

Dokumentation der praktischen Arbeit

zur Prüfung zum
Mathematisch-technischen Softwareentwickler

Thema: Simulation von Verkehrsflüssen

16.05.2025

Daniel Ebel

Prüfungsnummer: 101 20015

Bearbeitungszeitraum: 12.05.2025 - 16.05.2025

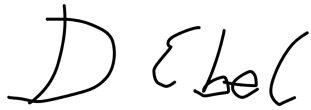
Programmiersprache: Java

Ausbildungsort: INFORM GmbH

Eigenständigkeitserklärung

Ich erkläre, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung als Prüfungsleistung ausschließt.

Aachen, 16.05.2025

A handwritten signature in black ink, appearing to read 'D. Ebel', written over a horizontal line.

Unterschrift

Inhaltsverzeichnis

1	Einleitung	3
2	Aufgabenanalyse	4
2.1	Beschreibung Eingabedatei	4
2.2	Ausgabeansforderungen	5
3	Verfahrensbeschreibung	8
3.1	Einlesen	8
3.2	Datenhaltung	14
3.3	Simulation	16
3.4	Ausgabe	22
3.5	Gesamtablauf	25
4	Abweichungen vom ersten Entwurf	27
5	Tests	29
5.1	Fehlerflle	29
5.2	Grenzflle	40
5.3	Normalflle	44
5.4	Ausfhren der Tests	44
5.4.1	BadInputTestRunner	44
6	Erweiterbarkeit	46
7	Projektstruktur	48
8	Benutzeranleitung	50
9	Entwicklungsumgebung	51
10	Zusammenfassung und Ausblick	52

Kapitel 1

Einleitung

Zur Bewertung von Verkehrsflüssen und zur Identifikation kritischer Straßenabschnitte soll eine Verkehrssimulation entwickelt werden. Grundlage der Simulation ist eine Eingabedatei mit Einfallspunkten und Kreuzungspunkten sowie ein definierter Simulationszeitraum. Fahrzeuge treten an Einfallspunkten mit bestimmten Taktraten auf, bewegen sich mit zufälliger, individuell festgelegter Geschwindigkeit durch das Netz und treffen an Kreuzungen zufallsbasiert Abbiegeentscheidungen gemäß festgelegter Wahrscheinlichkeiten.

Die Eingabe erfolgt über eine Textdatei mit folgendem Beispielinhalt:

```
1 # Beispielhausen
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 C 0 2 B 5
8 D 4 0 E 3
9 F 4 2 E 2
10 G 5 1 E 3
11
12 Kreuzungen:
13 B 0 1 A 20 C 30 E 50
14 E 4 1 D 20 F 20 G 10 B 50
```

Die erste Zeile gibt einen Kommentar wieder. Danach folgen der Simulationszeitraum (hier 50 Sekunden mit Ausgaben alle 1 Sekunde), gefolgt von Einfallspunkten. Jeder Einfallspunkt enthält Position, Zielkreuzung und Taktzeit. Anschließend werden Kreuzungen definiert, die jeweils eingehende Straßen mit Richtungsverteilungen enthalten.

Ziel der Simulation ist es, Fahrzeugbewegungen zu modellieren, deren Positionen für jeden Zeitschritt in einer Datei ausgegeben und mit einem Visualisierungstool dargestellt werden. Zusätzlich werden für jede Straße statistische Kenngrößen wie kumulierte Fahrzeuganzahl und maximale Streckenauslastung berechnet.

Kapitel 2

Aufgabenanalyse

2.1 Beschreibung Eingabedatei

Die Eingabedatei muss UTF-8-kodiert sein; anderenfalls wird eine Exception mit der Meldung „Eingabedatei ist nicht UTF-8 kodiert“ ausgelöst. Die Datei gliedert sich in exakt drei Abschnitte: Zeitraum:, Einfallspunkte: und Kreuzungen:. Jeder Abschnitt darf höchstens einmal vorkommen – doppelte Abschnitte führen zu einem Abbruch mit einer spezifischen Fehlermeldung.

Im Abschnitt Zeitraum: werden zwei Parameter erwartet: die Gesamtdauer der Simulation in Sekunden (maxTime) sowie die Taktfrequenz (clockrate), also das Intervall in Sekunden, in dem Fahrzeuge generiert werden. Beide Werte müssen positive Ganzzahlen sein. maxTime darf dabei 24 Stunden (86.400 Sekunden) nicht überschreiten. Ein Eintrag wie z.,B. „50.0“ würde eine Fehlermeldung hervorrufen, da Dezimalzahlen nicht erlaubt sind. Auch muss clockrate kleiner oder gleich maxTime sein.

Der Abschnitt Einfallspunkte: definiert Punkte, an denen Fahrzeuge in das System eintreten. Jede Zeile folgt dem Format Name X Y Ziel Takt. Dabei dürfen Namen maximal 100 Zeichen lang sein und nicht doppelt vorkommen. Die Koordinaten X und Y müssen Gleitkommazahlen im Bereich von -1000 bis +1000 sein. Zudem wird geprüft, dass kein Punkt näher als 0,1 Einheiten (entspricht 10 Metern) zu einem bereits definierten Punkt liegt. Als Ziel wird eine Kreuzung angegeben, zu der das erste Fahrzeug weitergeleitet wird. Der Takt bestimmt die Häufigkeit der Fahrzeuggenerierung in Sekunden und muss eine positive ganze Zahl sein.

Im Abschnitt Kreuzungen: wird jede Zeile im Format Name X Y Ziel1 Anteil1 Ziel2 Anteil2 ... erwartet. Auch hier sind Namen auf 100 Zeichen begrenzt und dürfen nicht mehrfach vorkommen. Die Ziel-Anteil-Paare müssen mindestens zweimal vorkommen, wobei die Anzahl der Teile ungerade sein muss (wegen Name, X und Y zu Beginn). Der Anteil (relative Wahrscheinlichkeit) muss als Dezimalzahl mit Punkt angegeben sein und zwischen 10^6 und 10^{-6} liegen. Auch hier dürfen Koordinaten nicht zu nahe beieinanderliegen.

Orte dürfen nicht gleichzeitig Einfallspunkte und Kreuzungen sein. Zudem müssen alle in Einfallspunkten oder Kreuzungen genannten Zielorte tatsächlich existieren – entweder als Kreuzung oder als anderer Einfallspunkt.

2.2 Ausgabeanforderungen

Ausgabe

Für jede Simulation soll ein eigener Ordner angelegt werden, dessen Name aus dem Präfix `output_`, gefolgt vom Namen der Eingabedatei ohne Dateiendung, besteht. Beispielsweise wird die Ausgabe der Datei `Beispielhausen.txt` im Ordner `output_Beispielhausen` gespeichert. Die Ausgabe des Programms besteht aus drei Dateien. Falls im Ordner bereits gleichnamige Dateien vorhanden sind, werden diese überschrieben. Der neuerzeugte Ordner befindet sich auf gleicher Ebene, wie die ausführbare `.jar`.

1. **Die Datei `Plan.txt`** beinhaltet die Koordinaten der Straßen. Jede Zeile besteht dabei aus vier Zahlen, die die x - und y -Koordinaten der Start- und Zielpunkte jeder Straße angeben.

Für das folgende Beispiel hat die Datei `Plan.txt` folgenden Inhalt:

```
0.0 0.0 1.0 0.0
1.0 0.0 2.0 0.0
2.0 0.0 3.0 0.0
0.0 0.0 0.0 1.0
0.0 1.0 0.0 2.0
0.0 2.0 0.0 3.0
0.0 3.0 1.0 3.0
1.0 3.0 2.0 3.0
2.0 3.0 3.0 3.0
3.0 3.0 3.0 2.0
3.0 2.0 3.0 1.0
3.0 1.0 3.0 0.0
3.0 0.0 4.0 0.0
4.0 0.0 5.0 0.0
4.0 0.0 4.0 1.0
4.0 1.0 4.0 2.0
4.0 2.0 4.0 3.0
4.0 3.0 5.0 3.0
5.0 3.0 5.0 2.0
5.0 2.0 5.0 1.0
5.0 1.0 5.0 0.0
```

2. **Die Datei `Statistik.txt`** beinhaltet Statistiken zur Streckenauslastung jeder Straße. Es sollen zwei Werte pro Straße angegeben werden:
 - *Gesamtanzahl Fahrzeuge pro 100 m*: Die Summe aller Fahrzeuge, die den jeweiligen Abschnitt befahren haben, normiert auf 100 m.
 - *Maximale Anzahl Fahrzeuge pro 100 m*: Die höchste gleichzeitige Anzahl an Fahrzeugen auf diesem Straßenabschnitt, ebenfalls normiert auf 100 m.

Beispielhafte Ausgabe der Datei Statistik.txt:

Gesamtanzahl Fahrzeuge pro 100 m:

A -> B: 203.0
B -> A: 37.0
B -> C: 84.0
C -> B: 87.5
C -> D: 129.0
D -> C: 128.0
E -> F: 40.0
F -> E: 97.5
E -> G: 125.0
G -> E: 120.0

Maximale Anzahl Fahrzeuge pro 100 m:

A -> B: 10.0
B -> A: 2.0
B -> C: 6.0
C -> B: 2.0
C -> D: 8.0
D -> C: 8.0
E -> F: 2.0
F -> E: 4.75
E -> G: 8.75
G -> E: 4.0

Für die Ausgabe der Plan.txt und Statistik.txt wurde zusätzlich festgehalten, dass die Ausgabe sortiert nach den Namen der Orte (Einfallspunkte oder Kreuzungen) geschehen soll, sowie in den oben angegebenen Beispielen.

Das geschriebene Programm berechnet die Gesamtanzahl der Fahrzeuge wie folgt: Die Summe aller Fahrzeuge, die den jeweiligen Abschnitt befahren haben, normiert auf 100 m. Mit dieser Methode lassen sich die Gesamtanzahlen aus den Beispielen nicht erreichen. Für die Beispiel-Gesamtanzahlen wurden vermutlich die Fahrzeuge kumuliert über alle Zeitschritte summiert und anschließend auf 100 m normiert. Von dieser Methode wird abgewichen: Fahrzeuge werden auf einem Streckenabschnitt nur einmal gezählt. Die Gesamtanzahl sollte nicht von der gewählten allgemeinen Taktrate abhängen.

3. Die Datei **Fahrzeuge.txt** enthält pro Zeitschritt alle sichtbaren Fahrzeuge mit:

- aktuellen Koordinaten,
- Koordinaten der nächsten Zielkreuzung,
- sowie fortlaufender Fahrzeug-ID.

Jeder Zeitschritt beginnt mit einer Zeile der Form:

```
*** t = <Zeitpunkt>
```

Beispielhafte Ausgabe:

```
*** t = 0
4.0 2.0 4.0 1.0 0
4.0 0.0 4.0 1.0 1
...

*** t = 5
1.6215051939730939 3.0 4.0 1.0 0
1.626456506716547 3.0 4.0 1.0 1
...
```

Visualisierung

Die Simulation kann mit dem bereitgestellten Tool **Plot.py** visualisiert werden. Die Zielkreuzungen steuern die Fahrspurwahl (rechts oder links), und die Fahrzeug-ID bestimmt die Farbgebung. Für jeden Zeitschritt wird eine PNG-Datei erstellt.

