



Silas Hoffmann, inf103088
3. Fachsemester
3. Verwaltungssemester

5. Januar 2019

Dokumentation

Programmierpraktikum

im Wintersemester 2018/19

Dozent: Prof. Dr. Andreas Häuslein

Fachbereich Informatik

Fachhochschule Wedel

Inhaltsverzeichnis

I. Allgemeine Problemstellung	4
1. Erklärung des Spiels	4
1.1. Spielregeln	4
1.2. Anlegeregeln	4
1.3. Spielende	5
2. Implementierungsdetails	5
2.1. KI	5
2.2. Oberfläche	6
2.3. Karten	6
2.4. Log	7
2.5. Spielstand	7
II. Benutzerhandbuch	9
3. Ablaufbedingungen	9
4. Programminstallation / Programmstart	9
5. Bedienungsanleitung	9
5.1. Hintergrundinformationen	9
5.2. Programmfunctionalitaet	14
III. Programmierhandbuch	19
6. Entwicklungskonfiguration	19
7. Problemanalyse und Realisation	20
7.1. Problemanalyse	20
7.2. Realisationsanalyse	20
8. Beschreibung grundlegender Klassen	24
8.1. token	25
8.2. dataPreservation	29
8.3. playerState	33
8.4. bankSelection	40
8.5. differentPlayerTypes	44
8.6. logicTransfer	49
9. Programmorganisationsplan	54

A. Im Anhang Eins

ii

Literaturverzeichnis
noch anführen

Abbildungsverzeichnis

1.	Erstes Selektieren	10
2.	Schwebender Domino ueber gueltiger Position	11
3.	Schwebender Domino ueber ungultiger Position	11
4.	Menueoptionen	12
5.	Nachtraegliches Abspeichern	12
6.	Filechooser	13
7.	Unterschiedliche Fehlermeldungen beim Einlesen einer Datei	14
8.	Spielbeginn	15
9.	Initiales Selektieren - oben	15
10.	Initiales Selektieren - mittig	16
11.	Erstes Rotieren	17
12.	Erste Ablage	17
13.	UML-Darstellung des token packages	25
14.	UML-Darstellung des dataPreservation packages	29
15.	Vereinfachte UML-Darstellung des gesamten Projekts	54
16.	Auswahlfenster	56

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,- P1H0
```

- 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
```

herausbekomme
wie man
Spielfeld-
syntax auf
eine Seite
bekommt

Teil II.

Benutzerhandbuch

3. Ablaufbedingungen

Ueberblick ueber die benoetigten Hardware und Softwarekomponenten die fuer die Ausfuehrung des kompilierten Programms benoetigt werden.

Softwarekomponenten	
Name	Version
Java Runtime Environment	1.8

Fussnote mit Link zur Webseite einbinden

4. Programminstallation / Programmstart

5. Bedienungsanleitung

5.1. Hintergrundinformationen

Spielinteraktion

Genauen Namen angeben

Font angeben

Muss noch gemacht gemacht werden

Selektierungsvorgang Um einen gewuenschten Domino zu selektieren muss der Spieler auf einen der schwarzen Kaesten rechts neben dem angezeigten Domino per Mausklick auswaehlen (siehe Abbildung 10) und es erscheint die Zahl 1 in diesem Feld. Um dem Spieler deutlich zu machen von welcher Bank, oder ob er ueberhaupt in seinem aktuellen Zug einen Domino selektieren darf, kann er nur auf der Bank welche nicht verschwommen ist einen Domino auswaehlen. Um jederzeit ablesen zu koennen welcher Spieler gerade am Zug ist gibt es hierfuer ein Feld oberhalb der Bank fuer die naechste Runde.

Justierung des Dominos Nachdem der Spieler erfolgreich saemtliche Selektierungsschritte auf den beiden Baenken absolviert hat, kann er seinen zuvor ausgewahlten Domino in dem dafuer vorgesehenen Kasten drehen. Um den Domino um 90 Grad zu drehen muss der Spieler lediglich einen Mausklick auf dem Domino ausfuehren.

Positionierung auf dem Spielfeld Um den justierten Domino nun auf dem Feld zu platzieren zieht der Benutzer den Domino an die gewuenschte Stelle auf dem Spielfeld. Waehrend dem Ziehen faerben sich zugrunde liegenden Felder jeweils gruen, falls es moeglich sein sollte den Domino an dieser Stelle anzulegen (siehe Abbildung 11), beziehungsweise rot, falls dies nicht der Fall sein sollte (siehe Abbildung 3, fuer genauere Informationen siehe Abschnitt 5.2). Falls der Domino an der gewuenschten Stelle nicht

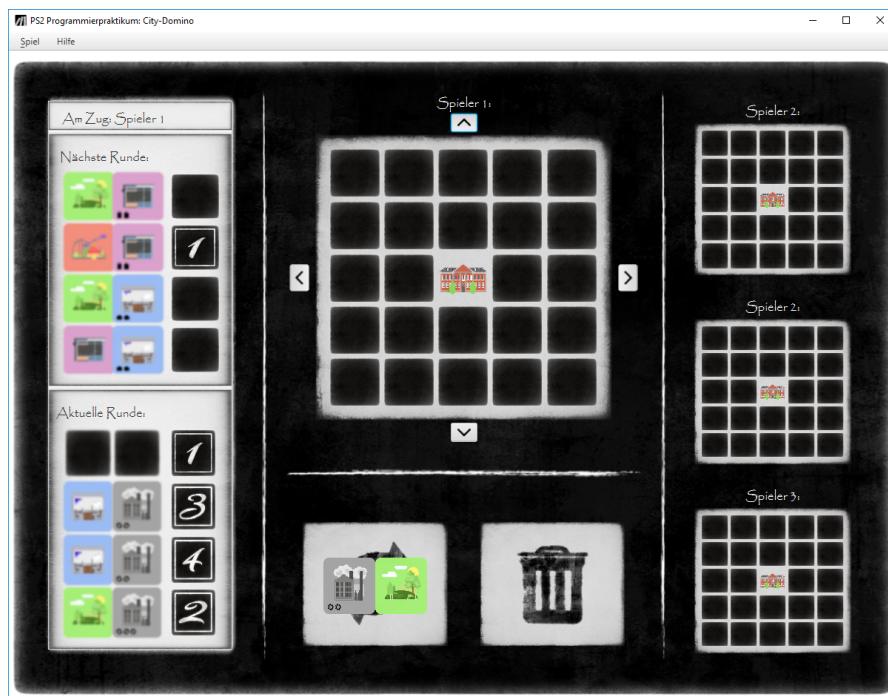


Abbildung 1: Erstes Selektieren auf der Bank fuer die naechste Runde

passen sollte und dennoch versucht wird ihn an dieser Stelle zu platzieren, passiert nichts denn der Domino befindet sich weiterhin in dem Kasten zum justieren der Ausrichtung und es kann ein neuer Versuch gestartet werden.

