

# **Programmierpraktikum**

## **Scotland Yard**

**Benutzer- und Programmierhandbuch**

**Guillaume Fournier-Mayer**

**tinf101922@fh-wedel.de**

Fachrichtung Technische Informatik

7. Fachsemester

9. Verwaltungssemester

3. Februar 2020, Hamburg





# Inhaltsverzeichnis

<b>1 Allgemeine Problemstellung</b>	<b>2</b>
1.1 Spielregeln . . . . .	2
1.2 Darstellung . . . . .	2
1.3 Steuerung . . . . .	4
1.4 KI . . . . .	5
1.5 Log . . . . .	9
1.6 Spielstand . . . . .	10
1.7 Zwischenstand . . . . .	11
<b>2 Benutzerhandbuch</b>	<b>13</b>
2.1 Ablaufbedingungen . . . . .	13
2.2 Programminstallation/Programmstart . . . . .	13
2.3 Bedienungsanleitung . . . . .	13
2.3.1 Oberfläche . . . . .	14
2.3.2 Neues Spiel starten . . . . .	15
2.3.3 Spielprinzip . . . . .	18
2.3.4 Spiel speichern . . . . .	21
2.3.5 Spiel laden . . . . .	23
2.3.6 Spiel beenden . . . . .	25
2.3.7 Godmode . . . . .	25
2.3.8 Spiellog . . . . .	25
2.4 Fehlermeldungen . . . . .	25
2.5 Datenstrukturen . . . . .	28
<b>3 Programmierhandbuch</b>	<b>31</b>
3.1 Entwicklungskonfiguration . . . . .	31
3.2 Problemanalyse und Realisation . . . . .	32
3.2.1 Spielfeld . . . . .	32
3.2.2 Spieler . . . . .	34
3.2.3 Spiellogik . . . . .	43
3.2.4 Speichern und Laden von Spielständen . . . . .	48
3.2.5 Grafische Oberfläche . . . . .	50
3.3 Beschreibung grundlegender Klassen . . . . .	51
3.3.1 Gui-Quellpaket . . . . .	51
3.3.2 Logic-Quellpaket . . . . .	56

## *Inhaltsverzeichnis*

3.3.3	Logic.util-Quellpaket . . . . .	61
3.3.4	Logic.board-Quellpaket . . . . .	66
3.3.5	Logic.player-Quellpaket . . . . .	70
3.3.6	Test.gui Quellpaket . . . . .	76
3.3.7	Benutze Datenstrukturen . . . . .	77
3.4	Programmorganisationsplan . . . . .	78
3.5	Programmtests . . . . .	80

# 1 Allgemeine Problemstellung

Zu implementieren ist eine Computervariante des Brettspielklassikers *Scotland Yard*, bei dem eine Gruppe von Detektiven in den Straßen Londons versucht, den ominösen Mister X zu fangen. Für einen ersten Eindruck ist evtl. dieses Beispielprogramm hilfreich (das im selben Ordner den Spielplan und das Netz benötigt) Bindend bleibt bei Differenzen aber immer diese Aufgabenstellung!

## 1.1 Spielregeln

Es sollen die Originalspielregeln 1:1 umgesetzt werden - bis auf folgende Abweichungen bzw. Konkretisierungen:

- die Anzahl der Detektive muß 3, 4 oder 5 sein
- Mister X hat keine farblose Spielfigur, sondern wird durch ein Fragezeichen dargestellt (siehe Kapitel Darstellung)
- die Spieler und Mister X *ziehen* keine Startkarten, sondern es wird ihnen per Zufall eine noch nicht vergebene Startposition zugeteilt. Die möglichen Startpositionen sind 13, 26, 29, 34, 50, 53, 91, 94, 103, 112, 117, 132, 138, 141, 155 ,174, 197, 198
- (Klarstellung) Mister X kann zusätzlich zu seinem Startvorrat nur die Tickets benutzen, die er von den Detektiven durch ihre Züge bekommen hat. Wenn diese also z.B. nie U-Bahn fahren, dann gehen auch Mister X irgendwann die U-Bahn-Tickets aus (die Spielregeln sind hier widersprüchlich)
- es gibt zur Vereinfachung keine Doppelzüge für Mister X

## 1.2 Darstellung

Die Oberfläche des Programms wird im Wesentlichen vom Spielplan eingenommen. Dieser soll initial in Originalgröße dargestellt werden (1081x814 Pixel). Bei Vergrößerung/Verkleinerung des Fensters wächst/schrumpft der Spielplan gleichermaßen. Auf dem Spielplan werden an den jeweils aktuellen Positionen die Detektive angezeigt. Mister X ist an seiner aktuellen Position sichtbar,

## 1 Allgemeine Problemstellung

wenn er von einem menschlichen Spieler und nicht von der KI gespielt wird, ansonsten an seiner letzten Zeigeposition (in den ersten beiden Zügen, also vor dem ersten Zeigen, wird Mister X dann gar nicht angezeigt). Die Detektive sind dabei jeweils durch ein graues Dreieck mit schwarzer Außenlinie und einen farbigen Kopf (ebenfalls mit schwarzer Außenlinie) darzustellen. Die maximal 5 Detektive haben in dieser Reihenfolge die Farben blau, gelb, rot, grün und schwarz. Mister X wird als orangefarbener Kreis mit einem weißen Fragezeichen darin angezeigt (siehe Beispielprogramm). Für Mister X soll es zusätzlich eine Cheatmöglichkeit geben, mit der er immer (also unabhängig ob er sich zeigen muß oder nicht und ob er KI-gesteuert ist oder nicht) an seiner aktuellen Position angezeigt werden kann.



Neben dem Spielplan ist die jeweils aktuelle Fahrtentafel von Mister X zu sehen - also eine Anzeige seiner Züge bzw. der jeweils dafür verbrauchten Tickets. Die Aufteilung soll wie im Beispielprogramm 3 Spalten und eine Nummerierung von 1 bis 24 vorsehen und außerdem kenntlich machen, welche der Züge verdeckt und welche die *Zeigezüge* von Mister X sind. Ob Ihr hier ein komplettes Bild der Fahrtentafel ladet und nutzt oder eine eigene Darstellungsform wählt, ist Euch freigestellt. Es müssen hier nur die verbrauchten Tickets sichtbar sein, die Zugstationen sind ja eh geheim.

## 1 Allgemeine Problemstellung

Weiterhin soll die Oberfläche den am Zug befindlichen Spieler (bzw. Mister X) und seine noch vorhandenen Tickets anzeigen, wobei jeweils das entsprechende Bild des Tickets sowie als Textausgabe die vorhandene Anzahl darzustellen sind. (Im Archiv der Ticketbilder findet sich auch eines für eine mögliche Startposition. Wer möchte, darf auch diese irgendwie auf der Oberfläche mit darstellen)

### 1.3 Steuerung

Zu Beginn eines neuen Spiels wird der Spieler gefragt, ob er Mister X und/oder die Detektive steuern möchte (andernfalls macht dies jeweils die KI) und wie viele Detektive (3, 4 oder 5) mitspielen sollen. Danach sind immer abwechselnd Mister X und die Detektive in der o.g. Reihenfolge am Zug. Die Detektive werden immer komplett entweder von der KI oder vom Spieler gesteuert.

Der Spieler kann mit der Maus auf eine beliebige Stelle des Spielplans klicken. Es soll dann die Station von 1 bis 199 bestimmt werden, die den geringsten Abstand zu der Klickposition hat (gemeint ist hier die Luftlinie, also der euklidische Abstand – $\sqrt{2}$  Satz des Pythagoras). Allerdings soll hier ein Maximalabstand definiert werden, so daß nur Klicks, die eine Station auch grob treffen (und nicht irgendwo im Park liegen), als zu dieser Station gehörig gezählt werden. Falls diese Station zu derjenigen *benachbart* ist, an der sich der aktuelle Spieler gerade befindet (also mindestens ein Verkehrsmittel dahin führt) und dieser Spieler auch noch ein passendes Ticket hat, wird er auf die benachbarte Station bewegt und das passende Ticket wird aus seinem Vorrat entfernt bzw. geht an Mister X. Falls es mehrere mögliche Verkehrsmittel für die Strecke gibt, soll der Spieler (z.B. über ein Popupmenü wie im Beispielprogramm) entscheiden können, welches davon er nehmen möchte.

Falls der Spieler auch Mister X steuert, ist das Verhalten hier dasselbe, nur daß verbrauchte Tickets einfach aus seinem Vorrat verschwinden. Die Fahrten-tafel wird parallel zu seinen Zügen mit Tickets gefüllt.

Kann ein Detektiv nicht mehr ziehen, weil er keine Tickets mehr hat, die von seiner aktuellen Station wegführen, wird er im Ablauf aller Detektive ausgelassen. Kann kein Detektiv mehr ziehen, hat Mister X gewonnen.

Wo sich welche Station befindet und welche Verkehrsmittel dort wohin genommen werden können, ist in der netz.json hinterlegt. Diese beinhaltet

## 1 Allgemeine Problemstellung

- die Nummer der Station
- die Pixelposition auf dem Spielplan (0/0 ist oben links, 1/1 ist unten rechts)
- die per U-Bahn zu erreichenden Stationen aufsteigend
- die per Bus zu erreichenden Stationen aufsteigend
- die per Taxi zu erreichenden Stationen aufsteigend
- die per Boot zu erreichenden Stationen aufsteigend.

Hilfe zum Parsen einer JSON-Datei findet sich unten.

Es bietet sich an, alle 199 Stationen in einem Array zu verwalten. Die Verbindungen zu anderen Stationen eines Typs lassen sich jeweils gut in einer Menge ablegen.

### 1.4 KI

Sowohl die Detektive als auch Mister X können optional auch von einer KI gesteuert werden. Auch die Steuerung von beiden *Parteien* in einem Spiel durch die KI soll möglich sein. Es wird von der KI dann jeweils der optimale nächste Zug bestimmt und die Spieler bzw. Mister X bewegen sich entsprechend zu den neuen Positionen. Natürlich sollen auch hier nur Züge auf *benachbarte* Stationen möglich sein und es können nur Verkehrsmittel verwendet werden, für die noch ein Ticket vorliegt (die Spielregeln sind also einzuhalten).

Zur Bestimmung eines optimalen Spielerzuges soll folgendes Vorgehen umgesetzt werden (wodurch Spielsituationen vergleichbar und nachvollziehbar werden):

- es werden folgende verschiedene Taktiken alle nacheinander durchgespielt (Achtung: Die Beispiele mit  $-i$  beziehen sich hier immer nur auf genau EINE Taktik. In Wirklichkeit sollen ALLE durchprobiert und die bestbewertete genommen werden!)
  - bewege Dich auf eine mögliche Zielposition (s.u., auf dem Bild rechts hellgrün eingekreist. Mister X ist also von der letzten Zeigeposition bei 116 aus 1x Taxi gefahren). Gibt es mehrere erreichbare, wähle davon die mit der kleinsten Stationsnummer  $-i$ , der blaue Detektiv würde sich demnach zu Station 118 begeben

## 1 Allgemeine Problemstellung

- bewege Dich auf eine direkt benachbarte U-Bahn-Station (um schnell zu anderen Standorten zu kommen). Gibt es mehrere erreichbare freie, wähle davon die mit der kleinsten Stationsnummer  $-i$ , der gelbe Detektiv würde sich demnach zu Station 185 begeben
  - bewege Dich in Richtung auf die letzte Zeigeposition von Mister X (s.u.). Benutze dafür den kürzesten (derzeit) freien Weg  $-i$ , der rote Detektiv würde sich demnach zu Station 70 (und folgend zu 87, 86, 116) begeben, weil dies die erste auf dem kürzesten Weg ist und sie zudem eine kleinere Nummer als die 89 hat (s.u.)
  - bewege Dich auf diejenige direkt benachbarte freie Position mit der kleinsten Stationsnummer (quasi als Fallback, wenn nichts anderes funktioniert)
- nach jedem versuchten Zug wird die neue Spielsituation (Spieler zieht auf Station x) anhand der folgend beschriebenen Bewertungsfunktion eingeschätzt. Der bestbewertete Zug wird umgesetzt (bei mehreren mit identischer Bewertung wird die Station mit der kleineren Nummer genommen). Die Bewertungsfunktion setzt sich aus diesen Einzelwerten und Gewichten zusammen, die aufaddiert werden. Ein größerer Wert ist besser:
    - wie viele mögliche Zielpositionen sind für alle Detektive im nächsten Zug erreichbar [genauer formuliert am 29.10.: wie viele der möglichen Zielpositionen sind im nächsten Zug von jeweils mindestens einem Detektiv erreichbar] geteilt durch die Gesamtzahl aller möglichen Zielpositionen (für die noch nicht gezogenen Detektive sollen hierbei noch ihre alten Stationen berücksichtigt werden)  
**Gewichtung:** Anzahl erreichbarer Zielpositionen durch Gesamtzahl der Zielpositionen \* 10  
 $-i$  nachdem der blaue Detektiv auf Station 118 gezogen ist, blieben noch 3 Zielpositionen übrig (104, 117, 127), die aber alle nicht erreicht werden können  
 $-i \ 0 / 3 * 10 = 0$
    - wie weit ist die *mittlere mögliche Zielposition* (s.u.) entfernt (damit die Detektive, die weiter von Mister X weg sind, sich grob in seine Richtung bewegen und nicht dumm in der Gegend rumlaufen)  
**Gewichtung:** wenn 10 Stationen oder mehr weg = 0, ansonsten (10 - Anzahl Stationen auf dem kürzesten Weg zur *mittleren möglichen Zielposition*)  
 $-i$  nachdem der blaue Detektiv auf Station 118 gezogen ist, blieben noch 3 Zielpositionen übrig (104, 117, 127). Deren Koordinaten sind

## 1 Allgemeine Problemstellung

765/341, 848/447 und 693/447, gemittelt also 769/412. Daran am dichtesten liegt Station 116, die genau 1 Station von 118 entfernt ist  
 $-i (10 - 1) = 9$

- wie viele Stationen sind von der aktuellen Position aus mit den vorhandenen Tickets in einem Zug erreichbar Gewichtung: Anzahl Stationen geteilt durch 13 (da dies bei Station 67 die meisten unterschiedlichen erreichbaren Stationen sind) \* 4
  - $-i$  angenommen, der blaue Detektiv hat noch 3 U-Bahn-, 4 Bus- und 3 Taxitickets, dann kann er von Station 118 aus 4 andere Stationen (alle per Taxi) erreichen
    - $-i 4 / 13 * 4 = 1,23$
  - was ist die geringste Ticketanzahl für ein Verkehrsmittel (hat man für eines gar kein Ticket mehr, schränkt das die Mobilität deutlich ein)
    - Gewichtung:** Wenn es noch mehr als 2 Tickets sind, ist die Bewertung 3, ansonsten Anzahl der Tickets.
    - $-i$  die geringste Zahl von Tickets beim blauen Detektiv (s.o.) ist 3
      - $-i = 3$
    - $-i$  die gesamte Bewertung wäre also  $0 + 9 + 1,23 + 3 = 13,23$
- zur besseren Nachvollziehbarkeit der KI-Schritte sollen die Einzelwerte der Bewertungsfunktion zusammengefasst und im Log (s.u.) ausgegeben werden können

Alle Spieler sind dabei nacheinander mit dem genannten Vorgehen an der Reihe. Es gibt somit keine *Gruppenabsprachen* der Detektive o.ä. Sollte tatsächlich einmal der Fall eintreten, daß alle Nachbarstationen besetzt sind, bewegt sich ein Detektiv in diesem Zug nicht, gibt dafür aber natürlich auch kein Ticket ab. Ansonsten muß ein Detektiv (und auch Mister X) aber ziehen, wenn dies möglich ist.

Immer dann, wenn bei einer Taktik mehrere Verkehrsmittel möglich sind, soll dasjenige genommen werden, von wem noch am meisten vorhanden sind (siehe Bewertungsfunktion). Gibt es davon gleich viele und mehrere sind für den Zug möglich, sollen diese in der Reihenfolge Taxi -i Bus -i U-Bahn (-i Boot) benutzt werden, also von *wertlos* zu *wertvoll*.

Eine **mögliche Zielposition** ist eine Station, an der sich Mister X gerade befinden könnte. Dies hängt offenkundig davon ab, wo er sich zuletzt gezeigt hat, wie viele Züge seitdem vergangen sind, wo die Detektive sich befinden und welche Tickets er benutzt hat. Diese Menge von Stationen soll von Euch für

## *1 Allgemeine Problemstellung*

die KI berechnet und genutzt werden. Die mittlere mögliche Zielposition ergibt sich aus einer Mittelung der Koordinaten in X und Y (aus der Netz.json) aller möglichen Zielpositionen und anschließender Bestimmung derjenigen Station, die sich am dichtesten (Luftlinie) an dieser mittleren Koordinate befindet.

Bei einer **Bewegung in Richtung auf die letzte Zeigeposition** von Mister X soll im Netz der Weg mit den wenigsten Zwischenstationen gesucht werden. Hierbei ist zu berücksichtigen, über welche Tickets ein Spieler noch verfügt. Hat er beispielsweise keine Bustickets mehr, darf die Route auch keine Busstrecken beinhalten. Dies kann im Extremfall heißen, daß kein Weg möglich ist. Ebenso dürfen keine Knoten im Weg vorkommen, die (derzeit) von Detektiven belegt sind. Falls mehrere Wege gleich kurz sind (also über dieselbe Anzahl an Zwischenstationen gehen), soll davon derjenige benutzt werden, der im ersten Schritt zur kleineren Stationsnummer führt.

Die KI für Mister X arbeitet identisch zu der der Spieler-KI, nur daß eine andere Taktik und andere Bewertungen berücksichtigt werden sollen. Teilweise können hier aber Funktionen der Spieler-KI wiederverwendet werden!

- **Taktik** von Mister X

- bewege Dich auf eine Station, die möglichst wenige Spieler im nächsten Zug erreichen können. Hierfür werden bei Mister X einfach alle erreichbaren Stationen durchgetestet

- **Bewertungsfunktion** von Mister X

- wie viele Detektive können die Position von Mister X im nächsten Zug erreichen Gewichtung: (Gesamtzahl Detektive - Anzahl Detektive, die Mister X erreichen können) \* 10 Hierbei ist zu berücksichtigen, welche Tickets die Detektive noch haben (dies weiß Mister X im echten Spiel ja auch). Ggf. kann also ein Detektiv Mister X auf der direkten Nachbarstation nicht erreichen, weil ihm das passende Ticket fehlt...
  - wie viele Stationen kann Mister X von der aktuellen Position aus mit den vorhandenen Tickets in einem Zug erreichen, auf denen kein Detektiv steht Gewichtung: Anzahl Stationen geteilt durch 13 (da dies bei Station 67 die meisten unterschiedlichen erreichbaren Stationen sind) \* 4
  - was ist die geringste Ticketanzahl für ein Verkehrsmittel (hat man für eines gar kein Ticket mehr, schränkt das die Mobilität deutlich ein) Gewichtung: Wenn es noch mehr als 2 Tickets sind, ist die Bewertung 3, ansonsten Anzahl der Tickets

## 1 Allgemeine Problemstellung

- auch hier sollen die Schritte mit in die Logdatei geschrieben werden (s.u.)

Mit dem beschriebenen KI-Vorgehen ist es leicht möglich, weitere Taktiken zu ergänzen, bestehende zu entfernen oder andere Gewichtungen der Bewertung vorzunehmen. Sie stellt somit eine brauchbare und in der KI verbreitete Mittellösung zwischen *alle möglichen Kombinationen von Zügen in einen Baum stecken und durchlaufen für x Ebenen* und dem simplen *wenn Zug A möglich ist, mache den, sonst wenn B möglich ist mache ...* dar.

