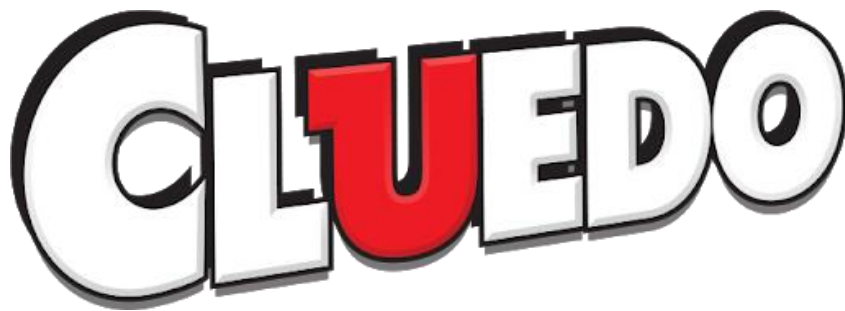


# Programmierpraktikum



1

*Cluedo Wintersemester 20/21*

Michael Smirnov - Medieninformatik

Matrikelnummer : minf103430

Fachsemester: 6

Verwaltungssemester: 6

---

<sup>1</sup> [http://shuffle.cards/de\\_de/license/cluedo](http://shuffle.cards/de_de/license/cluedo)

# 1. Inhaltsverzeichnis

1. Inhaltsverzeichnis .....	1
2. Benutzerhandbuch für den User .....	2
2.1 Ablaufbedingungen .....	2
2.2 Programminstallation/Programmstart .....	2
2.3 Bedienungsanleitung .....	3
2.3.1 Das Spiel Cluedo.....	3
2.3.2 Spielziel .....	3
2.3.3 Die Spielregeln .....	4
2.3.3 Das User Interface (die GUI).....	10
2.4 Fehlermeldungen .....	16
2.4.1 Fehler bei Programmstart .....	16
2.4.1 Fehler während dem Laden oder Speichern eines Spielstandes .....	17
2.4.2 Fehler zur Laufzeit .....	19
3. Programmiererhandbuch für Software-Entwickler .....	20
3.1 Entwicklungskonfiguration .....	20
3.2 Problemanalyse und Realisation .....	21
3.2.1 Das User Interface (die GUI).....	21
3.2.2 Die Spiellogik:.....	24
3.3 Programmorganisationsplan .....	32
3.4 Beschreibung grundlegender Klassen.....	33
3.4.1 Die Klassen der GUI .....	33
3.4.2 Die Klassen der Logic .....	37
3.5 Programmtests.....	41
3.5.1 Spielstart .....	41
3.5.2 Während des Spielablaufs .....	42
3.5.3 Laden und speichern.....	43
3.5.4 Ausgewählte Spielsituationen .....	44

## 2. Benutzerhandbuch für den User

---

### 2.1 Ablaufbedingungen

Zur Ausführung des Programms "Cluedo" ist eine installierte Java Laufzeitumgebung der Version 11 oder aktueller unter Windows notwendig.

Für die Interaktion mit einigen Programmelementen ist außerdem eine Maus oder ein Trackpad notwendig.

Software	Java Laufzeitumgebung Version 11 oder aktueller.
Hardware	Eine Maus oder ein Trackpad für die Interaktion mit einigen Programmelementen.

### 2.2 Programminstallation/Programmstart

Für den Start des Programms, muss die `Cluedo_Smirnov_ws20.jar` Datei ausgeführt werden. Dies geschieht, bei einer korrekt gesetzten Dateiendung und vorhandener Java Laufzeitumgebung  $\leq 11$  unter Windows mit einem Doppelklick auf die Datei automatisch. Ansonsten kann das Programm ebenfalls über die Konsole ausgeführt werden. Dafür muss zuerst in das Verzeichnis navigiert werden, in welchem sich die `.jar` Datei befindet. Danach kann das Programm mit folgendem Aufruf ausgeführt werden:

```
java -jar Cluedo_Smirnov_ws20.jar
```

Weitere Dateien zur Ausführung sind nicht notwendig.

## 2.3 Bedienungsanleitung

### 2.3.1 Das Spiel Cluedo<sup>2</sup>

Bei Cluedo handelt es sich ursprünglich um ein Brettspiel welches erstmals 1948 veröffentlicht wurde und sich seitdem großer Beliebtheit erfreut. Es können 2-6 Spieler an einer Partie teilnehmen, welche zwischen 30 und 45 Minuten dauert. Dabei beginnt jede Spielfigur an einer festen Position und bewegt sich abhängig von der gewürfelten Zahl auf dem Spielfeld. Dabei können Räume betreten werden, um diese zu untersuchen und in diesen eine Verdächtigung auszusprechen. Durch Hinweise, welche von den Mitspielern nach einer Verdächtigung geäußert werden, soll ermittelt werden, wie genau der Mord geschehen ist. Dabei sind Tatort, Mordwaffe und Täter zu identifizieren.

### 2.3.2 Spielziel

Das Ziel des Spiels ist es einen Mordfall in dem Haus aufzuklären, indem man sich mit seinen Mitspielern befindet. Dazu müssen die Räume auf dem Spielplan untersucht und Mitspieler nach Hinweisen gefragt werden, um durch Kombination der Hinweise nach dem Ausschlussverfahren den Mord aufzuklären. Gewonnen hat derjenige Spieler, welcher alle drei für den Mordfall relevanten Fakten richtig ermittelt. Gelingt das dem Spieler nicht, hat er verloren und die Spielpartie ist zu Ende.

---

<sup>2</sup> <https://en.wikipedia.org/wiki/Cluedo>

## 2.3.3 Die Spielregeln<sup>3</sup>

### 2.3.3.1 Spielfeld

Das Spielfeld auf der linken Seite des Programmfensters zeigt die Räume des Hauses, in dem der Mord verübt wurde. Dieses besteht aus neun Räumen, welche alle durch den Flur miteinander verbunden sind. Die sechs Spielfiguren repräsentieren die verdächtigen Personen, welche alle als Mörder in Frage kommen.



Figure 1 Spielfeld mit den verschiedenen Spielfiguren

Die Spielfigur	Ihre Farbe auf dem Spielfeld
Fräulein Ming	Rot
Oberst von Gatow	Gelb
Frau Weiß	Weiß
Direktor Grün	Grün
Baronin von Porz	Blau
Professor Bloom	Violett

Außerdem befinden sich zu Beginn des Spiel jeweils eine von insgesamt sechs möglichen Tatwaffen in den Räumen des Hauses.

<sup>3</sup> Teilweise aus den Originalspielregeln übernommen und angepasst.  
<https://intern.fh-wedel.de/fileadmin/mitarbeiter/ne/SA/WS2021/Spielanleitung.pdf>

### 2.3.3.2 Karten im Spiel

Die Insgesamt 21 Karten repräsentieren somit die neun Räume, sechs Waffen und sechs Personen. Diese Karten werden zu Beginn eines Spiels zufällig gemischt und eine Karte jedes Kartentyps (Ort, Waffe, Person) werden als Lösung des Spiels festgelegt. Diese werden aus den zur Verfügung stehenden Karten entnommen. Der Rest der Karten wird reihum beginnend bei dem User verteilt, bis keine Karten mehr übrig sind.

Die zur Verfügung stehenden Karten sind in der ersten Spalte der Notizen, welche sich auf der rechten Seite des Programmfenters befinden, zu sehen.

	Fraulein Ming	Oberst von Gallow	Frau Weiß	Direktor Grün	Baronin von Porz	Professor Bloom
Fraulein Ming	eigene Karte	.....	.....	.....	.....	.....
Oberst von Gallow	.....	.....	.....	.....	.....	.....
Frau Weiß	.....	.....	.....	.....	.....	.....
Direktor Grün	.....	.....	.....	.....	.....	.....
Baronin von Porz	eigene Karte	.....	.....	.....	.....	.....
Professor Bloom	.....	.....	.....	.....	.....	.....
Pistole	eigene Karte	.....	.....	.....	.....	.....
Dolch	.....	.....	.....	.....	.....	.....
Seil	eigene Karte	.....	.....	.....	.....	.....
Kerzenleuchter	.....	.....	.....	.....	.....	.....
Rohrzange	.....	.....	.....	.....	.....	.....
Heizungsrohr	.....	.....	.....	.....	.....	.....
Eingangshalle	.....	.....	.....	.....	.....	.....
Veranda	eigene Karte	.....	.....	.....	.....	.....
Speisezimmer	.....	.....	.....	.....	.....	.....
Küche	.....	.....	.....	.....	.....	.....
Salon	.....	.....	.....	.....	.....	.....
Musikzimmer	.....	.....	.....	.....	.....	.....
Billardzimmer	eigene Karte	.....	.....	.....	.....	.....
Bibliothek	.....	.....	.....	.....	.....	.....
Arbeitszimmer	.....	.....	.....	.....	.....	.....

Figure 2 Notizen des Spielers mit den Karten im Spiel

### 2.3.3.3 Karten auf der Hand

Die Karten auf der Hand des Users können aus den drei Listen, welche jeweils eine Kategorie von Karten enthalten, entnommen werden. Sie befinden sich in der Mitte des Programmfensters. Diese wurden am Anfang des Spiels mit Karten gefüllt und bleiben im gesamten Spielverlauf dieselben.

Eigene Karten	
Personen	
Frau Weiß	
Tatwaffen	
Rohrzange	
Kerzenleuchter	
Räume	
Arbeitszimmer	
Salon	
Speisezimmer	

Figure 3 Eigene Karten auf der Hand

#### 2.3.3.4 Ein Spielzug

Der User beginnt das Spiel immer mit der Figur von Fräulein Ming. Zu Beginn jedes Zuges wird ein (W6) Würfel gewürfelt und angezeigt, dieser repräsentiert die für diese Runde zur Verfügung stehenden Schritte.

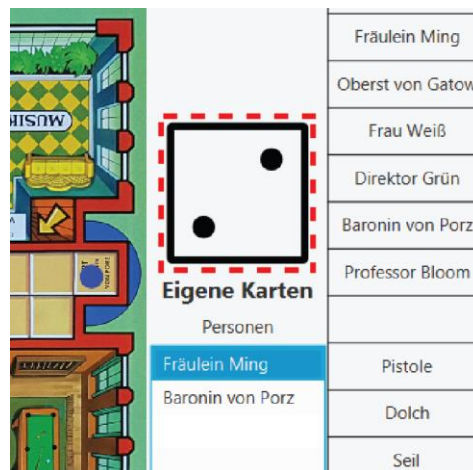


Figure 4 Würfel für die Schrittzahl

Der Spieler klickt nun mit der linken Maustaste eine Kachel des Flures an, um sich dorthin zu bewegen oder in einen Raum, um in diesen hinein zu geben, vorausgesetzt die Anzahl der Schritte reicht dafür aus. Falls dies nicht zutrifft erscheint ein Dialog, welcher den User darauf hinweist, dass die ausgewählte Position nicht erreichbar ist.

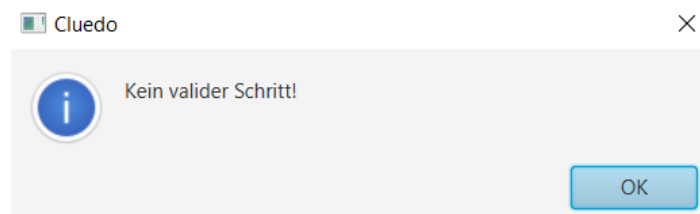


Figure 5 Dialog bei ungültigem Schritt

Die schwarzen Kreise auf dem Spielfeld heben dabei die Positionen hervor, welche in diesem Zug erreichbar sind.



Figure 6 Spielfeld mit Markierungen für die Möglichen Schritte

Es müssen pro Zug alle Schritte verbraucht werden und es kann nicht diagonal über die Kacheln im Flur gegangen werden. Eine Ausnahme bildet dabei der Fall, dass der Spieler auf dem Flur von allen Seiten von anderen Spielfiguren oder Spielfeldelementen (Wände oder Räume) eingeschlossen ist. Dann ist es, unabhängig von der gewürfelten Schrittzahl, möglich auf die aktuell besetzte Position zu klicken, um einen Zug zu tätigen. Falls der Spieler in einem Raum eingeschlossen wurde, muss dieser in diesem Raum bleiben und mit einem Klick auf jenen erneut eine Verdächtigung aussprechen. Außerdem kann eine Kachel auf dem Flur niemals von zwei Spielern gleichzeitig besetzt werden. Auch das Überspringen ist nicht zulässig. Der Spieler auf dem Spielfeld muss umgangen werden und falls dieser einen Raum blockiert, kann jener über diese Tür nicht mehr betreten werden. In einem Zimmer wiederum, dürfen sich mehrere Spieler und Tatwaffen aufhalten.

#### 2.3.3.5 Geheimgänge

Es existieren Geheimgänge, welche die Veranda mit dem Musikzimmer und die Küche mit dem Arbeitszimmer verbinden. Diese können genutzt werden, indem auf den Zielraum, während der User am Zug ist und sich bereits in einem dieser Räume befindet, geklickt wird.



Figure 7 Markierung für einen Geheimraum

#### 2.3.3.6 Verdächtigung aussprechen

Betritt ein Spieler einen Raum, muss dieser eine Verdächtigung aussprechen, welche aus dem gerade betretenen Raum, einer Waffe und einer Person besteht. Die Auswahl von Person und Waffe erfolgt jeweils über ein Dropdown Element. Die Verdächtigung wird mit einem Klick auf den "Ok"-Button bestätigt.

Gib deine Verdächtigung ein!

Person:

Tatwaffe:

Raum:

Figure 8 Verdächtigungsdialog



Der betretene Raum kann nicht in demselben Zug wieder verlassen werden. Der verlassene Raum kann auch nicht in demselben Zug wieder betreten werden. Ein Spieler darf einen Raum auch nicht besetzen, indem er eine Runde aussetzt. Er muss weiterrücken oder den Raum über eine Geheimtür verlassen. Eine Ausnahme davon bildet der Fall, dass der Spieler in einem Raum ohne Geheimtür eingeschlossen wurde. Nur dann darf dieser durch einen Klick auf den Raum, in dem er sich aktuell befindet, erneut eine Verdächtigung aussprechen.

