Industrie- und Handelskammer
Aachen

Abschlussprüfung Mathematisch-technischer Softwareentwickler/ Mathematisch-technische Softwareentwicklerin (IHK)

Entwicklung eines Softwaresystems zum Thema:

Indentifizierung von Straßenabschnitten Für Firma Kommunen

Sommer 2025

12.05.2025 - 16.05.2025

Ajouka, Tarek

Bereich	Berufsnummer	IHK-Nummer	Prüflingsnummer
30	6511	101	20080

Programmiersprache	Java
Ausbildungsort	IT Center RWTH Aachen

Aachen, 15. Mai 2025

Eigenständigkeitserklärung

Ajouka, Tarek

Bereich	Berufsnummer	IHK-Nummer	Prüflingsnummer
30	6511	101	20080

Ich erkläre verbindlich, 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 des Prüfprodukts als Prüfungsleistung ausschließt.

Aachen, 15. Mai 2025

Tarek Ajouka

(Ajouka, Tarek)

Inhaltsverzeichnis

1 Problembeschreibung

Der zunehmende Verkehr in vielen Städten führt zu einer hohen Belastung des Straßennetzes, insbesondere an Kreuzungen und Hauptzufahrten. Dies hat negative Folgen für Umwelt, Wirtschaft und Lebensqualität. Kommunen stehen vor der Herausforderung, geeignete Maßnahmen zur Reduzierung von Staus zu identifizieren und fundiert zu bewerten.

Die Firma MATSE Ing. unterstützt Kommunen in diesem Prozess durch verkehrstechnische Beratung. Um die Entscheidungsfindung zu verbessern, soll eine Lösung entwickelt werden, die typische Stauursachen nachvollziehbar macht und Zusammenhänge im Verkehrsverhalten aufzeigt.

Dazu ist ein System notwendig, das komplexe Verkehrssituationen realistisch modellieren kann und als Grundlage für zukünftige Planungsentscheidungen dient.

1.1 Eingabe

Die Eingabe der Verkehrssimulation erfolgt über eine strukturierte Textdatei, die alle für den Ablauf relevanten Informationen enthält. Diese Datei besteht aus mehreren logisch getrennten Abschnitten: dem Zeitraum der Simulation, den Einfallspunkten und den Kreuzungen.

1.1.1 Zeitraum

Der Zeitraum legt die Gesamtdauer der Simulation sowie das Zeitintervall zwischen den Ausgabeschritten fest. Angegeben werden zwei Zahlen: die Endzeit der Simulation in Sekunden und das Intervall für die Ausgabe (ebenfalls in Sekunden). Die Simulation beginnt immer bei Sekunde 0.

Beispiel: Zeitraum: 50 1

→ Die Simulation läuft von t = 0 bis t = 50, und es erfolgt eine Ausgabe alle 1 Sekunde.

1.1.2 Einfallpunkte

Einfallspunkte stellen die Stellen im Straßennetz dar, an denen neue Fahrzeuge regelmäßig in das System eintreten. Jeder Einfallspunkt ist eindeutig benannt und durch seine Koordinaten sowie die zugehörige Zielkreuzung und die Taktrate der Fahrzeuge definiert. Der Takt gibt an, in welchem zeitlichen Abstand (in Sekunden) neue Fahrzeuge an diesem Punkt erscheinen. Aufbau pro Zeile:

<Name> <x-Koordinate> <y-Koordinate> <Zielkreuzung> <Takt in Sekunden>

1.1.3 Kreuzungen

Kreuzungen bilden die Verbindungspunkte im Verkehrsnetz. Sie besitzen ebenfalls einen Namen und Koordinaten, gefolgt von einer Liste möglicher Zielkreuzungen mit den zugehörigen Verteilungsanteilen. Diese Anteile geben an, wie sich die Fahrzeuge auf die verschiedenen Richtungen aufteilen sollen.

Ein Beispiel: B 0 1 A 2 C 3 E 5

→ Die Kreuzung B liegt bei (0,1) und ist mit den Kreuzungen A, C und E verbunden. Das Verhältnis 2:3:5 gibt die Weiterfahrwahrscheinlichkeit in die jeweiligen Richtungen an, ausgenommen der Richtung, aus der ein Fahrzeug kommt.

Wichtig ist dabei: Fahrzeuge fahren niemals zurück in die Richtung, aus der sie gekommen sind. Die Anteile werden stets auf die verbleibenden Richtungen umgerechnet, sodass sie immer 100% ergeben.

1.2 Format der Ausgabe

Das Simulationsprogramm erzeugt drei Ausgabedateien, die jeweils unterschiedliche Aspekte des Verkehrsflusses darstellen. Diese Dateien dienen der Analyse, Visualisierung und Bewertung der simulierten Verkehrsdaten.

1.2.1 Plan.txt

Die Datei Plan.txt enthält die geometrischen Informationen des Straßennetzes in Form von Koordinatenpaaren. Jede Zeile stellt eine Straße dar und besteht aus vier Zahlenwerten: den x- und y-Koordinaten des Startpunkts sowie den x- und y-Koordinaten des Zielpunkts.