### 1.5 Log

Zur Nachvollziehbarkeit soll pro Spiel eine Logdatei (als einfache Textdatei mit der Endung \*.log) geschrieben werden (es gibt genau eine, zu Beginn eines neuen Spiels wird diese also gelöscht bzw. überschrieben). Sie soll folgende Informationen exakt im angegebenen Format beinhalten:

- die Anzahl der Detektive (ganze Zahl)
- ob MisterX von der KI gesteuert wird (true oder false)
- ob die Detektive von der KI gesteuert werden (true oder false)
- die Startposition von MisterX
- die Startpositionen aller teilnehmenden Detektive
  - alle Werte bis hier werden kommasepariert in einer Zeile abgelegt
- alle Züge der Spieler bzw. von Mister X jeweils kommasepariert in einer Zeile, wie folgt:
  - welcher Spieler zieht (ganze Zahl ab 1 für die Detektive, 0 für MisterX)
  - bisherige Station
  - neue Station
  - restliche Tickets für U-Bahn, Bus, Taxi, Boot (bei den Detektiven stets 0)
  - welche der Taktiken (ab 1 in der o.g. Reihenfolge) wurde gewählt (bei einem menschlichen Spielzug = 0)
  - wie sieht die Bewertung (siehe oben) dazu aus (bei einem menschlichen Spielzug = 0.0)
    - hierbei soll zur besseren Lesbarkeit ein Dezimalpunkt verwendet werden

## *1 Allgemeine Problemstellung*

- wer das Spiel gewonnen hat (0 für Mister X, 1 für die Detektive)

Hier findet sich ein Beispiellog für ein relativ kurzes Spiel zum Vergleich (Init + 2 KI-Züge von Mister X und dazwischen ein menschlicher Zug von 3 Detektiven).

### **1.6 Spielstand**

Der aktuelle Spielstand soll jederzeit in einer vom Benutzer wählbaren Datei gespeichert und aus einer frei wählbaren Datei wieder geladen werden können. Seht hier bitte ggf. Rückfragen an den Nutzer vor, damit keine Spielstände ungewollt überschrieben oder gelöscht werden.

Eine Spielstandsdatei (ebenfalls eine JSON-Datei) hat folgende Einträge:

- MisterX
  - ob er von der KI gesteuert wird (true oder false)
  - die möglichen Zielpositionen
  - die letzte Zeigeposition (oder 0, wenn es noch keine gab)
  - die aktuelle Station
  - alle restlichen Tickets von Mister X (ganze Zahlen für U-Bahn, Bus, Taxi, Boot)
  - die belegten Einträge der Fahrtentafel (Ordinalwerte der verbrauchten Tickets ab 0 in der Reihenfolge U-Bahn, Bus, Taxi, Boot)
- Detektive
  - die Anzahl der Detektive (ganze Zahl)
  - ob sie von der KI gesteuert werden (true oder false)
  - für jeden Detektiv
    - \* die aktuelle Station
    - \* alle restlichen Tickets (ganze Zahlen für U-Bahn, Bus, Taxi)
- wer am Zug ist (0 = Mister X, Detektive ab 1)
- die wievielte Runde es ist (beginnend bei 1)
- ob das Spiel schon gewonnen wurde (true oder false)

## *1 Allgemeine Problemstellung*

Hier findet sich ein Beispielspielstand für ein relativ kurzes Spiel zum Vergleich (passend zum obigen Log). Sollten beim Speichern Fehler auftreten, wird der Nutzer darüber mit einer Meldung informiert. Sollten beim Laden Fehler auftreten, soll der Nutzer ebenfalls informiert und der bisherige Zustand des Spiels weiterverwendet werden. Nach einem erfolgreichen Laden wird das bisherige Log gelöscht und ein neues angelegt, als hätte gerade ein neues Spiel begonnen. Als Startpositionen werden die aktuellen Positionen genommen.

### **1.7 Zwischenstand**

Bei der Abnahme des Zwischenstands müssen die Vorgaben erfüllt werden:

- **GUI**
  - In einer FXML-Datei müssen alle Bedienelemente enthalten sein, die für eine Bedienung des Programms notwendig sind.
  - Es muss erkennbar sein, wie angezeigt wird, welche Fahrkarten ein Spieler noch zur Verfügung hat.
  - Eine Fahrtentafel mit Kennzeichnung der Züge, bei denen Mister X sich zeigt, muss sichtbar sein.
  - Nach Programmstart muss das Spielfeld sichtbar sein und sich bei Änderung der Fenstergröße gleichermaßen ändern.
  - Es muss eine Station angewählt werden können, was zu einer sichtbaren Reaktion führt (es genügt die Markierung mit einem Kreis). Die Markierung muss bei Änderung der Fenstergröße auf der Station bleiben.
- **Datenstrukturen** müssen erkennen lassen
  - wie und wo die Spieler definiert werden
  - wie die Netzstruktur abgelegt wird
  - welche Struktur einen Spielstand abbildet
  - wie/wo die verschiedenen Taktiken eines Spielers eingesetzt werden (welche Methoden welcher Klassen)
- **Tests** dürfen noch fehlschlagen, müssen aber vorhanden und kompilierbar sein und natürlich die korrekten Erwartungswerte enthalten. Mehrere Konstruktoren sind zu erstellen, um eine Spielsituation für einen Test erstellen zu können. Mindestens folgende Testfälle müssen abgedeckt werden:

## *1 Allgemeine Problemstellung*

- auf welchen Positionen kann Mister X im aktuellen Zug stehen (mögliche Zielpositionen)
- wurde Mister X gefangen
- jeweils zwei sinnvolle Tests für jede der 3 erstgenannten Taktiken; einer davon prüft, ob aus mehreren möglichen Stationen die kleinere gewählt wird
- ein Test für die viertgenannte Taktik
- ein Test für jede der 4 Einzelbewertungen einer Situation
- die Wahl der bestbewerteten Taktik

## 2 Benutzerhandbuch

### 2.1 Ablaufbedingungen

Um das Programm *ScotlandYard\_Fournier* Ausführen zu können, wird folgende Software benötigt.

Tabelle 2.1: Programm Anforderung

Software	Version
Java Laufzeitumgebung	1.8.0_232
JavaFX	8.0
GSON	2.8.0

### 2.2 Programminstallation/Programmstart

Das Programm wird als JAR-Paket unter dem Namen *ScotlandYard\_Fournier.jar* ausgeliefert und lässt sich ohne weitere Parameter wie folgt in der Kommandozeile ausführen:

Tabelle 2.2: Programmstart

---

```
java -jar ScotlandYard_Fournier.jar
```

---

Dabei ist zu beachten, dass sich in dem *lib*-Ordner die *GSON*-Bibliothek namens *gson-2.8.0.jar* befinden muss.

### 2.3 Bedienungsanleitung

Das Aussehen des Spielesfensters, vor allem die Menüleisten, werden durch die Fensterverwaltung des Betriebssystems vorgegeben. Abweichungen sind somit nicht ausgeschlossen. Die Abbildungen in dieser Anleitung wurden auf einem Ubuntu-System mit der *Xfce4*-Fensterverwaltung aufgenommen.

## 2 Benutzerhandbuch

### 2.3.1 Oberfläche

Beim starten des Spieles öffnet sich ein Fenster mit dem Titel *Scotland Yard*. Die Spieloberfläche ist dreigeteilt. Auf der rechten Seite befindet sich die eigentliche Spielkarte. Auf der linken Seite ist das Fahrtenbuch von Mister-X zu sehen. Die in orange gekennzeichneten Spielrunden sind jene Runden indem sich Mister-X zeigt. Am unteren Ende des Fensters sind die verfügbaren Tickets zu sehen. In der linken unteren Ecke steht der Spieler der zurzeit am Zug ist. Da noch kein Spiel gestartet worden ist, sind die verfügbaren Tickets sowie der aktuelle Spieler durch Initialwerte ersetzt. In der Menüleiste sind die Menüpunkte *Game* und *Edit* zu sehen durch die das Spiel gesteuert werden kann.

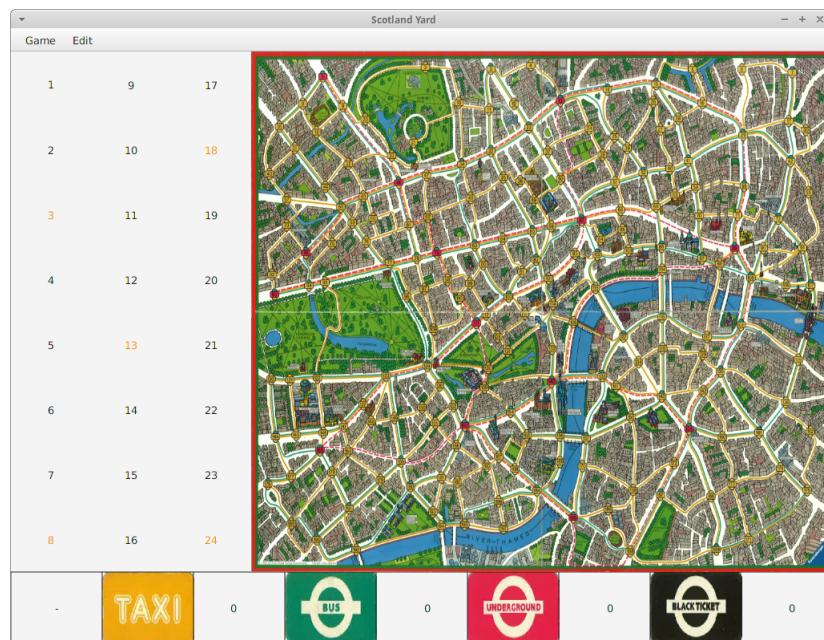


Abbildung 2.1: Spieloberfläche nach starten des Programmes

## 2 Benutzerhandbuch

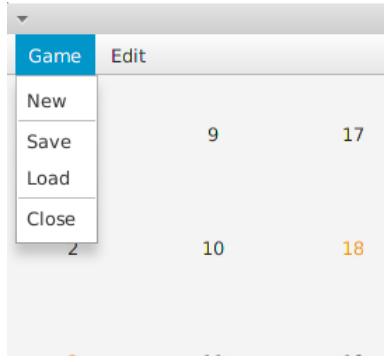


Abbildung 2.2: Menüpunkt *Game*

Abb. 2.2 zeigt alle verfügbaren Unterpunkte des Menüpunktes *Game*. Ein neues Spiel wird durch einen Klick auf *New Game* gestartet. Um ein Spiel zu Speichern oder zu laden, wird auf *Save* bzw. *Load* geklickt. Das Spiel wird über *Close* beendet.

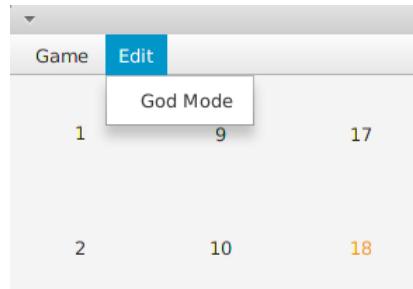


Abbildung 2.3: Menüpunkt *Edit*

Wie in Abb. 2.3 gezeigt hat der Menüpunkt *Edit* nur einen Unterpunkt *Godmode*. Durch einen Klick auf diesen Menüpunkt ist Mister-X dauerhaft auf der Spielkarte zu sehen. Durch einen weiteren Klick ist Mister-X wieder versteckt.

### 2.3.2 Neues Spiel starten

Um ein neues Spiel zu starten muss auf den Menüpunkt *New Game* geklickt werden (siehe Abbildung 2.2). Daraufhin öffnet sich ein Fenster welches die

## 2 Benutzerhandbuch

Spieleinstellungen abfragt. Die Einstellung *MisterX AI?* legt fest, ob Mister-X durch die künstliche Intelligenz (KI) oder einen menschlichen Mitspieler gesteuert werden soll. Äquivalent gilt dies für *Detectives AI?*. Zu beachten ist jedoch, dass diese Einstellung für alle Detektive gleichermaßen gilt. Über *How many Detectives* lässt sich die Anzahl an Detektiven bestimmen die am Spiel teilnehmen. Gültige Werte sind hier 3, 4 und 5.

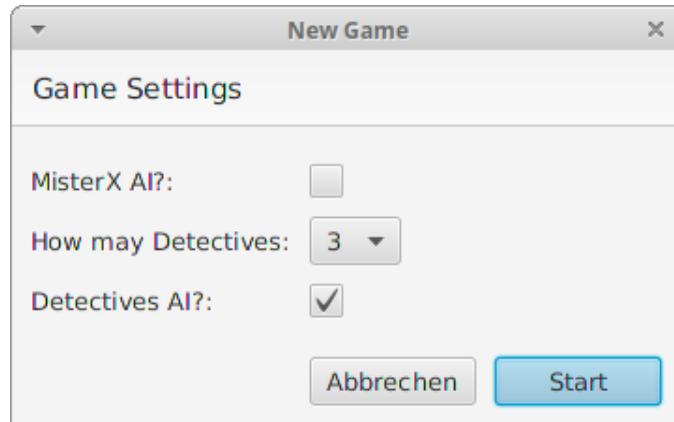


Abbildung 2.4: Dialog: Standardeinstellungen für ein neues Spiel

Sobald alle Einstellungen nach Wunsch vorgenommen wurden, kann das Spiel über *Start* gestartet werden. Nun werden die verfügbaren Tickets und der Spieler der zurzeit an der Reihe ist angezeigt.

## 2 Benutzerhandbuch

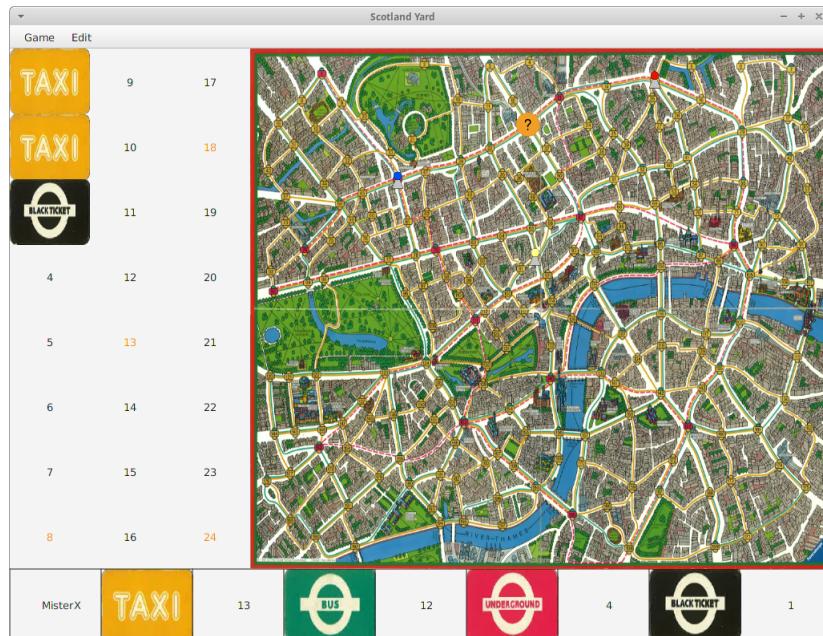


Abbildung 2.5: Spielfeld nach ein paar Spielzügen

Falls schon ein Spiel gestartet wurde, wird mit einer entsprechenden Meldung davor gewarnt, dass das aktuelle Spiel verloren gehen wird.

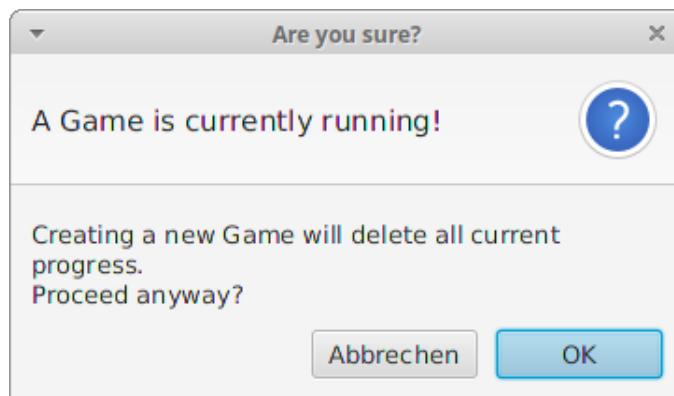


Abbildung 2.6: Warnung: Aktuelles Spiel geht beim starten verloren

### 2.3.3 Spielprinzip

Die Kernaufgabe des Spiels *Scotland Yard* besteht darin Mister-X zu fangen. Dabei Spielen die Detektive gegen Mister-X. Gespielt werden 24 Runden. Dabei ist Mister-X für die Detektive nur in bestimmten Runden zu sehen. Jeder Spieler startet dabei auf einer zufälligen Startposition.

#### Fahrentenbuch

MisterX ist für die Detektive nur in den Spielrunden 3, 8 13 18 und 24 Sichtbar. In den anderen Runden ist er verdeckt. Dabei wird jedoch jedes verbrauchte Ticket in dem Fahrentenbuch aufgelistet. Dadurch können die Detektive mit Hilfe der letzten Zeigeposition von Mister-X die möglichen Zielpositionen errechnen.

#### Spielende

Das Spiel ist für die Detektive gewonnenen sobald Mister-X gefangen wurde. Das heißt, dass einer der Detektiven auf die aktuelle Position von Mister-X gezogen ist. Des Weiteren können die Detektive Mister-X auch umkreisen, sodass dieser nicht mehr ziehen kann.



Abbildung 2.7: Meldung: Letzte runde erreicht. Mister-X hat gewonnen

## 2 Benutzerhandbuch



Abbildung 2.8: Meldung: Mister-X wurde gefangen

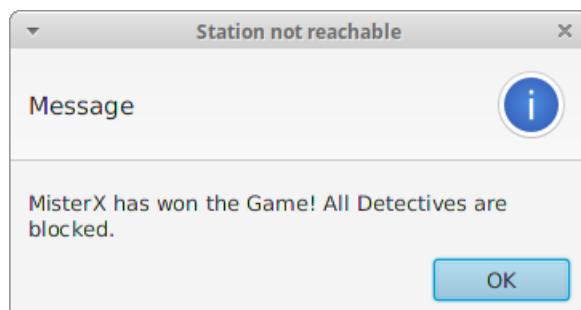


Abbildung 2.9: Meldung: Alle Detektive sind blockiert

Für Mister-X besteht die Chance zu gewinnen dadurch, dass den Detektiven die Tickets ausgehen. Zusätzlich kann Mister-X auch gewinnen falls die letzte Runde erreicht worden ist und dieser noch nicht gefangen wurde.

### Tickets

Um von einer Station zur anderen zu gelangen benötigen die Spieler Tickets für die Verkehrsmittel. Dabei stehen den Detektiven 10 Taxi-, 8 Bus- und 4 U-Bahntickets zur Verfügung. Mister-X hingegen stehen 10 Taxi-, 8 Bus-, 4 U-Bahn- und 2 Blacktickets zur Verfügung. Eine Besonderheit des Blacktickets ist hierbei, dass dieses für alle Verkehrsmittel, also auch für das Boot, gilt. Zusätzlich werden alle verbrauchten Tickets der Detektive, Mister-X gut geschrieben. Somit kann diesem nie die Tickets ausgehen.

### Züge

Alle Spieler die auf eine Station ziehen wollen, benötigen jeweils das richtige bzw. noch genügend Tickets um die Station zu erreichen. Hat ein Spieler nicht mehr genügend Tickets übrig, muss dieser für diese Runde aussetzen.

