur innerhalb der Klasse selbst verwendet werden. Es gibt allerdings eine statische Getter-Methode um auf das Objekt zugreifen zu können. Hierbei ist zu beachten, dass ein neues Objekt nur erstellt wird, wenn vorher *kein* Objekt der Klasse existiert (siehe 9, Singleton-Muster). Dieses Muster eignet sich hervorragend für Objekte, bei denen es von Nachteil ist, wenn es mehrere Instanzen gibt.

Die Logger-Klasse benötigt nur eine Instanz, da eine gegebene Log-Datei stets erweitert werden soll. Da ständig von unterschiedlichen Ebenen des Projekts auf diese Instanz zugegriffen wird, ist es sehr praktisch eine statische (globale) Instanz zu halten, welche von allen Klassen problemlos erreicht werden kann.

Beim Loader gestaltet sich dies ähnlich. Zwar gibt es nicht allzu viele Zugriffspunkte, allerdings gibt es nur eine globale Instanz der Datei in die gespeichert werden soll. Insbesondere der Fall des Speicherns eines bereits vorher im Spielverlauf gespeicherten Spiels gestaltet sich einfacher, da man bereits die zu überschreibende Datei in der statischen Instanz mit verwaltet.

**Logger** Die Logger-Klasse hält, wie im Paragraph Singleton-Muster bereits beschrieben, ein Feld namens *singleInstance* vom Typ der Klasse *Logger* um die einzige globale Instanz zu verwalten. Diese wird im Konstruktor entsprechend gesetzt (genau wie in Listing 9, Singleton-Muster).

Listing 9: Singleton-Muster

```

1 public class Singleton {
2     private static Singleton singleInstance;
3
4     \\ further instance variables
5
6     private Singleton() {}
7
8     public static Singleton getInstance() {
9         if (singleInstance == null) {
10             singleInstance = new SingleInstance();
11         }
12     }
13     return singleInstance;
14
15     \\ further methods
16
17 }
```

Außerdem besitzt die Klasse ein Feld mit dem Datentyp *File* um eine Datei zu speichern beziehungsweise zu erweitern. Im Konstruktor wird diese mit einem Konstanten beschrieben, welche eine Datei im Ordner *./dataOutput/logFile.txt* referenziert. Die Datei ist allerdings nicht *final* und kann über diverse Setter entsprechend verändert werden.

Um eine Nachricht auf der Konsole auszugeben, sowie in eine Datei zu schreiben, wird die Methode *printAndSafe(...)* verwendet (siehe 11, Logger - printAndSafe). Die Methode verwendet neben dem standard Ausgabestrom eine weitere Hifsmethode um die Nachricht an die Log-Datei anzuhängen (siehe Listing 12, Logger - appendFileWithNewMove). Über ein Flag wird erst einmal überprüft, ob in einem vorherigen Schreibvorgang der Log-Datei ein Fehler festgestellt worden ist (Zeile 2). Anschließend wird eine *try-with-resources*-Anweisung beschrieben. Diese besitzt den Vorteil, dass der Ausgabestrom im Fehlerfall noch geschlossen werden kann. Andernfalls müsste man dieses Exception-handling über einen finally-Block abfangen [2]. Der Ausgabestrom wird also sicher geöffnet und der gegebene Text an diesen per *write*-Operation angehängt. Im Fehlerfall wird der ein kurzer Hinweis, dass es nicht möglich sei eine Logdatei für das laufende Spiel anzufertigen, auf dem standard Fehlerstrom ausgegeben. Anschließend wird die Nachricht der ausgelösten Exception angezeigt. Um in Zukunft nicht erneut in diesen Catch-Block zu laufen wird das Flag *loggingPossible* auf *false* gesetzt.

**Loader** Diese Klasse ist zum Laden und Speichern von Spielständen gedacht. Im Konstruktor wird ein Filechooser-Objekt initialisiert. Dieses wird immer wieder verwendet, wenn ein Filechooser-Fenster geöffnet werden soll um ein neues File-Objekt zu erstellen. In diesem soll der Spielstand gespeichert werden.

Listing 10: Logger - printAndSafe

```
1 public void printAndSafe(String inputLog) {  
2     System.out.println(inputLog);  
3     appendFileWithNewMove(inputLog);  
4 }
```

Listing 11: Logger - printAndSafe

```
1 public void printAndSafe(String inputLog) {  
2     System.out.println(inputLog);  
3     appendFileWithNewMove(inputLog);  
4 }
```

Listing 12: Logger - appendFileWithNewMove

```
1 private void appendFileWithNewMove(String logInput) {  
2     if (this.loggingPossible) {  
3         try (Writer outputStream = new BufferedWriter(new FileWriter(this.dir, true))) {  
4             outputStream.write(logInput + "\n");  
5         } catch (IOException e) {  
6             System.err.println(ERROR_DELIMITER  
7                     + "\nThe current game will not have a logfile available:\n"  
8                     + e.getMessage() + "\n" + ERROR_DELIMITER + "\n");  
9             this.loggingPossible = false;  
10        }  
11    }  
12 }
```

Listing 13: Loader - actualSavingProcess

```

1 private static void actualSavingProcess(File output, String text) {
2     try (PrintWriter outputStream = new PrintWriter(output)){
3         outputStream.print(text);
4     } catch (FileNotFoundException e) {
5         Logger.getInstance().printAndSafe("Could not save file\n" + e.getMessage());
6     }
7 }
```

Listing 14: Loader - saveFileAs

```

1 public void saveFileAs(String input) {
2     if (null == stage) {
3         this.stage = new Stage();
4     }
5     if (null != this.file) {
6         this.fChooser.setInitialDirectory(this.file.getParentFile());
7     }
8     this.file = fChooser.showSaveDialog(stage);
9     if (null == this.file) {
10         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER
11             + "\nUser aborted the saving process\n" + Logger.ERROR_DELIMITER + "\n");
12     } else {
13         actualSavingProcess(this.file, input);
14         Logger.getInstance().printAndSafe("User saved the game as \""
15             + this.file.getName() + "\" to " + this.file.getPath() + "\n");
16     }
17 }
```

**Speichern** Um ein Spiel initial abzuspeichern, benötigt man in jedem Fall den beschriebenen Filechooser. Hierzu verwendet man die Methode *saveFileAs*, welche eine Stage initialisiert, den Chooser öffnet und eine weitere Hilfsmethode für den Schreibprozess aufruft (siehe Listing 14, S. 37, Loader - saveFileAs). Diese Methode nennt sich *actualSavingProcess* und öffnet einen Ausgabestrom auf den der gegebene Text geschrieben wird (siehe Listing 13, S. 37, Loader - actualSavingProcess). Es wird von dem einen Ressourcen-Block Gebrauch gemacht, der den Ausgabestrom selbst bei Abbruch mit einer Exception sicher schließt [2]. Neben dem Speichern mit der expliziten Verwendung des Filechooser-Objekts gibt es ebenfalls die Methode *saveFile*, welche die Datei überschreibt falls diese bereits vorhanden ist. Falls dies nicht der Fall sein sollte, wird erneut *saveFileAs* aufgerufen.

## 8.4. bankSelection

Dieses Package umfasst alle benötigten Datenstrukturen, welche bei allen Selektiervorgängen benötigt werden (siehe Listing 16, S. 39, UML-Darstellung des token packages).

**Bank** Die Bank-Klasse dient als Datenstruktur auf deren Basis die Spielteilnehmer Dominos für die aktuelle sowie die nächste Runde wählen können. Sie verwaltet hauptsächlich einen Array von Entry-Objekten.

**Konstruktoren** Die Klasse verfügt über drei verschiedene Konstruktoren (siehe Listing 15, S. 39, Bank - Konstruktor).

1. Konstruktor: Wird beim standardmäßigen Erstellen einer Bank im Spiel genutzt. Es wird lediglich die Anzahl der Spieler gegeben, sodass die Bank entsprechend viele leere Entry-Plätze generieren kann.
2. Konstruktor: Wird beim Testen ohne Dateiverarbeitung genutzt. Es wird ein bereits generierter Array aus Entry-Objekten gesetzt. Außerdem besteht die Möglichkeit ein beliebiges Random Objekt zu setzen (sehr nützlich, um vermehrt die selbe Spielsituation generieren zu können).
3. Konstruktor: Wird zum Testen mit Dateiverarbeitung genutzt. Hierbei wird ein String hereingereicht, welcher derartig verarbeitet wird, dass am Ende eine Liste an validen Entry-Daten zustande kommt.

Bei dem String-Input des Konstruktors wird davon ausgegangen, dass die Syntax stimmt. Diese wird bereits in der Converter-Klasse überprüft. Bei der Konvertierung des Strings in einen Entry-Array wird der String bei jedem Komma geteilt, sodass es anschließend möglich ist diesen von hinten nach vorne zu durchlaufen und jeden String-Wert jeweils einem Entry-Konstruktor zu übergeben. Dieser ist in der Lage, zusammen mit den teilnehmenden Spielern, ein gültiges Entry-Objekt zu generieren.

**Methoden** Neben zahlreichen Gettern und Settern ist es möglich eine Bank aufzulösen ohne eine neue Instanz anzufordern, indem man die Methode *clearAllEntries* verwendet, welche lediglich alle Entry-Referenzen auf den Null-Pointer setzt, die Bank-Referenz aber unverändert lässt.

Die wohl wichtigste Methode der Bank-Klasse nennt sich *selectEntry* und ermöglicht es einem Spieler einen gewünschten Domino auszuwählen. Hierfür wird der Entry an dem gegebenen Bank-Index mit der Spieler-Referenz belegt (siehe Listing 16, S. 40, Bank - selectEntry).

**Choose** Diese Klasse dient den Bots als Hilfsstruktur beim Analysieren der Dominos auf den Bänken und stellt nichts weiter als ein reines Datenobjekt dar. Außerdem wird gespeichert, wie viele Punkte der Domino an seiner Position einbringt (bzw. wie viele Punkte das Brett danach insgesamt wert ist). Um dem Spieler mitzuteilen wo der Domino

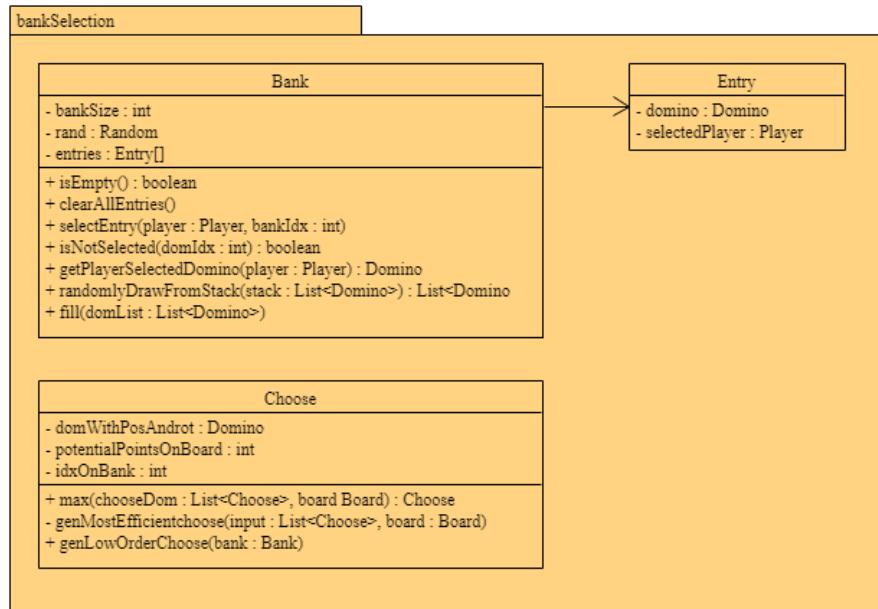


Abbildung 16: UML-Darstellung des token packages

Listing 15: Bank - Konstruktor

```

1 // standard constructor
2 public Bank(int playerCnt) {
3     this.entries = new Entry[playerCnt];
4     this.bankSize = playerCnt;
5     this.rand = new Random();
6 }
7
8 // testing constructor - no fileIO
9 public Bank(Entry[] entries, Random pseudoRandom) {
10    this.entries = entries;
11    this.rand = pseudoRandom;
12    this.bankSize = entries.length;
13 }
14
15 // testing constructor - with fileIO
16 public Bank(String preallocation, List<Player> players, Random rand) {
17     assert null != preallocation && null != players && null != rand;
18     this.bankSize = players.size();
19     this.entries = new Entry[this.bankSize];
20     if (0 < preallocation.length()) {
21         String[] singleEntries = preallocation.split(SEPARATOR_STRING_REPRESENTATION);
22         int offset = this.bankSize - singleEntries.length;
23         for (int i = singleEntries.length - 1; i >= 0; i--) {
24             this.entries[i + offset] = new Entry(singleEntries[i], players);
25         }
26     }
27 }

```

Listing 16: Bank - selectEntry

```

1 public void selectEntry(Player player, int bankIdx) {
2     assert null != player && isValidBankIdx(bankIdx) && null != this.entries
3         && null != this.entries[bankIdx] && isNotSelected(bankIdx);
4     this.entries[bankIdx].selectEntry(player);
5 }
```

wiederzufinden ist, wird ebenfalls der Index auf der Bank gespeichert. Im Konstruktor werden diese gesetzt und, da sie als final gekennzeichnet sind, anschließend auch nicht mehr verändert.

**Methoden** Neben Gettern stellt diese Klasse lediglich eine equals- und eine compareTo-Methode bereit. Durch die Implementierung des Comparable-Interfaces ist es möglich diese in der defaultAIKlasse zu verwenden um das maximale Choose-Objekt einer Liste zu finden.

