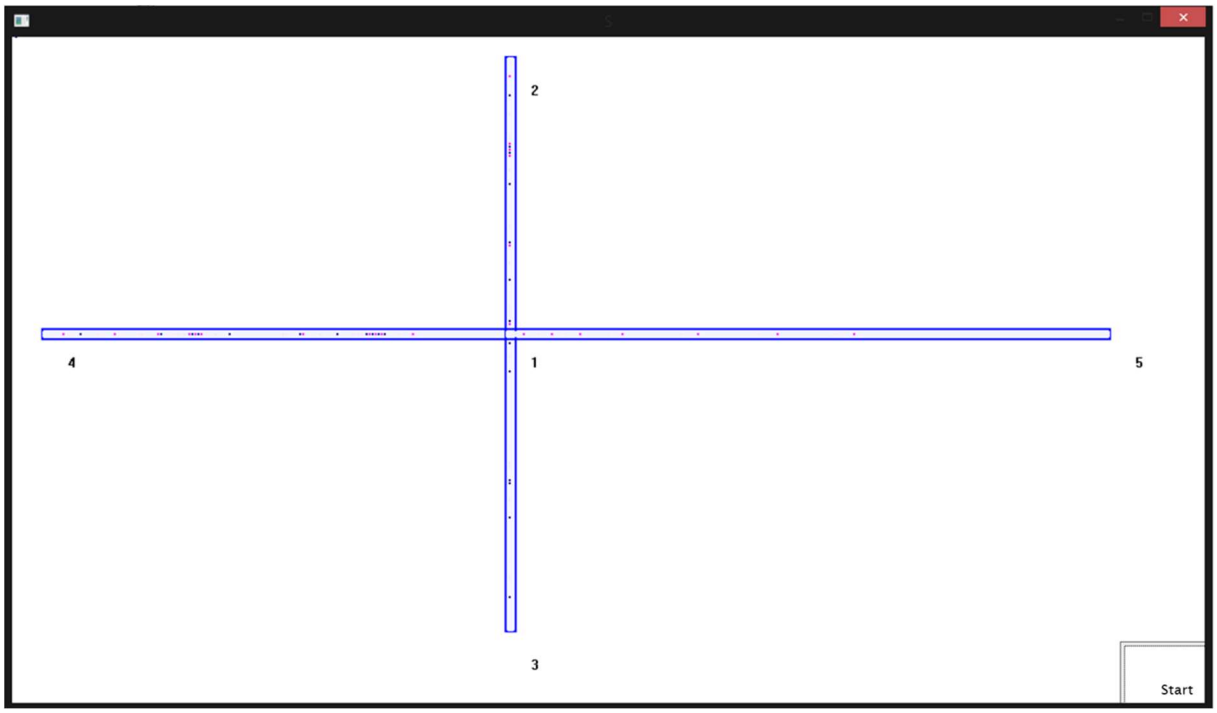
Programmierung eines Verkehrssimulators mit C++



September 2020

Bernhard Gepp

Inhalt

1 Der Verkehrssimulator - Programmbeschreibung aus verkehrsplanerischer Sicht

Dieses Kapitel behandelt die Grundlagen jener Modellbildungen, welche in diesem „Verkehrssimulator“ implementiert wurden.

Der Zweck eines (mikroskopischen) Verkehrssimulators besteht darin, das dynamische Verhalten von Fahrzeugen in einem Verkehrsnetzwerk zu simulieren und graphisch darzustellen. Ferner dient ein Verkehrssimulator als Planungsinstrument, indem beispielsweise bestehende Verkehrsszenarien künstlich nachgebaut werden und um in einem Folgeschritt Auswirkungen von Veränderungen ergründet werden können.

1.1 Eigenschaften eines Verkehrssimulators

1.1.1 Grundsätzliche Zusammenhänge

Das Verkehrsgeschehen wird bei lokaler Betrachtung durch drei in Wechselwirkung stehende Größen beschrieben:

1. Verkehrsstärke (Fahrzeuge pro Zeiteinheit)
2. Verkehrsdichte (Fahrzeuge pro Längeneinheit)
3. Geschwindigkeit

Diese Wechselwirkungen ergeben sich durch die Interaktion zwischen der Infrastruktur (Straßen und Kreuzungen) und den fluiden Elementen (Fahrzeuge) der Simulation. Darüber hinaus spielt die gegenseitige Beeinflussung der Fahrzeuge untereinander eine wichtige Rolle.

1.1.2 Simulation der Verkehrsflusscharakteristiken

Bei geringem Verkehrsaufkommen (niedrige Verkehrsstärke) steht es jedem Fahrzeug frei selbst eine bevorzugte Reisegeschwindigkeit zu wählen. Dadurch ergibt sich ein heterogenes Geschwindigkeitsbild aller Fahrzeuge. Bei hohen Verkehrsstärken (im sogenannten gebundenen Verkehr) resultiert ein homogenes Geschwindigkeitsbild über das gesamte Fahrzeugkollektiv.

In diesem Programm wird jedem simulierten Fahrzeug ein per Zufallsgenerator erzeugter Wert einer Reisegeschwindigkeit (aus dem Wertebereich 60 km/h bis 110 km/h) zugewiesen. Zur Verkehrsflussmodellierung wurde ein „Car-Following-Model“ implementiert. Dieses Model regelt den Verkehrsfluss, wenn Fahrzeuge unterschiedlichen Reisegeschwindigkeiten aufeinandertreffen. Ferner regelt dieses Model bei 2-streifigen Fahrbahnen allfällige Überholvorgänge.

1.1.3 Arten der Verkehrssimulation

<u>Simulation ohne bzw. mit Beeinträchtigungen:</u>

Ein wichtiges Feature von Verkehrssimulationsapplikationen ist das Simulieren von Schwachstellen. Per Mausklick kann ein Fahrstreifen blockiert werden, was in einem Stau resultiert. Das Programm erkennt die Verkehrsstockung und leitet die Fahrzeuge auf andere Routen im Netzwerk um.

1.1.4 Infrastruktur

Netzwerke

Ein grundlegender Bestandteil aller Verkehrssimulationssoftwaretools ist die Repräsentation **eines oder mehrerer Netzwerke**, in denen das Verkehrsgeschehen simuliert wird.

Erstellung eines oder mehrerer Verkehrsnetzwerke:

- In diesem Programm wurde ein Ansatz gewählt, der es dem Anwender ermöglicht, selbst per Mausklick ein Verkehrsnetzwerk in der Applikation bestehend aus Richtungsfahrbahnen mit einem oder zwei Fahrsteifen zu zeichnen.
- Dieser einfache Ansatz muss jedoch dahingehend verbessert werden, dass die erstellte Software dazu fähig ist, aus qualitativ nicht belastbaren Usereingaben ein oder mehrere Netzwerke zur Verkehrssimulation verlässlich zu erstellten.

Routen

Aufbauend auf den Netzwerkstrukturen sind die „**Routen**“ innerhalb Netzwerke von zentraler Bedeutung bei der Verkehrssimulation. Vor Beginn der Simulation müssen alle möglichen Routen zwischen Start- und Endpunkten erfasst und qualitativ bewertet werden. Im Simulationsbetrieb wird eine für Simulationssoftwaren typische Vorschrift verwendet, die jedem Fahrzeug die Route mit der kürzesten Reisezeit durch das Netzwerk zuweist. Beispiele für Verkehrssimulationen mit dazugehöriger Routentabelle sind in Abbildung 1 dargestellt. Die Route 2-1-3, welche in der Routentabelle von Bsp. A abgebildet ist, bedeutet, dass ein Fahrzeug vom Startpunkt 2 über den Knotenpunkt 1 zum Endpunkt 3 gelotst wird. In Bsp. B sind z.B. vom Startpunk 10 acht verschiedene Routen durch das Netzwerk möglich. Weiters ist in der zugehörigen Routentabelle ersichtlich, dass vom Startpunkt 10 zwei mögliche Routen zum Endpunkt 16 führen.

Wie bereits im Punkt 2.1.3 beschrieben können mit diesem Programm ebenso Verkehrsstaus simuliert werden. Falls beispielsweise in Abb. 1.B Fahrzeuge vom Startpunkt 10 zum Endpunkt wollen, und zwischen den Verkehrsknotenpunkten 1 und 2 ein Stau auftritt, dann werden die Fahrzeuge über die Route 10-1-5-6-2-11 umgeleitet.

1.1.5 Fluide Elemente

Die wichtigste Eigenschaft der fluiden Simulationselemente ist es, dass sie Parameter in sich tragen, welche ihren Zustand beschreiben.

1.1.6 Iterativer Prozess

Eine Verkehrssimulation ist ein iterativer Prozess. Jeder Simulationsschritt löst Implikationen aus, welche wiederum Grundlage für den nächsten Simulationsschritt sind.

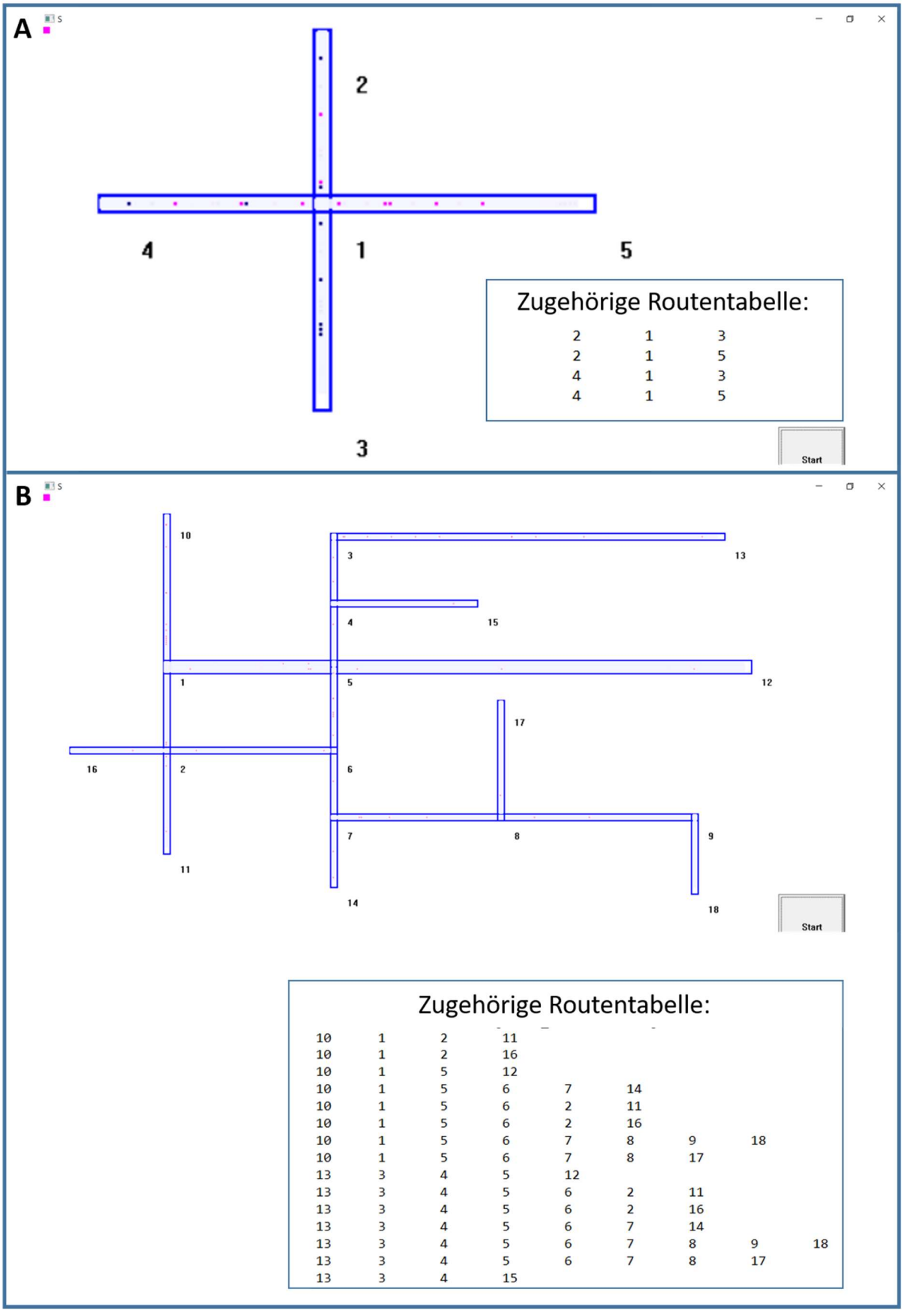


Abbildung 1: **Beispiele für Verkehrssimulationen. A) Einfache Simulation.** Abgebildet sind zwei Fahrbahnen mit jeweils einem Fahrstreifen welche sich kreuzen. Nummern kennzeichnen Start- (Nr. 2,4), End- (Nr. 3,5) und den Knotenpunkt (Nr. 1). Punkte in den Fahrbahnen symbolisieren Fahrzeuge. Rosa Fahrzeuge bewegen sich Richtung Endpunkt Nr. 5, blaue Fahrzeuge in Richtung Nr. 3. **B) Komplexere Simulation.** Von Nr.1 – Nr. 12: Fahrbahn mit 2 Fahrstreifen.