Um die Spielerfigur zu bewegen, wird in die Nähe der gewünschten Zielstation geklickt. Kann die Zielstation durch mehrere Verkehrsmittel erreicht werden, wird ein Dialog zur Auswahl des gewünschten Tickets geöffnet. (siehe Abb. 2.10)

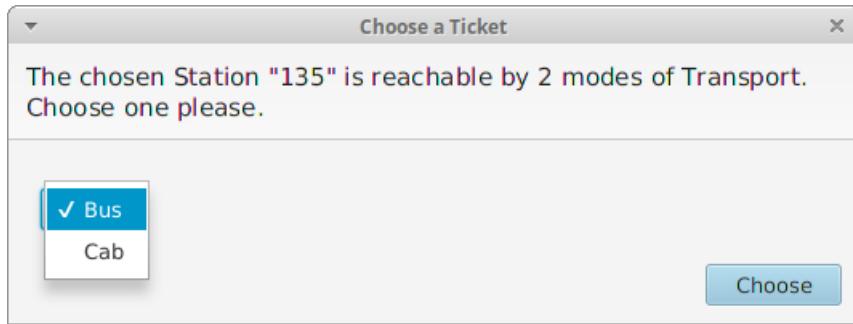


Abbildung 2.10: Dialog: Ticketauswahl

Falls auf eine Station gezogen werden soll die nicht erreichbar bzw. blockiert ist, oder der Spieler nicht genügend Tickets übrig hat, erscheint eine entsprechende Meldung. (siehe folgende Abbildungen)

## 2 Benutzerhandbuch

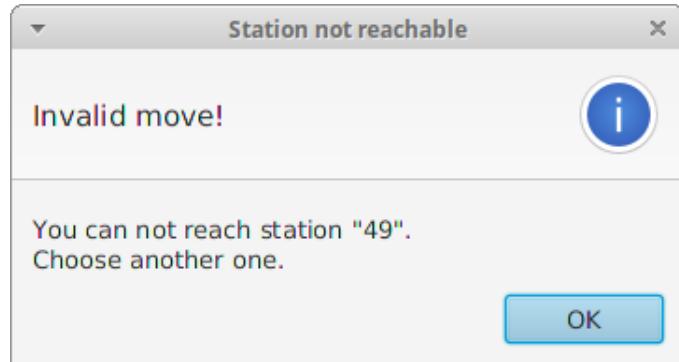


Abbildung 2.11: Meldung: Station nicht erreichbar

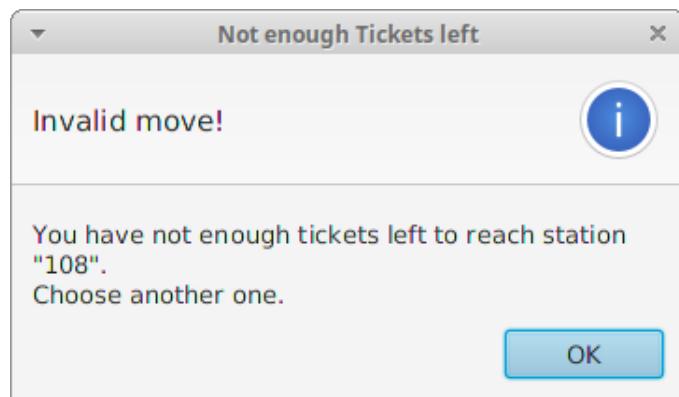


Abbildung 2.12: Meldung: Ungenügend Tickets

### 2.3.4 Spiel speichern

Um ein laufendes Spiel zu speichern, wird auf den Menüpunkt *Save* des Oberpunktes *Game* geklickt. Dadurch wird ein Datei-Navigator zur Auswahl des Speicherortes und des Namens des Spielstandes geöffnet. Hierbei ist zu beachten, dass die Datei die Endung *.sy* beinhalten muss. Die Endung wird nicht automatisch hinzugefügt, wird jedoch für das Laden eines Spielstandes benötigt.

Falls noch kein Spiel gestartet worden ist, schlägt das Speichern mit einer entsprechenden Meldung fehl.

## 2 Benutzerhandbuch

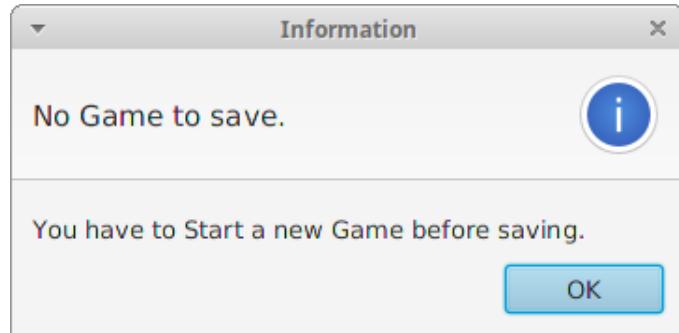


Abbildung 2.13: Meldung: Speicherung eines nicht existierenden Spielstandes

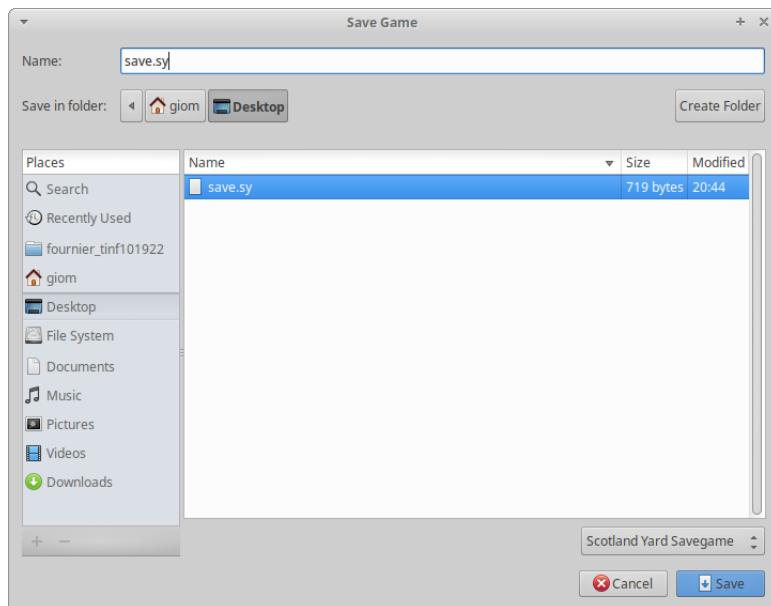


Abbildung 2.14: Datei Auswahl zum speichern des Spielstandes

Falls die Datei schon existiert und damit eine Überschreibung eines Spielstandes droht, öffnet sich eine Meldung.

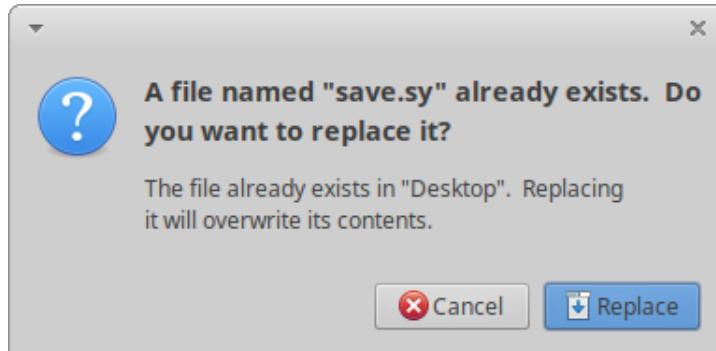


Abbildung 2.15: Meldung: Überschreibung eines Spielstandes

### 2.3.5 Spiel laden

Um ein Spiel zu laden, wird auf den Menüpunkt *Load* des Oberpunktes *Game* geklickt. Dadurch öffnet sich ein Datei-Navigator der dazu auffordert eine Datei auszuwählen aus der, der Spielstand geladen werden soll. Dabei ist zu beachten, dass der Datei-Navigator nur Dateien einblendet die eine Endung **.sy** beinhalten. Falls zur Zeit des Ladens ein Spiel am laufen ist, erscheint eine Meldung, dass das laufende Spiel dabei verloren gehen wird.

## 2 Benutzerhandbuch

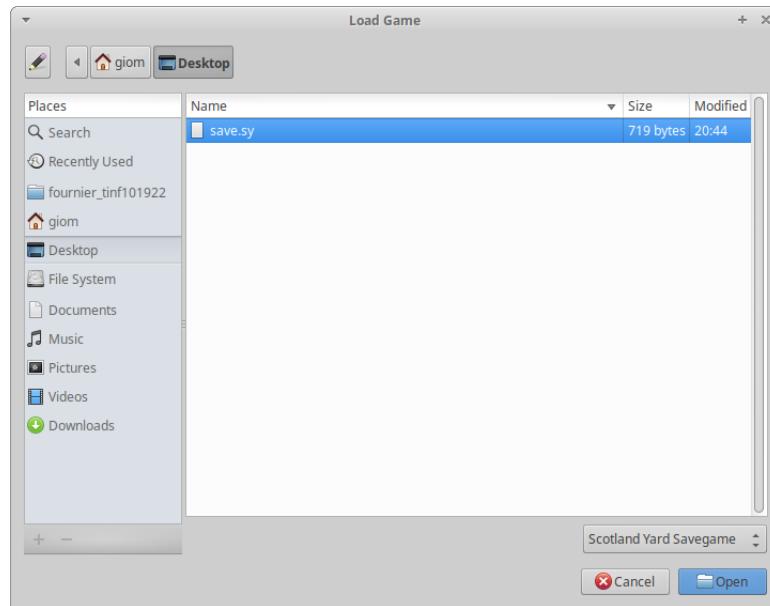


Abbildung 2.16: Datei-Navigator zum laden eines Spielstandes

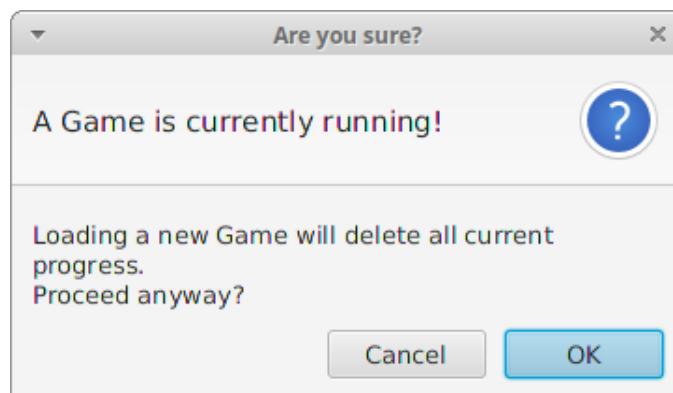


Abbildung 2.17: Meldung: Der aktuelle Spielstand wird gelöscht

### 2.3.6 Spiel beenden

Um ein Spiel zu beenden, wird auf den Menüpunkt *Close* des Oberpunktes *Game* geklickt. Falls zur Zeit des Beendens ein Spiel am laufen ist, erscheint eine Meldung, dass das laufende Spiel dabei verloren gehen wird.

### 2.3.7 Godmode

Beim Klicken auf den Menüpunkt *Godmode* des Oberpunktes *Edit*, wird der Godmode aktiviert. Dadurch ist Mister-X auch ausserhalb seiner Zeigerunden sichtbar. Durch erneutes Klicken auf den Menüpunkt wird dieser Modus deaktiviert.

### 2.3.8 Spiellog

Um das Spiel im nachhinein analysieren zu können, schreibt das Programm alle nötigen Informationen in eine Datei namens *game.log*. Diese Datei wird vor jedem Spiel automatisch angelegt oder überschrieben falls diese noch nicht existiert. Dabei wird die Struktur im Kapitel 2.5 erläutert.

## 2.4 Fehlermeldungen

Das Programm zeigt im Falle eines Fehlers folgende Fehler an.

Folgende Abbildung zeigt die Fehlermeldung die erscheint, wenn das zu ladende Spielfeld fehlerhaft formatiert ist. Um den Fehler zu beseitigen muss sichergestellt werden, dass die Spielfelddatei eine valide *JSON*-Struktur ist und diese alle Felder besitzt die in 2.5 beschrieben sind. Zusätzlich müssen die Datentypen stimmen.

## 2 Benutzerhandbuch

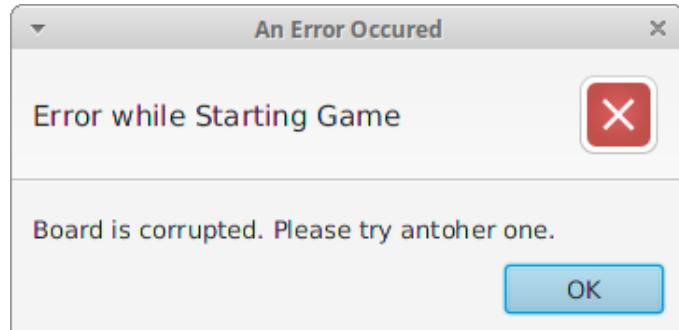


Abbildung 2.18: Fehler: Spielfeld falsch formatiert

Folgende Abbildung zeigt die Fehlermeldung die erscheint wenn das zu ladene Spielfeld fehlerhafte Werte hält. Um den Fehler zu beseitigen muss sichergestellt werden, dass die Werte korrekt sind. Dazu zählen, Stationsidentifikationsnummern die auch wirklich existieren.

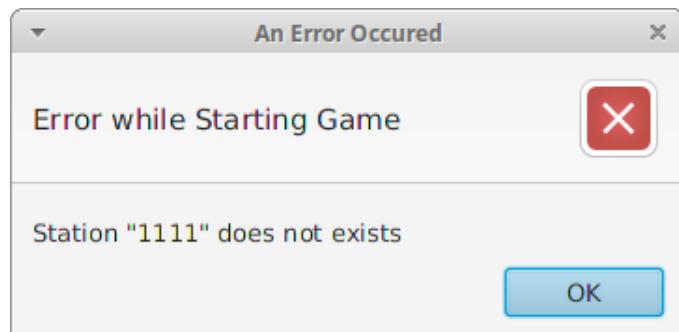


Abbildung 2.19: Fehler: Spielfeld beinhaltet ungültige Werte

Folgende Abbildung zeigt die Fehlermeldung die erscheint wenn das zu ladene Spielfeld nicht existiert. Um den Fehler zu beseitigen muss sichergestellt werden, dass die Datei im *JAR* unter dem Ordner *Logik/Board* liegt und *network.json* heißt.

## 2 Benutzerhandbuch

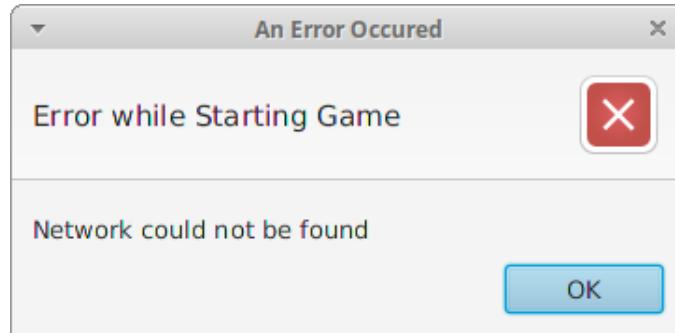


Abbildung 2.20: Fehler: Spielfeld nicht gefunden

Folgende Abbildung zeigt die Fehlermeldung die erscheint wenn die zu ladene Spielstandsdatei fehlerhafte Formatiert ist. Um den Fehler zu beseitigen muss sichergestellt werden, dass die Spielstandsdatei eine valide *JSON*-Struktur ist und diese alle Felder besitzt die in 2.5 beschrieben sind. Zusätzlich müssen die Datentypen stimmen.

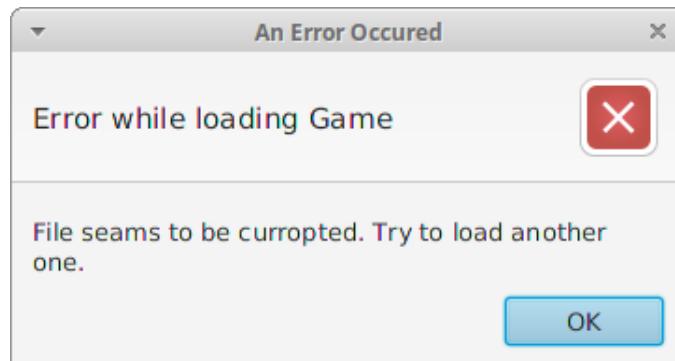


Abbildung 2.21: Fehler: Spielstandsdatei falsch formatiert

Folgende Abbildung zeigt die Fehlermeldung die erscheint wenn das zu ladenne Spielstandsdatei fehlerhafte Werte hält. Um den Fehler zu beseitigen muss sichergestellt werden, dass die Werte korrekt sind. Dazu zählen, keine Negative Tickets und Stationsidentifikationsnummern die auch wirklich existieren.

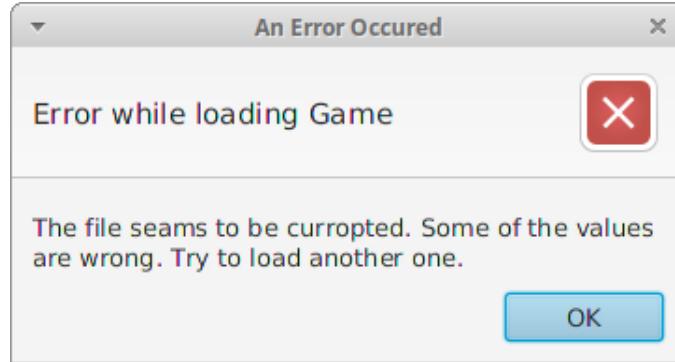


Abbildung 2.22: Fehler: Spielstandsdatei beinhaltet falsche Werte

## 2.5 Datenstrukturen

### Spielfeldsdatei

Die Datei indem das Spielfeld gespeichert ist, ist im *JSON*-Format kodiert. Im Grunde beinhaltet diese ein homogenes *Json*-Array aus *Json*-Objekten. Dabei bilden die Objekte eine Station mit Identifikationsnummer, Position und verfügbaren Nachbarstationen, erreichbar über ein Verkehrsmittel. Die Datenstruktur sieht dabei wie folgt aus:

- stations - Array aus folgenden Objekt
  - identifier
  - position
    - \* x: Koorinate als double
    - \* y: Koorinate als double
  - tube: Ein Array aus Identifikationsnummern.
  - bus: Ein Array aus Identifikationsnummern
  - cab: Ein Array aus Identifikationsnummern
  - boat: Ein Array aus Identifikationsnummern

### Spielstandsdatei

Die Datei indem der Spielstand gespeichert ist, ist im *JSON*-Format kodiert. Dabei beinhaltet die Datei folgende Felder:

## 2 Benutzerhandbuch

- MisterX: Object
  - ai: boolean
  - possibleTargets: int-Array
  - lastShownPos: int
  - currPos: int
  - remainingTickets: int-Array
  - journeyBoard: int-Array
- detectives: Object-Array
  - noOfDetectives: int
  - ai: boolean
  - players: Object-Array in der länge von *noOfDetectives*
    - \* position: int
    - \* remainingTickets: int-Array
- whosTurn: int
- currRoundNo: int
- gameIsWon: boolean

### Spiellog

Die logdatei ist in einer Struktur aufgebaut die, die Werte durch Kommata trennt. Dabei stellt die erste Zeile die Spielkonfiguration dar.