#### 2.3.3.7 In einen Raum zitiert werden

Wird eine Verdächtigung ausgesprochen wird die verdächtige Person und Waffe in den Raum "zitiert" in welchem die Verdächtigung geäußert wurde, unabhängig davon wo sich die Person/Waffe vorher befand. Alle Spielfiguren können unter Verdacht gestellt werden, auch die eigene. Wurde man in einen Raum "zitiert" hat man, sobald man wieder am Zug ist, die Wahl, ob man durch einen Klick auf eben diesen Raum eine Verdächtigung äußern oder den Raum verlassen möchte.

#### 2.3.3.8 Reaktion auf einen Verdacht

Nach einem Verdacht muss jeder Spieler den Verdacht widerlegen oder bestätigen, indem dieser eine in dem Verdacht genannte Karte demjenigen der den Verdacht ausgesprochen hat zeigt, oder - falls derjenige keine dieser Karten besitzt - bestätigt, indem er keine Karte zeigt.

Dem User wird dabei ein Dialog angezeigt in welchen die möglichen Optionen vorausgefüllt und klickbar sind.

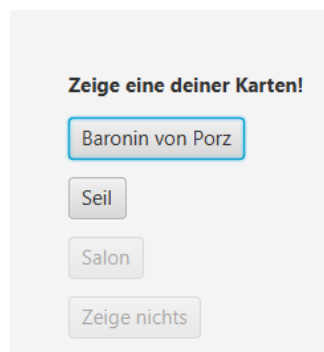


Figure 9 Dialog beim Zeigen einer Karte

Auch wenn niemand in der Lage ist die Verdächtigung durch Vorzeigen einer der genannten Karten zurückzuweisen, geht das Spiel normal weiter oder der Spieler, der die Verdächtigung äußert, erhebt Anklage.

### 2.3.3.9 Anklage erheben

Wenn der User sich sicher ist die Lösung des Mordes zu kennen, kann er, sobald er wieder am Zug ist, mit einem Klick auf den "Anklage"-Button den Anklage-Dialog öffnen. Dabei spielt es keine Rolle wo sich der User gerade befindet. In dem Dialog kann die Tatperson, Tatwaffe und den Tatraum mithilfe eines Drop-Down-Menüs ausgewählt, und mit einen Klick auf den "Ok"-Button bestätigen werden. Alternativ kann mit dem "Abbrechen"-Button der Anklage-Dialog geschlossen und das Spiel ganz normal fortgesetzt werden.

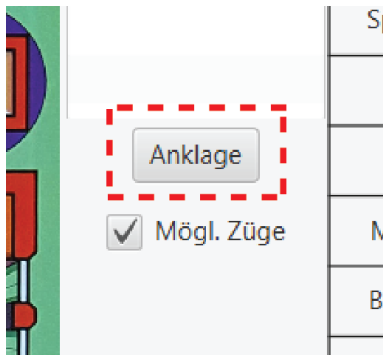


Figure 11 Button, um die Anklage zu äußern

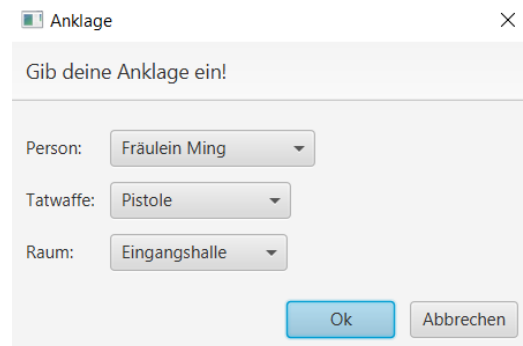


Figure 10 Dialog, um die Anklage zu äußern

Trifft die Annahme zu ist das Spiel gewonnen, falls nicht, ist es verloren. Die Mitspieler können ebenfalls eine Anklage erheben, sobald diese am Zug sind. In jedem Fall wird das Spiel mit einem entsprechenden Dialog beendet und ein neues wird gestartet.

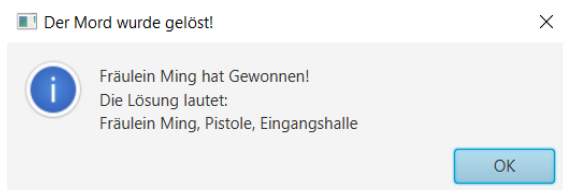


Figure 12 Dialog bei Sieg

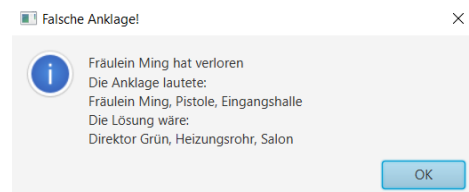
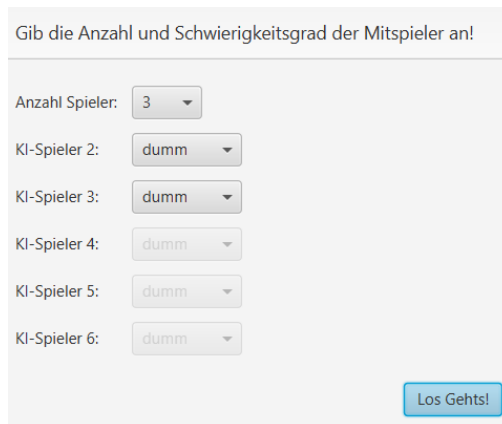


Figure 13 Dialog bei Niederlage

## 2.3.3 Das User Interface (die GUI)

### 2.3.3.1 Spielstart

Wird das Spiel das erste Mal gestartet, muss der User zunächst die Anzahl der Spieler, sowie die Schwierigkeitsstufen je Computergegner angeben. Die mögliche Anzahl der Spieler insgesamt beträgt drei bis sechs. Es können die drei Schwierigkeitsstufen "dumm", "normal" und "schlau" jeweils pro Spieler über ein Drop-Down-Menü ausgewählt werden. Initial sind drei Spieler und die Schwierigkeitsstufen "dumm" ausgewählt.



The screenshot shows a dialog box titled "Gib die Anzahl und Schwierigkeitsgrad der Mitspieler an!". It contains several input fields: "Anzahl Spieler:" with a dropdown menu showing "3", and five "KI-Spieler" labels (2 through 6) each followed by a dropdown menu showing "dumm". A blue button labeled "Los Gehts!" is located at the bottom right of the dialog.

Figure 14 Dialog beim Spielstart

Nachdem der Spieler diese Angaben getätigt hat, kann ein neues Spiel über einen Klick auf den Button "Los Gehts!" gestartet werden. Da diese Angaben zwingend erforderlich sind, kann der User während der Dialog offen ist das restliche Programm nicht bedienen.

Nach der Auswahl der Spieleranzahl und deren Schwierigkeitsstufen wird das Fenster, in dem das Hauptspiel stattfindet, wird mit allen notwendigen Informationen und Grafiken beladen.

### 2.3.3.2 Spielfenster

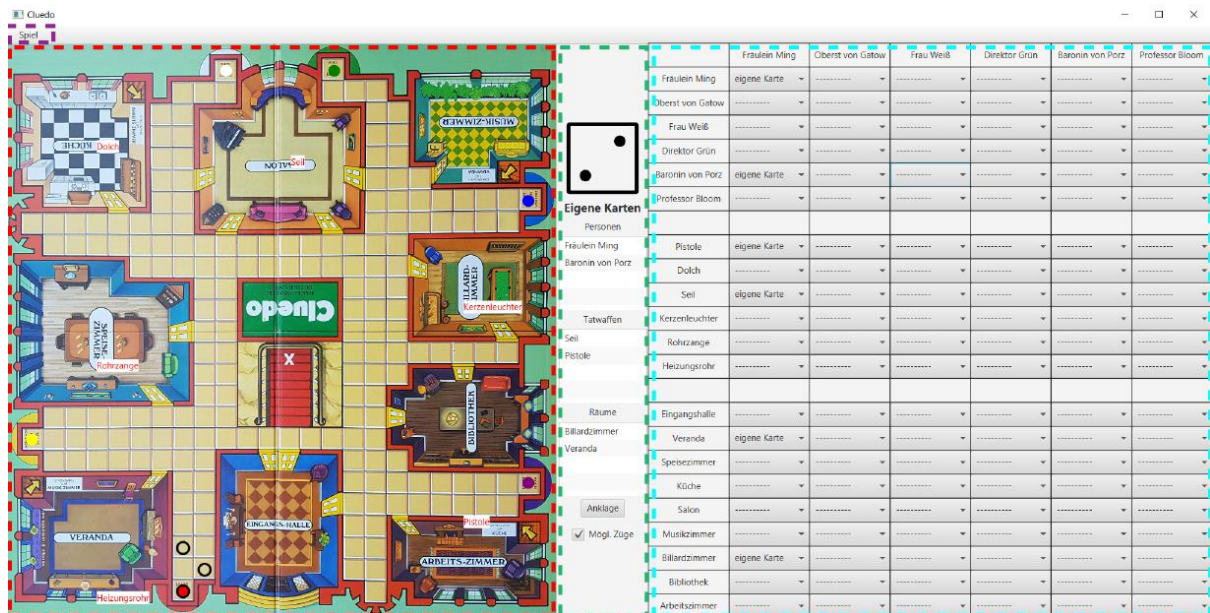


Figure 15 Die vier Bestandteile des Spielfensters. Das Spielfeld (rot), Die Spielinformationen und Bedienelemente des aktuellen Spiels (grün), Die Notizen des Users (hellblau), Die Spieloptionen (violett)

#### 2.3.3.2.1 Spielfeld

Das Spielfeld, auf dem sich die Spielfiguren bewegen (Fig.15, rot) zeigt an, welche Spielfigur sich gerade wo befindet. Außerdem sind dort die sechs Tatwaffen zu sehen, welche sich initial auf die neun Räume verteilen. Wenn der User an der Reihe ist, kann durch einen Linksklick auf das Spielfeld ein Zug getätigt werden. Die schwarzen Kreise heben dabei die Positionen hervor, zu welchen sich der Spieler, gegeben der gewürfelten Schrittzahl, hinbewegen kann.

#### 2.3.3.2.2 Spielinformationen und Bedienelemente des aktuellen Spiels

Die aktuellen Spielinformationen und Bedienelemente für den User (Fig.15, grün) werden in der Mitte des Fensters dargestellt. Ganz oben wird die zuletzt gewürfelten Würfelseite angezeigt. Mittig befinden sich die Karten auf der Hand des Users, welche nach Kartentyp sortiert sind.

Darunter befindet sich ein Button, um eine Anklage zu äußern und das aktuelle Spiel somit zu beenden. Ganz unten ist noch eine Checkbox "Mögl. Züge", welche es dem User ermöglicht die Markierung der betretbaren Spielfelder ein- und auszublenden.

### 2.3.3.2.3 Notizen des Users

Die Notizen für den User (Fig.15, hellblau) befinden sich auf der linken Seite des Fensters. Die Spalten beschreiben hierbei die Notizen über den oben angegebenen Spieler. Die Zeilen beschreiben die Karte, zu der diese Notiz gehört.

Die Notizen über die Spielfigur des Users "Fräulein Ming" unterscheiden sich von denen der anderen Spieler. Bei den eigenen Notizen kann zu jeder Karte eingetragen werden, ob es sich dabei um eine Karte handelt, welcher der User selbst auf Hand hat und wie häufig er diese bereits einem anderen Spieler gezeigt hat.

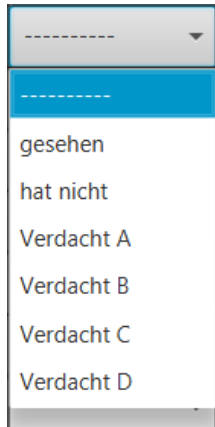


Figure 17 Notizen über andere Spieler

Bei den Notizen über die anderen Spieler kann eingetragen werden, ob diese Karte von einem bestimmten anderen Spieler gezeigt wurde oder, ob diese Karte teil eines Verdachts ist. Das passiert, wenn erkannt wird, dass der Spieler auf einen Verdacht eines anderen Spielers hin eine Karte gezeigt hat, jedoch noch nicht bekannt ist um welche der drei möglichen Karten es sich genau handelt.

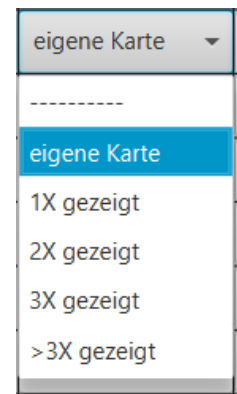


Figure 16 Eigene Notizen

### 2.3.3.2.4 Die Spieloptionen

Die Menüleiste den Spieloptionen (Fig.15, violett) Hier kann der User mit den jeweiligen Menüpunkten ein neues Spiel Starten, ein bereits gespeichertes Spiel laden oder das aktuelle Spiel speichern. Dieses ist nur bedienbar, wenn der User am Zug ist.

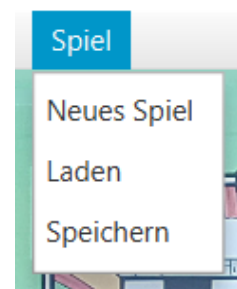


Figure 18 Spieloptionen

### 2.3.3.3 Während eines Spiels

Innerhalb des Spielverlaufs, welcher zu großen Teilen bereits im Kapitel “2.3.3 Die Spielregeln” beschrieben wurde, sind weitere Interaktionen mit dem Spiel notwendig. Im Folgenden wird auf diese und deren Besonderheiten eingegangen.