Kapitel 3

Verfahrensbeschreibung

3.1 Einlesen

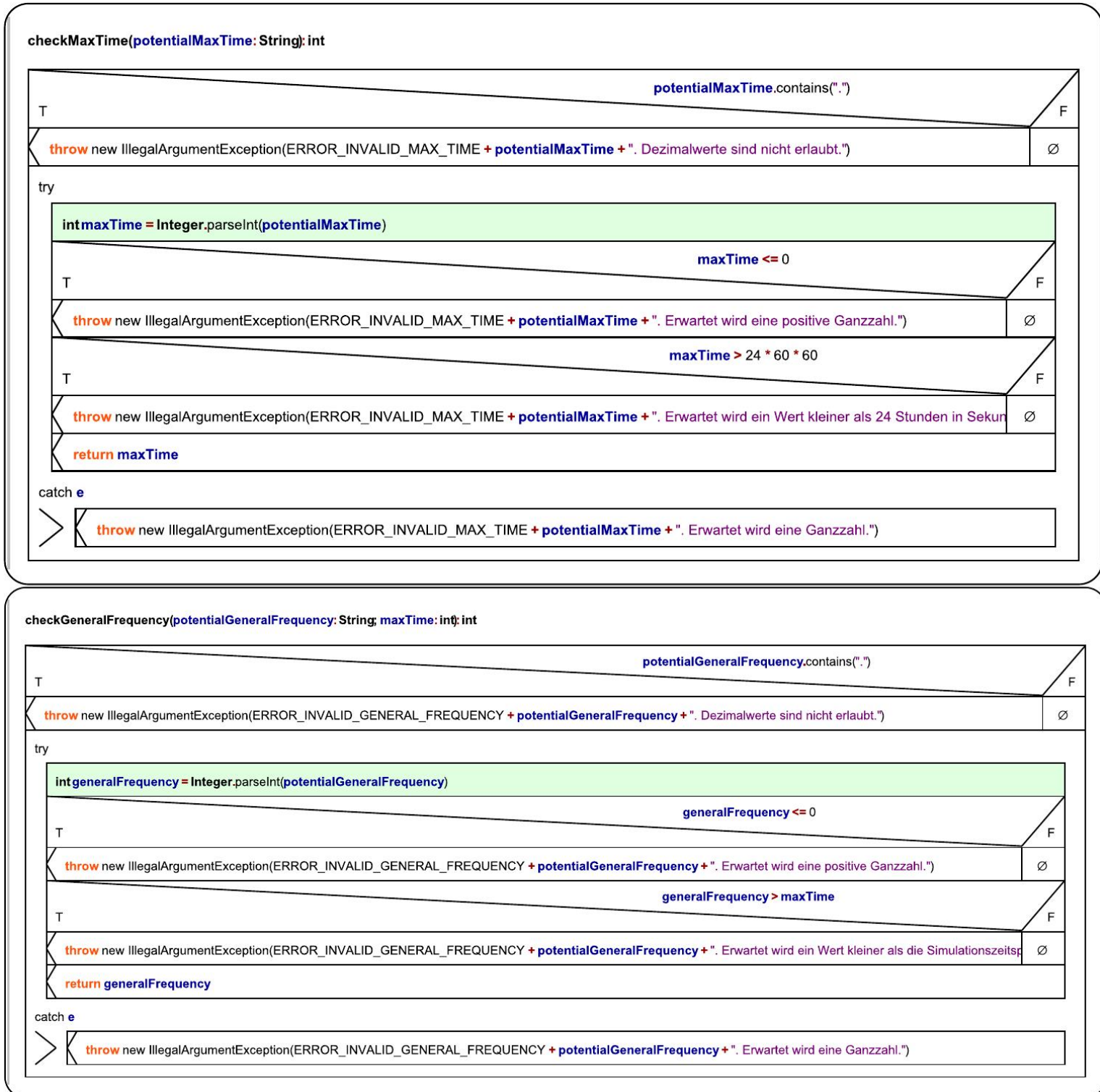
Die Klasse `TextFileReader` ist für das strukturierte Einlesen und Validieren der Eingabedatei zuständig. Sie implementiert das Interface `Reader` und liefert mit der Methode `read()` ein vollständig initialisiertes `CityDTO`-Objekt, das alle für die Simulation notwendigen Daten kapselt.

Der Leseprozess beginnt mit einer Prüfung der Dateixistenz sowie der korrekten UTF-8-Kodierung. Anschließend wird die Datei zeilenweise eingelesen. Kommentare (durch `#` eingeleitet) werden entfernt, leere Zeilen ignoriert. Die Datei ist in drei klar strukturierte Abschnitte gegliedert:

- **Zeitraum:** Enthält zwei ganzzahlige Werte:
 - `maxTime` – Die gesamte Simulationsdauer in Sekunden.
 - `clockRate` – Das Intervall in Sekunden, in dem der Zustand aller Fahrzeuge gespeichert wird.

Es wird sichergestellt, dass beide Werte gültige positive Ganzzahlen sind. Dezimalzahlen und ungültige Werte führen zu klaren Fehlermeldungen.

Abbildung 3.1: Zeitraum Prüfmethode in Nassi-Shneiderman-Diagrammen dargestellt:



- **Einfallspunkte:** Jede Zeile beschreibt einen Einfallspunkt im Format:

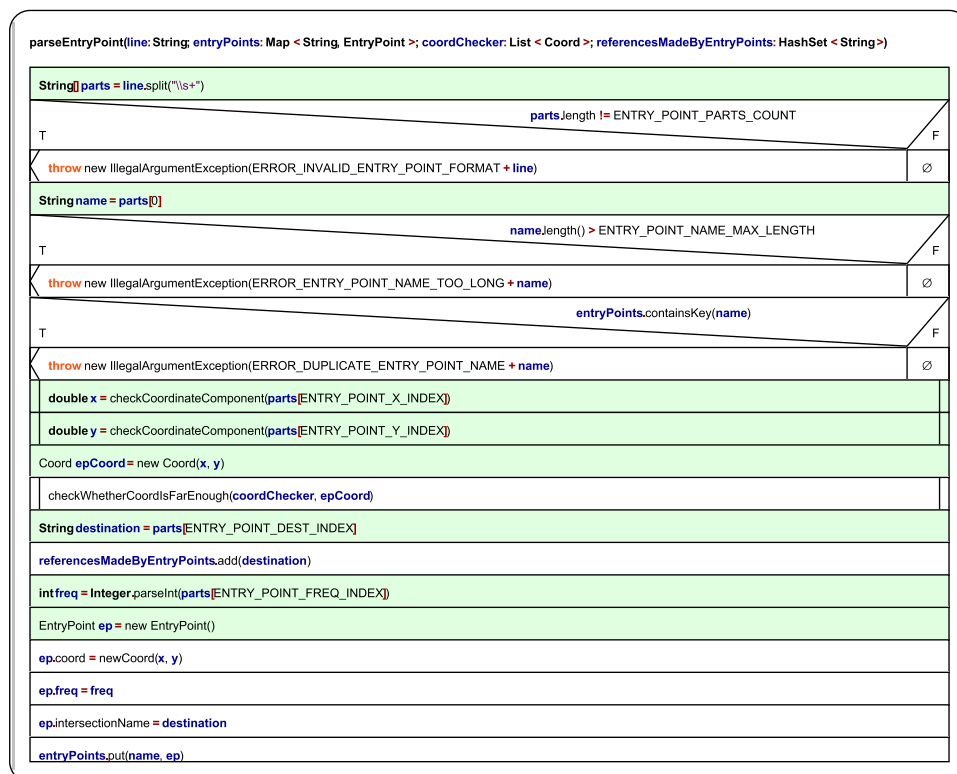
<Name>	<x>	<y>	<Zielkreuzung>	<Frequenz>
--------	-----	-----	----------------	------------

Dabei stehen **x** und **y** für die Koordinaten, **Zielkreuzung** für das Ziel neu erzeugter Fahrzeuge, und **Frequenz** für das Erzeugungsintervall. Es wird validiert, dass:

- jeder Name eindeutig ist,
- die Zielkreuzung später im Kreuzungsabschnitt existiert,
- Koordinaten einen Mindestabstand (0,1 Einheiten) zueinander einhalten,
- die Frequenz eine positive Ganzzahl ist.

Jeder gültige Eintrag wird als `EntryPoint`-Objekt gespeichert.

Einfallspunkt Parsemethode in Nassi-Shneiderman-Diagramm (bitte reinzoomen):



- **Kreuzungen:** Jede Zeile definiert eine Kreuzung und deren ausgehende Kanten:

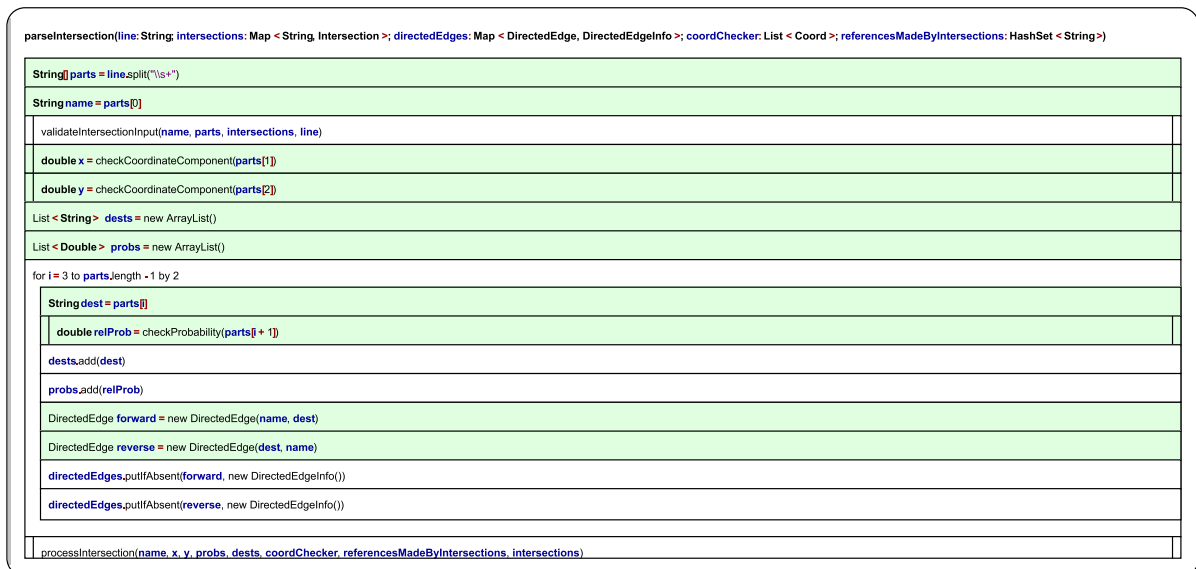
`<Name> <x> <y> <Ziel1> <Anteil1> <Ziel2> <Anteil2> ...`

Dabei werden:

- Koordinaten (x , y) eingelesen,
- Zielorte mit zugehörigen relativen Wahrscheinlichkeiten versehen,
- Wahrscheinlichkeiten automatisch normalisiert.

Für jede Verbindung wird ein `DirectedEdge` (inklusive Gegenrichtung) angelegt. Die zugehörigen statistischen Informationen werden in einem `DirectedEdgeInfo`-Objekt gespeichert. Jede Kreuzung wird als `Intersection`-Objekt abgelegt.

Kreuzungs Parsemethode in Nassi-Shneiderman-Diagramm (bitte reinzoomen):



Nach Abschluss des Parsings werden verschiedene Konsistenzprüfungen durchgeführt:

- Einfallspunkte und Kreuzungen dürfen nicht überlappen,
- Alle referenzierten Orte müssen existieren,
- Wahrscheinlichkeiten und Koordinatenwerte müssen im zulässigen Bereich liegen.

