Praktikum Informatik: Aufgabenstellung

September 19, 2024

Contents

1 Motivation

Während des Praktikums sollen alle wesentlichen Elemente objektorientierter Softwareentwicklung und ihre Umsetzung im Sprachumfang von $C++$ an einem (vereinfachten) Beispiel eingesetzt und geübt werden. Dabei sollen die modernen Pointerkonzepte der Smartpointer eingesetzt werden. Zur Darstellung oft benutzter Datenstrukturen wie Vektor, Liste oder Assoziativspeicher sollen die Klassen der Standard Template Library (STL) kennengelernt und benutzt werden. Die Aufgabe besteht aus aufeinander aufbauenden Teilaufgaben, die schließlich zu der Gesamtlösung führen. Die einzelnen Aufgaben sind zu zwei Blöcken zusammengefasst. Für jeden Block wird ein Testat abgenommen. Die Aufgaben sollen so implementiert werden, dass für jeden Block ein neues Projekt mit Eclipse angelegt wird. Dazu wird der vorhandene Block kopiert und dann erweitert. Somit sollte ein Testprogramm aus Block1 auch am Ende des Praktikums noch funktionieren.

2 Zielaufgabe

Es soll der Straßenverkehr in einer wenig erschlossenen Gegend modelliert und simuliert werden (siehe 2.1).

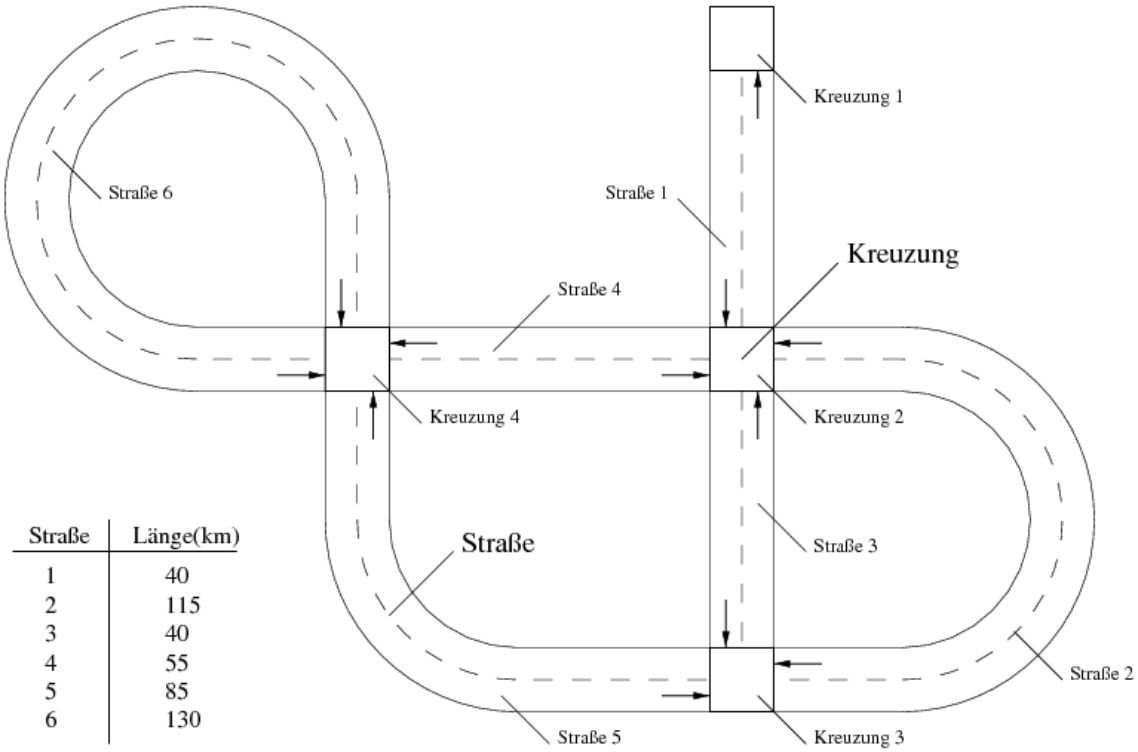
Verschiedene Arten von Fahrzeugen (PKW, Fahrrad) werden zu einem individuellen Startzeitpunkt von einem Knotenpunkt (Kreuzung) losgeschickt. Jedes Fahrzeug besitzt einen Zeit- und einen Streckenzähler sowohl für die Gesamtstrecke als auch für den Streckenabschnitt, auf dem es sich gerade befindet. Die Daten des Streckennetzes und der eingesetzten Fahrzeuge werden über Konstruktoren gesetzt oder eingelesen.

Das Modell setzt sich aus drei verschiedenen Simulationsobjekten zusammen: Fahrzeuge, Wege und Kreuzungen. Eine Verbindung zwischen zwei Kreuzungen wird durch eine Straße realisiert, die aus zwei entgegen gerichteten Wegen (Hin- und Rückweg) gebildet wird. Jeder Weg verwaltet die auf dem Weg befindlichen Fahrzeuge, jede Kreuzung die aus ihr abgehenden Wege. Wege können sowohl fahrende als auch parkende Fahrzeuge annehmen. Wenn die Simulation den Startzeitpunkt des parkenden Fahrzeugs erreicht, wird aus diesem ein fahrendes Fahrzeug.

Alle Simulationsobjekte enthalten eine Funktion, die einen Simulationsschritt ausführt. Kreuzungen simulieren dabei die von ihnen abgehenden Wege und Wege die auf ihnen befindlichen Fahrzeuge. Zu jedem Zeitschritt werden also durch die Simulation aller Kreuzungen des Systems nacheinander alle Simulationsobjekte bearbeitet. Das System wird durch einen globalen Zeittakt gesteuert. In jedem Zeittakt werden alle im System befindlichen Objekte genau **einmal** simuliert. Dies wird erreicht, indem jeweils die letzte Simulationszeit des Simulationsobjektes mit der globalen Zeit verglichen und anschließend synchronisiert wird. Auf einigen Wegen des Simulationssystems herrscht aufgrund ihrer Art (Stadtstrasse, Landstrasse, Autobahn) ggf. eine Geschwindigkeitsbegrenzung.