#### 2.3.3.3.1 Eine Verdächtigung wurde ausgesprochen

Sobald ein anderer Spieler eine Verdächtigung äußert wird dieser Verdächtigungs-Dialog angezeigt. Dieser informiert den User darüber wer die Verdächtigung ausgesprochen hat und welche Karten diese beinhaltet. Währenddessen kann nicht mit dem Spielfenster interagiert werden. Danach wird der User aufgefordert auf diesen Verdacht zu reagieren dies wird in dem Kapitel “2.3.3.8 Reaktion auf einen Verdacht” näher beschrieben.

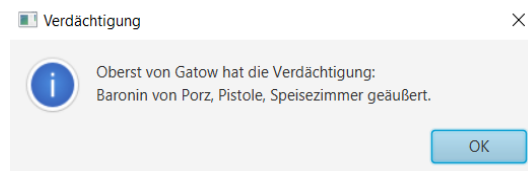


Figure 19 Dialog bei Verdächtigung

#### 2.3.3.3.2 Verdächtigungsergebnis

Nachdem alle Spieler im Spiel auf eine Verdächtigung reagiert haben wird ein Dialog angezeigt, welcher eine Zusammenfassung über die geäußerte Verdächtigung sowie die Reaktionen der Spieler auf diese Verdächtigung anzeigt. Derjenige der die Verdächtigung geäußert hat wird dort natürlich nicht aufgeführt. Je nachdem, ob der User selbst die Verdächtigung geäußert hat oder ein anderer Spieler, stehen bei den Spielernamen die gezeigten Karten oder nur, ob diese eine Karte gezeigt haben. Während dieser Dialog angezeigt wird, können nur die Notizen genutzt werden um daraus gewonnen Erkenntnisse einzutragen. Die anderen Bedienelemente und das Spielfeld reagieren währenddessen nicht auf die Klicks des Users.

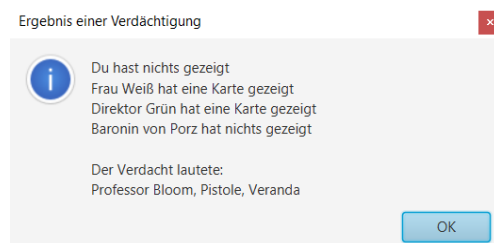


Figure 21 Dialog bei eigener Verdächtigung

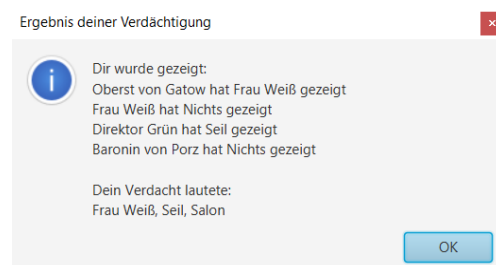


Figure 20 Dialog bei Verdächtigung eines anderen Spielers

#### 2.3.3.3.2 Unterbrechung des laufenden Spiels

Falls während des Spiels mit einem Klick auf den roten “schließen”-Button oben rechts im Fenster das Spiel beendet werden, über das Spielmenü oben links ein neues Spiel begonnen oder ein Spiel geladen werden möchte, erscheint zuerst ein Dialog. Dieser Dialog weist den User darauf hin, dass diese Aktion das laufende Spiel beendet. Der User hat nun die Möglichkeit diese Aktion abubrechen und weiterzumachen, das aktuelle Spiel zu speichern oder ohne zu speichern die Aktion fortzuführen. Falls gespeichert werden soll erscheint der Spiel-Speichern-Dialog auf den in Kapitel “2.3.3.4 Speichern” näher eingegangen wird.

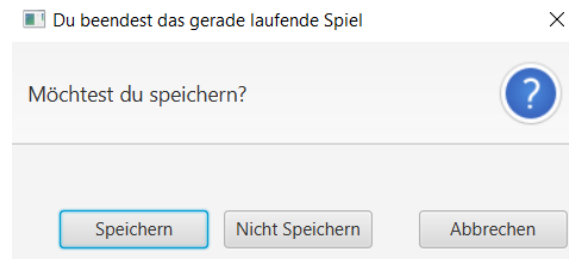


Figure 22 Dialog bei Beendung eines laufenden Spiels

### 2.3.3.4 Speichern

Es ist für den User möglich das Spiel, während er am Zug ist, zu speichern. Das ist entweder aus den Spieloptionen oben links im Spielfenster mit einem Klick auf “Spiel Speichern” möglich oder wird dem User vor einer Aktion, welche den aktuellen Spielfortschritt verwerfen würde, vorgeschlagen.

Dabei öffnet sich ein Fenster in dem Verzeichnis in dem die `Cluedo_Smirnov_ws20.jar` liegt.

Von dort aus kann der User einen beliebigen Ort in dem Dateisystem auswählen, um den Spielstand dort zu speichern. Um Speichern zu können, muss in der Leiste unten im Dialog der Dateiname angegeben werden, welcher die Spielstandsdatei identifiziert. Mit einem Klick auf den “Speichern”-Button unten rechts wird der Speichervorgang angestoßen. Der Spielstand wird im JSON-Format abgespeichert und die notwendige Dateiendung wird, falls nicht schon vorhanden, automatisch ergänzt. Alternativ kann der User sich gegen das Speichern entscheiden und mit einem Klick auf den “Abbrechen”-Button zu dem vorherigen Zustand zurückkehren.

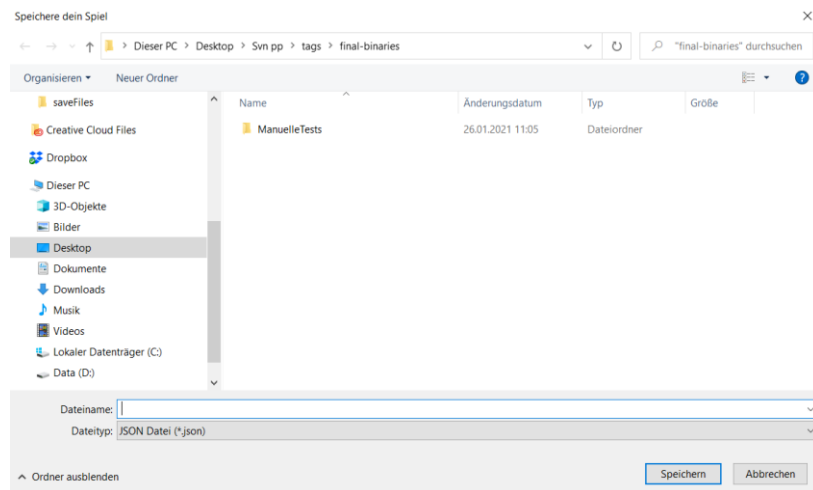


Figure 23 Dialog zur Auswahl des Speicherortes

### 2.3.3.5 Laden

Das Laden eines Spielstandes kann über den “Spiel Laden”- Eintrag in der Liste der Spieloptionen, welche sich oben links im Spielfenster befindet, ausgeführt werden.

Vorerst wird jedoch noch gefragt, ob das aktuell laufende Spiel noch gespeichert werden soll.

Das Laden eines Spielstandes läuft genauso ab wie das Speichern des Spielstandes. Jedoch werden in diesem Dialog nur Dateien angezeigt, welche die notwendige `.json` Dateiendung besitzen. Falls die Spielstandsdatei in dem Ordner nicht zu finden ist, könnte es an einer falschen Dateiendung liegen.

Nach dem Auswählen der zu ladenden Datei kann mit einem Klick auf den “Laden”-Button unten rechts das Laden der ausgewählten Datei bestätigt werden.

Die zu ladende Datei wird auf Korrektheit überprüft und es wird, falls dabei alles gut ging, das gespeicherte Spiel wieder angezeigt. Falls bei dem Laden eine Fehlermeldung auftritt kann im Kapitel “2.4 Fehlermeldungen” nachgelesen werden was diese bedeuten, warum sie auftreten und wie diese zu beheben sind.

Anmerkung: Leider besteht bei dem Laden eines Spielstandes noch der Fehler, dass falls sich in dem Spielstand mehrere Spieler in demselben Raum befinden, diese exakt übereinander dargestellt werden und somit nur ein Spieler in dem Raum angezeigt wird. Das behindert den Spielfluss nicht, da nach einem Zug alles wieder korrekt dargestellt wird.



## 2.4 Fehlermeldungen

Im Laufe des Spielverlaufes können Fehlermeldungen auftreten. Diese können vor allem bei dem Laden und Speichern auftreten. Diese sind meistens auf ein Fehlverhalten des Users zurückzuführen. Im Folgenden ist eine Auflistung mit allen Fehlermeldungen und wie diese zu beheben sind.

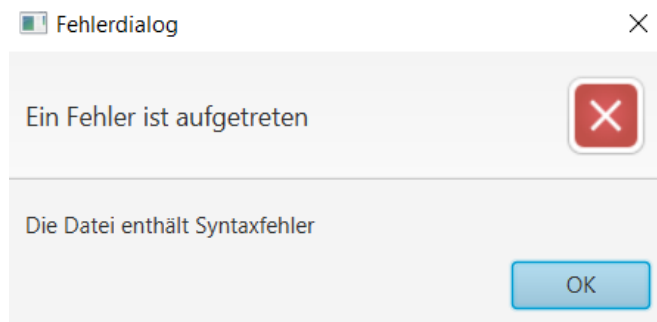


Figure 24 Ein Fehlerdialog

### 2.4.1 Fehler bei Programmstart

Fehlermeldung	Ursache	Behebungsmaßnahme
Die Initialisierungsdatei wurde an der Stelle: ... nicht gefunden.	Die Initialisierungsdatei, welche die Startpositionen, verfügbaren Waffen, Räume und das Spielfeld enthält wurde nicht gefunden.	Das Programm erneut herunterladen und keine Dateien in der .jar verändern.
Beim Laden der Initialisierungsdatei ist ein Fehler aufgetreten.	Beim Laden der Initialisierungsdatei ist ein Ladefehler aufgetreten.	Das Programm neustarten.
Beim Laden der Initialisierungsdatei ist ein Syntaxfehler aufgetreten.	In der angegebenen Initialisierungsdatei sind JSON-Syntaxfehler vorhanden.	Das Programm neu herunterladen und keine Dateien in der .jar verändern.
Das Spiel benötigt diese Informationen zum Starten	Der Dialog, welcher beim Start eines neuen Spiels angezeigt wird, wurde z.B. mit ALT+F4 geschlossen.	Das Spiel erneut öffnen und den Dialog nicht schließen.

### 2.4.1 Fehler während dem Laden oder Speichern eines Spielstandes

Fehlermeldung	Ursache	Behebungsmaßnahme
Die Datei enthält Syntaxfehler.	In der angegebenen Spielstandsdatei sind JSON-Syntaxfehler vorhanden.	Eine andere Datei auswählen oder den JSON-Syntaxfehler beheben. Siehe: <a href="https://www.json.org/json-de.html">https://www.json.org/json-de.html</a>
Ungültiger Wert bei den Notizen über andere Spieler	In der angegebenen Spielstandsdatei ist ein ungültiger Wert bei den Notizen über andere Spieler eingetragen	Einen gültigen Wert eintragen oder eine andere Spielstandsdatei laden.
Ungültiger Wert in eigenen Notizen	In der angegebenen Spielstandsdatei ist ein ungültiger Wert bei den eigenen Notizen eingetragen	Einen gültigen Wert eintragen oder eine andere Spielstandsdatei laden.
In der Spielstandsdatei sind mehr Waffen als bei der Initialisierung des Spiels	In der angegebenen Spielstandsdatei sind mehr verschiedene Waffen vorhanden als in der Initialisierungsdatei.	Die ungültige Waffe aus der Spielstandsdatei entfernen oder eine andere Spielstandsdatei laden.
Der Name der Waffe existiert nicht	In der angegebenen Spielstandsdatei ist ein ungültiger Waffenname vorhanden.	Die ungültige Waffe aus der Spielstandsdatei entfernen oder eine andere Spielstandsdatei laden.
Der Name des Raumes existiert nicht	In der angegebenen Spielstandsdatei ist ein ungültiger Raumname vorhanden.	Den ungültigen Raum aus der Spielstandsdatei entfernen oder eine andere Spielstandsdatei laden.
Der Name der Person existiert nicht	In der angegebenen Spielstandsdatei ist ein ungültiger Personennamen vorhanden.	Den ungültigen Personennamen aus der Spielstandsdatei entfernen oder eine andere Spielstandsdatei laden.
Der Schwierigkeitsgrad der KI existiert nicht	In der angegebenen Spielstandsdatei ist ein ungültiger Wert für den Schwierigkeitsgrad des Computergegners eingetragen.	Den ungültigen Wert für den Schwierigkeitsgrad des Computergegners aus der Spielstandsdatei entfernen oder eine andere Spielstandsdatei laden.
Die Spielerposition stimmt nicht mit dem Raum überein	In der angegebenen Spielstandsdatei ist die Position des Spielers nicht mit dem Raum überein, in dem er sich befinden sollte.	Eine andere Spielstandsdatei laden.
Der Spieler befindet sich nicht in der Mitte des Raums	In der angegebenen Spielstandsdatei steht der Spieler nicht in dem Mittelpunkt des Raumes.	Die Position des Spielers auf den Raummittelpunkt aus der Initialisierungsdatei setzen oder eine andere Spielstandsdatei laden.