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

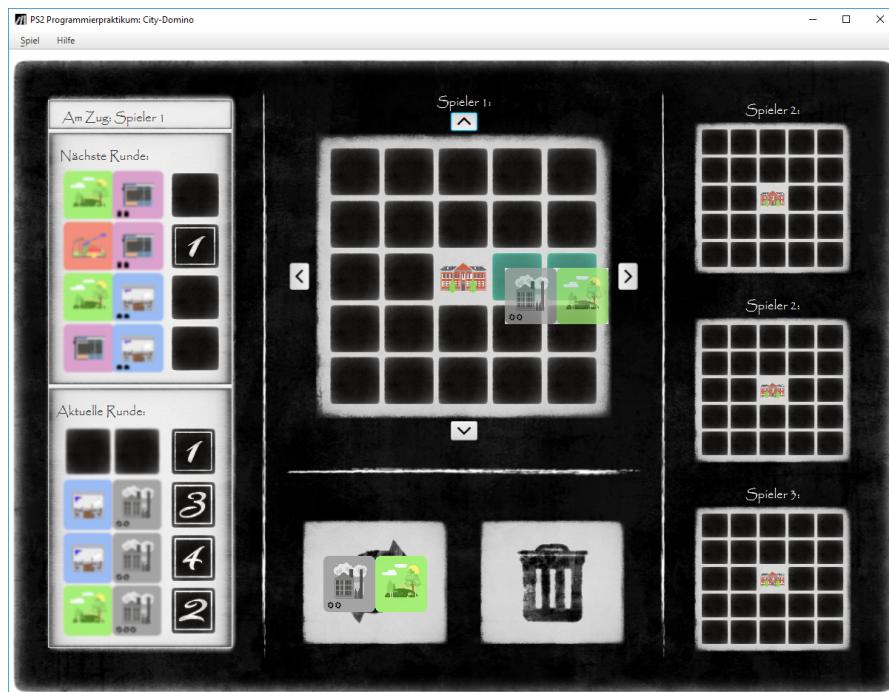


Abbildung 2: Schwebender Domino ueber gueltiger Position

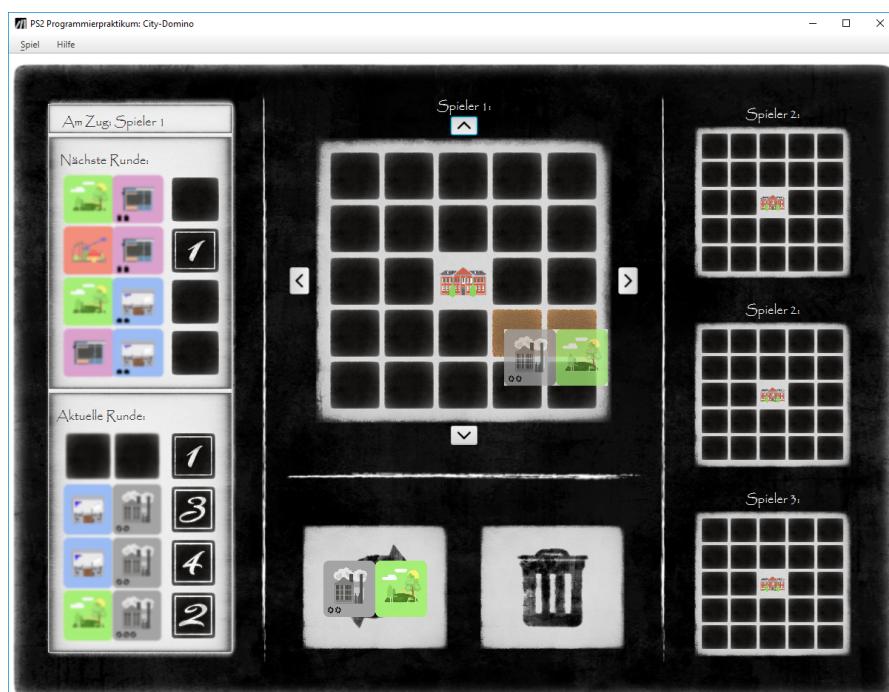


Abbildung 3: Schwebender Domino ueber ungueltiger Position



Abbildung 4: Menueoptionen

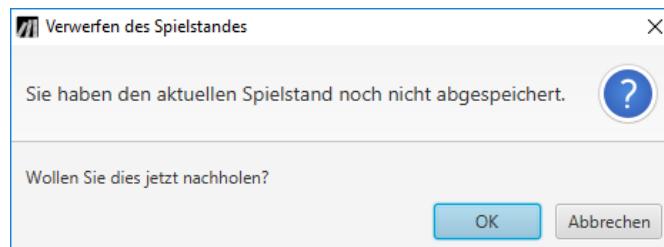


Abbildung 5: Nachtraegliches Abspeichern

Menueinteraktion

Starten bzw. Schliessen Um ein neues Spiel zu starten wählt der Benutzer den Menuepunkt "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 folgenden Abschnitt). Möchte er fortfahren ohne den aktuellen Spielstand muss der Button mit der Aufschrift **Abbrechen** betätigt werden (siehe Abbildung 5).

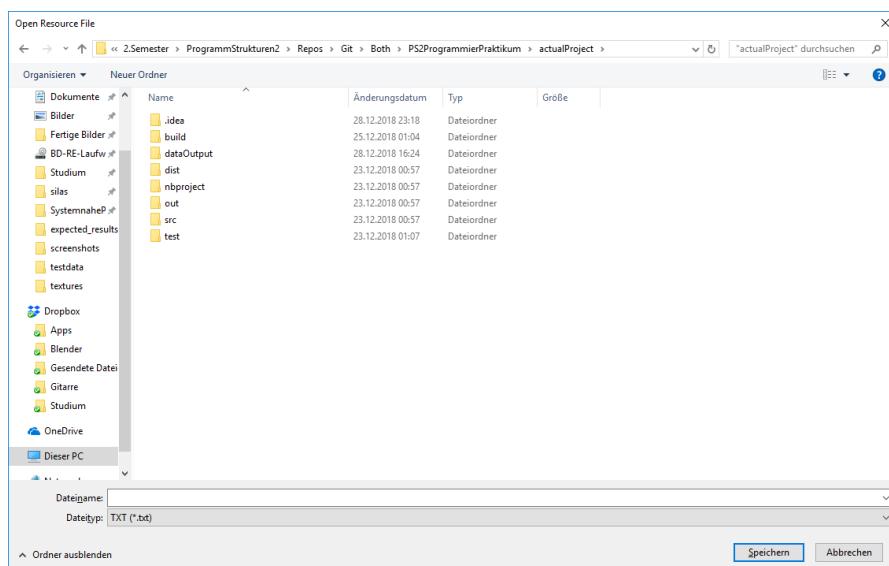


Abbildung 6: Filechooser zum Abspeichern eines Spielstandes

Spielstand abspeichern Um ein Spielstand zu speichern gibt es zwei Reiter mit folgenden Moeglichkeiten.

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

Speichern unter gibt dem Benutzer die Moeglichkeit, ausgehend vom Ablageverzeichnis den gewuenschten Speicherort anzugeben. Hierzu navigiert man mit dem gegebenen Filechooser an den gewuenschten Speicherort und gibt der abzuspeichernden Datei einen Namen, die benoetigt Dateiendung *.txt* ist bereits ausgewaehlt, sodass der Benutzer seine Auswahl lediglich auf dem Feld *Speichern* per Mausklick zu bestaetigen braucht (siehe Abbildung 6).

Der Reiter *Speichern* ermoeglicht es einen bereits gespeicherten Spielstand ohne oeffnen eines Filechoosers zu ueberschreiben. Falls der Benutzer diesen Reiter betaetigt ohne dass zuvor ein Spielstand des aktuellen Spiels abgespeichert wurde ist, oeffnet sich bei der Auswahl dieses Reiters dennoch ein Filechooser und es wird nach der Funktionsweise von *Speichern unter* vorgegangen.

Oeffnen Aehnlich wie beim Reiter *Speichern unter* wird hier ein Filechooser geoeffnet. Dieser wird jedoch dazu verwendet eine Datei auszuwaehlen um aus dieser einen Spielstand zu lesen. Falls die Datei nicht der geforderten Syntax entspricht , erscheint eine Fehlermeldung in Form eines Popup-Fensters mit dem einer groben Fehlermeldung wo der Fehler liegt(Siehe ...) . Falls der Benutzer vor dem Oeffnen eines neuen Spielstands den alten nicht gespeichert hat wird er aehnlich wie beim Beenden darauf hingewiesen und es wird per Filechooser eine Moeglichkeit bereitgestellt dies nachzuholen.