Einen Überblick über den Hauptteil der Simulation bietet die in Figure 2.2 dargestellte Klassenstruktur. Unterstützende Klassen finden Sie in Figure 2.3. Obwohl sicher auch andere Strukturen und Implementationen möglich wären, gehen wir im Praktikum von dieser Struktur aus. Es werden nicht alle Funktionen aufgeführt (keine Konstruktoren/Destruktoren), teilweise werden die Funktionen nur in der hierarchisch am höchsten gelegenen Klasse dargestellt. Sie können durchaus in abgeleiteten Klassen überschrieben werden. Bitte verwenden Sie in Ihrer Implementierung die dort aufgeführten Klassen- und Funktionsnamen, um den Betreuenden bei Fragen eine schnelle Orientierung zu ermöglichen. Selbstverständlich können (und müssen) Sie zur Implementierung und zum Test einzelner Module weitere Klassen und/oder Funktionen einführen.

Die Aufgaben bauen aufeinander auf, so dass am Ende des Praktikums die Simulation komplett implementiert ist. Die Funktionen für die grafische Ausgabe werden Ihnen zur Verfügung gestellt. In der Grafik sind PKWs durch rote und Fahrräder durch grüne



Straße	Länge(km)
1	40
2	115
3	40
4	55
5	85
6	130

Figure 2.1: Simulationsmodell

Punkte dargestellt. Das in der Liste jeweils selektierte Fahrzeug wird blau markiert. **Bevor Sie mit den Aufgaben beginnen, lesen Sie bitte die gesamte Aufgabenstellung durch, damit Sie wissen, wozu Klassen und Funktionen später genutzt werden.**

Im Anschluss finden Sie noch einige Vorgaben und Programmierhinweise.