Die Spielerposition ist in einer Wand	Die Position des Spielers ist in einer Wand auf dem Spielfeld.	Die Position des Spielers auf eine Position innerhalb des Spielfeldes setzen oder eine andere Spielstandsdatei laden.
Die Länge der eigenen Notizen stimmt nicht	Die Anzahl der Einträge in den eigenen Notizen eines Spielers stimmt nicht mit der Anzahl aus der Initialisierungsdatei überein.	Die Anzahl der Einträge anpassen oder eine andere Spielstandsdatei laden.
Die Länge der Notizen über andere Spieler stimmt nicht	Die Anzahl der Einträge in den Notizen über einen andere Spieler stimmt nicht mit der Anzahl aus der Initialisierungsdatei überein.	Die Anzahl der Einträge anpassen oder eine andere Spielstandsdatei laden.
Fehler beim Laden der eigenen Notizen der Spieler	Das Format der eigenen Notizen eines Spielers stimmt nicht mit dem Speicherformat überein.	Eine andere Spielstandsdatei laden.
Die Datei wurde nicht gefunden	Die zu ladende Spielstandsdatei wurde nicht gefunden.	Das Laden erneut versuchen. Ansonsten eine andere Spielstandsdatei laden.
Fehler beim Schreiben des Spielstandes	Beim Speichern der Spielstandsdatei ist ein unerwarteter Fehler aufgetreten.	Den Spielstand erneut speichern.
Der Spieler wurde in einen Raum gewünscht, befindet sich aber nicht in einem Raum	Laut der Spielstandsdatei wurde ein Spieler in einen Raum "zitiert" befindet sich jedoch nicht in einem Raum.	Eine andere Spielstandsdatei laden.
In der Spielstandsdatei steht null oder es fehlt ein Feld	In der Spielstandsdatei fehlen notwendige Daten.	Eine andere Spielstandsdatei laden.

## 2.4.2 Fehler zur Laufzeit

Fehlermeldung	Ursache	Behebungsmaßnahme
Der Spieler wurde nicht gefunden	Ein Spieler wurde im während der Programmausführung nicht gefunden.	Das Programm neustarten. Falls der Fehler häufiger auftritt den Entwickler kontaktieren.
Die Waffe wurde nicht gefunden	Eine Waffe wurde im während der Programmausführung nicht gefunden.	Das Programm neustarten. Falls der Fehler häufiger auftritt den Entwickler kontaktieren.
Die Person wurde nicht gefunden	Eine Person wurde im während der Programmausführung nicht gefunden.	Das Programm neustarten. Falls der Fehler häufiger auftritt den Entwickler kontaktieren.
Der Name der Karte existiert nicht	Ein Name der Karte existiert nicht.	Das Programm neustarten. Falls der Fehler häufiger auftritt den Entwickler kontaktieren.
Kritischer Fehler! Entschuldigung, das hätte nicht passieren dürfen! Das Spiel wird beendet ~\_(\ツ)\_/	Ein unerwarteter kritischer Fehler ist während der Programmausführung aufgetreten.	Das Programm neustarten. Falls der Fehler häufiger auftritt den Entwickler kontaktieren.

## 3. Programmiererhandbuch für Software-Entwickler

---

### 3.1 Entwicklungskonfiguration

Das Programm wurde mit der Entwicklungsumgebung IntelliJ Community Edition Version 2020.2.1 erstellt. Das Betriebssystem des zur Entwicklung verwendeten Computers ist Windows 10 der Version 2004.

Komponenten	Version
Betriebssystem	Windows 10 der Version 2004
Entwicklungsumgebung	IntelliJ Community Edition Version 2020.2.1
Java Development Kit	OpenJDK Runtime Environment AdoptOpenJDK (build 13.0.2+8)
FXML Editor für das User Interface (GUI)	SceneBuilder Version 11.0.0

## 3.2 Problemanalyse und Realisation

Die Problemanalyse wurde nach den beiden Hauptpackages des Projekts `gui` und `logic` unterteilt. Diese sind nach einigen zentralen Problemen gegliedert, welche während der Realisierung des Programms gelöst werden mussten.

Dazu gehören: Das Spielfeld und die Notizen in der GUI, die Erstellung eines neuen Spiels, das Laden eines Spielstandes, der Implementation der Animation im Spielablauf und die Wegfindung der Computerspieler in der Logik.

### 3.2.1 Das User Interface (die GUI)

#### 3.2.1.1 Das Spielfeld

##### Problemanalyse

Das Spielfeld, auf dem sich die Spielfiguren bewegen, soll sich in seiner Größe anpassen, sobald die Fenstergröße verändert wird. Dabei soll das Seitenverhältnis des Originalbildes beibehalten werden. Die sich auf dem Spielfeld befindlichen Elemente wie Spielfiguren und Waffen sollen dabei ihre Position relativ zum Spielfeld beibehalten.

##### Realisationsanalyse

- Eine Möglichkeit das Spielfeld zu gestalten wäre, über den `ImageView` mithilfe eines `StackPanels` ein transparentes `GridPane` zu legen. Dies hätte den Vorteil, dass die Indexberechnung der geklickten Spielzellen bereits von dem `GridPane` übernommen werden würde. Dagegen spricht die Tatsache, dass das Bild des Spielfeldes nicht genau quadratisch, sondern durch leichte Schräglage der Kamera bei der Aufnahme geschert ist. Dadurch ist der Abstand zwischen dem Bildrand und Anfang des Spielfeldes nicht einheitlich, sodass der Mittelpunkt einer `GridPane`-Zelle nicht überall mit dem Bild übereinstimmt.
- Eine weitere Möglichkeit wäre es kein `Stack`- und `GridPane` zu verwenden, sondern nur ein einfaches `Pane` in welches der `ImageView` Container platziert wurde. Dies hätte den Nachteil, dass die Umrechnung der Klick- in Spielfeldkoordinaten selbst implementiert werden müsste. Ein Vorteil wäre jedoch, dass aufgrund dieser Tatsache die Umrechnung besser an das Spielfeldbild angepasst werden könnte.

### Realisationsbeschreibung

Zunächst wurde der Ansatz mit dem GridPane gewählt, da dieser keine Umrechnung der Koordinaten erfordert und die Ungenauigkeiten nicht sehr hoch sein sollten. Jedoch sind bei der Implementation Probleme aufgetreten, wie z.B. Tatsache, dass sich das Spielfeldbild innerhalb des ImageView Containers nicht sofort an die Größe des StackPanes angepasst hat, sondern erst mit einer leichten Verzögerung. Außerdem kam es nach einer Veränderung der Fenstergröße vor, dass das Bild leicht geflimmert hat, als wäre die Breite oder Höhe immer wieder von einem, auf den anderen Wert, gesprungen.

Aufgrund dieser Probleme wurde der Ansatz ohne GridPane gewählt. Um im Spielverlauf die aktuelle Breite und Höhe des Bildes zu berechnen, musste im Konstruktor, ein `NumberBinding` verwendet werden, welches auf die folgende Art berechnet wird:

```
ratio_original = W_original / H_original  
→ W_curr = H_parent * ratio_original  
→ H_curr = W_parent / ratio_original
```

Um ein Element auf das Spielfeldbild zu platzieren muss ein prozentualer Wert für die Breite und Höhe des Elementes innerhalb des Bildes berechnet werden. Dieser Wert berechnet sich für das Spielfeld über die Position des Elements in Spielfeldkoordinaten (beginnend bei 0,0 in der oberen linken Ecke der Kacheln).

```
offsetPercentage + (pos + 0.5) * cellPercentage
```

Dabei ist `offsetPercentage` der prozentuale Abstand von dem Bildrand zu der ersten Spielfeldzelle, `(pos + 0.5)` der Mittelpunkt der Spielfeldzelle in Spielfeldkoordinaten und `cellPercentage` die prozentuale Breite oder Höhe einer Spielfeldzelle. Die Werte für den Abstand von dem Bildrand zu der ersten Spielfeldzelle, wurden für jede Seite pixelgenau berechnet, um eine möglichst geringe Quote falscher Klicks zu erzielen.

Um nun die Position auf dem Spielfeld zu für eine bestimmte Spielfeldzelle zu berechnen, muss nur noch prozentualen Wert mit der aktuellen Höhe/Breite des Spielfeldbildes multipliziert werden. Somit erhält man die x- und y-Koordinate auf, welche ein Element gesetzt werden kann damit es, gegeben der aktuellen Größe des Bildes, in der Mitte eines Spielfeldelementes erscheint.

Dieses Vorgehen wird in den Methoden `setLabelPosition` und `setCirclePosition` in der Klasse `JavaFXGUI` genutzt.

### 3.2.1.2 Notizen für den User

#### Problemanalyse

Es soll dem User die Möglichkeit gegeben werden, während des Spiels Notizen über die eigenen Karten und die der anderen Spieler zu machen. Dabei unterscheiden sich die eigenen Notizen zu denen der anderen Spieler.

#### Realisationsanalyse

- Eine Möglichkeit die Notizen zu realisieren wäre ein Table View zu nutzen und in diesen Combo Box Elemente einzufügen welche, je nachdem um welche Notizen Art es sich handelt, dem User die notwendigen Einträge zur Auswahl anbieten.  
Ein Nachteil dieses Ansatzes wäre jedoch, dass zuerst die Zeile bzw. Spalte angeklickt werden muss, um erst danach mit den sich darin befindlichen Elementen interagieren zu können. Das würde für den User bedeuten, dass dieser drei Mausklicks machen müsste, um eine Notiz zu bearbeiten. Den ersten für die Tabelle, den zweiten für die Combo Box und den dritten für die eigentliche Auswahl der Notiz.
- Eine weitere Möglichkeit wäre es, statt einem Table View ein GridPane zu verwenden, um das Problem mit dem extra Klick zu umgehen.
- Bei diesen Ansätzen wäre es von Vorteil die Struktur komplett dynamisch aus den Initialisierungsdaten des Spiels zu generieren. Somit wäre es sehr einfach möglich, im Nachhinein mehr Spielfiguren oder Karten zu unterstützen.

#### Realisationsbeschreibung

Die Möglichkeit ein Table View zu nutzen, erschien aus UX-Gründen sehr unpraktisch, sodass dieser Ansatz verworfen wurde.

Somit wurde für die Notizen in SceneBuilder ein GridPane verwendet, welches in die rechte VBox platziert wurde. Die erste Zeile und Spalte wurden mit Labels befüllt, um diese Beschriften zu können. Anschließend wurde der Rest mit Comboboxen befüllt. Zwischen den verschiedenen Kartentypen wurde eine Zeile freigelassen, um eine klare Trennung für den User vorzunehmen.

Auf den Ansatz die Notizenstruktur komplett dynamisch aufzubauen wurde aus Zeitgründen komplett verzichtet. Jedoch wurde die Befüllung der Combo Boxen und Labels nicht in der FXML-Datei vorgenommen, da deren Inhalt dynamisch aus der Initialisierungsdatei geladen werden soll. Somit ist der Inhalt, aber nicht die Anzahl der Notizen in der GUI dynamisch über die Initialisierungsdatei änderbar.



### 3.2.2 Die Spiellogik:

#### 3.2.2.1 Ein Spiel erstellen

##### Problemanalyse

Erstellung eines Ausgangszustandes auf Basis dessen das Spiel gestartet werden kann. Bei Programmstart soll zunächst abgefragt werden, wie viele und welche KI-Stärken die Mitspieler haben sollen. Danach soll ein Spiel generiert werden, bei dem der User als erster mit der Figur von "Fräulein Ming" am Zug ist.

##### Realisationsanalyse

Es bedarf eines Konstruktors, welcher eine Instanz der `GameLogic` eines beliebigen Zustandes erstellt. Dafür werden diesem alle notwendigen Objekte übergeben, um damit die Attribute der `GameLogic` zu befüllen oder weitere, davon abgeleitete, zu initialisieren.

- Eine Möglichkeit wäre es die Daten für das Spiel nicht aus der Initialisierungsdatei zu laden, sondern all diese Informationen als Enums im Spiel abzulegen. Dies hätte jedoch den Nachteil, dass keine anderen Spieler, Waffen, Räume oder Spielfelder mehr hinzugefügt werden könnten.
- Eine andere Möglichkeit wäre es die Initialisierungsdatei zu nutzen, um all diese Daten dynamisch bei dem Spielstart zu laden. Dies hätten den Vorteil, dass nur diese eine Datei angepasst werden müsste, um das Spiel zu verändern. Ein Nachteil wäre, dass die `GameLogic` allen anderen Klassen übergeben werden müsste, falls diese Spieler, Waffen, Räume oder das Spielfeld nutzen würden.

##### Realisationsbeschreibung

Aufgrund der erhöhten Erweiterbarkeit wurde der Ansatz mit der Initialisierungsdatei gewählt. Diese enthielt zunächst folgenden Inhalt im JSON Format:

- Namen und Startpositionen der der möglichen Spielfiguren.
- Raumnamen, Position des Raummittelpunktes, Positionen der Türen.
- Die Namen der Waffen im Spiel.

Da beschlossen wurde, dass das Spielfeld und die Ziele der Geheimgänge ebenfalls änderbar sein sollten, wurden der Initialisierungsdatei folgende Daten hinzugefügt:

- Die Breite, Höhe sowie die einzelnen Zellen des Spielfeldes als String.
- Die Räume wurden um einen Zielraum erweitert. Dieser ist jedoch optional, je nachdem, ob ein Geheimgang vorhanden sein soll, oder nicht.

Diese Daten wurden bei dem Spielstart geladen, jedoch mussten diese noch für den Konstruktor angepasst werden.

Dafür ist folgende Methode in der `GameLogic` zuständig:

```
public static GameLogic createInitialGameLogicFromJSON(InitialGameDataJSON  
initGameDataJSON, int playerAmount, AIDifficulty[] difficulties)
```

Die Überlegungen hinter dieser Methode sahen folgendermaßen aus:

- Zunächst wäre es naheliegend diese Methode zu einem Konstruktor zu ändern, welcher am Ende der Verarbeitung der `InitialGameDataJSON` Klasse den Konstruktor der `GameLogic` aufruft. Jedoch ist der Aufruf eines Konstruktors innerhalb eines anderen Konstruktor mit `this()` nur als erste Anweisung möglich. Daher käme diese Möglichkeit nicht in Frage.
- Um diese Einschränkung zu umgehen, wäre es möglich dieselben Zuweisungen wie in dem Konstruktor der `GameLogic` vorzunehmen. Dies wäre jedoch Codedopplung und solle aus offensichtlichen Gründen vermieden werden.