Screenshots
der
Fehlermel-
dungen
und
Referenz
auf Er-
klaerung
einfuegen

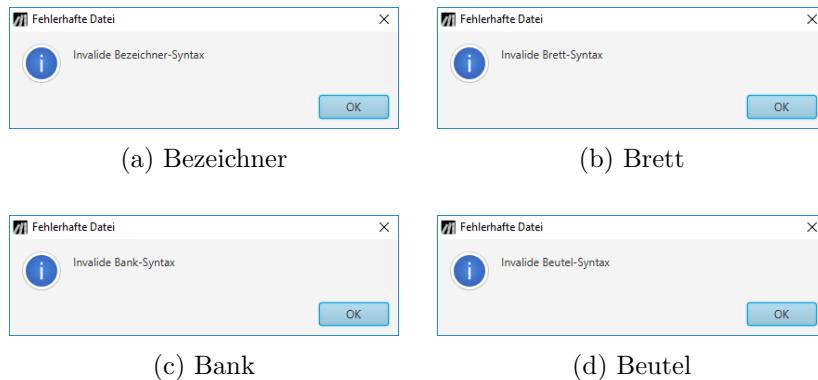


Abbildung 7: Unterschiedliche Fehlermeldungen beim Einlesen einer Datei

Hilfestellung Unter dem Reiter *Hilfe* ist die Aufgabenstellung im Pdf-Format zu finden. Beim Auswaehlen des Menuepunktes *Aufgabenstellung* oeffnet sich diese im, vom Benutzer standardmaessig genutzten, Pdf-Reader.

5.2. Programmfunctionalitaet

Generell gilt es zwischen einem standardmaessig ausgefuehrten Spiel und einem 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 uebernommen, sodass der erste Spielzug des Benutzers das initiales Selektieren vom ersten Auswahlbereich (hier mit *Aktuelle Runde* gekennzeichnet) darstellt. Um einen Domino selektieren zu koennen werden vier Karten gezogen und im ersten Auswahlbereich angezeigt. Dabei wird die niedwertigste 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

