Programmierpraktikum

Scotland Yard

Löffler, Alexander

Informatik inf102691

Fachsemester: 7 Verwaltungssemester: 7

1. Benutzerhandbuch

1.1 Ablaufbedingungen

Notwendige Komponenten

- Es muss ein betriebsfähiger Computer mit installierter Java JRE1.8.0 Version oder höher

1.2 Programmstart

Um das Programm korrekt zu starten muss sich die **pp_sy_loeffler.jar** Datei in demselben Ordner befinden wie der **lib** Ordner, welcher die Dateien **gson-2.8.0.jar**, **gson-2.8.0-javadoc.jar** und **gson-2.8.0-sources.jar enthält.**

1.3 Bedienungsanleitung

1.3.1 Ziel des Spieles

Das Ziel des Spiels Scotland Yard ist entweder Mister X zu fangen, für den Fall das man einen Detektiv spielt, oder aber vor den Detektiven wegzulaufen, falls man Mister X spielt. Das Spiel hat maximal 24 Runden und ist genau dann zu ende, wenn Mister X es schafft in diesen Runden nicht gefangen zu werden oder wenn die Detektive ihn vorher fangen.

1.3.2 Das Spiel

Gespielt wird auf einer Karte auf der unterschiedliche Stationen markiert sind. Die Spieler dürfen sich nur über diese Stationen auf der Karte fortbewegen.
Jede Station ist durch verschiedene Verkehrsmittel zu erreichen und sind dementsprechend markiert.
Es gibt die Verkehrsmittel:

- Taxi (Gelb)
- Bus (Grün)
- U-Bahn (Rot)

- Boot/Black (Schwarz)

Um sich von einer Station zur nächsten zu bewegen, so müssen beide Stationen das gewählte Verkehrsmittel unterstützen und über dieses direkt Benachbart sein. Das bedeutet, dass wenn sich der Spieler z.B. mit der U-Bahn fortbewegen möchte, er nur von der aktuellen U-Bahn Station zur nächsten reisen darf, nicht aber zur Übernächsten.

Um solch eine Reise umzusetzen, erhalten die Spieler sogenannte Tickets. Es gibt für jedes Verkehrsmittel eine eigene Ticketart. Wenn ein Spieler mit einem Verkehrsmittel zu einer anderen Station reist, so wird stets genau 1 Ticket des gewählten Verkehrsmittel aufgebraucht.
Es gibt Taxi-Tickets, Boots-Tickets, Bahn Tickets und Black-Tickets. Die Black-Tickets haben eine Sonderstellung. Sie können für jedes Verkehrsmittel genutzt werden als eine Art Joker und Mister X ist der einzige, der über diese Tickets verfügt. Außerdem sind sie die einzigen Tickets, die für eine Reise per Boot zulässig sind.
Jeder Spieler darf nur ein Verkehrsmittel benutzen, wenn er auch über entsprechende Tickets verfügt. Hat er keine Taxi Tickets mehr, so darf er auch nicht mehr mit einem Taxi reisen.

Ist eine Station mit mehreren Verkehrsmitteln erreichbar und der Spieler hat auch dementsprechend Tickets, so kann er auswählen welches der Tickets er benutzen möchte.

Die Position von Mister X ist für die Detektive nicht sichtbar. Er zeigt sich nur in speziellen Runden, in denen er auftauchen muss. Die Detektive müssen also eine Strategie entwickeln wie sie den “unsichtbaren” Mister X am besten fangen.

Neben einen Vorrat an diversen Tickets haben die Spieler einsicht auf eine Fahrtentafel. Auf ihr wird markiert welches Verkehrsmittel Mister X in welchem Zug genutzt hat. Mit ihr lassen sich die Züge von Mister X weiter eingrenzen.
Die Züge in denen sich Mister X zeigt sind Farblich hervorgehoben.

In dem Moment wo ein Detektiv auf der gleichen Station ist, wie Mister X haben die Detektive das Spiel gewonnen. Gelingt dies nicht bis zum Ende der letzten Runde, so gewinnt Mister X.

1.3.3 Starten des Spieles

Nach dem der Benutzer das Programm gestartet hat, so sieht er verschiedene Elemente im Programm:

- **Karte:** Auf ihr wird das Spiel gespielt. Im laufenden Spiel werden hier die verschiedenen Spieler angezeigt. Die Detektive werden mit kleinen Lupen und entsprechenden Nummer markiert, Mister X durch ein gelbes Fragezeichen.

- **Ticket Vorrat:** Am oberen Rechten Spielrand findet der Aktuelle Spieler seinen Ticket Vorrat. Dort werden die verbleibenden Tickets für die jeweiligen Verkehrsmittel angezeigt.

- **Fahrtentafel**: Am unteren Rechten Rand des Programms befindet sich die sogenannte Fahrtentafel. Sie ist das wichtigste Hilfsmittel für die Detektive. Auf ihr wird angezeigt in welcher Runde Mister X welches Verkehrsmittel genutzt hat.

Außerdem sieht der Benutzer oben Links zwei Menüreiter, welche ein Dropdown Menü öffnen.

- **Spiel:** Hier kann der benutzer Auswählen **ein neues Spiel zu Starten**, **ein Spiel aus einer Datei zu laden**, oder aber **das aktuelle zu Speichern**.
- **Einstellungen:** Hier findet der benutzer einen “Cheat-Mode”, der ihm erlaubt Mister X immer sichtbar zu machen.

1.3.4 Ein neues Spiel starten

Um ein Spiel zu starten, muss der Benutzer “Neues Spiel” auswählen. Er sieht nun ein kleines Menü in dem er entscheiden kann wie das Spiel gespielt wird. Er kann entscheiden alle Detektive und oder Mister X von einer KI gespielt werden sollen. Sollte dies nicht der Fall sein, so spielt der Benutzer alle Detektive oder Mister X oder eben beide Fraktionen. Er kann außerdem auswählen wie viele Detektive am Spiel Teilnehmen dürfen. Spielt der benutzer Mister X selber, so ist dieser natürlich dauerhaft sichtbar.

Es können 3-5 Detektive gleichzeitig spielen.

Nachdem das neue Spiel gestartet wurde, bekommen alle Spieler einen Vorrat an Tickets, welche im Ticket Vorrat angezeigt werden. Die Tickets für Mister X sind nicht fix. Jedes Ticket welches ein Detektiv verbraucht bekommt Mister X. Benutzt der Detektiv z.B. ein Taxi, so erhält Mister X ein Taxi Ticket mehr.

Gespielt wird reihum beginnend mist Mister X und dann Detektiv 1, 2, usw.

Ist der Benutzer am Zug (unabhängig ob er Mister X oder die Detektive spielt) so kann er nun auf eine Station auf der Karte klicken. Die getroffene Station wird mit einem kleinen Kreis markiert. Ist die markierte Station in diesem Zug erreichbar so bewegt sich der Spieler auf die neue Station und ein entsprechendes Ticket wird verbraucht. Kann er mehr als ein Ticket benutzen, so kann der Spieler auswählen welches er nutzen möchte.

Ist eine Station von einem Detektiv besetzt ist, so kann sich kein weiterer Detektiv auf diese Station bewegen.

Spiel die KI eine Fraktion, so werden die Züge dementsprechend automatisch ausgeführt.

1.3.4 Spiel speichern

Wenn der Benutzer auf Spiel speichern drückt, so wird eine Speicherdatei in dem Ordner erzeugt in dem die **pp_sy_loeffler.jar** Datei liegt.

1.3.5 Spiel laden

Um einen Spielstand zu laden, kann der Benutzer auf **Spiel Laden** drücken und eine von ihr erstellte Speicherdatei laden. Die Datei hat den Namen **savegame.txt.**

1.4 Fehlermeldungen

Während eines Spiels können dem Benutzer verschiedene Warnungen und Fehler angezeigt werden. Diese sind da, um den Benutzer einen Aussagelräfiten Hinweis zu vermitteln.
Die folgenden Meldungen sind möglich:

- "Mister X hat gewonnen"
- "Detektive haben gewonnen"
- "Der Spieler kann sich nicht bewegen. Der Nächste Spieler ist dran!"
- "Auf der Station ist ein anderer Spieler!"
- "Station nicht erreichbar"
- "Spielstand konnte nicht geladen werden!"”
- "Spiel konnte nicht gespeichert werden!");

2. Programmierhandbuch

2.1 Entwicklungskonfiguration

Betriebssystem	JDK Version	IDE
Windows 10 Version 1909	8u241	Netbeans 8.2

2.2 Problemanalyse und Realisation

Folgende Probleme gab es in der Entwicklung und Konzeption zu lösen:

- Strukturierung und Aufteilung des Programms
- Umsetzung des Spielers und der Fraktionen
- Umsetzung des Spielfeldes und der Spielregeln
- Kommunikation zwischen GUI und der Logik
- Erstellung der GUI
- Steuerung auf der GUI und manipulation des Spielgeschehens
- Laden von Komponenten, welche für das Spiel Notwendig sind
- Laden und Speichern des Spieles und schreiben der Logs
- Sinnvolle Umsetzung und Einbindung der KI

2.2.1 Strukturierung und Aufteilung des Programms

Problemanalyse

Essentziell für ein gutes Programm ist natürlich die Interne Struktur und Aufteilung der Komponenten. Die wichtigste Frage ist hier, wie ich das Programm so aufteile um so gut wie möglich zu Kapseln, Wiederverwendbarkeit zu gewährleisten und Problemanalyse und Implementierung zu vereinfachen. Natürlich ist es genauso Essenziell, dass das Programm

im Rahmen der Objektorientierung sinnvoll aufgebaut ist und sich dem Spiel nahtlos anpasst.