2 Der Verkehrssimulator - Die erstellte Applikation

2.1 Grundmodi bei der Applikation

Ziel dieses Programms ist die Schaffung eines Verkehrssimulators. Der Ablauf in der geschaffenen Applikation gliedert sich in zwei Grundmodi, welche nacheinander einmalig ausgeführt werden.

- Erstellung des „Simulationsnetzwerks“ durch Userinteraktion (Grundmodus 1)
- Es folgt ein Transformationsschritt von Grundmodus 1 zu Grundmodus 2.
- Ausführung der Simulation (Grundmodus 2)

Diese Einteilung des Programmablaufs in zwei Grundmodi ist für die Beschreibung des Codes von großer Bedeutung.

2.2 Programmbeschreibung aus Sicht der Informatik

2.2.1 Grundsätzliches

- Das Programm wurde mit der Programmiersprache C++ erstellt.
- Als Entwicklungsumgebung für die Programmierung wurde Microsoft Visual Studio gewählt. Weiters wurde der C++-Compiler von Microsoft Visual Studio verwendet.

- Es handelt sich hierbei um eine Windows- Desktop -Applikation. Das aus dem Programm hervorgegangene Programm-File (.exe) ist daher nur auf Geräten mit dem Betriebssystem „Windows“ lauffähig.

- Wichtiges Feature im Programmcode ist Verwendung der Betriebssystemschnittstelle „WinAPI“

2.2.2 WinAPI:

Einbettung der WinAPI ins Programm:

Jene „.cpp-Datei“ des Programms, in welcher die „WinMain“-Funktion deklariert ist, trägt den Namen „main.cpp“. *Diese erste und wichtigste Datei des Proramms wird in dieser Codebeschreibung fortan an „main-Datei“ bezeichnet*.

Die Verknüpfung zwischen der selbst geschaffenen Applikation und dem „Windows Application Programming Interface“ (kurz: WinAPI) ist durch die Include-Datei „<windows.h>“ gegeben. Diese „Include-Datei“ stellt sämtliche Deklarationen der WinAPI-Funktionen bereit.

Beginn des Kontrollflusses des Programms:

Im Gegensatz zu einer klassischen Konsolen-Anwendung ist der Startpunkt des Kontrollflusses der Anwendung, nicht durch die Funktionsdeklaration „main()“ repräsentiert, sondern durch das Kennwort „WinMain“.

Code, welcher den Beginn des Programmes repräsentiert:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, PSTR lpCmdLine, INT nCmdShow){