**Entry** Diese Klasse dient als Datenstruktur für die einzelnen Bankelemente. Sie verhält sich ähnlich zu einer herkömmlichen Map. Eine Map bietet allerdings nicht die Möglichkeit einen Nullpointer als Schlüssel zu verwenden, dies ist bei einem Entry-Objekt jedoch essentiell, daher wurde hierfür eine eigene Klasse angelegt.

**Konstruktor** Diese Klasse besitzt zwei Felder zum Speichern eines Dominos und dem Spieler der diesen Domino verwendet. Im ersten Konstruktor werden diese ohne weitere Aktionen gesetzt. Im zweiten wird eine Zeichenfolge und die Liste der teilnehmenden Spieler hereingereicht. Es wird hierbei davon ausgegangen, dass der String bereits beim initialen Einlesen derartig überprüft wurde, dass die Syntax den Richtlinien entspricht. Nun wird der eingegebene String an der Stelle geteilt, an der sich das Leerzeichen befindet. Der vordere Teil repräsentiert den Spieler und wird daher zu einem Integer geparsed (falls dies möglich sein sollte). Anschließend wird der Domino gebildet. Neben zahlreichen Gettern und Settern besitzt die Klasse noch eine *toString*- sowie eine *copy*- und *equals*-Methode.

## 8.5. playerState

Dieses Package vereint alle verwaltungstechnischen Klassen, welche im Zuge der Spielerverwaltung benötigt werden. Die unterschiedlichen Spielertypen sind hierbei nicht beinhaltet (UML siehe Abbildung 17, S. 42).

### District

**Einleitung** Klasse, welche die Punkte eines Spielers verwaltet. Hierbei werden die zusammenhängenden Positionen sowie SingleTiles eines Distrikts in entsprechenden Listen gespeichert.

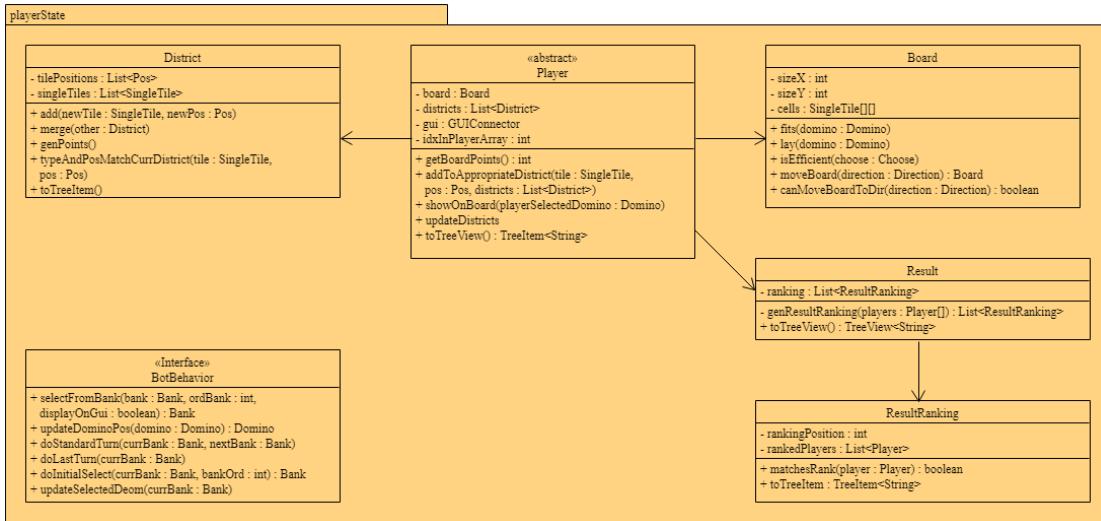
**Konstruktoren** Es gibt drei Ausprägungen des Distrikt-Konstruktors (siehe Listing 17, S. 43, District - Konstruktoren).

1. Zum Erstellen eines neuen Distrikts. Es wird eine neue Liste für die Positionen sowie die SingleTiles erzeugt und die gegebenen Daten werden mithilfe eines Aufrufs der Methode *add* (siehe Listing 18, District - add) in diese eingepflegt.
2. Zum Zusammenführen mehrerer gegebener Distrikte. Die gegebenen Distrikte werden hierbei durchlaufen und jede Teilkomponente (SingleTile / Position) wird per Aufruf der Listen-Methode *addAll* in die Listen der aufrufenden Instanz eingepflegt.
3. Zum Testen. Der Programmierer setzt hierbei lediglich die gewünschten SingleTiles sowie Positionen, die in dem Distrikt enthalten sein sollen. Wie bei allen Formen des Konstruktors dieser Klasse wird lediglich erwartet, dass die gegebenen Argumente keine Null-Pointer enthalten, ansonsten werden die Daten aber direkt (und ohne jegliche Überprüfung) übernommen.

**Methoden** Die Klasse District besitzt eine Methode namens *typeAndPosMatchCurrDistrict* welche, zum Beispiel im abstrakten Typ Player, verwendet wird um festzustellen ob ein SingleTile-Objekt an einer bestimmten Position in einen bestimmten Distrikt passt oder nicht (siehe Listing 19, S. 43, District - typeAndPosMatchCurrDistrict). Hierbei wird geschaut, ob die SingleTile-Referenz den Distrikttyp mit dem aufrufenden Distrikt teilt. Dazu wird die Methode *matchingDistrictTypes* verwendet (siehe Listing 20, S. 43, District - matchingDistrictTypes).

Um festzustellen, ob sich die gegebene Position neben dem aufrufenden Distrikt befindet, wird die Methode *elemPosIsNextToExistingElem* verwendet (siehe Listing 21, S. 44, District - elemPosIsNextToExistingElem). Es werden jeweils die Nachbarn der einzelnen Distriktelemente gebildet und nach der gegebenen Position durchsucht. Falls es eine Übereinstimmung geben sollte, wird die Schleife direkt abgebrochen.

In einigen Hilfsmethoden wird außerdem eine Kopie eines Distrikts verlangt. Hierfür gibt es die Methode *copy*, die ein Deep-Copy erstellt. Dafür werden die Daten der Listen

Abbildung 17: UML-Darstellung des `playerState` packages

einzelne in neue Ausgabelisten kopiert. Ein neuer Distrikt wird per Konstruktoraufzugriff geformt und zurückgegeben.

Um das Ergebnis darstellen zu können, wird ein `TreeItem` generiert, das die Punkte sowie den Namen des Distrikts enthält.

**Board** Diese Klasse dient als Datenobjekt für alle Spielertypen, um Dominos ablegen zu können. In erster Linie besitzt die Klasse einen zweidimensionalen `SingleTile`-Array auf dem sämtliche Dominos platziert werden können. Es bietet allerdings auch noch weitere Funktionalitäten, die es den Spielern erlauben, ihre Züge einfacher zu planen und absolvieren zu können.

**Konstruktoren** Auch bei dieser Klasse gibt es drei verschiedene Konstruktoren (siehe Listing 22, S. 45, `Board` - Konstruktor).

1. Konstruktor: Standard-Konstruktor bekommt die Dimensionen übergeben und generiert ein leeres Feld.
2. Konstruktor: Konstruktor zur Evaluation eines Strings. Der gegebene String wird an den Zeilenumbrüchen und anschließend an den Leerzeichen geteilt. Danach wird ein Feld mit der entsprechenden Größe gebildet und die Elemente werden per Schleifendurchlauf gesetzt.
3. Konstruktor: Fungiert als Copy-Konstruktor, um eine neue Referenz eines bereits existierenden Objekts zu erzeugen.

Listing 17: District - Konstruktoren

```

1 // Constructor used for standard round
2 public District(SingleTile fstDistrictMember, Pos pos) {
3     assert null != fstDistrictMember && null != pos;
4     this.tilePositions = new LinkedList<>();
5     this.singleTiles = new LinkedList<>();
6     add(fstDistrictMember, pos);
7 }
8
9 // Constructor used for merging multiple districts
10 public District(List<District> districts) {
11     assert null != districts;
12     this.singleTiles = new LinkedList<>();
13     this.tilePositions = new LinkedList<>();
14     for (District currDistrict : districts) {
15         this.singleTiles.addAll(currDistrict.singleTiles);
16         this.tilePositions.addAll(currDistrict.tilePositions);
17     }
18 }
19
20 // Constructor used for testing
21 public District(List<SingleTile> singleTiles, List<Pos> pos) {
22     assert null != singleTiles && null != pos;
23     this.singleTiles = singleTiles;
24     this.tilePositions = pos;
25 }
```

Listing 18: District - add

```

1 public void add(SingleTile newTile, Pos newPos) {
2     assert null != newTile && null != newPos && !this.tilePositions.contains(newPos);
3     this.singleTiles.add(newTile);
4     this.tilePositions.add(newPos);
5 }
```

Listing 19: District - typeAndPosMatchCurrDistrict

```

1 public boolean typeAndPosMatchCurrDistrict(SingleTile tile, Pos pos) {
2     assert null != tile && null != pos;
3     return matchingDistrictTypes(tile) && elemPosIsNextToExistingElem(pos);
4 }
```

Listing 20: District - matchingDistrictTypes

```

1 private boolean matchingDistrictTypes(SingleTile tile) {
2     return tile.getDistrictType() == this.singleTiles.get(0).getDistrictType();
3 }
```

Listing 21: District - elemPosIsNextToExistingElem

```

1 private boolean elemPosIsNextToExistingElem(Pos pos) {
2     boolean isNextToDistrictMember;
3     int tileCnt = this.tilePositions.size();
4     int i = 0;
5     do {
6         isNextToDistrictMember = this.tilePositions.get(i).getNeighbours().contains(pos);
7         i++;
8     } while (!isNextToDistrictMember && i < tileCnt);
9     return isNextToDistrictMember;
10 }
```

**Methoden** Um diverse Vermutungen über das Board anstellen zu können, bietet es eine Reihe an Methoden um spezifische Dinge abzuprüfen. Es ist zum Beispiel möglich zu prüfen ob sich eine gegebene Position auf dem Board befindet, ohne händisch die Dimensionen prüfen zu müssen. Außerdem ist es möglich zu prüfen ob ein Domino, mit einer gegebenen Position, auf das Board passt oder nicht (siehe Listing 23, S. 46, Board - fits). Da die fits Methode die wohl wichtigste Prüfungsmethode darstellt, folgt eine kurze Erläuterung. Zuerst werden die beiden Positionen der Domino-*Tiles* ermittelt. Wenn beide valide und leer sein sollten, werden von beiden Positionen alle berührenden Nachbardominos ermittelt. Wenn mindestens ein Nachbar gefunden werden kann und die Typen passen, liefert diese Methode den Wahrheitswert *true*, ansonsten entsprechend *false*.

Neben dem Bereitsstellen von Prüfungsmethoden, implementiert diese Klasse auch die Methoden um das Board selbst zu verändern.

- Mit der Methode *lay* wird ein Domino auf dem Board platziert. Dabei wird noch einmal geprüft ob es überhaupt passt. Anschließend werden die entsprechenden Werte in das intern gespeicherte *SingleTile*-Array gespeichert.
- Die Methode *remove* ermöglicht es ein gegebenes Domino vom Board zu löschen. Hierbei wird nicht geschaut, ob das Domino tatsächlich an der Stelle positioniert ist, es wird lediglich die Position des Dominos mit dem Wert eines leeren Feldes belegt.
- Um das Stadtzentrum in eine Richtung zu bewegen, wurde die Methode *moveBoard* implementiert. Diese kopiert die Elemente nach einer Prüfung, ob dies überhaupt möglich ist, aber einfach nur entsprechend um.

Um ein Board sachgemäß abspeichern bzw. ausgeben zu können, wurden die beiden Methoden *toString* sowie *toFile* implementiert. Die *toString*-Methode durchläuft das Board und hängt den jeweiligen Enum-Wert an einen StringBuilder an. Die *toFile*-Methode benutzt diese Methode, löscht allerdings vorher den selektierten Domino auf einer der beiden Bänke. Dies ist nötig, da die interne Board-Repräsentation von der Darstellung auf der Gui abweicht.

Listing 22: Board - Konstruktor

```

1 // default constructor
2 public Board(int sizeX, int sizeY) {
3     assert 0 < sizeX && 0 < sizeY;
4     this.sizeX = sizeX;
5     this.sizeY = sizeY;
6     this.cells = initCCinMiddleOfBoard(sizeX, sizeY);
7     this.cells = fillEmptyCellsWithTile(this.cells);
8 }
9
10 // with String evaluation (used for fileIO)
11 public Board(String input) {
12     assert input != null && input.length() > 0;
13     // filled input String
14     String[] lines = input.split("\n");
15     String[][] inputCells = new String[lines.length][];
16     for (int i = 0; i < lines.length; i++) {
17         // split at whitespaces
18         inputCells[i] = lines[i].trim().split("\\s+");
19     }
20
21     // setting the actual SingleTile cells field
22     this.sizeX = inputCells[0].length;
23     this.sizeY = inputCells.length;
24     this.cells = new SingleTile[sizeX][sizeY];
25     for (int y = 0; y < sizeY; y++) {
26         assert inputCells[y].length == sizeX;
27         for (int x = 0; x < sizeX; x++) {
28             String currentElement = inputCells[y][x];
29             if (currentElement.equals(STRING_EMPTY_CELL)) {
30                 cells[x][y] = SingleTile.EC;
31             } else {
32                 this.cells[x][y] = SingleTile.valueOf(currentElement);
33             }
34         }
35     }
36 }
37
38 // copy-Konstruktor
39 public Board(Board other) {
40     assert null != other;
41     this.sizeX = other.sizeX;
42     this.sizeY = other.sizeY;
43     this.cells = new SingleTile[this.sizeX][this.sizeY];
44     for (int y = 0; y < this.sizeY; y++) {
45         for (int x = 0; x < this.sizeX; x++) {
46             this.cells[x][y] = other.cells[x][y];
47         }
48     }
49 }
```

Listing 23: Board - fits

```

1 public boolean fits(Domino domino) {
2     assert null != domino;
3     Pos posFst = domino.getFstPos();
4     Pos posSnd = domino.getSndPos();
5     if (isValidPos(posFst) && isValidPos(posSnd) && isEmpty(posFst) && isEmpty(posSnd)) {
6         List<Pos> fstTouchingNeighbour = genTouchingCells(posFst);
7         List<Pos> sndTouchingNeighbour = genTouchingCells(posSnd);
8         return (!fstTouchingNeighbour.isEmpty() || !sndTouchingNeighbour.isEmpty())
9             && checkIfNeighborsAreValid(domino.getFstVal(), fstTouchingNeighbour)
10            && checkIfNeighborsAreValid(domino.getSndVal(), sndTouchingNeighbour);
11    } else {
12        return false;
13    }
14 }
```

Die künstlichen Spieler setzen die Dominos direkt beim Selektieren um, um im nächsten Schritt den gerade selektierten Domino mit in den Evaluierungsprozess des Selektierens einzubeziehen. Bei der Darstellung des Spiels auf der Gui ist es aber nicht gewollt, dass dieser Domino bereits zu sehen ist. Er wird daher erst später gesetzt. Beim Abspeichern muss man darauf achten, ihn zu löschen um keine Unterschiede zur Gui festzuhalten.