Diese Datei ist besonders wichtig für die grafische Darstellung, da sie die Grundlage für die Visualisierung der Straßen bildet.

1.2.2 Statistik.txt

In Statistik.txt werden quantitative Auswertungen zur Streckenauslastung festgehalten. Für jede Straße werden zwei Werte berechnet:

- Gesamtanzahl Fahrzeuge pro 100 m: Gibt an, wie viele Fahrzeuge über die gesamte Simulationsdauer kumulativ den Straßenabschnitt befahren haben, normiert auf 100 Meter Streckenlänge.
- Maximale Anzahl Fahrzeuge pro 100 m: Zeigt die höchste gleichzeitige Anzahl an Fahrzeugen auf diesem Straßenabschnitt während eines beliebigen Zeitpunkts der Simulation, ebenfalls normiert auf 100 Meter.

Die Darstellung erfolgt in zwei Blöcken – einmal für die kumulierten Werte, einmal für die Maximalwerte.

1.2.3 Statistik.txt

Die Datei Fahrzeuge.txt enthält eine Momentaufnahme der Verkehrslage zu jedem Zeitschritt. Für jeden Zeitpunkt werden alle aktuell auf der Karte befindlichen Fahrzeuge aufgelistet, inklusive ihrer Position, der Zielkreuzung und der eindeutigen Fahrzeug-ID.

Ein neuer Zeitschritt beginnt immer mit einer Zeile im Format:

*** t = <Zeitpunkt>

Darauf folgen pro Fahrzeug:

<x-Position> <y-Position> <Ziel-x> <Ziel-y> <ID>

Diese Datei dient sowohl der textbasierten Analyse als auch der grafischen Darstellung mit dem bereitgestellten Tool Plot.py.

2 Modelierung

Um das gegebene Problem mithilfe von einem Progamm zu lösen ist eine Modelierung notwendig.

2.1 Datenstrukturbeschreibung

Um das Problem im Programm dar zustellen wurden verschiedene Klassen genutzt. Hierbei wurde auch berücksichtiget, dass diese für Erweiterungen und Veränderungen offen sind.

Punkt

Repräsentiert eine zweidimensionale Position mit x- und y-Koordinaten. Dient als Basisklasse für Einfallpunkt und Kreuzung.

Punkt
+x : double +y : double
«abstract» getType () : string «abstract» getName() : String

Abbildung 2.1: Punkt

Einfallpunkt

Ein Einfallpunkt ist eine Stelle, an der Fahrzeuge in das Netz eintreten. Diese Klasse enthält Informationen über den Ort, das Ziel, das die Fahrzeuge ansteuern, und die Frequenz, mit der sie erzeugt werden (Takt). Sie erbt von Punkt, das heißt, sie besitzt auch x- und y-Koordinaten. Außerdem stellt sie Methoden zur Verfügung, um Ziel, Name und Typ abzufragen.

Einfallpunkt «extends Punkt»
+name : String +ziel : Kreuzung +takt : int
Einfallpunkt(String, x, y, ...) getType () : string setZiel(Punkt ziel) : void

Abbildung 2.2: Einfallpunkt

Kreuzung

Die Klasse Kreuzung modelliert einen Knotenpunkt im Straßennetz. Sie erbt von Punkt und besitzt daher eine Position. Über die Liste verbindungen sind alle ausgehenden Verbindungen von dieser Kreuzung definiert. Fahrzeuge treffen hier Entscheidungen, wie sie weiterfahren.

Kreuzung «extends Punkt»
+name : String +verteilungsAnteile : Map¡Punkt,Integer¿
kreuzung(String, x, y) getType () : string addVerbindung(Verbindung v) : void

Abbildung 2.3: Kreuzung

Verbindung

Diese Klasse stellt eine Verbindung (gerichtete Kante) zwischen zwei Kreuzungen dar. Jede Verbindung ist mit einem anteil versehen, der angibt, wie hoch die Wahrscheinlichkeit ist, dass ein Fahrzeug diesen Weg wählt. Die Zielkreuzung wird ebenfalls gespeichert.

Verbindung
+von : Punkt +nach : Punkt +gesamt : double +max : double
Verbindung(Punkt,Punkt) getLaenge () : double addVerbindung(Verbindung v) : void

Abbildung 2.4: Verbindung

Fahrzeuge

Die Klasse Fahrzeug repräsentiert ein einzelnes Fahrzeug innerhalb der Simulation. Sie verwaltet die Position, aktuelle Verbindung sowie die konstante Geschwindigkeit. Die Methode bewegen() ist zentral für die Fortbewegung der Fahrzeuge entlang der Straßenverbindungen. Fahrzeuge können durch die Kreuzungen geleitet werden und ändern entsprechend ihre Richtung.

Fahrzeuge
+x : double +y : double +actuell : Verbindung +speed : double
Fahrzeug(Einfallpunkt,Verbindungen) bewegen(Int sec,Set<Verbindung>,boolean ifZeitsSchritt)

Abbildung 2.5: Fahrzeuge

Einlesen

Die Klasse Einlesen ist zuständig für das Parsen der Eingabedatei. Sie liest Zeilenweise Informationen zu Einfallspunkten, Kreuzungen, Verbindungen und Zeiteinstellungen ein. Die Daten werden in entsprechenden Listen und Maps gespeichert und stehen der Simulation zur Verfügung. Diese Klasse ist essenziell für die Initialisierung.