    return 0;}
```

2.3 Aktivitätsdiagramm der Applikation

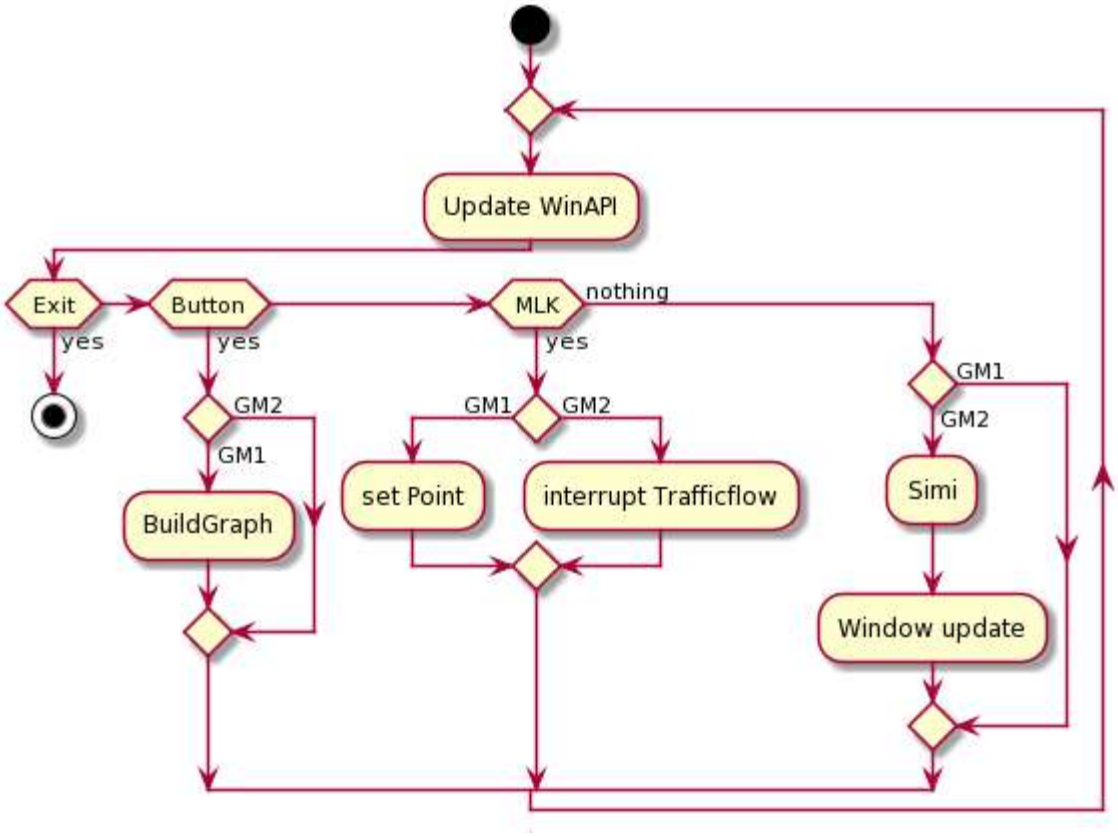


Abbildung 2: Aktivitätsdiagramm der Applikation

2.3.1 Beschreibung des Inhalts des Aktivitätsdiagramms:

Zentraler Bestandteil der Hauptfunktion(WinMain) ist eine Schleife(„Message-Loop“), welche über nahezu die gesamte Laufzeit der Anwendung ausgeführt wird.

Die erste Aktivität in dieser Schleife ist der Aktionsknoten „Update WinAPI“.

Der Zweck dieser Aktivität besteht darin Meldungen auszulesen, welche vom Betriebssystem Windows über die Schnittstelle übergeben werden. Im Aktivitätsdiagramm wird diese Informationsübertragung in Form eines „Objektknotens“ (Titel: „OS Events“) dargestellt.

Danach folgen im Aktivitätsdiagramm drei Verzweigungen, deren Entscheidungsgrundlage im Inhalt des Objektknotens (d.h. bei den OS Events) liegen. Entspricht der Inhalt des Objektknotens „OS Events“ den Bedingungen der jeweiligen Verzweigungen, so wird der Anweisung folgend der jeweilige Pfad eingeschlagen.

Verzweigung „Exit“:

Wird vom User während des Betriebs der Applikation das „rote Kreuzchen“ an der rechten oberen Ecke des Applikationsfenster geklickt, so wird vom Betriebssystem ein „Schließbefehl“ generiert. Dieser „Schließbefehl“ wird in Gestalt des Objektknotens –„OS Events“ der Verzweigung „Exit“ zugeführt, was die Erfüllung der Bedingung bewirkt und im Zuge der Ausführung der Anweisung in der Schließung der Applikation mündet.

Das Schließen der Applikation ist im Aktivitätsdiagramm durch den Endknoten dargestellt.

Verzweigungen „Button“ sowie „linker Mausklick“:

Anhand der Verzweigungen „Button“ und „MK_L“ wird deutlich, dass diese Applikation im Wesentlichen über einen Button sowie über die linke Taste der Maus zu steuern ist. Je nach entsprechendem Grundmodus werden beim Betätigen des Bottons sowie der linken Maustaste unterschiedliche Aktivitäten ausgeführt.

Durch das Klicken der linken Maustaste wird im Grundmodus 1 eine Aktivität ausgeführt, welche zur Erstellung der Netzwerkgraphen dient. In Grundmodus 2 wird durch das Klicken der linken Maustaste eine Aktivität ausgeführt mit welcher auf das Simulationsgeschehen zugegriffen werden kann.

Im Grundmodus 1 wird durch das Klicken des Buttons die Aktivität der „Graphen Generierung“ ausgeführt, sofern bereits Daten dafür generiert wurden. Im Grundmodus 2 wird durch das Klicken des Buttons keine Aktivität ausgelöst.

Pfad „Default Case“:

Entspricht der Inhalt des Objektknotens „OS Events“ keiner Bedingung der implementierten Verzweigungen, so wird ein Standardpfad („Default Case“) einschlagen, welcher im Grundmodus 1 ohne Aktivität abläuft. Im Grundmodus 2 wird ein Simulationsschritt ausgeführt, welcher als Aktionsknoten „Simi“ im Aktivitätsdiagramm dargestellt ist.

Ende der Schleife im Aktivitätsdiagramm:

Schlussendlich führen alle Pfade, ausgenommen jener Pfad, welcher das Programm beendet, wieder zurück zum Beginn der Schleife.

2.4 Codemodule

2.4.1 Definition des Begriffs „Modul“

Der Code des gesamten Programmierprojektes ist modular aufgebaut. Als Modul wird in diesem Programm eine Einheit betrachtet, welche einen genau definierten Zweck erfüllt.

- Dieser Zweck kann in einer bestimmen auszuführenden Handlung bestehen.
- Die Beschreibung des Zwecks eines Moduls kann aber auch globaler gefasst sein, indem beispielsweise Koordinationsaufgaben zwischen Modulen erfüllt werden.

2.4.2 Kapselung

Einzelne Module des Codes können gegebenenfalls verändert werden, ohne dass dies unmittelbare Auswirkungen auf andere Module hätte. Die jeweiligen Module sind so konzipiert, dass diese jeweils gekapselt gegenüber anderen Modulen sind. Das bedeutet, dass abseits von exakt definierten Schnittstellen zwischen den Modulen keine Abhängigkeiten bestehen.

2.4.3 Ausgestaltung der Module

- Module können entweder aus einer einzelnen Methode einer Klasse bestehen.
- Module können aber auch aus einer Vielzahl an Klassen bestehen.

Größere Module, welche aus mehreren Klassen bestehen, sind größtenteils nach Vorbild von so genannten „Software-Entwurfsmustern“ erstellt. Je nach Verwendungszweck und passender Möglichkeit wurden adäquate Entwurfsmuster ausgewählt.

Grundsätzlich wird mit dem Terminus „Software-Entwurfsmuster“ eine exakt definierte Anordnung von Objekten beschrieben. Durch die Übernahme dieser Strukturen ist es möglich klar und unmissverständlich zu kommunizieren, wie die betreffenden Module der Software aufgebaut sind.

2.4.4 Aufzählung der Module des Programms

Aus der Spezifikation des Programms ergeben sich folgende Module:

- Modul zur Verarbeitung von „Eingabeevents“ (User Interaktion)
    - Implementation der „WinAPI“ in der Hauptfunktion (WinMain)
- Modul zur Erstellung der „Netzwerk-Grundlage“
    - Implementiert als Methode in der Klasse „Network“
- Modul zu Erstellung der Simulationsgrundlage
    - Implementiert als Methode in der Klasse „Network“
- Modul zur Erzeugung von Zufallszahlen
    - Implementiert in der Klasse „Random Speed New“, Instanz der Klasse „Network“
- Modul zur Ausführung der Simulation
    - u.a. durch Implementierung des Software-Entwurfsmuster „Visitor“
- Modul zur Einflussnahme auf die Simulation (während der Ausführung)
    - u.a. durch Implementierung des Software-Entwurfsmuster „Observer“
- Modul zur Ausgabe von Simulationsergebnissen
    - Implementiert in der Klasse „PrintInGDIWindow“
- Modul „Pool Allocator“
    - Implementiert in der Klasse „Pool Allocator“

2.5 Gesamte Klassenstruktur der Applikation

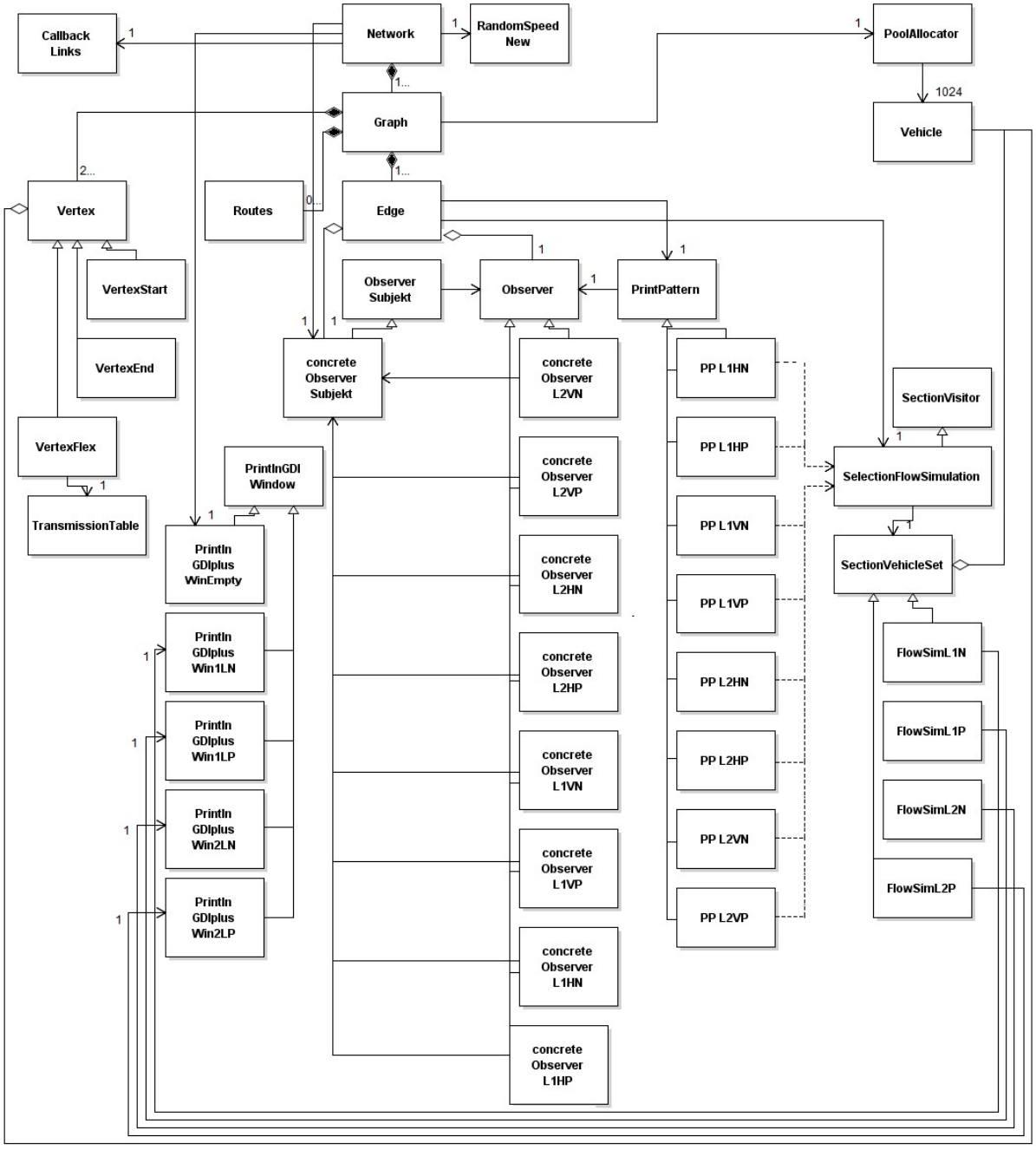


Abbildung 3: Vollständiges Klassendiagramm der Applikation.

2.6 Beschreibung der Klassenstruktur im Detail

2.6.1 Klasse „Network“

Die Klasse „Network“ repräsentiert die Pforte zu einem Komplex aus Klassen, welche sämtliche Aufgaben des „Verkehrssimulators“ ausführen.

Um die einzigartige Bedeutung der Klasse „Network“ hervorzuheben ist anzumerken, dass die Klasse „Network“ als Singleton implementiert. Das bedeutet, dass in der gesamten Anwendung die Klasse „Network“ nur ein einziges Mal instanziiert werden kann.

Abbildung 4: Klasse „Network“

2.6.1.1 Prominente Methoden der Klasse „Network“

Die Klasse „Network“ erfüllt zwei wesentliche Aufgaben:

Die Methode “network::establishLane“ dient nach Start der Anwendung in Grundmodus 1 dazu, dass aus aufeinanderfolgender Mausklickevents Punktpaare erfasst werden, welche einen Anfangspunkt und einen Endpunkt eines Fahrstreifens darstellen. Diese Punktpaare bilden die Grundlage für eine Netzwerkstruktur in welcher die Simulation in Grundmodus 2 ausgeführt werden soll.

Die Methode „network::establishVertexOfGraph“ erzeugt aus den generierten Punktpaaren gerichtete Graphen, welche aus Ecken(Vertex) und Kanten(Edges) bestehen. Sowohl Graphen, Ecken und Kanten sind in diesem Programm als Klassen implementiert, welche dynamisch erzeugt werden und dessen Zeiger(Pointer) in Vektoren abgelegt werden und über diese angesprochen werden können.