Realisation Analyse

Ausgehend von den Tipps der Aufgabenstellung scheint es natürlich am Sinnvollsten die gesamte Logik von der Darstellung sauber zu trennen. Das ermöglicht einmal bessere Lesbarkeit und Struktur im Sinne der Objektorientierung und natürlich auf einfache Austauschbarkeit und Wartung der Komponenten. Abgesehen davon ermöglicht es effizienteres Testen und generell die Kapselung von so vielen Komponenten wie nur möglich.

Realisationsbeschreibung

Das Programm wird dementsprechend in 2 Pakete Aufgeteilt. Wir haben einmal die **GUI** die für die Darstellung des Spielfeldes und der Menüs zuständig ist und auf die Eingaben des Benutzers reagiert. Außerdem haben wir die **Logik** die das Gehirn des Spieles ist und auch die weiteren logischen Aufgaben erledigen soll.

- **GUI:** In der Gui sind 2 Komponenten natürlich Zentral. Einmal der Controller**,** dessen Aufgabe es ist eingaben der GUI an unsere Spiellogik zu übergeben und eine GUI Klasse, dessen aufgabe es ist verschiedene Teile der GUI zu Manipulieren. So erhalten wir einen nachvollziehbaren Strom in zwei Richtungen. GUI zu Logik und Logik zu GUI.

- **Logik:** Die Logik ist der Zentrale Punkt des Programms. Von hier aus wird das gesamte Verhalten des Programms gesteuert. Zentral ist hier die Klasse GameLogic, welche genau dieses verhalten umsetzen soll. Sie verbindet alle Komponenten die wir benötigen, erstellt und bindet diese in den Spielfluss ein, reicht Ergebnisse von anderen Komponenten weiter und protokolliert diese..

Ausgehend von diesen zwei Haupt-Paketen brauchen wir natürlich diverse andere Komponenten welche die Teil-Anforderungen umsetzen. Diese werden nachfolgend erläutert.

2.2.2 Umsetzung des Spielers und der Fraktionen

Problemanalyse

Genau wie die grundlegende Aufteilung des Programms ist natürlich auch die Struktur einzelner Komponenten und Teil-Anforderungen essenziell.

Welche Mechaniken und Komponenten setze ich in einer Klasse um ? Welche realisiere ich direkt im Spielfluss ? Wie rufe ich diese Klassen wo, warum und wie auf ? An welchen Stellen im Programm ist die Nutzung der Objektorientierung besonders von Vorteil und wo kann ich diese Vorteile besonders gut ausnutzen ?

Zentral ist der Spieler bzw. die Fraktionen im Spiel.

Realisationsanalyse

Gerade an der Stelle gibt es sicherlich unzählige Möglichkeiten das Problem zu lösen. Ich möchte natürlich so weit wie möglich gebraucht der Objektorientierung machen und sinnvoll implementieren. Demnach ist es am besten Komponenten des Brettspiels so umzusetzen, dass diese auch im Code genauso repräsentiert werden. Die Spieler sollten natürlich über Klassen realisiert werden, welche ihr Verhalten und die eigenarten Wiederspiegeln. Genauso sollten auch andere Komponenten gekapselt werden, welche sich mit dem Spielverlauf ändern oder und dabei helfen sollen diesen umzusetzen. Auch besondere Aufgaben wie zb Laden einiger Komponenten sollten so ausgelagert werden, dass sie in der Game Logik nicht deplaziert wirken und es schwerer machen die “Idee” zu verstehen. Es wäre natürlich auch möglich viele Dinge direkt in der Hauptklasse zu realisieren, doch dies wäre sehr Gefährlich, da es uns viel Schwerer machen würde das Programm sinnvoll zu strukturieren und vor allem auch viel Fehleranfälliger macht. Beispielsweise könnte man die gesamte Karte auch in der **GameLogic** Implementieren, nur würde dies zu immensen Chaos führen und viele Probleme mit sich bringen.

Realisationsbeschreibung

Der Spieler soll dabei eine eigene Klasse abbilden, welche genau das verhalten dieses Wiederspiegeln sollen. Außerdem muss ich unterscheiden zwischen einem Detektiv und Mister X aber auch alle gemeinsamkeiten vereinen. Dafür ist einer **Klasse Player** für alle Grundlegenden Dinge und Gemeinsamkeiten. Außerdem die **Klassen Mister** und **Detective**, welche von **Player** erben und auf die besonderheiten der Fraktionen eingehen und diese sinnvoll umsetzen. Hier wird also verwaltet wo ein Spieler sich befindet, welche Tickets er hat und wie wir diese Dinge verändern können und dürfen.

2.2.3 Umsetzung des Spiels und der Spielregeln

Problemanalyse

Wie schon erwähnt ist die **GameLogic** unser zentraler Anlaufpunkt des Spiels. Hier sollen also die Spielregeln implementiert werden und die einzelnen Komponenten verbunden werden. Die Frage die sich stellt ist, wie diese Klasse sinnvoll strukturiert wird, so dass wir den Spielfluss für alle Szenarien realisieren können (so allgemein wie möglich) und auch Anforderungen umsetzen die über dem Spielfluss stehen, wie z.B. ein Spiel zu Laden statt neu zu Starten.

Realisationsanalyse

Dies ist der Punkt an dem meiner Meinung nach die Meisten verschiedenen Umsetzungen möglich sind. Schon die Tatsache, dass das Spiel auch durch eine KI gesteuert werden kann macht klar, dass es wichtig ist, dass ich so Strukturiert, dass der Spielfluss allgemein und Logisch nachvollziehbar gehalten wird. Die Klasse soll aber trotz ihrer Zentralen Rolle immer noch nachvollziehbar sein und das Spiel geschehen sinnvoll abbilden.

Realisationsbeschreibung

Für einen Spielstart muss ich unterscheiden zwischen einem neuen Spiel und einem geladenen Spiel.
Nach dem initialen Laden sollen aber beide Möglichkeit ineinander übergehen, so dass ein geladenes Spiel keine eigene Logik braucht. Deswegen wird genau zwischen den beiden Möglichkeiten als Spielstart unterscheidenden, während der Rest gleich bleiben soll. **LoadGame** und **StartGame**. Anschließend soll das Spielgeschehen beginnen, welches im Prinzip immer gleich ist. Der aktuelle Spieler wird aufgerufen, der aktuelle Spieler macht seinen Zug, der nächste Spieler ist dran. Ob dieser jetzt von einem Menschen oder der Ki gesteuert wird, soll sich nur in den Unterschieden bemerkbar machen, nicht aber in der Grundfolge des Spiels.
An der Stelle ich es natürlich noch wichtig, dass an den korrekten Stellen des Flusses diverse Regeln überprüft und umgesetzt werden (z.B. ob ein Spieler gewonnen hat).

2.2.4 Kommunikation zwischen GUI und der Logik

Problemanalyse

Wie bereits gesagt soll hier natürlich eine gewisse Kapselung sinnvoll umgesetzt werden, so stellt sich die Frage wer mit wem Kommunizieren darf. Da in beide Richtungen kommuniziert werden soll, ist die Struktu hier sinnvoll zu wählen.

Realisationsanalyse

Hier ist die in der Aufgabenstellung gestellte Hilfe die mit Abstand sinnvollste. Um die Möglichkeit zu haben die GUI Komponenten theoretisch zu Kapseln macht es durchaus Sinn, ihre Verwendung und Funktionalität in einem Interface zu deklarieren, so dass dieses dann Implementiert werden muss um als API zu fungieren.
Das hat den Vorteil, dass unsere Logik nur über diese API kommunizieren muss und kann und man somit die nötige Kapselung erhält.
Man muss natürlich auch in die andere Richtung Kommunizieren, was am sinnvollsten mit dem Controller umgesetzt werden sollte.