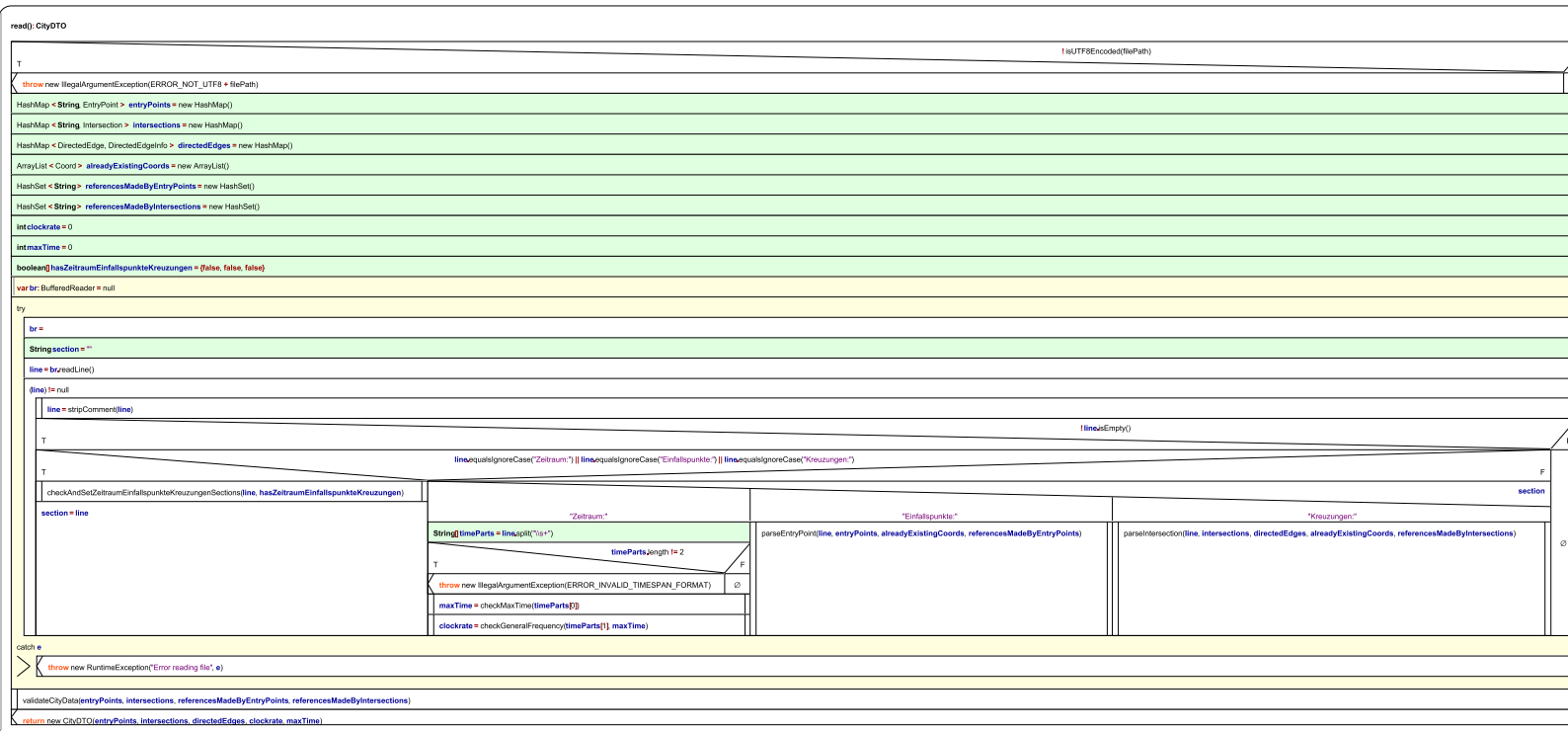
Abschließend werden alle Daten in ein `CityDT0`-Objekt überführt, das als Grundlage für die Initialisierung der Simulation dient.

Auf der folgenden Seite lässt sich in das Nassi-Shneiderman Meisterwerk der, das `CityDT0` erzeugenden `read`-Methode hineinzoomen.

Hinweis

In den folgenden Nassi-Shneiderman-Diagrammen sind Java `try-with-resource`-Statements aufgrund von Limitationen der Nassi-Shneiderman-Diagramm-Erstellungssoftware *Structorizer* wie folgt dargestellt:

```
var br: BufferedReader = null
try
...
br =
catch e
```



3.2 Datenhaltung

Die Datenhaltung der Simulation erfolgt über Klassen, welche die Elemente des Verkehrsnetzes sowie den Zustand der Simulation abbilden.

- **CityDTO**

Dieses Datenübertragungsobjekt (*Data Transfer Object*) bündelt alle zur Initialisierung der Simulation notwendigen Informationen:

- `entryPoints` – Einfallspunkte als Map von Namen auf `EntryPoint`-Objekte,
- `intersections` – Kreuzungen als Map von Namen auf `Intersection`-Objekte,
- `directedEdges` – Gerichtete Kanten zwischen Knoten, jeweils mit zugehörigem `DirectedEdgeInfo`,
- `clockRate` und `maxTime` – Zeitparameter der Simulation.

- **Coord**

Diese Klasse repräsentiert zweidimensionale Koordinaten. Sie enthält Methoden zur Vektoroperation (Addition, Subtraktion, Normalisierung, Skalierung) sowie zur Berechnung von Distanzen. Die Koordinaten dienen als Grundlage für Bewegungsrichtungen und Standortspeicherungen.

- **DirectedEdge**

Modelliert eine gerichtete Verbindung zwischen zwei Punkten im Verkehrsnetz. Zwei Felder (`from`, `to`) beschreiben die Richtung. `equals()` und `hashCode()` sind überschrieben, um die Kante als Schlüssel in HashMaps verwenden zu können.

- **DirectedEdgeInfo**

Diese Klasse hält statistische Informationen zu jeder Kante:

- `totalCount` – Gesamtanzahl aller Fahrzeuge, die die Kante jemals befahren haben,
- `currentNum` – Aktuelle Anzahl an Fahrzeugen auf der Kante,
- `maxNum` – Maximalanzahl an Fahrzeugen, die gleichzeitig auf der Kante waren.

Methoden wie `increment()`, `decrement()` und `updateMaxNum()` aktualisieren die Werte während der Simulation.

- **EntryPoint**

Stellt einen Punkt dar, an dem Fahrzeuge ins Netz eintreten. Neben der Position (`coord`) enthält das Objekt den Namen der Zielkreuzung sowie die Erzeugungsfrequenz (`freq`) neuer Fahrzeuge.

- **Intersection**

Repräsentiert eine Kreuzung im Verkehrsnetz. Enthält eine Position und ein `NamesAndProbabilities`-Objekt, das die möglichen Ausfahrten mit Wahrscheinlichkeiten beschreibt. Methoden wie `getNewDestinationByProbability()` wählen anhand eines Zufallswerts ein neues Ziel aus.

- **NamesAndProbabilities**

Eine einfache Container-Klasse für Kreuzungsverbindungen. Sie enthält zwei Arrays:

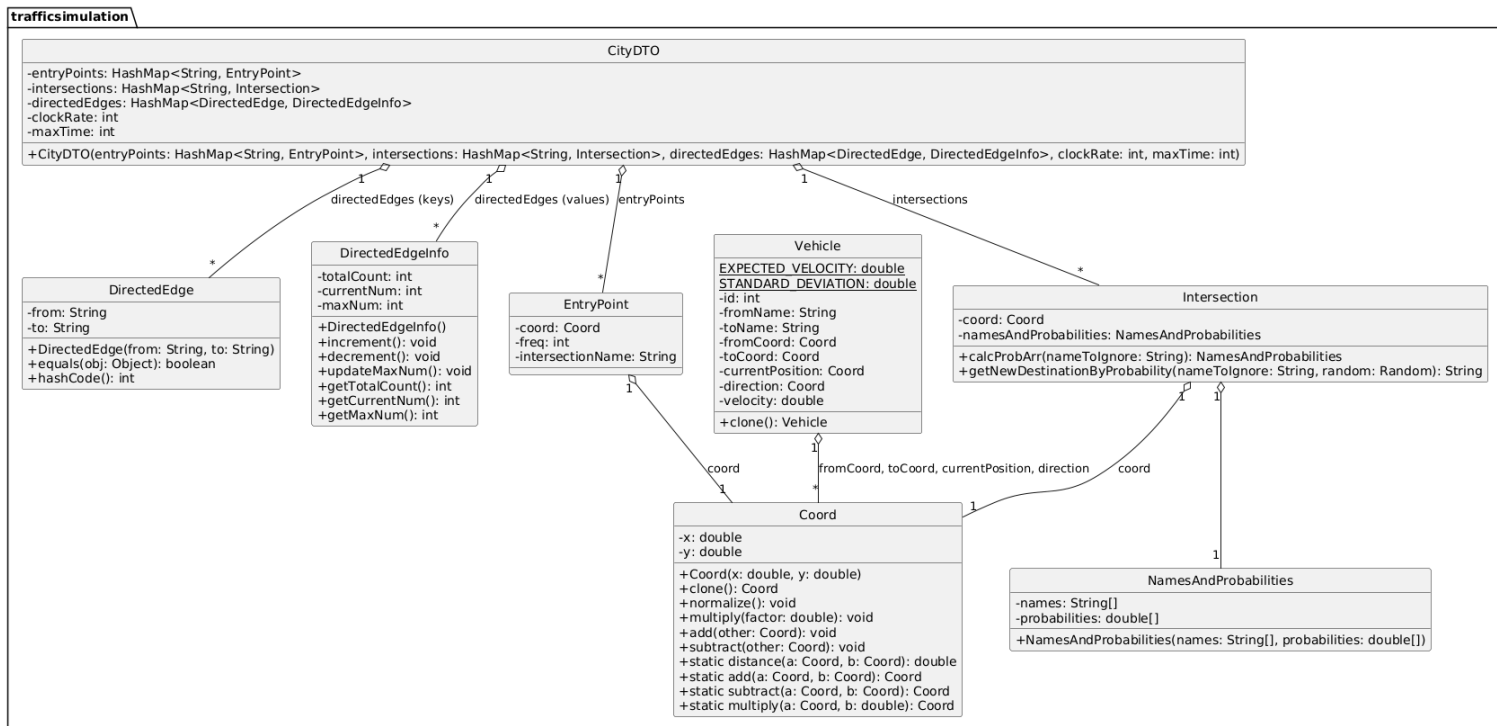
- **names** – Zielknoten,
- **probabilities** – Normalisierte Übergangswahrscheinlichkeiten.

Diese Struktur erleichtert die Auswahl des nächsten Ziels für Fahrzeuge an Kreuzungen. Denn die Wahrscheinlichkeiten müssen an jeder Kreuzung normalisiert werden, da Fahrzeuge nicht in die Richtung zurückfahren können, aus der sie gekommen sind.

• **Vehicle**

Diese Klasse beschreibt den Zustand eines einzelnen Fahrzeugs. Sie enthält Informationen zur aktuellen Position, Bewegungsrichtung, Geschwindigkeit sowie zur Herkunft und dem Ziel. Die `clone()`-Methode erlaubt die Erstellung tiefer Kopien zur Historisierung der Fahrzeugbewegungen. Geschwindigkeit und Richtung werden kontinuierlich in der `simulate()` Methode der `City` Klasse berechnet.

Das folgende Diagramm zeigt die Struktur der Datenhaltungsklassen und deren Beziehungen zueinander.



3.3 Simulation

Die Klasse `City` übernimmt die zentrale Steuerung der Simulation und verwaltet alle strukturellen Elemente des Straßennetzes. Dazu zählen Einfallspunkte (`EntryPoint`), Kreuzungen (`Intersection`) sowie gerichtete Straßenverbindungen (`DirectedEdge`). Neben diesen Strukturinformationen enthält die Klasse eine Liste aktuell aktiver Fahrzeuge sowie eine Historie aller Fahrzeugzustände, die zu Analyse Zwecken gespeichert wird.