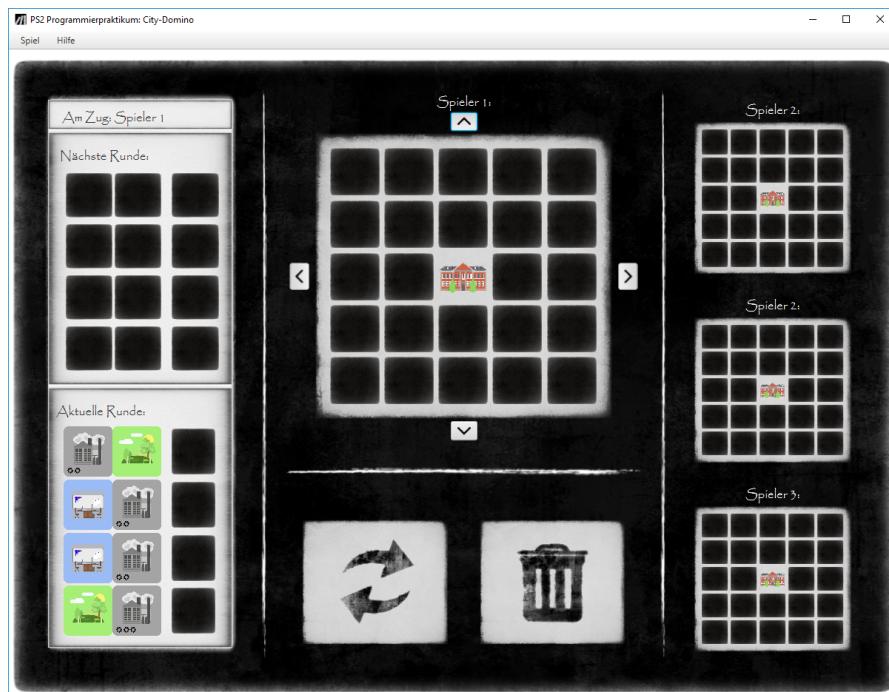


Abbildung 8: Spielbeginn nach Programmstart

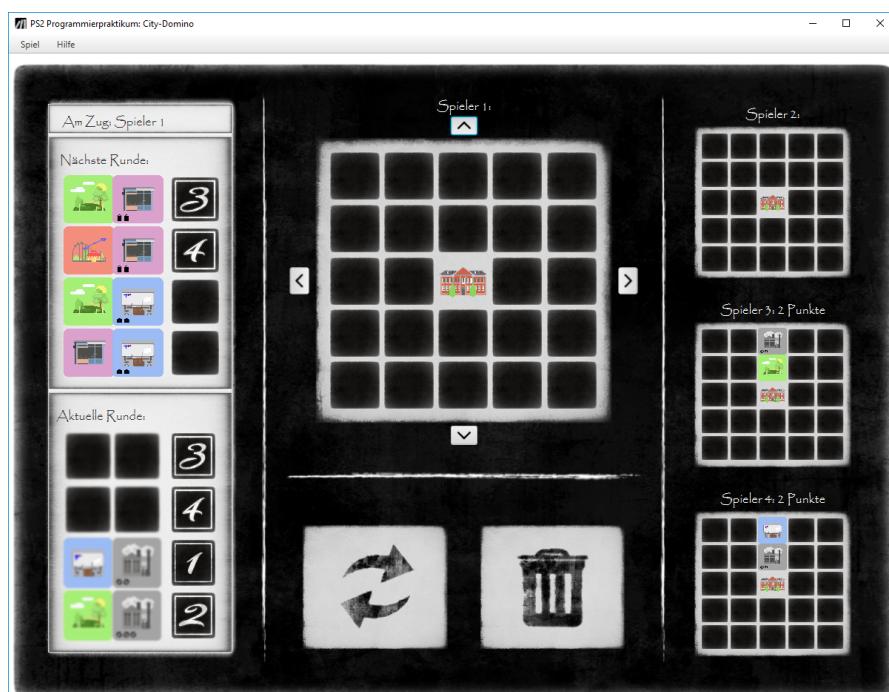


Abbildung 9: Initiales Selektieren (oben) nach Programmstart

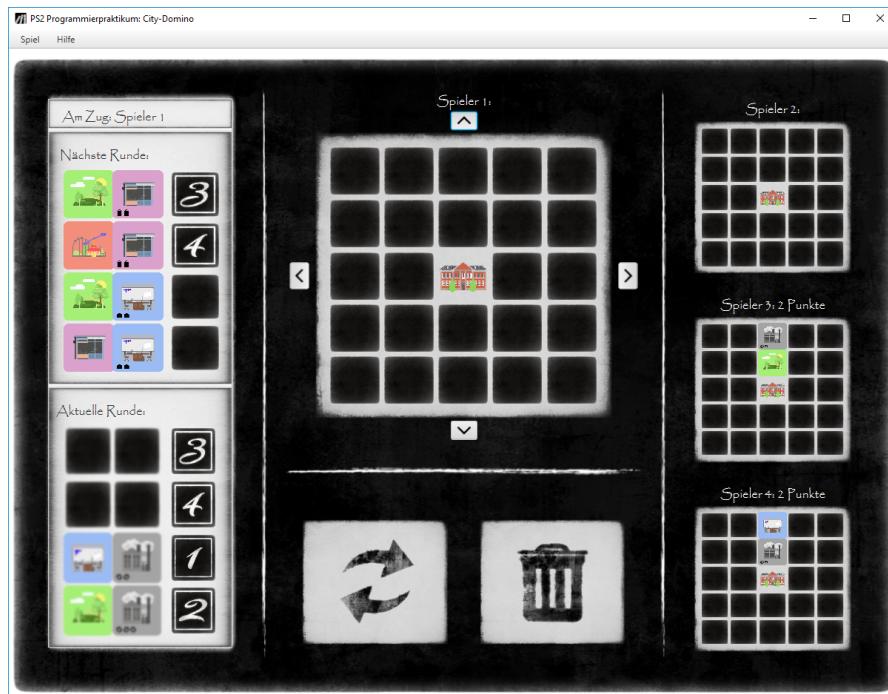


Abbildung 10: Initiales Selektieren (mittig) nach Programmstart

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 wo der Benutzer noch gar nicht an der Reihe war. Nun kann der Benutzer einen Domino auf dem naechsten Auswahlstapel bzw. der naechsten Bank einen der uebrig gebliebenen Dominosteine auswaehlen. Nun wird der Domino welcher als erstes Selektiert wurde in den Kasten zum rotieren geladen (siehe Abbildung 11) und der Benutzer kann den Stein auf sein Brett legen (Abbildung 12). Nach dieser Aktion kann der Benutzer einen Domino auf der Bank fuer die naechste Runde auswaehlen. Danach muss er wieder "warten" bis er an der Reihe ist um einen ausgewaehlten 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 waehrend des initialen Selektierens oder waehrend einer standardmaessigen Runde abgespeichert wurde, muss der Benutzer entweder von der Bank fuer die aktuelle Runde oder von der Bank der naechsten Runde Selektieren. Generell gelten hier die gleichen Regeln zum Spielablauf wie beim Spielen ohne gespeicherten Spielstand, es kann nur der Schritt des initialen Selektierens uebersprungen werden.

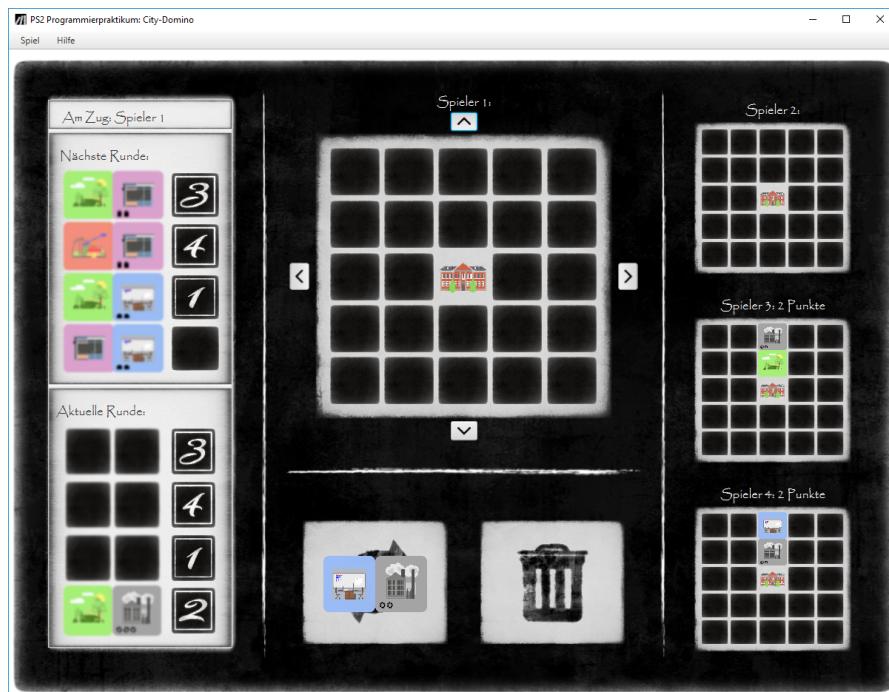


Abbildung 11: Erstes Rotieren

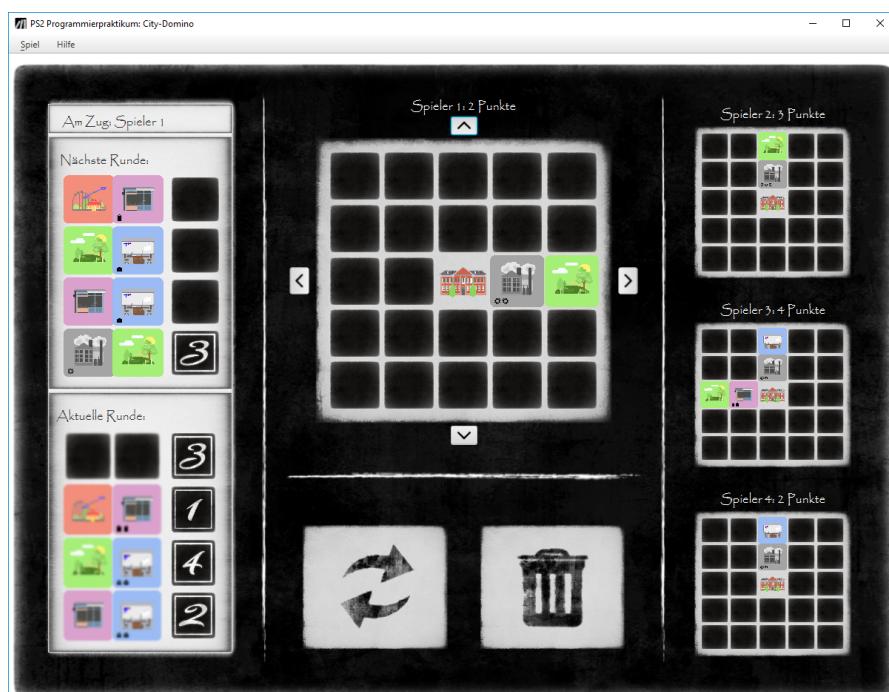


Abbildung 12: Erste Ablage auf dem Spielfeld

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]

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

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 verlangt, oder Konstanten statt Zahlen ohne Kontext, verlangt. Hierzu wurden die Vorgaben der Übung *Algorithmen und Datenstrukturen* verwendet. Die entsprechende XML-Datei findet sich hier xxxx.

Referenz
auf die
Checksty-
le Datei
einbinden

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 fuer welchen Teilbereich diese Komponente verantwortlich ist. Anschliessend ist es etwas einfacher sowie uebersichtlicher sich auf die Struktur festzulegen, da man sämtliche Nomen als Klassen ansehen kann (hier einmal orange dargestellt). Verben spiegeln die benoetigten Methoden wieder (hier gruen 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 .

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 genauer beschrieben werden. Letztendlich bieten diese allerdings sämtliche benötigte

Kapitel mit Beschreibung der Dominos einbinden

Schnittstellen um einen Domino mit einer bestimmten Aufschrift an mit einer 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 ein 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 eine genaue Zuordnung, da es ja möglicherweise Spieler mit identischem Index geben könnte.

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 abzuwägen ob man eventuell die "Antwort" des Programms gleich in diese Schnittstelle mit aufnimmt. Dies führt allerdings zu unübersichtlichem Code.

Anführungszeichen

Spieler 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. Hier gilt es abzuwägen 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, über die Benutzeroberfläche, arbeitet, während die Bots diese mehr oder weniger ignorieren und isoliert ihre 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) überlassen, dann wäre ein Spieler aber nicht mehr so unabhängig vom Spiel wie in diesem Fall. Es ist so möglich eine minimale Schnittstelle dem Spiel gegenüber bereitzustellen, indem der Spieler selbst alle wichtigen Schritte ausführt um einen Zug zu machen.

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, aber 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.