Realisationsbeschreibung

Konkret heißt das, dass die Logik über das Interface **GUIConnector** mit der GUI kommuniziert und diese über den **FXMLDocumentController** auf eingaben reagiert und diese an die Logik weitergibt.

2.2.5 Erstellung der GUI

Problemanalyse

Die Erstellung dieser ist relativ Straight forward durch die Kapselung die eingehalten werden soll. Der Punkt der zu Diskutieren ist, ist wie GUI Spezifische besonderheiten umgesetzt werden sollen. Beispielsweise wenn sich ein neuer Dialog (wie beim erstellen eines neuen Spiels öffnet.

Realisationsanalyse

Hier gab es direkt zwei möglichkeiten die erstmal Sinnvoll erschienene. Wenn sich ein neuer Dialog bzw ein neues Fenster öffnet, so erzeugt unser Controller den Dialog einfach temporär. Oder wir lagern den Dialog in einen eigenen Controller aus, der sich dann um diesen kümmert und vom “Haupt” Controller aufgerufen wird.
Mehrere FXML Dateien in einem Controller zu verwalten ist zwar möglich, aber nicht sinnvoll und schlecht umsetzbar. Dementsprechend habe ich mich dazu entschieden neue Fenster in eigenen Controllern zu verwalten

Realisatoinsbeschriebung

Jeder Dialog erhält einen eigenen Controller und natürlich eine eigene FXML Dateie. Führt etwas zum öffnen dieses Dialogs, so wird unser **FXMLController** eine Instanz des **DialogController** erzeugen und präsentieren.
Wichtig ist auch hier, dass theoretisch in beide Richtungen kommuniziert werden muss. Der neue Controller braucht also Infos vom Hauptcontroller und muss diese auch wieder an diesen zurückgeben.
Deswegen muss jeder Dialog Controller auch eine Instanz der **FXMLDocumentController** halten. z.B.

2.2.6 Steuerung auf der GUI und manipulation des Spielgeschehens

Problemanalyse

Während die Kommunikation von Controller zur Logik schon fast Intuitiv wirkt, so stellen sich in die andere Richtung jedoch Probleme. Unsere Logik soll über ein interface zur GUI

kommunizieren und so z.B. den dargestellten Ticket Vorrat aktualisieren. Was ist aber zu tun, wenn dieser ein Feedback zurück zur Logik senden muss. Beispielsweise wenn der Spieler Auswählen soll, welches Ticket er benutzen will.

Realisationsanalyse

Hier gibt es mehrere Möglichkeiten dies Sinnvoll umzusetzen. Ein umweg über den Hauptcontroller wäre durchaus denkbar, da dieser ja zugriff auf GUI und Logik hat. Auf der anderen Seite stellt die Auswahl des Tickets eine besondere Form von Dialog da, die sich auf Reaktion der Logik öffnet und nicht auf Reaktion der GUI (so wie bei neues Spiel).
Unter berücksichtigung dieser Tatsache erscheint es mir sinnvoll den Dialog durch das Interface zu verwalten, so dass dieser einen Dialog Controller aufruft und dieser wiederum in dem speziellen Fall zurück zur Logik kommunizieren darf (indem wir ihm die Logik übergeben).

Realisationsbeschriebung

Unsere Implementation des GUI Interfaces, **JavaFXGUI** erzeugt den Ticket Dialog Controller, welcher sich bei seiner Initialisierung selbst präsentiert und eine Referenz zur **GameLogic** erhält.
Nach auswahl des Tickets gibt dieser die information an die Logik weiter.

2.2.7 Laden von Komponenten, welche für das Spiel Notwendig sind

Problemanalyse

Wie Laden wir die Spielkomonenten Sinnvoll und bilden deren Struktur ab. Die Frage stellt sich vor allem bei dem Laden des Netzes. Wo das laden der Karte zur GUI oder der Spielfiguren trivial sind und sich aus den Controllern bzw der JavaFXGUI ergeben, so ist das laden des Netzes weitaus komplizierter. Hier ist nicht nur die Frage wie ich etwas lade, sonder wie ich etwas durchaus komplexeres als ein Bild lade. Das Netz hat ja eine eigene Struktur die erstmal im Programm umgesetzt werden muss bevor diese geladen werden. Die frage die sich auch noch stellt ist, ob die Struktur so wie sie ist auch sinnvoll für unsere Logik ist.

Realisationsanalyse

Wie schon gesagt wäre es zwar möglich dies direkt in unserer GameLogic umzusetzen, aber definitiv nicht sinnvoll. Es ist also notwendig das eine eigene Klasse die Struktur des Netzes abbildet, so dass man das Netz in die Struktur laden können.

Dafür ist eine Klasse für die Struktur notwendig aber auch für die einzelnen Bestandteile der Struktur. So hat ein Netz viele Stationen, die wiederum haben Positionen, die bestehen wiederum aus Koordinaten. Es ist also durchaus sinnvoll für jede Teilstruktur auch eine Klasse abzubilden sofern diese in irgendeiner Form Abstrakt ist und nicht ein einfacher Wert. Am anfang war ich durchaus der Meinung, dass die geladenen Struktur in ihrere Form durchaus ausreichend für unsere Logik ist, dennoch habe ich bei der Umsetzung schnell bemerkt das ihr Hinweis intern die Stationen per Referenz abzuspeichern der bessere Vorschlag ist, da danach deutlich leichter ist mit dieses zu arbeiten.

Realisationsbeschriebung

Konkret habe ich eine eigene Klassen für das Netzs (**NetJSON** und **Net**), die Stationen (**StationJSON**, **Station**) und Position (**Position**) deklariert um so jeweils eine Klasse für die JSON Form als auch für die interne Repräsentation für die Logik zu erhalten. Nur bei **Position** war dies nicht Notwendig wegen der primitiven Datentypen. Der der GameLogic lade ich also Daten in die JSON Repräsentation und wandel diese durch die Konstruktoren der Klassen für die interne Repräsentation um.

2.2.8 Laden und Speichern des Spieles und schrieben der Logs

Problemanalyse

Auch hier stelle sich die Frage, wie man am sinnvollsten eine Struktur umsetzen kann, die man speicher, laden und relativ einfach in das Spiel importieren kann. Außerdem gibt es auch viele Möglichkeiten etwas zu laden und zu speichern bzw. Logs zu schreiben.

Realisationsanalyse

Für das Laden und Speichern bietet sich das empfohlene GSON sehr gut an. Dies ermöglicht es einfach JSON Strukturen abzuspeichern und zu laden. Natürlich sollte auch dafür eine sinnvolle Struktur als Klasse umgesetzt werden, so dass das Format direkt übertragen werden kann.
Auch die Logs sollten eine eigene Klasse erhalten. Dies ist zwar weniger Notwendig als beim Speicher/Laden, da das Format relativ simpel als String in einen Stream reingeschrieben wird, trotzdem ist es im Sinne der Objektorientierung genau dies auszulagern.

Realisationsbeschreibung

Für die Logs habe ich eine Log Klasse erstellt die mit einfachen Methoden den Anfang, die Zwischenteile und das Ende in einen **Log** schreiben kann!.Für das speichern und laden habe ich auch eigene Klasse gewählt, die der JSON Repräsentation entsprechen. So gibt es eine Klasse **SaveGame**, **MisterSave** und **DetectiveSave**.

2.2.9 Sinnvolle Umsetzung und Einbindung der KI

Problemanalyse

Einmal ist es entscheidend wo wir in der Gamelogik wie gebrauch von der KI machen ohne unsere Grundstruktur komplett zu verändern, außerdem ist die Struktur für die KI sehr entscheidend, da es offensichtlich mehrere möglichkeiten gibt diese einzubinden.

Realisationsanalyse

Für mich kamen 4 Möglichkeiten in Frage. Entweder bekommt die KI eine eigene Klasse, was sehr gut für die Kapselung ist und auch Sinnvoll für die Objektorientierung aber dadurch auch viele parameter benötigt.
Die KI hätte aber auch in der Logik umgesetzt werden könne, da wir dort zugriff auf die meisten Komponenten haben und somit das herumreichen einzelner Parameter am einfachsten wäre, da sich der Großteil der Logik genau dort befindet.
Noch eine möglichkeit wäre es natürlich in den Klassen für die Fraktionen zu implementieren. Dies hätte den großen Vorteil das wir die KI einfach für den aktuellen Spieler ausführen können und je nach Fraktion automatisch der korrekte Algorithmus gewählt wird.
Die letzte und auch die am wenigsten sinnvolle Methode wäre es in der Klasse des netzes zu machen. Dies hätte den großen Vorteil, dass wir viele direkte Netz Methoden direkt mitbenutzen können aber ehrlich gesagt wirkte es auf mich deplaziert. Ich schätze eine eigene Klasse für die KI am sinnvollsten ein, da sie für mich ein guter Kompromiss aus allen beschriebenen Vorteilen bildet.