Der Simulationsablauf wird durch die Methode `simulate()` umgesetzt, die für jeden Zeitschritt folgende Operationen durchführt:

Bewegung der Fahrzeuge: Die Methode `updateVehicles()` iteriert über alle aktiven Fahrzeuge und aktualisiert deren Position. Wenn ein Fahrzeug sein Ziel erreicht hat, wird es entweder entfernt (bei Ankunft an einem Einfallspunkt) oder ihm wird anhand der Wahrscheinlichkeiten an der aktuellen Kreuzung ein neues Ziel zugewiesen. An jeder Kreuzung werden die relativen Wahrscheinlichkeiten neu berechnet, da Fahrzeuge nicht in die Richtung zurückfahren dürfen, aus der sie gekommen sind. Die Neuberechnung der Wahrscheinlichkeiten erfolgt durch Normalisierung. Dabei wird die Wahrscheinlichkeit für ein Wenden ignoriert, und alle verbleibenden Wahrscheinlichkeiten werden durch die Summe der übrigen Wahrscheinlichkeiten (alle außer die Wendewahrscheinlichkeit) geteilt.

Zur Auswahl der Straße, auf die mit proportionaler Wahrscheinlichkeit abgebogen werden soll, wird der „Roulette-Wheel-Selection“-Algorithmus verwendet. Dabei wird zunächst eine gleichverteilte Zufallszahl zwischen 0 und 1 generiert. Anschließend wird überprüft, ob der Wert der ersten Wahrscheinlichkeit größer oder gleich der erzeugten Zufallszahl ist. Falls dies nicht der Fall ist, wird die nächste Wahrscheinlichkeit aufaddiert und der Vergleich wiederholt, bis die kumulierte Wahrscheinlichkeit größer oder gleich der Zufallszahl ist. Der Index der zuletzt hinzugefügten Wahrscheinlichkeit entspricht der mit proportionaler Wahrscheinlichkeit ausgewählten Straße.

Die Bewegung wird unter Beibehaltung der Restgeschwindigkeit auf die neue Verbindung fortgesetzt.

Fahrzeugerzeugung: Zu festgelegten Zeitintervallen (abhängig vom Taktwert des jeweiligen Einfallspunkts) wird ein neues Fahrzeug erzeugt. Die Methode `createNewVehicle()` initialisiert dessen Position, Ziel, Bewegungsrichtung und Geschwindigkeit. Die Geschwindigkeit wird normalverteilt mit einem Erwartungswert von 45 km/h und einer Standardabweichung von 10 km/h erzeugt. Hierzu wird eine standardnormalverteilte Zufallsvariable ($N(0, 1)$) generiert, mit der Standardabweichung multipliziert und anschließend zum Erwartungswert addiert. Die Geschwindigkeit wird zuvor in die Einheit von 100 Metern umgerechnet. Jede Zufallszahl wird mithilfe eines Pseudozufallszahlengenerators (`java.util.Random`) erzeugt. Außerdem wird die zugehörige Kante im Verkehrsnetz bei Fahrzeugeintritt aktualisiert.

Statistikerhebung: Nach jedem Zeitschritt wird für jede Kante im Netz die aktuelle Anzahl an Fahrzeugen geprüft und ggf. ein neuer Maximalwert gespeichert (`updateDirectedEdgesMaxima()`).

Aufzeichnung des Systemzustands: In Abständen, die durch den `clockRate` bestimmt werden, wird der Zustand aller Fahrzeuge als tiefe Kopie in einer Historie (`vehicleHis-`

tory) gespeichert. Diese erlaubt eine spätere zeitbasierte Rekonstruktion der Fahrzeugbewegungen.

Das folgende Klassen-Diagramm zeigt die Struktur der City-Klasse und die nachfolgenden Nassi-Shneiderman-Diagramme verdeutlichen die oben beschriebene Funktionsweise.

City
<div><div>-entryPoints: HashMap<String, EntryPoint> -intersections: HashMap<String, Intersection> -directedEdges: HashMap<DirectedEdge, DirectedEdgeInfo> -clockRate: int -maxTime: int -vehicles: ArrayList<Vehicle> -vehicleHistory: ArrayList<ArrayList<Vehicle>> -random: Random</div><div>+City(cityDTO: CityDTO) +getDirectedEdges(): HashMap<DirectedEdge, DirectedEdgeInfo> +getEntryPoints(): HashMap<String, EntryPoint> +getIntersections(): HashMap<String, Intersection> +getVehicleHistory(): ArrayList<ArrayList<Vehicle>> +getClockRate(): int +getAlphabeticallySortedDirectedEdges(): List<Entry<DirectedEdge, DirectedEdgeInfo>> +simulate(): void</div></div>

simulate()

for $i = 1$ to maxTime

updateVehicles()

foreach **entry** in entryPoints.entrySet()

T

$i \% \text{entry.getValue().freq} == 0$

F

vehicles.add(createNewVehicle(helper.getId(), **entry**.getKey(), **entry**.getValue().intersectionName))

Ø

updateDirectedEdgesMaxima()

T

$i \% \text{clockRate} == 0$

F

updateVehicleHistory()

Ø

updateVehicles()

```
Iterator < Vehicle > iterator = vehicles.iterator()
```

```
iterator.hasNext()
```

```
    updateVehicle(iterator)
```

```
createNewVehicle(id: int, from: String, to: String): Vehicle
```

```
Vehicle vehicle = new Vehicle()
```

```
vehicle.id = id
```

```
vehicle.fromName = from
```

```
vehicle.toName = to
```

```
vehicle.currentPosition = entryPoints.get(from).coord
```

```
vehicle.toCoord = intersections.get(to).coord
```

```
vehicle.fromCoord = entryPoints.get(from).coord
```

```
vehicle.direction = subtract(vehicle.toCoord, vehicle.fromCoord)
```

```
vehicle.direction.normalize()
```

```
double randomValue = random.nextGaussian() * Vehicle.STANDARD_DEVIATION + Vehicle.EXPECTED_VELOCITY
```

```
vehicle.direction.multiply(randomValue)
```

```
vehicle.velocity = randomValue
```

```
DirectedEdge fromTo = new DirectedEdge(from, to)
```

```
directedEdges.get(fromTo).increment()
```

```
return vehicle
```

updateDirectedEdgesMaxima()

```
foreach entry in directedEdges.entrySet()
```

```
    DirectedEdgeInfo info = entry.getValue()
```

```
    info.updateMaxNum()
```

updateVehicleHistory()

```
ArrayList < Vehicle > deepCopiedVehicles = new ArrayList()
```

```
foreach vehicle in vehicles
```

```
    deepCopiedVehicles.add(vehicle.clone())
```

```
vehicleHistory.add(deepCopiedVehicles)
```

Darüber hinaus stellt die Klasse Funktionen zur Verfügung, um alle gerichteten Kanten alphabetisch sortiert auszulesen (`getAlphabeticallySortedDirectedEdges()`), was für die strukturierte Ausgabe der Ergebnisse genutzt wird.

3.4 Ausgabe

Nach Abschluss der Simulation werden die Ergebnisse in drei getrennte Ausgabedateien geschrieben. Die hierfür verantwortlichen Klassen befinden sich im Output Verzeichnis des `trafficsimulation` packages.

- **Plan.txt (Klasse PlanWriter)**

Diese Datei enthält alle gerichteten Straßenverbindungen als Koordinatenpaare:

$$x1 \ y1 \ x2 \ y2$$

wobei `x1 y1` die Start- und `x2 y2` die Zielposition einer Kante angeben. Die Ausgabe erfolgt alphabetisch sortiert nach Start- und Zielnamen.

```
write(filePath: String, alphabeticallySortedDirectedEdges: List< Map.Entry< DirectedEdge, DirectedEdgeInfo > >, entryPoints: Map< String, EntryPoint >, intersections: Map< String, Intersection >)
```

```
var writer: BufferedWriter = null
```

```
try
```

```
    writer =
```

```
    foreach edge in alphabeticallySortedDirectedEdges
```

```
        Coord from = getCoord(edge.getKey().from, entryPoints, intersections)
```

```
        Coord to = getCoord(edge.getKey().to, entryPoints, intersections)
```

```
        writer.write(from.x + " " + from.y + " " + to.x + " " + to.y)
```

```
        writer.newLine()
```

```
catch exception
```

```
> 
```

- **Statistik.txt (Klasse StatisticWriter)**

Für jede Kante werden zwei Werte normiert auf 100 Meter Straßenlänge berechnet und ausgegeben:

- **Gesamtanzahl Fahrzeuge:** Wie viele Fahrzeuge die Kante insgesamt passiert haben.
- **Maximale Anzahl Fahrzeuge:** Die maximale Anzahl gleichzeitig auf der Kante befindlicher Fahrzeuge während der Simulation.

Beide Werte werden berechnet, indem die Rohwerte durch die euklidische Länge der jeweiligen Kante geteilt werden, wobei die Berechnung auf 100 Meter normiert ist. Die Ausgabe erfolgt ebenfalls alphabetisch sortiert nach Start- und Zielnamen.

```

write(filePath: String, alphabeticallySortedDirectedEdges: List < Map.Entry < DirectedEdge, DirectedEdgeInfo > >, entryPoints: Map < String, EntryPoint >, intersections: Map < String, Intersection >)

var writer: BufferedWriter = null

try
    writer =
    writer.write("Gesamtanzahl Fahrzeuge pro 100 m:\n")

    foreach entry in alphabeticallySortedDirectedEdges
        DirectedEdge edge = entry.getKey()
        DirectedEdgeInfo info = entry.getValue()
        double length = getDistance(edge, entryPoints, intersections)
        double normalized = info.getTotalCount() / length
        writer.write(edge.from + " -> " + edge.to + ": " + normalized + "\n")

    writer.newLine()
    writer.write("Maximale Anzahl Fahrzeuge pro 100 m:\n")

    foreach entry in alphabeticallySortedDirectedEdges
        DirectedEdge edge = entry.getKey()
        DirectedEdgeInfo info = entry.getValue()
        double length = getDistance(edge, entryPoints, intersections)
        double normalized = info.getMaxNum() / length
        writer.write(edge.from + " -> " + edge.to + ": " + normalized + "\n")

catch exception
    >
    
```


- **Fahrzeuge.txt (Klasse VehicleWriter)**

Diese Datei protokolliert die Positionen aller Fahrzeuge zu bestimmten Zeitpunkten der Simulation (gesteuert durch den Parameter `clockRate`). Jeder Zeitstempel ist mit `*** t = <Sekunden>` markiert. Darunter folgen Zeilen mit:

`<x_pos> <y_pos> <x_ziel> <y_ziel> <ID>`

Dadurch kann der zeitliche Verlauf der Fahrzeugbewegungen nachvollzogen werden. Die Daten stammen aus der `vehicleHistory`.