Mögliche
Erweiterung
referenziern...
das intro
fenster

Spiel Dieser Begriff beschreibt koordinierte Abarbeitung der Spielerzüge. Es wird der Benutzeroberfläche mitgeteilt was angezeigt werden soll. Außerdem werden sämtliche sämtliche Stapel von Dominos bereitgestellt die für ein vollwertiges Spiel genutzt werden sollen (Beutel, Baenke). Alle benötigten Dateioperationen werden hier eingeleitet aber nicht direkt in dieser Klasse bearbeitet. Durch das ganze Exceptionhandling wird

es ziemlich unuebersichtlich alles in das Spiel zu packen, da dieses in erster Linie fuer die uebergeordnete 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 drauf zugegriffen wird. Das Muster wird eignet sich besonders gut fuer das Loggen von Daten, da man alles in die selbe Datei schreiben möchte und diese nicht jedesmal neu *suchen* muss. Im Logger kann man zum Beispiel einfach ein entsprechendes Feld anlegen.

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*). Und auch das Speichern und Laden tritt immer wieder vermehrt und verteilt ueber den gesamten Code auf. Alternativ könnte man auch ein klassische Klasse verwenden, wegen den genannten Punkten empfiehlt sich aber gerade für diese beiden Anwendungsfälle dieses Muster.

Benutzerhandbuch einbinden

Verweis auf genauere Beschreibung des Musters einbinden

Große Klassenübersicht Mit dieser Uebersicht bin ich zu folgender groben Klassenstruktur gelagt:

- Interfaces:
 - GUI2Game
 - GUIConnector
- Klassen
 - Game
 - Player
 - Verschiedene Ausprägungen der abstrakten Spieler-Klasse
 - District
 - Bank
 - Entry
 - Board
 - Domino
 - Pos
 - Logger
- Aufzählungstypen

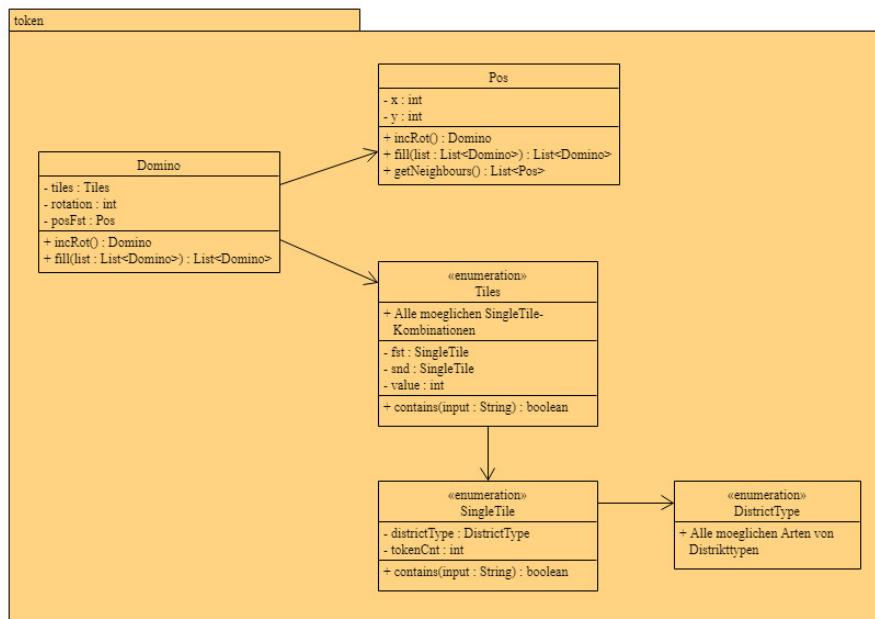
- Tiles
- SingleTile
- DistrictType

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 ...

Kapitelref
einfuegen

- 8.1 token
 - Pos
 - DistrictType
 - Tiles
 - SingleTile
 - Domino
- 8.2 dataPreservation
 - Loader
 - Logger

Abbildung 13: UML-Darstellung des `token` packages

8.1. token

Das package `token` (siehe Abbildung 13) vereint sämtliche Klassen die benötigt werden um einen Domino (siehe `Domino`) erstellen zu können.

Pos Diese Klasse wurde von der Bonusaufgabe der PS2-Übung übernommen. Es wurde lediglich `toString()`-Methode sowie `getNeighbours()` abgeändert und einige Konstanten hinzugefügt um den Checkstyle-Richtlinien (siehe Abschnitt 6 - Entwicklungs-konfiguration) 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 ein die Differenz der beiden jeweils gleichnamigen Komponenten gebildet und am Ende verglichen ob nur jeweils eine der beiden Differenzen gleich Null ist (siehe Listing 1).

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 2).

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

Listing 1: 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 2: 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 }
```

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 3). Dieser Aufzählungstyp spielt vor allem beim Auszählen der Punkte beziehungsweise 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 einem Anzahl an Punkten welche auf dem betrachteten Tile verfügbar sind (siehe Listing 4).

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

Tiles Dieser Aufzählungstyp beschreibt alle möglichen *SingleTile*-Kombinationen die im Stapel einer normalen Partie des Spiels möglich sind (siehe listing 5). Auch hier gibt es wieder einen Konstruktor der diverse Werte an die jeweiligen Enum-Werte bindet. Es

Listing 3: DistrictType

```

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

Listing 4: 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 5: 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 6: 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 }
```

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 willkürliche festgelegt).

Um einfacher Testen zu können wurden mehrere Methoden eingeführt um von außerhalb einfacher bestimmte Tile-Kombinationen zu bekommen. 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 eine 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 6, 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 *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 7, Domino - compareTo).

Listing 7: 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 }
```

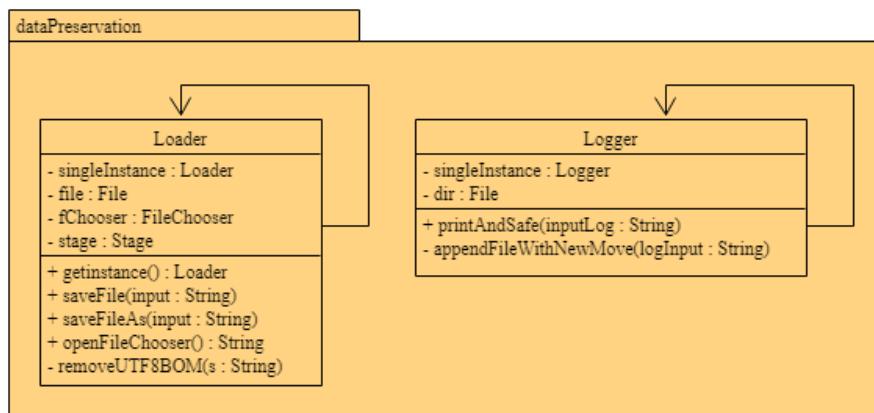


Abbildung 14: UML-Darstellung des dataPreservation packages

8.2. 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 8, Singleton-Muster). Dieses Muster eignet sich hervorragend für Objekte wo 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. Insbe-

Listing 8: 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 }
```

sondere der Fall des Speichern 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 8, Singleton-Muster).

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 10, Logger - printAndSafe). Die Methode verwendet neben dem standard Ausgabestrom eine weitere Hifsmethode um die Nachricht an die Log-Datei anzuhängen (siehe Listing 11, 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

Listing 9: Logger - printAndSafe

```

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

Listing 10: Logger - printAndSafe

```

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

diesen Catch-Block zu laufen wird das Flag *loggingPossible* auf *false* gesetzt.

Loader

Einleitung Diese Klasse ist zum laden und speichern von Spielständen gedacht. Im Konstruktor wird ein Filechoose-Objekt initialisiert. Dieses wird immer wieder verwendet wenn ein Filechooser-Fenster geöffnet werden soll um ein neues File-Objekt zu erstellen in dem der Spielstand gespeichert werden soll.

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 Schreib-

Listing 11: 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 12: 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 13: 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 }
```

prozess aufruft (siehe Listing 13, S. 32, Loader - saveFileAs). Diese Methode nennt sich *actualSavingProcess* und öffnet einen Ausgabestrom auf den der gegebene Text geschrieben wird (siehe Listing 12, S. 32, Loader - actualSavingProcess). Es wird von den einem Ressourcen-Block gebraucht, 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.