- die Anzahl der Detektive (ganze Zahl)
- ob MisterX von der KI gesteuert wird (true oder false)
- ob die Detektive von der KI gesteuert werden (true oder false)
- die Startposition von MisterX
- die Startpositionen aller teilnehmenden Detektive

Die darauffolgenden Zeilen repräsentieren ein Zug durch einen Spieler:

- welcher Spieler zieht (ganze Zahl ab 1 für die Detektive, 0 für MisterX)
- bisherige Station

## *2 Benutzerhandbuch*

- neue Station
- restliche Tickets für U-Bahn, Bus, Taxi, Boot
- welche der Taktiken wurde gewählt (bei einem menschlichen Spielzug = 0)
- die Bewertung der Taktik

Die letzte Zeile besagt, ob und von wem ein Spiel gewonnen wurde. Dabei steht die 0 für Mister-X und die 1 für die Detektive.

## 3 Programmierhandbuch

### 3.1 Entwicklungskonfiguration

Das Programm *Scotland Yard* wurde mit hilfe folgenden Komponenten entwickelt.

Tabelle 3.1: Programmanforderung

Software	Version
Betriebssystem	Manjaro Linux - Kernel 5.3.18
Java Development KIt	OpenJDK 1.8
JavaFX	8.0
Software zur Erstellung der GUI	Scene Builder 8.4.1
IDE	Netbeans 8.2

## 3.2 Problemanalyse und Realisation

In diesem Abschnitt werden Probleme aus der Aufgabenstellung analysiert, die daraus resultierenden Überlegungen dargestellt und die finalen gewählten Lösungen begründet.

### 3.2.1 Spielfeld

Das Spielfeld von *Scotland Yard* besteht aus 199 Stationen die jeweils über Verkehrsmittel verbunden sind. Dabei gibt es Taxi-, Bus-, U-Bahn- und Bootsverbindungen. In diesem Netz können sich die Spieler mit Hilfe ihrer Verkehrsrestickets bewegen.

#### Parsen des Spielfeldes

Das Spielfeld welches durch die Aufgabenstellung vorgegeben ist und in einer JSON-Datei abgelegt ist, wird für jeden Programmstart benötigt. Durch parsen der JSON-Datenstruktur werden benötigte Spielfeldinformationen extrahiert und in interne Repräsentationen übersetzt. Dabei besteht die JSON-Datenstruktur aus mehreren Stationen die untereinander durch die Verkehrsmittel verbunden sind. Jede Station besitzt zusätzlich eine eindeutige Identifikationsnummer und eine Position aus X- und Y-Koordinaten.

Der Vorgang des parsen wird über eine Bibliothek namens *GSON* realisiert. Diese ermöglicht es die JSON-Datenstruktur ähnlich wie einen Baum zu durchlaufen oder komplett Strukturen direkt in eine interne Repräsentation zu übersetzen.

Durch die jeweilige Verbindungen der Stationen untereinander entsteht eine Netzstruktur.

Der erste Ansatz ist die JSON-Datenstruktur zu durchlaufen und die Stationen direkt zu erstellen und diese über ihre Identifikationsnummer zu Verknüpfen. Der Vorteil hierbei ist, dass die JSON-Datenstruktur nur einmal durchlaufen werden muss. Ein offensichtlicher Nachteil ist jedoch, dass das Netz nicht ohne die dahinter liegende interne Struktur durlaufen werden kann. Das heißt, dass von einer Station nicht direkt auf eine andere referenziert werden kann.

Ein anderer Ansatz ist es, die interne Repräsentation so aufzubauen, dass eine Station direkt auf all ihre Nachbarn zeigt. Auf der einen Seite ist somit sichergestellt, dass von jeder Station aus das ganze Netz durlaufen werden kann. Auf der anderen Seite muss nun die Json-Datenstruktur zweimal durchlaufen

werden. Einmal um alle Stationen und somit ihre Referenzen zu erstellen. Das zweite Mal um die Stationen miteinander zu verbinden.

Obwohl der zweite Ansatz das parsen der JSON-Datenstruktur komplexer macht, wurde dieser, für die interne Repräsentation des Spielfeldes, gewählt da somit alle Operationen auf dem Netz enorm vereinfacht werden.

Nach dem Parsen hält die Klasse *Board* nun alle Stationen in einer *ArrayList*. Somit kann effizient, mit hilfe der Identifikationsnummer einer Station, indiziert auf eine Station zugegriffen werden.

**Validieren der Datenstruktur** Da beim Parsen der *JSON*-Datenstruktur Fehler auftreten können, muss diese zunächst validiert werden. Dabei wird zwischen Speicherstrukturfehlern, die also die formale Korrektheit des *JSON*-Formats betreffen und den inhaltlichen Fehlern, die sich in ungültigen Werten wiederspiegeln, unterschieden. Die *GSON*-Bibliothek fängt dabei die formalen Fehler ab und schmeißt in solch einem Fall eine *JsonSyntaxException*. Die statischen Methoden der *Json Validator*-Klasse überprüfen hingegen die Korrektheit der inhaltlichen Struktur und schmeißt eine *IllegalArgumentException* falls ein Feld oder der Datentyp des Feldes falsch ist.

Um ein fehlerhaftes Spielfeld zu unterbinden wird zusätzlich geprüft, ob die verknüpften Stationen auch wirklich existieren. Dabei wird im Fehlerfall eine *IllegalArgumentException* geworfen.

#### Stationen

Die Stationen sind als Knotenpunkte und deren Verbindungen als Kanten innerhalb des Netzes zu sehen. Da eine Station somit auf all ihre direkten Nachbarn referenziert, kann von einer Station, über Zwischentationen, jede andere Station erreicht werden.

**Occupied-Flag** Eine Station bietet jedoch immer nur Platz für einen Spieler. Daraus folgt, dass eine Station besetzt sein kann und dass kein anderer Spieler die Verbindungen der besetzten Stationen nutzen kann. Dies wird über ein boolesches Flag (*Occupied*-Flag) innerhalb der *Station*-Klasse erreicht. Somit ist die information ob eine Station besetzt ist, in dem Netz enthalten.

Dies hat den Vorteil, dass ein Spieler die anderen Spielerpositionen nicht kennen muss um sich Regelkonform durch das Netz zu bewegen. Daraus folgt auch, dass jeder Spieler der auf eine Station ziehen möchte, zunächst das Flag testen muss. Zusätzlich muss jeder Spieler nach jedem erfolgreichen Zug das Flag in der Zielstation auf *true* und in der verlassenen Station auf *false* setzen. Zieht jedoch ein Spieler auf eine besetzte Station und versucht das *Occupied*-Flag zu

### 3 Programmierhandbuch

setzen, wird eine *IllegalStateException* geworfen. Somit können Programmierfehler leicht erkannt werden.

**Traversieren des Netzes** Da die *Station*-Klasse zuständig ist, um sich durch das Netz zu bewegen bzw. Nachbarstationen zu erhalten, spielt diese eine große Rolle für viele Algorithmen. Deshalb wurden grundsätzlich zwei Verfahren gewählt um sich durch das Netz zu bewegen.

Die eine Methode *getStationsReachableBy* gibt alle Stationen wieder die über ein bestimmtes Ticket erreichbar sind. Falls keine Station über dieses Ticket erreichbar ist, wird ein leeres *Set* zurück gegeben.

```
Set<Station> getStationsReachableBy(Ticket ticket)
```

Dabei ist zu beachten, dass aus der Aufgabenstellung hervorgeht, dass das *Blackticket* ein Allzweckticket ist. Somit werden alle Nachbarstationen zurück gegeben falls diese Methode mit dem *Blackticket* aufgerufen wird.

Die andere Methode *getTicketsToReachableStation* gibt das Ticket wieder welches benötigt wird um sich zu einer bestimmten Station zu bewegen. Falls die übergebene Station nicht über die Verbindungen dieser Station erreichbar ist, wird ein leeres *Set* zurück gegeben.

```
Set<Ticket> getTicketsToReachableStation(Station station)
```

Auch hierbei ist zu beachten, dass das *Blackticket* jedes Mal mit zurückgegeben wird sobald die übergebenen Station eine Nachbarstation ist.

#### 3.2.2 Spieler

Das wohl wichtigste Spielement ist der Spieler und seine Funktionen sich durch das Netz zu bewegen. Dabei wird zwischen Mister-X und einem Detektiv als Spielertypen unterschieden. Jeder der beiden Typen nutzt dabei allgemeine aber auch spezialisierte Methoden um am Spielgeschehen teilzunehmen.

#### Obertyp

Allgemeine Funktionen wie die Ticketverwaltung, die Position, sowie die Bewegung durch das Netz und einige KI-Anteile sind in einem abstrakten Obertypen namens *Player* zusammengefasst. Dadurch lässt sich ein allgemeiner Spielertyp modellieren und Spezialisierungen in den jeweiligen Unterklassen formulieren.

**Ticket** Da alle Spieler Tickets brauchen um sich durch das Netz zu bewegen, sind diese sowie ihre Verwaltung innerhalb der Oberklasse angesiedelt. Konkret ist ein Ticket ein *Enum* und besteht aus den Werten *CAB*, *BUS*, *TUBE* und *BLACK*. Nun gibt es mehrere Ansätze die Tickets zu speichern und den Zugriff auf diese zu realisieren.

Ein Ansatz wäre es für jede Ticketart eine eigene Variable zu benutzen. Dies verkompliziert jedoch den Zugriff auf die Tickets massiv, da jeder Zugriff in langen *Switch*-Konstrukten resultiert. Zusätzlich muss für jedes neue Ticket ein *Switch-Case*, in jeder Methode die die Tickets verarbeitet hinzugefügt werden. Ein alternativer Ansatz wäre es die Tickets in einem *Array* zu speichern. Dies hat den Vorteil, dass direkt über den Ordinalwert des Ticketenums, indiziert auf das *Array* zugegriffen werden kann. Neu hinzugefügte Tickets sind somit wartbarer.

Somit kann das Array dynamisch aus den Tickets konstruiert werden.

```
new int[Ticket.values().length]
```

Aus diesen Gründen wurde für die Implementierung der Tickets der zweite Ansatz gewählt.

**Aktuelle Position** Jeder Spieler hält seine aktuelle Position, das heißt die Station auf der er sich befindet. Somit hat jeder Spieler einen Einstiegspunkt in das Netz. Alle Methoden wie z.B. die Überprüfung ob eine Station erreichbar ist oder welche Tickets benötigt werden um eine Station zu erreichen, arbeiten somit direkt mit diesem Einstiegspunkt.

**Erreichbare Stationen** Eine grundsätzliche und für fast alle Algorithmen wiederkehrende Problematik ist das Bestimmen der erreichbaren Stationen. Eine erreichbare Station ist jene, die über eine Verbindung der aktuellen Station erreichbar ist. Als Einschränkungen müssen jedoch das *Occupied*-Flag und die verfügbaren Tickets beachtet werden. Um alle erreichbaren Stationen eines Spielers zu bekommen, kann somit auf die *getReachableStations*-Methode zurückgegriffen werden.

---

```
public Set<Station> getReachableStations() {
    return Arrays.stream(Ticket.values())
        .filter(ticket -> this.getTicketNum(ticket) > 0)
        .map(ticket -> this.currentStation.getStationsReachableBy(ticket))
        .flatMap(Collection::stream)
        .distinct()
        .filter(station -> !station.isOccupied())
```

### 3 Programmierhandbuch

```
    .collect(Collectors.toCollection(HashSet::new));
}
```

---

Dabei wird zunächst ein *Stream* aus allen Tickets erstellt. Für jedes einzelne Ticket wird nun ermittelt, ob der Spieler noch welche vorhanden hat.

```
this.getTicketNum(ticket) > 0
```

Falls dies der Fall sein sollte, werden alle Stationen, die von der aktuellen Station über das Ticket erreichbar sind, ermittelt. Nun werden im letzten Schritt alle diese Stationen gefiltert und nach folgender Bedingung gefiltert:

```
!station.isOccupied()station.is
```

Übrig bleiben diejenigen Stationen die über die verfügbaren Tickets erreichbar sind und nicht das *Occupied*-Flag gesetzt haben.

**Bewegung** Dadurch, dass alle Spieler sich nach festen Regeln durch das Netz bewegen und um sicherzustellen, dass alle Spieler dies auch einhalten, liegt es nahe diese Bewegungsmethoden innerhalb der Oberklasse zu implementieren. Eine regelkonforme Bewegung besteht aus:

1. Überprüfen, ob genügend Tickets übrig sind um die Zielstation zu erreichen
2. Prüfen des *Occupied*-Flags der , ob schon besetzt
3. Setzen des *Occupied*-Flags der aktuellen Station auf *false*
4. Setzen des *Occupied*-Flags der Zielstation *true*
5. Abziehen des genutzten Tickets

Hat ein Spieler nicht genügend Tickets übrig, oder ist die Zielstation schon besetzt, darf dieser nicht auf die Station ziehen. Das heißt im Extremfall gibt es keinen gültigen Zug für einen Spieler.

**Wegfindung** Zunächst wird ein Netz durch ein *Array* von *StationDistance* aufgebaut. Die *StationDistance*-Klasse repräsentiert generell den Abstand zu einer Station, in diesem konkreten Fall, den Abstand von der Startstation zur Station mit der Identifikationsnummer am jeweiligen *Array*-Index (Identifikationsnummer - 1). Wurde die Station mit einer bestimmten Identifikationsnummer noch nicht besucht, ist der Eintrag am Index *null*.

### 3 Programmierhandbuch

Von der Startstation wird nun das *Array* gefüllt. Dabei werden, ausgehend von der Startstation, alle Nachbarstationen ermittelt die erreichbar sind.

Zunächst wird überprüft, ob die aktuelle Station mit der Zielstation übereinstimmt. Ist dies der Fall, ist der Algorithmus fertig. Als nächstes werden alle Werte des *Ticketenums* durchchiteriert. Nun werden, ausgehend von der aktuellen Position, die Nachbarstationen ermittelt die durch dieses Ticket erreichbar sind. Gibt es keine Nachbarstationen oder hat der Spieler keine Tickets dieser Art mehr übrig, wird das nächste Ticket betrachtet. Falls der Spieler noch genügend Tickets übrig hat, wird ein Ticket dieser Art aus seinem Vorrat genommen. Die somit gewonnenen Nachbarstationen werden nun durch zwei Bedingungen gefiltert. Der eine Filter betrachtet das *Occupied-Flag*. Ist dieses gesetzt, wird die Station nicht betrachtet da keine Verbindungen über sie genutzt werden kann. Der zweite Filter filtert, ob die zu betrachtende Nachbarstation schonmal besucht worden ist. Falls dies nicht der Fall sein sollte, gibt es keine weiteren Prüfungen. Falls die Station jedoch schonmal besucht worden ist, wird zunächst ermittelt, ob die aktuelle Distanz (rekursionstiefe) größer ist als zu dem Zeitpunkt als die Station das letzte Mal besucht worden ist. Sollte dies zutreffen, wird die Nachbarstation ausgetiltert da sonst ein potentiell kürzerer Weg überschrieben wird. Ist jedoch die aktuelle Distanz kleiner, wird die Nachbarstation wieder in Betracht gezogen. Für jede übriggebliebene Station wird nun der Eintrag, der aus der aktuellen Distanz (rekursionstiefe) und der Vorgängerstation besteht, im *Array* erstellt bzw. überschrieben. Zusätzlich wird nun für jede übriggebliebene Nachbarstation die Rekursion weiter getrieben. Dazu werden diese, nacheinander, als aktuelle Station gesetzt und die Funktion mit den übriggebliebenen Tickets sowie der neuen Distanz (alte Distanz + 1) wird erneut aufgerufen.

Die Abbruchbedingungen für die Rekursion sind somit die, ob die Zielstation erreicht worden ist oder dem Spieler die Tickets ausgehen.

Nach beendigung der Rekursion durch eine der Abbruchbedingungen, liegt nun ein Graph vor aus dem noch der kürzeste Weg extrahiert werden muss. Dazu wird bei der Zielstation begonnen. Ist diese *null*, wurde diese nie besucht, was bedeutet, dass es keinen Weg zu extrahieren gibt. Falls es jedoch einen kürzeren Weg gibt, wird ausgehend von der Zielstation jeweils der Vorgänger betrachtet bis die Startstation erreicht ist. Dabei werden alle Zwischenstationen vorne in eine *LinkedList* eingefügt. Somit ist, obwohl hinten angefangen wird, die Reihenfolge wieder gegeben. Zu beachten ist jedoch, dass die Startstation nicht mit in den Weg aufgenommen wird.

### 3 Programmierhandbuch

**KI** Jeder Spieler besitzt ein boolesches Flag *isAi* welches aussagt, ob dieser Spieler von der KI oder einem menschlichen Spieler gesteuert werden soll. Alternativ könnte die Information, ob die Detektive oder Mister-X durch die KI gesteuert wird, in der Logik gespeichert werden. Ist nun das Flag gesetzt, wird in jedem Rundendurchlauf die *play*-Methode auf dem Spieler aufgerufen. Dieser Ansatz wurde jedoch nicht gewählt da mit dem erst genannten Ansatz z.B. zwei Detektive durch die KI und einer durch einen menschlichen Spieler gesteuert werden könnte. In dieser Methode berechnet der Spieler dann seinen besten Zug und spielt diesen. Im Grunde besteht die KI aus verschiedenen Taktiken und deren anschließende Bewertung. Die Taktiken wählen dabei, unter Berücksichtigung bestimmter Faktoren, eine Zielstation aus. Diese Zielstation wird dann bewertet. Die bestbewertete Taktik wird schlussendlich gewählt. Die Implementierung der jeweiligen Taktiken sowie der Bewertungen wird den Unterklassen überlassen.

Die Vorgehensweise in beiden Unterklassen *MisterX* und *Detektiv* ist dabei identisch. Zunächst werden alle Taktiken berechnet und diese dann einzeln bewertet. Dazu zieht der Spieler temporär auf die Station die die Taktik vorschlägt und zieht das entsprechende Ticket ab. Nun werden alle Bewertungsfunktionen angewandt und summiert. Sobald das Ergebnis fest steht, wird der Spieler wieder auf die Ausgangsstation gesetzt und das abgezogene Ticket wieder dazu addiert. Dies wird für alle verfügbaren Taktiken angewandt und das jeweilige Ergebnis in einer List von *TacticResult* gespeichert. Dabei speichert *TacticResult* jeweils die Bewertung der Taktik sowie den Spielzug als *Move*. Die Auswertung der besten Taktik kann nun anhand der Liste stattfinden, wobei die beste Bewertung ausgewählt wird.

#### Mister X

Als Mister-X wird die Unterkasse *MisterX* verstanden die die abstrakte Oberklasse *Player* erweitert. Somit implementiert diese alle Spezialisierungen die benötigt werden um Regelkonform als Mister-X am Spielgeschehen teilzunehmen.

**Bewegung** Da die Detektive das *Occupied*-Flag einer Station testen bevor sie sich bewegen, darf Mister-X dieses nicht setzen sobald er auf eine Station zieht. Würde er dies trotzdem tun, würden die Detektive nie auf die Station von Mister-X ziehen (ihn fangen). Aus diesem Grund wird das *Occupied*-Flag bei der Bewegung von MisterX nicht gesetzt.

### 3 Programmierhandbuch

**Letzte Zeigeposition** Mister-X ist für die Detektiven die meiste Zeit unsichtbar. Somit wissen diese, ob durch KI oder durch einen menschlichen Spieler gesteuert, nicht wo sich Mister-X zur Zeit befindet. Eine Ausnahme sind die Runden 3, 8, 13, 18 und 24. In diesen *Zeigerunden* erscheint Mister-X und bleibt bis zu seinen nächsten *Zeigezug* an der Position sichtbar. Die Detektive müssen also die letzte Zeigeposition und das Fahrtenbuch nutzen um Mister-X zu fangen.