Somit erschien die Lösung den initialen Spielzustand herzustellen, indem eine statische Methode die Initialisierungsdatei entgegennimmt und auf Basis der dort angegebenen Informationen die notwendigen Objekte erzeugt, mit der die `GameLogic` konstruiert werden kann, als beste Möglichkeit.

Mit dem gewählten Ansatz wäre es möglich, eine komplett neue Version des Programms zu erzeugen, indem nur die Initialisierungsdatei, die Konstanten in der `JavaFXGUI` und das Spielfeldbild angepasst werden müssten.

### 3.2.2.2 Einen Spielstand laden und speichern

#### Problemanalyse

Eine weitere Anforderung an das Programm war es, Spielstände im JSON-Format laden und speichern zu können. Bei dem Ladevorgang bedarf es zusätzlicher Validierungslogik, da die zu ladende Datei in einem fehlerhaften Format vorliegen kann. Aus den Daten der geladenen Datei soll der Spielzustand wiederhergestellt werden können.

Die Konvertierung von interner Repräsentation des Spiels in die Spielstandsklasse muss für das Lesen und Schreiben in beide Richtungen funktionieren.

#### Realisationsanalyse

Da ein Beispielspielstand im JSON-Format vorgegeben war, muss dieser in eine Java-Klasse überführt werden können. Da die Spielstandsdatei aus mehreren JSON-Objekten besteht, muss für jedes dieser Objekte eine eigene Java-Klasse angelegt werden, aus denen wiederum die "Hauptspielstandklasse" besteht.

- Eine Möglichkeit das Konvertieren von "interner" zu "externer" Spielzustandsrepräsentation umzusetzen, ist es in jeder relevanten Klasse, eine Methode zu implementieren, welche sich um die Konvertierung der jeweiligen Klasse kümmert.  
Ein Nachteil davon wäre eine Einschränkung der Modularisierung. Das bedeutet, dass falls das Speicherformat angepasst werden sollte, ebenfalls jede dieser Klassen angepasst werden müssten. Außerdem wäre die nachträgliche Unterstützung eines neuen Speicherformates ebenfalls mit viel Aufwand verbunden.
- Eine andere Möglichkeit wäre es eine Klasse zu schreiben, welche aus statischen Methoden besteht und für die gesamte Konvertierung zuständig ist. Somit würde nur diese eine Klasse das Spielstandformat kennen und alle potenziellen Änderungen betreffen nur diese eine Klasse.

Für die Realisierung des Ladens eines Spielstandes kommen ebenfalls mehrere Möglichkeiten in Frage, da bei dem Ladevorgang Fehler auftreten können und diese unterschiedlich behandelt werden könnten.

- Eine Möglichkeit wäre es, die geladenen Daten direkt in die `GameLogic` zu schreiben. Falls zu irgendeinem Zeitpunkt ein Fehler aufträte, müsste die Instanz verworfen werden. Danach müsste ein neues Spiel gestartet werden, da die `GameLogic` sich nicht mehr in einem validen Zustand befinden würde. Dies wäre für den User nicht sehr benutzerfreundlich.
- Eine weitere Möglichkeit wäre es bei dem Laden immer eine neue Instanz der `GameLogic` zu erzeugen. Ein Nachteil davon wäre, dass die Initialdaten des Spiels immer wieder neu transformiert und in die `GameLogic` geschrieben werden müssten.
- Eine andere Möglichkeit wäre es eine Klasse zu schreiben, welche die Daten aus der Spielstandsdatei vorerst aufnimmt und erst nachdem bei dem Ladevorgang die gesamten Daten validiert wurden, den Zustand der `GameLogic` damit beschreibt. Somit würden die Initialdaten erhalten bleiben und es würden in der `GameLogic` keine fehlerhaften Zustände durch ungültige Daten auftreten. Zusätzlich wäre ein Fehler bei dem Ladevorgang kein Problem, da das vorherige Spiel unverändert weiterlaufen könnte.

### Realisationsbeschreibung

Es wurde sich aufgrund der besseren Modularisierung und der erhöhten Wartbarkeit des Programms für die Variante der Klasse mit den statischen Konvertierungsmethoden entschieden. Dabei handelt es sich um die Klasse `GameDataConverter`. Sie kümmert sich um die Konvertierung und die Validierung der Spielstandsdateien. Dafür sind die Methoden `convertToGameDataJSON` und `convertToLoadedGameLogic` zuständig. Die erste kümmert sich um das Speichern und die zweite um das Laden und die Validierung. Um das Problem mit den fehlerhaften Spielstandsdateien zu behandeln, wurde die Signatur der Methode `convertToLoadedGameLogic` folgendermaßen gewählt:

```
public static LoadedGameLogic convertToLoadedGameLogic(GameDataJSON loadedGame,
GameLogic logic) throws CluedoException
```

Dabei ist der Typ des Rückgabewertes zu beachten. Es handelt sich um eine Instanz der Klasse `LoadedGameLogic`. Diese Klasse enthält u.A folgende Daten welche, um einen neuen Spielzustand zu erzeugen notwendig sind:

```
private final Room[] weaponInRooms
```

Die Zuordnung der Waffen zu den Räumen.

```
private final Player[] players
```

Die Spieler in diesem Spielstand mit all ihren Daten.

```
private final CardTriple envelope
```

Die Lösung des Spiels.

Falls die Validierung fehlschlägt, wird eine `CluedoException` geworfen und behandelt. Falls alles in Ordnung ist, wird auf der Instanz der Klasse `LoadedGameLogic` die Methode `commit` mit der aktuellen `GameLogic` als Parameter aufgerufen. Die Methode `commit` schreibt nun den Inhalt ihrer Attribute in die `GameLogic` und lädt somit einen neuen Spielstand.

Anmerkung: Die Initialdatei musste, um das Laden des Beispielspielstandes zu ermöglichen angepasst werden, da der Raummittelpunkt der Küche sich in den Dateien unterschieden hat.

### 3.2.2.3 Die Implementation der Animation im Spielablauf

#### Problemanalyse

Im Spielverlauf sollen Spielfiguren und Waffen während ihrer Bewegung linear zwischen Start- und Endpunkt animiert werden. Dafür ist eine Unterbrechung der Spiellogik notwendig, da diese Animationen in den Spielablauf integriert sind und je nach Eingabe des Users unterschiedlich sein können.

#### Realisationsanalyse

- Eine Möglichkeit die Animationen zu realisieren wäre es, alle Züge der Spieler zu speichern und diese alle hintereinander zu animieren. Jedoch ist dies aufgrund der Tatsache nicht umsetzbar, dass teilweise zwischen den Animationen auf Input des Users gewartet werden muss.
- Eine weitere Möglichkeit wäre es den Spielablauf in der `GameLogic` in kleine Methoden zu teilen, welche die notwendigen Operationen, je nach Spielzustand und Userinput ausführen. Die Auswahl welche Methode aufgerufen werden soll, würde in der Klasse `JavaFXGUI` geschehen. Die `GameLogic` müsste speichern, in welchem Zustand sich das Spiel gerade befindet, um nach der Animation zu wissen, wo weitergemacht werden soll. Dafür muss der vorherige Zustand aus den gespeicherten Daten wiederhergestellt werden. Ein klarer Nachteil dieses Ansatzes wäre es, dass der Ablauf des Programms in sehr viele unübersichtliche Teile zerlegt werden müsste, was eine erhöhte Komplexität der Spiellogik bedeuten würde.
- Denkbar wäre ebenfalls der Einsatz von Koroutinen. Diese würden ihren internen Zustand bis zu ihrem nächsten Aufruf beibehalten. Bei dem nächsten Aufruf dieser Koroutine, würde diese dort weitermachen, wo sie vorher aufgehört hat. Dieser Ansatz hätte den Vorteil, dass der Programmcode der Spiellogik als eine große Schleife geschrieben werden könnte, welche wiederum Methoden aufruft, die je nach Spielsituation an verschiedenen Stellen pausieren. Somit könnte die Spiellogik sehr übersichtlich an einer Stelle implementiert werden.

#### Realisationsbeschreibung

Da das Problem der Animation im Projektverlauf sehr spät angegangen wurde und die Spiellogik zu diesem Zeitpunkt bereits als große Methode fertiggestellt war, wurde sich dagegen entschieden die `makeMove` Methode aufzuteilen. Dieses Vorgehen wäre nötig gewesen, um den Ansatz mit den verschiedenen Spielzuständen in der `GameLogic` zu realisieren. Es bestand die Sorge, bei diesem Prozess viele neue Fehler einzuführen und diese in der verbleibenden Bearbeitungszeit nicht mehr beheben zu können. Daher wurde die Alternative der Koroutinen näher betrachtet, da diese den Vorteil hätte den Code nicht mehr anpassen zu müssen. Da Java jedoch keine Koroutinen als Sprachbestandteil anbietet, musste eine ähnliche Funktionalität selbst implementiert werden, da keine zusätzlichen Bibliotheken genutzt werden sollten. Dabei wurde diese <sup>4</sup> Seite gefunden, welche Thread-basierte Koroutinen in Java vorschlägt. Da der Anwendungsfall der Animationen keine hohe Leistung benötigt, ist der Nachteil des erhöhten Leistungs- und Speicheraufwandes für die benötigten Threads akzeptabel. Da blockierende Operationen in dem JavaFX Application Thread dazu führen würden, dass währenddessen nicht mehr mit der GUI interagiert werden könnte, muss diese Operation außerhalb davon stattfinden. Um Elemente der GUI manipulieren zu können muss diese Manipulation jedoch innerhalb des JavaFX Application Threads stattfinden. Somit besteht die Notwendigkeit den JavaFX Application Thread verlassen und zu einem späteren Zeitpunkt wieder betreten zu können.

Dafür dienen die Klassen `AsyncJavaFXGUI` und `AsyncGameLogic`. Sie implementieren dieselben Methoden wie ihre "normalen" Gegenstücke, wechseln dabei jedoch den Threadkontext. Um den

---

<sup>4</sup> <https://stackoverflow.com/questions/2846664/implementing-coroutines-in-java>

Kontextwechsel in den JavaFX Application Thread zu realisieren wird der `GameLogic` die `AsyncJavaFXGUI` übergeben welche ebenfalls das `GUIConnector` Interface implementiert. Diese führt GUI-Operationen asynchron in dem JavaFX Application Thread aus und wartet auf das Ende der GUI-Operation.

Aufrufe aus der `JavaFXGUI` an die `GameLogic`, welche wiederum die GUI aufrufen (um z.B in `makeMove` das Bewegen der Spielfigur zu animieren), müssen daher den JavaFX Application Thread verlassen, damit die GUI, während die Logik auf die Animation wartet, nicht zu blockieren. Aus diesem Grund wird `makeMove` auf der `AsyncGameLogic` aufgerufen, wenn der User das Spielfeld anklickt.

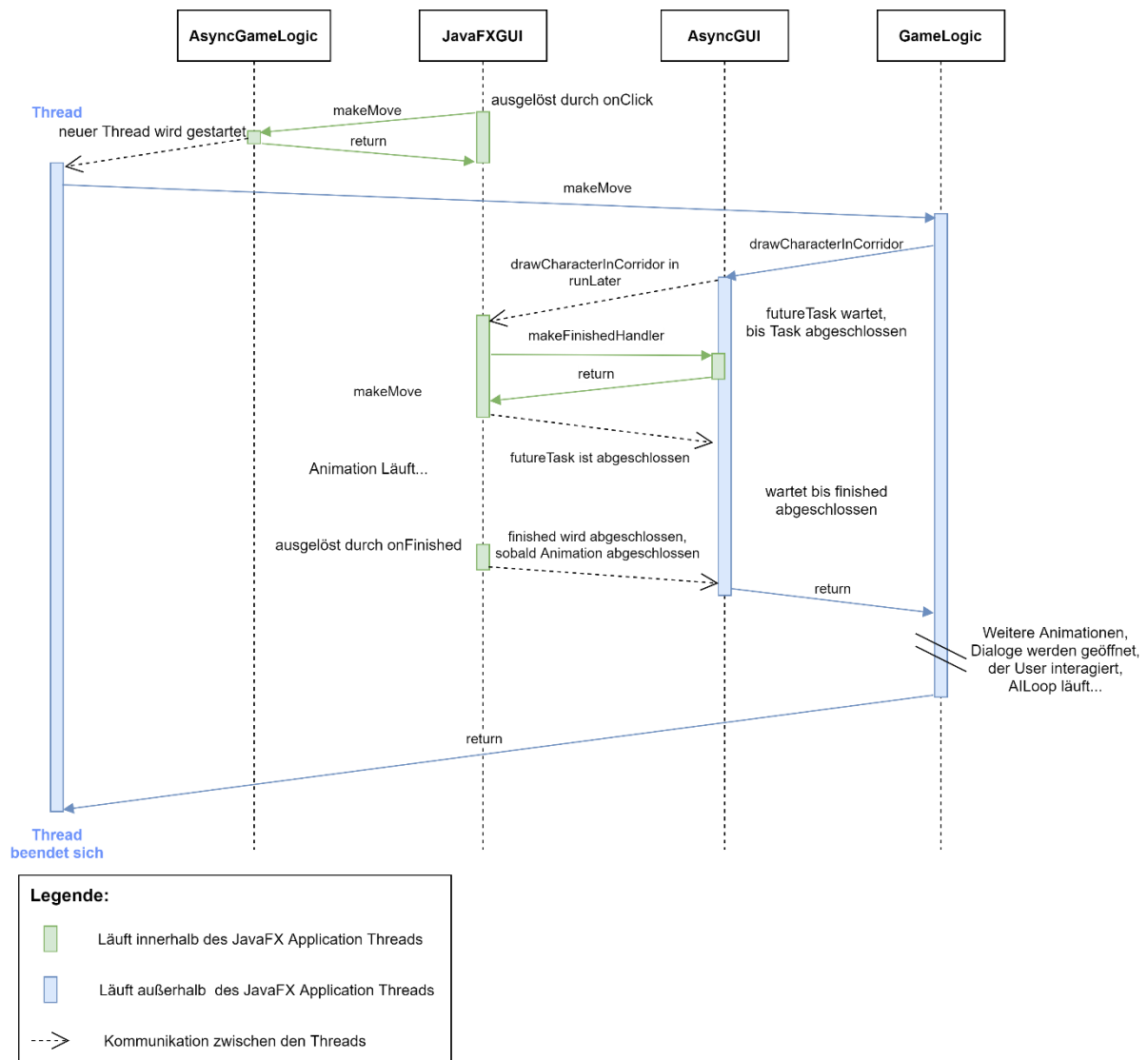


Figure 25 Ablauf bei Aufruf der Methode `makeMove` (erstellt mit Draw.io)

Diese Grafik dient der Visualisierung des Ablaufes einer Animation am Beispiel der `makeMove` Methode.