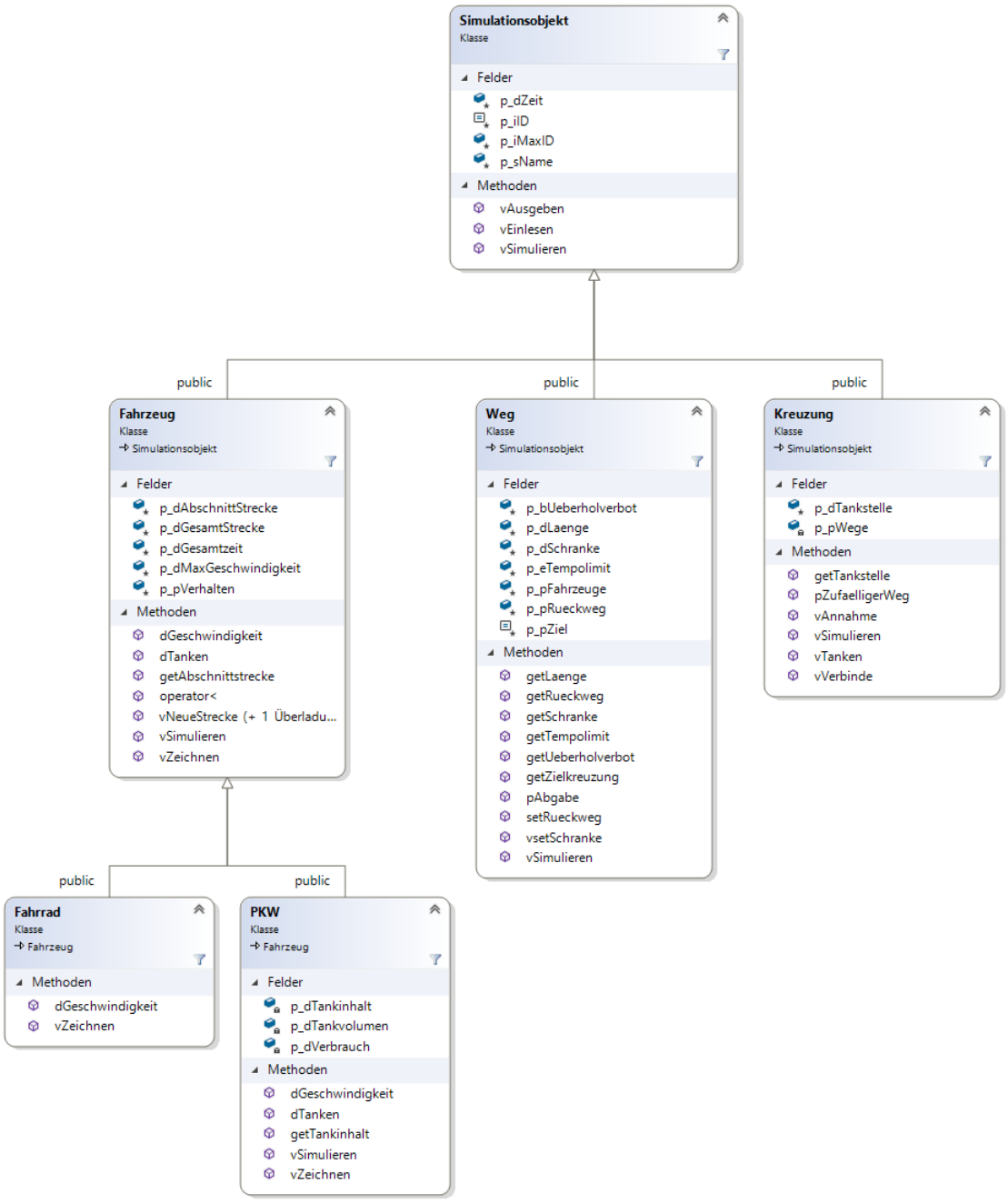


Figure 2.2: Klassenhierarchie Simulationsobjekt

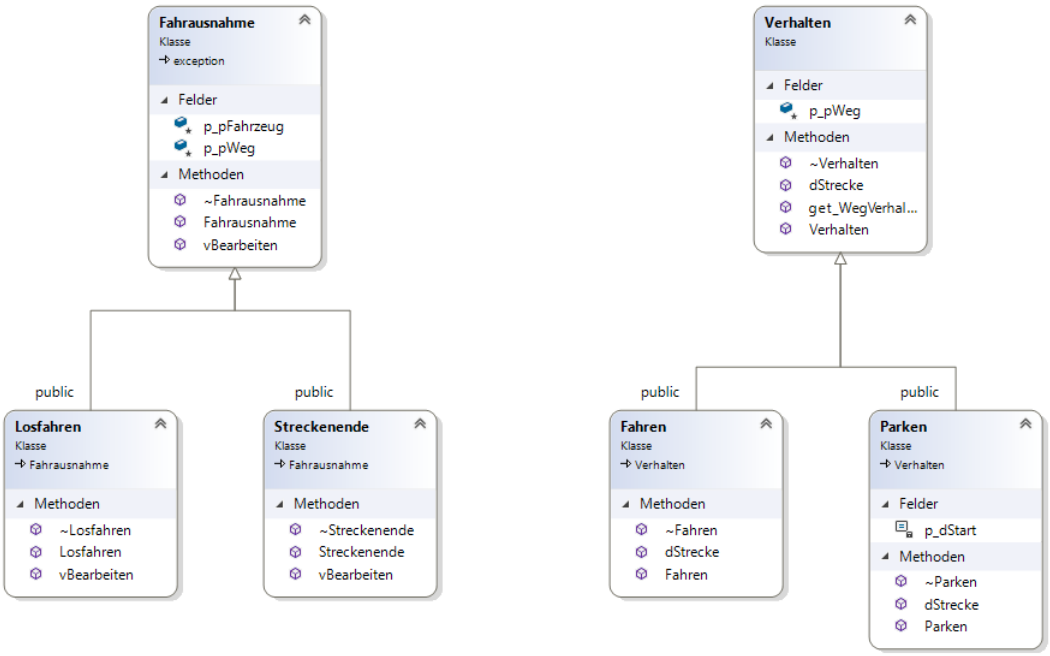


Figure 2.3: Ausnahme- und Verhaltensklassen

3 Hinweise zur Implementierung

Im Folgenden werden einige Hinweise zum Durchführen des Praktikums gegeben. Diese sollten nach Möglichkeit eingehalten werden, um den Betreuenden die Arbeit zu erleichtern und mögliche Fehlerquellen zu verringern. Die Aufgaben des Praktikums werden mit der Entwicklungsumgebung Eclipse bearbeitet. Sie werden feststellen, dass Eclipse viele Möglichkeiten für die C++-Programmierung bietet. Im Rahmen des Praktikums wird nur auf einen kleinen Teil der Funktionalitäten von Eclipse eingegangen. Ihnen steht es natürlich frei, sich tiefergehend in Eclipse einzuarbeiten und weitergehende Funktionen zu verwenden.

3.1 Verwaltung der Verkehrssimulation

Die Aufgaben sind in zwei Aufgabenblöcke unterteilt. Jeder Aufgabenblock soll als Projekt Aufgabenblock_X (mit X=1,2) angelegt werden und in dem ausgewählten Eclipse Workspace gespeichert werden. In der zugehörigen main()-Funktion soll dann für jede Aufgabe eine entsprechende Funktion vAufgabe_X() aufgerufen werden, welche die Funktionalität der entsprechenden Aufgabe testet.

3.2 Verwaltung der Projekte

3.2.1 Neues Projekt anlegen

Da in der Vergangenheit dabei immer wieder Probleme aufgetreten sind, gibt es an dieser Stelle nun eine detaillierte Anleitung.

1. Öffnen Sie $File \rightarrow New \rightarrow Project$.
2. Im geöffnetem Fenster wählen Sie $C/C++ \rightarrow C/C++\ Project$ aus und bestätigen mit $Next$.
3. Wählen Sie in dem neuen Fenster nun $All \rightarrow C++\ Managed\ Build$ aus und bestätigen erneut mit $Next$.
4. Nun geben Sie unter *Project name* den entsprechenden Projektnamen (z.B. Aufgabenblock_1) ein und wählen als $Project\ type \rightarrow Executable \rightarrow Empty\ Project$ aus. Auf der rechten Seite wählen sie als $Toolchain\ MinGW\ GCC$ aus. Die *Location* können Sie auf der Standardeinstellung lassen. Bestätigen Sie nun mit $Finish$ (s. Figre 3.1).

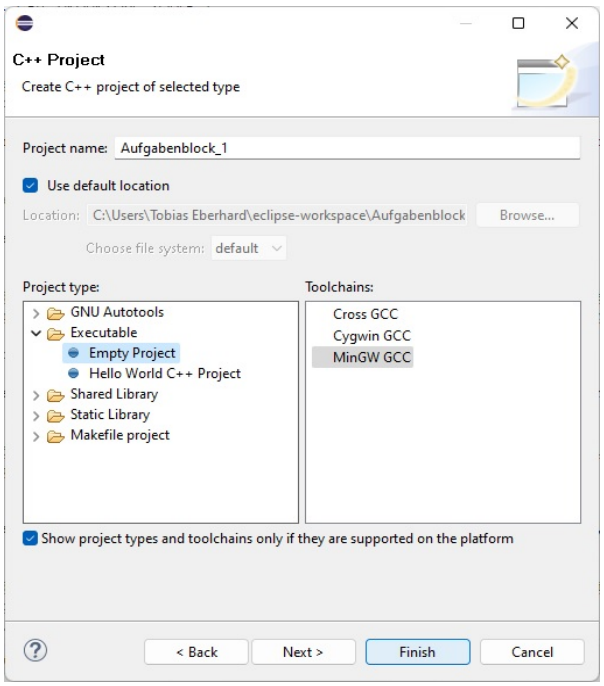


Figure 3.1: Neues Projekt anlegen

3.2.2 C++-Dateien erzeugen

1. Wählen Sie im *Project Explorer* das entsprechende Projekt aus. Klicken Sie mit der rechten Maustaste auf das Projekt. In dem sich öffnenden Kontextmenü wählen Sie *New*. Nun haben Sie unter anderem die Auswahl zwischen

   a) *Header File*: Hiermit können Sie eine leere *.h*-Datei erzeugen. Dabei müssen Sie den Dateinamen (in Anlehnung an die Klasse, z.B. Fahrzeug.h) angeben und ein Template auswählen. Als Template wählen Sie hier Default C++ header template aus.

   b) *Source File*: Hierüber erzeugen Sie eine leere $C++$-Datei. Als Dateinamen geben Sie z.B. Fahrzeug.cpp an. Wählen Sie als Template das Default C++ source template aus.

   c) *Class*: Mit Hilfe dieses Menüeintrags lässt sich eine gesamte Klasse mit Header- und Source-Dateien erzeugen. In der Abfrage müssen Sie ein *Source folder* angeben (das Projekt, z.B. Aufgabenblock_1), den Klassennamen definieren (z.B. Fahrzeug) und über *Method stubs* lassen sich einige grundlegende Konstruktoren und Destruktoren automatisch erzeugen.