**Fahrtenbuch** Das Fahrtenbuch ist eine Besonderheit von Mister-X. Dort werden alle seine verbrauchten Tickets angezeigt.

Konkret ist das Fahrtenbuch eine *LinkedList* von Tickets. Wenn Mister-X am Zug ist, wird das verwendete Ticket der Liste hinzugefügt. Da die Detektive die verbrauchten Tickets, ausgehend von der letzten Zeigeposition, bis zur aktuellen Runde benötigen, kann nun mit einer *Sublist* gearbeitet werden. Dazu wird die Runde der letzten Zeigeposition als Startindex für die *Sublist* zwischengespeichert. Als Endindex wird die Länge der *LinkedList* benutzt.

**KI** Die Taktiken von Mister-X sind durch die Aufgabenstellung klar vorgegeben. Aus dieser geht hervor, dass Mister-X nach folgender Taktik spielt:

1. bewege Dich auf eine Station, die möglichst wenige Spieler im nächsten Zug erreichen können. Hierfür werden bei Mister X einfach alle erreichbaren Stationen durchgetestet

Für diese Taktik werden zunächst alle erreichbaren Stationen in Betracht gezogen und durch folgenden Bewertungsfunktion die die Aufgabenstellung vorgibt bewertet:

1. wie viele Detektive können die Position von Mister X im nächsten Zug erreichen.
2. wie viele Stationen kann Mister X von der aktuellen Position aus mit den vorhandenen Tickets in einem Zug erreichen, auf denen kein Detektiv steht
3. was ist die geringste Ticketanzahl für ein Verkehrsmittel

Für die erste Bewertungsfunktion werden in der *getRankingByReachableTargetStations*-Methode alle Detektive durchiteriert und die Schnittmenge der erreichbaren Stationen mit der aktuellen Position von Mister-X gebildet. Ist diese resultierende Menge nicht leer, kann dieser Detektiv Mister-X in der nächsten Runde erreichen. Gezählt wird nun wie viele Detektive Mister-X erreichen können.

```
float getRankingByReachableTargetStation(List<Detective> detectives)
```

Die zweite und dritte Bewertungsfunktion sind trivial da hierfür die Bewertungsfunktion der Oberklasse benutzt werden kann.

```
float getRankingByDirectlyReachableStations()
```

```
int getRankingBySmallestTicketAmount()
```

### Detektiv

Als Detektiv wird die Unterklasse *Detective* verstanden die die abstrakte Oberklasse *Player* erweitert. Somit implementiert diese alle Spezialisierungen die benötigt werden um Regelkonform als Detektiv am Spielgeschehen teilzunehmen.

**Black Ticket** Durch die Aufgabenstellung geht hervor, dass Mister-X der einzige Spieler ist der die *Blacktickets* verwenden darf. Dies wird durch den Konstruktor des Detektiven sichergestellt, da dieser keinen Parameter für dieses Ticket erlaubt.

**Mögliche Zielpositionen** Um die möglichen Zielpositionen zu berechnen wird die letzte Zeigeposition von Mister-X genommen und mit Hilfe des Fahrtenbuchs rekursiv die Stationen ermittelt. Dies wird über die statische Methode innerhalb der *Detective*-Klasse erreicht:

```
Set<Station> getPossibleTargetPositions(LinkedList<Ticket> logbook,  
Station lastStation, Set<Station> targetStations
```

Hierfür wird eine *LinkedList* benötigt, da mit der speziellen *pollFirst*-Methode dieser Liste gearbeitet wird um effizient das erste Ticket in der Liste zu ermitteln und dabei die Liste zu verkleinern.

Um nun die möglichen Zielpositionen zu ermitteln, wird beim ersten Ticket angefangen und über die *getStationsReachableBy*-Methode der letzten Zeigeposition, die Nachbarstationen ermittelt die mit diesem Ticket angefahren werden konnten. Die somit gewonnene Stationen werden zunächst in die *targetPositions* eingefügt. Zusätzlich wird jede ermittelte Station als letzte Zeigeposition gesetzt und die Rekursion weiter getrieben. Dabei beachtet der Algorith-

### 3 Programmierhandbuch

mus nicht das *Occupied*-Flag der Stationen da dieses das Ergebnis verfälschen würde.

**KI** Die Taktiken und Bewertungen für einen Detektiven sind durch die Aufgabenstellung klar vorgegeben. Dabei sind zwei wiederkehrende Probleme, die für jede Taktik relevant sind in *BinaryOperators* formuliert:

**Reduzierung auf die Station mit der kleinsten Identifikationsnummer** Wenn mehrere Stationen für eine Taktik in Frage kommen, fordert die Aufgabenstellung die Reduzierung der Stationensmenge auf die Station mit der kleinsten Identifikationsnummer. Die Bedingung für die Reduzierung der Menge ist somit in dem *filterStationsBySmallestId*-Binäroperator formuliert und steht allen Taktiken zu Verfügung.

**Reduzierung der Tickets** Da eine Station durch mehrere Tickets angefahren werden kann, müssen auch diese deterministisch reduziert werden. Dabei fordert die Aufgabenstellung eine Reduzierung nach der Anzahl verfügbarer Tickets. Dabei ist das Reduzieren nach der Ticketanzahl trivial. Falls jedoch ein Spieler von zwei Tickets die gleiche Anzahl hat, wird zusätzlich nach der Wertigkeit eines Tickets reduziert. Dabei ist die aufsteigende Reihenfolge der Wertigkeit Taxi, Bus, U-Bahn, Black.  
Die Bedingungen sind dabei in dem *filterTicketsByQuantityAndValue*-Binäroperator formuliert und stehen allen Taktiken zu Verfügung.

Aus der Aufgabenstellung gehen folgenden Taktiken hervor:

1. bewege Dich auf eine mögliche Zielposition. Gibt es mehrere erreichbare freie, wähle davon die mit der kleinsten Stationsnummer
2. bewege Dich auf eine direkt benachbarte U-Bahn-Station (um schnell zu anderen Standorten zu kommen). Gibt es mehrere erreichbare freie, wähle davon die mit der kleinsten Stationsnummer
3. bewege Dich in Richtung auf die letzte Zeigeposition von Mister X (s.u.). Benutze dafür den kürzesten (derzeit) freien Weg
4. bewege Dich auf diejenige direkt benachbarte freie Position mit der kleinsten Stationsnummer

Die erste Taktik bildet dabei die Schnittmenge der erreichbaren Stationen und der möglichen Zielstationen. Die entstandene Menge muss jedoch noch auf eine Station reduziert werden und die resultierende Station noch auf das beste Ticket.

### 3 Programmierhandbuch

```
Move getMoveToPossibleTargetPosition(Set<Station> targets)
```

Die zweite Taktik filtert die erreichbaren Stationen nach denen die eine U-Bahn-Verbindung besitzen. Dabei wird davon ausgegangen, dass wenn eine Station über ein U-Bahn-Ticket erreichbar ist, diese auch eine U-Bahnstation sein muss. Dabei werden wieder die Ergebnisstation durch die beiden genannten Bedingungen reduziert.

```
Move getMoveToDirectTubeStation()
```

Die dritte Taktik benutzt die Wegfindung der Oberklasse um sich in die Richtung der letzten Zeigeposition von Mister-X zu bewegen. Dabei wird nur nach der besten Ticketbedingung reduziert, da die Zielstation fester Bestandteil des kürzesten Weges ist.

```
Move getMoveInDirectionOfLastseenPosition(Station lastSeen)
```

Die vierte Taktik ist die Ausweichtaktik falls keine der anderen Taktiken funktioniert. Dabei wird auf die nächstgelegenen Station gezogen. Auch hier wird wieder nach den beiden Bedingungen reduziert.

```
Move getMoveToDirectReachableStation()
```

Für die Bewertungen die aus Aufgabenstellung hervorgehen:

1. wie viele mögliche Zielpositionen sind für alle Detektive im nächsten Zug erreichbar
2. wie weit ist die "mittlere mögliche Zielposition" (s.u.) entfernt
3. wie viele Stationen sind von der aktuellen Position aus mit den vorhandenen Tickets in einem Zug erreichbar
4. was ist die geringste Ticketanzahl für ein Verkehrsmittel

Die erste Bewertungsfunktion bildet die Schnittmenge der möglichen Zielpositionen von Mister-X und aller erreichbaren Stationen der Detektive. Die Bewertung wird wie folgt berechnet:

Anzahl an Stationen in der Schnittmenge dividiert durch die Anzahl an möglichen Zielpositionen multipliziert mit Zehn.

```
getRankingByReachableTargetStations(Set<Station> targets,  
List<Detective> detectives)
```

### 3 Programmierhandbuch

Die Zweite Bewertungsfunktion berechnet den kürzesten Weg von der aktuellen Position bis hin zur mittleren Zielposition. Die mittlere Zielposition ist dabei die Station die am nächsten an dem Mittelpunkt aller möglichen Zielpositionen liegt. Die Bewertung wird wie folgt berechnet:

Länge des Weges subtrahiert von Zehn oder Null falls Ergebnis Negativ werden sollte.

```
int getRankingByDistanceToStation(Station destination)
```

Die dritte Bewertungsfunktion betrachtet die erreichbaren Stationen. Die Bewertung wird wie folgt berechnet:

Anzahl an erreichbaren Stationen dividiert durch 13 und multipliziert mit vier.

```
float getRankingByDirectlyReachableStations()
```

Die vierte Bewertungsfunktion betrachtet die geringste Ticketanzahl. Die Bewertung wird wie folgt berechnet:

Wenn es noch mehr als 2 Tickets sind, ist die Bewertung 3, ansonsten Anzahl der Tickets.

```
int getRankingBySmallestTicketAmount()
```

#### 3.2.3 Spiellogik

Die Spiellogik ist das wohl wichtigste Teil des Programms. Hier werden alle Elemente wie Mister-X, die Detektive und das Spielfeld erzeugt. Zudem regelt die Spiellogik den korrekten Ablauf zwischen den Spielern, verarbeitet menschliche Eingaben und überprüft, ob ein Spieler gewonnen hat. Die folgenden Unterpunkte beleuchten dabei wesentliche Teile der Spiellogik.

##### Konfiguration

Dadurch, dass die Aufgabenstellung viele Parameter, wie z.B. die Starttickets der einzelnen Spieler, vorgibt müssen diese konstanten Werte irgendwo verwaltet werden. Denkbar ist es diese in der jeweiligen Klasse zu definieren in der sie benötigt werden. Ein Nachteil dieser Lösung ist jedoch, dass eine Änderung einer Konstanten zunächst bedeutet sie zu finden. Zusätzlich kommt, dass man-

### 3 Programmierhandbuch

che Konstanten in verschiedenen Klassen benötigt werden. Aus diesem Grund wurde ein Mechanismus gewählt der eine Klasse *Config* innerhalb der Spielelogik nutzt um alle Konstanten zentral zu verwalten. Somit sind alle Konstanten direkt ersichtlich und Änderungen können einfacher vorgenommen werden. Diese Klasse ist *static* und *final* und bietet keinerlei Logik sondern nur folgende Konstanten:

**WIDTH** Die Breite des Fensters beim Start

**HEIGHT** Die Höhe des Fensters beim Start

**FILE\_NAME** Der name der Logdatei

**BOARD\_SIZE** Die Größe des Spielfelds (Stationen)

**MAX\_ROUNDS** Die maximalen Spielrunden

**LAST\_SEEN\_ROUNDS** Die Runden inder sich Mister-X zeigt

**DETECTIVES\_NUMS** Anzahl an Detektiven die an dem Spiel teilnehmen können.

**TICKET\_CAB** Die Taxi-Starttickets

**TICKET\_BUS** Die Bus-Starttickets

**TICKET\_TUBE** Die U-Bahn-Starttickets

**TICKET\_BOAT** Die Boots/Black-Starttickets

**PLAYER\_SIZE** Die Spielergröße

**MAX\_DISTANCE** Die maximale Distanz zu einer Station in der ein Klick noch registriert wird

**START\_POSITIONS** Die Startpositionen aus denen Zufällig gewählt wird

**DELAY** Die Zeit in Millisekunden die die KI für ihren Zug verzögert

**NANO\_TO\_MILI\_FACTOR** Ein Faktor zur Umrechnung von Nano- zu Millisekunden

## Zustandsautomat

Die *Gamelogic*-Klasse erweitert die *AnimationTimer*-Klasse und überschreibt dabei die *start*- und *handle*-Methode. Die *start*-Methode ist dabei zuständig den Zustandsautomat zu initialisieren. Dabei wird eine *IllegalStateException* geworfen wenn der aktuelle Zustand nicht auf *STOPPED* gesetzt ist. Somit wird verhindert, dass dieser während seiner Laufzeit erneut gestartet wird. Wurde *start* jedoch erfolgreich ausgeführt, wird die *handle*-Methode kontinuierlich, mit einem Zeitstempel in Nanosekunden als Parameter *timeStamp*, aufgerufen.

```
void handle(long timeStamp)
```

Die Zustände des Automaten sind die Folgenden:

**STOPPED** Das Spiel ist angehalten. In diesem Zustand sind keinerlei Interaktionen mit der Logik möglich. Die Logik wartet auf den Aufruf der *start*-Methode und wechselt dann in den *NEXT\_TURN* Zustand.

**NEXT\_TURN** Die Logik ist bereit für den nächsten Spielzug. Falls der nächste Spieler ein menschlicher Spieler ist, wird in den *HUMAN\_PLAYING* Zustand gewechselt. Falls dieser von der KI gesteuert wird, wird in den *AI\_PLAYING* Zustand gewechselt.

**AI\_PLAYING** Die KI ist zurzeit am spielen. Es wird zunächst um *DELAY*-Millisekunden verzögert. Anschließend wird die *play*-Methode des Spielers aufgerufen. Sobald diese durchgelaufen ist, wird wieder in den *NEXT\_TURN* Zustand gewechselt.

**HUMAN\_PLAYING** Ein menschlicher Spieler ist zurzeit am Spielen. Es wird auf die Interaktion des Spielers gewartet und gegebenenfalls ein Dialog geöffnet.

**Spielerauswahl für den aktuellen Zug** Beim Umschalten in den *NEXT\_TURN* Zustand muss zunächst der nächste Spieler ermittelt werden. Da alle Spieler von der Oberklasse *Player* stammen, können diese, ob *Mister-X* oder Detektiv in einer Liste von Spielern gespeichert werden.

```
private final List<Player> players = new ArrayList<>()
```

Zusätzlich wird über *whosTurn* ein Index gespeichert. Nun kann mit dem Index und der Liste der Spieler ermittelt werden der aktuell dran ist. Nach jedem

### 3 Programmierhandbuch

Zug muss *whosTurn* dabei inkrementiert werden, sodass der nächste Spieler ermittelt werden kann. Hierbei wurde absichtlich eine *ArrayList* gewählt, da indizierte Zugriffe somit schneller sind. Da der aktuelle Spieler des Öfteren benötigt wird, wird dieser in eine Variable *turn* abgespeichert.

#### KI-Züge

**Verzögerung** Da die KI einen Zug sehr schnell berechnet, würden mehrere Züge von KI gesteuerten Spielern, als ein Zug wahrnehmbar sein. Aus diesem Grund bietet die Aufgabenstellung an, mittels *Sleep* eine gewisse Zeit zwischen den Zügen zu verzögern. Dieser Mechanismus ist zwar einfach implementiert, ist jedoch aus Programmiersicht eine unelegante Lösung.

Als gewählter Ansatz für die Verzögerung wurde deshalb Gebrauch des *timeStamp*-Parameters der *handle*-Methode gemacht. Dabei wird beim wechseln des Zustandes von *NEXT\_TURN* in *AI\_PLAYING* der aktuelle *timeStamp* in *turnBeginTimeStamp* zwischengespeichert. Nun wird bei jedem Aufruf der *handle*-Methode *turnBeginTimeStamp* mit dem aktuellen *timeStamp* verglichen. Beträgt die Differenz mehr als *DELAY* (siehe Konfiguration in 3.2.3) wird der Zug ausgeführt. Somit ist gewährleistet, dass die KI immer zumindest eine bestimmte Zeit wartet bevor sie den nächsten Zug ausführt.

**Züge verarbeiten** Sobald eine KI lange genug verzögert wurde, kann ihr eigentlicher Spielzug verarbeitet werden. Dies passiert in der *GameLogic*-Klasse in der *handleAiMove*-Methode. Dabei wird zunächst unterschieden, ob der Spieler der aktuell am Zug ist, Mister-X oder ein Detektiv ist. Ist Mister-X am Zug, müssen keine weiteren Informationen berechnet werden, da dieser nur die Detektive benötigt um seinen Zug zu berechnen. Zusätzlich muss das grafische Fahrtenbuch aktualisiert werden.

Ist der aktuelle Spieler jedoch ein Detektiv müssen die benötigten Informationen zunächst berechnet werden, die dann dem Detektiven über die *play*-Methode übergeben werden. Zusätzlich gibt die Aufgabenstellung vor, jedes verbrauchte Ticket der Detektiven, Mister-X gut zu schreiben.

#### Menschliche Züge

**Züge verarbeiten** Sobald in den *HUMAN\_PLAYING* Zustand gewechselt wurde, wartet die Spieldelogik auf eine menschlichen Eingabe. Dazu muss die *handleHumanPlaying*-Methode aufgerufen werden. Dies passiert durch den *FXML-Controller* sobald dieser ein Klicken auf das Spielfeld registriert. Der *position*-Parameter repräsentiert dabei die Koordinaten auf dem Spielfeld.

### 3 Programmierhandbuch

```
void handleHumanPlaying(Position position)
```

Wird diese Methode aufgerufen obwohl der Zustandsautomat sich zurzeit in einem anderem Zustand befindet, geschieht nichts. Dies soll verhindern, dass ein menschlicher Spieler Eingaben tätigen kann obwohl die KI zurzeit am Zug ist. Grundsätzlich besteht ein menschlicher Zug daraus eine Station auszuwählen auf die er ziehen möchte. Dabei muss sichergestellt werden, dass die Eingaben korrekt sind. Zunächst muss überprüft werden über welches Ticket die Zielstation erreichbar ist und ob der Spieler dieses Ticket noch besitzt. Dazu wird die Schnittmenge der verfügbaren Tickets des Spielers mit der Ticketmenge gebildet, die benötigt werden um die Station zu erreichen

```
availableTickets.retainAll(neededTickets)
```

Ist die resultierende Menge leer, kann der Spieler diese Station nicht erreichen und die *GUI* wird aufgefordert dem menschlichen Spieler eine entsprechende Meldung auszugeben. Falls jedoch die resultierende Menge mehr als einen Eintrag besitzt, das heißt die Station kann über mehrere Tickets erreicht werden, wird wieder die *GUI* beauftragt dem Spieler einen Dialog zu zeigen der ihn auffordert sich für ein Ticket zu entscheiden. Ist das Ticket ausgewählt, kann die Spielfigur auf die entsprechende Station gesetzt werden, gegebenenfalls wird das Fahrtenbuch von Mister-X aktualisiert bzw. das benutzte Ticket wird Mister-X gutgeschrieben. Abschließend wird der Zustand auf *NEXT\_TURN* gesetzt werden, damit der Zustandsautomat weiter schalten kann.

### Gewinnbedingungen

Nach den Regeln von *Scotland Yard* gibt es mehrere Gewinnbedingungen die es zu überprüfen gilt. Dabei können entweder Mister-X oder alle Detektive das Spiel gewinnen. Vor jedem Zug wird überprüft, ob der vorherige Zug zum Sieg geführt hat. Alle Bedingungen werden in der *isGameWon*-Methode nacheinander geprüft, und der jeweilige Status zurück gegeben.