<<interface>> **Einlesen**
+endeZeit : int +zeitSchritt : int +punkte : Map¡String, Punkt¿ +einfallpunkte : ArrayList +kreuzungen : ArrayList +verbindungen : HashSet
ladeAusDatei : void

Abbildung 2.6: Einlesen

Simulation

Führt die gesamte Simulation aus: Zeitschritte, Fahrzeugerzeugung, Bewegung und Ausgabe.

Simulation
start(Pfad eingabe,Pfad ausgabe) : void

Abbildung 2.7: Simulation

Beziehungen der Klassen Punkt,Einfallpunkt und Kreuzung

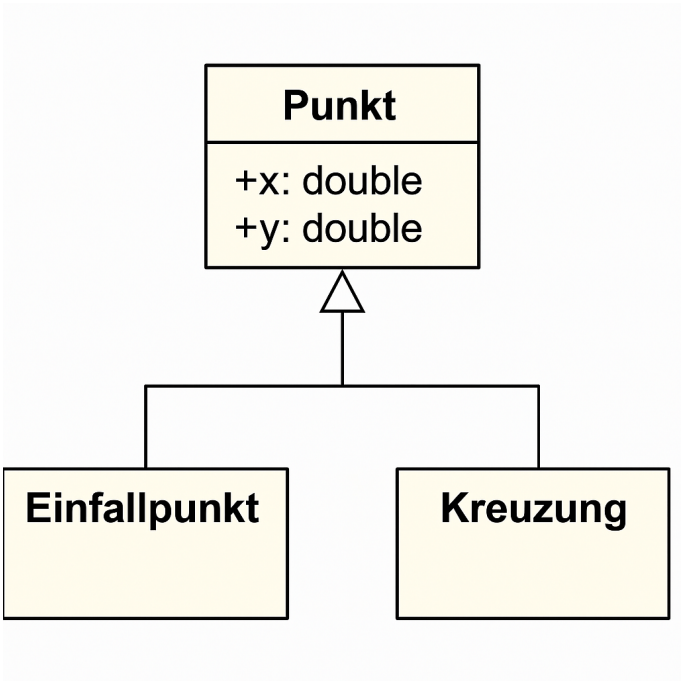

Abbildung 2.8: Die Beziehung

2.2 Programmbeschreibung

Das Programm hat seinen Einstiegspunkt in der main-Funktion der Klasse Main. Dort wird zunächst die Methode ladeAusDatei() aus der Klasse Einlesen aufgerufen, um die für die Simulation notwendigen Daten aus einer strukturierten Textdatei einzulesen. Dabei handelt es sich um Informationen zu Einfallpunkten, Kreuzungen, Verbindungen sowie zur Gesamtlaufzeit und den Zeitschritten der Simulation. Nach erfolgreichem Einlesen werden die erzeugten Objekte über Getter-Methoden aus Einlesen abgerufen, darunter Listen von Einfallpunkt-, Kreuzung- und ggf. vorbereiteten Verbindung-Objekten sowie die Angaben zu Startzeit und Schrittweite. Diese Daten werden anschließend an ein Simulation-Objekt übergeben, das die eigentliche Verkehrssimulation durchführt. Das Simulation-Objekt verwaltet die Bewegung der Fahrzeuge entlang der definierten Straßenabschnitte über mehrere Zeitschritte hinweg. Zu Beginn jedes Schritts wird überprüft, ob neue Fahrzeuge erzeugt werden müssen (abhängig vom Takt der Einfallpunkte). Bestehende Fahrzeuge bewegen sich entsprechend ihrer Geschwindigkeit entlang der definierten Verbindungen. Die Entscheidung, an Kreuzungen in welche Richtung weitergefahren wird, basiert auf den im Eingang festgelegten Verteilungsanteilen. Während der Simulation werden drei Ausgabedateien erstellt:

- `Plan.txt`: enthält alle Koordinaten der Verbindungen im Straßennetz.

- `Fahrzeuge.txt`: dokumentiert die Position aller Fahrzeuge zu jedem Zeitschritt.

- `Statistik.txt`: fasst die kumulierten und maximalen Fahrzeugzahlen pro Streckenabschnitt zusammen.