**Botbehavior** Dieses Interface beschreibt alle Methoden, die ein künstlicher Spieler benötigt, um alle Spielzüge ausführen zu können. Außerdem wird die Schnittstelle in der Game-Klasse an manchen Stellen verwendet, um die Bots von dem menschlichen Spieler zu unterscheiden.

**Result** Die Klasse Result verwaltet eine Liste von *ResultRanking*-Objekten und ist in der Lage, die Ergebnisse des Spiels zu verwalten, sowie zu generieren.

**Konstruktor** Der einzige Konstruktor dieser Klasse erwartet einen *Player*-Array als Eingabe und generiert direkt im Aufruf das Ergebnis (siehe Listing 24, S. 47, Result - Konstruktor). Hierbei wird der Playerarray zuerst in eine Liste umgewandelt und per *sort*-Methode des *Collections*-Frameworks sortiert. Dies ist möglich, da die Player-Klasse das *Comparable*-Interface implementiert. Hierbei werden allerdings keine gleichwertigen Spieler zusammengefasst. Um dies nachzuholen erfolgt ein Aufruf der Methode *orderRanking*.

**Methoden** OrderRanking stellt die wohl wichtigste Methode der Klasse dar. Hierbei wird eine Liste eingelesen, welche eine aufsteigend sortierte Reihenfolge der Spieler einhält. Diese wird rekursiv durchlaufen und entsprechend an die Output-Liste angehängt (siehe Listing 25, S. 48, Result - orderRanking). Es wurde hier bewusst ein spezieller Listen-Typ (LinkedList) gewählt, da dieser Typ die Möglichkeit des direkten Zugriffs auf

Listing 24: Result - Konstruktor

```
1 private List<ResultRanking> genResultRankingList(Player[] players) {
2     assert players.length > 0;
3     LinkedList<Player> rankedWithoutEqualTemperedPlayers =
4         new LinkedList<>(Arrays.asList(players));
5     Collections.sort(rankedWithoutEqualTemperedPlayers);
6
7     return orderRanking(new LinkedList<>(), rankedWithoutEqualTemperedPlayers);
8 }
```

das Listenende gewährleistet (Methode *getLast*). Außerdem wird eine Gui-Darstellung generiert. Ansonsten besitzt diese Klasse wenig Funktionalität.

**ResultRanking** Diese Klasse dient als Datenobjekt für die Result-Klasse. Es wird eine Ranking-Position sowie eine Liste an Spielern, welche dasselbe Ranking teilen, gespeichert. Diese werden entsprechend im Konstruktor gesetzt.

**Methoden** Neben den benötigten Gettern und einer Methode um das Ranking auf der Gui darstellen zu können, gibt es keinerlei wichtige Methoden in dieser Klasse.

Listing 25: Result - orderRanking

```

1  private LinkedList<ResultRanking> orderRanking(LinkedList<ResultRanking> output,
2                                              LinkedList<Player> players) {
3      assert null != output && null != players;
4      // exit condition -> sorted when no players are left
5      if (players.isEmpty()) {
6          return output;
7      }
8      // output is empty -> first initialize Result Ranking in List,
9      if (output.isEmpty()) {
10         ResultRanking resRank = new ResultRanking(1);
11         output.add(resRank);
12         return orderRanking(output, players);
13     }
14     // Put highest ranking player in ResultRanking
15     if (output.getLast().isEmpty()) {
16         ResultRanking resRank = output.getLast();
17         resRank.addPlayer(players.removeLast());
18         return orderRanking(output, players);
19     }
20     // last player always most valuable one, matches current ranking
21     if (output.getLast().matchesRank(players.getLast())) {
22         output.getLast().addPlayer(players.removeLast());
23         return orderRanking(output, players);
24     }
25     // last player doesn't match last rank in list
26     output.add(new ResultRanking(output.getLast().getRankingPosition() + 1));
27     return orderRanking(output, players);
28 }
```

## 8.6. differentPlayerTypes

Dieses Package umfasst alle verschiedenen Spielertypen sowie die Klasse zum Instantiieren der Spielerreferenzen (siehe Abbildung 18, S. 49, UML-Darstellung des differentPlayerTypes packages).

**DefaultAIPlayer** Diese Klasse implementiert die in der Aufgabenstellung vorgegebene künstliche Intelligenz. Sie erbt von der Player-Klasse, um auf sämtliche Grundfunktionen und Attribute, die jeder Spieler haben sollte, zugreifen zu können. Außerdem implementiert sie das Interface *Botbehavior* um die KI spezifischen Methoden zu implementieren.

**Methoden** Der Spieler ist in der Lage, zwischen dem initialen Selektieren auf der Bank der aktuellen Runde, und dem auf der Bank der nächsten Runde, zu unterscheiden. Hierzu werden zwei unterschiedliche Methoden verwendet:

Die Methode *doInitialSelect* selektiert einen Domino auf der *aktuellen* Bank (siehe Listing 26, S. 50, DefaultAIPlayer - *doInitialSelect*). Hierbei gilt es zu beachten, dass der Bot direkt nach dem Selektieren bereits den Domino auf seinem Board positioniert. Dies ist erforderlich, um diesen beim Selektieren des nächsten Dominos, bei der Berechnung der Position mit einzubeziehen.

Um einen Domino auf einer Bank zu selektieren wird die Methode *selectFromBank* verwendet (siehe Listing 27, S. 51, DefaultAIPlayer - *selectFromBank*). Diese sucht für jeden Domino der Bank, die am besten geeignete Position, und schreibt diese in ein Choose-Objekt. Dies geschieht für jeden Domino, sodass eine Liste geformt wird. Diese Liste wird darauf untersucht, welches Choose-Objekt insgesamt am besten geeignet ist. Dieses Choose-Objekt wird anschließend weitergereicht, um das Selektieren auf der normalen Bank zu verändern. Im Anschluss wird die interne Board-Repräsentation mit dem Domino belegt, und es wird per Flag entschieden, ob diese Änderung auch auf der Gui dargestellt werden soll (Testdurchlauf oder richtiger Spieldurchlauf). Nachdem alle nötigen Züge veranlasst wurden, wird alles in der Logdatei festgehalten. Die Methode *bestOverallChoose* ruft eine Schleife auf, mit welcher alle Dominos der Bank durchgegangen und an die Methode *genBestChoose* weitergereicht werden (siehe Listing 28, S. 51, DefaultAIPlayer - *genBestChoose*). Diese Methode setzt die Position des Dominos einzeln auf alle Felder des Spielfelds und schaut, ob der Domino passt. Dies ist nur möglich, da vorher ein Deep-Copy angefertigt wurde, um den Domino auf der Bank nicht zu verändern. Falls dies der Fall ist, wird ein neues Choose-Objekt erstellt. Falls dieses



Abbildung 18: UML-Darstellung des differentPlayerTypes packages

Listing 26: DefaultAIPlayer - doInitialSelect

```

1  @Override
2  public Bank doInitialSelect(Bank currBank, int bankOrd) {
3      Bank output = selectFromBank(currBank, bankOrd, true);
4      Domino playerSelectedDomino = currBank.getPlayerSelectedDomino(this);
5      // update board -> has to be done to prevent the bot from laying the
6      // second draft directly on the first domino
7      this.board.lay(playerSelectedDomino);
8      return output;
9  }

```

effizienter sein sollte als das aktuelle maximale Choose-Objekt, so überschreibt dieses das alte (siehe Listing 29, S. 52, DefaultAIPlayer - mostEfficient).

Um einen gesamten Spielzug auszuführen wurde die Methode *doStandardTurn* implementiert. Diese fasst das Selektieren auf der nächsten Bank, sowie das Darstellen des zuvor gewählten Dominos auf der Gui, zusammen. Für den Fall, dass das Spielende erreicht wurde und der letzte Zug ausgeführt werden muss, gibt es die Methode *doLastTurn*. Diese agiert sehr ähnlich zur Methode *doStandardTurn*, es wird allerdings das Selektieren auf der nächsten Bank weggelassen.

**HumanPlayer** Da sämtliche Spielaktionen durch die Gui-Interaktionen bestimmt werden, befindet sich in dieser Klasse keinerlei weitere Funktionalität. Für diese Spielerart reichen die durch den abstrakten Supertyp *Player* implementierten Methoden völlig aus.

**playerType** Dieser Enum dient als Blaupause für sämtliche Spielertypen. Er kann anstelle von konkreten Instanzen als Platzhalter dienen. Des Weiteren bietet er eine Methode, die als statische Factory zum Erstellen von Spielerreferenzen dient.

Listing 27: DefaultAIPlayer - selectFromBank

```

1  @Override
2  public Bank selectFromBank(Bank bank, int ordBank, boolean displayOnGui) {
3      if (null == bank || bank.isEmpty()) {
4          return bank;
5      }
6      Bank bankCopy = bank.copy();
7      List<Choose> bestChoosesForEachPossibleBankSlot = bestChooseForEachDom(bankCopy);
8      Choose overallBestChoose = bestOverallChoose(bankCopy,
9          bestChoosesForEachPossibleBankSlot);
10     doSelect(bank, overallBestChoose);
11     updateBoard(ordBank, displayOnGui, overallBestChoose);
12
13     // log selection
14     String roundIdentifier = ordBank == 0 ? "current" : "next";
15     Logger.getInstance().printAndSafe("\n" + String.format(
16         Logger.SELECTION_LOGGER_FORMAT,
17         getName(), overallBestChoose.getDomWithPosAndRot().toString(),
18         overallBestChoose.getIdxOnBank(), roundIdentifier));
19
20     // return the bank, although bank reference is modified internally
21     // (just to make sure it is evident, pos and rot modified)
22     return bank;
23 }
```

Listing 28: DefaultAIPlayer - genBestChoose

```

1  private Choose genBestChoose(Domino domino, int bankSlotIndex) {
2      Choose currChoose;
3      Choose maxChoose = null;
4      for (int y = 0; y < this.board.getSizeY(); y++) {
5          for (int x = 0; x < this.board.getSizeX(); x++) {
6              domino.setPos(new Pos(x, y));
7              for (int i = 0; i < Board.Direction.values().length; i++) {
8                  if (this.board.fits(domino)) {
9                      currChoose = genChoose(domino, bankSlotIndex);
10                     maxChoose = mostEfficient(maxChoose, currChoose);
11                 }
12                 domino.incRot();
13             }
14         }
15     }
16     return null == maxChoose ? genChoose(domino.setPos(new Pos(0, 0)),
17                                         bankSlotIndex) : maxChoose;
18 }
```

Listing 29: DefaultAIPlayer - mostEfficient

```

1 private Choose mostEfficient(Choose fstChoose, Choose sndChoose) {
2     Choose res;
3     if (null == fstChoose) {
4         res = sndChoose;
5     } else if (null == sndChoose) {
6         res = fstChoose;
7     } else if (fstChoose.getPotentialPointsOnBoard()
8             > sndChoose.getPotentialPointsOnBoard()) {
9         res = fstChoose;
10    } else if (fstChoose.getPotentialPointsOnBoard()
11        < sndChoose.getPotentialPointsOnBoard()) {
12        res = sndChoose;
13    } else {
14        // tie
15        int fstSingleCellCount = countSingleCells(fstChoose);
16        int sndSingleCellCount = countSingleCells(sndChoose);
17
18        if (fstSingleCellCount < sndSingleCellCount) {
19            res = fstChoose;
20        } else if (fstSingleCellCount > sndSingleCellCount) {
21            res = sndChoose;
22        } else {
23            res = fstChoose.getDomWithPosAndRot().compareTo(
24                sndChoose.getDomWithPosAndRot()) <= 0 ? fstChoose
25                : sndChoose;
26        }
27    }
28    return res;
29 }
30
31 private int countSingleCells(Choose choose) {
32     assert null != choose;
33     List<District> temp = updateDistricts(this.districts, choose.getDomWithPosAndRot());
34     int out = 0;
35     for (District currDistrict : temp) {
36         if (currDistrict.getTilePositions().size() == 1) {
37             out++;
38         }
39     }
40     return out;
41 }
```

## 8.7. logicTransfer

**Converter** Diese Klasse beschäftigt sich mit dem Generieren sämtlicher Teilkomponenten der Game-Klasse aus einem gegebenen String. Neben dem Erstellen der Objekte ist sie außerdem für die Überprüfung der Syntax verantwortlich, das Laden der Dateien wurde hier allerdings explizit abgegrenzt und findet in der *Loader*-Klasse statt. Um einen String zu interpretieren wird die Methode *readStr* verwendet (siehe Listing 30, S. 54, Converter - *readStr*).