Realisierungsbeschreibung

Ich habe eine Klasse **Ai** umgesetzt, die bei der Initialisierung die Komponenten bekommt die sie braucht. So kann ich in der GameLogic an den richtigen Stellen einfach zwischen Mensch und KI unterscheiden und der KI über Parameter die aktuelle Spielsituation übergeben, so dass sie für mich die Optimale nächste Station berechnet.

2.2.10 Design der KI

Problemanalyse

Im vergleich zu den meisten Methoden sind viele Berechnungen der KI sehr komplex und benötigen daher besondere Aufmerksamkeit. Ist es sinnvoll wirklich alles in der KI zu machen ? Oder gibt es Dinge die ich vorher ausrechnen sollte um sie dann an die KI zu übergeben ? Wie unterscheide ich zwischen Mister X und Detektiv. Und wie berechne ich die teilweise komplexen Anforderungen für die verschiedenen Taktiken.

Realisierungsanalyse

Die grenzen der Umsetzung sind wahrscheinlich Endlos so habe ich mich für einen Hybriden entschieden. Es macht schon Sinn das meiste für das Ergebnis der Ki auch dort berechnen zu lassen aber die “Möglichen Zielpositionen” für Mister X haben da eine Sonderstellung. Diese gehören meiner Meinung nach in die Gamelogik, da diese zwar von Mister X sind, aber keine Angelegenheit seiner Klasse sind.
Im Brettspiel sind das die Positionen die wir uns als Spieler selbst herleiten und deswegen macht die Berechnung dieser in der GameLogic als Parallele berechnung zum Spielgeschehen am meisten Sinn.
So kann ich die Ergebnisse an die KI übergeben und dort den Rest erledigen.

Realisierungsbeschreibung

In der **Ai** Klasse unterscheide ich zwei Methoden einmal für die Detektive und einmal für Mister X. In ihnen werden dann mit den Parametern der aktuellen Spielsituation und den Referenzen aus der Initialisierung der Ai Klasse die verschiedenen Taktiken durchgespielt und bewertet.

2.2.11 Kürzeste Route berechnen

Problemanalyse

Welchen Algorithmus benutze ich um eine Kürzeste Route in unserem Netz zu finden.

Realisierungsanalyse

Ich habe mich hier auf Grund der theoretisch vielem möglichkeiten (z.B. Dykstra) an die Hilfestellung gehalten, da diese für mich nach einen guten Kompromiss von Komplexität der Implementierung und Effizienz sind.

Realisierungbeschreibung

Den Algorithmus habe ich in die **Net** Klasse ausgelagert, da sie in dieser Klasse am Sinnvollsten erscheint. Demnach werden ausgehend von der Station alle Nachbarn durchprobiert und bewertet.
Außerdem wird für jeden Nachbarn die Station gespeichert aus der wir kommen, danach werden die Stationen als besucht markiert.
Der Algorithmus wird so lange immer wieder für jeden Nachbarn durchgeführt bis unsere Zielstation markiert wurde. Mit dem abspeichern der letzten Station können wir dann von der Zielstation zurück zur Ausgangssituation gereicht werden um so die nächste Station zu bestimmen.
Wichtig ist hier auch die Tickets mit einzubeziehen und auch ob Stationen besucht sind. Außerdem hat eine Nachbarstation ja Teilweise die gleichen Nachbarn wie eine vorangegangene Station. So werden immer nur Stationen durchlaufen, die noch gar nicht besucht wurden.

2.2.12 Berechnung der Möglichen Zielpositionen

Problemanalyse

Auch die Berechnung der möglichen Zielpositionen erwies sich als nicht trivial. Zwar war es Sinnvoll dies in der GameLogic zu tun und an die KI zu übergeben, trotzdem ist die berechnung dieser essentiell für die KI und deswegen ist die Korrektheit extrem wichtig.

Realisierungsanalyse

Eine Möglichkeit wäre die Möglichen Zielpositionen für den aktuellen und letzten Zug abzuspeichern. Nur ist es für mich sinnvoller alle möglichen Zielpositionen für jede runde abzuspeichern. Dies hat den Vorteil der besseren Wartbarkeit und lässt sich besser abspeichern.

Realisierungsbeschreibung

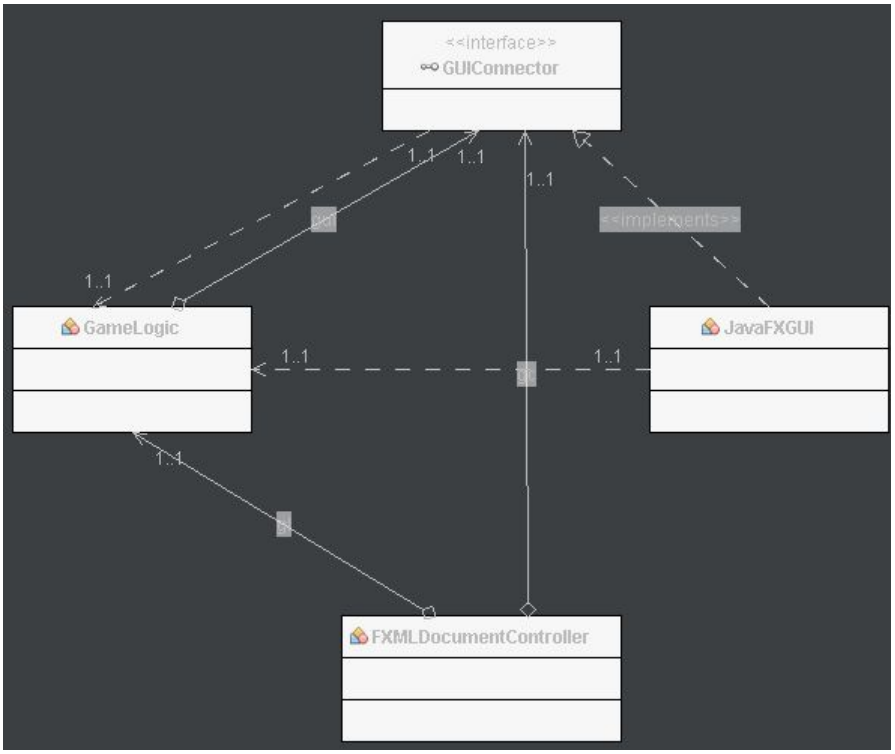
Deswegen wird für die berechnung ausgehend von der letzten Zeigeposition von Mister X (also ab Runde 3) jede mögliche Station berechnet auf der er sein kann. Das bedeutet auch, dass es keine Stationen sein darf auf die er nicht reisen kann, die besetzt sind oder die unmöglich sind von der Rundenanzahl her. Wenn Mister X in Runde 3 auf Station 4 ist, so kann er in Runde 4 zwar auf den umliegenden Stationen sein, nicht aber nochmal auf 4. Dies muss natürlich auch berücksichtigt werden. Demnach habe ich einen Array angelegt der für jede Runde ein Set von Stationen abspeichert.

Diese werden auf der Grundlage der Stationen der vorherigen Runde berechnet und anhand der eben genannten kriterien filtern.