### 3.2.2.4 Der Wegfindungsalgorithmus der Computerspieler

#### Problemanalyse

Zu lösen ist das Problem den kürzesten Weg von einem Feld zu einer Menge an Feldern auf dem Spielfeld zu finden. Dabei ist zu beachten, dass Spieler den Weg versperren und somit eine ausschließlich statische Struktur für die Repräsentation der Spielfeldsituation nicht in Frage kommt. Außerdem ist zu beachten, dass je nach Computergegnerstärke die Geheimgänge berücksichtigt werden sollen. Eine Besonderheit, welche zu beachten ist, dass Türen pro berechnetem Zug nur einmal passiert werden dürfen.

#### Realisationsanalyse

- Eine Möglichkeit einen Weg von der aktuellen zu der gewünschten Position zu suchen, wäre die Breitensuche<sup>5</sup> (*breadth-first search*, *BFS*) bei der immer nur die Nachbarn vom Ausgangspunkt betrachtet werden, bis das Ziel entweder gefunden oder keine Felder mehr zu besuchen sind. Pro Gefundenem Feld speichert man zusätzlich noch von welchem Feld dieses erreicht wurde umso im Nachhinein rekonstruieren zu können, wie genau der Weg vom Ausgangspunkt zum Ziel aussieht. Dafür geht man alle Felder "rückwärts" vom Ziel zum Ursprung ausgehend von deren Herkunft ab und speichert diese Folge von Feldern in einer Liste. Bei einem Spielfeld, bei dem ein Schritt die gleiche Anzahl an Ressourcen kostet, wird so der kürzeste verfügbare Weg berechnet. Der Nachteil hierbei ist jedoch, dass alle besuchten Felder gespeichert werden müssen und im ungünstigsten Fall alle möglichen Pfade zu allen möglichen Feldern betrachtet werden müssen.
- Eine weitere Möglichkeit wäre die Suche nach dem kürzesten Weg mit dem Dijkstra-Algorithmus<sup>6</sup>. Dabei werden zusätzlich die Kosten eines Schrittes von einem zu dem nächsten Feld mitbetrachtet, indem die Kanten gewichtet werden (nur positive Werte). Die bereits besuchten Felder wird nach den Kosten vom Startpunkt zu dem jeweiligen Feld aufsteigend sortiert vorgehalten. Die Grundidee hierbei ist, dass immer derjenigen Weg gefolgt wird, der den kürzesten Streckenabschnitt verspricht. Anderen Wegen wird erst gefolgt, wenn keine kürzeren Strecken mehr vorhanden sind. Somit wird garantiert, dass falls das Ziel gefunden wurde, dieses auf dem kürzesten Pfad erreicht wurde.

Bei den genannten Verfahren wird der Weg in alle Richtungen erschlossen, bis das Ziel gefunden wurde. Ist das Ziel aber hinter einem Abschnitt von Feldern, welche nur mit hohen Kosten zu erreichen sind, dann wird der Dijkstra-Algorithmus zuerst alle anderen Richtungen priorisieren und erschließen, bevor dieser sich weiter in die Richtung des Ziels vorarbeitet. Das führt bei ungünstigen Graphkonstellationen, welche das Spielfeld repräsentieren zu unnötig hohen Laufzeiten.

- Um das Problem zu umgehen kann eine Schätzfunktion verwendet werden, um zusätzlich dafür zu sorgen, dass sich der Algorithmus ungefähr in die gewünschte Zielrichtung bewegt. Unabhängig davon wie hoch die umliegenden Kosten der Wege von dem aktuellen Feld sind. Das sorgt für eine Zielgerichtetheit und somit schnellere Erschließung des Spielfeldes. Jedoch ist eine passende Schätzfunktion für die konkrete Problemstellung nicht immer trivial. In dem Fall der Wegfindung auf einem 2-D Schachbrett ähnlichen Spielfeld würde sich die Distanz zwischen dem Start- und Zielpunkt anbieten, welche sich mit der absoluten Differenz der Einzelkoordinaten berechnen ließe.

---

<sup>5</sup> [https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search)

<sup>6</sup> [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

$$d(s,d) = |s_x - d_x| + |s_y - d_y|$$

Einen solchen Ansatz verfolgt der A\*-Suchalgorithmus<sup>7</sup>. Bei diesem werden die noch abzuarbeitenden Felder nicht nur nach den Kosten der Wege, welche dorthin führen sortiert, sondern nach der Summe zwischen Kosten zu dem aktuell betrachteten Feld und dem Ergebnis der Schätzfunktion für das Feld. Das führt dazu, dass keine Zeit auf wenig versprechende Felder verschwendet wird, da sich diese eher von dem Ziel wegbewegen. Somit wird die Wahrscheinlichkeit erhöht, um das Ziel früher finden zu können.

### Realisationsbeschreibung

Da das Spielfeld eine überschaubare Größe von 24\*25 Feldern hat und selbst bei einer Variation davon nicht absehbar ist, dass sich die Größe signifikant erhöht, wurde für die Wegfindung der Computergegner der Ansatz der Breitensuche gewählt.

Das lag unter anderem daran, dass in diesem Spiel alle Schritte immer gleich viele Ressourcen (Schritte) kosten und daher der Vorteil von Dijkstra oder sogar A\* keinen wirklichen Mehrwert hätten. Außerdem würden diese den Grad der Komplexität erhöhen. Das hätte den Nachteil eines höheren Aufwandes bei der Fehlerbehebung und Wartung des Codes. Der Vorteil des geringeren Berechnungsaufwandes für ein Spielfeld dieser Größe kann aufgrund der genannten Nachteile vernachlässigt werden.

Diese Funktion wurde in der Klasse `GameLogic` mit der Methode `getShortestPath` implementiert, da dafür alle Positionen der Spielfiguren als auch das Spielfeld selbst benötigt werden. Diese bekommt eine `Position` übergeben, welche den Startpunkt der Suche darstellt. Außerdem wird ein `Set<Position>` übergeben, welches die Zielpositionen darstellt, zu denen der kürzeste Weg gefunden werden soll. Diese Menge sollte nicht leer sein, da die Methode sonst alle Felder in dem Spielfeld absucht, um erst danach zu erkennen, dass kein Weg gefunden werden konnte. Daher wird in diesem Fall sofort `null` zurückgegeben. `null` beschreibt hierbei die Abwesenheit eines Weges und muss, von den Methoden die `getShortestPath` aufrufen, berücksichtigt werden. Der Rückgabewert von `getShortestPath` ist `List<Position>` welche den kürzesten gefundenen Weg zu einem der Positionen in der übergeben Menge darstellt. Dieser beginnt mit dem ersten Schritt und endet bei der Zielposition.

```
public List<Position> getShortestPath(Position startPosition, Set<Position>
destinations)
```

Anmerkung zu dem Verhalten der Zielsuche und Wegfindung der "dummen" Spielstärke:

Falls kein Weg zu dem nächsten offenen Raum vorhanden ist, da entweder nur noch ein Raum als Tatraum in Frage kommt und sich der Computerspieler bereits in diesem befindet, oder alle Räume die noch in Frage kommen durch andere Spielfiguren versperrt wurden, wird ein zufälliger Raum (außer dem Raum in dem sich die Spielfigur eventuell befindet) als Ziel festgelegt. Falls dieser Raum ebenfalls komplett versperrt ist, setzt der Computerspieler diesen Zug aus. Diese Lösung wurde gewählt, da dieser Fall im normalen Spielverlauf mit sehr geringer Wahrscheinlichkeit auftritt und das Spiel trotzdem sinnvoll weitergehen kann.

---

<sup>7</sup> [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)



### 3.3 Programmorganisationsplan

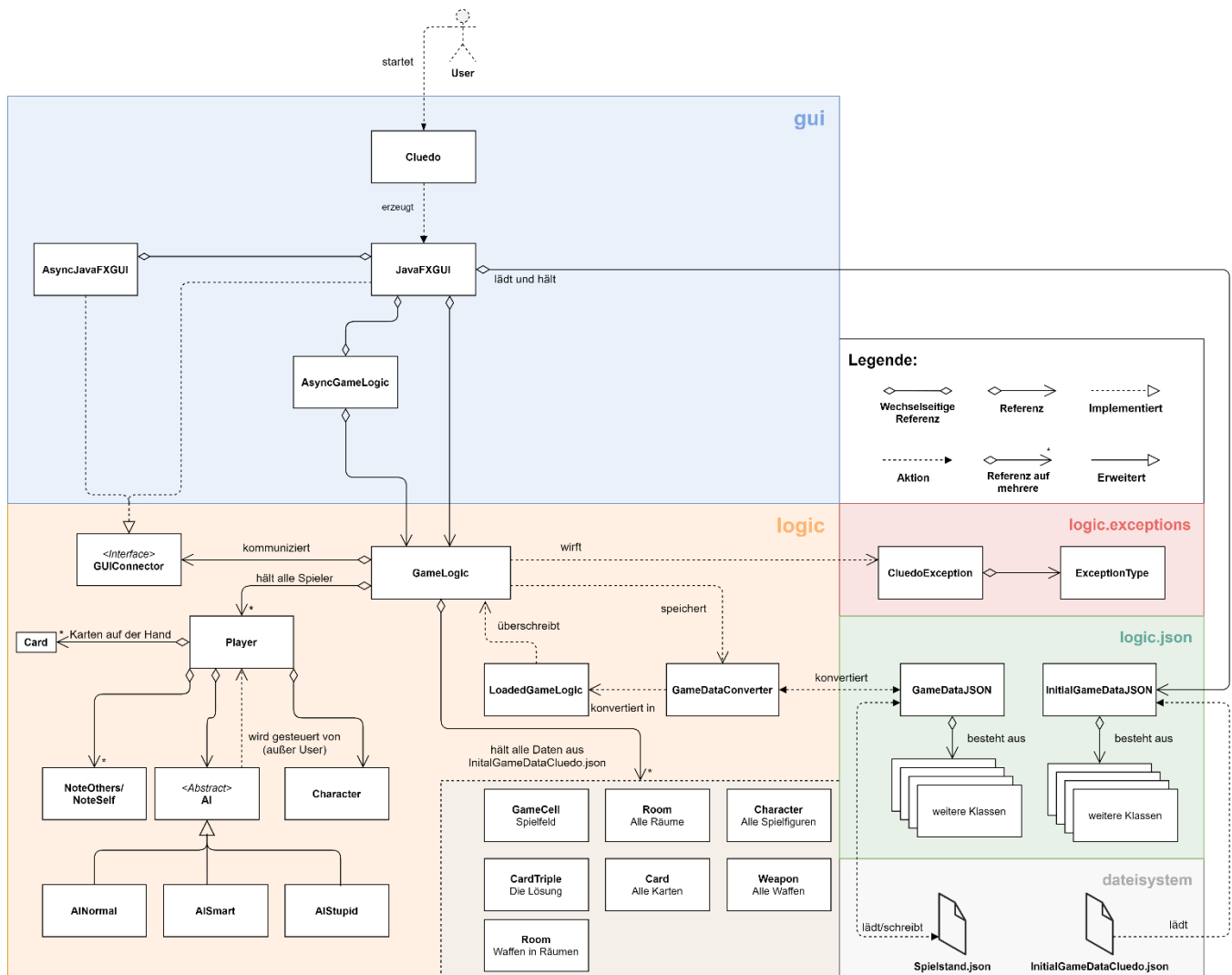


Figure 26 Programmaufbau (erstellt mit Draw.io)

Diese Grafik stellt den Programmaufbau, die Zusammenhänge zwischen den wichtigsten Klassen sowie einigen Aktionen dar, welche während der Programmausführung ablaufen. Dabei wurden die Klassen u.A. nach Packages im Programm gegliedert und farblich voneinander getrennt. Zusätzlich wird die Festplatte des Users als “dateisystem” dargestellt, da während des Programmablaufs einige Dateien relevant sind.

Zur Vereinfachung der Grafik wurden einige Abstraktionen eingesetzt. Diese bestehen u.A. aus der Zusammenfassung von Klassen zu einem Bündel wie im Package `logic` oder `logic.json`. Die Referenzpfeile stellen die wichtigsten Attribute der Klassen dar. Die Pfeilart “Referenz auf mehrere” stellt eine Ansammlung von Referenzen im Attribut der Klasse dar.

## 3.4 Beschreibung grundlegender Klassen

### 3.4.1 Die Klassen der GUI

#### 3.4.1.1 Cluedo

Cluedo ist die Einstiegsklasse für das JavaFX Programm. Sie erweitert `javafx.application.Application` und implementiert die `main` und `start` Methoden, welche das gesamte Programm starten und initialisieren.

Sie enthält außerdem das Folgende Interface:

```
public interface Runnable {  
    void run() throws CluedoException;  
}
```

Ein eigenes `Runnable` Interface, welches auch `CluedoExceptions` werfen kann. Ist für die Ausführung der Methoden, welche außerhalb des JavaFX Application Threads stattfinden, relevant.

#### 3.4.1.2 FXMLDocumentController

`FXMLDocumentController` ist der Controller für das Hauptspielfenster.

Diese Klasse dient nur dazu die `JavaFXGUI` zu erstellen und alle Aufrufe der Eventhandler-Funktionen an diese weiterzuleiten.