1. Schritt: Es wird eine grobe Struktur geschaffen, um sämtliche Teilbereiche analysieren zu können. Hierbei wird ein zweidimensionales Array gebildet, welches in der ersten Dimension jeweils den Tag-Bezeichner des jeweiligen Objektes (<Spielfeld...>, <Bänke> oder <Stapel>) und in der zweiten die wirklichen Daten enthält.
2. Schritt: Die einzelnen Felder des Arrays werden auf ihre syntaktische Richtigkeit überprüft.
3. Schritt: Die Strings in den Array-Feldern werden zu Objekten konvertiert.

**Rohdaten aufteilen** Bevor es zur wirklichen Aufteilung der Rohdaten kommt, wird initial erst einmal überprüft ob die Bezeichner stimmen oder nicht. Dies wird über folgenden regulären Ausdruck getan: (siehe MATCHES\_TAGS)

```
"(<Spielfeld[^>]*>\n(?s)[^<>]*)*<Bänke>\n(?s)[^<>]*<Beute1>\n[^<>]*"
```

Hierbei wird abgeprüft, ob die Bezeichner Namen stimmen und mit spitzen Klammern eingeleitet sowie beendet werden. Nach dieser ersten Prüfung wird das Array mit den Rohdaten erstellt. Hierzu wird die Methode *genDescriptiveField* aufgerufen (siehe Listing 31, S. 55, Converter - *genDescriptiveField*). Hier werden im ersten Schritt die Blöcke in die einzelnen Komponenten aufgebrochen, es wird nämlich bei jeder öffnenden spitzen Klammer geteilt. Im zweiten Schritt werden diese noch weiter verfeinert, indem bei jeder schließenden spitzen Klammer geteilt wird. Man erlangt also grob die Darstellung aus Tabelle 1. Anschließend werden die Bezeichner aus den ersten Feldern herausgelöst (es werden also zum Beispiel alle spitzen Klammern verworfen).

Im nächsten Schritt werden die Daten über die Funktion *fillFieldsWithDescriptiveBlocks* interpretiert (siehe Listing 32, S. 56, Converter - *fillFieldsWithDescriptiveBlocks*). Hier werden die im vorherigen Schritt erzeugten Felder durchlaufen, es wird jeweils das Feld dem Bezeichner zugeordnet. Für jeden unterschiedlichen Bezeichner gibt es eine eigene Methode zum interpretieren der Daten. Wenn man zum Beispiel mit *i == 0* auf ein Feld mit einem Spielfeld-Bezeichner im ersten Feld des zweidimensionalen Arrays zugreift, wird im Case-Verteiler der BOARD\_IDENTIFIER greifen und es wird zuerst die Spielfeld-Syntax überprüft. Hierbei spielen Dinge wie Leerzeichen oder die Dimensionen des Spielfeldes eine Rolle, aber auch invalide Zellen werden abgefangen. Da in meinem Modell die Spieler jeweils ein Feld mit dem Spielfeld besitzen, habe ich den Besitzer des Spielfeldes gleich mit initialisiert. So wird die Liste der Distrikte gleich mit aufgebaut. Hierzu wird die Methode *convertStrToPlayerWithDefaultOccupancy* aufgerufen. Diese ist eine Hilfsmethode und macht nichts anderes, als anhand des übergebenen Indices

Listing 30: Converter - readStr

```

1 public String readStr(GUIConnector gui, String input) {
2     try {
3         if (input == null || input.length() == 0) {
4             throw new IOException(UNSUCCESSFUL_READ_MESSAGE);
5         }
6         // Tag syntax roughly checked -> further analysis further down the line
7         if (!input.matches(MATCHES_TAGS)) {
8             System.out.println(input);
9             throw new WrongTagException();
10        }
11        String[][] descriptionBlocks = genDescriptiveField(input);
12        fillFieldsWithDescriptiveBlocks(descriptionBlocks, gui);
13        return SUCCESSFUL_READ_MESSAGE;
14    } catch (Exception e) {
15        return e.getMessage();
16    }
17 }
```

festzustellen, ob es sich um einen Bot oder um den menschlichen Spieler handelt. Um den Spieler letztendlich zu initialisieren wird die statische Factory-Methode des *Player-Type*-Enums aufgerufen. Wieso hier eine Factory benutzt wird, wird im Abschnitt 8.6 auf S. 50 erläutert.

Wenn nach und nach alle Felder des Arrays mit Spielfeldern abgearbeitet wurden, folgen die Bänke. Auch hier wird wieder zuerst die Syntax mit der Methode *checkBankSyntax* überprüft um anschließend per *convertStrToBanks* die benötigten Bänke zu generieren (siehe Listing 33, S. 57, Converter - convertStrToBanks). In dieser Methode wird der Eingabe-String der Bänke am Zeilenumbruch geteilt und entsprechend dem Bank-Konstruktor übergeben. Nach dem Generieren der Bänke in dem Case-Verteiler der *fillFields...-Methode* wird allerdings ebenfalls die aktuelle Bankposition der Bank für die jeweils aktuelle Runde berechnet. Diese wird im Spiel benötigt um zu kennzeichnen, welcher Spieler gerade am Zug ist. Dies soll zwar beim Speichern bzw. Laden stets der menschliche Spieler sein, dennoch ist es sinnvoll, hier ein Feld entsprechend zu setzen, um sich später im Spiel eine weitere Suche nach der Bankposition zu sparen.

Um den Beutel zu generieren wird genauso wie bisher vorgegangen. Es wird per *checkStackSyntax* die Beutelsyntax überprüft. Anschließend wird die Zeichenkette mit den Beutelementen in die einzelnen Elemente zerteilt und dem Domino Konstruktor über Umwege weitergereicht, denn es muss vorher erst einmal ein Tiles-Objekt erzeugt werden, welches dem Domino übergeben werden kann (siehe Listing 34, S. 57, Converter - convertStrToStack).

| Beispiel    |                             |
|-------------|-----------------------------|
| Bezeichner  | Daten                       |
| Spielfeld 1 | - - - - -, etc.             |
| Spielfeld 2 | - - - - -, etc.             |
| Spielfeld 3 | - - - - -, etc.             |
| Spielfeld 4 | - - - - -, etc.             |
| Bänke       | 3 A1H0,- A1H0,- A1H0,1 P0S1 |
| Stapel      | P0P0,H0H0,P0S0,H0A0         |

Tabelle 1: 1. Schritt: Rohdaten grob aufgeteilt

Listing 31: Converter - genDescriptiveField

```

1 public String[][] genDescriptiveField(String input) throws WrongTagException {
2     List<String> blocks = new LinkedList<>();
3     // overall sections (board/banks/stack) are seperated
4     for (String currBlock : input.split("<")) {
5         blocks.add(currBlock);
6     }
7     blocks.remove(0); // First element may be empty because of split()
8
9     // Data seperated from Identifier
10    String[][] output = new String[blocks.size()][2];
11    for (int i = 0; i < blocks.size(); i++) {
12        output[i][DESCRIPTION_IDX] = genTag(blocks.get(i));
13        output[i][DATA_IDX] = genData(blocks.get(i));
14    }
15
16    return output;
17 }
```

Listing 32: Converter - fillFieldsWithDescriptiveBlocks

```

1 public void fillFieldsWithDescriptiveBlocks(String[][] descriptionBlocks,
2                                             GUIConnector gui)
3     throws WrongTagException, WrongBoardSyntaxException, WrongBankSyntaxException,
4     WrongStackSyntaxException {
5     // TODO delete before final commit
6     int[] dimensions = new int[]{NOT_INITIALIZED, NOT_INITIALIZED};
7     for (int i = 0; i < descriptionBlocks.length; i++) {
8         switch (descriptionBlocks[i][DESCRIPTION_IDX]) {
9             case BOARD_IDENTIFIER:
10                 dimensions = checkBoardSyntax(dimensions, descriptionBlocks[i][DATA_IDX]);
11                 this.players.add(i, convertStrToPlayerWithDefaultOccupancy(
12                     descriptionBlocks[i][DATA_IDX], i, gui));
13                 break;
14             case BANK_IDENTIFIER:
15                 checkBankSyntax(descriptionBlocks[i][DATA_IDX], this.players.size());
16                 Bank[] banks = convertStrToBanks(descriptionBlocks[i][DATA_IDX]);
17                 this.currentBank = banks[Game.CURRENT_BANK_IDX];
18                 this.nextBank = banks[Game.NEXT_BANK_IDX];
19                 this.currBankPos = 4 - descriptionBlocks[Game.CURRENT_BANK_IDX].length;
20                 break;
21             case STACK_IDENTIFIER:
22                 checkStackSyntax(descriptionBlocks[i][DATA_IDX]);
23                 this.stack = convertStrToStack(descriptionBlocks[i][DATA_IDX]);
24                 break;
25             default:
26                 throw new WrongTagException(String.format(
27                     WrongTagException.DEFAULT_MESSAGE,
28                     descriptionBlocks[i][DESCRIPTION_IDX]));
29         }
30     }
31 }
```

Listing 33: Converter - convertStrToBanks

```

1 private Bank[] convertStrToBanks(String input) {
2     assert null != input && input.contains("\n");
3     // both banks empty
4     if (input.length() == 0 || "\n".equals(input)) {
5         return new Bank[]{{new Bank(this.players.size()),
6             new Bank(this.players.size())};
7     }
8     String[] bothBanks = input.split("\n");
9     Bank[] output = new Bank[2];
10    output[Game.CURRENT_BANK_IDX] = new Bank(bothBanks[0], this.players, new Random());
11    // determines if the next round bank is filled
12    if (bothBanks.length > 1) {
13        output[Game.NEXT_BANK_IDX] = new Bank(bothBanks[1], this.players, new Random());
14    } else {
15        output[Game.NEXT_BANK_IDX] = new Bank(this.players.size());
16    }
17    return output;
18 }
```

Listing 34: Converter - convertStrToStack

```

1 private List<Domino> convertStrToStack(String input) {
2     // Stay maybe empty -> must be checked
3     List<Domino> output = new LinkedList<>();
4     if (0 < input.length()) {
5         String[] dominosStr = input.split(",");
6         // Last element is \n doesn't have to be evaluated
7         String temp;
8         for (int i = 0; i < dominosStr.length; i++) {
9             temp = dominosStr[i].substring(0, 4);
10            output.add(new Domino(Tiles.fromString(temp)));
11        }
12    }
13    return output;
14 }
```

**Game** Die Game-Klasse verwaltet sämtliche Spieleraktionen und stellt die benötigten Bänke und den Beutel bereit. Sie implementiert das *GUI2Game*-Interface um in der Lage zu sein, die Gui-Interaktionen des Benutzers zu verarbeiten.

**Konstruktor** In der Klasse werden die beiden Bänke, der Beutel, der Domino im Rotationsfeld des Spielers, eine Gui-Schnittstelle sowie das Feld, welches sich gerade im Fokus befindet, verwaltet. All diese Attribute werden im Konstruktor gesetzt. Hierbei gibt es drei verschiedene Ausprägungen (siehe Listing 35, S. 59, Game - Konstruktoren). Die erste Version bekommt lediglich ein gui-Objekt, sowie die Anzahl der Spieler, gegeben. Wie in Abschnitt 7 - Problemanalyse und Realisation bereits beschrieben, wurde die letztendliche Initialisierung der Spieler in die *loadGuiAfterLoadingFile*-Methode bzw. die *PlayerType*-Klasse verlagert.

In der zweiten Ausprägung wird ein String gegeben, welcher vom Converter interpretiert wird. Die Idee dabei war, sämtliche Verarbeitungsschritte zur Konvertierung der Daten in die jeweiligen Objekte dieser Klasse zu verlagern. Über diverse Getter können diese vom Spiel „abgeholt“, werden. Um die generierten Werte zu setzen wird eine Hilfsfunktion namens *initTestingLoadingConstructor* verwendet, welche nichts anderes tut, als die Attribute der Klasse zu belegen. Anschließend wird die Gui mit einer entsprechenden Methode beschrieben und es werden die Boards der Spieler intern mit den bereits selektierten Dominos belegt. Dies muss wie bereits angesprochen geschehen, damit der Spieler in der Lage ist, bei der Evaluierung der Dominos auf der Bank den Domino der momentan (auf der gui) noch nicht auf der Bank liegt, mit in die Überlegungen einzubeziehen. Sämtliche Verarbeitungsschritte werden dem Logger noch kenntlich gemacht.

**Methoden** Wichtige Methoden (betreffen vor allem den menschlichen Spieler):

- startGame
- selectDomOnCurrBank
- selectDomOnNextBank
- setOnBoard
- moveBoard

startGame (siehe Listing 43): Im ersten Schritt werden sämtliche Spieler mit `emph-createNewPlayers` initialisiert. Die muss der Gui mit dem Aufruf `emphupdatePlayers` mitgeteilt werden. Anschließend wird der Beutel per *fill*-Methode gefüllt. Nachdem dies geschehen ist, wird die Bank der aktuellen Runde mit zufällig gezogenen Dominos befüllt. Auch dies wird wieder auf der Gui dargestellt. Nachdem der aktuelle Bank-Slot mit dem Wert 0 beschrieben wurde, wird dem Logger mitgeteilt, dass das Spiel gestartet wurde.