**Mister-X** Mister-X kann durch zwei Bedingungen gewinnen. Die erste ist, dass die letzte Runde erreicht wurde. Diese Bedingung ist dabei sehr trivial und wird in der *isLastRound*-Methode geprüft. Dabei ist die Bedingung wie folgt formuliert:

```
Config.MAX_ROUNDS <= this.gameRound
```

Die zweite Bedingung, ob die Detektive keine Tickets mehr haben, wird in

### 3 Programmierhandbuch

der *areAllDetectivesBlocked*-Methode geprüft. Dabei wird aus den Detektiven ein *Stream* gebildet und mit einem *AnyMatch* folgende Bedingung geprüft:

```
detective.getAvailableTickets().isEmpty()
```

Falls eine dieser Bedingung zutrifft, wird der *WinningState* auf *MISTERX\_WIN* gesetzt.

**Detective** Auch die Detectives können durch zwei Bedingungen gewinnen. Die erste ist dabei die, ob Mister-X gefangen worden ist. Konkret bedeutet das, dass ein Detective auf die Position von Mister-X gezogen ist. Dies ist möglich da Mister-X das *Occupied*-Flag nicht setzt. Die *isMisterXCatched*-Methode bildet dafür wieder ein *Stream* aus den Detectives und prüft mit einem *AnyMatch* folgende Bedingung:

```
this.misterX.getCurrentStation()  
.equals(detective.getCurrentStation())
```

Falls dies zutreffen sollte, wird der *WinningState* auf *MISTERX\_CATCHED* gesetzt. Als zweite Bedingung gilt es zu testen, ob Mister-X sich noch bewegen kann. Da Mister-X alle verbrauchten Tickets von den Detectives gutgeschrieben bekommt, reicht das Prüfen, ob er noch verfügbare Tickets besitzt nicht aus. Deshalb macht die *isMisterXSurrounded*-Methode gebraucht von der *getReachableStations*-Methode der *Player*-Klasse (siehe *Ereichbare Stationen* in 3.2.1). Daraus formuliert sich folgende Bedingung:

```
this.misterX.getReachableStations().isEmpty()
```

Falls die Bedingung zutrifft, wird der *WinningState* auf *MISTERX\_SURROUNDED* gesetzt.

#### 3.2.4 Speichern und Laden von Spielständen

Spielstände werden in einem JSON-Dateiformat, welches durch die Aufgabenstellung vorgegeben wird, abgespeichert. Dabei werden grundlegende Spielinformationen in einer Datei abgelegt. Durch das parsen dieser Datei zu einem späteren Zeitpunkt, können Spielstände wieder aufgenommen werden. Hierbei wird wieder die *GSON*-Bibliothek benutzt.

**Serialisieren von Spielständen** Um ein Spiel zu speichern müssen zunächst die internen Strukturen des Spiels in das vorgegebene Format gebracht werden.

### 3 Programmierhandbuch

Dazu werden *JsonSerializer*-Klassen erstellt die jeweils eine interne Struktur serialisieren. Diese benutzerdefinierten Serialisierer werden dabei im Konstruktor der *GameLogic*-Klasse registriert. Dabei ist der *GameLogicSerializer* so konstruiert, dass dieser bei der Serialisierung benutzung vom *MisterXSerializer* und *DetectiveSerializer* macht.

**Parse von Spielständen** Um ein Spiel zu laden muss die generierte JSON-Datei wieder in interne Strukturen umgewandelt werden. Dafür kann dem *GameLogic*-Klasse Konstruktor ein *Reader* übergeben werden welcher die zu ladende Datei repräsentiert.

```
GameLogic(Reader fileToLoad, Reader jsonBoard, GUIConnector gui)
```

Zunächst wird die Datei eingelesen und mittels der *GSON*-Bibliothek in ein *JsonObject* umgewandelt. Dieses Objekt ermöglicht es nun die Datenstruktur wie ein Baum zu traversieren. Dabei wandeln die Hilfsmethoden

```
private MisterX loadMisterX(JsonObject root)  
und  
private List<Detective> loadDetectives(JsonObject root)
```

den Baum in Mister-X bzw. in die Detektive um. Dabei ähneln die Semantiken der Hilfsmethoden die eines benutzerdefinierten Deserialisierer. Der Grund, dass diese Methoden sich jedoch in der *GameLogic* wieder finden, ist, dass es keine ersichtliche Möglichkeit gibt Parameter an die Deserialisierer zu übergeben. Da jedoch eine *Board*-Instanz benötigt wird, um einen Spieler zu erstellen, konnten die Hilfsmethoden nicht ausgelagert werden.

**Validierung der Spielstände** Da beim Speichern die Daten aus den internen Strukturen direkt übernommen werden, kann hier davon ausgegangen werden, dass die Spielstandsdatei zu diesem Zeitpunkt korrekt ist. Da die Datei jedoch in einem textbasierten und deshalb für Menschen leicht änderbaren Format gespeichert wird, muss eine inhaltliche Überprüfung des Spielstandes beim Laden stattfinden, was ein ungültiges Spiel ausschließen soll.

Um die formalen Fehler kümmert sich bereits die *GSON*-Bibliothek (siehe *Validierung der Datenstruktur* in 3.2.1). Die *JsonValidator*-Klasse übernimmt auch hier wieder die Aufgabe, Fehler in der inhaltlichen Struktur zu finden. Dabei wird der komplette Baum traversiert und überprüft, ob die benötigten Felder vorhanden sind und ob diese den richtigen Datentyp entsprechen. Das Überprüfen der Werte auf Korrektheit der Felder, übernehmen die Hilfsmethoden

### 3 Programmierhandbuch

den *loadMisterX* und *loadDetectives*. Dabei wird geprüft, ob die Stationen überhaupt innerhalb des Spielfelds liegen und ob die Tickets nicht Negativ sind. Ist dies in beiden Fällen nicht gegeben, schmeißt der Konstruktor eine *IllegalArgumentException*. Das Anzeigen des Fehlers übernimmt hierbei der *FXMLDocumentController*. Da die Logik zum Teil schon initialisiert sein kann, muss das Spielfeld trotzdem zurück gesetzt werden.

#### 3.2.5 Grafische Oberfläche

Die grafische Oberfläche (GUI) bietet die Schnittstelle zwischen dem menschlichen Spieler und dem Spiel. Die Oberfläche wurde dabei im *Gluon Scenender* ersterstellt. Die Logik kann über die *JavaFXGui* mit der GUI kommunizieren. Dabei werden der GUI nur logische Daten übergeben, sodass die GUI diese Daten in textuelle Form umwandeln kann. Somit können jegliche Sprachübersetzungen in der GUI getätigter werden.

#### Fahrtenbuch

Das Fahrtenbuch wird durch eine 3x8 *GridPane* die Bilder hält abgebildet. Um ein Ticket hinzuzufügen wird die *setLogbookEntry*-Methode der *JavaFXGui* benutzt.

```
void setLogbookEntry(int round, Ticket ticket)
```

Über den *round*-Paramter wird die Zelle im *GridPane* berechnet.

#### Spieler Darstellung

Die Spieler werden auf dem Spielfeld angezeigt. Sobald sich ein Spieler bewegt hat werden alle Spieler, durch die *drawPlayers*-Methode, auf einmal aktualisiert. Dies hat zwar den Nachteil, dass das Aktualisieren langsamer ist da zunächst alle Spieler gelöscht und wieder angezeigt werden müssen. Allerdings ist das bei maximal sechs Spielern Vertretbar. Ein alternativer Ansatz hierbei wäre es, nur den Spieler zu aktualisieren der sich auch bewegt hat. Dies würde jedoch bedeuten, dass die GUI die Spieler halten muss und zunächst den Spieler ermitteln der sich bewegt hat. Aus diesem Grund wurde der erste Ansatz gewählt um die Spieler zu aktualisieren.

### 3.3 Beschreibung grundlegender Klassen

#### 3.3.1 Gui-Quellpaket

##### ScotlandYard

Tabelle 3.2: Klasse ScotlandYard

Eigenschaft	Beschreibung
Name	ScotlandYard
Ort	Source-Paket <i>gui</i>
Zweck	Enthält die <i>main</i> -Methode die eine JavaFX-Applikation startet.

Beschreibung

- Lädt die FXML-Datei und baut die *Stage* und die *Scene* auf
- Übergibt die Kontrolle an den *FXMLController*

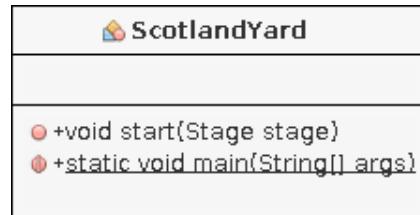


Abbildung 3.1: ScotlandYard UML-Klassendiagramm

### **FXMLDocumentController**

Tabelle 3.3: Klasse FXMLDocumentController

Eigenschaft	Beschreibung
Name	FXMLDocumentController
Ort	Source-Paket <i>gui</i>
Zweck	Wird beim Programmstart geladen. Die Klasse kennt alle Element aus der FXML-Datei. Führt für Interaktionen mit der GUI die entsprechenden Handler aus.
Beschreibung	<ul style="list-style-type: none"> <li>• Instanziert eine <i>JavaFXGui</i>-Instanz und übergibt dieser alle benötigten FXML-Elemente</li> <li>• Instanziert eine <i>GameLogic</i>-Instanz und übergibt dieser die <i>JavaFXGui</i>-Instanz</li> <li>• Reicht alle Interaktionen von der GUI weiter an die <i>GameLogic</i> damit sie dort verarbeitet werden können</li> </ul>

---

### 3 Programmierhandbuch

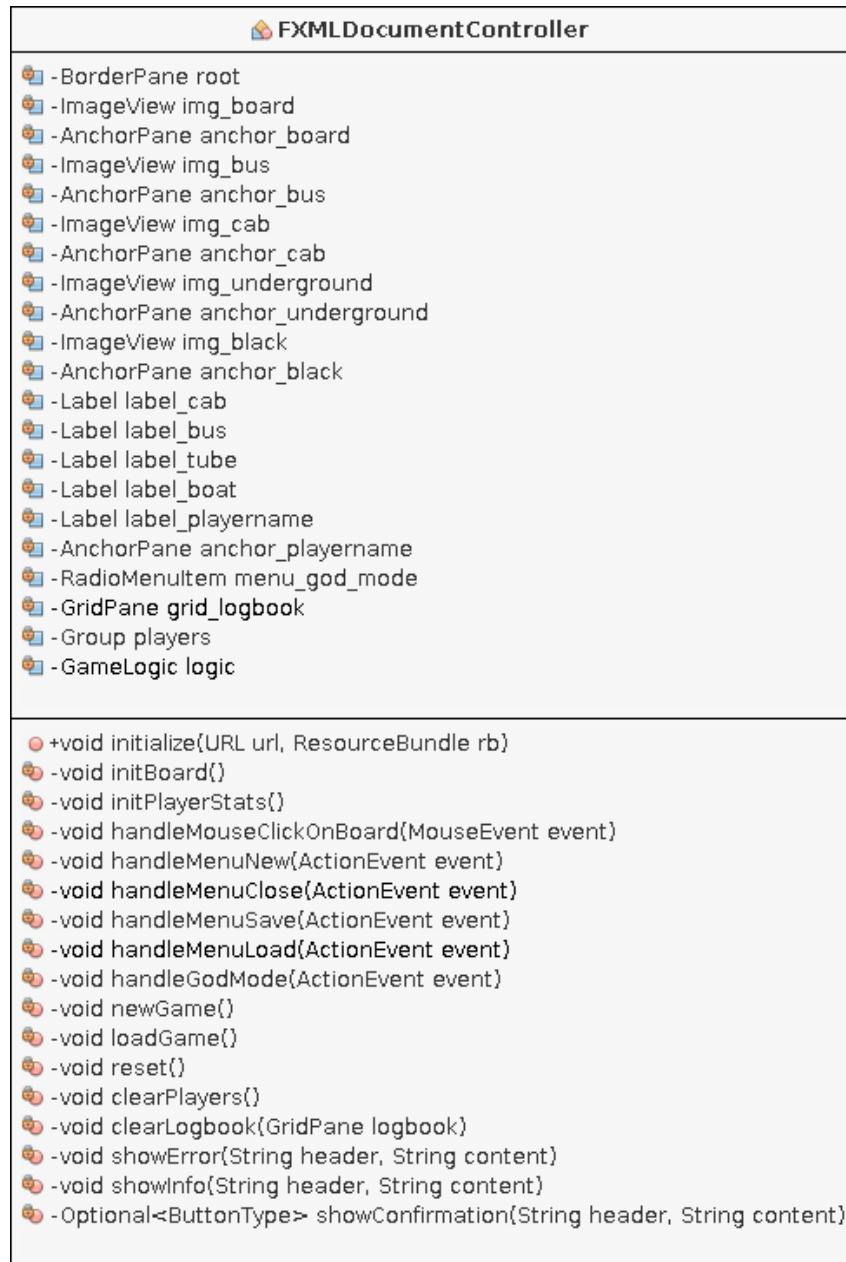


Abbildung 3.2: FXMLDocumentController UML-Klassendiagramm

### 3 Programmierhandbuch

#### **FXMLDocument.fxml**

Die Datei enthält alle graphischen Elemente die für das *Scotland Yard* benötigt werden. Darunter zählen *Id's* oder auch Methoden die eine Interaktion erlauben. Diese Datei wird benötigt um die GUI aufzubauen und mit ihr zu interagieren.

#### **JavaFXGui**

Tabelle 3.4: Klasse JavaFXGui

Eigenschaft	Beschreibung
Name	JavaFXGui
Ort	Source-Paket <i>gui</i>
Zweck	Implementiert, das von der <i>Logic</i> vorgeschriebene <i>GUIConector-Interface</i> . Dadurch kann die <i>GameLogic</i> auf Elemente der graphischen Oberfläche zugreifen.
Beschreibung	<ul style="list-style-type: none"><li>• Erhält beim Instanziieren alle nötigen Elemente der graphischen Oberfläche</li><li>• Implementiert Methoden zur Darstellung der Spieler und ihrer Informationen</li><li>• Implementiert Methoden zur Darstellung von Dialogen und sowie Infomeldungen</li></ul>

### 3 Programmierhandbuch

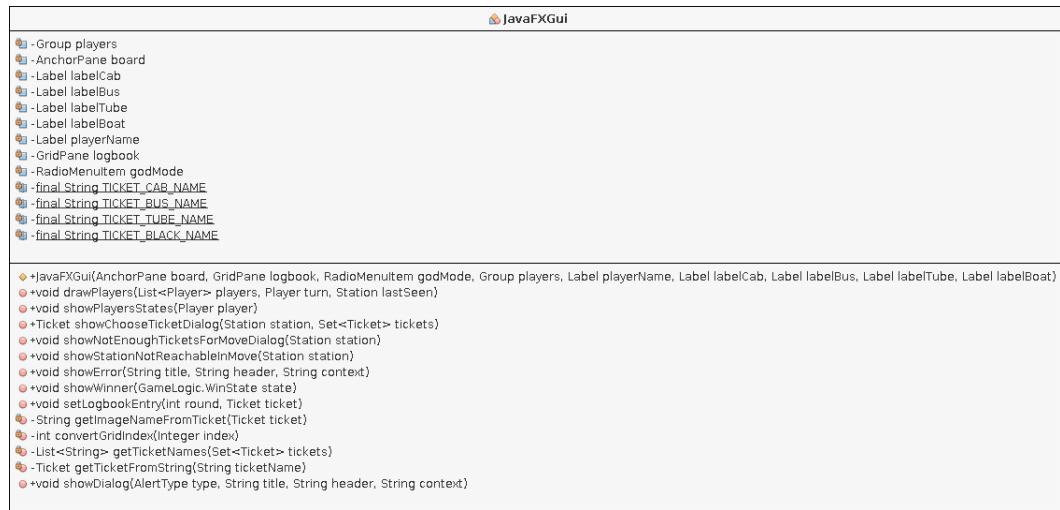


Abbildung 3.3: JavaFXGui UML-Klassendiagramm

### 3.3.2 Logic-Quellpaket

#### GUIConnector

Tabelle 3.5: Interface GUIConnector

Eigenschaft	Beschreibung
Name	GUIConnector
Ort	Quellpaket <i>logic</i>
Zweck	Das Interface bildet, durch vorgegebene Methoden, eine Schnittstelle zwischen der Logik und der graphischen Oberfläche. Die Klassen <i>JavaFxGUI</i> und <i>FakeGui</i> implementieren dieses Interface.

#### Struktur

- Bietet eine Schnittstelle zur Manipulation der graphischen Elemente
- Fordert alle Unterklassen alle Methoden zu implementieren

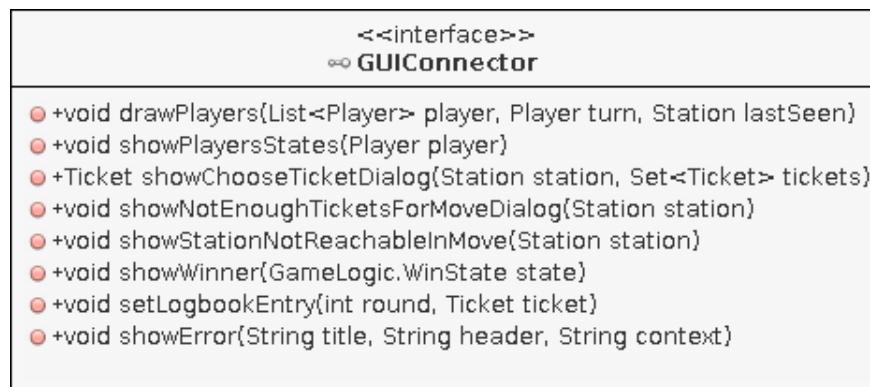


Abbildung 3.4: GUIConnector UML-Klassendiagramm

## GameLogic

Tabelle 3.6: Klasse GameLogic

Eigenschaft	Beschreibung
Name	GameLogic
Ort	Quellpaket <i>logic</i>
Zweck	Verwaltet alle nötigen Spielemente und lässt diese miteinander interagieren. Des Weiteren werden Benutzereingaben vom <i>FXMLDocumentController</i> , wie ein Klicken auf ein Element in der graphischen Oberfläche, an die <i>GameLogic</i> delegiert um dort verarbeitet zu werden.
Struktur	<ul style="list-style-type: none"> <li>• Hält eine statische Klasse <i>Config</i> in der alle wichtigen Konstanten abgelegt sind</li> <li>• Verwaltet das Spielfeld in einer <i>Board</i>-Instanz</li> <li>• Verwaltet eine Gruppe an Detektiven in mehreren <i>Detective</i>-Instanzen und Mister-X in einer <i>MisterX</i>-Instanz</li> <li>• Nimmt Benutzereingaben entgegen und verarbeitet diese ihren <i>handler</i>-Methoden</li> <li>• Ein Spiel kann über einen speziellen Konstruktor geladen werden</li> <li>• Regelt den Ablauf zwischen jedem Zug und steuert die KI der einzelnen Spieler</li> <li>• Hält zusätzliche Informationen über das Spiel wie die Spielrunde oder wer aktuell dran ist</li> </ul>