2. Sie finden die erzeugte Datei nun in Ihrem Projektverzeichnis und können Sie entsprechend mit Inhalt und Funktionalität füllen.

3.2.3 Inhalte in neue Projekte übernehmen

1. Zunächst folgen Sie den ersten drei Schritten um ein neues Projekt anzulegen.
2. Als *Name* wählen Sie nun Aufgabenblock_2.
3. Nun bestätigen Sie und in Ihrem *Workspace* wird ein neues Unterverzeichnis Aufgabenblock_X angelegt.
4. Kopieren Sie nun die Sourcen aus dem Verzeichnis des alten Projekts in das neue Projektverzeichnis (wichtig: es sollen nur die *.h und *.cpp Dateien kopiert werden!). Kopieren Sie zusätzlich, sofern vorhanden, aus Moodle.
5. In Eclipse müssen Sie nun die neuen Dateien im neuen Projekt laden. Dazu wählen Sie im *Project Explorer* das neu erstellte Projekt aus und drücken $F5$. Dadurch lädt Eclipse die neuen Dateien und diese werden nun im Projekt angezeigt.

Hinweis: Um ein Projekt auszuführen, wählen Sie es im *Project Explorer* **aus und wählen nun** *Project* $\rightarrow$ *Build Project* **aus. Dadurch wird Ihr Projekt kompiliert. Als nächstes können Sie das Projekt über** *Run* $\rightarrow$ *Run as* $\rightarrow$ 2 *local C / C++ Application* **ausführen.**

3.3 Namenskonvention

Benutzen Sie bitte in all Ihren Lösungen folgende Präfixe für Variablen und Funktionen. Es erleichtert Ihnen (und auch uns) das Lesen des Quellcodes, da aus dem Namen unter anderem auch schon der Typ der Variablen/Funktionen ersichtlich ist.

- Instanzvariablen der Klassen werden durch ein $p\_$ ganz vorne am Variablennamen gekennzeichnet.
- Darauf folgt ein Buchstabe, der den Typ der Variablen bzw. den Rückgabewert der Funktion beschreiben.

```
i=int; t=struct; d=double; v=void; b=bool;
s=string; p=pointer; e=enum;
```

- Danach folgt dann der eigentliche Variablenname, wobei der erste Buchstabe eines jeden Teilwortes großgeschrieben wird.
- Im speziellen Fall, dass die Funktion nur eine private/protected Variable setzt oder zurückliefert, ist das erste Wort des eigentlichen Variablennamens *get* bzw. *set* und danach folgt der Name der Variablen ohne die obigen Präfixe. Diese Funktionen heißen auch getter/setter-Methoden.
- Eine Funktion wird dadurch gekennzeichnet, dass dem Namen runde Klammern folgen.

Beispiele:

- p_iID: protected/private-Variable vom Typ *int*
- bIstFertig(): Funktion, die einen boolschen Wert zurückliefert (*true* oder *false*
- getID(): getter für die Variable (p_iID).
- vFunktion(): Funktion, die nichts (*void*) zurückliefert.

3.4 Programmierhinweise

- Implementieren Sie für jede Klasse eine Datei *Klassenname.h* zur Deklaration der Variablen und Funktionen. Weiterhin jeweils eine Datei Klassenname.cpp zur Definition des Codes.
- Neben den Funktionen, die zur Lösung der Aufgaben vorgegeben werden, können Sie natürlich zusätzlich noch eigene Funktionen implementieren.
- Kommentieren Sie Ihre Programme ausreichend, sodass auch Außenstehende (Betreuende) Ihren Code nachvollziehen können. Dieser Punkt geht auch mit in die Bewertung ein.
- Entscheiden Sie, ob es bei der Definition von Funktionen, Variablen oder Parametern sinnvoll ist, diese als *const* zu deklarieren. Wählen Sie dies, wo immer es möglich ist.
- Sie können, wenn Sie mehrere Elemente der *STL* verwenden hinter die jeweiligen Includes using namespace std; schreiben. Bedenken Sie, dass dies nicht die Regel ist.
- Alle Dateien, die wir Ihnen im Laufe des Praktikums zur Verfügung stellen, finden Sie im RWTHmoodle Lehr- und Lernbereich des Praktikums unter Vorgabedateien.
- Denken Sie an die aufeinander aufbauende Programmstruktur der Aufgabe.
- Testen Sie alle erstellten Klassen und Funktionen für (weitgehend) alle denkbaren Situationen. Ein einzelner Test zeigt noch nicht die Korrektheit des Programms. Wählen Sie entsprechend mehrere repräsentative Testfälle und begründen Sie Ihre Testauswahl. Dieser Punkt (z.B. "'Übersehene"' Testfälle oder nur automatisierte Zufallstests) geht auch in die Bewertung ein.

4 Aufgabenblock 1: Grundlagen des Verkehrssystems

4.1 Motivation

In diesem ersten Aufgabenblock werden Klassen für die zu simulierenden Fahrzeuge erstellt, PKWs und Fahrräder, die sich selbst fortbewegen können und simulierbar sind. Ein Mini-Eventhandler ruft eine entsprechende Simulationsschrittmethode aller Fahrzeuge mehrmals auf und gibt den aktuellen Stand der Fahrzeuge nach jedem Schritt auf dem Bildschirm aus.