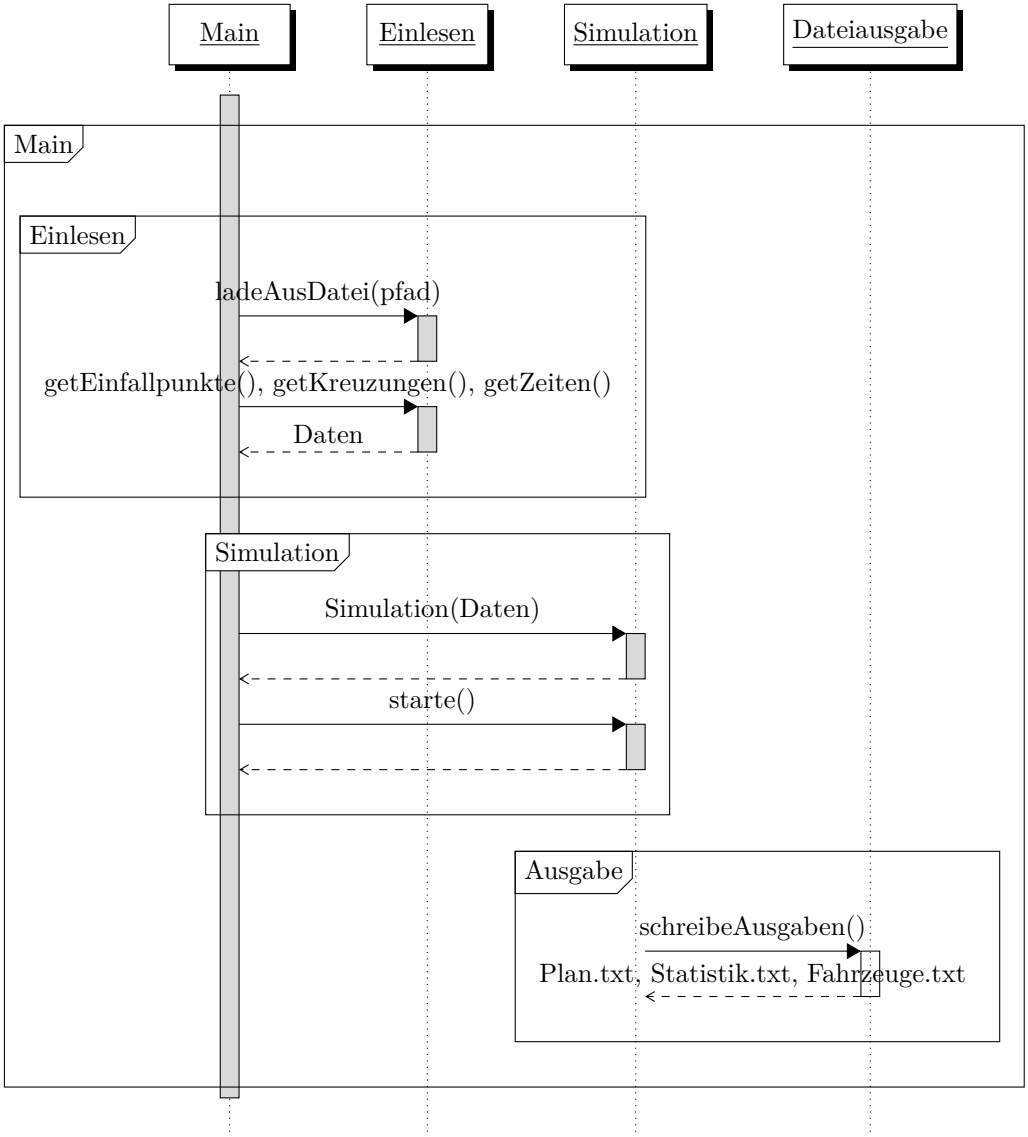

Abbildung 2.9: Ablauf der Hauptlogik im Verkehrssimulationsprogramm

3 Beschreibung des implementierten Lösungsverfahrens

3.1 Einlesen

Die Klasse Einlesen übernimmt die zentrale Aufgabe, alle benötigten Daten aus einer externen Eingabedatei in das Programm zu laden und daraus entsprechende Java-Objekte zu erzeugen. Sie dient als Brücke zwischen der externen Textdatei und der internen Datenstruktur der Simulation. Zu den Hauptaufgaben gehören das Öffnen und zeilenweise Einlesen der Datei, das Erkennen von Datenblöcken wie Zeitraum, Einfallpunkte und Kreuzungen sowie das Parsen dieser Abschnitte. Beim Einlesen werden entsprechende Objekte erzeugt – etwa Einfallpunkte, Kreuzungen und Verbindungen – und in geeigneten Datenstrukturen wie Listen oder Maps gespeichert. Zudem speichert die Klasse Informationen zur Taktung der Simulation, wie Zeitschritt und Endzeit. Namen von Zielkreuzungen werden zunächst als String verarbeitet und später aufgelöst, sobald alle Kreuzungen bekannt sind. Nach Abschluss des Ladevorgangs können die erzeugten Objekte und Werte über Methoden wie getEinfallpunkte(), getKreuzungen(), getZeitschritt() und getEndeZeit() abgerufen werden. Diese Werte werden anschließend an die Simulation übergeben. Die Klasse Einlesen enthält selbst keine Logik zur Steuerung der Simulation. Sie ist ausschließlich für die Datenbereitstellung zuständig und trennt damit sauber die Verantwortung zwischen Eingabe, Verarbeitung und Ablauf. Durch den modularen Aufbau kann die Klasse später leicht um neue Formate, Eingabestrukturen oder Validierungen erweitert werden.