### 3 Programmierhandbuch

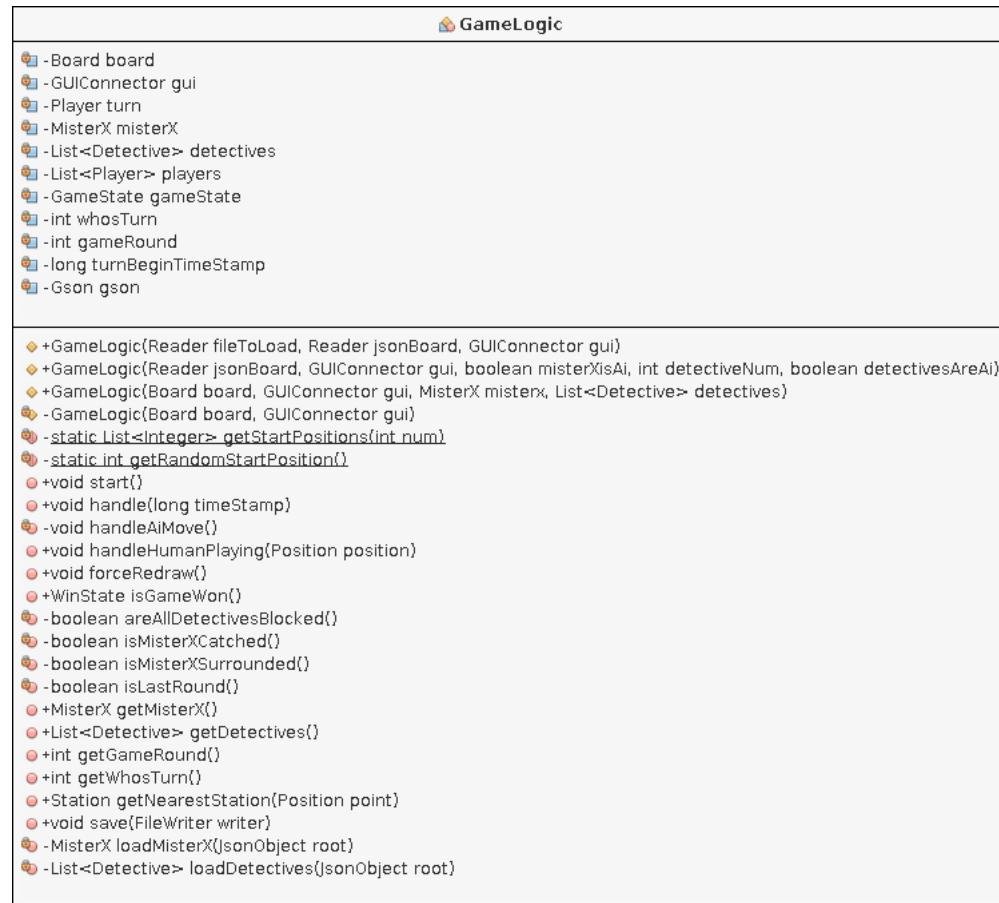


Abbildung 3.5: GameLogic UML-Klassendiagramm

### 3 Programmierhandbuch

#### Move

Tabelle 3.7: Klasse Move

Eigenschaft	Beschreibung
Name	Move
Ort	Quellpaket <i>logic</i>
Zweck	Repräsentiert einen Zug innerhalb des Netzes.
Struktur	

- Hält eine *Station*-Instanz die das Ziel des Zuges repräsentiert
- Hält ein *Ticket*-Wert der das Ticket für den Zug wieder-spiegelt

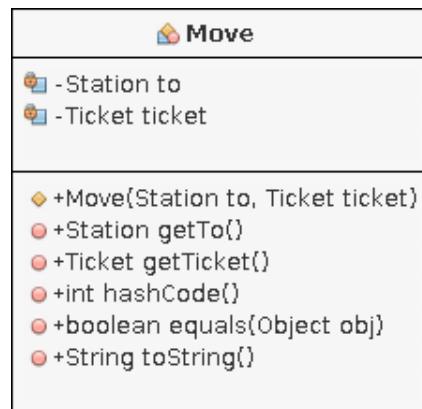


Abbildung 3.6: Move UML-Klassendiagramm

## Ticket

Tabelle 3.8: Klasse Ticket

Eigenschaft	Beschreibung
Name	Ticket
Ort	Quellpaket <i>logic</i>
Zweck	Bildet die Tickets für die verschiedenen Verkehrsmittel ab.
Struktur	

- Hält die Enumkonstanten *CAB*, *BUS*, *TUBE*, *BLACK*
- bietet über die *from*-Methode eine Möglichkeit über einen Ordinalwert ein Ticket zu generieren

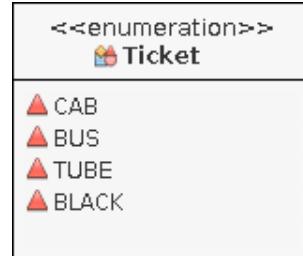


Abbildung 3.7: Ticket UML-Klassendiagramm

### 3.3.3 Logic.util-Quellpaket

#### Logger

Tabelle 3.9: Klasse Logger

Eigenschaft	Beschreibung
Name	Logger
Ort	Quellpaket <i>logic.util</i>
Zweck	Die Klasse <i>Logger</i> bietet die Möglichkeit Spielinformationen in einer Datei zu speichern
Struktur	

- Die *printNewGame*-Methode speichert die generellen Informationen des Spielfeldes
- Die *printMove*-Methode speichert jeweils einen Zug eines Spielers



Abbildung 3.8: Logger UML-Klassendiagramm

### JsonValidator

Tabelle 3.10: Klasse JsonValidator

Eigenschaft	Beschreibung
Name	JsonValidator
Ort	Quellpaket <i>logic.util</i>
Zweck	Bietet Methoden zur Überprüfung der <i>JSON</i> -Datenstruktur für einen Spielstand und des Netzes.
Struktur	<ul style="list-style-type: none"> <li>• Die <i>validateBoard</i>-Methode validiert die <i>JSON</i>-Datenstruktur des Spielfeldes</li> <li>• Die <i>validateSaveState</i>-Methode validiert die <i>JSON</i>-Datenstruktur der Spielstandsdatei</li> </ul>

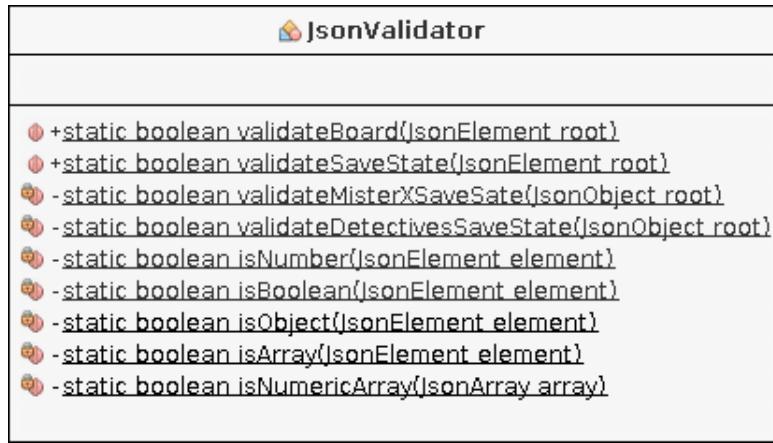


Abbildung 3.9: JsonValidator UML-Klassendiagramm

### 3 Programmierhandbuch

#### GameLogicSerializer

Tabelle 3.11: Klasse GameLogicSerializer

Eigenschaft	Beschreibung
Name	GameLogicSerializer
Ort	Quellpaket <i>logic.util</i>
Zweck	Die <i>GameLogicSerializer</i> -Klasse ist eine Hilfsmethode für die <i>GSON</i> -Bibliothek und bietet einen benutzerdefinierten Serialisierer für die <i>GameLogic</i> .

#### Struktur

- Wird von *GSON* selbst instanziert und verwendet. Bietet über die *serialize*-Methode die Möglichkeit eine *GameLogic*-Instanz in ein *JsonObject* nach Aufgabenstellung zu transformieren.

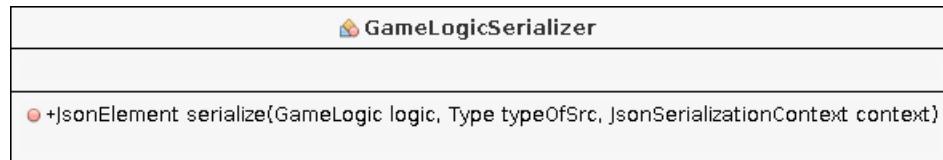


Abbildung 3.10: GameLogicSerializer UML-Klassendiagramm

## MisterXSerializer

Tabelle 3.12: Klasse MisterXSerializer

Eigenschaft	Beschreibung
Name	MisterXSerializer
Ort	Quellpaket <i>logic.util</i>
Zweck	Die <i>MisterXSerializer</i> -Klasse ist eine Hilfsmethode für die <i>GSON</i> -Bibliothek und bietet einen benutzerdefinierten Serialisierer für <i>MisterX</i> .

### Struktur

- Wird von *GSON* selbst instanziert und verwendet. Bietet über die *serialize*-Methode die Möglichkeit eine *MisterX*-Instanz in ein *JsonObject* nach Aufgabenstellung zu transformieren.



Abbildung 3.11: MisterXSerializer UML-Klassendiagramm

### 3 Programmierhandbuch

#### DetectiveSerializer

Tabelle 3.13: Klasse MisterXSerializer

Eigenschaft	Beschreibung
Name	MisterXSerializer
Ort	Quellpaket <i>logic.util</i>
Zweck	Die <i>DetectiveSerializer</i> -Klasse ist eine Hilfsmethode für die <i>GSON</i> -Bibliothek und bietet einen benutzerdefinierten Serialisierer für <i>Detective</i> .

#### Struktur

- Wird von *GSON* selbst instanziert und verwendet. Bietet über die *serialize*-Methode die Möglichkeit eine *Detective*-Instanz in ein *JsonObject* nach Aufgabenstellung zu transformieren.

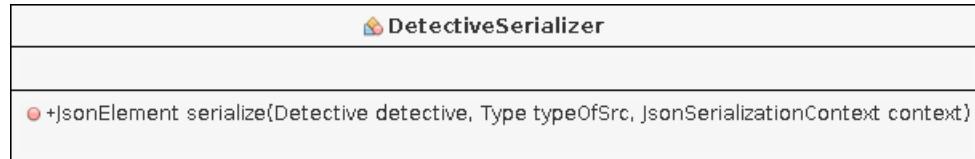


Abbildung 3.12: DetectiveSerializer UML-Klassendiagramm

### 3.3.4 Logic.board-Quellpaket

#### Board

Tabelle 3.14: Klasse Board

Eigenschaft	Beschreibung
Name	Board
Ort	Quellpaket <i>logic.board</i>
Zweck	Repräsentiert das komplette Spielfeld mit all seinen Stationen und Verbindungen.
Struktur	

- Das Spielfeld lässt sich über eine *JSON*-Datenstruktur, die das Spielfeld abbildet, laden.
- Hält eine *stations-ArrayList* aus *Station*-Instanzen
- Die *getStation*-Methode bietet die Möglichkeit eine Station über ihre Identifikationsnummer zu bekommen

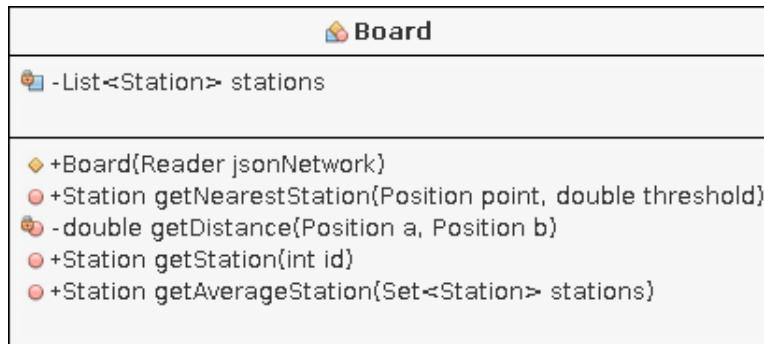


Abbildung 3.13: Board UML-Klassendiagramm

### 3 Programmierhandbuch

#### Station

Tabelle 3.15: Klasse Station

Eigenschaft	Beschreibung
Name	Station
Ort	Quellpaket <i>logic.board</i>
Zweck	Repräsentiert eine Station und somit einen Knotenpunkt im Netz.

#### Struktur

- Hält vier *Sets* von *Stations* die jeweils ein Verbindung zu anderen Stationen repräsentieren
- Besitzt ein *Occupied*-Flag welches aussagt, ob die Station besetzt ist oder nicht
- Gibt über die *getSurroundingStations*-Methode alle umliegenden Stationen zurück
- Gibt über die *getStationsReachableBy*-Methode alle Stationen wieder, die über das übergebene Ticket erreichbar sind
- Gibt über die *getTicketsToReachableStation*-methode das Ticket wieder, welches benötigt wird um die Station zu erreichen

### 3 Programmierhandbuch

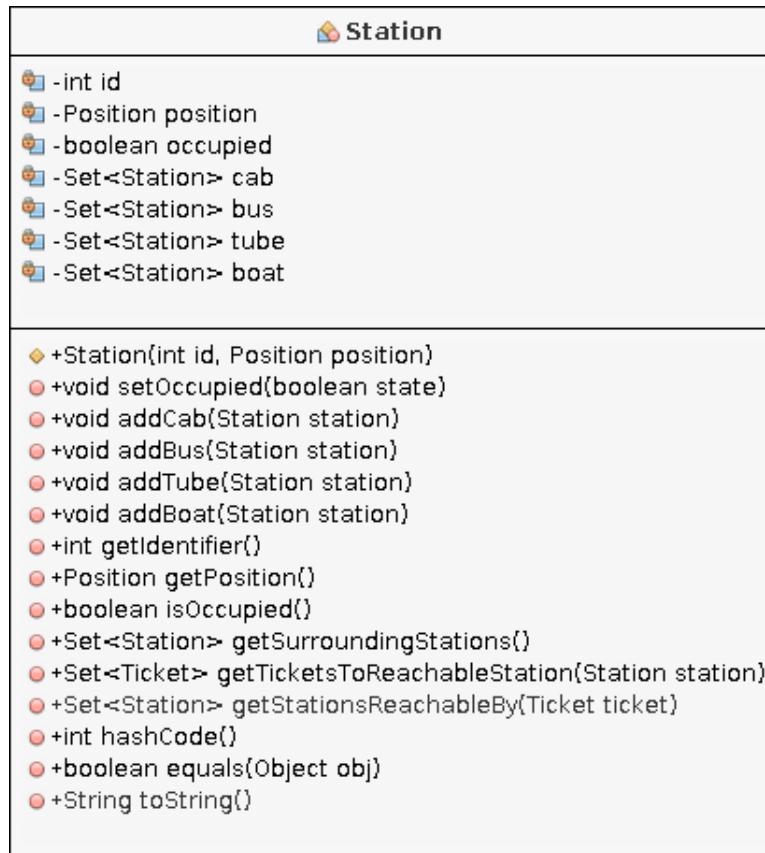


Abbildung 3.14: Station UML-Klassendiagramm

### 3 Programmierhandbuch

#### Position

Tabelle 3.16: Klasse Position

Eigenschaft	Beschreibung
Name	Position
Ort	Quellpaket <i>logic.board</i>
Zweck	Eine Hilfsklasse die zweidimensionale Koordinaten abbildet. Wird benötigt um die Position der Stationen zu speichern.

Struktur

- Hält jeweils den X und den Y Anteil einer Position
- bietet Funktionen die X und Y Werte auszulesen

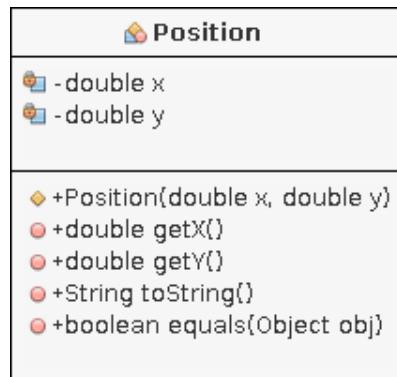


Abbildung 3.15: Position UML-Klassendiagramm

### 3.3.5 Logic.player-Quellpaket

#### Player

Tabelle 3.17: Klasse Player

Eigenschaft	Beschreibung
Name	Player
Ort	Quellpaket <i>logic.player</i>
Zweck	Bietet eine abstrakte Oberklasse für die <i>Detective</i> -Klasse und die <i>MisterX</i> -Klasse und implementiert dabei alle grundlegenden Funktionen die benötigt werden um als Spieler am Spielgeschehen teilzunehmen. Darunter zählen Bewertungsfunktionen für die KI die für alle Spieler gleich sind oder auch die Ticketverwaltung sowie die Wegfindung.

#### Struktur

- Hält alle grundlegenden Informationen über einen Spieler wie die Tickets in dem *ticket-Array*, die aktuelle Station in der *currentStation*-Variable als *Station*-Instanz und ob der Spieler durch die KI oder durch einen menschlichen Spieler gesteuert wird in der *ai*-Variable als *boolean*
  - Die *getShortestWay*-Methode bietet die Möglichkeit den kürzesten Weg durch das Netz zu finden und macht dabei Gebrauch von der *StationDistance*-Klasse
  - Fordert die Implementierung der *play*-Methode. Dadurch lässt sich die KI der Detektive und die von MisterX gleich behandeln.
-

### 3 Programmierhandbuch



Abbildung 3.16: Player UML-Klassendiagramm

### 3 Programmierhandbuch

#### MisterX

Tabelle 3.18: Klasse MisterX

Eigenschaft	Beschreibung
Name	MisterX
Ort	Quellpaket <i>logic.player</i>
Zweck	Die <i>MisterX</i> -Klasse erweitert die <i>Player</i> -Klasse um Spezialisierungen um als Mister-X am Spielgeschehen teilzunehmen. Dazu gehören die Taktiken sowie die Bewertungsfunktionen und das Fahrtenbuch.

#### Struktur

- Hält in der *logbook*-Variable das Fahrtenbuch welches als Liste von Tickets repräsentiert wird sowie die Station an der sich Mister-X zuletzt gezeigt hat in der *lastSeen*-Variable als *Station*-Instanz
- Implementiert, wie von der Oberklasse vorgegeben, die *play*-Methode
- Macht beim vergleichen der Taktiken gebrauch von der *TacticResult*-Klasse

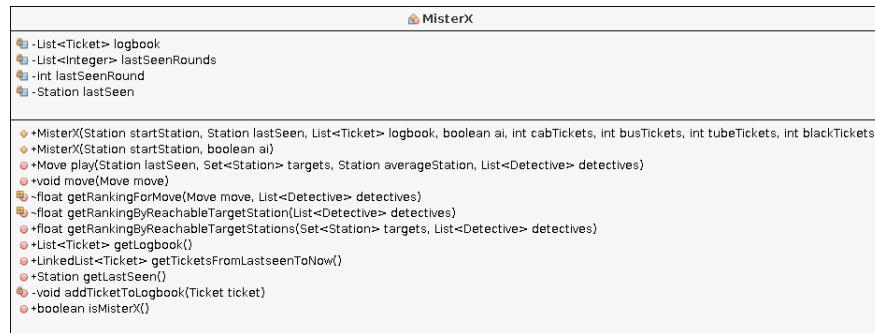


Abbildung 3.17: MisterX UML-Klassendiagramm

### 3 Programmierhandbuch

#### Detective

Tabelle 3.19: Klasse Detective

Eigenschaft	Beschreibung
Name	Detective
Ort	Quellpaket <i>logic.player</i>
Zweck	Die <i>Detective</i> -Klasse erweitert die <i>Player</i> -Klasse um Spezialisierungen um als Detektiv am Spielgeschehen teilzunehmen. Dazu gehören die Taktiken sowie die Bewertungsfunktionen

#### Struktur

- Implementiert, wie von der Oberklasse vorgegeben, die *play*-Methode
- Macht beim vergleichen der Taktiken gebrauch von der *TacticResult*-Klasse

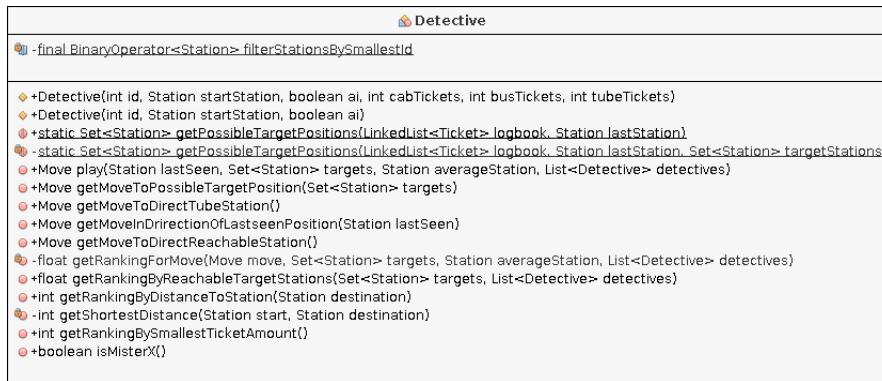


Abbildung 3.18: Detective UML-Klassendiagramm

### TacticResult

Tabelle 3.20: Klasse TacticResult

Eigenschaft	Beschreibung
Name	TacticResult
Ort	Quellpaket <i>logic.player</i>
Zweck	Die Hilfsklasse <i>TacticResult</i> wird benötigt um die Taktiken miteinander zu vergleichen.