Listing 14: Player - Konstruktoren

```

1 // Konstruktor zum starten des Spiels
2 public Player(GUIConnector gui, int idxInPlayerArray, int boardSizeX, int boardSizeY) {
3     this(gui, idxInPlayerArray, new Board(boardSizeX, boardSizeY));
4 }
5
6 // Konstruktor zum Testen ohne Dateiverarbeitung
7 public Player(GUIConnector gui, int idxInPlayerArray, Board board) {
8     this.idxInPlayerArray = idxInPlayerArray;
9     this.districts = new LinkedList<>();
10    this.gui = gui;
11    this.board = board;
12 }
13
14 // Konstruktor zum Testen mit Dateiverarbeitung
15 public Player(GUIConnector gui, int idxInPlayerArray, String strBoard) {
16     this(gui, idxInPlayerArray, new Board(strBoard));
17     this.districts = genDistrictsFromBoard(this.board);
18 }
```

8.3. playerState

player

Einleitung Diese Klasse legt die Grundfunktionen der einzelnen Spielertypen fest. Sie ist abstrakt da es durch den menschlichen Spieler und den vorgegebenen künstlichen Spielertyp mindestens zwei unterschiedliche Spielertypen geben muss.

Felder Ein Spieler besitzt einen Index im Array der *Game*-Klasse, welcher in der Player-Instanz selbst gespeichert wird. Dieser dient immer wieder als Anhaltspunkt um welchen Spieler es sich im Augenblick handelt. Weiterhin besitzt ein Spieler ein *Board* auf dem dieser verschiedene Dominos setzen kann. Um die erzielten Punkte jederzeit direkt errechnen zu können, ohne jedes Mal das Board aufs Neue untersuchen zu müssen, verwaltet der Spieler eine Liste von Distrikten. Weiterhin ist der Spieler, durch eine Gui-Referenz, in der Lage seine Züge direkt auf der Gui darzustellen.

Konstruktor Es gibt drei verschiedene Ausprägungen des Konstruktors (siehe Listing 14, Player - Konstruktoren). Bei dem Konstruktor welcher einen String einliest wird ebenfalls eine Liste an Distrikten gebildet. Im Abschnitt Aufbau der Distrikte wird dies näher erläutert.

Aufbau der Distrikte In der Methode *genDistrictsFromBoard* werden alle Zeilen eines gegebenen Board einzeln untersucht und in eine Liste von Distrikten entsprechend

Listing 15: Player - genDistrictsFromBoard

```

1 private List<District> genDistrictsFromBoard(Board board) {
2     List<District> futureDistrictList = new LinkedList<>();
3     SingleTile[][] cells = board.getCells();
4     for (int y = 0; y < board.getSizeY(); y++) {
5         for (int x = 0; x < board.getSizeX(); x++) {
6             futureDistrictList = addToAppropriateDistrict(cells[x][y],
7                 new Pos(x, y), futureDistrictList);
8         }
9     }
10    return futureDistrictList;
11 }
```

ingeordnet (siehe Listing 15). Hierzu wird eine Methode namens *addToAppropriateDistrict* aufgerufen (siehe Listing 16).

Diese Methode erweitert eine gegebene Liste an Distrikten an der Richtigen Stelle mit einem neuen Element, welches aus dem gegebenen Tile-Objekt und der Position zusammengesetzt wird. Allerdings wird die Liste nur erweitert, wenn das Tile-Objekt ungleich dem der leeren Zelle und dem Stadtzentrum ist. Anschließend wird folgendes Schema angewendet: (siehe Listing 16)

- Zeile 4: Es werden alle Distrikte zwischengespeichert, welche an die gegebene Position angrenzen und den selben Distrikt-Typen aufweisen wie das gegebene Tile-Objekt. Falls es keine solcher Distrikte geben sollte, wird ein neuer Distrikt erzeugt und ebenfalls zwischengespeichert.
- Zeile 6: Nun werden alle generierten / gefundenen Distrikte aus der Liste an Distrikten, welche der Player verwaltet, gelöscht um Duplikate zu vermeiden wenn man später die Distrikte wieder zusammenführt.
- Zeile 7: Die Distrikte vom Zwischenergebnis wird per Konstruktorauftruf der Distriktklasse zusammengeführt. Das Zusammenführen *mehrerer* Distrikte tritt nur auf wenn das gegebene Tile-Objekt in mehrere Distrikte passt.
- Zeile 8: Da das SingleTile-Objekt und die Position bisher nur benutzt wurden um Vergleiche zu bewerkstelligen, werden diese nun in den zusammengeführten Distrikt eingebunden.
- Zeile 9: Der neu generierte / überarbeitete Distrikt wird in die Distrikt-Liste des Spielers überführt.

Die Methode *findPossibleDistricts*, welche in der 4. Zeile verwendet wurde, durchsucht alle Distrikte nach allen Distrikten welche an die gegebene Position angrenzen und den selben Distrikttypen wie das gegebene SingleTile-Objekt besitzen (siehe Listing 17). Es wird hierbei auf eine Methode der Distrikt Klasse namens *typeAndPosMatchCurrDistrict* zugegriffen, welche genau das gerade beschriebene Verhalten implementiert. In der

Listing 16: Player - addToAppropriateDistrict

```

1 private List<District> addToAppropriateDistrict(SingleTile tile, Pos pos,
2                                         List<District> districts) {
3     if (SingleTile.EC != tile && SingleTile.CC != tile) {
4         List<District> possibleDistricts = findPossibleDistricts(tile, pos, districts);
5         districts.removeAll(possibleDistricts); // to avoid duplicates
6         District updatedDistrict = new District(possibleDistricts); // merging districts
7         updatedDistrict.add(tile, pos); // put new element in merged playdistrict
8         districts.add(updatedDistrict);
9     }
10    return districts;
11 }
```

Listing 17: Player - findPossibleDistricts

```

1 private List<District> findOrCreatePossibleDistricts(SingleTile tile, Pos pos,
2                                                 final List<District> districts) {
3     List<District> filteredDistrictList = new LinkedList<District>();
4     for (District currDistrict : districts) {
5         if (currDistrict.typeAndPosMatchCurrDistrict(tile, pos)) {
6             filteredDistrictList.add(currDistrict);
7         }
8     }
9     return filteredDistrictList;
10 }
```

Erklärung der Distrikt-Klasse wird hierauf näher eingegangen (siehe Abschnitt 8.3, Methoden). Falls die Prüfung zutreffen sollte, wird die Ausgabeliste mit dem Distrikt gefüllt und nachdem alle Distrikte durchlaufen wurden zurückgegeben.

Desweiteren bietet die Player Klasse noch eine Möglichkeit die bestehende Liste an bestehenden Distrikten zu überarbeiten ohne die bestehende Distriktliste zu verändern. Die Methode *updateDistricts* (siehe Listing 18, Player - updateDistricts) generiert zuerst ein Kopie des Distrikts (deep copy: neue Referenzen) um anschließend einen gegebenen Domino in diese Kopie einzupflegen (siehe Listing 16, Player - addToAppropriateDistrict). Dieser Mechanismus wird vor allem vom künstlichen Spieler benötigt, da dieser viele Dominos überprüfen möchte und für jeden dieser Dominos die Punkteanzahl berechnet. Hierbei soll die bestehende Distriktliste nicht verändert werden, dazu aber in der Beschreibung des *DefaultAIPlayers* mehr.