3. Programmorganistationsplan

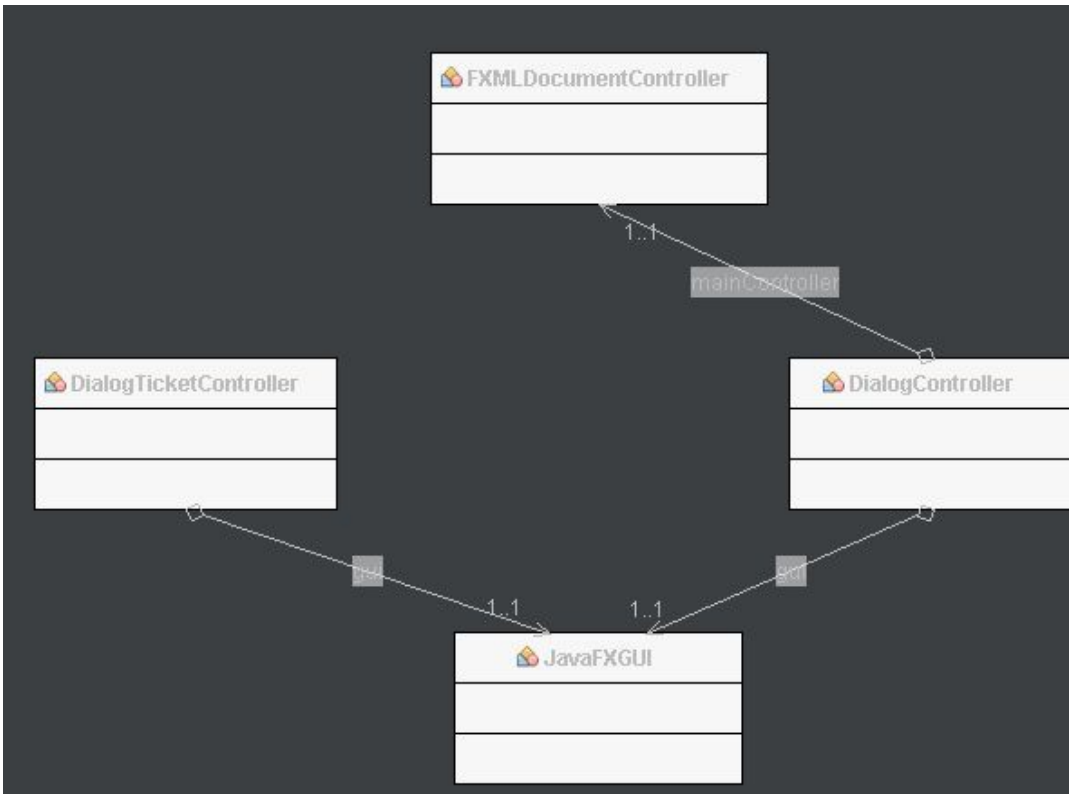
Zur Darstellung als UML wurde das Netbeans Plugin easyUML benutzt. Zur vereinfachung der Sachverhaltes habe ich die Attribute ausgeblendet und einzelne Sachverhalte als eigenes UML Dargestellt:

Struktur GUI und Logik



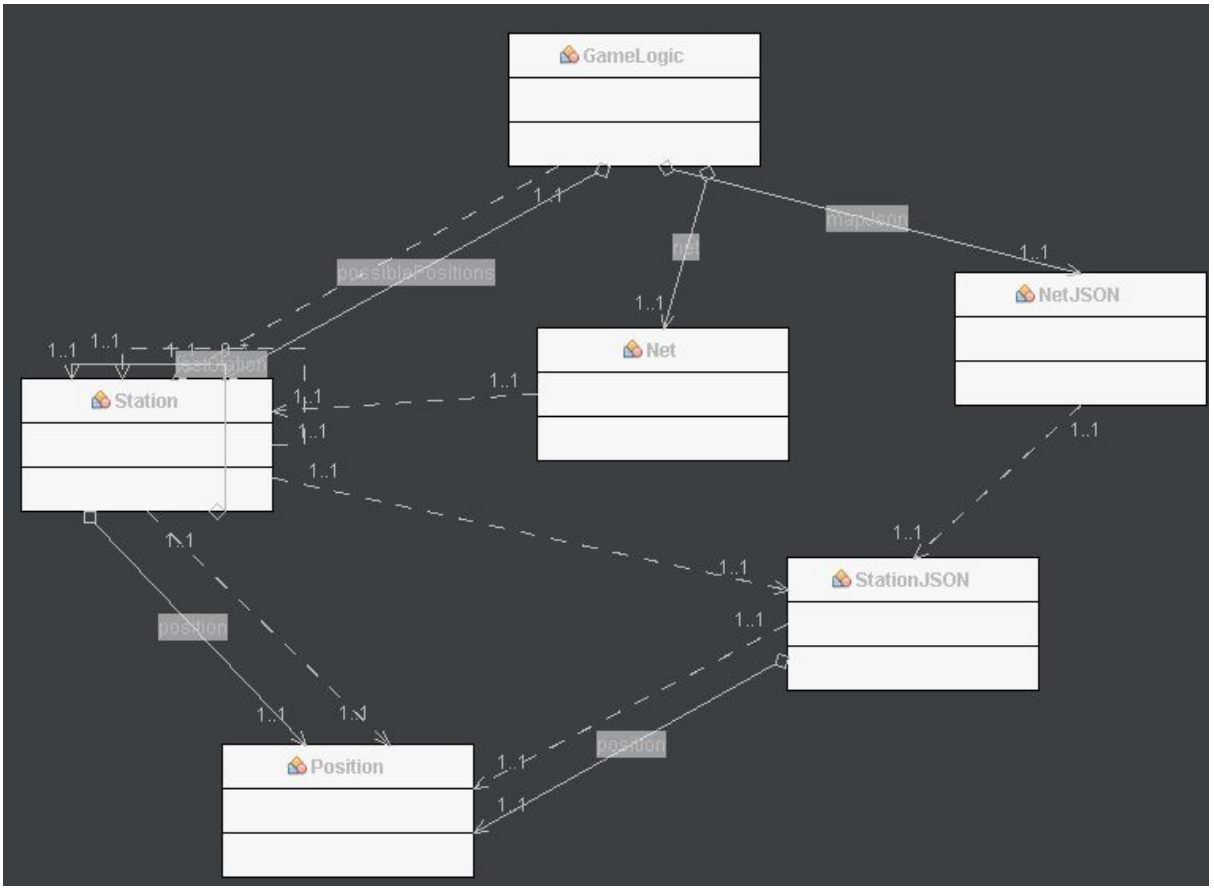
Hier ist zu sehen wie die Gamelogik nur die GUIConnector hält und das kommunizieren zu GUI über das Interface umgesetzt wird,
Außerdem hat den Controller natürlich Zugang zug Logik und GUI.

GUI Intern



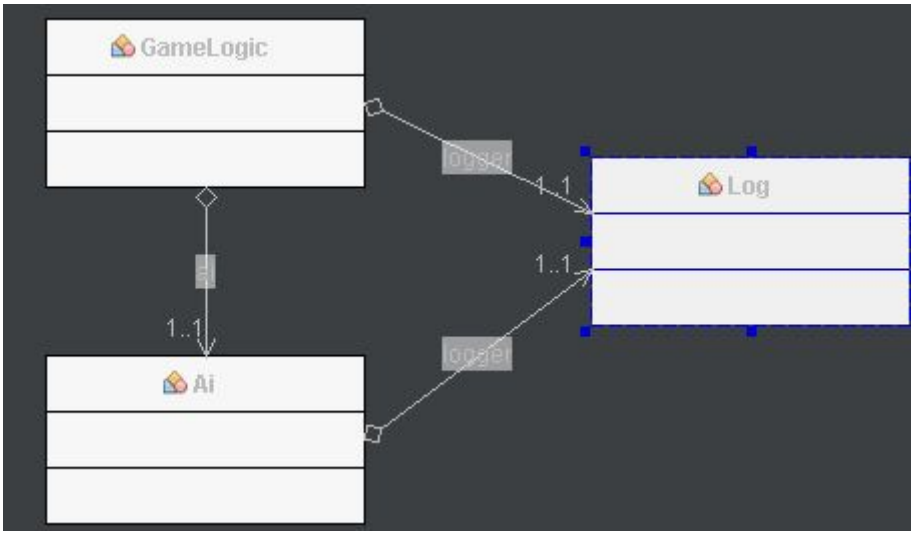
Der Hauptcontroller kann den DialogController Öffnen und die FXGUI den Ticket Controller.