3.2 Simulation.start()

Die Methode starte() der Klasse Simulation bildet das Herzstück der zeitbasierten Verkehrssimulation. Sie koordiniert den vollständigen Ablauf über die gesamte definierte Simulationsdauer hinweg. Zu Beginn setzt die Methode eine interne Zeitschleife auf, die von t = 0 bis zur definierten Endzeit in gleichmäßigen Schritten (abhängig vom Zeitschritt) läuft. Innerhalb jedes Zeitintervalls prüft die Methode, ob an bestimmten Einfallpunkten ein neues Fahrzeug erzeugt werden muss. Dies hängt vom definierten Takt des jeweiligen Einfallpunkts ab. Alle vorhandenen Fahrzeuge werden innerhalb jedes Schrittes über die Methode bewegeFahrzeuge() aktualisiert. Dabei wird ihre Position auf der aktuellen Verbindung angepasst, ihre Geschwindigkeit berücksichtigt und bei Erreichen einer Kreuzung ein neuer Zielpunkt gewählt. Nach der Bewegung der Fahrzeuge ruft starte() Methoden auf, um die aktuelle Verkehrssituation in einer Ausgabedatei festzuhalten. Dies betrifft insbesondere die Datei Fahrzeuge.txt, die zu jedem Zeitschritt die Position aller Fahrzeuge dokumentiert. Am Ende der gesamten Schleife werden zusätzlich die beiden Dateien Plan.txt und Statistik.txt erstellt. Diese enthalten die statischen Netzkoordinaten sowie eine statistische Auswertung der Fahrzeugverteilungen auf den Verbindungen. Insgesamt ist starte() dafür verantwortlich, dass die Simulation über alle Zeitschritte hinweg korrekt abläuft, neue Fahrzeuge erzeugt werden, bestehende sich realistisch bewegen und die entsprechenden Ergebnisse vollständig dokumentiert werden.

3.3 Fahrzeug.bewegen()

Die Methode bewegen() in der Klasse Fahrzeug ist für die Fortbewegung eines einzelnen Fahrzeugs innerhalb eines Zeitschritts zuständig. Sie berechnet die neue Position des Fahrzeugs entlang der aktuellen Verbindung, unter Berücksichtigung der konstanten Geschwindigkeit und der Länge der Verbindung. Zu Beginn prüft die Methode, ob das Fahrzeug überhaupt einer Verbindung zugeordnet ist. Falls nicht, wird eine geeignete Verbindung anhand der Zielkreuzung und möglicher Weiterfahrten zugewiesen. Danach wird die zurückgelegte Distanz auf Basis der verstrichenen Zeit und der Geschwindigkeit berechnet. Die Methode vergleicht dann die bisher zurückgelegte Strecke mit der Gesamtlänge der Verbindung. Wenn das Fahrzeug die Zielkreuzung innerhalb des aktuellen Zeitschritts erreicht, wird es dort platziert und eine neue Verbindung ausgehend von der Kreuzung gewählt – sofern das Fahrzeug weiterfahren soll. Falls die Zielkreuzung noch nicht erreicht wurde, wird die Position des Fahrzeugs entlang der Verbindung aktualisiert, wobei ein linearer Verlauf zwischen Start- und Zielkoordinaten berechnet wird. Die neue Position (x, y) wird entsprechend angepasst. Die Methode gibt abschließend zurück, ob das Fahrzeug noch im Netz aktiv ist oder ob es sein Ziel erreicht hat. Dies kann für spätere Verarbeitungsschritte wie Entfernung oder Auswertung genutzt werden.

4 Begründung der Erweiterbarkeit gemäß den Ideen des Auftraggebers

4.1 Ideen

- Die Fahrzeuge können nicht überholen, sondern müssen einen Mindestabstand zum vorausfahrenden Fahrzeug einhalten.
- Der Verkehrsfluss an den Kreuzungen wird über Ampeln, Kreisverkehre oder „Rechts vor Links" geregelt.
- Es gibt Strecken mit mehreren Fahrbahnen pro Richtung.
- Die Geschwindigkeiten der Fahrzeuge sind nicht konstant, sondern können sich abhängig von den Streckenabschnitten ändern.