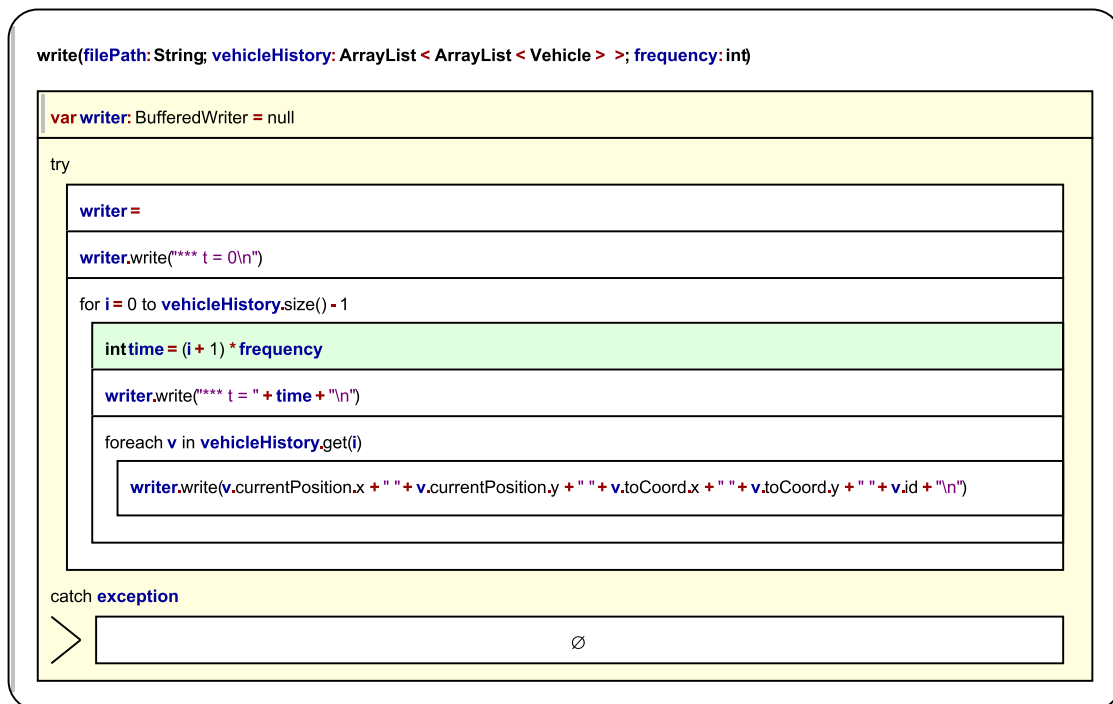
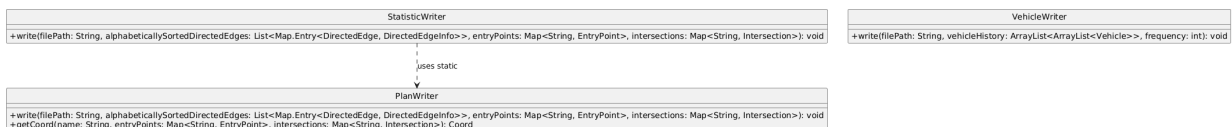


Abbildung 3.2: Klassenstruktur der Ausgabeklassen



3.5 Gesamtablauf

1. Programmstart (`Main.main`)

- Prüfung, ob ein gültiger Dateipfad als Argument übergeben wurde.
- Instanziierung eines Reader-Objekts vom Typ `TextFileReader`.

2. Einlesen der Stadtstruktur (`reader.read()`)

- Datei wird zeilenweise eingelesen und in Abschnitte gegliedert:
 - **Zeitraum:** Einlesen von `maxTime` und `clockRate`.
 - **Einfallspunkte:** Aufbau einer Map von Namen auf `EntryPoint`-Objekte.
 - **Kreuzungen:** Aufbau einer Map von Namen auf `Intersection`-Objekte sowie gerichteter Kanten mit zugehörigem `DirectedEdgeInfo`.
- Durchführung mehrerer Validierungen (z. B. doppelte Namen, unerlaubte Referenzen, Mindestabstand von Koordinaten).
- Rückgabe eines `CityDT0`-Objekts.

3. Initialisierung der Simulation (`new City(CityDT0)`)

- Übernahme aller Eingabedaten in das `City`-Objekt.

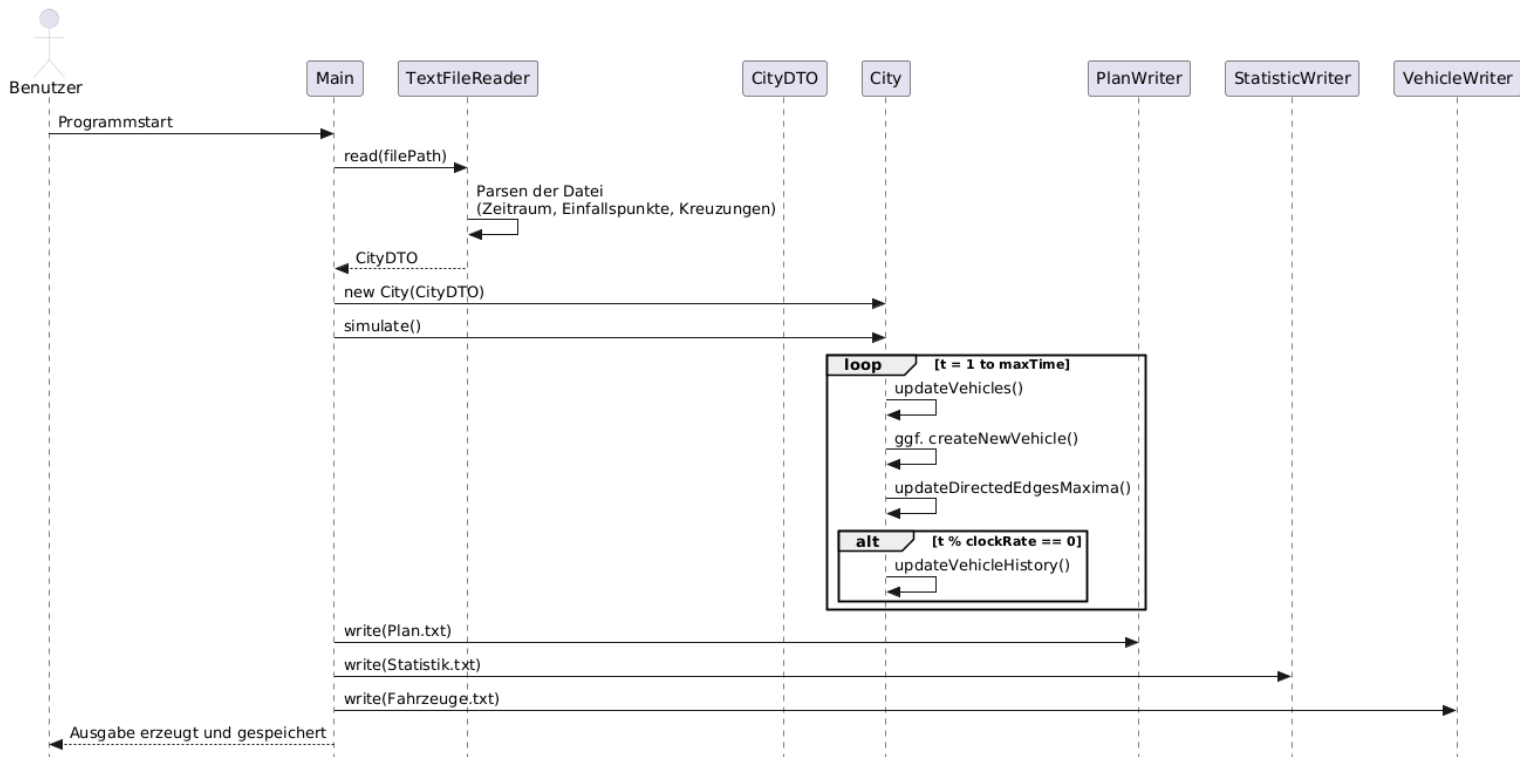
4. Simulationsablauf (`city.simulate()`)

- Für jede Zeiteinheit t von 1 bis `maxTime`:
 - `updateVehicles()`: Bewegung aller Fahrzeuge entlang ihrer aktuellen Kante. Bei Zielerreichung Auswahl eines neuen Ziels oder Entfernen des Fahrzeugs.
 - Fahrzeugerzeugung an jedem Einfallspunkt, sofern $t \bmod \text{freq} = 0$.
 - `updateDirectedEdgesMaxima()`: Aktualisierung der Maximalwerte pro Kante.
 - Speicherung eines Snapshots der Fahrzeugzustände in `vehicleHistory`, wenn $t \bmod \text{clockRate} = 0$.

5. Erzeugung der Ausgabedateien

- `PlanWriter.write`: Ausgabe aller Straßenverbindungen (Koordinatenpaare) in `Plan.txt`.
- `StatisticWriter.write`: Ausgabe von Nutzungsstatistiken (gesamt / maximal pro 100 m) in `Statistik.txt`.
- `VehicleWriter.write`: Ausgabe aller Fahrzeugpositionen zu gespeicherten Zeitpunkten in `Fahrzeuge.txt`.

Abbildung 3.3: Hauptablauf der Simulation



Kapitel 4

Abweichungen vom ersten Entwurf

Bei der Implementierung wurden einige Abweichungen vom ersten Entwurf vom Montag, den 12.05.2025, vorgenommen. So wurde eine `CityDTO`-Datentransferklasse geschrieben, die verwendet wird, um ein `City`-Objekt zu instanziiieren. Dies ist angenehmer als die Nutzung eines Konstruktors mit fünf Eingabeparametern und erfordert keine Implementierung von fünf `get`-Methoden in der `TextFileReader`-Klasse.

Die Klasse `TrafficNode`, die eine Kreuzung darstellen sollte, erhielt den passenderen Namen `Intersection`. Es wurde eine Helper-Klasse mit einer statischen `getId()`-Methode implementiert, um die Fahrzeug-IDs zu generieren.

Im UML-Klassendiagramm der `Vehicle`-Klasse wurde das Attribut `double velocity` (für Geschwindigkeit) vergessen (existiert in der finalen Implementierung). Anfangs wurde nicht spezifiziert, dass die Ausgabe in den Dateien `Plan.txt` und `Statistik.txt` nach alphabetisch sortierten Eingabeortsnamen erfolgen soll. Deswegen wurde die Methode `getAlphabeticallySortedDirectedEdges()` geschrieben.

Die Methode `updateVehicle()` erhält einen `Iterator<Vehicle>` anstelle eines einzelnen `Vehicle`-Objekts, um ein sicheres Entfernen zu gewährleisten, z. B. wenn ein Fahrzeug den Stadtplan über einen Einfallspunkt verlässt, während die Fahrzeugliste iteriert wird.

Die Methode `considerVehiclesThatAreStillDriving()`, die ursprünglich dazu gedacht war, Fahrzeuge, die nach Ende der Simulationszeit noch unterwegs sind, in die Statistik aufzunehmen, erwies sich als unnötig, da die Statistikberechnung Fahrzeuge bereits als den Abschnitt gefahren zählt, sobald sie auf den jeweiligen Abschnitt auffahren.

Eine Reihe von `get`-Methoden wurde hinzugefügt, um den Outputwritern (z. B. `PlanWriter`) die benötigten Informationen wie die `directedEdges` bereitzustellen.

Einige Anforderungen vom Montag wurden revidiert. So wurde ursprünglich festgelegt, dass jede Kreuzung mindestens drei Abbiegepunkte haben soll. Da dies die Eingabe des „IHK-Beispiels Tunneldorf mit Umfahrung über I“ ungültig machen würde, wurde davon abgewichen, sodass jede Kreuzung nun nur mindestens zwei verbindende Straßen haben muss. In der Anfangsspezifikation wurde fälschlicherweise angenommen, dass die relativen Wahrscheinlichkeiten einer Kreuzung in Prozent angegeben sind und sich zu

100 aufaddieren. Dies wurde geändert, sodass alle relativen Wahrscheinlichkeiten intern normiert werden.

Kapitel 5

Tests

Das Verifizieren, ob die Ausgabedateien korrekt sind, lässt sich aufgrund der zufälligen Natur der Simulation und der fehlenden Vergleichsmöglichkeiten mit erwartetem Output nur visuell über das PLOT.py Skript.

Die Normalfälle von der IHK Website getestet und visuell als passend empfunden. Ein Fokus wurde auf das Testen von Fehler und Grenzfällen bei der Einlese gelegt.

5.1 Fehlerfälle

Die Fehlerfälle befinden sich im Ordner `badInputTests`.

Testfall: Koordinaten zu nah beieinander

Datei: `ERROR_COORDINATES_TOO_CLOSE.txt`

```
1 # Test case: Coordinates too close
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 C 0.05 0.05 B 5
8
9 Kreuzungen:
10 B 0 1 A 20 C 30 E 50
```

Beschreibung: In diesem Testfall befinden sich die Punkte A und C zu nah beieinander. Die Minstdistanz zwischen zwei Punkten muss mindestens 0.1 Einheiten betragen, um gültig zu sein.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Koordinaten sind zu nah beieinander...“ ausgegeben.