Logik und JSON



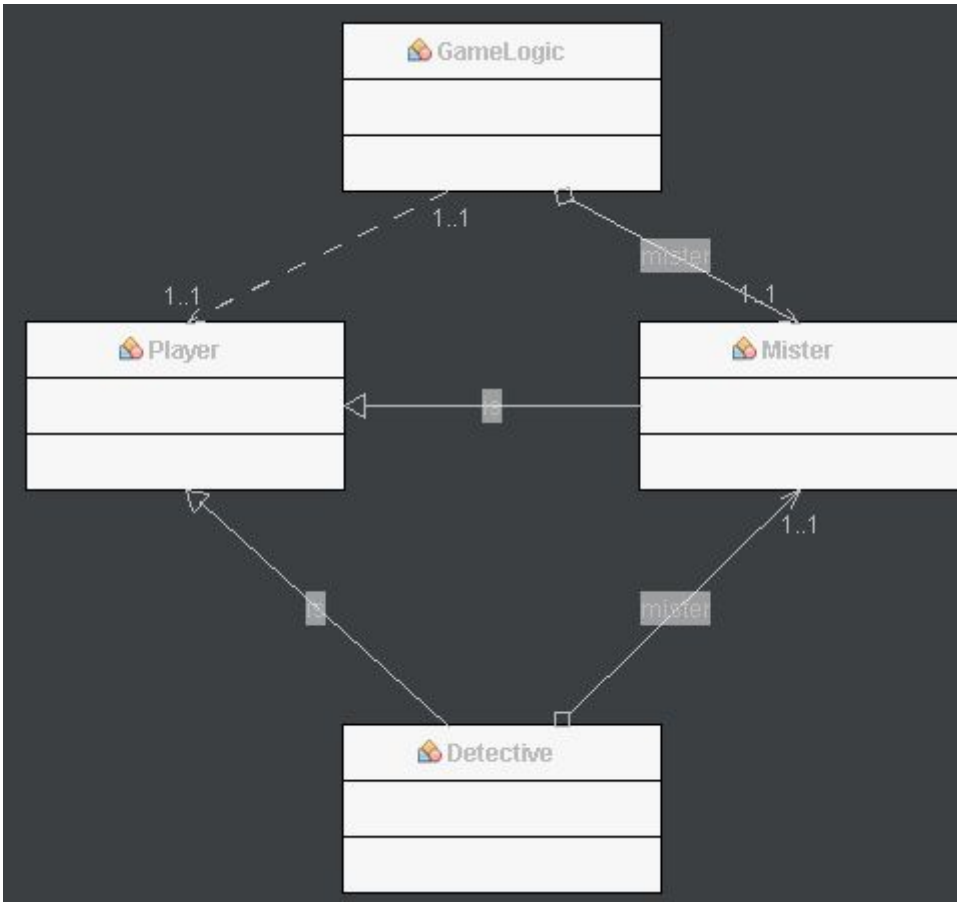
Die Logik hat den Zugang zu dem Netz, der JSON Repräsentation und zu Stationen. Positionen sind innerhalb der Stationen vertreten.

Logik, KI und Log



Hier ist gut zu erkennen wie Sowohl die KI als auch die Logik beide in die Logs schreiben.

Logik und Spieler



Während die Logik stets eine Referenz von Mister X hat, so werden die Detektive nur zusammen mit ihm als Array in Players verwaltet. Direkte Referenzen auf die Detektive haben keinen nutzen im Programm.

4. Beschreibung Grundlegender Klassen

4.1 Logik

4.1.1 NetJSON

Die NetJSON Klasse bildet eine Teilstruktur für die Abbildung des Netzes aus der JSON. Sie ist exakt so konstruiert, dass sie genau zu der übergebenen JSON Struktur passt.
In ihr wird ein array von **StationJSON** abgelegt um die Stationen des Netz zu speichern.

4.1.2 StationJSON

Die Klasse schließt sich der **NetJSON** an und bildet jede Station ab, die im JSON Format gespeichert wird. Sie entspricht der übergebenen JSON Struktur und hält damit für jede Station die übergebenen Attribute.

4.1.3 Position

Die Klasse Position speichert eine Position mit **x** und **y** Koordinaten als double. Sie wird grundlegend dafür benutzt, um eine Position jeder Station auf der Karte umzusetzen,

4.1.4 Net

Die Klasse Net bildet die interne Struktur für das Netz der Spielkarte ab. Eine **NetJSON** wird in die Klasse **Net** umgewandelt, da diese statt **IDs** für jede Station **Referenzen** abspeichert.
In ihr finden wir einmal einen Array von Stations **stations,** welcher alle Stationen des Netzes abbildet. Außerdem wird die Anzahl aller Stationen in **numberOfStations** gespeichert, um das spätere arbeiten und Iterieren über die Stationen zu vereinfachen.
Zusätzlich wird ein Maximaler Abstand **maxDistanceBetween** definiert, welcher für die bestimmung der Station welche am nähesten an den übergebenen Koordinaten dran ist.
Die folgenden Methoden beziehen sich auf Konkrete Berechnungen innerhalb des Netzes, deswegen habe ich auch Net als Ort für diese gewählt.

In der Methode **getClosestStationTo** wird die Station bestimmt welche den übergebenen Koordinaten am nächsten ist. Die Methode ist für die Eingabe des Users auf der Karte unerlässlich.

Die Methode **calcDistanceStations** bildet auch eine wichtige Funktion im Netz ab. Sie berechnet den Abstand zwischen zwei Stationen.

In der Methode **shortestPath** wird im Netz der kürzest mögliche Weg zwischen zwei Stationen gesucht. Sie gibt dem User die Nächste Station auf dieser Route zurück und ist ein wichtiger Bestandteil für spätere KI Berechnungen.

4.1.5 Station

Station beschreibt die interne Darstellung einer Station innerhalb des Netzes. Statt IDs für die Stationen abzuspeichern wie in der JSON repräsentation, speichert diese Referenzen auf die anderen Stationen. Abgesehen davon soll sie trotzdem die eigenart einer JSONStation grundlegend abbildet.
Da das füllen dieser Referenzen nicht zum Zeitpunkt der erstellung passieren kann (weil dann noch nicht alle Stationen existieren könne), trägt die Klasse die Methode **fillAllStations**, welche die Referenzen für alle Stationen einträgt um diese später einfach nutzbar zu machen. Ohne diese Methode wäre die Klasse also nicht sinnvoll nutzbar.

4.1.6 Player

Die Klasse Player bildet einen Spieler in Scotland Yard ab. Sie ist damit die Grundlegende Struktur für Mister X und auch die Detektive und trägt somit alle Gemeinsamkeiten der Fraktionen. Dementsprechend kann die aktuelle Station **currentStation** und die nächste Station **nextStation** abgespeichert werden.
Außerdem lassen sie die verschiedenen Tickets eines Spielers speichern (**taxiTickets, busTickets, undergroundTickets**) und ob der Spieler von der KI kontrolliert werden soll oder nicht.
Da die Klasse nur die Gemeinsamkeiten repräsentieren sollen, können hier keine Black Tickets gespeichert werden. Trotzdem werden auch getter und setter für diese Leer initialisiert, so dass diese in den Kinder Klassen überschrieben werden können.
Neben den üblichen Gettern und Settern sorgt die Methode **movePlayerToNextStation** für die Umsetzung einer Reise auf der Karte.
Abgesehen davon wird in der Klasse auch das benutzen von den diversen Tickets umgesetzt. So stellt die Methode **useBusTicket** Beispielsweise das nutzen eines Tickets für den Bus da und reduziert dementsprechend den Vorrat.

4.1.7 Detective

Die Klasse Detective ist eine Kindklasse von **Player**. Sie setzt die Besonderheiten eines Detektivs um.
Die größe Funktion dieser Spezialisierung besteht darin bei benutzen des Tickets das Ticket an Mister X zu “übergeben”. Aus dem Grund hält ein Detektiv auch eine Referenz auf **Mister** und überschreibt die **useTicket** Methoden der Elternklasse.

4.1.8 Mister

Genau die Detective ist die Klasse Mister eine Kindklasse von **Player**. Sie soll die besonderheiten von Mister X umsetzen und ist dementsprechend eine Spezialisierung eines Spielers.
Da Mister X als einziger “Black Tickets” halten kann werden diese hier auch entsprechend gespeichert.
Außerdem werden in der Klasse die “Letzte Zeigepositionen” von Mister X abgespeichert, welche für viele Berechnungen in der Logik essentiell sind. So speichert **turn3Station** z.B. die Station, an der sich Mister X in Runde 3 befindet.
Zusätzlich hält die Klasse alle Werte für die “Fahrtentafel” in einem array **billboardValues**. Damit ist es einfach die verschiedenen Tickets für die Fahrtentafel zu verwalten.

In der Methode **isMisterVisible** wird zusätzlich ausgegeben, ob Mister X aktuell sichtbar ist oder nicht.

Die Methode **updateMisterVisibility** aktualisiert die eben genannten Werte und wird somit in jeden Zug von Mister X aufgerufen.

4.1.9 MisterSave

Die Klasse MisterSave bildet die JSON Struktur zum speichern der Informationen von Mister X ab. Sie hält alle grundlegenden Informationen (Positionen, Tickets, Fahrtentafelwerte) um sie für das Speichern zu formatieren.

4.1.10 DetectiveSave

DetectiveSave hat die gleiche Aufgabe die **MisterSave** und bildet eine einfach JSON Struktur zum speichern der wichtigsten Informationen der Detektive ab.

4.1.11 SaveGame

SaveGame ist die “Hauptklasse” für das Speichern eines Spielstandes. Sie bindet **DetectiveSave** und **MisterSave** mit zusätzlichen Informationen ab um das Speichern in eine JSON Datei zu ermöglichen. So werden hier noch zusätzlich Informationen gehalten wie z.B. die aktuelle Runde, ob und welche Fraktion von einer KI gespielt wird und ob das Spiel bereits beendet ist oder nicht.