Listing 35: Game - Konstruktoren

```

1 // Konstruktor im normalen Spiel
2 public Game(GUIConnector gui, int playerCnt) {
3     this.gui = gui;
4     this.players = new Player[playerCnt];
5     this.currentRoundBank = new Bank(playerCnt);
6     this.nextRoundBank = new Bank(playerCnt);
7     this.stack = new LinkedList<>();
8     this.currBankIdx = 0;
9     this.currDomino = null;
10    this.currField = PossibleField.CURR_BANK;
11 }
12
13 // Konstruktor zum Einlesen einer Datei
14 public Game(GUIConnector gui, String input) {
15     this.gui = gui;
16     Converter gameContent = new Converter();
17     String returnMessage = gameContent.readStr(gui, input);
18     System.out.println(returnMessage);
19     if (Converter.SUCCESSFUL_READ_MESSAGE == returnMessage) {
20         initTestingLoadingConstructor(gui, gameContent.getPlayers(),
21             gameContent.getCurrBankPos(), gameContent.getCurrentBank(),
22             gameContent.getNextBank(), gameContent.getStack(), null);
23
24         Board humanBoard = this.players[HUMAN_PLAYER_IDX].getBoard();
25         loadGuiAfterLoadingFile(generateDefaultPlayerTypeArray(this.players.length),
26             humanBoard.getSizeX(), humanBoard.getSizeY());
27         for (Player player : this.players) {
28             if (player instanceof BotBehavior) {
29                 ((BotBehavior) player).updateSelectedDom(this.currentRoundBank);
30                 ((BotBehavior) player).updateSelectedDom(this.nextRoundBank);
31             }
32         }
33     } else {
34         this.gui.showPopUp(returnMessage);
35     }
36     Logger.getInstance().printAndSafe(Logger.GAME_SEPARATOR + "\nLoading process: "
37         + returnMessage);
38 }
```

Listing 36: Game - startGame

```

1  @Override
2  public void startGame(PlayerType[] playerTypes, int sizeX, int sizeY) {
3      // instanciate players with given playertypes
4      this.players = createNewPlayers(playerTypes, sizeX, sizeY);
5
6      for (int i = 0; i < this.players.length; i++) {
7          this.gui.updatePlayer(this.players[i]);
8      }
9
10     // fill stack
11     this.stack = Domino.fill(this.stack);
12
13     // fill current bank
14     this.stack = this.currentRoundBank.randomlyDrawFromStack(this.stack);
15     this.gui.setToBank(CURRENT_BANK_IDX, this.currentRoundBank);
16
17     this.currBankIdx = 0;
18     this.gui.showWhosTurn(HUMAN_PLAYER_IDX);
19
20     Logger.getInstance().printAndSafe(Logger.GAME_SEPARATOR + "\nStarted new game\n");
21 }
```

selectDomOnCurrBank (siehe Listing 37): Dieser Schritt wird während der initialen Selektierungsphase benötigt. Die Bank der aktuellen Runde selektiert den Bank-Slot am gegebenen Index mit der menschlichen Spielerreferenz. Dies wird wieder auf der Gui sowie im Logger dargestellt. Anschließend selektieren alle Bots mithilfe der Methode *botsDoInitialSelect* jeweils einen Domino auf der Bank der aktuellen Runde. Um die Bänke für die nächste Runde vorzubereiten, wird die Bank mit zufällig gezogenen Dominos gefüllt. Da nun eine reguläre Runde startet, ziehen sämtliche Bots, welche einen niedrigen Wertigen Domino gegenüber dem menschlichen Spieler gezogen haben, ihren Domino auf ihr Feld und selektieren einen neuen Domino auf der Bank der nächsten Runde. Um dem Benutzer klar zu machen, dass er im nächsten Schritt nur auf der nächsten Bank einen Domino selektieren darf, wird das Attribut *currField* entsprechend gesetzt. Dies wird auf der Gui anhand eines Gaussian-Blur-Effekts dargestellt. Die Bank der aktuellen Runde wird mit dem Aufruf der Methode *blurOtherFields* leicht verschwommen dargestellt.

selectDomOnNextBank (siehe Listing 38): Diese Methode ist Teil eines standardmäßigen Zuges während des Spielverlaufs. Ähnlich wie beim Selektieren auf der Bank der nächsten Runde, wird hier die entsprechende Bank aufgerufen und mit der Methode *selectEntry* der gewünschte Domino ausgewählt. Dies wird auf der Gui dargestellt. Anschließend wird allerdings der gewählte Domino der Vorrunde in der Drehbox des Benutzers dargestellt. Auch hier werden mit der Methode *blurOtherFields* die gerade nicht betrachteten Selektionsfelder verschwommen dargestellt.

Listing 37: Game - selectDomOnCurrBank

```

1  @Override
2  public void selectDomOnCurrBank(int idx) {
3      if (PossibleField.CURR_BANK == this.currField &&
4          this.currentRoundBank.isNotSelected(idx)) {
5          // update human player selection
6          this.currentRoundBank.selectEntry(this.players[HUMAN_PLAYER_IDX], idx);
7          this.gui.selectDomino(CURRENT_BANK_IDX, idx, HUMAN_PLAYER_IDX);
8          Logger.getInstance().printAndSafe(String.format(Logger.SELECTION_LOGGER_FORMAT,
9              this.players[HUMAN_PLAYER_IDX].getName(),
10             this.currentRoundBank.getDomino(idx), idx,
11             "current"));
12
13         botsDoInitialSelect();
14         randomlyDrawNewDominosForNextRound();
15         this.currBankIdx = botsDoTheirTurn(this.currBankIdx);
16         this.currField = PossibleField.NEXT_BANK;
17         this.gui.blurOtherFields(this.currField);
18     } else {
19         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER
20             + "\nHUMAN tried to select a domino from the " + "current bank\n"
21             + Logger.ERROR_DELIMITER + "\n");
22     }
23 }
```

Listing 38: Game - selectDomOnNextBank

```

1  @Override
2  public void selectDomOnNextBank(int idx) {
3      if (PossibleField.NEXT_BANK == this.currField
4          && this.nextRoundBank.isNotSelected(idx)) {
5          assert null == this.currDomino;
6          Player humanPlayer = this.players[HUMAN_PLAYER_IDX];
7          // Human player selects domino on next bank
8          this.nextRoundBank.selectEntry(humanPlayer, idx);
9          this.gui.selectDomino(NEXT_BANK_IDX, idx, HUMAN_PLAYER_IDX);
10         setToChooseBox(this.currentRoundBank.getPlayerSelectedDomino(humanPlayer));
11         this.currField = PossibleField.CURR_DOM;
12         this.gui.blurOtherFields(this.currField);
13         Logger.getInstance().printAndSafe("\n" + String.format(
14             Logger.SELECTION_LOGGER_FORMAT,
15             humanPlayer.getName(), this.nextRoundBank.getDomino(idx).toString(),
16             idx, "next"));
17     } else {
18         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER
19             + "\nHUMAN tried to make an impossible bank " + "selection\n"
20             + Logger.ERROR_DELIMITER + "\n");
21     }
22 }
```

Listing 39: Game - setOnBoard

```

1  @Override
2  public void setOnBoard(Pos pos) {
3      if (PossibleField.CURR_DOM == this.currField) {
4          this.currDomino.setPos(new Pos(pos.x(), pos.y()));
5          this.players[HUMAN_PLAYER_IDX].showOnBoard(currDomino);
6          this.currField = PossibleField.NEXT_BANK;
7          this.gui.blurOtherFields(this.currField);
8          setupCurrDomAndBotsDoTurn();
9      } else {
10         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER + "HUMAN tried to make "
11                                         + "an impossible bank selection\n" + Logger.ERROR_DELIMITER + "\n");
12     }
13 }
```

setOnBoard (siehe Listing 39: Eine Methode um den Domino in der Drehbox des Benutzers auf dem Board zu platzieren (siehe Listing 39, S. 62, Game - setOnBoard). Dazu wird die Position des Dominos angepasst und die Methode *showOnBoard* mit aufgerufen. Da diese Methode neben dem Zeigen auf der Gui ebenfalls noch die Distrikte entsprechend erweitert, ist es nicht mehr nötig, dies an dieser Stelle zu tun. Ansonsten werden auch hier die nicht betrachteten Felder verschwommen dargestellt, sämtliche Bots führen ihre Züge aus und die kommende Runde wird vorbereitet.

moveBoard (siehe Listing 40): Diese Methode verschiebt das Board des Spielers. In diesem Entwurf sind sämtliche benötigte Schritte in dieser Klasse implementiert. Falls es später allerdings dazu kommen sollte, dass ein künstlicher Spieler ebenfalls in der Lage sein soll dies zu tun, muss diese Funktionalität in die abstrakte Spieler-Klasse verschoben werden. Diese Methode geht jedoch wie folgt vor: Es ist eine Richtung angegeben, in die das Board bewegt werden soll. Die Board-Klasse besitzt bereits eine Methode die überprüft, ob es möglich ist, es in diese Richtung zu bewegen. Nach der Überprüfung wird mithilfe der Methode *moveBoard* der Board-Klasse das Spielfeld bewegt. Dies wird zuletzt der Gui mitgeteilt und entsprechend geloggt. Falls es nicht möglich sein sollte, das Board in die Richtung zu bewegen, wird auch dies dem Logger mitgeteilt.

disposeCurrDomino (siehe Listing 41, S. 63, Game - disposeCurrDomino): Hierbei wird sowohl intern als auch auf der Gui der Domino innerhalb der Drehbox des Benutzers entfernt (siehe Listing 41, S. 63, Game - disposeCurrDomino). Hierfür wird lediglich ein Nullpointer als aktueller Domino der *setToChooseBox*-Methode übergeben. Desweiteren wird das gerade fokussierte Feld neu gesetzt, die Bots ziehen und es wird die nächste Runde vorbereitet.

Listing 40: Game - moveBoard

```

1  @Override
2  public void moveBoard(Board.Direction dir) {
3      try {
4          Player humanPlayer = this.players[HUMAN_PLAYER_IDX];
5          if (humanPlayer.getBoard().canMoveBoardToDir(dir) && (humanPlayer instanceof
6              HumanPlayer)) {
7              HumanPlayer humanInstance = (HumanPlayer) humanPlayer;
8              // only Human player has a setter for the board -> need to cast
9              humanInstance.updateBoard(humanInstance.getBoard().moveBoard(dir));
10             this.gui.updatePlayer(players[HUMAN_PLAYER_IDX]);
11             Logger.getInstance().printAndSafe(String.format(Logger.CC_DRAG_LOGGER_FORMAT,
12                 humanInstance.getName(),
13                 humanInstance.getBoard().findPos(SingleTile.CC)));
14         } else {
15             Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER
16                 + "\nHUMAN tried to move board in an impossible direction\n"
17                 + Logger.ERROR_DELIMITER + "\n");
18         }
19     } catch(RuntimeException e) {
20         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER
21             + "\nNot possible to move board.\n"
22             + Logger.ERROR_DELIMITER + "\n");
23     }
24 }
```

Listing 41: Game - disposeCurrDomino

```

1  @Override
2  public void disposeCurrDomino() {
3      if (PossibleField.CURR_DOM == this.currField) {
4          Logger.getInstance().printAndSafe(String.format(Logger.DISMISSAL_LOGGER_FORMAT,
5              "HUMAN", this.currDomino.toString()));
6          setToChooseBox(null);
7          this.currField = PossibleField.NEXT_BANK;
8          setupCurrDomAndBotsDoTurn();
9          this.gui.blurOtherFields(currField);
10     } else {
11         Logger.getInstance().printAndSafe(Logger.ERROR_DELIMITER + "\nHUMAN tried to "
12             + "dispose the current domino\n" + Logger.ERROR_DELIMITER + "\n");
13     }
14 }
```

Listing 42: Enum - PossibleField

```
1 public enum PossibleField {  
2     NEXT_BANK, CURR_BANK, CURR_DOM  
3 }
```

**Exceptions** Im Projekt wurden mehrere eigene Exception-Klassen implementiert, welche in der Converter-Klasse verwendet werden. Bei invalider Syntax in den einzulesenden Dateien werden diese geworfen. Hierzu wurden die Fehler in die drei Hauptkomponenten (Board, Bank und Beutel) aufgeteilt und jeweils mit einer eigenen Exception-klasse versehen. Um schon relativ früh im Interpretationsprozess Syntaxfehler zu erkennen, wird eine weitere Exception-Klasse namens *WrongTagSyntax* verwendet, welche nur geworfen wird, wenn die Beschreibung der Daten bereits fehlerhaft sein sollte.

**GUI2Game** Dieses Interface bildet die Schnittstelle von der Gui zur Game-Klasse. Die Game-Klasse implementiert dieses und der FXMLDocumentController besitzt ein Attribut dieses Typs. Die Schnittstelle umfasst alle Aktionen, die ein Benutzer auf der Gui tätigen kann.

**GUIConnector** Dieses Interface beschreibt die Schnittstelle, welche vom Spiel selbst benötigt wird, um Veränderungen des Spielstands auf der Gui darstellen zu können. Die Game-Klasse besitzt ein Attribut dieses Typs. Die JavaFX-Klasse im *gui*-Package stellt letztendlich eine Implementierung bereit, indem sie dieses Interface implementiert.

**PossibleField** Dieser Enum umfasst alle möglichen Auswahlbereiche auf der Gui (siehe Listing 42, S. 64, Enum - PossibleField). Er wird benötigt um festzulegen, mit welchem Bereich der Gui der Benutzer im Moment interagieren kann und mit welchen nicht.