In diesem Aufgabenblock werden folgende Punkte betrachtet:

- Deklaration und Definition von Klassen,
- Implementierung von Konstruktoren und Destruktoren,
- Kapselung von Daten und Zugriff auf private Member,
- Elementare Verwendung von Smartpointern und static Variablen,
- Vererbung,
- Einsatz der *STL* (string, vector),
- Unterscheidung der Klassenbereiche public, private und protected,
- Unterscheidung einfache und virtuelle Vererbung,
- Überladen von Operatoren.

Um sich einen Überblick zu verschaffen, lesen Sie den ersten Aufgabenblock zunächst komplett durch.

4.2 Fahrzeuge (Einfache Klassen)

1. Starten Sie Eclipse und erstellen Sie in Ihrem Homebereich ein neues Projekt mit dem Namen Aufgabenblock_1.

2. Ändern Sie den C++-Dialekt Ihres Projektes auf C++-17. Dazu klicken Sie mit der rechten Maustaste auf Ihr Projekt und wählen "Properties" aus. Hier wählen Sie nun "*C/C++ Build → Settings*" aus. Auf der rechten Seite wählen Sie nun den Reiter "Tool Settings". Hier öffnen Sie nun "*GCC C++ Compiler → Dialect*". Im rechten Bereich wählen Sie nun unter "Language standard" "ISO C++ 17 (-std=c++17)".

3. Implementieren Sie eine Klasse *Fahrzeug* zur Verwaltung verschiedener Fahrzeuge. Die Klasse soll zunächst lediglich private Membervariablen haben, in denen der Name des Fahrzeugs (p_sName) und eine eindeutige ID (p_iID) zu jedem Objekt gespeichert wird. Benutzen Sie für den Namen den Datentyp *string*. Implementieren Sie einen Konstruktor, der einen *string* als Parameter hat und damit den Namen initialisiert. Weiterhin soll die Klasse mit einem Default-Konstruktor konstruierbar sein. Dann soll der Name leer ("") sein. Zum Initialisieren der Variablen können Sie diese direkt bei der Definition mit 0 bzw. "" vorbesetzen. Die ID soll im Konstruktor anhand einer hochzählenden, Klassenvariablen p_iMaxID vergeben werden. Nutzen Sie im Konstruktor eine Initialisierungsliste, um die Konstanten einmalig mit einem Wert zu besetzen. Da sich die ID nach der Konstruktion nicht mehr ändert, können Sie diese als *const* definieren. Wie Sie Klassenvariablen definieren, können Sie im Skript (Statische Klassenelemente) nachlesen.

   Erzeugen Sie in den Konstruktoren und dem Destruktor eine Ausgabe, welche den Namen und die ID des erzeugten bzw. gelöschten Objekts mit einem entsprechenden Hinweis auf die Operation auf der Standardausgabe *cout* ausgibt.

   Für das weitere Programm setzen Sie da, wo Sie können, Variablen und Funktionen bevorzugt *const*.

4. Beim Programmieren ist es meist ratsam, schnell ein lauffähiges Programm zu haben. Erzeugen Sie eine neue C++-Datei (main.cpp), die die Funktion vAufgabe_1() aufruft und implementieren Sie diese Funktion innerhalb der Datei main.cpp. Erzeugen Sie in dieser Funktion einige Elemente statisch (über Deklaration) und einige dynamisch (mit *new*). Löschen Sie zum Schluss die dynamisch erzeugten Elemente. Erzeugen und starten Sie das Programm und testen Sie das korrekte Erzeugen und Löschen der Objekte. Im Anschluss erzeugen Sie nun einige Smartpointer (siehe Skript). Benutzen Sie dazu make_unique<Fahrzeug> und make_shared<Fahrzeug>, um von beiden Typen je 2 Objekte vom Typ Fahrzeug zu erzeugen. Speichern Sie eines der als shared_ptr erzeugten Objekte in einem weiteren shared_ptr. Lassen Sie sich vor und nach der Zuweisung mit use_count die Anzahl der Referenzen ausgeben. Versuchen Sie dasselbe mit unique_ptr. Erzeugen Sie zusätzlich einen vector<unique_ptr<Fahrzeug>>. Speichern Sie die oben erzeugten Fahrzeuge in diesem Vektor. Welche Fahrzeuge können Sie dort speichern? Wie müssen Sie den Besitzwechsel anzeigen? Löschen Sie danach den Vektor mit *clear*. Beobachten Sie mit dem Debugger, was dort passiert und wann die Objekte durch Aufruf des Destruktors gelöscht werden.

Erzeugen Sie nun zusätzlich auch einen vector<shared_ptr<Fahrzeug>>. Welche Fahrzeuge können Sie dort speichern? Fügen Sie dem Vektor zwei Fahrzeuge hinzu, eins mit und eins ohne *move*. Lassen Sie die Anzahl der Referenzen ausgeben. Beobachten Sie wieder mit dem Debugger, was dort passiert und wann die Objekte durch Aufruf des Destruktors gelöscht werden. Bitte notieren Sie beim Debuggen, in welchen Zeilen die jeweiligen Objekte gelöscht werden. Machen Sie sich in dieser Aufgabe mit den Grundfunktionen des Debuggers (Schrittmöglichkeiten, Haltepunkte) bekannt. Beobachten Sie im Fenster für die Variablenüberwachung den Inhalt der Variablen. Die Benutzung des Debuggers wird im Skript beschrieben. Die Funktionen des Debuggers müssen bei der Abnahme ggf. vorgeführt und erläutert werden.

5. Erweitern Sie die Klasse um Membervariablen für die Maximalgeschwindigkeit des Fahrzeugs (p_dMaxGeschwindigkeit), die bisher zurückgelegte Gesamtstrecke (p_dGesamtStrecke), die gesamte Fahrzeit des Objektes (p_dGesamtZeit) und die Zeit, zu der das Fahrzeug zuletzt simuliert wurde (p_dZeit). Fügen Sie einen weiteren Konstruktor hinzu, der einen Namen und die maximale Geschwindigkeit als Parameter bekommt. Beachten Sie hier, dass die Geschwindigkeit immer positiv sein soll. Prüfen Sie dies in der Initialisierungsliste mit Hilfe des ?-Operators.