4.2 Begrüdung

Die Verkehrssimulation wurde von Beginn an objektorientiert konzipiert, mit dem Ziel, eine saubere Trennung zwischen Datenstrukturen, Logik und Ein-/Ausgabe zu gewährleisten. Dies erleichtert nicht nur die Wartung des Programms, sondern insbesondere auch seine zukünftige Erweiterung durch weitere Funktionalitäten. Die zentrale Idee besteht darin, dass jedes wichtige Element des Verkehrsnetzes – wie Fahrzeuge, Kreuzungen, Verbindungen oder Einfallpunkte – als eigene Klasse mit klar definierten Zuständigkeiten modelliert ist. Dadurch kann Verhalten, das nur bestimmte Objekte betrifft, direkt innerhalb der zugehörigen Klasse geändert oder erweitert werden, ohne bestehende Abläufe in anderen Klassen zu stören. Ein Beispiel ist die mögliche Einführung von Ampeln an Kreuzungen. Die bestehende Klasse Kreuzung könnte um ein neues Attribut wie ‘hatAmpel‘ oder ein ‘AmpelStatus‘ erweitert werden. In der Fahrzeugklasse, genauer in der Methode ‘bewegen()‘, müsste bei Ankunft an einer Kreuzung lediglich geprüft werden, ob diese eine Ampel besitzt und ob sie auf Rot steht. Darauf basierend könnte das Fahrzeug anhalten oder weiterfahren. Diese Erweiterung erfordert keine grundlegende Umstrukturierung des Programms – lediglich gezielte Ergänzungen an den betroffenen Klassen. Ein weiteres Szenario ist die Einführung mehrspuriger Straßen. Auch hierfür könnte die Klasse Verbindung um eine Anzahl an Fahrspuren erweitert werden. Fahrzeuge könnten dann eine Spur als zusätzliches Attribut speichern und beim Wechsel von Verbindungen berücksichtigen, ob ein Überholen möglich ist. Auch hier würde die Erweiterung durch gezielte Ergänzungen und ohne umfassende Neustrukturierung erfolgen. Zusätzlich wurde die Geschwindigkeit der Fahrzeuge in der Klasse Fahrzeug bewusst als separates Attribut implementiert. Eine künftige Erweiterung, bei der Fahrzeuge ihre Geschwindigkeit dynamisch anpassen – zum Beispiel in Abhängigkeit von der Verbindung oder der Verkehrslage – kann so unkompliziert realisiert werden. Die Verbindungsklasse könnte beispielsweise um ein Geschwindigkeitslimit ergänzt werden, das von Fahrzeugen bei der Bewegungsberechnung berücksichtigt wird. Dank der objektorientierten Struktur kann jede neue Regel (z.B. Mindestabstand, Vorrangregelung, Umweltzonen) innerhalb der betroffenen Klasse implementiert werden. Dies reduziert Komplexität, verhindert Seiteneffekte und erhöht

die Lesbarkeit sowie die Testbarkeit des Codes. Insgesamt schafft die gewählte Architektur eine stabile Grundlage für zukünftige Erweiterungen aller Art.

5 Dokumentation

5.1 Entwicklerdokumentation

Entwicklerdokumentation – Verkehrssimulation

1. Überblick Das Projekt simuliert den Verkehrsfluss in einem einfachen Straßennetz über eine definierte Zeitspanne. Fahrzeuge entstehen an Einfallpunkten, bewegen sich entlang von Verbindungen und durchqueren Kreuzungen, wobei Entscheidungen über die Fahrtrichtung getroffen werden. Die Simulation gibt verschiedene Dateien aus, die eine Analyse des Verkehrsverhaltens ermöglichen.

2. Projektstruktur Das Projekt folgt einem objektorientierten Ansatz mit klar getrennten Klassen für die wichtigsten Konzepte: - Punkt: Basisklasse für Positionen im Raum - Einfallpunkt: Erbt von Punkt, erzeugt Fahrzeuge in regelmäßigen Takten - Kreuzung: Erbt von Punkt, verteilt Fahrzeuge auf Verbindungen - Verbindung: Repräsentiert gerichtete Straßenabschnitte zwischen Kreuzungen - Fahrzeug: Bewegt sich entlang einer Verbindung mit fester Geschwindigkeit - Einlesen: Liest die Eingabedaten aus Datei und erstellt alle Objekte - Simulation: Führt die zeitgesteuerte Simulation durch - Main: Startpunkt des Programms

3. Programmablauf - Main ruft Einlesen auf, um Eingabedaten aus Datei zu laden. - Die geladenen Daten (Einfallpunkte, Kreuzungen, Verbindungen, Zeiten) werden an Simulation übergeben. - Die Methode starte() in Simulation steuert die Zeitschritte, Fahrzeugerzeugung, Bewegung und Dateiausgabe. - Ergebnisse werden in Plan.txt, Statistik.txt und Fahrzeuge.txt ausgegeben.

4. Zentrale Klassen und Methoden - Einlesen: - ladeAusDatei(String pfad): Liest Datei und erstellt Objekte - getEinfallpunkte(), getKreuzungen(), getZeitschritt(), getEndeZeit() - Simulation: - starte(): Führt die Hauptzeitschleife aus - bewegeFahrzeuge(): Aktualisiert Positionen der Fahrzeuge - schreibeAusgabe(): Erstellt die Ausgabedateien - Fahrzeug: - bewegen(): Berechnet neue Position oder entscheidet bei Kreuzung über nächsten Weg

5. Erweiterbarkeit Durch die objektorientierte Struktur lassen sich neue Regeln oder Konzepte wie Ampeln, mehrspurige Straßen oder dynamische Geschwindigkeiten gezielt in den zuständigen Klassen ergänzen.

6. Konventionen - Die Datei muss UTF-8 codiert sein. - Es werden nur Eingabedateien mit dem erwarteten Format verarbeitet. - Die Ausgabedateien werden im Arbeitsverzeichnis erstellt.

7. Abhängigkeiten - Standard-Java-API (keine externen Bibliotheken) - Voraussetzung: Java JDK 8 oder höher

8. Hinweise zur Weiterentwicklung - Neue Verkehrsregeln sollten möglichst innerhalb von Fahrzeug oder Kreuzung integriert werden. - Für Visualisierungen kann die Datei Plan.txt genutzt und mit Tools wie Plot.py erweitert werden. - Testfälle lassen sich durch gezielte Modifikation der Eingabedatei erzeugen.