4.1.12 Log

Die Log Klasse hat den zweck einen minimalistischen Log über das Spielgeschehen zu speichern.
Sie besteht wesentlich aus den drei Methoden **writeLogBeginning, writeLogForTurn** und **writeLogForGameEnd** um die drei möglichen Logausgaben abzubildet. So werden hier die entsprechenden Parameter übergeben um diese dann korrekt in die Log datei zu speichern. Die klasse und ihre Methoden kann somit immer genau dort aufgerufen werden, wenn der Log das Spielgeschehen abbildet soll.

4.1.13 Tickets

Das Enum Tickets ist ein einfaches Enum um die verschiedenen Möglichkeiten der Tickets Klassenübergreifend zu vereinheitlichen. So gibt es **Bus, Taxi, Underground, Black** und auch **None** um kein Ticket zu beschreiben.

4.1.14 Ai

Die Ai Klasse ist die Grundlegende Klasse für die KI berechnungen des Spiels. Sie hält alle wichtigen Referenzen wie **net, players, playerCount, mister** und auch eine Referenz für das schreiben der Logs **logger**.

Die zwei Hauptmethoden sind **detectiveAi** und **misterAi** welche die KI Berechnungen für die jeweilige Fraktion realisieren sollen. So gibt **detectiveAi** Beispielsweise die Station zurück, auf die sich der aktuell von der KI kontrolliere Detektiv bewegen soll.

Um diese Komplizierten Berechnungen übersichtlich zu gestalten, werden die verschiedenen Taktiken und deren Bewertungen in eigene Methoden umgesetzt. So berechnet **tactic1** beispielweise einen möglichen Zug für den Detektiv, die Methode **evaluateTactic1** bildet die erste Bewertungstaktik ab. So das **detectiveAi** alle Taktiken durchlaufen kann, diese Bewerten kann, mit **evaluateDetTactivs** zusammenrechnen kann

und schließlich mit **calculateWinnderDet** die “beste” Taktik bzw Station bestimmt wählen kann.
So wird das für alle 4 Taktiken und Bewertungen der Detektive gemacht.

Im Gegensatz dazu, ist der Aufbau von **misterAi** sehr ähnlich. Die Taktiken werden durchprobiert, bewertet und die bestbewertete wird genommen. Im Gegensatz zu den Detektiven, sind die Taktiken und Bewertungen nicht in Hilfsmethoden ausgelagert. Der Grund dafür ist einmal, dass diese deutlich weniger Komplex sind, aber vor allem einander benötigen. So erhalte ich beim berechnen der Taktik 1 für Mister X direkt Werte die ich für taktik 2 an der selben Stelle benutze. Aus effizienzgründen sind diese deshalb zusammengefasst.
Um einen gewinner für Mister X zu bekommen, werden die Bewertungen trotzdem in einer Methode **calculateWinnerMister** übergeben und der beste ausgewertet.

Zusätzlich zu erwähnen sind die Hilfsmethoden **stationsPlayersHaveAsNeighboar** und **stationsPlayerHasAsNeighboar** welche für den bzw die Spieler die Nachbarn berechnet, welche theoretisch Mögliche Zielpositionen von Mister X sein könnten.

Die Methode **getStationWithSmallestId** gibt uns die Station mit der kleinsten ID aus einem Set von Stationen zurück.

Die Struktur der Klasse ermöglicht es die Ki sehr einfach in den Spielfluss einzubinden. So werden ihre Methoden einfach für den aktuellen Spieler je nach Fraktion aufgerufen und geben uns die nächste Station für unseren KI gespielen aktuellen Spieler wieder.

4.1.15 GUIConnector

Das Interface GUIConnector ist ein Interface was die Realisierung der Kommunikation von Logik zur GUI umsetzt und die Grundbausteine dafür vorgibt.
Jede Manipulation in der GUI ausgehend von der Logik wird hier dementsprechend deklariert und durch entsprechende Klassen **JavaFXGUI** implementiert.

So können nachfolgend alle Manipulation der GUI in der Logik ausgeführt werden, indem die nachfolgenden Methoden einfach aufgerufen werden und trotzdem die Kapselung zwischen GUI und Logik eingehalten wird.

So finden wir in ihr die Methode **openTicketChooserDialog** dessen Aufgabe es sein wird einen Dialog in der GUI zu öffnen, damit ein Spieler auswählen kann, welches Ticket er benutzen soll (sofern mehrere verfügbar sind).p
Die Methoden **initsFigures** und **setFigureAt** sollen das Initialisieren bzw. Platzieren der Spielfiguren auf der Karte Realisieren.

Außerdem wird die Methode **displayInfo** den User über diverse Status Informationen in Form eines Alerts benachrichtigen.

Die Methode **setStatus** wird abhängig von der Runde und dem aktuellen Spielern genau diese Informationen in der GUI anzeigen.

Auch das Manipulieren der Fahrtentafel wird mit der Methode **setTicketAt** realisiert.

4.1.16 GameLogic

Die GameLogic bildet den zentralen Punkt der gesammten Spieles. Sie verwaltet die gesamte Spiellogik, setzt die Eingaben über die GUI um bzw delegiert diese und gibt Ergebnisse an diese weiter.
Dementsprechend hält die Klasse Referenzen zu fast allen beschriebenen Klassen. Sie verbindet die Spieler **players** und bindet sie in das Sielgeschehen ein. Sie initialisiert das Netz **net** bzw lädt die Informationen der JSON Dateien aus den **JSON** Klassen in die Korrespondierenden internen Klassen.
Sie hält außerdem ein Array von Stations Sets **possiblePositions**, welche alle Möglichen Zielpositionen von Mister X für jeden Zug im Spiel speichern soll.

Genereller Start bzw Ausgangspunkt sind die Methoden **startNewGame** bzw. **loadNewGame**, welche ein neues Spiel installieren und den Spielfluss starten. Abhängig von den Parametern werden hier auch die Spieler initialisiert, übergeben ob diese von der KI gespielt werden sollen und wie diese miteinander agieren werden.

Die Methode **loadPlayerToGuiAndMakeTurn** bildet einen Hauptschritt des Spielflusses ab. Einmal wird wie der Name suggeriert der aktuelle Spieler in die GUI geladen, außerdem wird danach auf Usereingaben gewartet (also der Klick auf die Karte) oder aber der zug der KI umgesetzt.
Die entsprechenden Verzögerungen der Animationen für die KI werden an der Stelle über keyFrames realisiert und blockieren weitere Eingaben bzw. heben diese Blockierung wieder auf.

Sollte die aktuell spielende Fraktion eine KI sein, so wird nun die Methode **performAi** aufgerufen, welche zusammen mit den durch die Methode **calcPossiblePositions** berechnetet Möglichen Zielpositionen von Mister X über die Klasse **Ai** die nächste Station ausrechnen lässt.

Das Ergebnis wird anschließend an die Methode **checkAndMoveToStation** weitergereicht. Für Menschliche Spieler würde der Klick auf der Karte dieselbe Methode aufrufen, so das die gleiche Methode sowohl für KI als auch Mensch nutzbar ist.
Sie soll das Ergebnis der Klicks bzw. der KI berechnung validieren und entscheiden ob alles legitim ist, der Spieler genug Tickets hat und auch welche Tickets benutzt werden können. So steuer sie auch, dass der **tickerChooserDialog** in der Gui geöffnet wird, sollten mehrere Tickets möglich sein.

Anschließend wird **checkAndMoveToStation** entweder direkt (im Falle der KI) oder über die Ticket Choose Eingabe (für den Fall das mehrere Tickets möglich sind) aufgerufen.
Nachdem wir in **checkAndMoveToStation** die Station und die umliegenden Informationen validiert haben wir jetzt in **checkAndMoveToStation** der eigentliche Zug umgesetzt, also das bewegen auf die nächste Station und das benutzen des korrekten Tickets. Ist der Zug vorüber wird **updateFiguresAndNext** aufgerufen.

In der Methode **updateFiguresAndNext** werden zumal die neuen Positionen an die GUI übertragen und aktualisiert (Im falle der Ki mit Keyframes um Animationen zu gewährleisten bzw. pausen).
Die Methode **nextTurn** wird schließlich aufgerufen, welche den Übergang eines Zuges zum nächsten repräsentiert. Sollte der letzte Detektiv dran gewesen sein, so wird die nächste Runde eingeleitet und schließlich wieder **loadPlayerToGuiAndMakeTurn** aufgerufen.