2.6.1.1.1 Exakte Beschreibung des Netzwerkserzeugungsalgorithmus in der Methode „network::establishVertexOfGraph“

- Die Erzeugung der Netzwerkstruktur wird dadurch bewerkstelligt, indem in einem ersten Schritt aus den gewonnenen Punktpaaren aus Grundmodus 1 Objekte der Klassen „Vertex“ (Knoten) und „Edge“ (Kanten) dynamisch erzeugt werden.
- In einem zweiten Schritt werden, abhängig von der örtlichen Lage der Objekte „Vertex“ und „Edge“ ein oder mehrere Objekte der Klasse „Graph“ dynamisch erzeugt.
- In einem dritten Schritt erfolgt für jedes Objekt der Klasse „Graph“ die Erzeugung der Objekte der Klasse „Route“.
- Nach Abschluss dieses Vorganges, wo alle notwendigen Objekte der Klassen „Vertex“, „Edge“, „Graph“ und „Route“ erzeugt wurden, ist der „Infrastrukturbildungsprozess“ abgeschlossen.

2.6.2 Klasse „callBackLinks“ sowie „randomSpeedNew“

Abbildung 5: Klasse „Network“ mit zwei weiteren Klassen. Die Klassen „callbackLinks“ und „randomSpeedNew“ sind in der Klasse „network“ instanziiert.

2.6.2.1 Funktionszeiger auf globale Funktionen der „Main-Datei“

Die Klasse „callbackLinks“ enthält eine Reihe an „function pointer“ (Funktionszeiger), welche auf globale Funktionen in der „main-Datei“ zeigen. Diese globale Funktionen der „main-Datei“ bzw. dessen Funktionszeiger in dieser Klasse erfüllen wichtige Aufgaben im Programm. Sämtliche Ergebnisse werden mit Hilfe dieser „Callback-Funktionen“ im Applikationsfenster grafisch dargestellt.

2.6.2.2 Zufallsgenerator

Die Klasse „randomSpeedNew“ beinhaltet einen Zufallsgenerator, welcher Werte zwischen 50 und 120 erzeugt. Diese Werte repräsentieren im Betrieb des Verkehrssimulators Geschwindigkeiten, welche jedem Fahrzeug während des Simulationsablaufs zugewiesen werden.

Besonders zu erwähnen ist, dass die Erzeugung der Zufallswerte nicht im Hauptthread der Anwendung stattfindet, sondern in einem anderen Thread. Immer wenn der Bedarf nach neuen Zufallszahlen im Programmablauf gegeben ist, wird ein Container mit Zufallszahlen im Nebenthread befüllt und nach Abschluss der Aufgabe der Anwendung im Hauptthread zur Verfügung gestellt. (Implementation eines „Futures“. „std::future“)

Insbesondere der Multithreadingansatz bei der Erzeugung von Zufallszahlen ließ die Erwägung reifen, diese Klasse nur ein einziges Mal im zentralen Bereich der Klassenhierarchie zu instanziieren.

2.6.3 Modul: Netzwerkstrukturen im Programm des Verkehrssimulators

Die Klasse „Network“ besitzt als Attribut ein oder mehrere Objekte der Klasse „Graph“. Diese wiederum besitzt ihrerseits als Eigenschaft ein oder mehrere Objekte der Klasse „Edge“, sowie zwei oder mehrere Objekte der Klasse „Vertex“ und darüber hinaus null oder mehrere Objekte der Klasse „Route“.

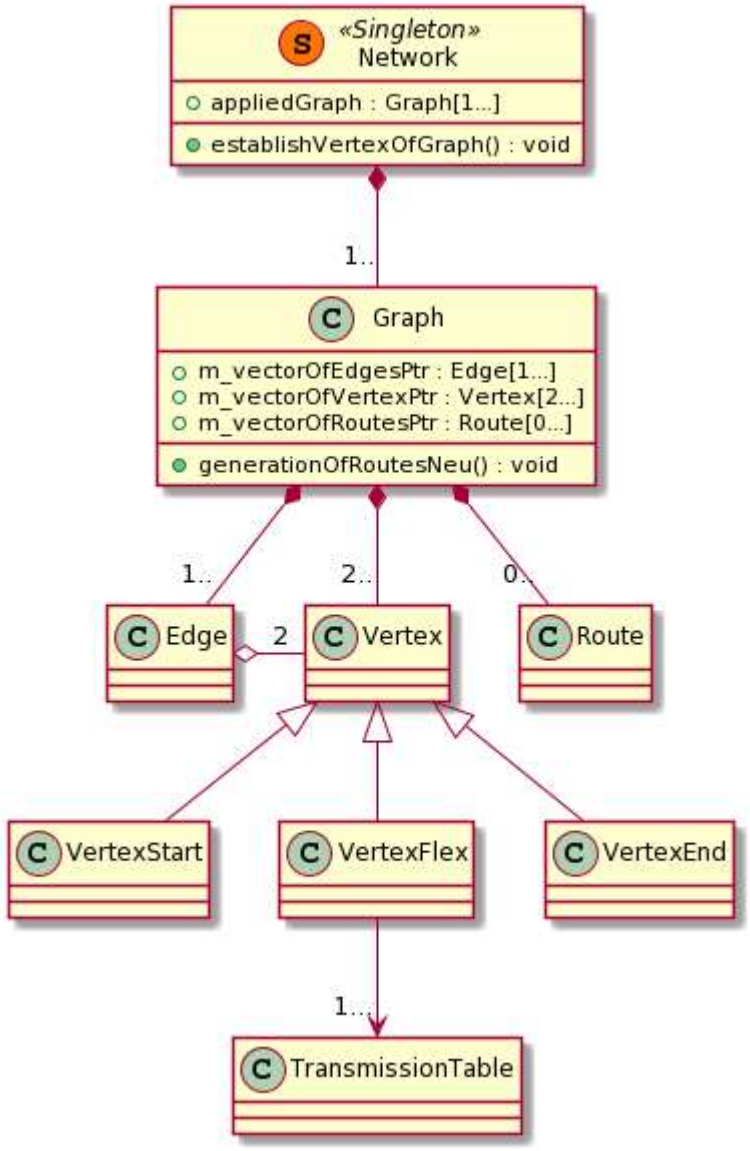


Abbildung 6: Auszug aus dem Klassendiagramm, Modul: Simulationsgrundlage

Bei den Beziehungen zwischen den Klassen „Network“ und „Graph“ sowie zwischen den weiteren genannten Klassen, welche in Containern in der Klasse „Graph“ instanziiert sind, handelt es sich um „Teil-Ganzes-Beziehungen“.

Zusammengefasst kann gesagt werden, dass ein Netzwerk aus einem oder mehreren (Teil-)Graphen besteht, welche wiederum aus Knoten und Kanten gebildet werden durch welche Pfade d.h. Routen verlaufen können. Im Sinne der UML-Modellierung im Klassendiagramm handelt es sich bei diesen „Teil-Ganzes-Beziehungen“ jeweils um Kompositionen, welche durch einen voll ausgefüllten Diamanten dargestellt werden.

Erläuterungen:

- Im Zusammenhang mit den „Teil-Ganzes-Beziehungen " ist es hier wichtig festzustellen, dass es sich um „Kompositionen“ handelt und nicht um „Aggregationen“. Denn die Objekte der Klassen „Edge“, „Vertex“ und „Route“ können nur unter einer hierarchisch übergeordneten Klasse „Graph“ instanziiert sein. Dasselbe gilt für die Klasse „Graph“ gegenüber der Klasse „Network“.
- Die Rolle des „Graphen“ ist in diesem Programm so definiert, dass ein Objekt der Klasse „Graph“ eine Menge an Kanten (Eges) exklusiv besitzt. Kein Objekt der Klasse „Edge“ darf gleichzeitig Instanz von verschiedenen Instanzen der Klasse „Graph“ sein.
- Eine Kante (Edge) ist ein „gerichtetes“ Teilstück (Link) eines Graphen. Jede Kante besitzt daher einen definierten Start- und Endpunkt. Die beiden Punkte sind als Objekte der Klasse „Vertex“ (Knoten) repräsentiert und sind Attribute in der Klasse „Edge“. Gleichzeitig sind die beiden Objekte der Klasse „Vertex“, ebenfalls in der Klasse „Graph“ instanziiert. Diese gleichzeitige Zugriffsmöglichkeit wird durch so genannte „Shared Pointer“ realisiert, indem ein und dieselbe Ressource über Referenzen an verschiedenen Stellen zugänglich gemacht wird. (Ferner kann durch den Einsatz von „Shared Pointer“ Problematiken mit unterschiedlichen Laufzeiten unterbunden werden. Erst wenn die letzte Instanz den Zugriff auf eine Ressource freigibt, wird dessen dafür verwendeter Speicher wieder freigeben.)
- Die Knoten im Netzwerk können wiederum unterschiedliche Zwecke erfüllen. Entweder sie sind Startknoten(„VertexStart“) im Netzwerk, Endknoten(„VertexEnd“) oder Zwischenknoten(„VertexFlex“).

Zuständige Methoden im Programm:

- Zur Erzeugung von Objekten der Klassen „Graph“, „Edge“ und „Vertex“(inklusive erbender Klassen von „Vertex“) wird die Methode „Network::establishVertexOfGraph()“ ausgeführt.
- Die Objekte der Klasse „Route“ werden durch die Methode „Graph::generationOfRoutesNeu()“ erzeugt. Diese Methode beinhaltet einen „Tiefensuche“-Algorithmus, welcher prinzipiell jeden Pfad durch den Graphen findet.

Eckpunkte der Routenfindung im Simulationsnetzwerk:

  - Der „Tiefensuche“-Algorithmus von jeden Startknoten („VertexStart“) ausgeführt.
  - Von jedem Startknoten durchläuft der Algorithmus sämtliche „Baumknoten“ („VertexFlex“) bis ein Endknoten („VertexEnd“) erreicht ist. Nach Erreichen eines Endknotens wird ein Objekt der Klasse „Route“ erzeugt, in welchem die Information über den gefundenen Pfad im Simulationsnetzwerk abgelegt ist.
  - Nach Erreichen eines Endknotens („VertexEnd“) wird jeweils um einen Baumknoten zurückgesetzt um andere Teilbäume des Simulationsnetzwerkes zu besuchen. Sofern Endknoten erreicht werden, werden auch hierbei Objekte der Klasse „Route“ erzeugt.
  - Kanten, welche von einem Teilbaum zu einem anderen Teilbaum führen (d.h. so genannte „Querkanten“) werden ebenfalls vom implementierten Algorithmus durchlaufen, weil alle Routen im Netzwerk erfasst werden sollten.
  - Bei Kanten, welche vom aktuell besuchten Baumknoten zu einem bereits besuchten Baumknoten zurückführen (d.h. so genannte Rückwärtskanten) werden detektiert. Der Suchvorgang wird im Folgeschritt für den Teilbaum abgebrochen und bei einem davorliegenden Baumknoten fortgesetzt.

2.6.4 Modul: „PoolAllocator“ (Klassen „PoolAllocator bzw. „Vehicle“)