.

5.2 Benutzerhandbuch: Verkehrssimulation

5.2.1 1. Programmstart

– Doppelklicken Sie auf die Datei $\texttt{run}_s imulation.batDasProgrammkompiliertautomatischalleJava-Dateien$

5.2.2 2. Hauptmenü

Nach dem Start erscheint folgendes Menü:

–
```
Waehlen Sie eine Option:
1 - Vordefinierte IHK-Beispiele ausfuehren
2 - Eigene Eingabedatei simulieren
3 - Beenden
```

5.2.3 3. Optionen im Detail

3.1 IHK-Beispiele (Option 1)

– Führt automatisch alle vordefinierten IHK-Beispiele aus

– Verarbeitet Beispieldateien aus dem Vorlage-Ordner

– Ergebnisse werden in den jeweiligen IHK-Ordnern gespeichert

3.2 Eigene Simulation (Option 2)

– Ermöglicht die Simulation mit eigener Eingabedatei

– Pfadeingabe-Möglichkeiten:

  * Datei im gleichen Ordner: `Eingabe.txt`
  * Datei in Unterordner: `Ordnername/Eingabe.txt`
  * Kompletter Pfad: `C:/MeinOrdner/Eingabe.txt`

3.3 Beenden (Option 3)

– Beendet das Programm

5.2.4 4. Ausgabedateien

Das Programm erstellt drei Ausgabedateien:

4.1 Fahrzeuge.txt

– Enthält Positionsdaten aller Fahrzeuge

– Wird bei jedem Zeitschritt aktualisiert

4.2 Plan.txt

- Zeigt alle Straßenverbindungen
- Format: Start- und Endkoordinaten

4.3 Statistik.txt

- Gesamtanzahl Fahrzeuge pro 100m
- Maximale Anzahl Fahrzeuge pro 100m

5.2.5 5. Fehlerbehebung

5.1 Datei nicht gefunden

- Überprüfen Sie die Schreibweise des Pfads
- Stellen Sie sicher, dass die Datei existiert
- Prüfen Sie das aktuelle Verzeichnis

5.2 Kompilierungsfehler

- Überprüfen Sie die Java-Installation
- Stellen Sie sicher, dass alle Java-Dateien vorhanden sind

5.2.6 6. Beispiel-Pfadeingaben

- `Eingabe.txt` (Gleicher Ordner)
- `./Eingabe.txt` (Alternative für gleichen Ordner)
- `MeinOrdner/Eingabe.txt` (Unterordner)
- `C:/MeineProjekte/Test/Eingabe.txt` (Vollständiger Pfad)

5.2.7 7. Wichtige Hinweise

- Ausgabedateien werden im gleichen Ordner wie die Eingabedatei erstellt
- Nach jeder Simulation kehren Sie zum Hauptmenü zurück
- Fehlermeldungen werden im Konsolenfenster angezeigt

Bei weiteren Fragen wenden Sie sich bitte an den Support.

6 Testbeispiele

6.1 automatischen Ausführung der vorgegeben Beispiele

Automatisierte Tests wurden mit Hilfe einer .bat Datei durchgeführt. Diese kann auf einem System mit Windows direkt ausgeführt werden. Ich habe die in der Aufgabenstellung geforderten Beispiele durchgeführt und dabei realistische Ausgaben erhalten. Auch wenn die Ergebnisse nicht exakt identisch mit den Beispielausgaben sind, stimmen sie in ihrer Struktur und ihrem Ablauf weitgehend überein. Dies ist nachvollziehbar, da jedes Fahrzeug beim Erzeugen eine individuelle Geschwindigkeit mit zufälliger Abweichung erhält. Zudem wählen Fahrzeuge an jeder Kreuzung zufällig eine der möglichen Verbindungen gemäß den festgelegten Verteilungsverhältnissen. Diese zufallsbasierten Elemente führen zu natürlichen Unterschieden im Ablauf, spiegeln jedoch das erwartete Verhalten der Simulation korrekt wider.

6.2 Weitere repräsentative Beispiele

Zusätzlich zu den IHK-Beispielen wurden eigene Testfälle definiert, um gezielt zentrale Funktionen des Programms zu prüfen. Die folgenden Tests wurden mit dem implementierten Code durchgeführt und decken wesentliche Bestandteile der Klassen `Einlesen`, `Simulation`, `Fahrzeug`, `Kreuzung` und `Verbindung` ab.

Testfall 1: Einfallpunkt mit Taktprüfung

Ein einzelner Einfallpunkt mit einem Takt von 2 Sekunden wurde verwendet. In der Methode `starte()` der Klasse `Simulation` erfolgt die Fahrzeugerzeugung nur dann, wenn `t % takt == 0` gilt.

Geprüft: Takterzeugung über Zeitschleife und zeitgesteuerte Fahrzeugerstellung.

Testfall 2: Verteilungslogik an Kreuzungen

Eine Kreuzung mit drei ausgehenden Verbindungen (Verteilungsanteile 30, 40, 30) wurde getestet. Die zufällige Auswahl der Weiterfahrt basiert auf dem Verteilungsverhältnis.