Außerdem enthält diese Klasse alle JavaFX Elemente, welche in der `FXMLDocument.fxml` eine ID zugewiesen bekommen haben.

#### Methode

```
public void initializeWithStage(Stage stage)
```

Initialisiert die GUI, jedoch bereits mit Stage, was das Schließen des Programms während der Instanziierung wesentlich einfacher macht, da kein Exception-Handling notwendig ist, sondern `stage.close()` benutzt werden kann.

### 3.4.1.3 JavaFXGUI

Diese Klasse ist diejenige, die den gesamten GUI-Code für das Programm enthält. Sie implementiert das `GUIConnector` Interface, um aus der Logik heraus aufgerufen werden zu können.

Jeglicher Text, der auf der GUI erscheint, wird hier festgelegt. Somit ist diese Klasse allein für die Sprache zuständig.

#### Attribute

Die Klasse enthält einige Konstanten für das Spielfeld, sowie für das Layout der Notizen. Außerdem sind hier alle JavaFX Elemente enthalten, welche von dem `FXMLDocumentController` bei der Initialisierung dieser Klasse übergeben wurden.

```
private InitialGameDataJSON initialGameDataJSON
```

Die Initialisierungsdaten für ein neues Spiel, welche aus der Datei `InitialGameDataCluedo.json` geladen wird.

```
private GameLogic logic
```

Die Logik des Spiels dient dazu, um nach der Initialisierung auf die notwendigen Elemente zuzugreifen, um diese darstellen zu können.

```
private AsyncGameLogic asyncLogic
```

Wird für die Aufrufe der Logik verwendet welche blockieren müssen.

```
private AsyncJavaFXGUI asyncGui
```

Wird bei der Initialisierung verwendet und um das Ende einer Animation signalisieren zu können.

#### Methoden

```
private void startGame() throws CluedoException
```

Startet ein neues Spiel. Zeigt den Dialog, welcher Spieleranzahl und die Schwierigkeitsstufen der Computergegner abfragt. Außerdem werden hier die Async Klassen, als auch die eigentliche Spiellogik erstellt.

```
public static InitialGameDataJSON loadInitialGameData(String path) throws  
CluedoException
```

Diese Methode lädt bei Initialisierung der Klasse die Initialdaten aus der `InitialGameDataCluedo.json` und legt diese in dem Attribut dieser Klasse ab.

```
public void handleMouseClicked(MouseEvent mouseEvent)
```

Kümmert sich um einen Mausklick auf das Spielfeld. Die Klickkoordinaten werden in Spielfeldkoordinaten umgerechnet und es wird ein kompletter Spielzug angestoßen, falls der User auf ein valides Feld ausgewählt hat.

```
public void handleException(CluedoException e)
```

Kümmert sich um die Exceptions, die während des Spielverlaufs auftreten können. Generiert eine aussagekräftige Fehlermeldung anhand des Typen und beendet bei einem kritischen Fehler das Spiel.

```
public void handleOwnSuspicionResult(Player[] allPlayers, Card[] shownByAI,  
CardTriple suspicion) throws CluedoException
```

Kümmert sich um das Fenster, welches dem User anzeigt, welche Karten ihm von welchem Spieler auf einen Verdacht hin gezeigt wurden.

```
public void handleOthersSuspicionResult(Player[] allPlayers, Player  
currentPlayer, Card[] shownCards, CardTriple suspicion) throws CluedoException
```

Kümmert sich um die Anzeige der Reaktionen der anderen Spieler, falls ein Computergegner eine Verdächtigung geäußert hat.

```
public void handleAISuspicion(String playerName, CardTriple suspicion)
```

Kümmert sich um die Verdächtigung eines KI-Spielers und zeigt diese an.

```
public Card handleShowCard(CardTriple suspicion, CardTriple possibleCardsToShow)
```

Kümmert sich um den Dialog, bei dem der User auf eine Verdächtigung hin eine Karte zeigen muss. Dafür wird die Klasse `ShowCardsDialog` benutzt, auf die später näher eingegangen wird.

```
public CardTriple handleExpressSuspicion(Card enteredRoom)
```

Kümmert sich darum, dass der User in einem Dialog seine Verdächtigung aussprechen kann. Der übergebene Raum ist dabei vorausgewählt.

```
public boolean handleSaveGame()
```

Kümmert sich um den Dialog, der den User einen Speicherort wählen lässt. Außerdem wird hier das Speichern in der Logik angestoßen.

```
public void handleLoadGame()
```

Kümmert sich um den Dialog, der den User eine zu ladende Datei wählen lässt. Außerdem wird hier das Laden in der Logik angestoßen.

#### 3.4.1.4 AsyncGameLogic

Diese Klasse ist dafür zuständig, Aufrufe an die Logik zu tätigen, welche blockieren und nicht im JavaFX Application Thread laufen dürfen, da sonst die GUI nicht interagirbar wäre. Dazu gehören der gesamte Spielzug, das Initialisieren der Logik, sowie das Laden und Speichern einer Datei.

Dafür startet die Klasse jede Operation in einem Thread, der seine Operation ausführt und danach stirbt.

##### Attribute

```
private final GameLogic logic
```

Die eigentliche Spiellogik, die alle oben erwähnten Operationen in einem eigenen Thread ausführen soll.

```
private final JavaFXGUI gui
```

Die eigentliche GUI, welche benötigt wird um die Buttons, welche während eines Spielzuges nicht bedienbar sein sollen zu an- und auszuschalten. Außerdem ist sie für das Behandeln von eventuell auftretenden Exceptions in der Methode, welche einen Spielzug anstößt, notwendig.

##### Methoden

```
public void doOutOfJavaFx(Cluedo.Runnable runnable)
```

Führt eine übergebene Operation (Spielzug, Initialisierung, Laden oder Speichern) außerhalb des JavaFX Application Thread Kontextes aus, damit in diesen blockierende Aufrufe stattfinden können.

Dafür wird jeweils ein neuer Thread gestartet und EventHandler für den Abschluss oder Abbruch der übergebene Operation installiert. Das eigene `Runnable` Interface ist notwendig, da das `java.lang.Runnable` Interface kleine checked Exceptions werfen kann.

### 3.4.1.5 AsyncJavaFXGUI

Diese Klasse kümmert sich darum, dass Aufrufe von Operationen aus anderen Threads diese Operationen innerhalb des JavaFX Application Threads ausführen können. Das ist notwendig, da es sonst nicht möglich ist die Elemente der JavaFX GUI zu manipulieren. Daher implementiert diese Klasse ebenfalls das `GUIConnector` Interface, um von der Logik aus aufgerufen werden zu können.

#### Attribute

```
private final JavaFXGUI gui
```

Die eigentliche JavaFX GUI an welche die Aufrufe weitergeleitet werden sollen.

```
private CompletableFuture<Void> finished
```

Ein zukünftiges Ereignis, welches signalisiert, dass die Operation fertig ist.

```
private static class WrappedCheckedException extends RuntimeException
```

Eine Hilfsklasse für die Methode `doInJavaFxWithCluedoException`.

Notwendig, da das Interface `java.lang.Runnable` keine checked Exceptions wirft. Diese Klasse verpackt eine checked Exception, um diese, falls sie bei der Ausführung von Operationen auftreten, verpacken, werfen und wieder entpacken zu können. Das ist notwendig, um diese Exceptions außerhalb des JavaFX Application Kontextes weiter behandeln zu können.

#### Methoden

Auf die wichtigen Methoden des `GUIConnector` Interface wurden bereits in der Beschreibung der Klasse `JavaFXGUI` näher eingegangen.

```
private <T> T doInJavaFx(Callable<T> callable)
```

Hauptmethode, um zu realisieren, dass Operationen innerhalb des JavaFX Application Thread Kontextes ausgeführt werden können. Andere Methoden dieser Klasse werden auf diese zurückgeführt. Dabei wird eine `FutureTask` erzeugt, da auf diese blockierend gewartet werden kann, bis die übergebene Operation abgeschlossen wurde. Um den Task nun innerhalb des JavaFX Application Thread Kontextes ausführen zu können, wird die `runLater` Methode verwendet.

### 3.4.1.6 ShowCardsDialog

Eine Klasse die einen Dialog darstellt, welcher dem User ermöglicht aus den verfügbaren Karten auf der Hand eine davon zu zeigen.

## 3.4.2 Die Klassen der Logic

### 3.4.2.1 GUIConnector

`GUIConnector` das Interface, welches benutzt wird, um die Kommunikation zwischen Logik und GUI zu trennen.

Dieses definiert die Methoden, welche notwendig sind, um Aktionen im Spielverlauf auf der GUI anzuzeigen. Das Interface wird von den Klassen `JavaFXGUI` und `AsyncJavaFXGUI` implementiert. Dazu gehören unter anderem, die Anklage, die Reaktion auf eine Anklage, die Verdächtigung, die Zusammenfassung einer Verdächtigungsrunde und einige weitere, welche im Detail dem Code zu entnehmen sind.

### 3.4.2.2 GameLogic

Repräsentiert die Spiellogik, mit allen Spielregeln und Daten des aktuell laufenden Spiels.

Hier findet der gesamte Spielablauf statt. Bei Spielstart wird eine statische Methode genutzt welche alle Daten aus der `InitialGameDataCluedo.json` vorbereitet, sodass am Ende dieser Methode ein Spiel erzeugt werden kann. Beim Laden von Spielständen wird keine neue Instanz erzeugt, sondern die bereits bestehende modifiziert.

#### Attribute

```
private final GameCell[][] gameField
private final Room[] rooms
private final Weapon[] weapons
private final Card[] cards
```

Alle sich im Spiel befindlichen Karten, Waffen, Räume und das Spielfeld. Diese Informationen werden aus der `InitialGameDataCluedo.json` geladen und sind nicht in dem Code festgelegt.

```
private Player[] players
```

Alle sich im Spiel befindlichen Spieler. Diese werden bei der Instanziierung der Klasse aus den Angaben des Users bei Spielstart erzeugt.

#### Methoden

```
public static GameLogic createInitialGameLogicFromJSON(InitialGameDataJSON
initGameDataJSON, int playerAmount, AIDifficulty[] difficulties)
```

Generiert aus der `InitialGameDataCluedo.json` die notwendigen Daten, um eine Instanz der Klasse erzeugen zu können.

```
public void makeMove(Position gameCellPosition) throws CluedoException
```

Wird bei jedem validen Klick des Users auf das Spielfeld aufgerufen und stößt einen gesamten Spieldurchlauf an. Diese Methode enthält die gesamte Spiellogik.

```
private void handleAILoop() throws CluedoException
```

Wird von der `makeMove` Methode aufgerufen und kümmert sich um alle Züge der Computergegner.

```
private Set<Position> generateValidMoves(Set<Position> startPositions, int
steps, boolean exact)
```

Liefert zu einer übergebenen Menge an Positionen, alle von dort aus erreichbaren Positionen auf dem Spielfeld. Dabei werden die Spieler auf dem Spielfeld mitberücksichtigt. Der parameter `exact` gibt an, ob alle übergebenen Schritte verbraucht werden müssen, oder ob noch welche übrig bleiben dürfen. Dies ist für das Betreten von Räumen wichtig.

```
public List<Position> getShortestPath(Position startPosition, Set<Position> destinations)
```

Liefert den kürzesten Weg von der übergebenen Startposition zu der übergebenen Menge an Zielpositionen. Dafür wird der Algorithmus der Breitensuche verwendet. Das Ergebnis ist eine Liste beginnend bei der Position des ersten Schrittes bis zu dem nächsten Ziel. Falls kein Weg gefunden werden konnte, wird `null` zurückgegeben.

#### 3.4.2.3 GameCell

Ein Spielfeldelement aus dem das Spielfeld besteht. Die Flur- und Wandelemente sind Singeltons und ein Raumelement hält eine Referenz auf den entsprechenden Raum.

#### 3.4.2.4 Player

Sind die Spieler, die an dem Spiel teilnehmen. Sowohl der User als auch die Computergegner. Dieser enthält alle Karten, die der Spieler auf der Hand hat, eine Referenz auf eine Instanz von `AI` (falls es sich um den User handelt ist diese `null`), die Spielfigur, welche der Spieler steuert und die Notizen, welche sich in eigene und Notizen über andere Spieler, unterscheiden. Eine Instanz der Klasse `AI` notwendig, da die „schlaue“ Spielstärke einige Attribute enthält.

#### Methoden

```
private static AI initAIByDifficulty(AIDifficulty aiDifficulty, int characterInGameCount, int cardsInGameCount)
```

Fabrikmethode für eine KI-Stärke. Diese wird je nach übergebenen Schwierigkeitsstufe erstellt. Die Anzahl der Spielfiguren und Karten im Spiel sind nur für die „schlaue“ KI relevant, da diese auf Basis dieser Informationen eigene Datenstrukturen erstellt.

#### 3.4.2.5 AIDifficulty

Ein Enum für die verschiedenen Schwierigkeitsstufen der Computergegner.

#### 3.4.2.6 AI

Bei der Klasse `AI` handelt es sich um eine Abstrakte Klasse. Diese definiert Methoden, welche von den Klassen für die verschiedenen stärken der Computergegner implementiert werden sollen. Da viele Methoden und Algorithmen in mehreren Implementierungen benötigt werden und teilweise aufeinander aufbauen, sind diese direkt in der Klasse `AI` implementiert.

#### 3.4.2.7 AIStupid, AINormal, AISmart

Diese Klassen sind die Implementierungen der verschiedenen Computergegner. Jeder `Player` hält eine Referenz auf eine Instanz dieser drei Klassen (außer dem `Player` des Users). Jedoch hält eine `AI` keine Referenz auf ihren `Player`. Daher muss dieser bei den meisten Methoden übergeben werden. Da viele Funktionen der verschiedenen Stärken wiederverwendet wurden, benutzen diese Klassen viele Methoden aus der `AI`.