6. Da dieses Programm noch nicht viel am Bildschirm ausgibt, schreiben Sie eine Memberfunktion vAusgeben(). Diese Memberfunktion soll spezifische Fahrzeugdaten ausgeben. Machen Sie innerhalb dieser Funktion keine Aufrufe von cout::endl, sondern programmieren Sie die Zeilenwechsel in der Hauptfunktion. Die Ausgabe soll so formatiert werden, dass unter einer Überschrift die Daten tabellarisch aufgelistet werden, in etwa folgendermaßen:

```
ID    Name      MaxGeschwindigkeit  Gesamtstrecke
-------------------------------------------------
  1  PKW1                   40.00           0.00
  2  AUTO3                  30.00           0.00
```

   Die Überschrift soll durch eine Klassenfunktion void vKopf() ausgegeben werden. Benutzen Sie für die Formatierung keine feste Anzahl von Leerzeichen, sondern die Input-Output-Manipulatoren der Standard C++ Bibliothek (<iomanip>). Schreiben Sie eine Klassenfunktion in Fahrzeug, um mit Fahrzeug::vKopf() die Überschrift ausgeben zu können. **Beachte**: Bei Verwendung von setiosflags() zum Setzen der Ausgabeausrichtung (rechts-/linksbündig) sollte zunächst die andere Ausrichtung mittels resetiosflags() zurückgesetzt werden.

7. Bevor die Simulationsfunktion der Fahrzeuge geschrieben werden kann, muss erst noch eine globale Uhr programmiert werden, damit die Fahrzeuge wissen, wie viele Stunden sie simulieren sollen. Zur Realisierung dieser Uhr definieren Sie eine globale Variable dGlobaleZeit, die Sie mit 0.0 initialisieren.

Beachte: Zur Benutzung dieser Variablen innerhalb anderer Klassen muss sie der Klasse erst mittels der *extern*-Deklaration bekannt gemacht werden.

8. Schreiben Sie nun die Memberfunktion Fahrzeug::vSimulieren(), welche dafür sorgt, dass die Fahrzeuge sich fortbewegen. Dazu wird mit Hilfe der globalen Uhr ermittelt, wieviel Zeit seit dem letzten Simulationsschritt vergangen ist, und entsprechend dieser Information wird der Zustand des Fahrzeugs aktualisiert (u.a. Gesamtstrecke um die im ermittelten Zeitraum aufgrund der Geschwindigkeit fahrbare Strecke erhöhen). Lassen Sie das Fahrzeug mit maximaler Geschwindigkeit fahren. Sorgen Sie durch einen Zeitvergleich dafür, dass ein Fahrzeug in einem Zeitschritt nur *einmal* bearbeitet wird, auch wenn es versehentlich zweimal innerhalb eines Zeitschritts aufgerufen wird. Aktualisieren Sie auch die Gesamtfahrzeit und die letzte Abfertigungszeit des Objektes. Momentan sind diese Werte noch gleich. Das wird sich später ändern.

9. Schreiben Sie eine neue Hauptfunktion vAufgabe_1a(). Lesen Sie Namen und Maximalgeschwindigkeit für 3 Fahrzeuge aus der Konsole ein, erzeugen Sie diese mit make_unique<Fahrzeug> und speichern sie diese in einem Vektor. Simulieren Sie Fahrzeuge über eine gewisse Zeitspanne. Erhöhen Sie dazu in einer Schleife die globale Uhr jeweils um einen Zeittakt und rufen Sie in der Schleife die Simulationsfunktion und die Ausgabefunktion der Fahrzeuge auf. Wählen Sie als Zeittakt auch Bruchteile von Stunden.

4.3 Fahrräder und PKW (Unterklassen)

1. Implementieren Sie zwei neue Klassen PKW und $Fahrrad$, die jeweils von der Basisklasse $Fahrzeug$ abgeleitet werden. Strukturieren Sie die Klasse Fahrzeug dementsprechend um. Überlegen Sie, welche Variablen *private* bleiben sollten und welche *protected* werden. Überlegen Sie weiterhin, welche Funktionen *virtual* werden. Benutzen sie synchron dazu das Schlüsselwort *override* für diese Funktionen in den abgeleiteten Klassen. Überlegen Sie zusätzlich welche Funktionen und/oder Variablen Sie *const* setzen können.

2. Da Fahrräder mit Muskelkraft und PKWs mit Motoren betrieben werden, benötigt die Klasse PKW zusätzliche PKW-spezifische Variablen. Fügen Sie der Klasse PKW die Variablen p_dVerbrauch (Liter/100km), p_dTankvolumen sowie p_dTankinhalt (Liter) hinzu.

   Ergänzen Sie die Klasse um einen entsprechenden Konstruktor, mit dem Sie zusätzlich zu den fahrzeugspezifischen Membervariablen auch Verbrauch und (optional, Default=55 l) Tankvolumen setzen können. Der Tankinhalt wird jeweils auf die Hälfte des Tankvolumens initialisiert. Nutzen Sie für die Einbeziehung der Konstruktoren der Basisklasse eine Initialisierungsliste.

   Des Weiteren schreiben Sie eine Funktion dTanken mit optionalem Parameter dMenge zum nachträglichen Betanken der PKWs. Wird kein Wert übergeben

(Defaultparameter) soll vollgetankt werden, ansonsten wird der gewünschte Wert getankt. Sie können die Konstante std::numeric_limits<double>::infinity() aus dem Header <limits> als Default-Wert verwenden, die größer als alle (anderen) *double*-Werte ist. Beachten Sie, dass maximal das Tankvolumen aufgefüllt werden kann. Geben Sie jeweils die tatsächlich getankte Menge zurück. Implementieren Sie die Funktion in der Klassenhierarchie so, dass sie für alle Fahrzeuge aufrufbar ist. Fahrräder und Fahrzeuge ohne Tank tanken bekanntlich nicht, d.h. die Funktion macht nichts und gibt immer 0 Liter zurück.