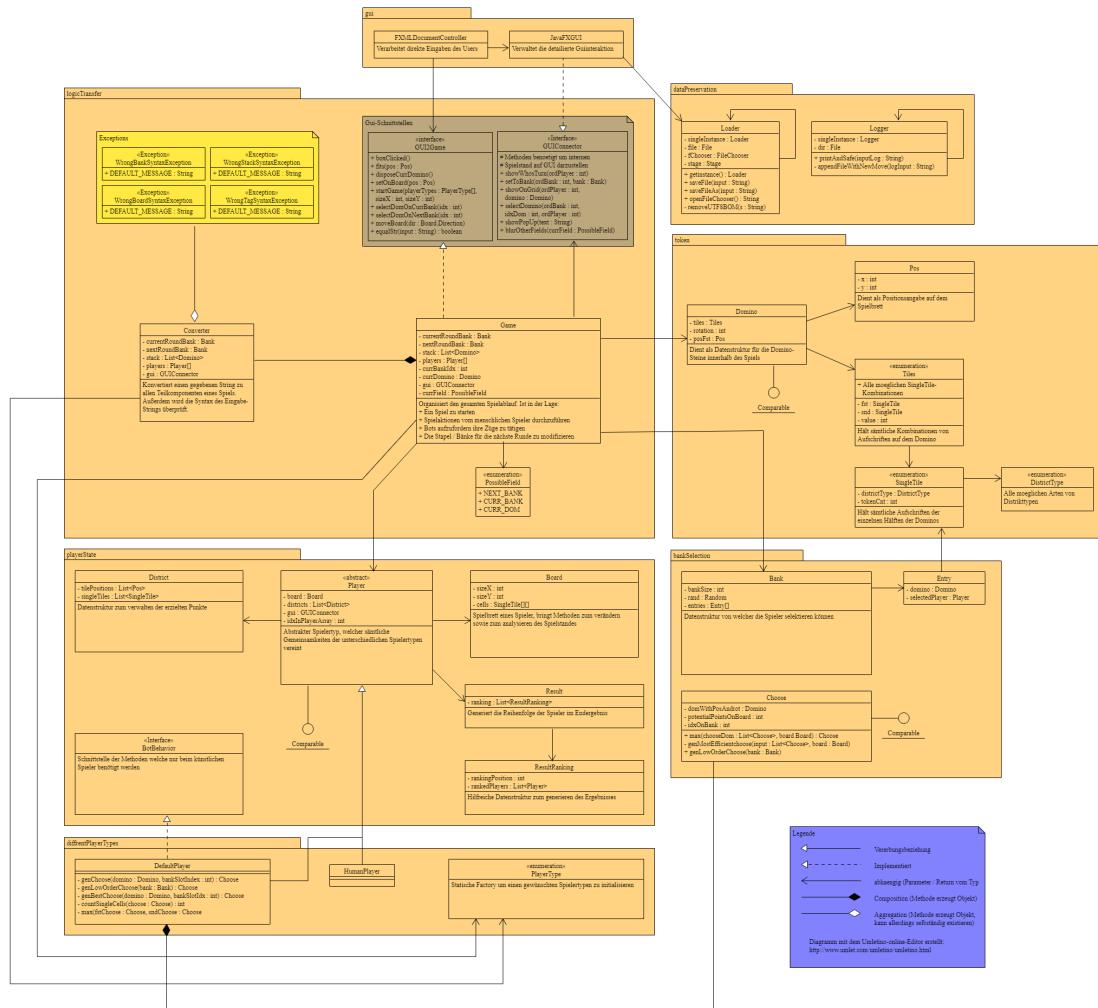


Abbildung 19: Vereinfachte UML-Darstellung des gesamten Projekts

## 9. Programmorganisationsplan

Im folgenden Abschnitt wird eine Gesamtübersicht über das Zusammenspiel sämtlicher Klassen gegeben. Das dazu verwendete UML-Diagramm (siehe Abbildung 19, S. 65) ist leider relativ groß geworden. Falls Sie eine vergrößerte Version in ihrem Bildbetrachter sehen möchten, folgen Sie bitte dem Link in der Lesezeichen- bzw. Anlagenübersicht Ihres Pdf-Readers<sup>3</sup>. Ansonsten liegt das Bild diesem Projekt auch bei (Name *PP18Vereinfacht.png*).

**Start einer herkömmlichen Runde** Die Klasse, welche die gesamte Anwendung startet, wurde im UML-Diagramm nicht eingezeichnet, da sie nichts anderes tut, als das

<sup>3</sup><https://www.acrobat-tutorials.de/2013/03/26/dateianlagen-und-seitenanlagen-in-pdf-dokumenten/>

entsprechende FXML-Dokument zu laden und ein neues Spiel zu starten. Mit dem Laden des FXML-Dokuments wird allerdings auch der dazugehörige Controller mit einer initialize-Methode geladen. Diese instantiiert ein JavaFX-Objekt sowie das Spiel selbst. Im Spiel werden alle Teilkomponenten einzeln initialisiert. Es werden demnach alle Spieler, die Gui-Schnittstelle, die Bänke, der Beutel, der aktuelle Bankindex und das gerade betrachtete Feld gesetzt, wobei für dynamische Datenstrukturen hier lediglich ein Typ festgesetzt wird. Die Spieler nehmen hierbei eine besondere Funktion ein, denn sie verwahren intern ihre eigenen Spielfelder, Punkte und Gui-Aktionen. Die Schnittstelle mit dem Spiel soll so minimal wie möglich gehalten werden, damit der Spieler möglichst autonom handeln kann, um die Erweiterung in einem späteren Entwicklungsstadium mit anderen Spielertypen zu ermöglichen. Im Konstruktor selbst werden die Spieler allerdings noch nicht instantiiert. Dies geschieht erst in der startGame-Methode. Ähnlich wie bei den anderen dynamischen Datenstrukturen werden hier die eigentlichen Werte gesetzt. Dazu wird die Hilfsmethode *createNewPlayers* aufgerufen. Diese bekommt einen Array an Spielertypen als „Blaupause“ für die zu erstellenden Spieler. In der Schleife wird die statische Factory-Methode aufgerufen, welche in der Lage ist, einen gegebenen Spielerotypen zu interpretieren und in eine Spielerinstanz umzuwandeln. Der Stack wird nach dem Initialisieren der Spieler ebenfalls gefüllt. Wieso dieser Aufwand betrieben wurde, wird im Abschnitt 7.2, auf Seite 24 - Alternative Spielerinitialisierung beschrieben.

**Selektieren** Um einen Domino zu selektieren, benötigt es eine Spielerreferenz und den Index des Dominos, welcher selektiert werden soll. Die Bänke bestehen aus Entry-Objekten. Ein Entry-Objekt hält einen Domino sowie eine Spielerreferenz. Auf der Bank ist es möglich, einen bisher noch nicht selektierten Eintrag (gekennzeichnet durch einen Null-Pointer an der Stelle der Spielerreferenz des Entry-Objektes) mit der eigenen Spielerreferenz zu überschreiben und somit zu selektieren. Wie bereits erwähnt, können die Bots ihre Züge selbstständig ausführen, wenn sie vom Spiel dazu aufgefordert und mit den entsprechenden Daten versorgt werden. Dabei bedienen Sie sich noch einer Hilfsstruktur (der *Choose*-Klasse) um die Dominos zu untersuchen. Da der menschliche Spieler zu großen Teilen von der Eingabe des Benutzers auf der Gui abhängt, befinden sich sämtliche Funktionalitäten dieses Spielertyps in der Game-Klasse selbst.

**Ablegen** Mit dem Ablegen eines Steins auf dem Spielfeld muss nicht nur das spelereigene Board erweitert werden, sondern auch die Liste der Distrikte. Dies geschieht bei beiden Spielertypen aber bereits im abstrakten Supertyp Player und muss nicht gesondert behandelt werden.

**Berechnung des Ergebnisses** Um das Ergebnis zu berechnen wird eine Instanz des Result-Objektes instantiiert. Diese erstellt selbstständig eine Liste mit dem Datenobjekt *ResultRanking*. Diese Art des Rankings erlaubt es mehreren Spielern denselben Platz zuzuweisen und bringt selbst noch einige hilfreiche Methoden, um die gesamte Rechnung sowie Darstellung zu vereinfachen.

**Logging** Die Logger-Klasse nimmt eine Sonderrolle ein, da sie nichts anderes tut als eine Nachricht abzuspeichern. Dies wird immer genutzt, wenn es darum geht, ein Zwischenergebnis festzuhalten. Daher wird sie in relativ vielen Klassen verwendet.

Listing 43: Game - startGame

```

1  @Override
2  public void startGame(PlayerType[] playerTypes, int sizeX, int sizeY) {
3      // instanciate players with given playertypes
4      this.players = createNewPlayers(playerTypes, sizeX, sizeY);
5
6      for (int i = 0; i < this.players.length; i++) {
7          this.gui.updatePlayer(this.players[i]);
8      }
9
10     // fill stack
11     this.stack = Domino.fill(this.stack);
12
13     // fill current bank
14     this.stack = this.currentRoundBank.randomlyDrawFromStack(this.stack);
15     this.gui.setToBank(CURRENT_BANK_IDX, this.currentRoundBank);
16
17     this.currBankIdx = 0;
18     this.gui.showWhosTurn(HUMAN_PLAYER_IDX);
19
20     Logger.getInstance().printAndSafe(Logger.GAME_SEPARATOR + "\nStarted new game\n");
21 }
22
23 private Player[] createNewPlayers(PlayerType[] playerTypes, int sizeX, int sizeY) {
24     Player[] output = new Player[playerTypes.length];
25     for (int i = 0; i < playerTypes.length; i++) {
26         output[i] = PlayerType.getPlayerInstanceWithGivenType(playerTypes[i], i,
27                     this.gui, sizeX, sizeY);
28     }
29     return output;
30 }
```

## 10. Programmtests

**Erklärung des Toolkits** Um effektiv Tests zum Einlesen sowie Ausgeben von Spielständen aus einer .txt Datei gestalten zu können, wurde eine Klasse geschrieben, welche dies erleichtern soll. Sie nennt sich *TestToolkit*. Bei der Erstellung habe ich mich stark an dem Toolkit aus der Übung *Algorithmen und Datenstrukturen* orientiert. Das gegebene Toolkit ist allerdings nur in der Lage, unter einer Unix-Umgebung Dateivergleiche durchzuführen, da das Programm allerdings unter Windows entwickelt wurde, mussten viele Methoden ausgetauscht werden. Die Datei aus *Algorithmen und Datenstrukturen* befindet sich ebenfalls im Verzeichnis des Anhangs, kann aber auch wieder alternativ über die Lesezeichen- bzw. die Anlageübersicht betrachtet werden.

Neben einer Festlegung, welcher Dateityp bearbeitet werden kann, liefert diese Klasse vor allem einen Pfad, in dem sämtliche Testdateien zu finden sind. Dies geschieht (ähnlich wie beim Logger) mit einem Formatstring. Um Testdateien zuallererst in Form eines Strings zu lesen, wird die Methode *readAsString* verwendet (siehe Listing 44, S. 68, *TestToolkit - readAsString*). Diese verwendet allerdings nur die bereits implementierte Methode der Loader-Klasse. Es war mir trotzdem wichtig sie hier mit aufzunehmen, um eine gemeinsame Schnittstelle für alle Tests der Dateiverarbeitung zu schaffen. In der Methode *read* wird diese Methode aufgerufen, um einen Game Konstruktor zu füllen und ein vollwertiges Spiel zurückzugeben.

Eine weitere Funktionalität des Toolkits besteht darin, zwei gegebene Dateien mit dem selben Namen auf ihre Gleichheit zu prüfen (siehe Listing 45, S. 69, *TestToolkit - assertFilesEqual*). Dies geschieht, indem beide Dateien aus den gegebenen Verzeichnissen (*results* sowie *expected\_results*) als Strings ausgelesen und anschließend per *assertEquals* verglichen werden. Die Methode *writeAndAssert* verhält sich hier sehr ähnlich, hierbei wird nur erst das gegebene Spiel in eine Datei geschrieben, bevor die *assertFilesEqual* der Klasse aufgerufen wird.

**Beschreibung der Testfälle** Um Testfälle gestalten zu können, ohne jedes Mal eine grafische Benutzeroberfläche starten zu müssen, habe ich, ähnlich wie in der Bonusaufgabe, eine Klasse namens *FakeGUI* erstellt, welche genau wie die JavaFXGUI das Interface *GUIConector* implementiert.

Um sinnvoll das Ein- sowie Auslesen von Dateien testen zu können wurde folgende

Listing 44: *TestToolkit - readAsString*

```

1 public static String readAsString(String filename) {
2     try {
3         File file = new File(String.format(PATH_FORMAT, filename));
4         return Loader.getInstance().openGivenFile(file.getPath());
5     } catch (FileNotFoundException e) {
6         return e.getMessage();
7     }
8 }
```

Listing 45: TestToolkit - assertEquals

```

1 public static void assertEquals(String filename) {
2     try {
3         File fileResult = new File("test" + File.separator + "fileTests" + File.separator
4             + "results" + File.separator + filename + ".txt");
5         File fileExpectedResult = new File("test" + File.separator + "fileTests"
6             + File.separator + "expected_results" + File.separator + filename
7             + ".txt");
8         Assert.assertEquals(Loader.openGivenFile(fileExpectedResult),
9             Loader.openGivenFile(fileResult));
10    } catch (FileNotFoundException e) {
11        assertTrue(false);
12    }
13 }
```

Ordnerstruktur gewählt: Im Ordner namens „test“ befindet sich ein Unterordner namens „fileTests“. Dies ist der Oberordner, in welchem sich alle Testdateien zum Dateieinlesen sowie Dateiausgaben befinden. Die Dateien mit dem Präfix „inv\_“ spiegeln invalide, während die mit dem Präfix „val\_“ valide Dateien wiederspiegeln. Da im Converter drei unterschiedliche Komponenten generiert werden, wurden die Testfälle hieran angelehnt. Es gibt also jeweils Testfälle für Fehler im Spielbrett, den Bänken und dem Stapel. Zu validen Spielsituationen gibt es deutlich weniger Tests, da viele der Situationen in den Unitests zu den einzelnen Datenstrukturen bereits behandelt wurden. Invalide Dateitests laufen immer nach dem selben Schema ab (siehe Listing 46, S. 71, InvalidFileRead-Tests - test\_noTagOpenerAtBeginningOfDoc). Es wird die erwartete Fehlermeldung in eine Variable gespeichert. Anschließend wird der String aus der Datei mittels *readAsString* aus der Toolkit-Klasse gelesen und dem Converter übergeben, um dieselbe Situation zu erzeugen, wie es beim Einlesen im normalen Spielverlauf der Fall ist. Im letzten Schritt wird die String-Ausgabe mit der Fehlermeldung vom Anfang verglichen.