Eine Ausnahme bildet hierbei die Klasse `AISmart`. Diese Klasse ist die einzige Stärke, welche eigene Attribute besitzt und nicht ausschließlich auf den Attributen des Players arbeitet. Außerdem enthält diese eine innere statische Klasse `Turn` und ein Enum `TempNotes`. Diese sind notwendig, um den Spielverlauf zu speichern. Der Spielverlauf wird verwendet, um im Nachhinein Rückschlüsse zu ziehen.

### Attribute

```
private final int[][] shownCardsCount
```

Hat dieselbe Struktur wie die Notizen über andere Spieler und wird dafür genutzt, um zu speichern, welche Karte, welchem Spieler wie häufig gezeigt wurde.

```
private final List<Turn> gameHistory
```

Repräsentiert die Spielhistorie und enthält alle Züge, bei denen die der Computerstärke selbst keine Verdächtigung geäußert hat.

### Methoden

```
private void analyzeHistory(Player[] players, Card[] cards, Player self)
```

Diese Methode wertet die gespeicherte Spielhistorie aus, um neue Erkenntnisse über die Karten der Mitspieler zu gewinnen. Sie wird immer aufgerufen, wenn ein Zug eines beliebigen anderen Spielers beendet wurde. Dafür werden die Notizen des Players genutzt, um für jeden Eintrag in der Historie zu überprüfen, ob neue Erkenntnisse gewonnen werden konnten. Dafür werden die Zeilen der Notizen betrachtet und ggf. Spieler ausgeschlossen, falls diese Karte von einem bestimmten Spieler besessen wird.

#### 3.4.2.8 Exceptions

Die Klasse `CluedoException` dient dazu, alle Fehler, welche im Spielverlauf auftreten zu repräsentieren. Die `CluedoException` unterscheidet sich anhand des Attributes `ExceptionType` type.

Bei der Klasse `ExceptionType` handelt es sich um ein Enum, welches alle Fehlerarten, die im Spiel auftreten können, repräsentiert. Der `ExceptionType` wird in der `handleException` Methode der `JavaFXGUI` Klasse genutzt, um einen Fehlermeldungstext zu generieren und diese ggf. zu behandeln.

#### 3.4.2.9 Card und CardType

Die Klasse `Card` repräsentiert eine Spielkarte im Spiel. Diese wird nach den drei Kartentypen (Raum, Waffe, Person), welche im Enum `CardType` definiert sind, unterschieden. Die Klasse `Card` enthält ebenfalls einige Hilfsmethoden, welche aus einer Sammlung von Karten einen bestimmten Kartentypen filtert.

#### 3.4.2.10 CardTriple

Diese Klasse repräsentiert drei Karten verschiedener Kartentypen. Diese Klasse wird für Verdächtigungen, Anklagen und Reaktionen auf Anklagen verwendet. Dabei können einige der Attribute `null` sein, um deren Abwesenheit anzuzeigen.

#### 3.4.2.11 Character, Weapon und Room

Diese Klassen stellen Spielelemente dar, welche dynamisch aus der `InitialGameDataCluedo.json` geladen und erzeugt werden. Die Klasse `Room` enthält außer dem Namen noch den Mittelpunkt des Raumes auf dem Spielfeld, welcher genutzt wird, um zu repräsentieren, dass ein Spieler sich in diesem Raum befindet. Außerdem sind in dieser Klasse noch die Positionen der Türen und der Raum, welcher über den Geheimgang, falls vorhanden, erreicht werden kann, festgelegt.



#### 3.4.2.12 NoteSelf und NoteOthers

Sind Enums, aus denen die Notizen der Spieler über die eigenen und die Karten der Mitspieler bestehen. Diese enthalten außerdem noch die Stringrepräsentation welche zum Speichern und Laden verwendet werden. Die Notizen des Users und der Computergegner unterscheiden sich nicht, da diese von derselben Klasse `Player` repräsentiert werden.

#### 3.4.2.13 Position

Stellt eine Position auf dem Spielfeld dar und besteht aus x- und y-Koordinate.

#### 3.4.2.14 StartGameInfo

Stellt die Informationen, welche bei dem Start eines neuen Spiels abgefragt werden, dar. Dazu gehören die Anzahl der Spieler, als auch deren Schwierigkeitsstufen. Auf Basis dieser Informationen wird ein neues Spiel erstellt.

#### 3.4.2.15 Das Package JSON

Enthält alle Klassen, welche der Struktur der JSON Dateien entsprechen, welche in diesem Programm gelesen und geschrieben werden. Dazu gehören die Spielstände als auch die Initialisierungsdatei. Die Initialisierungsdatei wird bei dem Start des Programms gelesen und enthält alle für das Spiel relevante Daten, wie Spielfiguren, Räume, Waffen und das Spielfeld. Die Daten in dieser Datei werden als Grundlage, für die Validierung bei dem Einlesen von Spielständen genutzt. Falls dieser abweichen, kann die Spielstandsdatei nicht geladen werden.

#### 3.4.2.16 GameDataConverter

Diese Klasse enthält ausschließlich statische Methoden die dazu dienen den internen Spielzustand in das Format der Spielstandsdateien zu konvertieren und umgekehrt.

#### 3.4.2.17 LoadedGameLogic

Diese Klasse enthält die Daten, welche nach dem erfolgreichen Laden eines Spielstandes notwendig sind, um die `GameLogic` entsprechend anzupassen. Die Daten liegen in derselben Struktur wie in der `GameLogic` vor. Diese Klasse dient dem Zweck die Daten bei dem Leseprozess zu sammeln und erst in die `GameLogic` zu schreiben, nachdem die Validierung erfolgreich war.

#### **Methoden**

```
public void commit(GameLogic logic)
```

Schreibt die gesammelten Daten in die übergebene `GameLogic`. Diese Methode wird nur aufgerufen, wenn bei der Validierung der Daten keine Fehler entstanden sind. Ansonsten werden die Daten der Klasse verworfen und das ursprüngliche Spiel wird nicht verändert.

## 3.5 Programmtests

### 3.5.1 Spielstart

Testfall	Ergebnis
Das Programm wird ausgeführt.	Das Hauptfenster erscheint und darüber befindet sich ein Dialog, welcher den User auffordert, die Gesamtanzahl der Spieler, sowie die Stärken der Computergegner auszuwählen.
Das Programm wird gestartet und versucht dieser sofort mit der Tastenkombination ALT+F4 oder über die Programmleiste des Betriebssystems zu schließen.	Das Programm lässt sich nicht schließen. Es muss ein Spiel gestartet werden. Erst danach kann das Spiel mit einem Klick auf das X-Symbol in dem Fenster geschlossen werden.
Das Programm wird ausgeführt und in dem erscheinenden Dialog wird eine beliebige Auswahl getroffen. Anschließend wird diese mit der Auswahl des "Los Gehts!" - Buttons bestätigt.	Das Dialogfenster verschwindet und das Hauptfenster wird mit Bildern und Text beladen. Auf der linken Seite des Fensters erscheint das Spielfeld, auf dem sich die Spielfiguren an deren Startpositionen befinden. Außerdem befinden sich insgesamt sechs Waffen in den Räumen, wobei maximal eine Waffe in einem Raum enthalten ist. Es wird eine Würfelseite in der Mitte des Fensters angezeigt und darunter befinden sich die Karten des Users welche je nach Kartentyp auf die drei Listen aufgeteilt sind. In den Notizen stehen überall die Einträge "-----". Eine Ausnahme bilden dabei die eigenen Notizen in der ersten Spalte. Bei diesen sind die eigenen Karten bereits eingetragen.
Im laufenden Spiel wird, während der User am Zug ist, über die Schaltfläche "Spiel" oben links im Fenster mit der Auswahl des Menüpunktes "Neues Spiel" ein neues Spiel gestartet.	Der User wird darauf hingewiesen, dass dieser ein laufendes Spiel beendet und gefragt, ob das aktuelle Spiel gespeichert werden soll. Nachdem der User eine Entscheidung getroffen hat und mit der Aktion fortfährt, erscheint wieder der Anfangsdialog und der User kann die Parameter für eine neue Spielrunde einstellen.

### 3.5.2 Während des Spielablaufs

Testfall	Ergebnis
Der User klickt auf ein Feld, welches mit der gewürfelten Anzahl an Schritten nicht erreicht werden kann.	Es erscheint ein Dialog mit dem Inhalt: "Kein valider Schritt!". Nach dem Schließen des Dialogs ist der User weiterhin am Zug und kann eine andere Auswahl auf dem Spielfeld tätigen.
Der User klickt auf ein Feld, welches mit der gewürfelten Anzahl an Schritten erreicht werden kann.	Die Spielfigur des User (rot) wird an die ausgewählte Position animiert. Die Animation ist linear und dauert unabhängig von der zurückgelegten Distanz eine Sekunde. Danach sind die Mitspieler am Zug und bewegen sich ebenfalls mit einer Animation über das Spielfeld und äußern ggf. eine Verdächtigung.
Der User wurde von einem Mitspieler in einen Raum "zitiert" und ist nun wieder an der Reihe.	Der User hat nun die Wahl, ob dieser in dem Raum, in den er "zitiert" wurde, eine Verdächtigung äußert, oder den Raum verlässt. Um eine Verdächtigung zu äußern kann der User auf den Raum klicken. Es erscheint der Verdächtigungsdialog. Falls dabei die "Mögliche Züge-Option" eingeschaltet ist, wird der Raum, in dem der User sich befindet, nicht markiert. Das liegt daran, dass diese Option vorerst nur zum Debuggen eingebaut wurde und erst kurz vor Fertigstellung des Spiels, für den User zugänglich gemacht wurde. Da dieses Feature nicht in den Anforderungen enthalten war, wurde es aufgrund von Zeitmangel nicht komplett fertiggestellt.
Der User kann mit der Checkbox "Mögl. Züge" während des Spiels, die validen Züge auf dem Spielfeld, ein- und ausblenden.	Die für den User erreichbaren Positionen auf dem Spielfeld werden mit schwarzen Kreisen markiert. Die Einschränkungen dieser Features wurden bereits eine Zeile höher erwähnt.
Der User macht einen Spielzug, indem er auf ein valides Spielfeld klickt. Direkt danach versucht er eine Anklage zu äußern oder einen Eintrag in dem Spielmenü oben links auszuwählen.	Die UI-Elemente sind ausgegraut und reagieren nicht auf die Interaktion des Users. Erst, wenn dieser wieder an der Reihe ist, können die Elemente wieder bedient werden.

### 3.5.3 Laden und speichern

Testfall	Ergebnis
Während der User am Zug ist, kann der Spielstand über den "Spiel Speichern"-Menüeintrag, oben links im Fenster gespeichert werden. Alternativ geschieht dies auch, wenn eine Aktion ausgeführt wird, welche den aktuellen Spielstand verwirft. Der User wählt dabei den "Speichern"-Button in dem sich öffnenden Dialog.	Ein Speicherdialog öffnet sich in demselben Verzeichnis indem auch die <code>Cluedo_Smirnov_ws20.jar</code> liegt. Nach der Eingabe eines Dateinamens wird eine Spielstandsdatei mit diesem Namen im ausgewählten Verzeichnis erstellt.
Der User kann die soeben gespeicherte Spielstandsdatei laden, indem der "Spiel Laden"-Eintrag in dem "Spiel"-Menü gewählt wird.	Es erscheint ein Ladedialog, in dem der User aufgefordert wird, eine Datei mit der Endung <code>.json</code> auszuwählen. Die soeben gespeicherte Datei wird gewählt und der Spielzustand wiederhergestellt. Der User ist am Zug. Falls eine andere <code>.json</code> Datei gewählt wurde oder beim Laden ein Fehler auftritt, erscheint eine aussagekräftige Fehlermeldung und die Datei wird nicht geladen.
Die Beispielspielstandsdatei kann geladen werden, während der User am Zug ist.  Datei: <code>Spielstand.json</code>	Das Spiel nimmt den Zustand aus der Spielstandsdatei an und der User ist am Zug.
Eine fehlerhafte Spielstandsdatei wird vom User versucht zu laden, während dieser am Zug ist.  Datei: <code>TestUngültigesJson.json</code>	Es erscheint eine Fehlermeldung mit dem Inhalt: "Die Datei enthält Syntaxfehler". Nachdem diese geschlossen wurde, befindet sich das Spiel in demselben Zustand wie davor.
Eine fehlerhafte Spielstandsdatei wird vom User versucht zu laden, während dieser am Zug ist.  Datei: <code>TestMenschImFlurObwohlLautStringInVeranda.json</code>	Es erscheint eine Fehlermeldung mit dem Inhalt: "Die Spielerposition stimmt nicht mit dem Raum überein". Nachdem diese geschlossen wurde, befindet sich das Spiel in demselben Zustand wie davor.

### 3.5.4 Ausgewählte Spielsituationen

Für die hier beschriebenen Testfälle werden die Spielstandsdateien aus dem Verzeichnis "ManuelleTests" geladen. Diese befinden sich zusammen mit dem Programm in dem Verzeichnis "final-binaries".

Testfall	Ergebnis
Ein Computergegner wurde auf dem Flur eingesperrt.  Datei: TestKIEingeschlossenImFlur.json	Der Computerspieler, welcher die gelbe Spielfigur steuert, setzt aus. Es findet keine Animation der gelben Spielfigur statt und die weiße Spielfigur macht direkt einen Zug.
Ein Computergegner wurde in einem Raum eingesperrt.  Datei: TestKIEingeschlossenImSpeisezimmer.json	Falls eine ungerade Schrittzahl gewürfelt wurde und der User sich vor die Tür des Speisezimmers stellt, äußert der Computerspieler direkt eine Verdächtigung. Dies geschieht, obwohl die Spielfigur sich bereits in diesem Raum befand.
Der User wurde von anderen Spielfiguren im Flur eingeschlossen.  Datei: TestMenschEingeschlossenImFlur.json	Der User kann auf die Position klicken, auf der er sich bereits befindet. Unabhängig von der gewürfelten Schrittzahl.