Neben den Netzwerksstrukturen mit den Klassen „Graph“, „Edge“, „Vertex“ und „Route“ sind die Fahrzeuge, die fluiden Elemtente der Simulation, jene Bestandteile die das Simulationsprogramm kompletieren. Die Fahrzeuge werden durch Objekte der Klasse „Vehicle“ repräsentiert. Aus der Natur der fluiden Elemete der Simulation ist zu schließen, dass diese während der Simulation (1)fortwährend in Erscheinung treten, (2)sich fortbewegen und zuletzt (3) auch wieder aus dem Geschehen verschwinden. Dieses naturgemäße Verhalten könnte man in einer Weise implementieren, dass für die Objekte der Klasse „Vehicle“ fortwährend Speicher alluziert bzw. wieder freigegeben werden muss. Um ein häufiges Alluzieren von Speicher, welches im Falle vom Haldenspeicher (Heap) auch „teuer“ (aufwendig) ist, abzuwenden, wurde in diesem Programm ein anderer Ansatz gewählt: Zur Bereitstellung der Objekte der Klasse „Vehicle“ wird vor Siumlationsbeginn von einem „Pool-Allocator“ eine große jedoch limitierte Anzahl an Objekten generiert.

Eigenschaften des Pollallocators:

- Das Design ist so gestaltet, dass pro instanzierten Objekt der Klasse „Graph“ im Programm ein Poolallocator mit 1024 Objekten der Klasse „Vehicle“ angelegt wird.
- Die “Placement new Syntax” ist das zentrale Feature des Pool-Allocators. Sobald ein „Vehicle“-Objekt vom Simulationsnetzwerk entfernt wird, steht es unmittelbar für den erneuten Gebrauch zur Verfügung.

Design des Simulation:

- Die Simlation wird in der Art und Weise ausgeführt, indem mittels Zeiger(Pointer) auf die involvierten Fahrzeuge d.h. auf Objekte der Klasse „Vehicle“ zugegriffen wird*. Die exakte Beschreibung des Simulationsvorgangs folgt in Kapitel 3.6.5.2.*
    - Die „Vehicle“-Objekte verändern während der gesamten Simulation nie ihren angestammten Ort im Arbeitsspeicher, in welchem sie instanziiert wurden.
    - Es gibt keine Kopiervorgänge von „Vehicle“-Objekten während der gesamten Laufzeit der Anwendung.

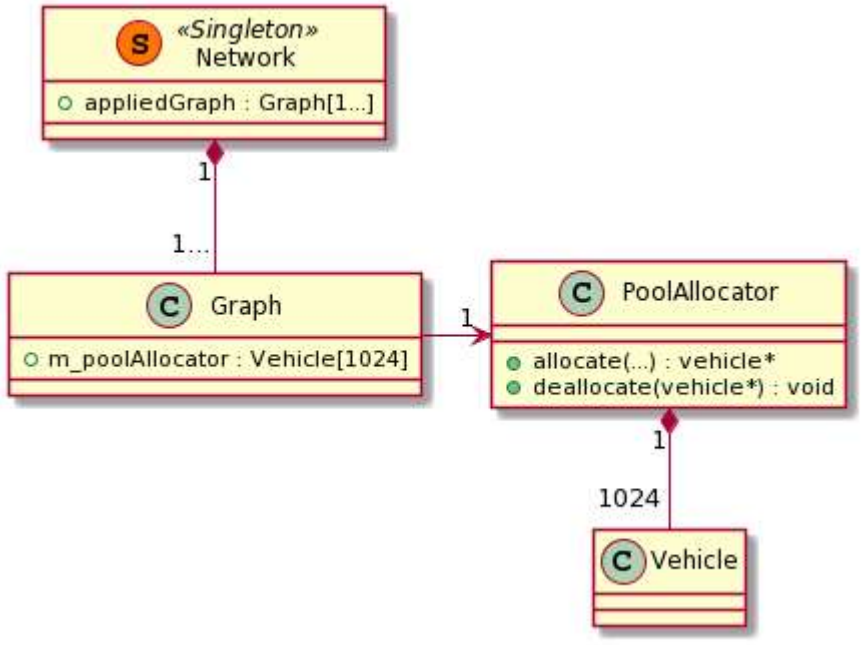


Abbildung 7: Auszug aus dem Klassendiagramm, Modul: „Pool Allocator“.

2.6.5 Modul: Ausführung der Simulation

2.6.5.1 Klassendiagramm jener Klassen, welche am Simulationsvorgang beteiligt sind

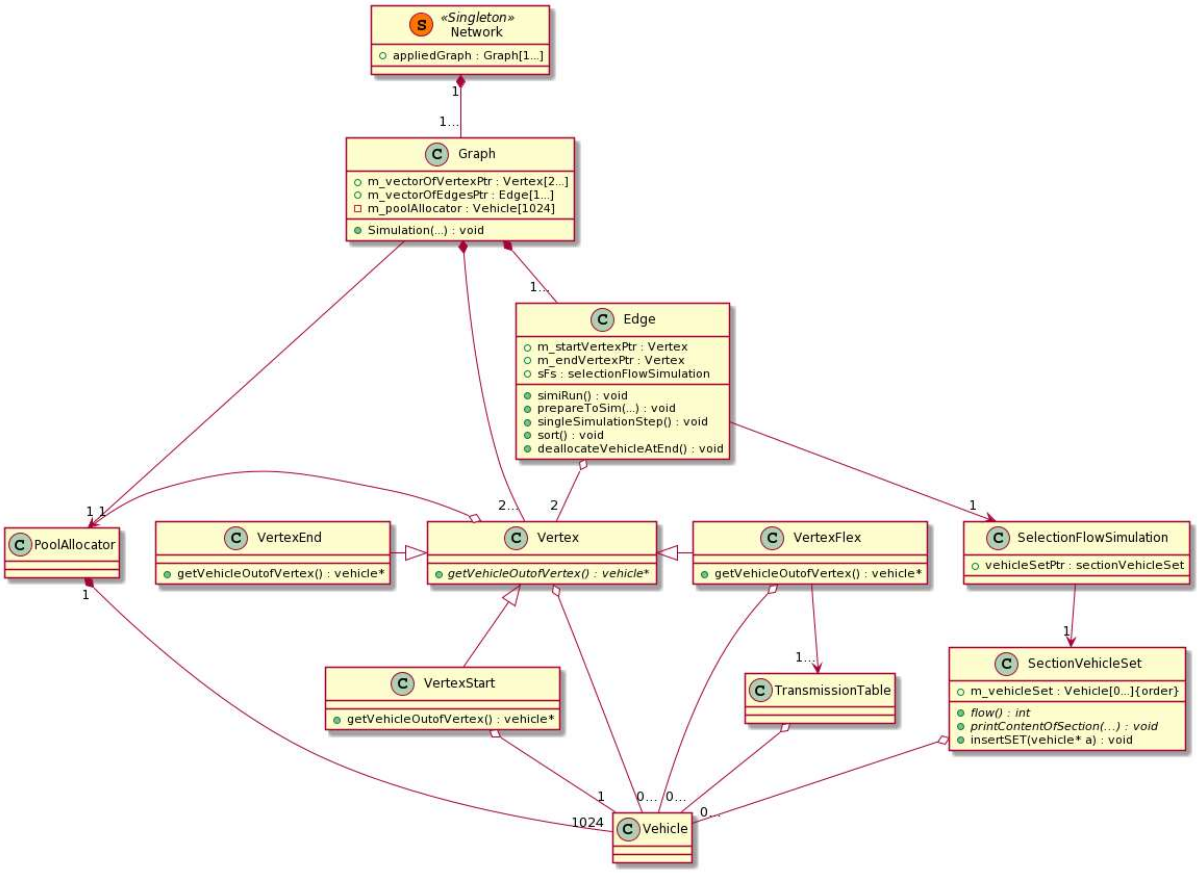


Abbildung 8: Auszugsweises Klassendiagramm mit jenen Klassen, welche beim Simulationsprozess aktiv beteiligt sind. Sämtliche an der Simulation involvierten Eigenschaften und Methoden sind im Klassendiagramm dargestellt.

2.6.5.2 *Die algorithmenbasierte Interaktion der Fahrzeuge auf den Fahrsteifen*

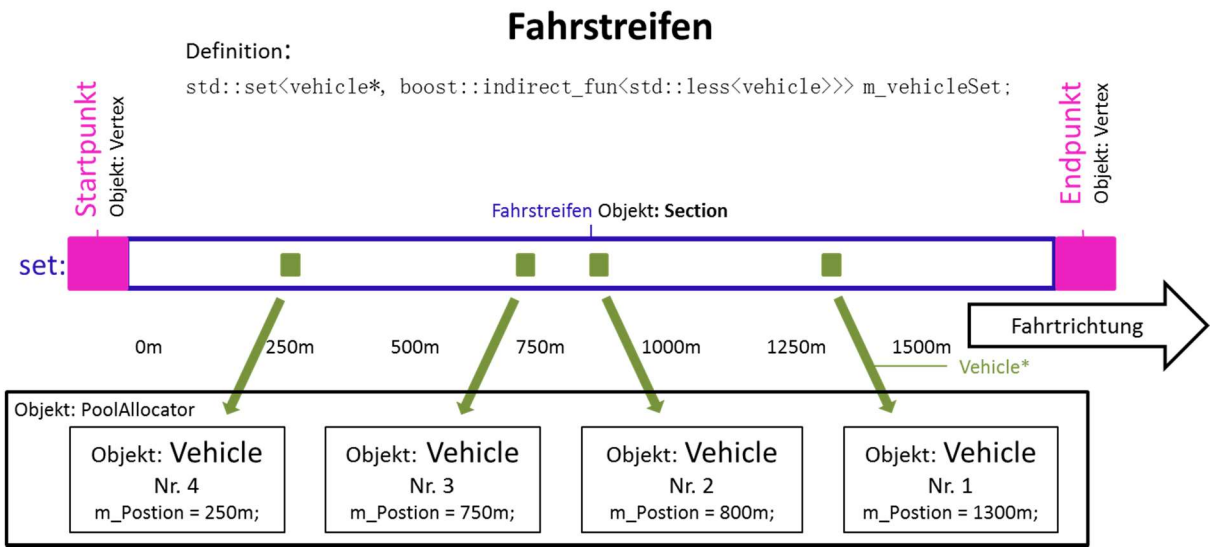


Abbildung 9: Simulationsschritt in der Klasse „Edge“ bzw. in der Klasse „Section“. Der Verkehrsflusssimulationsalgorithmus verändert während eines Simulationsschritts den Wert der Eigenschaft m_Postion der „Vehicle“-Objekte. Dadurch entsteht die Imagination der bewegten Fahrzeuge (dargestellt durch grüne Punkte), welche sich im Fahrsteigen voran bewegen.

Die Simulation des Verkehrsgeschehens auf einer Richtungsfahrbahn (Fahrsteifen) ist im blauen Rahmen der Abb. 9 dargestellt und gestaltet sich folgendermaßen:

- Der Fahrstreifen, welcher als blauer Rahmen dargestellt ist, entspricht der Datenstruktur eines „Sets“. Diese Datenstruktur ist Eigenschaft der Klasse „SectionVehicleSet“, welche wiederum Attribut der Klasse „Edge“ ist.
- Ausschließlich „Pointer“ der Objekte der Klasse „Vehicle“ werden in den Objekten der Klasse „SectionVehicleSet“ gespeichert.
- Der Simulationsalgorithmus „flow(…)“ ist eine abstrakte Methode der Hauptklasse „SectionVehicleSet“.

  Die Simulation wird so ausgeführt, dass per Zeiger auf die Parameter (Eigenschaften) „Vehicle“-Objekte zugriffen wird. So werden die Parameter der „Vehicle“-Objekte vom Simulationsalgorithmus im Sinne der Simulation verändert.

- In den Kindklassen der Klasse „SectionVehicleSet“ sind die konkreten Implementierungen der unterschiedlichen Simulationsalgorithmen erstellt.
- Wie die Auswahl erfolgt, welcher Verkehrsflusssimulationsalgorithmus zu Anwendung kommt wird im Kapitel 3.6.5.6 beschrieben.

2.6.5.3 Einbettung des Simulationsalgorithmus ins Programm

Vor bzw. nach der eigentlichen Ausführung des Simulationsalgorithmus müssen einige Prozeduren vollzogen werden. Dieses Kapitel erklärt wie ein Simulationsschritt erfolgreich durchgeführt wird.

2.6.5.3.1 Sequenzdiagramm für einen Simulationsschritt

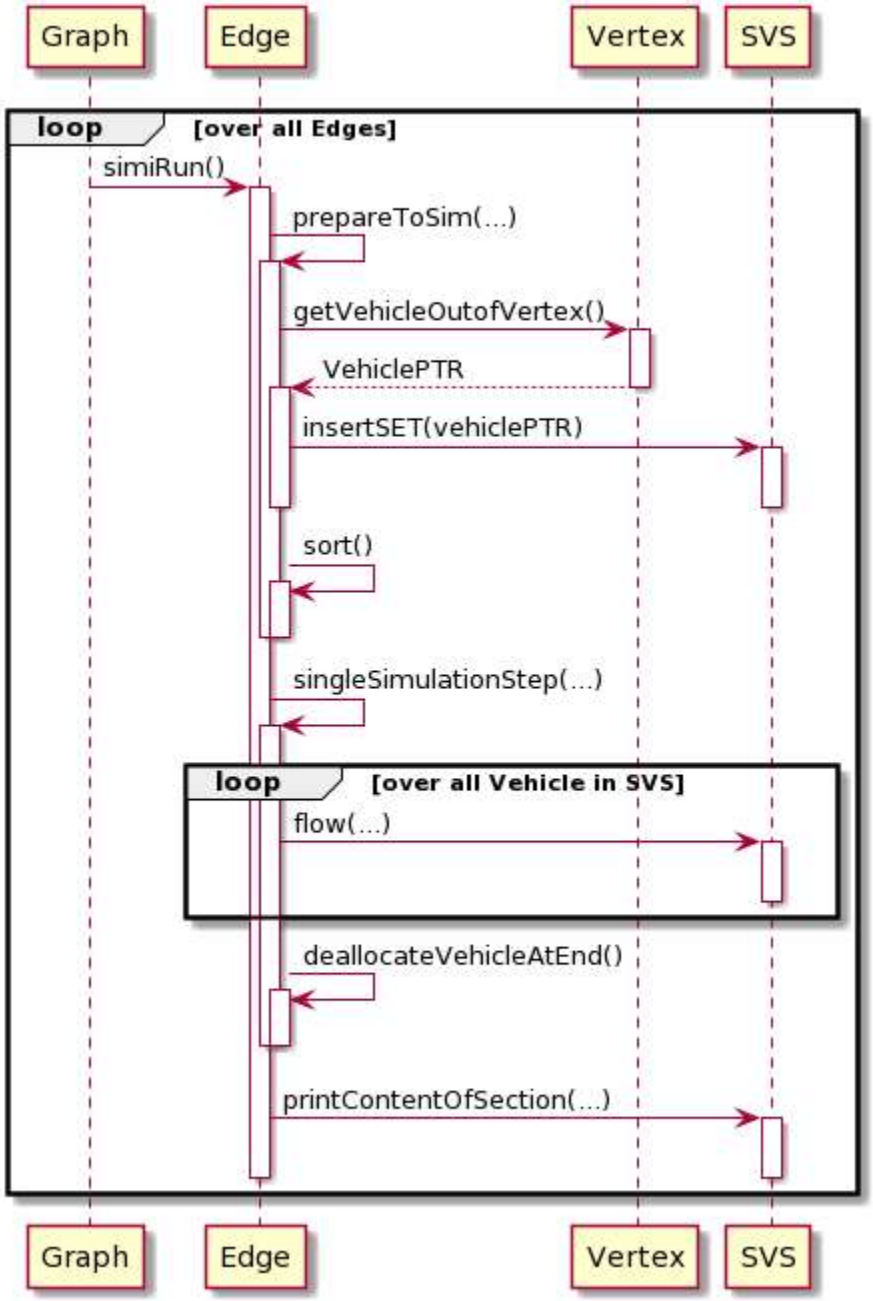


Abbildung 10: Sequenzdiagramm. Der Simulationsbetrieb erfolgt wie im Sequenzdiagramm dargestellt.

2.6.5.3.2 Verbale Beschreibung der Simulationsvorganges

Aus dem Aktivitätsdiagramm (Abb. 2) der Applikation geht hervor, dass sofern in der " Message-Loop" der „default-Pfad" ausgeführt wird ein Simulationsschritt getätigt wird. Das bedeutet konkret, dass aus der „main-Datei" heraus eine Schleife in Gang gesetzt wird, welche über alle Objekte der Klasse „Graph" iteriert. In den Objekten der Klasse „Graph" wird dabei die Methode Graph::Simulation(…) ausgeführt, welche ihrerseits darüber verfügt, dass über sämtliche Objekte der Klasse „Edge" iteriert wird um einen dreigliedrigen Vorgang auszulösen, welcher wie folgt beschrieben ist:

Der Simulationsvorgang (edge::simiRun())erstreckt sich über drei Teile.

- Im ersten Teil wird im Beginnknoten Nachschau gehalten, ob ein Fahrzeug-Objekt (Vehicle) bereit steht, welches in die Kante eingesetzt werden soll, um Teil der Simulation zu werden(Methode: getVehicleOutofVertex()). Darüber hinaus werden die „Fahrzeug-Objekte" (Klasse: „Vehicle") für die Simulation vorbereitet, indem beispielsweise gewisse Flags in den „Vehicle"-Objekten gesetzt werden, welche für den Simulationsalgorithmus wesentlich sind.
- Der eigentliche Simulationsprozess „sectionVehicleSet::flow(…)" findet in einer Schleife statt, wo über alle in der Kante befindlichen „Vehicle-Objekten" iteriert wird. In diesem Schritt wird jeweils die Position eines Fahrzeuges gegenüber anderen Fahrzeugen im Fahrsteifen (d.h. Kante/Edge) verändert. Bei mehrstreifigen Kanten/Fahrbahnen kann hier auch ein Fahrstreifenwesel erwirkt werden.
- Nach der eigentlichen Simulation werden jene „Vehicle"-Objekte, welche das Ende der Kante erreicht haben, aus dieser (d.h. aus dem Fahrstreifen) entnommen und dem Objekt des Endknotens (Klasse Vertex) übergeben. Der Endkonten der Kante bearbeitet die „Vehicle"-Objekte weiter.
- Als letzter Schritt des Simulationsprozesses folgt die Darstellung des Simulationsergebnisses gegenüber dem User (printContentOfSection(…)).

2.6.5.4 Vorwort zu den Klassen „PrintPattern“, „SectionVehicleSet“, „Observer“ sowie „PrintInGDIPlusWinEmpty“

In den vorhergehenden Kapiteln wurde erörtert wie das Simulationsnetzwerk aufgebaut ist bzw. nach welchem Konzept die Simulation durchgeführt wird. In den folgenden Kapiteln wird auf die Klassen „PrintPattern“, „SectionVehicleSet“, „Observer“ sowie „PrintInGDIPlusWinEmpty“ eingegangen.

Diese Klassen sind alle Hauptklassen (Oberklassen), welche mit den dazugehörenden erbenden Subklassen instanziiert werden müssen. Dies ist dadurch begründet, weil in den Hauptklassen sogenannte „pure virtual function“ enthalten sind. Das sind Funktionen, welche in den Hauptklassen über keine Implementierung verfügen und deshalb auf Implementierungen in den Subklassen zuzugreifen müssen.