Testfall: Doppelte Einfallspunktnamen

Datei: ERROR_DUPLICATE_ENTRY_POINT_NAME.txt

```
1 # Test case: Duplicate entry point name
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 A 1 1 C 3
8
9 Kreuzungen:
10 B 0 1 A 20 C 30 E 50
```

Beschreibung: In diesem Testfall wird der Name A für zwei Einfallspunkte verwendet. Einfallspunktnamen müssen eindeutig sein.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Doppelter Einfallspunktname: ...“ ausgegeben.

Testfall: Doppelte Kreuzungsnamen

Datei: ERROR_DUPLICATE_INTERSECTION_NAME.txt

```
1 # Test case: Duplicate intersection name
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7
8 Kreuzungen:
9 B 0 1 A 20 C 30 E 50
10 B 1 1 D 20 F 30 G 50
```

Beschreibung: In diesem Testfall wird der Name B für zwei Kreuzungen verwendet. Kreuzungsnamen müssen eindeutig sein.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Doppelter Kreuzungsname: ...“ ausgegeben.

Testfall: Einfallspunkt und Kreuzung doppelt verwendet

Datei: ERROR_DUPLICATE_LOCATIONS.txt

```
1 # Not unique entrypoints and intersections
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 B 0 2 D 3
8 C 0 4 D 5
9 D 2 2 E 2
10 E 4 2 G 5
11
12 Kreuzungen:
13 B 0 17 A 20 C 20 E 60
14 E 4 42 D 40 F 10 G 50
```

Beschreibung: In diesem Testfall tauchen die Orte B und E sowohl als Einfallspunkte als auch als Kreuzungen auf, was nicht erlaubt ist.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Orte können nicht gleichzeitig Einfallspunkte und Kreuzungen sein.“ ausgegeben.

Testfall: Einfallspunktname zu lang

Datei: ERROR_ENTRY_POINT_NAME_TOO_LONG.txt

```
1 # Test case: Entry point name too long
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 ThisIsAnExcessivelyLongEntryPointNameThatExceeds... 0 0 B 2
7
8 Kreuzungen:
9 B 0 1 A 20 C 30 E 50
```

Beschreibung: Der Name des Einfallspunkts überschreitet die erlaubte Maximallänge von 100 Zeichen (wurde hier abgeschnitten).

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Einfallspunktname ist zu lang: ...“ ausgegeben.

Testfall: Kreuzungsname zu lang

Datei: ERROR_INTERSECTION_NAME_TOO_LONG.txt

```
1 # Test case: Intersection name too long
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7
8 Kreuzungen:
9 ThisIsAnExcessivelyLongIntersectionName... 0 1 A 20 C 30 E 50
```

Beschreibung: Der Name der Kreuzung überschreitet die erlaubte Maximallänge von 100 Zeichen (wurde hier abgeschnitten).

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Kreuzungsname ist zu lang: ...“ ausgegeben.

Testfall: Ungültige Koordinatenkomponente

Datei: ERROR_INVALID_COORDINATE_COMPONENT.txt

```
1 # Test case: Invalid coordinate component
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 2000 0 B 2
7
8 Kreuzungen:
9 B 0 1 A 20 C 30 E 50
```

Beschreibung: Der X-Wert der Koordinate von Einfallspunkt A liegt außerhalb des erlaubten Bereichs von -1000 bis 1000 .

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Ungültiger Wert für Koordinatenkomponente: ...“ ausgegeben.

Testfall: Ungültige Einfallspunktreferenz

Datei: ERROR_INVALID_ENTRY_POINT_REFERENCE.txt

```
1 # Test case: Entry points reference non-existent intersections
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 Z 2
7 C 0 2 Y 5
8
9 Kreuzungen:
10 B 0 1 A 20 C 30 E 50
11 E 4 1 D 20 F 20 G 10 B 50
```

Beschreibung: Die Einfallspunkte A und C verweisen auf nicht existierende Kreuzungen Z bzw. Y.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Nicht alle Referenzen von Einfallspunkten existieren tatsächlich.“ ausgegeben.

Testfall: Ungültige allgemeine Taktrate

Datei: ERROR_INVALID_GENERAL_FREQUENCY.txt

```
1 # Test case: Invalid general frequency
2 Zeitraum:
3 50 60
4
5 Einfallspunkte:
6 A 0 0 B 2
7
8 Kreuzungen:
9 B 0 1 A 20 C 30 E 50
```

Beschreibung: Die allgemeine Taktrate (60) ist größer als die Simulationszeitspanne (50 Sekunden), was nicht erlaubt ist.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Ungültiger Wert für allgemeine Taktartrate: ...“ ausgegeben.

Testfall: Ungültige Kreuzungsreferenz

Datei: ERROR_INVALID_INTERSECTION_REFERENCE.txt

```
1 # Test case: Intersections reference non-existent locations
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 C 0 2 B 5
8
9 Kreuzungen:
10 B 0 1 A 20 C 30 X 50
11 E 4 1 D 20 F 20 G 10 Y 50
```

Beschreibung: Die Kreuzungen verweisen auf Orte (X und Y), die nicht als Einfallspunkte oder Kreuzungen existieren.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Nicht alle Referenzen von Kreuzungen existieren tatsächlich.“ ausgegeben.

Testfall: Ungültiger Maximalzeitwert

Datei: ERROR_INVALID_MAX_TIME.txt

```
1 # Test case: Decimal value for maxTime
2 Zeitraum:
3 50.0 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7
8 Kreuzungen:
9 B 0 1 A 20 C 30 E 50
```

Beschreibung: Der Wert für die maximale Simulationszeitspanne ist eine Dezimalzahl, es wird jedoch eine ganze Zahl erwartet.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Ungültiger Wert für die Simulationszeitspanne: ... Dezimalwerte sind nicht erlaubt.“ ausgegeben.

Testfall: Ungültige Wahrscheinlichkeit

Datei: ERROR_INVALID_PROBABILITY.txt

```
1 # Test case: Invalid probability in percentage
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 C 4 5 B 2
8 E 2 3 B 2
9
10 Kreuzungen:
11 B 0 1 A -20 C 30 E 150
```

Beschreibung: Die relative Wahrscheinlichkeit fürs Abbiegen nach A an Kreuzung B ist negativ. Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Ungültiger Wert für Wahrscheinlichkeit: ...“ ausgegeben.

Testfall: Ungültige Orts-Wahrscheinlichkeits-Paare

Datei: ERROR_LOCATION_PROBABILITY_PAIRS_EXPECTED.txt

```
1 # Test file for ERROR_LOCATION_PROBABILITY_PAIRS_EXPECTED
2 Zeitraum:
3 3600 1
4
5 Einfallspunkte:
6 EntryPoint1 0 0 Intersection1 1
7 EntryPoint2 3 3 Intersection1 1
8 EntryPoint3 5 5 Intersection1 1
9 EntryPoint4 4 4 Intersection1 1
10
11 Kreuzungen:
12 Intersection1 0 0 EntryPoint1 EntryPoint3 EntryPoint4
    EntryPoint2 0.5
```

Beschreibung: In der Kreuzung fehlen Wahrscheinlichkeiten für einige Ziele — es wird erwartet, dass jede Zielangabe von einer Wahrscheinlichkeit begleitet wird.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Ungültiges Format für Kreuzung, alle Ortsangaben werden mit relativen Wahrscheinlichkeiten erwartet: ...“ ausgegeben.

Testfall: Mehrere Zeitabschnitte

Datei: ERROR_MULTIPLE_TIMESPAN_SECTIONS.txt

```
1 # Test case: Duplicate sections
2 Zeitraum:
3 50 1
4
5 Zeitraum:
6 60 2
7
8 Einfallspunkte:
9 A 0 0 B 2
10
11 Kreuzungen:
12 B 0 1 A 20 C 30 E 50
```

Beschreibung: Die Datei enthält zwei „Zeitraum:“-Abschnitte, was laut Spezifikation nicht erlaubt ist.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Mehrere 'Zeitraum:' Abschnitte sind nicht erlaubt.“ ausgegeben.

Testfall: Keine Einfallspunkte

Datei: ERROR_NO_ENTRY_POINTS.txt

```
1 # Test case: Empty entry points
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6
7 Kreuzungen:
8 B 0 1 A 20 C 30 E 50
```

Beschreibung: Es sind keine Einfallspunkte in der Datei definiert.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Keine Einfallspunkte gefunden.“ ausgegeben.

Testfall: Keine Kreuzungen

Datei: ERROR_NO_INTERSECTIONS.txt

```
1 # Test case: Empty intersections
2 Zeitraum:
3 50 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7
8 Kreuzungen:
```

Beschreibung: Es sind keine Kreuzungen in der Datei definiert.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Keine Kreuzungen gefunden.“ ausgegeben.

Testfall: Zu wenig Verbindungen an Kreuzung

Datei: ERROR_TOO_FEW_CONNECTED_STREETS.txt

```
1 # Test file for ERROR_TOO_FEW_CONNECTED_STREETS
2 Zeitraum:
3 3600 1
4
5 Einfallspunkte:
6 EntryPoint1 0 0 Intersection1 1
7 EntryPoint2 1 1 Intersection1 1
8 EntryPoint3 2 2 Intersection1 1
9
10 Kreuzungen:
11 Intersection1 0 0 EntryPoint1 2
```

Beschreibung: Die Kreuzung Intersection1 hat keine verbundenen Straßen, was unter der minimal erforderlichen Anzahl liegt.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Diese Kreuzung hat eventuell zu wenig verbindende Straßen: ...“ ausgegeben.

Testfall: Zu viele Verbindungen an Kreuzung

Datei: ERROR_TOO_MANY_CONNECTED_STREETS.txt

```
1 # Test file for ERROR_TOO_MANY_CONNECTED_STREETS
2 Zeitraum:
3 3600 1
4
5 Einfallspunkte:
6 EntryPoint1 0 0 Intersection1 1
7 EntryPoint2 1 1 Intersection1 1
8 EntryPoint3 2 2 Intersection1 1
9 EntryPoint4 3 3 Intersection1 1
10 EntryPoint5 4 4 Intersection1 1
11 EntryPoint6 5 5 Intersection1 1
12 EntryPoint7 6 6 Intersection1 1
13 EntryPoint8 7 7 Intersection1 1
14 EntryPoint9 8 8 Intersection1 1
15 EntryPoint10 9 9 Intersection1 1
16 EntryPoint11 10 10 Intersection1 1
17 EntryPoint12 11 11 Intersection1 1
18 EntryPoint13 12 12 Intersection1 1
19 EntryPoint14 13 13 Intersection1 1
20 EntryPoint15 14 14 Intersection1 1
21 EntryPoint16 15 15 Intersection1 1
22 EntryPoint17 16 16 Intersection1 1
23 EntryPoint18 17 17 Intersection1 1
24 EntryPoint19 18 18 Intersection1 1
25 EntryPoint20 19 19 Intersection1 1
26 EntryPoint21 20 20 Intersection1 1
27 EntryPoint22 21 21 Intersection1 1
28
29 Kreuzungen:
30 Intersection1 1 0 EntryPoint1 0.5 EntryPoint2 0.5 EntryPoint3
    0.5 EntryPoint4 0.5 EntryPoint5 0.5 EntryPoint6 0.5
    EntryPoint7 0.5 EntryPoint8 0.5 EntryPoint9 0.5 EntryPoint10
    0.5 EntryPoint11 0.5 EntryPoint12 0.5 EntryPoint13 0.5
    EntryPoint14 0.5 EntryPoint15 0.5 EntryPoint16 0.5
    EntryPoint17 0.5 EntryPoint18 0.5 EntryPoint19 0.5
    EntryPoint20 0.5 EntryPoint21 0.5
```

Beschreibung: Die Kreuzung Intersection1 hat mehr als die erlaubten 20 verbundenen Straßen.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Diese Kreuzung hat eventuell mehr als 20 verbindende Straßen: ...“ ausgegeben.