Etwas komplexer gestalten sich die Tests bezüglich der validen Spielsituationen. Hierbei wird ein Spiel mit den erwarteten Daten erzeugt. Danach wird die Methode *read* der TestToolkit-Klasse verwendet, um ein Spiel zu erzeugen. Diese wurde bereits im vorherigen Absatz erwähnt, denn sie ruft lediglich die *readAsString*-Methode auf, um den generierten String dem Game-Konstruktor zu übergeben. Wie der Einleseprozess im Detail funktioniert, wird in der Erklärung der Converter-Klasse beschrieben (siehe Abschnitt 8.7, S. 53). Das so generierte Spiel wird mit einem Spiel verglichen, welches vorher „per Hand“ erzeugt wurde. Hierzu wird die *equals*-Methode der Game-Klasse verwendet, welche über *assertEquals* angesteuert wird. Im Anschluss wird das gelesene Spiel einmal abgespeichert und per *writeAndAssert* von der TestToolkit-Klasse überprüft, ob die Datei denselben Spielstand enthält. In der *writeAndAssert*-Methode wird zuerst eine neue Datei mit dem gegebenen Namen in dem Unterordner „results“ erstellt. Anschließend ist der Loader zum Abspeichern zuständig. Wie der Loader vorgeht, um eine Datei abzurufen, wird im Abschnitt 8.3, S. 35 näher erläutert. In Abbildung 20 - Testcoverage ist außerdem eine Auflistung der Testcoverage zu sehen.

**Auflistung wichtiger Testfälle** In der folgenden detaillierten Auflistung (ValidFile-ReadTests) werde ich mich auf die Tests, welche sich mit dem Dateieinlesen beschäftigen, beschränken. Beschreibungen der anderen Testklassen folgen anschließend.

| Name              | Testfall   | erwartetes Ergebnis                               | erzieltes Ergebnis                                |
|-------------------|--|---|---|
| newGame           | Das Spiel wurde während des initialen Selektierens abgespeichert. Es wurde lediglich die <i>currentRoundBank</i> geladen. Hierbei hat noch kein Spieler eine Auswahl getroffen. Dies ist der Zustand welcher nach dem Aufruf der startGame-Methode vorzufinden ist. Es wird ein Spiel generiert und mit dieser Methode begonnen. Anschließend wird das Spiel mit seinem Namen (wie der Test) aus dem Expected-Ordner geladen und mit dem Spiel verglichen. | Das eingelesene und generierte Spiel sind gleich. | Das eingelesene und generierte Spiel sind gleich. |
| noNext-BankYet    |  |   |   |
| fullCurrBank      | Spiel wurde während einer standardmäßigen Runde abgespeichert. Es wird erwartet, eine von allen Spielern selektierte <i>currentRoundBank</i> und eine nicht selektierte <i>nextRoundBank</i> vorzufinden. Außerdem wurden die Spielfelder der Spieler bereits verändert und müssen eingelesen werden. Im Testaufruf wird die <i>equals</i> -Methode der Game-Klasse aufgerufen, da diese alle beteiligten Teilkomponenten prüft.                           | Das eingelesene und generierte Spiel sind gleich. | Das eingelesene und generierte Spiel sind gleich. |
| fullStack         |  |   |   |
| noStack           | Es wird ein Spiel generiert, welches einen leeren Beutel besitzt. Sämtliche Felder der Spieler müssen geladen werden und es werden die letzten Züge der noch fehlenden Spieler absolviert. Wichtig hierbei, der menschliche Spieler muss seinen letzten Zug ausführen, der Domino befindet sich allerdings noch auf der Bank.  | Das eingelesene und generierte Spiel sind gleich. | Das eingelesene und generierte Spiel sind gleich. |
| lastTurnIn-RotBox | Es wird eine ähnliche Situation wie im vorherigen Test generiert. Allerdings wurde hierbei der Domino der Bank bereits in die Rotationsbox des menschlichen Spielers geladen. Wie bei allen diesen Testfällen wird auch hier wieder das Spiel mit der gleichnamigen Datei verglichen.  | Das eingelesene und generierte Spiel sind gleich. | Das eingelesene und generierte Spiel sind gleich. |

| 93% classes, 72% lines covered in package 'logic' |            |             |               |
|---|------------|-------------|---------------|
| Element   | Class, %   | Method, %   | Line, %       |
| bankSelection                                     | 100% (3/3) | 91% (41/45) | 88% (144/163) |
| dataPreservation                                  | 100% (2/2) | 65% (13/20) | 55% (54/98)   |
| differentPlayerTypes                              | 80% (4/5)  | 46% (25/54) | 40% (115/283) |
| logicTransfer                                     | 87% (7/8)  | 67% (42/62) | 63% (260/412) |
| playerState                                       | 100% (7/7) | 88% (77/87) | 88% (382/432) |
| randomizer  | 100% (2/2) | 100% (5/5)  | 100% (14/14)  |
| token   | 100% (5/5) | 97% (48/49) | 97% (179/183) |

Abbildung 20: Testcoverage

Listing 46: InvalidFileReadTests - test\_noTagOpenerAtBeginningOfDoc

```

1 @Test
2 public void noTagOpenerAtBeginningOfDoc() {
3     String expOutput = WrongTagException.DEFAULT_MESSAGE;
4     String fileOutput = TestToolkit.readAsString("inv_noTagOpenerAtBeginningOfDoc");
5     String actOutput = new Converter().readStr(new FakeGUI(), fileOutput);
6     assertEquals(expOutput, actOutput);
7 }
```

**Überblick über weitere Testklassen** Um noch einen Überblick über sämtliche andere Testklassen zu geben folgt eine kurze Zusammenfassung der jeweiligen Klassen.

| Name     | Testfall   | erwartetes Ergebnis                                      | erzieltes Ergebnis                                       |
|----------|--|--|--|
| BankTest | Konstruktor: Stringeingaben sowie Entry-Arrays werden als Eingabe getestet.<br>Selektieren: Es werden Plätze auf der Bank selektiert. Anschließend werden die Spielerreferenzen verglichen.<br>Getter: Bänke werden erstellt und gewünschte Objekte per Getter geholt und verglichen.<br>Zufälliges Ziehen aus dem Stapel: Eine Bank wird gefüllt, allerdings wird der Bank ein PseudoRandom Objekt übergeben. Mit diesem ist es möglich, stets dieselbe Auswahl zu treffen. | Die generierten Bänke enthalten die erwarteten Einträge. | Die generierten Bänke enthalten die erwarteten Einträge. |

|   |   |  |  |
|---|---|--|--|
| EntryTest                                 | Es wird die String-Präsentation sowie die Equals Methode eines Eintrags getestet. Mit entsprechendem Konstruktor werden die Tests geladen und per <code>toString</code> / <code>equals</code> geprüft.  | Die Einträge entsprechen der erwarteten String-Präsentation bzw. dem generierten Referenzobjekt. | Die Einträge entsprechen der erwarteten String-Präsentation bzw. dem generierten Referenzobjekt. |
| Game-Loading-ConstructorTest und GameTest | Hierbei wird genauso vorgegangen wie bei den <code>ValidFileReadTests</code> , es wird allerdings nicht gegen eine Datei getestet, sondern sämtliche Referenzobjekte werden vor Ort erstellt und auf Gleichheit mit dem erstellten Spiel getestet.  | Sämtliche Teilkomponenten entsprechen den Komponenten im Spiel.                                  | Sämtliche Teilkomponenten entsprechen den Komponenten im Spiel.                                  |
| Board-Test                                | Es werden diverse Spielsituationen per String-Eingabe im Konstruktor erzeugt. Da dies in erster Linie als Datenobjekt fungiert, werden über entsprechende Getter Werte verglichen.  | Die angeforderten Werte entsprechen den erwarteten.  | Die angeforderten Werte entsprechen den erwarteten.  |
| Default-AIPlayer-Test                     | Es wird ein künstlicher Spieler erzeugt und mit entsprechenden Spielsituationen konfrontiert. Dabei selektiert er jedes Mal von der Bank, da er hiermit nicht nur einen Domino wählt, sondern diesen auch gleichzeitig positioniert. Diverse Teilaufgaben werden mit jedem Test behandelt, obwohl die Tests alle relativ ähnlich aussehen. Es sei noch erwähnt, dass bei einem Konstruktorauftrag mit einer String-Eingabe gleichzeitig die Distrikte aufgebaut werden. Daher ist es auch hier möglich die Distrikte zu prüfen. | Auf Board des Spielers wird der erwartete Domino gesetzt.  | Auf Board des Spielers wird der erwartete Domino gesetzt.  |

|               |  |  |  |
|---------------|--|--|--|
| District-Test | Hierbei wird ein einzelner Distrikt getestet, wichtig hierbei sind insbesondere das Hinzufügen neuer Distriktelemente sowie das Zusammenführen mehrerer Distrikte. Es werden Listen mit entsprechenden Referenzobjekten gebildet und verglichen. | Die Liste der Referenzobjekte stimmt mit den generierten Werten überein. | Die Liste der Referenzobjekte stimmt mit den generierten Werten überein. |
| Player-Test   | In dieser Klasse werden spielerunspezifische Methoden getestet, ein Beispiel wäre das Werfen eines Dominos. Agiert ansonsten sehr ähnlich zum DefaultAIPlayerTest.   | Auf Board des Spielers wird der erwartete Domino gesetzt.                | Auf Board des Spielers wird der erwartete Domino gesetzt.                |
| Domino-Test   | Diese Testklasse wurde bereits in der Bonusaufgabe gegeben und wurde meinerseits mit leichten Modifikationen übernommen.   | Die generierten Dominos entsprechen den Erwarteten.                      | Die generierten Dominos entsprechen den Erwarteten.                      |

## Literatur

- [1] City Domino Aufgabenstellung. <http://intern.fh-wedel.de/mitarbeiter/klk/programmierpraktikum-java/aufgabetermine/ss18-citydomino/>. Aufgerufen am: 29-12-2018.
- [2] try-with-resources-Block Erklaerung. <https://www.baeldung.com/java-exceptions>. Unterpunkt 4.4, Aufgerufen am: 03-01-2019.
- [3] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Entwurfsmuster von Kopf bis Fuß* -. O'Reilly Germany, Köln, 1. aufl. edition, 2006.

## A. Anhang

### A.1. Entwicklung der Gui

Ein kurzer Überblick über die verschiedenen Stadien der Gui-Entwicklung.