Alle vier genannten Klassen bestimmen im Zusammenwirken mit ihren Subklassen **die Charakteristik der Darstellung**, sowie **die Charakteristik der Verkehrssimulation** der im Verkehrsnetzwerk enthaltenen Fahrsteifen (Richtungsfahrbahnen).

2.6.5.4.1 Charakteristik der Darstellung der Richtungsfahrbahn

Die Fahrbahnen verlaufen in der gegenwärtigen Ausführung des Programms entweder in waagrechter oder senkrechter Richtung. Des Weiteren gibt es Richtungsfahrbahnen mit einem oder zwei Fahrsteifen. Diese beiden Faktoren führen dazu, dass die simulierten Fahrzeuge aus optischen Gründen geringfügig anders dargestellt werden müssen.

Dabei handelt es sich um marginale Veränderungen, welche das Simulationsergebnis nicht prinzipiell verändern, sondern nur um wenige Bildpunkte im Ausgabefenster rücken. Dabei handelt es sich um Charakteristiken welche je nach Art der Richtungsfahrbahn jeweils unterschiedliche Korrekturen erfordern.

Im Kapitel 3.6.5.5.1 wird beschrieben wie ausgehend von der Instanziierung der Eigenschaft „PrintPattern“ mit passender Subklasse für jede Richtungsfahrbahn die adäquate Art und Weise der Darstellung bestimmt wird.

Je nach Art und Weise der Darstellung der Richtungsfahrbahn muss auch das Feature des „Eingriffs auf die Simulation per Mausklick“ anders implementiert werden. Je nach instanziierter Subklasse der Eigenschaft „PrintPattern“ müssen dafür unterschiedliche Subklassen der Klasse „Observer“ dafür instanziiert werden. *In Kapitel 3.6.7.2 wird dies näher beschrieben.*

2.6.5.4.2 Charakteristik der Verkehrssimulation

Folgende zwei Charakteristika bestimmen die Auswahl des Verkehrssimulationsalgorithmus:

1) Verfügt der Fahrsteifen über eine oder zwei Fahrspuren

   Die Verfügbarkeit von mehr als einen Fahrstreifen pro Richtungsfahrbahn erfordert einen erweiterten Verkehrssimulationsalgorithmus, welcher ebenfalls das Überholverhalten von Fahrzeugen in die Simulation mit einschließt.

2) Werden die Positionswechsel von simulierten Fahrzeugen in positiver oder in negativer Richtung in Relation zum Bezugskoordinatensystem vollzogen

   Prinzipiell sind die Algorithmen identisch aufgebaut, unabhängig davon ob ein simuliertes Fahrzeug die Position in ansteigender oder fallender Richtung in Relation zum Bezugskoordinatensystem vollzieht. Der Unterschild liegt lediglich bei den Vorzeichen der Rechenoperationen in welchen die Abstände zwischen den Fahrzeugen ermittelt werden.

Die beiden genannten Charakteristika lassen sich auch im Teilklassendiagramm bei der Oberklasse „SectionVehicleSet“ sowie bei dessen erbenden Unterklassen verifizieren:

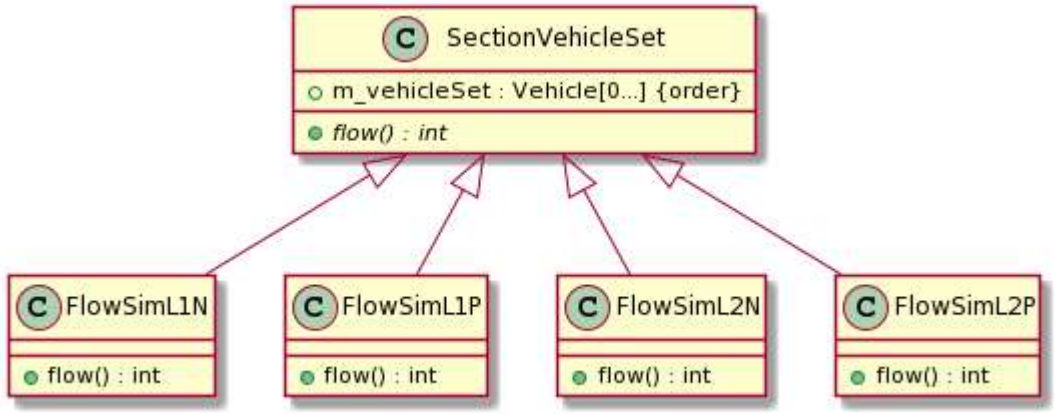


Abbildung 11: Auszug aus dem Klassendiagramm, Simulationsalgorithmen. Es wurden insgesamt vier verschiedene Verkehrsflusssimulationsalgorithmen implementiert. Konkret handelt es sich hier um die Methode „flow()“ in den Klassen „FlowSimL1N“, „FlowSimL1P“, „FlowSimL2N“ sowie „FlowSimL2P“. Die genannten Klassen erben von der Klasse „SectionVehicleSet“, wo die Methode „flow()“ als abstrakte Funktion (d.h.“ pure virtual function“) definiert ist. Aus der Notation geht folgendes hervor: „FlowSim“ steht für „Verkehrsflusssimulation“, „L1“ bzw. „L2“ steht für einen oder zwei Fahrsteifen, „N“ oder „P“ steht für negative oder positive Richtung gegenüber dem Bezugskoordinatensystem.

Die Verkehrsflusssimulationsalgorithmen sind in den Unterklassen gekapselt. Je nach Bedarf können diese Verkehrsflusssimulationsalgorithmen unabhängig von der Oberklasse modifiziert werden. Ferner wäre es problemlos möglich, zu den bereits vier implementierten Algorithmen weitere Verkehrsflusssimulationsalgorithmen in weiteren Unterklassen von „SectionVehicleSet“ zu ergänzen.

Die hier angewandte Struktur, entspricht dem Softwareentwurfsmuster der **„Strategie“**.

2.6.5.5 Ausgestaltung der Auswahl der Subklassen bei der Instanziierung der Objekte der Hauptklassen „PrintPattern“, „SectionVehicleSet“, „Observer“ sowie „PrintInGDIPlusWinEmpty“

Im Übergangsschritt zwischen Grundmodus 1 zu Grundmodus 2 werden aus den paarweise abgelegten mittels Mausklicks generierten Positionsdaten die Objekte des Simulationsnetzwerks erzeugt. Im engeren Sinne zählen zu den Objekten des Simulationsnetzwerkes Objekte der Klassen „Graph“, „Vertex“ sowie „Edge“.

Die Erzeugung der Objekte der Klassen „Graph“, „Vertex“ sowie „Edge“ wird während der Ausführung der Methode „Network::establishVertexGraph()“ durchgeführt.

2.6.5.5.1 Schritt 1: Instanziierung eines Objekts für die Eigenschaft „PrintPattern“ für jeden Fahrstreifen

Die Klasse „PrintPattern“ dient zur **Entkopplung** zwischen folgenden Sachverhalten:

- den basalen Objekten des Simulationsnetzwerkes („Vertex“, „Edge“, etc.)
- den Objekten, welche zur Darstellung des Simulationsnetzwerks gegenüber dem User verantwortlich sind
- welche Verkehrsflussalgorithmen eingesetzt werden

Die Differenzierung hinsichtlich unterschiedlicher Charakteristiken erfolgt durch die Instanziierung der Objekte der Klasse „PrintPattern“ mit unterschiedlichen erbenden Subklassen.

Im Zuge der dynamischen Generierung der Objekte der Klasse „Edge“ werden unmittelbar unter Verwendung der Positionsdaten der Anfangs- und Endpunkte Objekte der Klasse „PrintPattern“ erzeugt. Durch die Methode „network::choosePrintPattern(…)“, welche aus dem Konstruktor der Objekte der Klasse „Edge“ aufgerufen wird, wird ermittelt mit welcher Subklasse Objekte der Klasse „PrintPattern“ dynamisch instanziiert werden. Im Sequenzdiagramm (Abbildung 12) ist der Instanziierungsprozess der Objekte der Klasse „PrintPattern“ graphisch dargestellt. Der Prozess wird abgeschlossen, indem ein Zeiger auf das erzeugte „PrintPattern“-Objekt dem Objekt der Klasse „Edge“ zugewiesen wird.

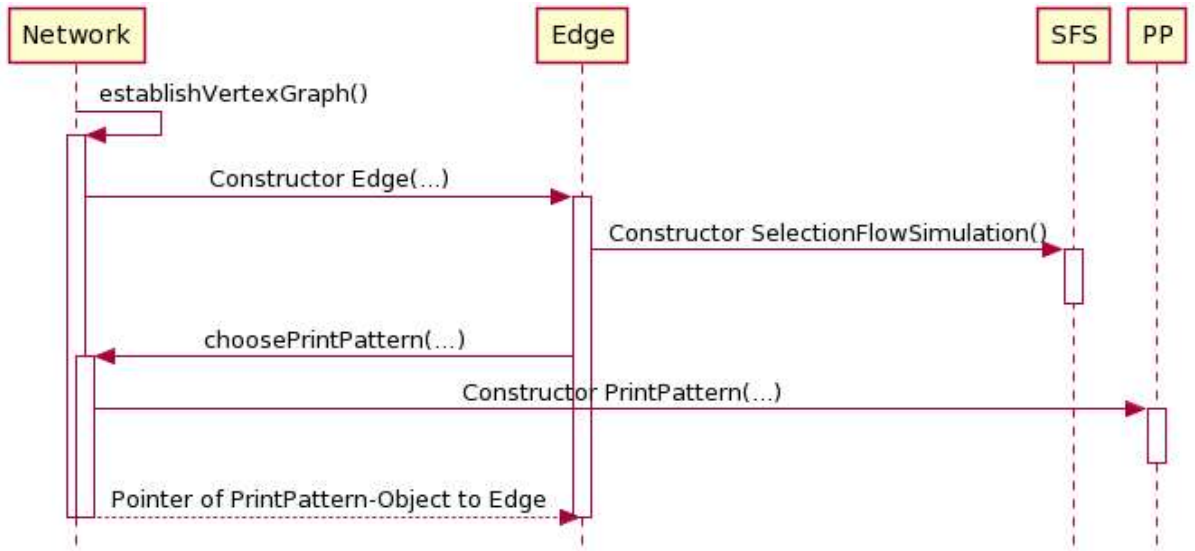


Abbildung 12: Sequenzdiagramm. Auswahlprozedur der passenden Subklasse von „PrintPattern“ bei der Erzeugung eines Objekts der Klasse „Edge“. (Glossar: PP = Klasse „PrintPattern“, SFS = Klasse „SectionFlowSelection“)

Nach Durchführung dieses Schrittes sieht das Klassendiagramm wie folgt aus:

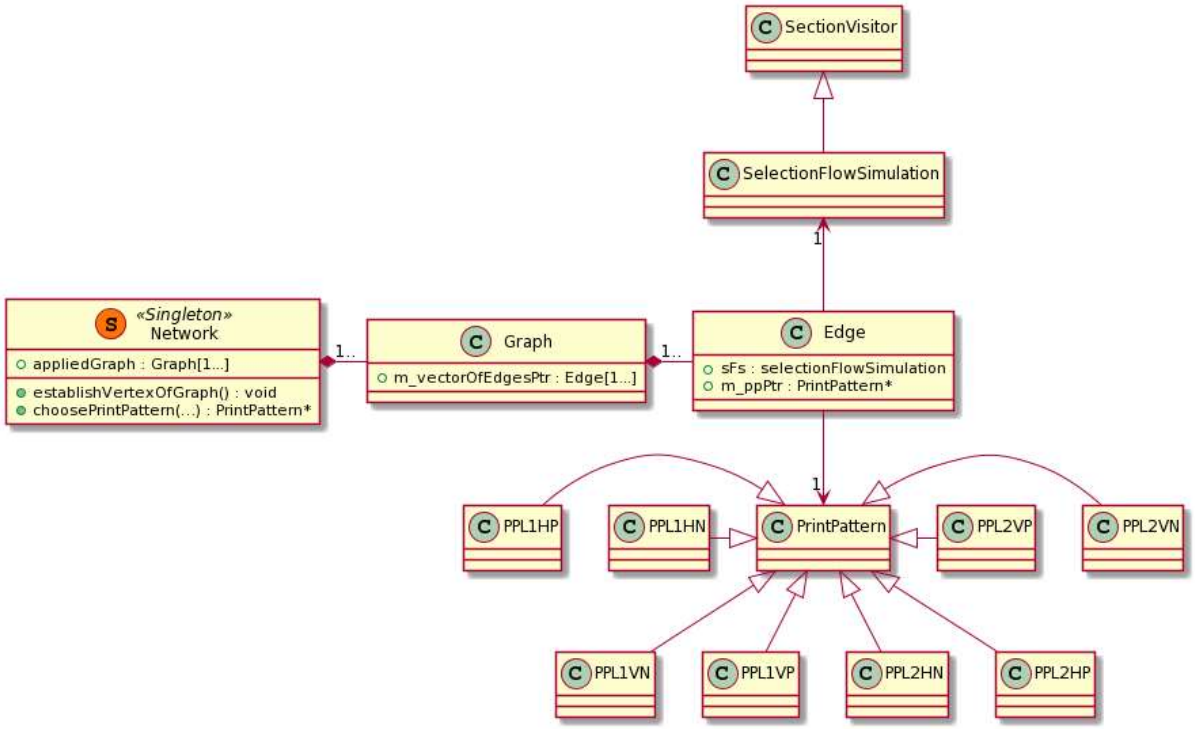


Abbildung 13: Auszug aus dem Klassendiagramm. Insgesamt besitzt die Klasse „PrintPattern“ acht verschiedene Implementationen von Subklassen, welche acht verschiedene Repräsentationsformen von Kanten(Edges) gegenüber dem User darstellen.

2.6.5.6 Schritt 2: Auswahl der passenden Verkehrsflusssimulationsalgorithmen

2.6.5.6.1 Auswahlkriterium

Die Auswahl des Verkehrsflusssimulationsalgorithmus wird bei einem Fahrstreifen dadurch bestimmt **mit welcher Subklasse die Eigenschaft „PrintPattern“** der Klasse „Edge“ instanziiert wurde.

Grundgedanke bei der Auswahl des Verkehrsflusssimulationsalgorithmus ist, dass **die Darstellung der Fahrstreifen/Netzwerkkanten (Edges)** gegenüber dem User das entscheidende Kriterium sein müsse, welches durch die Instanziierung der konkreten Subklassen der Eigenschaft „PrintPattern“ gegeben ist.