Bei jedem Simulationsschritt soll der Tankinhalt aktualisiert werden, bis der Tank leer ist. PKWs ohne Tankinhalt sollen liegenbleiben bis wieder nachgetankt wird. Danach sollen sie normal weiterfahren. Zur Vereinfachung soll die Reserve so groß sein, dass der PKW im letzten Schritt noch die komplette Teilstrecke fahren kann. Implementieren Sie dazu für PKW eine eigene Funktion vSimulieren(), die die zusätzliche Funktionalität von PKW implementiert. Für die allgemeine Simulation soll aber weiterhin Fahrzeug::vSimulieren() aufgerufen werden.

Gesamtverbrauch (berechnet aus Gesamtstrecke) und aktueller Tankinhalt sollen außerdem noch in vAusgeben ergänzt werden. **Beachte**: Um Codeduplizierung in den abgeleiteten Klassen zu vermeiden, sollen die Daten, die zu $Fahrzeug$ gehören, immer von Fahrzeug::vAusgeben ausgegeben werden. Rufen Sie diese Funktion also auch in den Ausgabefunktionen der abgeleiteten Klassen auf. Ergänzen Sie auch die Überschrift in Fahrzeug::vKopf() entsprechend.

3. Da Fahrradfahrer nicht immer mit maximaler Geschwindigkeit fahren können, soll eine Memberfunktion dGeschwindigkeit() implementiert werden. Sie wird in $Fahrzeug$ als virtuell deklariert und für $Fahrrad$ überschrieben. PKWs sollen immer mit ihrer vollen Geschwindigkeit fahren, Fahrradfahrer dagegen werden langsamer. Jeweils ausgehend von der gefahrenen Gesamtstrecke soll die Geschwindigkeit pro $20km$ um 10% abnehmen, minimal jedoch $12km/h$ betragen. Während eines Berechnungsschritts ist die Geschwindigkeit als konstant anzusehen. *Beispiel* : Nach 50 gefahrenen Kilometern beträgt die Geschwindigkeit im nächsten Zeittakt noch 81% der Maximalgeschwindigkeit, falls diese noch mehr als $12km/h$ beträgt.

   Stellen Sie nun Fahrzeug::vSimulieren() auf diese Funktionalität um (statt Maximalgeschwindigkeit). Ändern Sie die Methode vAusgeben, sodass für jedes Fahrzeug zusätzlich zu den Fahrzeugdaten die aktuelle Geschwindigkeit ausgegeben wird.

4. Schreiben Sie eine neue Funktion vAufgabe_2(): Lesen Sie die Anzahl der zu erzeugenden PKWs und Fahrräder aus der Konsole ein, konstruieren Sie entsprechende Objekte der Klassen PKW und $Fahrrad$ und verwalten Sie sie in *einem* vector<unique_ptr<Fahrz
   Warum können/sollten Sie keine Fahrzeugobjekte speichern? Warum können Sie PKWs und Fahrräder in einem gemeinsamen Vektor speichern?

   Führen Sie für diese Objekte mehrere Simulationsschritte durch. Nach genau 3 Stunden tanken Sie die PKWs nochmals voll. Die Zeitabfrage dazu soll im Testprogramm erfolgen, nicht innerhalb von dTanken(). Testen Sie dies mit verschiedenen

Zeittakten. Geben Sie die Ergebnisse (Daten aller Fahrzeuge) nach jedem Schritt aus.

Beachte: Gleichheit von *double*-Werten kann *immer* nur gegen eine Toleranz ϵ getestet werden, da Fließkomma-Berechnungen nicht komplett genau sind. Berechnen Sie dazu z.B. den Absolutbetrag der Differenz bei Gleichheit oder reduzieren Sie eine der Seiten des Vergleichs um ϵ bei $\geq$ oder $\leq$. Die Funktion für den Absolutbetrag std::fabs() finden Sie in der Bibliothek <cmath>. Beachten Sie dieses Rundungsproblem bei allen weiteren Vergleichen zwischen Fließkomma-Werten.

4.4 Ausgabe der Objekte (Operatoren überladen)

1. Am Ende dieser Aufgabe sollen Fahrzeuge mit dem Ausgabeoperator angezeigt werden können. Fahrzeug hat ja bereits eine Methode vAusgeben(). Diese Memberfunktion soll nun fahrzeugspezifische Daten auf dem übergebenen Ausgabe-Stream ausgeben. Dazu ändern sie diese nun in eine virtuelle Methode vAusgeben(ostream&) const. Markieren Sie in allen Fahrzeug-Klassen die geerbte vAusgeben-Methode als *override*, falls noch nicht geschehen.

2. Nun wollen wir Fahrzeug mit operator<< ausgeben können. Dazu müssen Sie den Ausgabeoperator überladen. Hier zur Erinnerung nochmal die Deklaration des Ausgabeoperators für eine Klasse X:

   ```
   ostream& operator<<(ostream& o, const X& x);
   ```

   Rufen Sie im Ausgabeoperator die vAusgeben-Methode mit dem übergebenen ostream auf.

   Beachte: Überladen Sie den Operator außerhalb der Klasse.

   Warum? Kommen Sie mit einer einzigen Definition für alle von Fahrzeug abgeleiteten Klassen aus? Verwenden Sie bitte keine friend-Deklaration.

3. Testen Sie den Ausgabeoperator, indem Sie Fahrzeuge, PKWs und Fahrräder damit auf *cout* ausgeben:

   Beispiel:

   ```
   std::cout << *aPKW << std::endl << *aFahrrad << std::endl;
   ```

   Verwenden Sie ab jetzt zur Ausgabe von Daten nur noch den <<-Operator.

4. Überladen Sie in der Klasse Fahrzeug den Vergleichsoperator operator<(). Dieser soll den Wert *true* liefern, falls die bisher zurückgelegte Gesamtstrecke vom aktuellen Objekt kleiner als die vom Vergleichsobjekt ist.