Testfall: Nicht UTF-8 kodierte Datei

Datei: ERROR_NOT_UTF8.txt



Beschreibung: Diese Datei ist nicht im UTF-8-Format kodiert.

Erwartete Fehlermeldung: „Eingabedatei ist nicht UTF-8 kodiert: “;

Testfall: Doppelter Abbiegepunktname

Datei: ERROR_DUPLICATE_TURNOFF_NAME.txt

```
1 #In the intersections there is A twice in one line
2 Zeitraum:
3 100 1
4
5 Einfallspunkte:
6 A 0 0 B 2
7 C 0 2 B 5
8 D 4 0 E 3
9 F 4 2 E 2
10 G 5 1 E 3
11
12 Kreuzungen:
13 B 0 1 A 20 C 30 E 50 A 40
14 E 4 1 D 20 F 20 G 10 B 50
```

Beschreibung: Die Kreuzung B hat den Abbiegepunkt A zweimal.

Erwartete Fehlermeldung: Es wird eine Fehlermeldung mit dem Text „Doppelter Abbiegepunktname: ...“ ausgegeben.

5.2 Grenzfälle

Die Grenzfalleingabedateien sind im Ordner `edgeCases`.

Testfall: Mindestabstand der Koordinaten wird eingehalten

Datei: `coordinatesHaveMinimumDistance.txt`

```
1 Zeitraum:
2 50 1
3
4 Einfallspunkte:
5 A 0 0 B 2
6 C 0.1 0.0 B 5
7 E 4.3 0.0 B 5
8
9 Kreuzungen:
10 B 0 1 A 20 C 30 E 50
```

Beschreibung: Der Mindestabstand von 0,1 Einheiten wird zwischen A und C eingehalten.

Testfall: Beliebige Reihenfolge der Abschnitte

Datei: `differentSectionOrder.txt`

```
1 Zeitraum:
2 50 3
3
4 Kreuzungen:
5 B 0 1 A 20 C 30
6
7 Einfallspunkte:
8 A 0 0 B 2
9 C 1 0.0 B 5
```

Beschreibung: Die Reihenfolge der Abschnitte entspricht nicht der Reihenfolge aus den Beispielen von Zeitraum, Einfallspunkte und Kreuzungen.

Testfall: Maximale erlaubte Simulationszeit

Datei: `maximalTime.txt`

```
1 Zeitraum:
2 86400 1
3 Einfallspunkte:
4 A 2 0 C 2
5 B 2 4 C 5
6 F 5 4 C 5
7
8 Kreuzungen:
9 C 0 1 A 20 B 30 F 50
```

Beschreibung: Die maximale Simulationsdauer von 24 Stunden (86400 Sekunden) wird akzeptiert.

Testfall: Maximale erlaubte Koordinatenwerte

Datei: maximumCoords.txt

```
1 Zeitraum:
2 86 1
3 Einfallspunkte:
4 A 2 0 C 2
5 B 1000 4 C 5
6 F 5 4 C 5
7
8 Kreuzungen:
9 C 0 1 A 0.1 B 30 F 50
```

Beschreibung: Der Wert 1000 für eine Koordinatenkomponente wird korrekt akzeptiert.

Testfall: Maximale Namenslänge wird akzeptiert

Datei: maximumInputName.txt

```
1 Zeitraum:
2 86 1
3 Einfallspunkte:
4 A 2 0 C 2
5 B 1000 4 C 5
6 iiii... 5 4 C 5
7
8 Kreuzungen:
9 C 0 1 A 0.1 B 30 iiii... 50
```

Beschreibung: Ein Einfallspunkt- und Kreuzungsname mit genau 100 Zeichen (hier abgeschnitten dargestellt) wird akzeptiert.

Testfall: Maximale Anzahl verbundener Straßen

Datei: maximumNumEdges.txt

```
1 Zeitraum:
2 360 1
3
4 Einfallspunkte:
5 EntryPoint1 0 0 Intersection1 1
6 EntryPoint2 1 1 Intersection1 1
7 EntryPoint3 2 2 Intersection1 1
8 EntryPoint4 3 3 Intersection1 1
9 EntryPoint5 4 4 Intersection1 1
10 EntryPoint6 5 5 Intersection1 1
11 EntryPoint7 6 6 Intersection1 1
12 EntryPoint8 7 7 Intersection1 1
13 EntryPoint9 8 8 Intersection1 1
14 EntryPoint10 9 9 Intersection1 1
15 EntryPoint11 10 10 Intersection1 1
16 EntryPoint12 11 11 Intersection1 1
17 EntryPoint13 12 12 Intersection1 1
18 EntryPoint14 13 13 Intersection1 1
19 EntryPoint15 14 14 Intersection1 1
20 EntryPoint16 15 15 Intersection1 1
21 EntryPoint17 16 16 Intersection1 1
22 EntryPoint18 17 17 Intersection1 1
23 EntryPoint19 18 18 Intersection1 1
24 EntryPoint20 19 19 Intersection1 1
25
26 Kreuzungen:
27 Intersection1 1 0 EntryPoint1 0.5 EntryPoint2 0.5 EntryPoint3
    0.5 EntryPoint4 0.5 EntryPoint5 0.5 EntryPoint6 0.5
    EntryPoint7 0.5 EntryPoint8 0.5 EntryPoint9 0.5 EntryPoint10
    0.5 EntryPoint11 0.5 EntryPoint12 0.5 EntryPoint13 0.5
    EntryPoint14 0.5 EntryPoint15 0.5 EntryPoint16 0.5
    EntryPoint17 0.5 EntryPoint18 0.5 EntryPoint19 0.5
    EntryPoint20 0.5
```

Beschreibung: Eine Kreuzung mit 20 verbundenen Straßen wird erfolgreich verarbeitet.

Testfall: Minimal gültiges Beispiel

Datei: minimalExample.txt

```
1 Zeitraum:
2 50 1
3 Einfallspunkte:
4 A 2 0 C 2
5 B 2 4 C 5
6
7 Kreuzungen:
8 C 0 1 A 20 B 30
```

Beschreibung: Das minimale gültige Setup mit zwei Einfallspunkten und einer Kreuzung funktioniert korrekt.

Testfall: Wahrscheinlichkeit nahe Null

Datei: probabilityAlmostZero.txt

```
1 Zeitraum:
2 86 1
3 Einfallspunkte:
4 A 2 0 C 2
5 B 2 4 C 5
6 F 5 4 C 5
7
8 Kreuzungen:
9 C 0 1 A 0.000001 B 30 F 50
```

Beschreibung: Wahrscheinlichkeiten im gültigen unteren Grenzbereich (nahe 0.000001) werden korrekt akzeptiert.

Testfall: Takt gleich Zeitspanne

Datei: timeSpanIsClockRate.txt

```
1 Zeitraum:
2 50 50
3
4 Einfallspunkte:
5 A 0 0 B 2
6 C 1 0.0 B 5
7
8 Kreuzungen:
9 B 0 1 A 20 C 30
```

Beschreibung: Der Takt ist gleich der maximalen Zeitspanne – dies ist gültig und wird korrekt akzeptiert.

Nicht .txt Dateien

Dateien, welche nicht auf „.txt“ enden, werden auch verarbeitet, wenn diese valide in UTF-8 kodierte Simulationsstadtpläne enthalten.

5.3 Normalfälle

Die Normalfälle wurden mit den Beispielen von der IHK-Website getestet. Die Dateien sind im Ordner `normalCases` gespeichert.

5.4 Ausführen der Tests

5.4.1 BadInputTestRunner

Bei der Arbeit an dem `TextFileReader` wurde mehrfach refactored und die Bedingungsprüfungen wurden nacheinander implementiert und nach jeder Bedingung geprüft, ob frühere Fehlerfälle immer noch die erwartete Ausgabe liefern. Um diesen Prozess zu automatisieren, wurde das `BadInputTestRunner`-Testprogramm für Regressionstests geschrieben.

Der Quellcode ist die gleichnamige Java-Datei im `tests`-Ordner, ebenfalls im `trafficsimulation`-Package. Das Programm führt die folgenden Schritte aus:

- **Verzeichnisprüfung:** Es wird geprüft, ob das Fehlerfallverzeichnis existiert und Eingabedateien enthält.
- **Definition der Testfälle:** Für jede fehlerhafte Eingabedatei wird die erwartete Fehlermeldung definiert. Beispiele für Fehler sind:
 - Fehlende Einfallspunkte (`ERROR_NO_ENTRY_POINTS`),
 - Ungültige Koordinaten (`ERROR_INVALID_COORDINATE_COMPONENT`)
- **Testausführung:** Jede Eingabedatei wird mit dem `TextFileReader` eingelesen. Falls die Datei fehlerhaft ist, wird eine Ausnahme ausgelöst.
- **Ergebnisprüfung:** Die tatsächliche Fehlermeldung wird mit der erwarteten Fehlermeldung verglichen. Stimmen sie überein, gilt der Test als bestanden.
- **Endergebnis:** Am Ende wird eine Übersicht ausgegeben, ob alle Tests bestanden wurden oder ob Fehler aufgetreten sind.

Der `BadInputTestRunner` kann wie normale jars mit:

```
1 java -jar BadInputTestRunner.jar <Pfad/zum/Fehlerfallordner>
```

ausgeführt werden. Wird kein Startparameter angegeben, wird nach dem relativen Pfad 'badInputTests' gesucht.

Zum Testen des `trafficsimulation`-Programms steht je ein Skript für Windows und Linux/macOS zur Verfügung. Unter Windows wird das Skript `test_run.bat` verwendet. Die Skripte führen standardmäßig die Datei `trafficsimulation.jar` für jede Datei im Ordner `edgeCases` einzeln aus und übergeben die Datei als Argument an das Programm.

Um andere Verzeichnisse mit Eingabedateien zu testen, können die Skripte angepasst werden in dem die Variable `INPUT_DIR`, welche den Pfad des Testverzeichnisses speichert, geändert wird.

Das Skript kann entweder per Doppelklick oder über die Eingabeaufforderung mit `testScript.bat` gestartet werden.

Für Linux oder macOS wird das Bash-Skript `testScript.sh` genutzt. Auch hier wird standardmäßig die `trafficsimulation.jar` für jede Datei im Ordner `edgeCases` aufgerufen.

Vor dem ersten Ausführen muss dem Skript Ausführungsrecht gegeben werden, z.B. durch den Befehl `chmod +x testScript.sh`. Danach kann es mit `./testScript.sh` ausgeführt werden. Alternativ kann das Skript auch ohne Ausführungsrecht mit `bash testScript.sh` gestartet werden.

Im folgenden die IHK Dokumentation zum `Plot.py` Skript aus der `Plot_howto.txt`:

Das Skript „`Plot.py`“ dient dazu, die Verkehrssimulation zu visualisieren.

Der Aufruf von „`Plot.py`“ erfordert genau einen Parameter: den absoluten Pfad des Verzeichnisses, in dem alle Dateien des Testfalls liegen. Jeder Testfall sollte in einem eigenen Verzeichnis liegen. Darin liegt ein Unterverzeichnis „`plots`“, in dem die PNG-Plots gespeichert werden. Existiert „`plots`“ nicht, wird es angelegt.

Die Visualisierung zeigt alle Zeitschritte mit Überschrift an. Am Ende die Simulation beginnt sie automatisch von vorne. Der erste Durchgang ist langsamer, weil die PNG-Bilder erzeugt werden.

Im Skript gibt es folgende zwei Parameter, mit der die Visualisierung den eigenen Wünschen angepasst werden kann: 1. „`REL_LANE_OFFSET`“ legt fest, wie weit die Fahrbahnen neben der Fahrbahnmitte liegen. Die Angabe erfolgt relativ bezogen auf das Minimum der Breite und Höhe des Bildausschnitts. 2. „`PLOT_INTERVALL`“ legt die Wartezeit zwischen den Zeitschritten beim Plot und damit die Geschwindigkeit der Visualisierung fest.