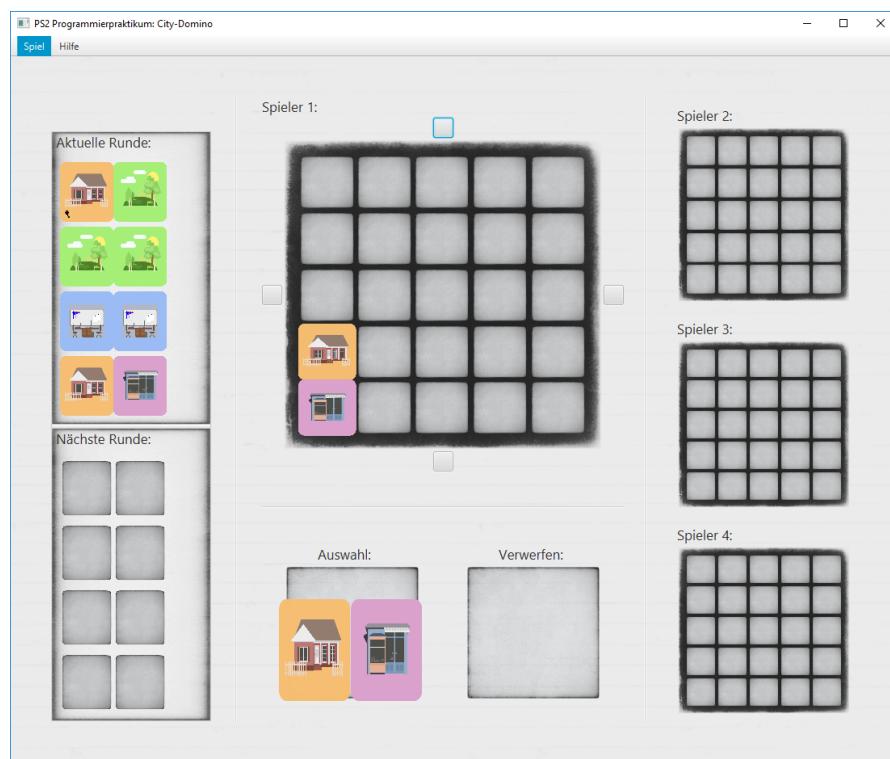


Abbildung 21: Gui-Entwicklung 1

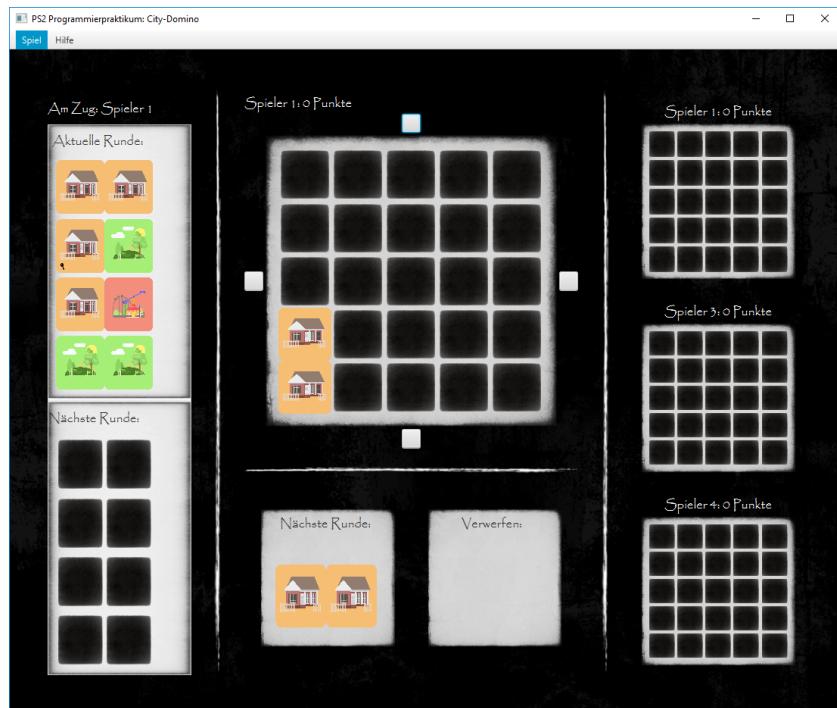


Abbildung 22: Gui-Entwicklung 2

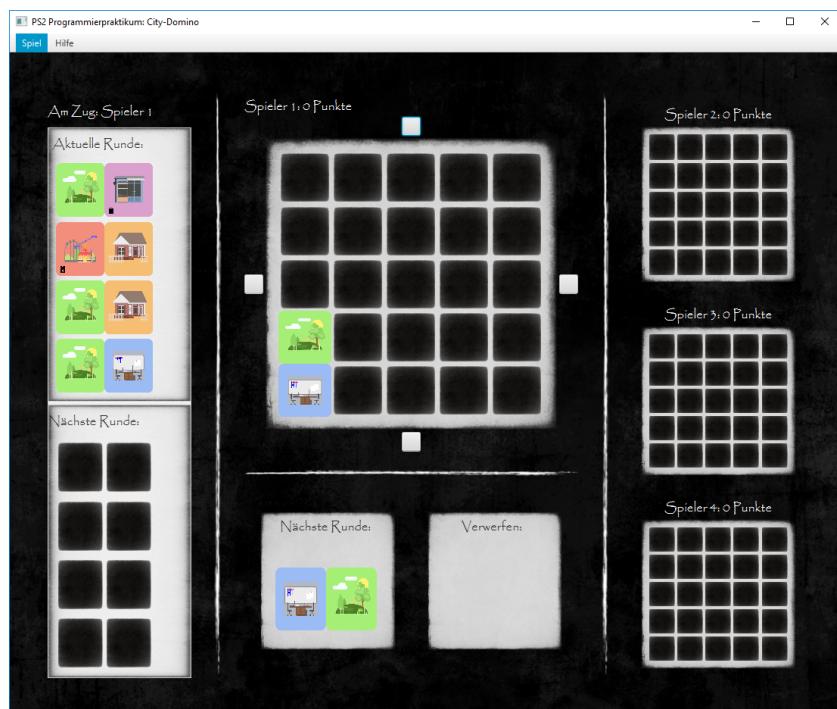


Abbildung 23: Gui-Entwicklung 3

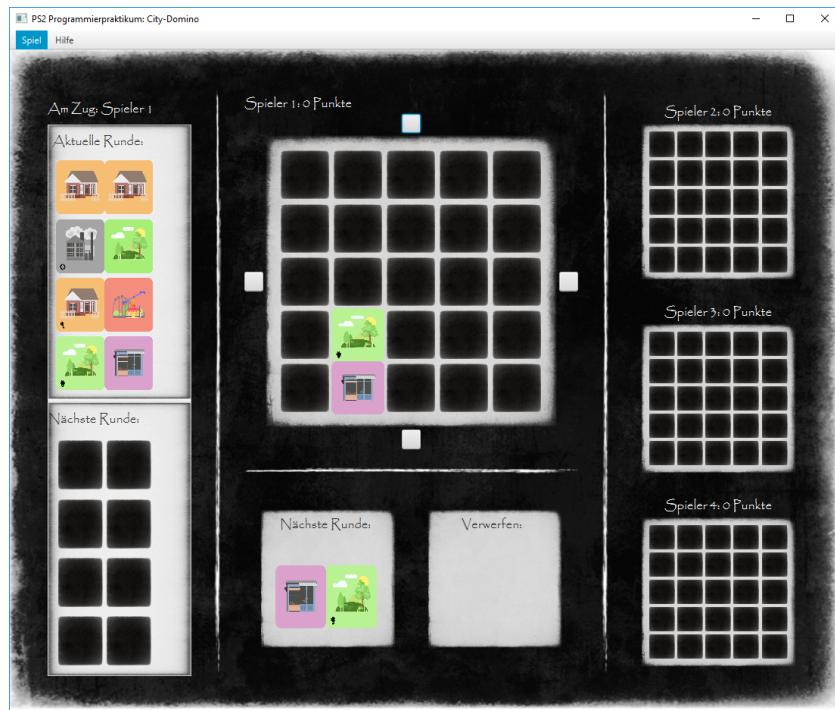


Abbildung 24: Gui-Entwicklung 4

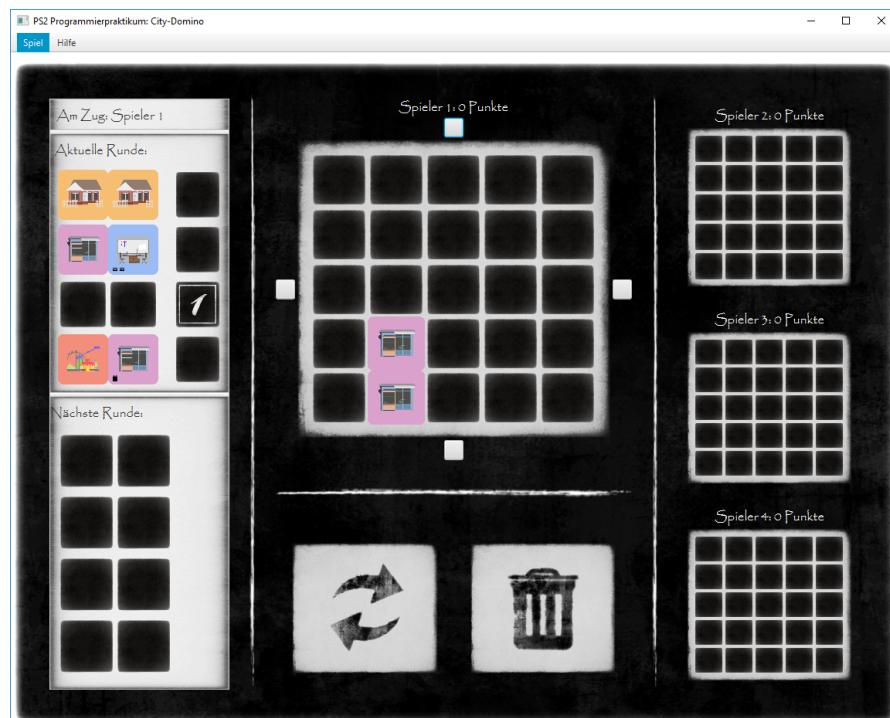


Abbildung 25: Gui-Entwicklung 5

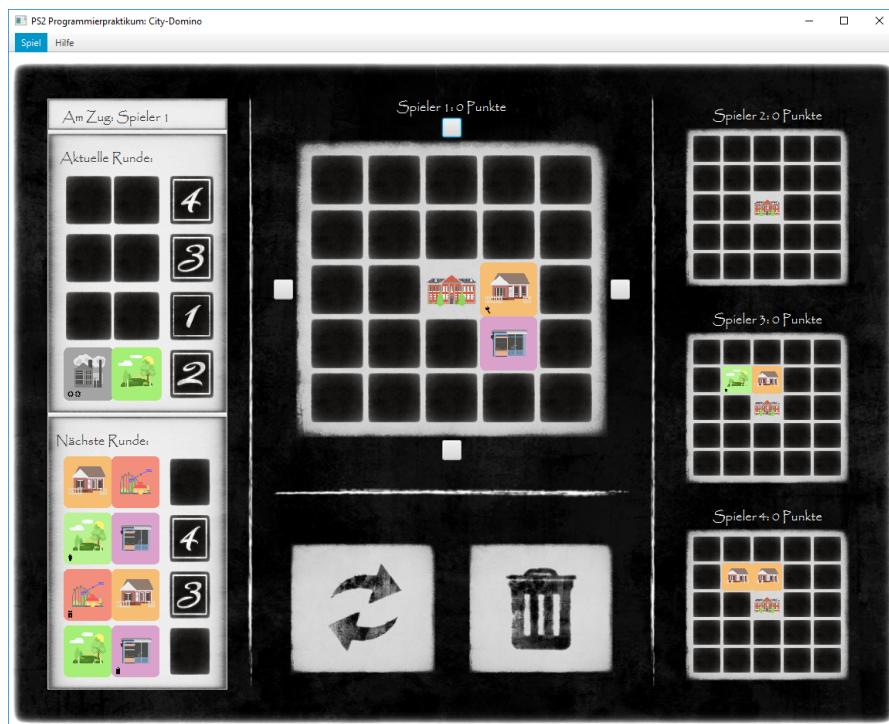


Abbildung 26: Gui-Entwicklung 6

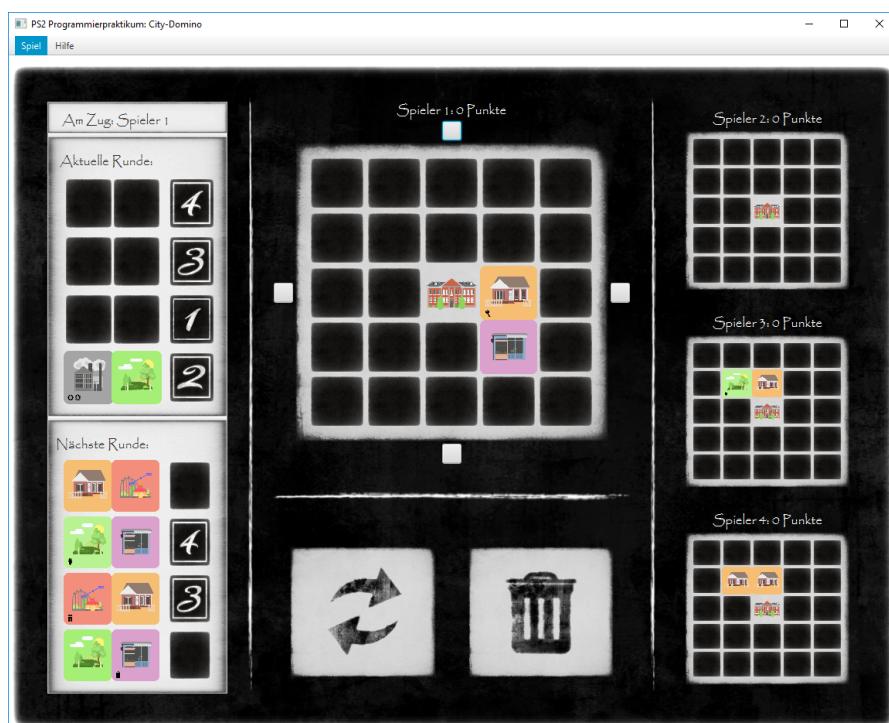


Abbildung 27: Gui-Entwicklung 7

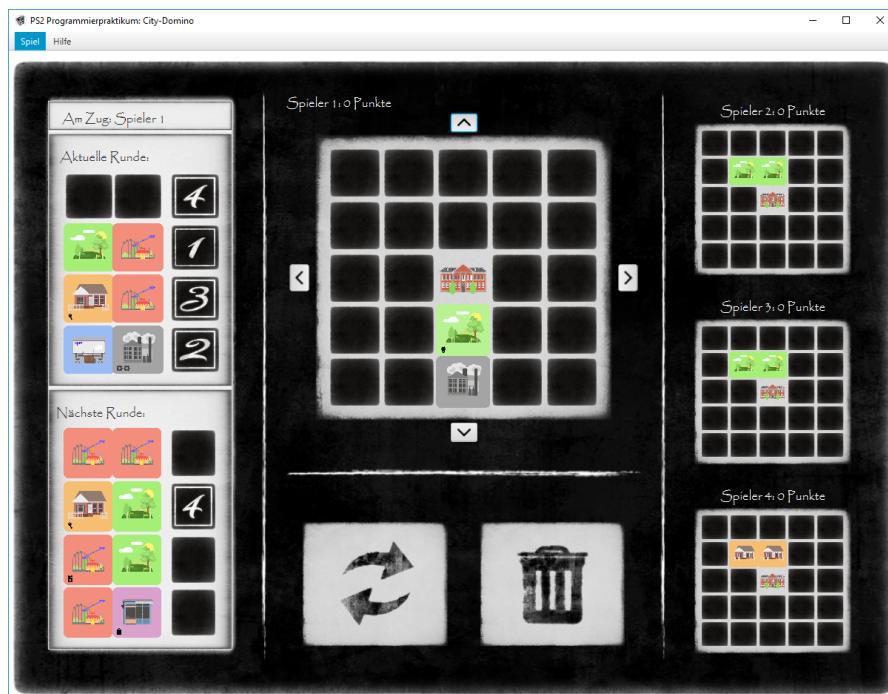


Abbildung 28: Gui-Entwicklung 8

## A.2. Bildquellen

- Trash icon base texture: [https://www.flaticon.com/free-icon/dustbin\\_1570](https://www.flaticon.com/free-icon/dustbin_1570)
- Rotation icon base texture:  
<https://www.iconsdb.com/white-icons/available-updates-icon.html>
- Sonstige Texturen: <https://www.poliigon.com/>