Die Struktur dieser Methoden gewährt das aufteilen eines Spielzuges in sinnvolle Zwischenschritte welche die besonderheiten eines Spielflussen gewähren und eine einfache unterscheidung zwischen mensch und Ki gewährleisten.
Außerdem können so beispielsweise das schreiben der Logs einfach an den entsprechenden Stellen umgesetzt werden.

Neben dem Hauptspielfluss finden wir Methoden wie **saveGame** bzw. **loadGame** zum speichern und laden von Spielständen und diverse Helper wie **closestStationToAveragePossibleStation,** welche die nächste Station an der durchschnitts Möglichen Zielposition von Mister X berechnet oder z.B. **shuffleArray** welche einen Array mit Hilfe des Fisher Yates Algorithmuses mischt, um zufällige Startpositionen auszuwählen.

4.2 GUI

4.2.1 FXMLDocumentController

Die Controller Klasse ist der Hauptcontroller für die GUI. Hier werden einmal die Logik und Gui initialisiert und aneinander übergeben und vor allem auf Eingaben der GUI reagiert.
So finden wir hier einmal alle FXML Variablen für alle Objekte der Gui (also Labels, Buttons, ImageViews etc.) als auch Methoden die Eingaben an die Logik weiterreichen sollen.

So wird mit der Methode **startNewGame** ein neues Spiel in der Logik angesetzt oder mit **handeMpuseClickPane** ein Klick in der Karte in Form von Koordinaten an die Logik weitergereicht.
Außerdem werden bei der Initialisierung der JavaFXGUI (welche das Interface GUIConnector implementiert) alle notwendigen FXML Objekte weitergegeben.

Eine weiter Aufgabe ist zb das öffnen eines weiteren Controllers mit **newGameClicked.** Die Methode initialisiert einen weiteren Controller, welcher in dem Fall den Neues Spiel Dialog kontrolliert.

4.2.2 DialogController

Der DialogController ist für den Neues Spiel Dialog zuständig. Er initialisiert nötige Checkboxen und reagiert auf die Auswahl des Users.
Sie hält außerdem eine Referenz des **FXMLDocumentController** um in der Action Methode **startClicked** die Auswahl des users zurück an den Hauptcontroller und so schließlich zur Logik zu übertragen.

4.2.3 DialogTicketController

Der Controller ist für den Ticker Chooser Dialog verantwortlich. Also der Dialog der erscheint wenn ein User auswählen soll, welches Ticket er für den zug nutzen soll. Da er im Gegensatz zum DialogController nicht über den FXMLDocumentController sondern über die JavaFXGUI initialisiert wird, hält der Controller eine direkte Referenz zur Logik, so das die Ticket Chooser eingabe direkt an die Logik übergeben werden kann.
So reagiert die Methode **busTicketClicked** auf die Action des Users, also das Klicker auf den dazugehörigen Button und ruft anschließend die **moveToStation** der Logik auf.

4.2.4 JavaFXGUI

Die Klasse JavaFXGUI implementiert das Interface GUIConnector.

4.2.5 TestGUI

Die Klasse JavaFXGUI implementier das Interface GUIConnector. Sie hält nur leere Implementationen um das Testen auch ohne eine GUI zu realisieren.

4.3 Zusätzliche Dateien

4.3.1 Logik/data

netz.json
Die Netzstruktur in JSON Format.

4.3.2 gui/images

Bilder der Tickets:

- **BlackTicket.jpg**
- **Bus.jpg**
- **Taxi.jpg**
- **Underground.jpg**

Spielplan.jpg
Die Spielkarte.

Spieler (detektive):

- **newPlayer1.png**
- **newPlayer2.png**
- **newPlayer3.png**
- **newPlayer4.png**
- **newPlayer5.png**

mister.png
Mister X Icon.

5. Programmtests

5.1 GUI Tests

Testfall	**Ergebnis**
Korrektes Verhalten aller Menüpunkte prüfen	Alle Buttons wie Start Game, Load Game, Cheat Monde führen zum gewünschten ergebnis
Klick auf die Karte während die KI am Zug ist	Der Click wird solange nicht registriert wie der andere Spieler noch am Zug ist
Markieren der nächsten Station	Ein Klick auf die Karte lässt stets die nächste Station durch einen Kreis markieren

Prüfen ob Stationen markiert werden kann wenn Mister X von einer KI gespielt wird und nicht zu sehen ist	Es werden keine Stationen markiert wenn Mister X nicht sichtbar ist (von der KI gespielt wird)
Ein Spieler wird auf eine Station geschickt, die bereits besetzt ist	Der Spieler kann sich nicht zur ausgewählten Station bewegen und erhält eine Info Nachricht über das Problem
Ein Detektiv versucht eine Boots route zu nehmen	Der Detektiv bewegt sich nicht über das Wasser
Das Spiel wird während eines KI Spieles gespeichert. Danach wird ein neues Spiel gestartet und es werden einige Runden gespielt. Anschließend wird das gespeicherte Spiel mitten im Zug geladen	Das geladene spiel wird geladen und automatisch fortgesetzt
Ein Spieler hat keine Tickets mehr für ein Verkehrsmittel und versucht aber sich über dieses Fortzubewegen	Der Spieler kann den Weg nicht passieren
Der Benutzer versucht eine Datei zu laden die kein Savegame ist	Dem User wird eine Warnung in der GUI präsentiert

5.2 Logik Tests

Nach den Umbau der Logik war ein Großteil meiner geplanten Tests leider unbrauchbar, weswegen ich zum größten Teil manuell testen musste und auch wollte, da die Spielsituationen komplexer wurden als ich anfangs vermutet habe (gerade nach der Implementation der KI). Über Savegames und mit “beobachtung” der GUI waren diese besser nachvollziehbar.
Mir ist natürlich bewusst, dass das nicht die Norm sein sollte und programmierte Tests die bessere Lösung sind. Leider habe ich es in der Zeit nicht geschafft diese ein weiteres mal zu schreiben.
Dementsprechend habe ich die Härtefälle aufgelistet, welche ausgiebig getestet wurden:

Testfall	**Ergebnis**
Ein Detektiv wird von der KI gesteuert und alle Nachbar Stationen bis auf eine ist besetzt	Die korrekte und freie Station wird gewählt
Für Mister X werden die möglichen Ziel Stationen berechnet, während manche von denen besetzt sind	Die besetzen Stationen gehören nicht zu dem Set der Zielstationen

Die kürzeste Route wird berechnet, wobei die kürzeste Route über eine Bahn Station führen würde und der Spieler keine U-Bahn Tickets mehr hat	die kürzeste Route verläuft nicht über die Bahnstation
Analyse der einzelnen Taktiken und deren Wirksamkeit	Es wird stets die korrekte Taktik ausgewählt
Die KI hat mehr als ein mögliches Ticket für die reise	Es wird das Ticket benutzt, welches am meisten Vorhanden ist
Ein Spieler wird von der KI gesteuert, hat keine Tickets mehr und wird übersprungen	Der Spieler wird im Verlauf übersprungen
Alle Spieler werden von der KI gespielt und der Benutzer macht UI eingaben	Alle Eingaben auf der Karte werden ignoriert
Es wird ein kürzester Weg berechnet und der Weg ist nicht passierbar	Es wird keine Station über die Taktik gefunden und dementsprechend bewertet
Der Spieler verfügt noch über Tickets, hat aber keine um von der aktuellen Station zu entkommen	Der Spieler wird übersprungen und darauf hingewiesen (wenn Mensch)
Für einen Detektive werden die erreichbaren Zielpositionen (für alle Detektive) ausgerechnet, während die “anderen” Detektive theoretisch auf die Position könnten, aber über keine Tickets dafür verfügen	Die Zielpositionen sind nicht als erreichbar für Detektive markiert, wenn diese nicht die nötigen Tickets haben
Die möglichen Zielpositionen werden auch nach geladenen Spiel weiter berechnet	Die Zielpositionen sind immer noch korrekt und wurden korrekt geladen
Die möglichen Zielpositionen werden für Züge berechnet die die Detektive nicht kennen können (Runde 2 auf Basis von Runde 1)	Die Position von Runde 1 fließt nicht in die Berechnung der Zielpositionen ein, da die Detektive Mister X erst in Runde 3 sehen
Bei der Berechnung für die “Durchschnitts Station” der letzten Zielpositionen wird durch 0 geteilt (Set war leer)	Es wird kein Fehler ausgelöst, da der Fall abgefangen wurde. Die Berechnungen werden erst ab Runde 3 zugelassen und lassen kein teilen durch 0 zu
Die KI kann überhaupt keine Station bestimmen	Der Zug wird nicht wiederholt, da dies bedeutet dass der Spieler sich nicht bewegen kann oder darf