„`Plot.py`“ wurde in der Python-Version 3.10 mit dem Package „`matplotlib`“ in der Version 3.8.4 und dem package „`numpy`“ in der Version 1.26.4 getestet.

Kapitel 6

Erweiterbarkeit

Eine vorstellbare weitere Anforderung wäre, dass Fahrzeuge nicht überholen können, sondern einen Mindestabstand zum vorausfahrenden Fahrzeug einhalten müssen. Diese zusätzliche Anforderung ließe sich durch Anpassen der `DirectedEdgeInfo` Klasse und der `updateVehicle` Methode erreichen. Die Fahrzeuge würden nicht mehr zentral in der `vehicles` Liste gespeichert werden, sondern in der `DirectedEdgeInfo` Klasse in einer Queue Struktur. Neue Fahrzeuge auf der Straße würden ans Ende der Queue eingefügt werden und Fahrzeuge, die die Straße verlassen, würden, würden am Anfang der Queue entfernt werden. Beim Aktualisieren der Fahrzeuge in der `simulate()` Methode würde über die `DirectedEdgeInfos` in der `directedEdges` HashMap iteriert werden und die Position des Fahrzeugs, welches am nächsten am Verlassen der Straße ist, würde als erstes aktualisiert werden, also das Fahrzeug, welches am Anfang der Straßenqueue ist. Bei dem ersten aktualisierten Fahrzeug würde berechnet werden, wo ein Fahrzeug, welches den Minimalabstand zu dem Fahrzeug am Anfang der Queue einhält, wäre und diese Position wird an das zweite Fahrzeug weitergegeben. Nun wird die Position des zweiten Fahrzeuges so angepasst, dass dieses entweder normal weiterfährt, wenn es nicht den Minimalabstand unterschreiten würde oder, wenn es den Minimalabstand unterschreiten würde die Position auf die übergebene zuvor berechnete Position gesetzt. Am Ende aller Iterationen aller `DirectedEdges`, werden die Positionen der Fahrzeuge neu berechnet, welche Straßen an Kreuzungen gewechselt haben.

Wenn der Verkehrsfluss an Kreuzungen über Ampeln geregelt werden würde, würde die Position von Fahrzeugen, die eine Straße an einer Kreuzung wechseln würde, um die Anzahl an Takten, wie lange die Ampel grün bleibt, gestoppt, bevor diese weiterfahren dürfen.

Beim Hinzufügen von Kreisverkehren wäre eine ähnliche Logik anwendbar. Ein Fahrzeug würde für die Taktlänge eines im Stadtplan durch ein extra Zeichen angegebenen Kreisverkehrs an der Kreuzung verweilen, bevor es weiterfährt.

Für das Implementieren einer Rechts-vor-Links-Überprüfung müsste man für jedes Fahrzeug, das eine Kreuzung passieren würde, zunächst prüfen, welche Straßen rechts liegen. Dafür würde man die xy-Komponenten des Anfangs-End-Vektors der gerade gefahrenen Straße tauschen und danach die x-Komponente negieren, um den links-orthogonalen Vektor zu dem gerade gefahrenen Straßenvektor zu erhalten. Nun findet man alle `directedEdges`, bei denen „to“ der Kreuzung entspricht, an der abgebogen

werden soll. Von diesen sind die Straßen rechts, bei denen das Skalarprodukt des links-orthogonalen Vektors mit dem Anfangs-End-Vektor der potenziellen rechten Straße größer als 0 ist.

Jetzt müsste berechnet werden, ob mehrere Autos in einem vorher festgelegten Zeitraum an den oben beschriebenen Straßen an einer Kreuzung stehen. Für eine korrekte Implementierung müsste vorher geklärt werden, wie groß der Zeitraum ist, den Autos bei Rechts-vor-Links auf andere Autos warten müssten, und wie weit entfernt Fahrzeuge maximal von einem an einer Kreuzung wartenden Fahrzeug sein können, damit die Rechts-vor-Links-Regelung gilt. Für Stadtpläne mit niedrigen Einfallspunkttaktraten könnte es jedoch zu Deadlocks führen, wenn an einer Kreuzung jedes Auto auf das Auto rechts warten muss.

Mehrere Fahrbahnen pro Strecke erfordern eine Lockerung des Einleseformats, so müssen doppelte Einfallspunktzeilen, bei denen sich lediglich die Taktrate unterscheidet, zugelassen werden. Zudem müssen auch in Kreuzungszeilen mehrfach die gleichen Abbiegenamen mit unterschiedlichen Wahrscheinlichkeiten zugelassen werden.

Ein Problem bei der jetzigen Implementierung ist die Verwendung von HashMaps. Diese erlauben keine mehrfach gleichen Schlüssel. Ein Trick könnte die Durchnummerierung der Punkte vor dem Anwenden der Hashfunktion sein, sodass ein Einfallspunkt mit zwei Fahrbahnen z. B. als zwei Punkte mit den gleichen Koordinaten B1 und B2 gehandhabt wird. Hierbei ist natürlich zu überprüfen, dass nicht schon ein Punkt B2 existiert. Eine sauberere Lösung wäre eine Ort-Klasse mit einer ID und einem String als Namen und die Verwendung einer solchen Klasse in HashMaps (natürlich müsste vorher Vergleichen und die Hashfunktion überschrieben werden).

Nicht konstante Geschwindigkeiten, die abhängig von Streckenabschnitten sind, könnten eine Modifikation der dafür zuständigen DirectedEdgeInfos erfordern. In solch einem Fall würde jede DirectedEdgeInfo einen Geschwindigkeitsparameter haben oder eine Funktion, die die Geschwindigkeit von sich auf dem jeweiligen Straßenabschnitt befindenden Fahrzeugen ändern kann. Für eine Implementierung ist der genaue Geschwindigkeitsänderungsmechanismus erforderlich.

Für die oben beschriebenen potenziellen Folgeanforderungen kann zudem die Benutzung von Polymorphismus hilfreich sein. So kann beispielsweise DirectedEdgeInfo oder Vehicle eine Basisklasse sein, von der für andere Simulationsmodi erbende Klassen benutzt werden.

Kapitel 7

Projektstruktur

Das Projekt ist wie folgt strukturiert:

```
.gitignore
testScript.bat
testScript.sh
trafficSimulation.jar
6511-225_Download-Vorlagen/
    Plot_howto.txt
    Plot.py
    IHK_01/
    IHK_02/
    IHK_03/
    ...
badInputTests/
doku/
edgeCases/
META-INF/
normalCases/
badInputTests/
src/
BadInputTestRunner.jar
Dokumentation.pdf
```

Die wichtigsten Verzeichnisse und Dateien sind:

- `doku/`: Enthält die Dokumentation, einschließlich der LaTeX-Dateien wie `docu.tex`, `intro.tex`, und `tests.tex`. Die gebaute PDF befindet sich im `build` Verzeichnis.
- `src/`: Der Quellcode des Projekts. Die Javadeien befinden sich im `trafficSimulation` package.
- `normalCases/`, `edgeCases/` und `badInputTests/`: Testfälle für Normal, Grenz und Fehlerfälle.
- `trafficSimulation.jar`: Die ausführbare JAR-Datei der Simulation.
- `6511-225_Download-Vorlagen/`: Die IHK Vorlagen und das Skript `Plot.py`.

- `testScript.bat` `testScript.sh`: Die Testskripte zum automatischen Testen von Verzeichnissen
- `BadInputTestRunner.jar`: Testjava Hilfstool zum automatischen Testen von Fehlerfällen

Kapitel 8

Benutzeranleitung

Um das Java-Programm auszuführen, gehen Sie wie folgt vor:

1. Stellen Sie sicher, dass Java (mindestens Version 21) auf Ihrem System installiert ist.
2. Öffnen Sie eine Eingabeaufforderung (CMD) oder ein Terminal.
3. Navigieren Sie in das Verzeichnis, in dem sich die JAR-Datei befindet.
4. Führen Sie das Programm mit folgendem Befehl aus und geben Sie dabei den gewünschten Eingabepfad als Parameter an:

```
1 java -jar trafficSimulation.jar <Pfad/zur/Eingabedatei>
```

5. Ersetzen Sie <Pfad/zur/Eingabedatei> durch den tatsächlichen Pfad zu Ihrer Eingabedatei.

Beispiel:

```
java -jar trafficsimulation.jar C:\Users\debel\input.txt
```

Für jede Simulation wird ein eigener Ausgabeordner angelegt, dessen Name aus dem Präfix `output_`, gefolgt vom Namen der Eingabedatei ohne Dateiendung, besteht. Beispielsweise wird die Ausgabe der Datei `Beispielhausen.txt` im Ordner `output_Beispielhausen` gespeichert. Die Ausgabe des Programms besteht aus drei Dateien. Falls im Ordner bereits gleichnamige Dateien vorhanden sind, werden diese überschrieben. Der neuerzeugte Ordner befindet sich auf gleicher Ebene, wie die ausführbare `.jar`.

Kapitel 9

Entwicklungsumgebung

Programmiersprache: Java

Build-Tool: IntelliJ IDEA internes Build-Tool (kein externes Build-Tool wie Maven oder Gradle verwendet)

JDK: Eclipse Temurin 21.0.2

IDE: IntelliJ IDEA Community Edition 2024.1.4

UML-Tool: PlantUml

Nassi-Shneiderman-Diagramm-Tool: Structorizer 3.32-19

Prozessor: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 1.80 GHz

Betriebssystem: Windows 11 Enterprise

Kapitel 10

Zusammenfassung und Ausblick

Im Rahmen der praktischen Prüfung wurde ein Java-Programm zur Simulation von Verkehrsflüssen entwickelt. Ziel war es, Fahrzeugbewegungen in einem vordefinierten Straßennetz zu modellieren und auszuwerten. Die Eingabedateien beschreiben dabei Einfallspunkte, Kreuzungen sowie einen Simulationszeitraum. Die Fahrzeuge bewegen sich mit individuell zufälligen Geschwindigkeiten durch das Netz und treffen an Kreuzungen Entscheidungen gemäß konfigurierter Wahrscheinlichkeiten.

Die Simulation erzeugt drei zentrale Ausgabedateien:

- `Plan.txt` enthält die Straßengeometrie,
- `Statistik.txt` fasst die Streckenauslastung zusammen,
- `Fahrzeuge.txt` ermöglicht eine Visualisierung der Fahrzeugpositionen im Zeitverlauf.

Die Implementierung basiert auf einer strukturierten Klassenarchitektur und verwendet Techniken zur Fehlerbehandlung, Datenhaltung und Erweiterbarkeit. Ein umfangreiches Testkonzept gewährleistet die korrekte Verarbeitung von Fehler-, Grenz- und Normalfällen. Zusätzlich wurde ein automatisiertes Testwerkzeug zur Validierung der Eingabedateien entwickelt.

Ausblick

Für zukünftige Erweiterungen bieten sich vielfältige Möglichkeiten:

- Berücksichtigung von Verkehrsregeln wie *Rechts-vor-Links* oder *Ampelschaltungen*,
- realitätsnähere Fahrdynamik durch Mindestabstände oder Überholverbote,
- Erweiterung auf mehrspurige Straßen sowie variable Geschwindigkeiten je Streckenabschnitt,
- Anbindung an eine GUI oder Weboberfläche zur interaktiven Parametereingabe und Echtzeitvisualisierung,
- Integration von OpenStreetMap-Daten zur Simulation realer Städte,

- Einbindung eines Seeds als Startparameter für die Zufallszahlengeneration, um Ergebnisse reproduzierbar zu machen,
- parallele Simulation und gleichzeitiges Schreiben der drei Ausgabedateien durch Multithreading.

Die entwickelte Simulation bietet somit eine stabile und erweiterbare Grundlage für weiterführende Projekte im Bereich Verkehrsmodellierung und -optimierung.