#### Struktur

- Hält den Zug der Taktik als *Move*-Instanz, die Identifikationsnummer und die Bewertung der Taktik.
- Bietet einen Konstruktor der alle *final* Attribute setzt
- Bietet Getter-Methoden für die Attribute

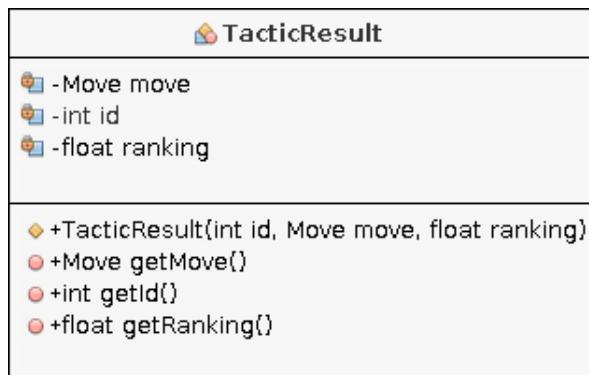


Abbildung 3.19: TacticResult UML-Klassendiagramm

### 3 Programmierhandbuch

#### StationDistance

Tabelle 3.21: Klasse StationDistance

Eigenschaft	Beschreibung
Name	StationDistance
Ort	Quellpaket <i>logic.player</i>
Zweck	Die Hilfsklasse <i>StationDistance</i> wird für die Wegfindung benötigt und repräsentiert den Abstand von einer Station. Durch eine Menge von <i>StationDistance</i> kann somit der kürzeste Weg ermittelt werden.

#### Struktur

- Hält eine *Station*-Instanz welche die Station ist, die den Abstand hat der in der *distance*-Variable gespeichert ist

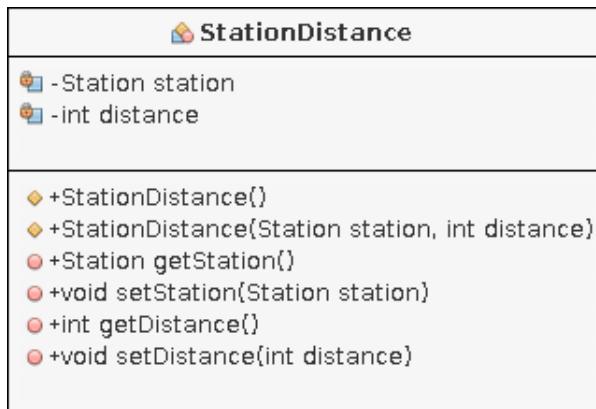


Abbildung 3.20: StationDistance UML-Klassendiagramm

### 3.3.6 Test.gui Quellpaket

#### FakeGUI

Tabelle 3.22: Klasse FakeGUI

Eigenschaft	Beschreibung
Name	FakeGUI
Ort	Quellpaket <i>test.gui</i>
Zweck	Eine Klasse die das <i>GUIConector</i> -Interface implementiert. Dabei sind die Methoden jedoch so implementiert, dass diese nichts tun. So können Tests ausgeführt werden ohne ein graphische Oberfläche zu benutzen.
Struktur	<ul style="list-style-type: none"> <li>• Alle Methoden haben einen leeren Funktionskörper</li> </ul>

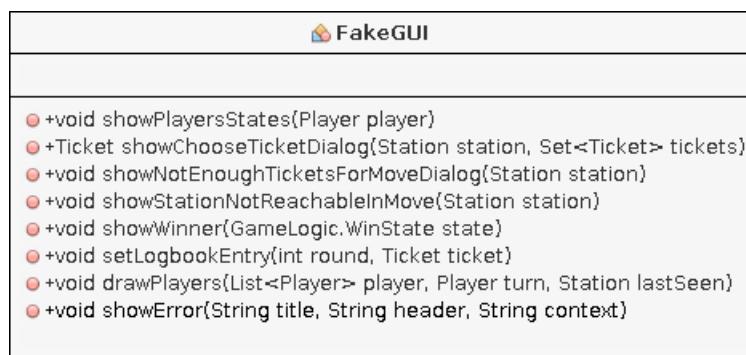


Abbildung 3.21: FakeGUI UML-Klassendiagramm

### *3 Programmierhandbuch*

#### **3.3.7 Benutze Datenstrukturen**

Die benutzen Datenstrukturen wurden in 2.5 behandelt.

### 3.4 Programmorganisationsplan

Folgende Abbildung zeigt eine grundlegende Übersicht der wichtigsten Klassen und deren Zusammenwirken.

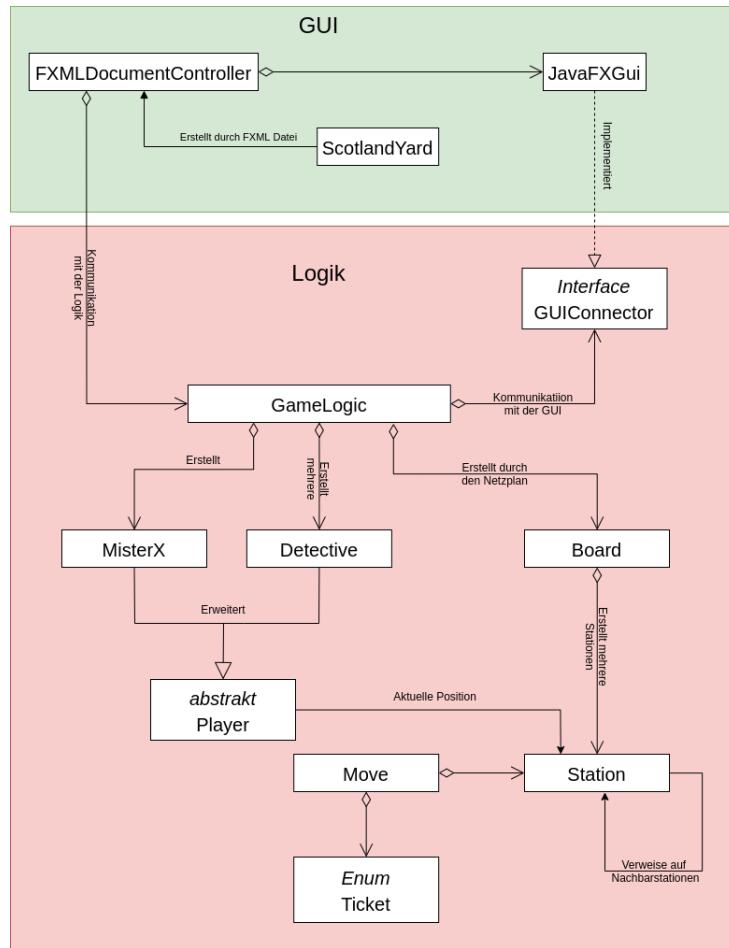


Abbildung 3.22: Vereinfachter Programmorganisationsplan

Beim Programmstart wird die *main*-Methode innerhalb der *ScotlandYard*-Klasse ausgeführt. Da die *ScotlandYard*-Klasse die *Application*-Klasse erweitert, wird

### 3 Programmierhandbuch

beim aufrufen ihrer *start*-Methode die FXML-Datei geladen und daraus der *FXMLDocumentController* erstellt. Der *FXMLDocumentController* besitzt alle graphischen Elemente der Oberfläche und erstellt daraus die *JavaFXGui*-Klasse und übergibt diese beim erstellen an die *GameLogic*.

Die *GameLogic* erstellt durch die Parameter ihres Konstruktors jeweils *MisterX*, mehrere Instanzen von *Detective* und das Spielfeld (*Board*). *MisterX* und *Detective* sind dabei Spezialisierungen der abstrakten *Player*-Klasse. Durch die übergebene *JavaFXGui*-Instanz, die den *GUICConnector* implementiert, kann die *GameLogic* die GUI direkt manipulieren.

Das *Board* erstellt beim Instanziieren alle nötigen Stationen die jeweils Referenzen auf ihre Nachbarstationen halten.

### 3.5 Programmtests

Neben den automatischen Tests kann die Logik auch mit Testdateien getestet werden. Diese Dateien sind speziell konstruierte Szenarien in denen ein Fehler Auftritt.

Tabelle 3.23: Spielfeld Tests

Testfall	Erwartetes Ergebnis	Erzieltes Ergebnis
Es wird getestet, ob das zu ladende Spielfeld fehlerfrei ist. Die entsprechenden automatischen Unit-Tests ( <i>CorruptedNetwork</i> *) befinden sich in <i>GameLogicTest</i> . Manuell kann dies auch mithilfe der mitgelieferten Testdateien getestet werden. Dazu muss jedoch das <i>JAR</i> entpackt werden und die <i>network.json</i> durch die <i>network_wrongJson.json</i> oder die <i>network_wrongFormat.json</i> oder die <i>network_wrongValues.json</i> ersetzt werden. Dabei ist zu beachten, dass die ersetzende Datei in <i>network.json</i> unbenannt werden muss. Das <i>JAR</i> muss zusätzlich neu gepackt werden.	Das Programm sollte eine Fehlermeldung ausgeben, dass das zu ladende Spielfeld korrupt ist.	Das Programm gibt die Fehlermeldung aus.

Tabelle 3.25: Spielstand Tests

Testfall	Erwartetes Ergebnis	Erzieltes Ergebnis
Es wird getestet, ob die Spielstandsdatei fehlerfrei ist. Die entsprechenden automatischen Unit-Tests ( <i>CorruptedSaveState</i> *) befinden sich in <i>GameLogicTest</i> . Manuell kann dies auch mithilfe der mitgelieferten Testdateien getestet werden. Dazu wird die Datei <i>save_wrongJson.sy</i> geladen um eine fehlerhafte JSON-Struktur zu testen. Die Datei <i>save_wrongFormat.sy</i> hingegen testet, ob die formale Struktur eingehalten wird. Dies bezieht fehlende Felder mit ein. Die <i>save_wrongValues.sy</i> testet, ob die Werte korrekt sind. Dies beinhaltet z.B. negative Ticketanzahlen.	Das Programm sollte eine Fehlermeldung ausgeben, dass der zu ladende Spielstand korrupt ist.	Das Programm gibt die Fehlermeldung aus.
Es wird getestet, ob ein Spielstand erfolgreich geladen wird. Dazu wird die <i>save.sy</i> Datei geladen.	Der Spielstand lädt erfolgreich. Dabei steht Mister-X auf der Station 138 und besitzt zehn Taxitickets, acht Bustickets, 4 U-Bahntickets und 2 Blacktickets. Das Fahrtenbuch ist dabei leer. Die Detektive stehen jeweils auf den Stationen 197, 34 und 94 und besitzen 10 Taxitickets, 8 Bustickets und 4 U-Bahntickets. Mister-X ist am Zug.	der Spielstand wird erfolgreich geladen und die Werte werden korrekt angezeigt.

Tabelle 3.27: Interaktions Tests

Testfall	Erwartetes Ergebnis	Erzieltes Ergebnis
<b>Station nicht erreichbar</b> Es wird getestet, ob der Fehler abgefangen wird falls ein Spieler eine Station anklickt die zu weit entfernt wird. Dafür wird das Spiel gestartet und auf eine Station geklickt die zu weit weg ist als, dass diese erreicht werden kann.	Das Programm sollte eine Infomeldung ausgeben, dass die Station nicht in einem Zug erreicht werden kann. Zudem sollte der Spieler die erneute Möglichkeit haben eine Station zu wählen.	Das Programm gibt die Infomeldung aus.
<b>Station über mehrere Tickets erreichbar</b> Es wird getestet, ob dem Spieler eine Möglichkeit geboten wird zwischen mehreren Tickets zu wählen, falls dieser eine Station ausgewählt hat die über mehrere Tickets angefahren werden kann. Dazu klickt der Spieler auf eine erreichbare Station die über mehrere Verbindungen erreichbar ist.	Dem Spieler sollte ein Dialog gezeigt werden, sodass dieser ein Ticket auswählen kann. Steuert der Spieler Mister-X wird ihm immer das Blackticket angeboten solange dieser noch über genügend Blacktickets verfügt. Falls der Spieler über kein Ticket mehr verfügt, sodass es keine Wahlmöglichkeit gibt, wird die Station angefahren und das Ticket entfernt.	Dem Spieler wird der Dialog mit den richtigen Tickets angezeigt. Falls dieser Dialog abgebrochen bzw. geschlossen wird, wird das Ticket benutzt welches ausgewählt wurde.
<b>Keine Verfügbaren Tickets mehr übrig</b> Es wird getestet, ob dem Spieler eine Infomeldung ausgegeben wird, wenn ein Spieler versucht ein Verkehrsmittel zu benutzen wofür er keine Tickets mehr hat. Dazu wird solange gespielt, bis von einem Verkehrsmittel keine Tickets mehr vorhanden sind.	Dem Spieler sollte eine Infomeldung angezeigt werden, dass ihm keine Tickets mehr zur Verfügung stehen.	Dem Spieler wird die Infomeldung angezeigt.

Tabelle 3.29: Einstellungs Tests

Testfall	Erwartetes Ergebnis	Erzieltes Ergebnis
<b>Godmode</b> Es wird getestet, ob der Godmode funktioniert. Dabei wird das Spiel gestartet und auf eine Spielsituation abgewartet, in der sich Mister-X nicht zeigt. Nun wird auf den <i>Godmode</i> -Menüpunkt gedrückt.	Mister-X sollte sich sofort zeigen. Nach wiederholten drücken auf den Menüpunkt sollte dieser sich sofort wieder verstecken.	Mister-X wird korrekt dargestellt.

## Abbildungsverzeichnis

2.1	Spieloberfläche nach starten des Programmes . . . . .	14
2.2	Menüpunkt <i>Game</i> . . . . .	15
2.3	Menüpunkt <i>Edit</i> . . . . .	15
2.4	Dialog: Standardeinstellungen für ein neues Spiel . . . . .	16
2.5	Spieldorf nach ein paar Spielzügen . . . . .	17
2.6	Warnung: Aktuelles Spiel geht beim starten verloren . . . . .	17
2.7	Meldung: Letzte runde erreicht. Mister-X hat gewonnen . . . . .	18
2.8	Meldung: Mister-X wurde gefangen . . . . .	19
2.9	Meldung: Alle Detektive sind blockiert . . . . .	19
2.10	Dialog: Ticketauswahl . . . . .	20
2.11	Meldung: Station nicht erreichbar . . . . .	21
2.12	Meldung: Ungenügend Tickets . . . . .	21
2.13	Meldung: Speicherung eines nicht existierenden Spielstandes . . . . .	22
2.14	Datei Auswahl zum speichern des Spielstandes . . . . .	22
2.15	Meldung: Überschreibung eines Spielstandes . . . . .	23
2.16	Datei-Navigator zum laden eines Spielstandes . . . . .	24
2.17	Meldung: Der aktuelle Spielstand wird gelöscht . . . . .	24
2.18	Fehler: Spieldorf falsch formatiert . . . . .	26
2.19	Fehler: Spieldorf beinhaltet ungültige Werte . . . . .	26
2.20	Fehler: Spieldorf nicht gefunden . . . . .	27
2.21	Fehler: Spielstandsdatei falsch formatiert . . . . .	27
2.22	Fehler: Spielstandsdatei beinhaltet falsche Werte . . . . .	28
3.1	ScotlandYard UML-Klassendiagramm . . . . .	51
3.2	FXMLDocumentController UML-Klassendiagramm . . . . .	53
3.3	JavaFXGui UML-Klassendiagramm . . . . .	55
3.4	GUIConector UML-Klassendiagramm . . . . .	56
3.5	GameLogic UML-Klassendiagramm . . . . .	58
3.6	Move UML-Klassendiagramm . . . . .	59
3.7	Ticket UML-Klassendiagramm . . . . .	60
3.8	Logger UML-Klassendiagramm . . . . .	61
3.9	JsonValidator UML-Klassendiagramm . . . . .	62
3.10	GameLogicSerializer UML-Klassendiagramm . . . . .	63
3.11	MisterXSerializer UML-Klassendiagramm . . . . .	64
3.12	DetectiveSerializer UML-Klassendiagramm . . . . .	65

*Abbildungsverzeichnis*

3.13 Board UML-Klassendiagramm . . . . .	66
3.14 Station UML-Klassendiagramm . . . . .	68
3.15 Position UML-Klassendiagramm . . . . .	69
3.16 Player UML-Klassendiagramm . . . . .	71
3.17 MisterX UML-Klassendiagramm . . . . .	72
3.18 Detective UML-Klassendiagramm . . . . .	73
3.19 TacticResult UML-Klassendiagramm . . . . .	74
3.20 StationDistance UML-Klassendiagramm . . . . .	75
3.21 FakeGUI UML-Klassendiagramm . . . . .	76
3.22 Vereinfachter Programmorganisationsplan . . . . .	78

## Tabellenverzeichnis

2.1 Programm Anforderung . . . . .	13
2.2 Programmstart . . . . .	13
3.1 Programmanforderung . . . . .	31
3.2 Klasse ScotlandYard . . . . .	51
3.3 Klasse FXMLDocumentController . . . . .	52
3.4 Klasse JavaFXGui . . . . .	54
3.5 Interface GUIConnector . . . . .	56
3.6 Klasse GameLogic . . . . .	57
3.7 Klasse Move . . . . .	59
3.8 Klasse Ticket . . . . .	60
3.9 Klasse Logger . . . . .	61
3.10 Klasse JsonValidator . . . . .	62
3.11 Klasse GameLogicSerializer . . . . .	63
3.12 Klasse MisterXSerializer . . . . .	64
3.13 Klasse MisterXSerializer . . . . .	65
3.14 Klasse Board . . . . .	66
3.15 Klasse Station . . . . .	67
3.16 Klasse Position . . . . .	69
3.17 Klasse Player . . . . .	70
3.18 Klasse MisterX . . . . .	72
3.19 Klasse Detective . . . . .	73
3.20 Klasse TacticResult . . . . .	74
3.21 Klasse StationDistance . . . . .	75
3.22 Klasse FakeGUI . . . . .	76
3.23 Spielfeld Tests . . . . .	80
3.25 Spielstand Tests . . . . .	81
3.27 Interaktions Tests . . . . .	82
3.29 Einstellungs Tests . . . . .	83