Methodisch erfolgt die Auswahl der Verkehrsflusssimulationsalgorithmen unter Anwendung einer sogenannten „Multimethode“, welche eingebettet ist in einer Implementation des Softwareentwurfsmusters „**Besucher/Visitor**“.

2.6.5.6.2 Problembeschreibung weshalb vom präferierten Ansatz der „Polymorphie“ nicht unmittelbar umgesetzt wird

Konkret wird die Prozedur von der Methode „selectionFlowSimulation::setStrategy(….) durchgeführt.

Im Grunde genommen sollte hier der Ansatz der „Polymorphie“ aufgegriffen werden. Es gibt mehrere Implementierungen des Methodenaufrufs „selectionFlowSimulation::setStrategy(..Subklasse von „PrintPattern“ als Übergabeparameter..)“ , welche sich alle nur dadurch unterscheiden, welche Subklasse von „PrintPattern“ übergeben wird. Dieser Ansatz, bei dem das sogenannte „Überladen“ einer Funktion zur Anwendung kommt, wäre an sich problemlos möglich, wenn als Distinktionsmerkmal zwischen den Implementierungen des Methodenaufrufs primitive Datentypen dienen würden, welche bereits zur „Compile-Time“ feststünden.

2.6.5.6.3 Diskussion über die Anwendung von „Run Time Type Information“

Der „Polymorphie-Ansatz“ wird hier aber um das Detail erschwert, dass zur Laufzeit ermittelt werden müsste, welcher dynamischer Subtyp der Oberklasse „PrintPattern“ als Übergabeparameter bei den unterschiedlichen Implementierungen des Methodenaufruf eingesetzt werden würde und somit als Entscheidungskriterium wirksam werden könnte.

Dieser skizzierte Ansatz lässt sich in dieser schlichten Form nicht unmittelbar umsetzen. Es müsste beim Überladen der Funktion „setStrategy(…)“ der Datentyp des Übergabeparameters zur Laufzeit ermittelt werden. Diese RTTI-Ermittlung (Run Time Type Information) müsste mit den Operator „`typeid()`“ oder mittels „dynamic_cast“, welches in eine IF-Abfrage eingebettet wäre, geschehen. Diese Ansätze sind jedoch relativ „teuer“, deshalb wurde ein anderer Ansatz gewählt, nämlich der Ansatz des „Double Dispatch“.

2.6.5.6.4 Implementierung anhand des Softwareentwurfsmusters des „Besuchers“ (Double Dispatch):

Dies erfolgt dadurch, dass das Attribut „PrintPattern“ der Klasse „Edge“ erneut näher betrachtet werden muss. Im vorhergehenden Kapitel wurde beschrieben, dass eine konkrete Instanziierung „PrintPattern-Objekte“ bereits im Zuge der Erzeugung der „Edge-Objekte“ erfolgt ist.

Im nächsten Schritt, unmittelbar nach der Instanziierung der Objekte „Edge“ sowie „PrintPattern“ wird bei jeder Richtungsfahrbahn, die Methode „PrintPattern::accept(…)“ ausgeführt. Als Übergabeparameter wird das Attribut der Klasse „Edge“ „selectionFlowSimulation“ mitgegeben.

Die Ausführung der Methode „accept(„selectionFlowSimulation“)“, welche in den Subklassen von „PrintPattern“ konkret implementiert ist, erwirkt, dass die über die Übergabeparameter von der Methode accept() eingebrachte Klasse „selectionFlowSimulation“ wiederum in Form der „setStrategy()-Methode“ unmittelbar ausgeführt wird. Die „setStrategy()-Methode der Klasse „selectionFlowSimulation“ beinhaltet als Übergabeparameter die Klasse „PrintPattern“. In diesem Fall wird mittels dem „this“-Operator die Eigenreferenz der Klasse „PrintPattern“ geliefert.

Je nach konkreter Implementierung der Klasse „PrintPattern“ wird eine konkrete Implementierung der Klasse „SectionVehicleSet“ generiert.

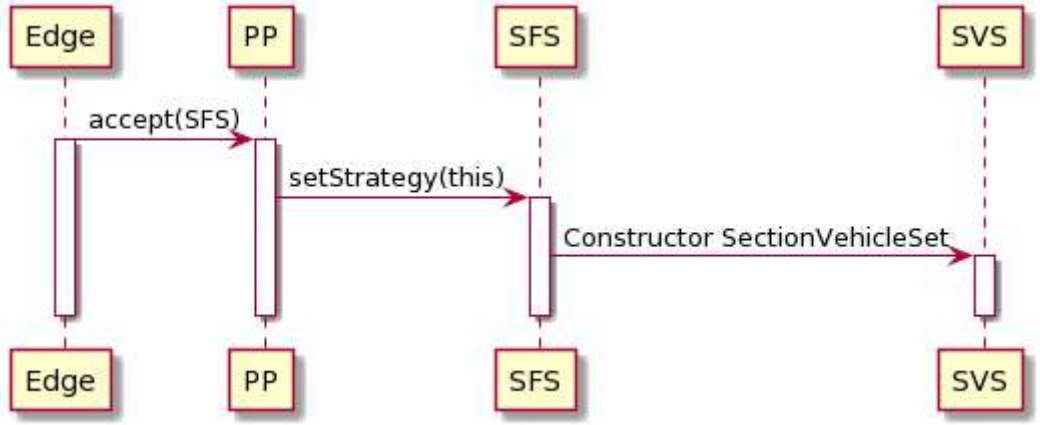


Abbildung 14: Sequenzdiagramm, „Double Dispatch“

Die „accept(…)“-Methode enthält einen Pointer auf das Objekt der Klasse „SelectionFlowSimulation“, welches ebenfalls Eigenschaft der Klasse „Edge“ ist. In der „accept(..)“-Methode wird eine abstrakte Methode der Klasse „SectionVisitor“ ausgeführt, welche in der erbenden Klasse „SelectionFlowSimulation“ für alle erbenden Objekte der Klasse „PrintPattern“ konkret implementiert ist. Diese Methode mit dem Namen „setStrategy“ erzeugt dynamisch die konkrete Implementierung der Klasse „SectionVehicleSet“, welche den passenden Verkehrssimulationsalgorthmus für die Kanten des Graphen bereitstellt.

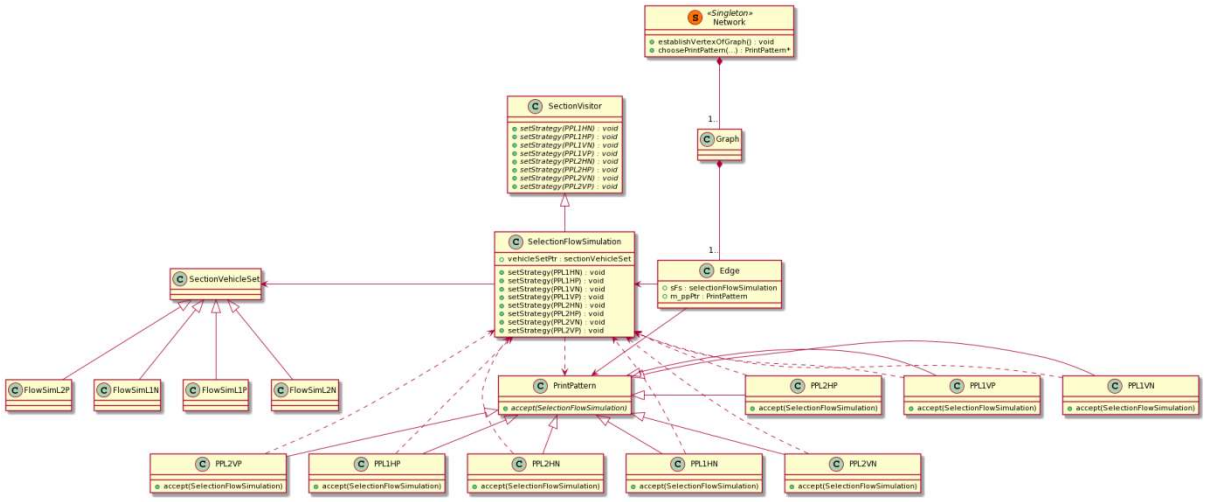


Abbildung 15: Auszug aus dem Klassendiagramm. Klassen die zur Auswahl des Simulationsalgorithmus dienen.

2.6.6 Modul: Ausgabe

Anknüpfend an die letzte Methode printContentOfSection(…)), welche im Rahmen des Simulationsabschnitts ausgeführt wird, soll hier beschrieben werden, wie die Ausgabe der Simulationsergebnisse gegenüber dem User gelöst wurde.

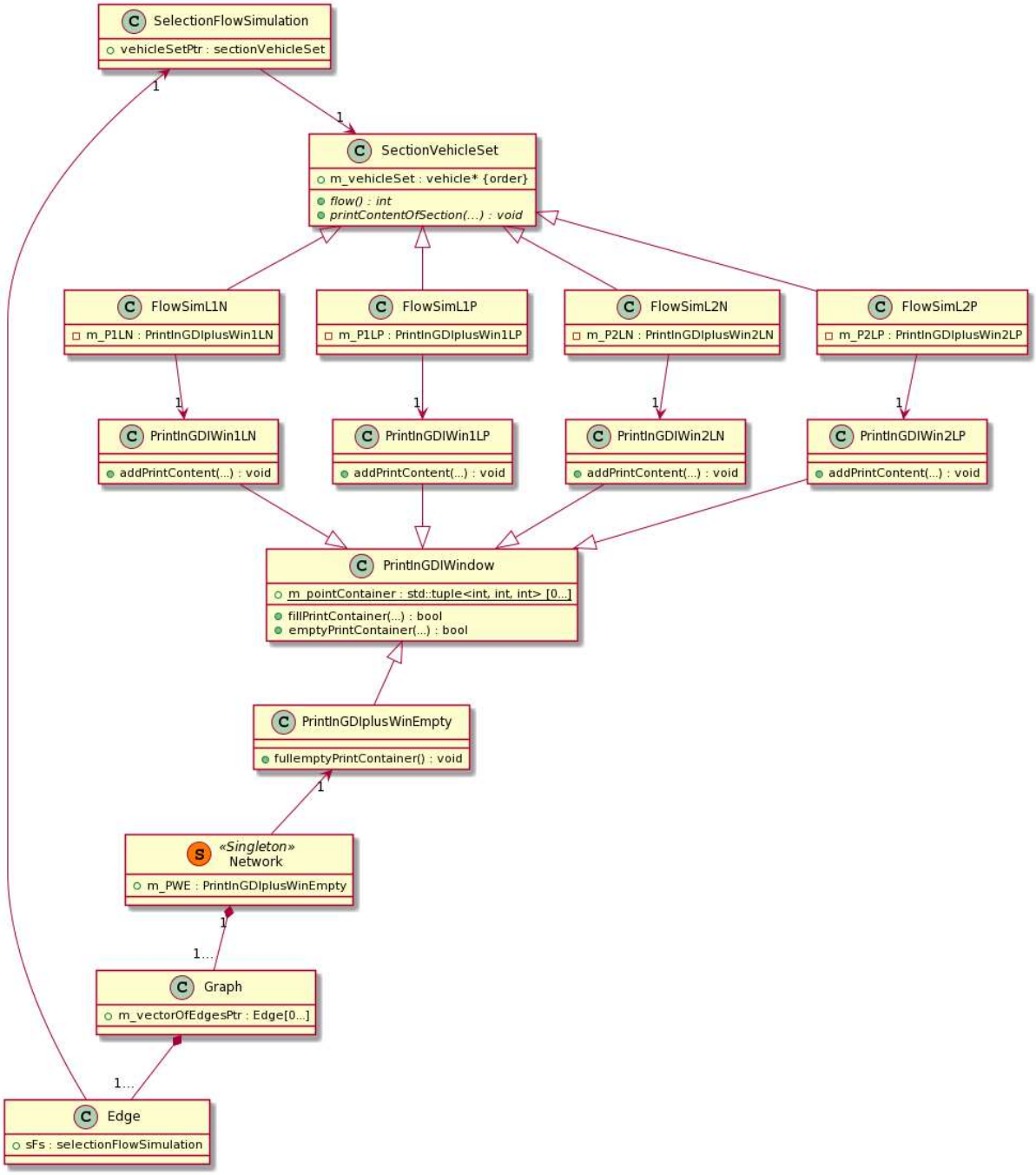


Abbildung 16: Auszug aus dem Klassendiagramm. Klassen zur Darstellung der Simulationsergebnisse

Im Zentrum steht die Klasse „PrintGDIWindow“. Sie enthält einen Container als Klassenvariable (static std::vector<std::tuple<int, int, int>> m_pointContainer) , in welchen sämtliche Simulationsergebnisse eines Simulationsschrittes abgelegt werden.

Die Implementation des Datencontainers als statische Eigenschaft (Klassenvariable) der Klasse „PrintGDIWindow“ hat den Vorteil, dass garantiert werden kann, dass diese Datenstruktur nur ein einziges Mal besteht. Dies ist auch dann gegeben, wenn die Klasse „PrintGDIWindow“ bzw. dessen Subklassen „PrintGDIWin1LN“, „PrintGDIWin1LP“, „PrintGDIWin2LN“, „PrintGDIWin2LP“ sowie „PrintInGDIplusWinEmpty“ in vielfacher Anzahl instanziiert werden.

Für das Befüllen des Containers „m_pointContainer“ mit Simulationsergebnissen stehen insgesamt vier verschiedene Subklassen der Klasse „PrintGDIWindow“ zur Verfügung. Die vier Subklassen „PrintGDIWin1LN“, „PrintGDIWin1LP“, „PrintGDIWin2LN“, „PrintGDIWin2LP“ der Oberklasse „PrintGDIWindow“ besitzen jeweils eine gleichlautende Methode „addPrintContent(…)“, welche jeweils dafür sorgt, dass die Simulationsergebnisse korrekt dargestellt werden. Die vier unterschiedlichen Implementierungen sind deshalb notwendig, weil je nach Lage der darzustellenden Kante (Edge) des Netzwerkes die Positionen der darzustellenden Simulationsdatenpunkte leicht korrigiert werden müssen.

Für das Entleeren des Container „m_pointContainer“ der Klasse „PrintGDIWindow“ steht die Methode „fullemptyPrintContainer(…)“ der Subklasse „PrintInGDIplusWinEmpty“ zur Verfügung. Bei Aufruf dieser Methode wird der gesamte Inhalt des Containers an der Oberfläche gegenüber dem User dargestellt, sowie im Folgeschritt aus dem Container gelöscht.

Im Klassendiagramm des Programms sind die Subklassen („PrintGDIWin1LN“, „PrintGDIWin1LP“, etc.) zum Befüllen des Datencontainers als Eigenschaften jener Subklassen instanziiert, welche die Verkehrssimulationsalgorithmen beinhalten („FlowSimL1N“, „FlowSimL1P“, etc.).

Aus dieser Anordnung lässt sich die Klassenstruktur wie folgt erklären: Bei der Erzeugung der Objekte der Klasse „Edge“ wird bereits die passende Subklasse der Klasse „PrintPattern“ erzeugt, welche wiederum in einem nächsten Schritt dazu führt, welcher bestimmte Verkehrsflusssimulationsalgorithmus ausgewählt wird. Diese Subklassen „FlowSimL1N“, „FlowSimL1P“, etc., wo die Verkehrsflusssimulationsalgorithmen implementiert sind, beinhalten als Eigenschaft eine Subklasse von „PrintGDIWindow“, welche zur korrekten Befüllung des „Containers“ mit den Simulationsergebnissen nötig sind.