Um die Reihenfolge der Spieler am Ende eines Spiels berechnen zu können wird das Interface *Comparable* implementiert (siehe Listing 19, S. 36, Player - compareTo). Hierbei muss gewährleistet sein, dass es sich bei dem übergebenen Objekt um eine Player-Referenz handelt, da sonst kein Vergleich durchgeführt werden kann. Anschließend wird geprüft ob beide Spieler gleich viele Punkte vorzuweisen haben. Falls dies der Fall sein sollte wird die Größe der Distrikte verglichen. Zum Vergleichen der Distrikte wird die Methode *getLargestDistrictSize* verwendet, welche nichts anderes tut als durch die Dis-

Listing 18: Player - updateDistricts

```

1 protected List<District> updateDistricts(final List<District> districts, Domino domino) {
2     // deep copy of whole district list
3     List<District> output = new LinkedList<>();
4     for (District currDistrict : districts) {
5         output.add(currDistrict);
6     }
7     // adding domino to slots
8     output = addToAppropriateDistrict(domino.getFstVal(), domino.getFstPos(), output);
9     output = addToAppropriateDistrict(domino.getSndVal(), domino.getSndPos(), output);
10    return output;
11 }
```

Listing 19: Player - compareTo

```

1 public int compareTo(Object o) {
2     assert null != o && (o instanceof Player);
3     Player other = (Player) o;
4     if (other.getBoardPoints() == this.getBoardPoints()) {
5         return this.getLargestDistrictSize() - other.getLargestDistrictSize();
6     } else {
7         return this.getBoardPoints() - other.getBoardPoints();
8     }
9 }
```

triktliste des jeweiligen Spielers zu iterieren und zu zählen wie viele Positionen die Liste jeweils hält. Am Ende wird die maximale Anzahl ausgegeben. Falls die Anzahl der Punkte nicht übereinstimmt werden einfach diese verglichen (beziehungsweise voneinander abgezogen).

Um das Ergebnis auf der Gui darstellen zu können wird ein TreeItem generiert. Dieses beinhaltet den Spielernamen, die erzielten Punkte und eine genaue Aufschlüsselung welche Punkte aus welchem Distrikt stammen.

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.

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 20, S. 38, 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 21, 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* die, 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 22, S. 38, District - typeAndPosMatchCurrDistrict). Hierbei wird geschaut ob die SingleTile-Referenz den Distriktyp mit dem aufrufendem Distrikt teilt oder nicht. Dazu wird die Methode *matchingDistrictTypes* verwendet (siehe Listing 23, S. 39, District - matchingDistrictTypes).

Um festzustellen ob sich die gegebene Position neben dem aufrufenden Distrikt befindet wird die Methode *elemPosIsNextToExistingElem* verwendet (siehe Listing 24, S. 39, 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 einzeln in neue Ausgabelisten kopiert. Ein neuer Distrikt wird per Konstruktoraufruf geformt und zurückgegeben.

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

Listing 20: 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 21: 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 22: District - typeAndPosMatchCurrDistrict

```

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

Listing 23: District - matchingDistrictTypes

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

Listing 24: 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 }
```

8.4. bankSelection

Bank

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

Konstruktoren Die Klasse verfügt über drei verschiedene Konstruktoren (siehe Listing 25, S. 41, 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 zu 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.

Verweis auf Test-Kapitel mit Random-Objekt setzen. (2. Punkt)

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 26, S. 41, Bank - selectEntry).

noch nicht fertig.

Listing 25: 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 26: 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 }
```

Choose

Entry

Bild einbinden

Einleitung 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 hier eine eigene Klasse dafür 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 das 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.

Referenz zu Syntax-Richtlinien einbetten.

Listing 27: 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  }

```

8.5. differentPlayerTypes

DefaultAIPlayer

Einleitung 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.

doInitialSelect selektiert auf der *aktuellen* Bank (siehe Listing 27, S. 44, 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 28, S. 45, DefaultAIPlayer - selectFromBank). Diese sucht für jeden Domino der Bank die am besten geeignete Position und schreibt die am besten geeignete Position 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 werden und an die Methode *genBestChoose* weitergereicht werden (siehe Listing 29, S. 46, DefaultAIPlayer - genBestChoose). Diese Methode setzt die Position des Dominos einzeln auf alle Felder des Bretts und schaut ob der Domino passt. Dies ist so nur möglich, da vorher ein Deep-Copy angefertigt wurde

Listing 28: 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 }
```

um den Domino auf der Bank nicht zu verändern. Falls dies der Fall ist wird ein neues Choose-Objekt erstellt. Falls dieses effizienter sein sollte als das aktuelle maximale Choose-Objekt, überschreibt dieses das alte (siehe Listing 30, S. 47, DefaultAIPlayer - mostEfficient).

Falls noch Zeit ist den Selektiervorgang genauer Beschreiben

Es fehlen doLastTurn und doStandardTurn

Listing 29: 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 }
```

HumanPlayer

Listing 30: 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 }
```

playerType

Listing 31: 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 }
```

8.6. logicTransfer

Converter

Einleitung 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 31, S. 49, Converter - readStr).

1. Schritt: Es wird eine grobe Struktur geschaffen um einfach 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 schon mal überprüft ob die Bezeichner stimmen oder nicht. Dies wird über folgenden regulären Ausdruck getan: (MATCHES_TAGS)

"(<Spielfeld[^>]*>\n(?s) [^<>]*)*<Bänke>\n(?s) [^<>]*<Beutel>\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 `genDescriptveField` aufgerufen (siehe Listing 32, S. 51, 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 in dem bei jeder schließenden spitzen Klammer geteilt. 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 33, S. 52, Converter - `fillFieldsWithDescriptiveBlocks`). Hier werden die im vorherigen Schritt erzeugten Felder durchlaufen, es wird jeweils der das Feld mit 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 erst Feld des zweidimensionalen Arrays zugreift wird im Case-Verteiler der `BOARD_IDENTIFIER` greifen und es wird zuerst einmal die Spielfeld-Syntax überprüft. Hierbei spielen Dinge wie Leerzeichen oder die Dimensionen des Bretts eine Rolle, aber auch invalide Zellen werden abgefangen. Da in meinem Modell die Spieler ein Feld mit dem Brett besitzen, habe ich den Besitzer des Spielfeldes gleich mit initialisiert, da so die Liste der Distrikte gleich mit aufgebaut werden. Hierzu wird die Methode `convertStrToPlayerWithDefaultOccupancy` aufgerufen. Diese ist eine Hilfsmethode und macht nichts anderes als anhand des übergebenen Indizes 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 8.5 auf S. 48 geläkt.