Geprüft: Zufallsbasierte Weiterleitungslogik gemäß definierter Wahrscheinlichkeiten.

Testfall 3: Fahrzeugbewegung über Verbindung

Eine lange Verbindung wurde verwendet, um die Berechnung der Position eines Fahrzeugs in `Fahrzeug.bewegen()` zu prüfen. Die Interpolation der Position anhand von Geschwindigkeit und Zeit wurde erfolgreich nachvollzogen.

Geprüft: Bewegung entlang einer Verbindung inkl. Distanz- und Zeitberechnung.

Testfall 4: Fahrzeugwechsel an Kreuzung

Ein Fahrzeug wurde so platziert, dass es bei der Kreuzung ankommt. Der Übergang zu einer neuen Verbindung wurde korrekt ausgeführt.

Geprüft: Auswahl und Wechsel zur nächsten Verbindung bei Erreichen der Kreuzung.

Testfall 5: Statistik-Ausgabe

Ein Test mit vielen Fahrzeugen auf kurzen Strecken wurde durchgeführt, um die Auswertung in `Statistik.txt` zu validieren.

Geprüft: Zählung und Ausgabe der Fahrzeuganzahl pro Verbindung im korrekten Format.

6.3 Grenz- und Fehlerfälle

Bei der Umsetzung der Verkehrssimulation habe ich verschiedene Grenz- und Fehlerfälle berücksichtigt, die während der Laufzeit auftreten können oder zu unerwartetem Verhalten führen könnten.

Einfallpunkte mit extrem kleinem oder großem Takt

Ein Einfallpunkt mit einem Takt von 1 erzeugt bei jedem Zeitschritt ein neues Fahrzeug, was zu einer sehr hohen Fahrzeugdichte führt. Ein extrem hoher Takt wiederum führt dazu, dass kaum Fahrzeuge erzeugt werden. Deshalb achte ich darauf, sinnvolle Werte für den Takt zu verwenden.

Verbindungen mit einem Verteilungsanteil von 0

Wenn eine Verbindung an einer Kreuzung den Anteil 0 hat, wird sie im Zufallsverfahren nie gewählt. Dieses Verhalten ist korrekt, sollte aber bewusst eingesetzt werden, um keine „toten“ Wege im Netzwerk zu haben.

Kreuzungen ohne ausgehende Verbindungen

Sollte ein Fahrzeug eine Kreuzung erreichen, an der es keine Weiterverbindung gibt, würde es dort stehen bleiben. Solche Kreuzungen sollten als Endpunkte betrachtet werden. Eine automatische Behandlung dieser Situation könnte das Fahrzeug aus dem Netzwerk entfernen.

Verbindungen mit einer Länge von 0

Eine Verbindung mit Start- und Zielpunkt an derselben Position führt zu einer Länge von 0. In meiner Bewegungsgleichung würde das zu einer Division durch Null führen. Deshalb müssen solche Verbindungen vermieden oder vorab abgefangen werden.

Fahrzeuge, die ihr Ziel zu schnell erreichen

Bei sehr kurzen Verbindungen oder hoher Geschwindigkeit kann es vorkommen, dass ein Fahrzeug innerhalb eines Zeitschritts sein Ziel erreicht. Ich achte darauf, dass in einem Zeitschritt maximal eine Verbindung durchfahren wird, um unrealistische Sprünge zu vermeiden.

Ungültige oder fehlerhafte Eingabedaten

Fehlerhafte Zielnamen oder falsch strukturierte Eingabedateien können beim Einlesen Probleme verursachen. Ich habe deshalb darauf geachtet, die Eingabedaten möglichst robust zu parsen und würde in einer späteren Erweiterung Validierungsmechanismen integrieren.

A Arbeitsumgebung und Arbeitshilfen

A.1 Hardware

Systemhersteller	Lenovo
Systemmodel	20S00005GE
System-SKU	LENOVO_MT_20S0_BU_Think_FM_ThinkPad T14 Gen 1
Systemtyp	x64-basierter PC
Betriebssystem	Microsoft Windows 10 Enterprise
Prozessor	Inter(R) Core(TM) i5-10210U CPU
Arbeitsspeicher	16,0 GB

Tabelle A.1: Angaben zur Hardware

A.2 Software

Programm	**Version**
Microsoft Visual Studio Community 2022 (64-Bit)	Version 17.9.5
Windows-Terminal	1.19.11213.0
Adobe Acrobat Reader DC x64	24.002.20736
Notepadd++ Team Notepad++ x64	8.6.5.0
Igor Pavlov 7-Zip	23.01.00.0
MikTex.org MikTex	21.12
Git Development Community Git	2.45.0.1
Mozilla Firefox ESR x64	115.11.0
Microsoft .NET SDK	8.0.202
TeXworks	0.6.6
PuzzleTester.jar	Unbekannt
JAVA SDK	21.0.3

Tabelle A.2: Angaben zur Software

B Literaturverzeichnis

C Tabellenverzeichnis

D Abbildungsverzeichnis