Die Subklasse „PrintInGDIplusWinEmpty“ ist wiederum als Eigenschaft der Klasse „Network“ angelegt. Bei jedem Simulationsschritt wird in der Klasse „Network“ die Eigenschaft der Subklasse „PrintInGDIplusWinEmpty“ m_PWE.fullemptyPrintContainer() aufgerufen.

2.6.7 Modul: Eingriff in das Simulationsgeschehen

Als zusätzliches Feature zur eigentlichen Verkehrsflusssimulation sollte es dem User der Applikation auch möglich sein direkt auf die Simulation Einfluss zu nehmen.

Die Einflussnahme sollte in folgender Gestalt gelöst sein:

Durch Anklicken von Flächen, welche auf der Useroberfläche als Fahrstreifen dargestellt sind, soll eine direkte Einflussnahme auf die Verkehrsflusssimulation möglich sein.

Konkret sollte dieses Problem so gelöst sein, dass sämtliche „Mausklick-Events“ innerhalb der Useroberfläche während der Simulation vom Programm erfasst werden. Um die „Mausklick-Events“ einem bestimmten Fahrstreifen zuordnen zu können, wurde für dieses Programmfeature eine Klassenstruktur implementiert, welche dem Softwareentwurfsmusters des „**Beobachters**“ (Oberservers) entspricht.

2.6.7.1 Beschreibung des Softwareentwurfsmusters „Beobachter“

Dieses Softwareentwurfsmuster besteht im Westlichen aus zwei Teilen. Zum einen setzt es sich aus einem Komplex zusammen, welcher einen Zustand anzeigt. Dieser Zustand kann aber fortwährend von Veränderungen betroffen sein. Dieser Teil wird in der Analogie dieses Softwareentwurfsmusters als „Observer Subjekt“ bezeichnet.

Zum anderen besteht dieses Softwareentwurfsmuster aus einem Komplex, welcher die Zustandsveränderungen genau beobachtet. Dieser zweite Teil wird als „Observer“ (zu Deutsch: Beobachter) bezeichnet.

2.6.7.2 *Implementierung des Softwareentwurfsmusters „Beobachter“*

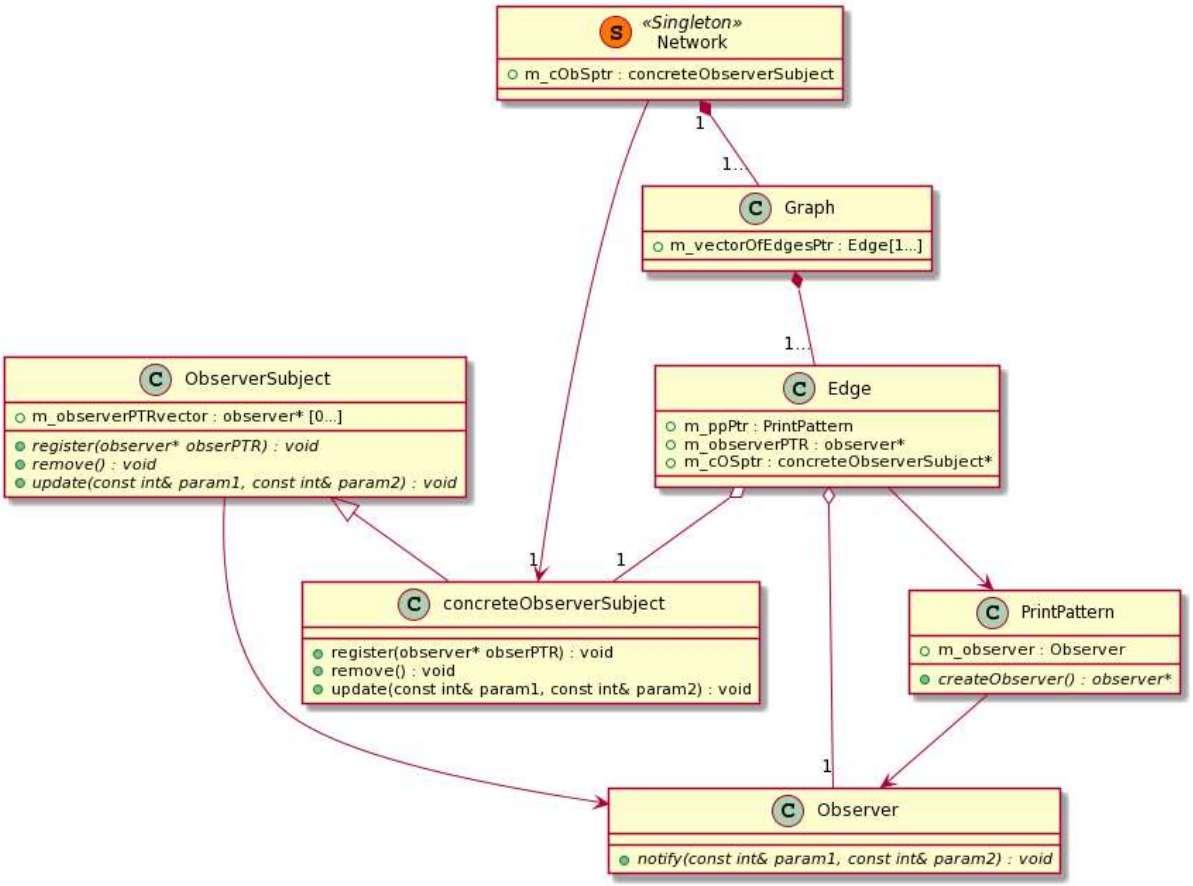


Abbildung 17: Auszug aus dem Klassendiagramm, Implementierung Softwareentwurfsmusters „Beobachter“. Die Subklassen der Klassen „PrintPattern“ sowie „Observer“ sind in diesem Klassendiagramm aus Übersichtsgründen nicht dargestellt.

2.6.7.2.1 „Observer Subject“:

Als „Subjekt“ des Softwareentwurfsmusters „Beobachter(Observer)“ werden bei dieser Implementation jene Nachrichten des Betriebssystems betrachtet, welche einen Mausklick der linken Maustaste innerhalb des Applikationsfenster melden. Diese Nachrichten der Betriebssystemschnittstelle (WinAPI) in der „Main-Datei“ beinhalten nach getätigten Mausklick Positionsdaten, welche vom zweiten Teil des Softwareentwurfsmusters, den „Beobachtern(Observer)“ genau analysiert werden.

Die in diesem Programm als Singleton instanziierte Klasse „Network“ wird von der „Main-Datei“ per Zeiger(Pointer) erreicht. In der Klasse „Network“ ist das „Observer Subject“ ein Attribut. Wird während der Ausführung der Simulation (Grundmodus 2) die linke Maustaste gedrückt, so wird über die WinAPI- Implementierung in der „main-Datei“ bzw. über die Klasse „Network“ die Methode „update(…)“ der Klasse „concreteObserverSubject“ ausgeführt, welche die Positionsdaten des Mauszeigers einer Prüfung zuführt.

2.6.7.2.2 „Observer“:

Die Objekte der Klasse „Observer“ sind Attribute der Klasse „PrintPattern“, welche wiederum Eigenschaften der Objekte der Klasse „Edge“ sind.

Die Prüfung der Positionsdaten des Mauszeigers, welche durch die Methode „update(…)“ der Klasse „concretObserverSubject“ ausgeht, wird durch die Methode „notify(….)“ der Objekte der Subklassen von „Observer“ durchgeführt.

In der „notify(…)“-Methode wird geklärt, ob die entsprechende Netzwerkkante, das heißt der entsprechende Fahrstreifen in der Simulation, vom „Mausklick-Event“ beeinflusst wird oder nicht.

2.6.7.3 Erzeugung des „Observers“

Das gesamte „Observer“-Klassenkonstrukt wird durch den Konstruktor der Klasse „Edge“ erzeugt. Dort wird über die Klasse „PrintPattern“ eine konkrete Implementierung eines „Observer“-Objektes erzeugt. Mittels eines „Pointers“ wird die Verbindung zwischen „Observer“ und „ObserverSubject“ hergestellt, indem die Methode „register(ObjectPTR)“ der Klasse „ObserverSubject“ aufgerufen wird.

2.6.7.4 Klassendiagramm des „Observer

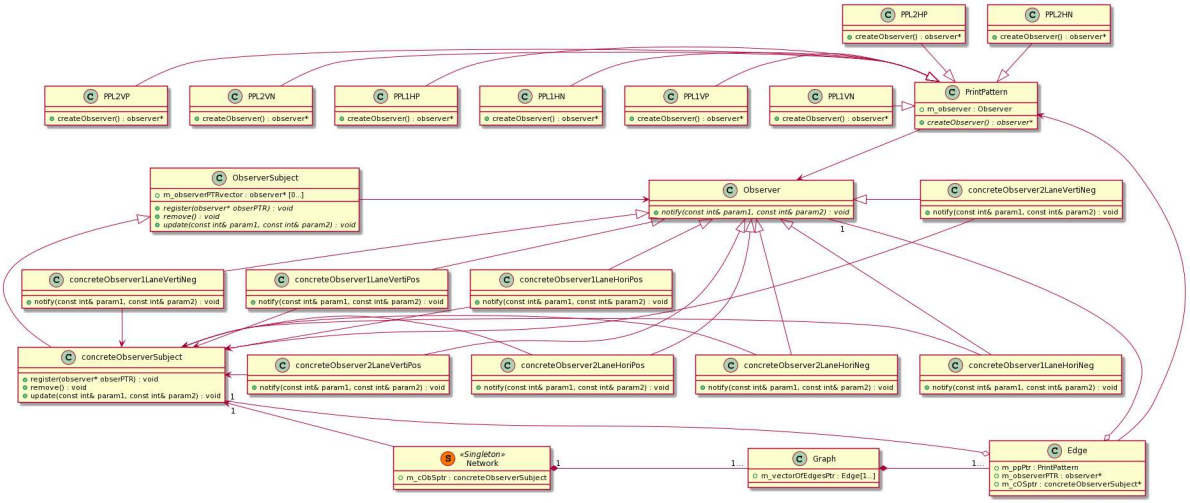


Abbildung 18: Auszug aus dem Klassendiagramm, Implementierung des Softwareentwurfsmusters „Beobachter“. Dieses Klassendiagramm enthält alle Subklassen der Hauptklassen „PrintPattern“ sowie „Observer“.

3 Eingesetzte Features

3.1 Windowsapplikation

Da es sich hier um eine Windowsapplikation handelt, wird der Startpunkt des Programmes, anders als wie für eine klassische Konsolenapplikation, nicht durch eine Funktion Namens „main()“ repräsentiert.

Die gesamte Applikation wird mit der Programmzeile „WinMain(…)“ gestartet. Zentral ist dabei, die so genannte „Include-Datei“ „Windows.h“. Sie stellt sämtliche Funktionaltäten bereit, damit das Applikationsfenster auf Userinput reagieren kann.

Konkret befindet sich im Kern der „WinMain“-Funktion eine Endlosschleife, welche nur durch schließen der Applikation beendet wird. Diese Endlosschleife ist so angelegt, dass sie auf Userinteraktion reagiert und je nach Programmstatus unterschiedliche Handlungen erwirkt.

3.2 PoolAllocator

Der PoolAllocator wurde verwendet, um Speicherplatz für eine große Menge an Fahrzeugen zu reservieren. Die “Placement new Syntax” ist das zentrale Feature des PoolAllocators. Sobald ein „Vehicle“ vom Netzwerk entfernt wurde, steht es für den erneuten Gebrauch zur Verfügung. Aufgrund der Struktur der Objekte „Vehicle“, welche nur aus nummerischen Eigenschaften bestehen, muss kein Augenmerk auf das Thema „Daten-Alignment“ gelegt werden.

3.3 Boost-Container

Die „Boost-Bibliotheken“ sind eine Sammlung an C++ Features, welche über die im „C++-Standard“ enthaltenen Features weit hinausgeht.

Im konkreten Programm wurde nach einem Feature gesucht, welche Pointer von Objekten beinhaltet. Ziel war es, die Objektzeiger in einem Container hinsichtlich einer ihrer Eigenschaften zu sortieren.

3.4 Threads (Async Futures)

Damit Prozesse beschleunigt bearbeitet werden können, kann dessen Bearbeitung auf verschiedene Threads verteilt werden.

In diesem Programm gibt es sehr viele Prozesse, welche aufgrund der iterierenden Struktur des Verkehrssimulators seriell abgearbeitet werden müssen.

Bei dieser Simulationsanwendung gibt es jedoch bei der Erzeugung der Zufallszahlen einen Prozess, welcher parallel zu anderen Abläufen ausgeführt werden kann. Dieser angesprochene Prozess wird bei Bedarf als „Future“ bzw. als asynchrone Funktion ausgeführt.

3.5 Unit-Tests

In diesem Programm wurden Unittests mit dem „Google Test“ C++-Test-Framework durchgeführt. Die Tests wurden nicht während der Erstellung des Programms durchgeführt, sondern nach dessen Fertigstellung. Als Konsequenz aus den Tests wurden ein paar wenige Korrekturen an den Methoden der Klassen „VertexFlex“ sowie „FlowSimL1P“ durchgeführt.

4 Abbildungsverzeichnis