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 34, S. 53, 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 / 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 35, S. 53, 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 32: 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 33: 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 }
```

Game**Exceptions** paragraphGUI2Game**GUIConnector****PossibleField**

Listing 34: 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 35: 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 }
```

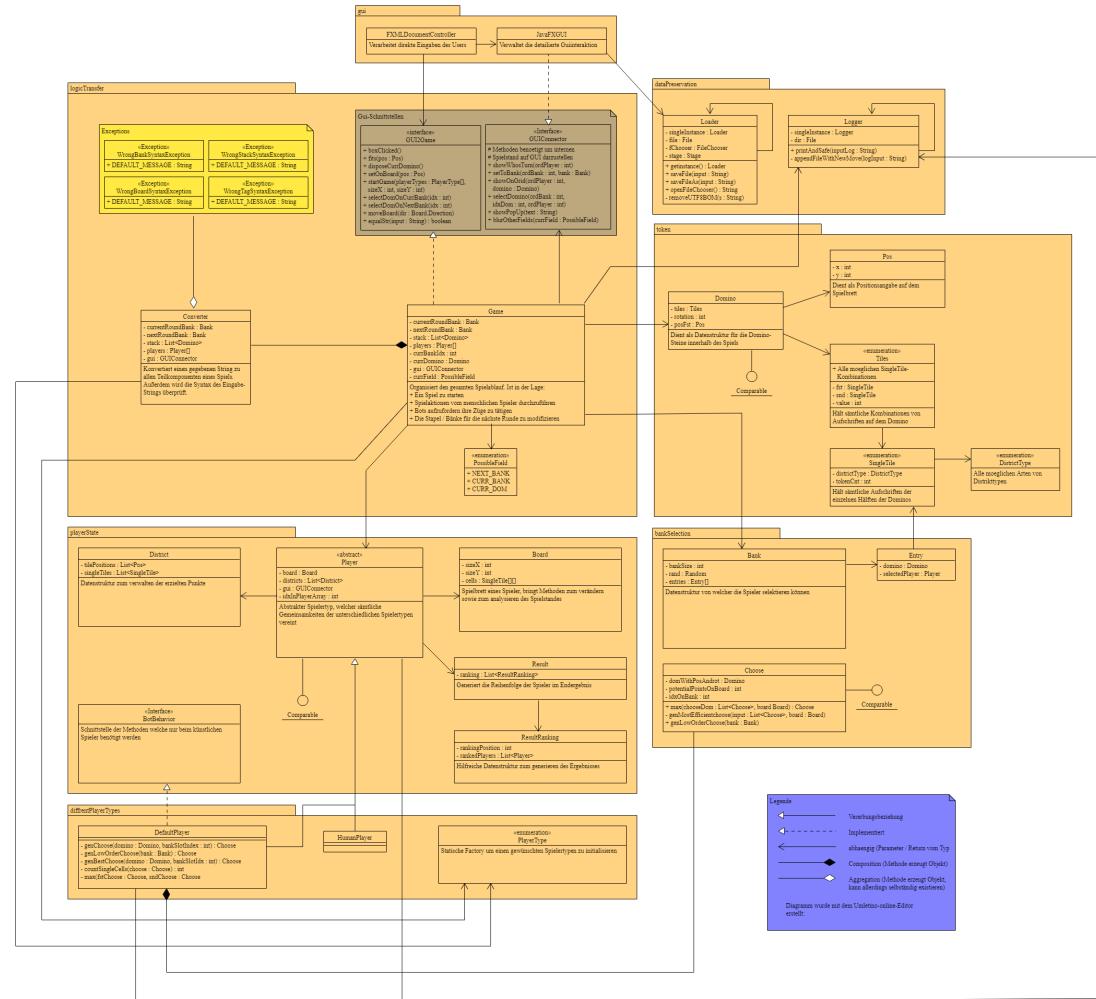


Abbildung 15: Vereinfachte UML-Darstellung des gesamten Projekts

9. Programmorganisationsplan

Einleitung Im folgenden Abschnitt wird eine Gesamtübersicht über das Zusammenspiel sämtlicher Klassen gegeben. Das dazu verwendete UML-Diagramm (siehe Abbildung 15, S. 54) ist leider relativ groß geworden. Falls Sie eine vergrößerte Version in Ihrem Bildbetrachter sehen möchten, folgen Sie bitte dem Link in Lesezeichen- bzw. Anlagenübersicht des Pdf-Readers¹. 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 das

Falls noch Zeit ist, Bild schoener ans Projekt anhaengen

¹<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 verwalten 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 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 Spielertypen zu interpretieren und in eine Spielerinstanz umzuwandeln. Der Stack wird nach dem Initialisieren der Spieler ebenfalls gefüllt.

Alternative Spielerinitialisierung Dieser ganze Vorgang ist relativ aufwendig man hätte sämtliche Operationen theoretisch auch im Konstruktor behandeln können, auf lange Sicht ist es aber effektiver Sie hier anzusiedeln da man sonst nur sehr umständlich ein Auswahlfeld für mögliche Spieler starten kann. Hierzu folgendes Szenario: Es gibt N Spielerarten unter denen der Benutzer seine Gegner wählen kann. Dafür gibt es ein eigenes Auswahlfenster, welches vor dem eigentlichen Spiel auftauchen soll (siehe Abbildung 16, S. 56). 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 Methode welche losgelöst vom Konstruktor das Spiel mit übergebenen Spielerarten 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 Möglichkeiten 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.

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 eine Domino sowie eine Spielerreferenz. Auf der Bank ist es mög-

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 }
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 }
```

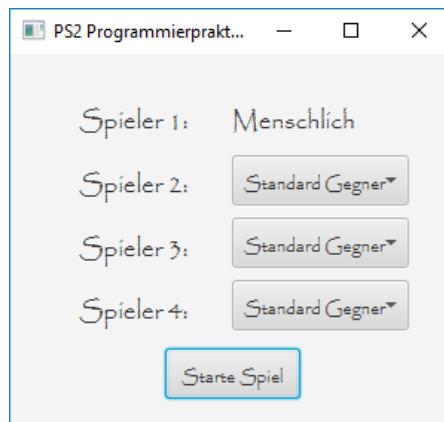


Abbildung 16: Auswahlfenster

lich 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 abhängen, befinden sich sämtliche Funktionalitäten dieses Spielertyps in der Game-Klasse selbst.

Ablegen Mit dem Ablegen eines Steins auf dem Spielfeld muss nicht nur das spielereigene 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 den selben Platz zuzuweisen und bringt selbst noch einige hilfreiche Methoden um die gesamte Rechnung / 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 37: 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 }
```

10. Programmtests

Erklärung des Toolkits Um effektiv Tests zum Einlesen sowie Ausgeben von Spielständen aus einer .txt Datei zu gestalten, wurde ein 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, musste viele Methoden ausgetauscht werden.

Neben einer Festlegung welcher Dateityp bearbeitet werden kann, liefert diese Klasse vor allem einen Pfad an 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 37, S. 58, 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 für die Dateiverarbeitung zu schaffen. In der Methode *read* wird diese Methode aufgerufen um einen Game Konstruktor zu füllen und ein vollwertiges Spiel zurückzugeben.

Toolkit
noch
einfü-
gen und
schreiben
wie man
dies findet

Eine weitere Funktionalität des Toolkits besteht darin zwei gegebene Dateien mit dem selben Namen auf ihre Gleichheit zu prüfen (siehe Listing 38, S. 59, 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 Im Ordner namens „test“ befindet sich ein Unterordner namens „fileTests“. Dies ist der Oberordner, in welchem sich alle Testdateien zum Dateieinlesen / ausgeben befinden. Die Dateien mit dem Präfix „inv_“ spiegeln invalide, während die mit dem Präfix „val_“ valide Dateien wiederspiegeln. Da es 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

Listing 38: TestToolkit - assertFilesEqual

```

1 public static void assertFilesEqual(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 }
```

Unitests zu den einzelnen Datenstrukturen bereits behandelt wurden. Invalide Datei-tests laufen immer nach dem selben Schema ab (siehe Listing 39, S. 60, 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 werden 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 8.6, S. 49). 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 den selben Spielstand enthält. In der *writeAndAssert*-Methode wird zuerst eine neue Datei mit dem gegebenen Namen in dem Unterordner namen „results“ erstellt. Anschließend ist der Loader zum abspeichern zuständig. Wie der Loader vorgeht um eine Datei abzurufen wird im Abschnitt 8.2, S. 31 näher erläutert.

Testfälle			
Auflistung wichtiger Testfälle	Testfall	erwartetes Ergebnis	erzieltes Ergebnis
	test	asdf	abc

Tabelle
unvoll-
staendig

Listing 39: 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 }
```

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. Im Anhang Eins