5. Verbieten Sie den Copy-Konstruktor, benutzen Sie hierzu *delete*. Überdenken Sie, wieso dies sinnvoll ist. Wo wird er implizit aufgerufen? Was bedeutet dies in diesem Szenario? Definieren Sie den Zuweisungsoperator (operator=()), so dass nur die Stammdaten (Daten die bei der Erstellung festgelegt werden) kopiert werden. Können Sie alle Daten kopieren? Was müssen Sie bei der ID beachten? Was würde passieren, wenn Sie keine eigene Definition des Operators erstellen würden? Was passiert, wenn Sie Elemente der Unterklasse zuweisen? Machen Sie sich den Unterschied zwischen Copy-Konstruktor und Zuweisungsoperator klar. Es geht hier nur um das prinzipielle Verständnis der Funktion des Zuweisungsoperators. Die genaue Implementierung des Zuweisungsoperators spielt für die weitere Aufgabe keine Rolle.

6. Testen Sie alle in dieser Aufgabe neu erstellten Operatoren in einer Funktion vAufgabe_3().

5 Aufgabenblock 2: Erweiterung des Verkehrssystems

5.1 Motivation

In diesem Aufgabenblock werden folgende Punkte betrachtet:

- Erweiterung einer Klassenhierarchie,
- abstrakte Klassen und rein virtuelle Methoden,
- Aufzählungsklasse,
- Unterscheidung und Nutzung unique_ptr, shared_ptr, weak_ptr,
- Templateklassen nutzen und erstellen,
- Exception handling und eigene Exceptionklassen,
- Nutzung einer externen Bibliothek,
- Erzeugung gleichverteilter Zufallszahlen.

In diesem zweiten Aufgabenblock wird die Klassenhierarchie um eine Klasse Weg erweitert. Da diese Klasse einige Eigenschaften mit Fahrzeugen gemeinsam hat (Name, Simulationszeit, Simulationsfunktion, Ausgabefunktion usw.), ist es sinnvoll, die Klassenhierarchie um eine abstrakte Oberklasse zu erweitern und sowohl Weg, als auch $Fahrzeug$ von dieser Klasse abzuleiten. Die gemeinsamen Dienste werden dann in diese abstrakte Oberklasse verlagert. Beim letzten Aufgabenpunkt wird noch eine Klasse $Kreuzung$ eingeführt, die ebenfalls von der abstrakten Oberklasse abzuleiten ist. Dies ist eine bei der objektorientierten Programmierung häufig auftretende Situation.

Ein Weg verwaltet eine Liste von Fahrzeugen und kann simuliert werden, indem alle auf dem Weg befindlichen Fahrzeuge simuliert werden.

Für die Berechnung der Strecke, die ein Fahrzeug in einem Simulationsschritt zurücklegt, wird eine neue Klasse erstellt, die ein Verhaltensmuster implementiert. Ein Verhaltensmuster ist ein Beispiel von einem Entwurfsmuster (*design pattern*). Jedes Fahrzeug besitzt eine Instanz dieser Klasse und kann in seinem Simulationsschritt diese Instanz fragen, wie weit es fahren darf. Auftretende Sondersituationen (parkendes Fahrzeug

fährt los, fahrendes Fahrzeug kommt am Ende des Weges an) werden durch Ausnahmebehandlung (Exceptions) abgehandelt. Um die Simulation etwas anschaulicher zu machen, wird eine Bibliothek mit Funktionen zur grafischen Darstellung verwendet.

Oft wiederkehrende Datenstrukturen und Algorithmen können durch Templates allgemein beschrieben werden. Die *STL* stellt eine Fülle solcher vorgefertigten Strukturen bereit. Einige davon sollen hier benutzt werden. Schließlich soll für eine spezielle Listenart (verzögerte Aktualisierung) ein eigenes Template erstellt werden.

Um sich einen Überblick zu verschaffen, lesen Sie den zweiten Aufgabenblock zunächst komplett durch.

5.2 Kopieren des Projektes

1. Erzeugen Sie ein neues leeres Projekt mit dem Namen Aufgabenblock_2. Kopieren Sie alle Sourcen (nur *.h und *.cpp Dateien) aus Aufgabenblock_1 und machen Sie diese Dateien dem neuen Projekt bekannt (s. Kapitel 3.2.3).

2. Ändern Sie den C++-Dialekt Ihres Projektes auf C++-17. Dazu klicken Sie mit der rechten Maustaste auf Ihr Projekt und wählen "Properties" aus. Hier wählen Sie nun "$C/C++\ Build \rightarrow Settings$" aus. Auf der rechten Seite wählen Sie nun den Reiter "Tool Settings". Hier öffnen Sie nun "$GCC\ C++\ Compiler \rightarrow Dialect$". Im rechten Bereich wählen Sie nun unter "Language standard" "ISO C++ 17 (-std=c++17)".

5.3 Simulationsobjekte und Wege

1. Als erstes soll eine neue abstrakte Oberklasse *Simulationsobjekt* geschaffen werden, welche die gemeinsamen Eigenschaften von $Fahrzeug$ und einer neuen Klasse Weg zusammenfasst. Fahrzeuge und Wege sind Simulationsobjekte, die einen Namen, eine ID und eine lokale Zeit besitzen. Sie können simuliert und ausgegeben werden. Integrieren Sie $Fahrzeug$ in diese neue Klassenhierarchie, indem Sie die Variablen für Name, ID und Simulationszeit sowie alle Funktionen zur gemeinsamen Nutzung von $Fahrzeug$ und Weg aus der Klasse $Fahrzeug$ in die Klasse *Simulationsobjekt* übertragen. Verschieben Sie die Ausgabe im Destruktor in die neue Klasse, um weiterhin das richtige Löschen der Objekte kontrollieren zu können. Löschen Sie die jetzt überflüssigen Variablen und Funktionen in $Fahrzeug$.

   Beachten Sie, dass die Variablen und Methoden in $Fahrzeug$ angepasst bzw. gelöscht werden müssen. *Simulationsobjekt* ist eine abstrakte Klasse, besitzt also mindestens eine rein virtuelle Methode. Überlegen Sie, welche Funktion hierzu am besten geeignet ist. Welche Methoden/Variablen müssen *private*, *protected* oder *public* deklariert werden? Gibt es weitere Methoden, die virtuell oder rein virtuell deklariert werden können oder müssen? Verbieten Sie auch hier den Copy-Konstruktor, um fehlerhafte Elemente durch Kopien zu vermeiden. Wie müssen

Sie den Zuweisungsoperator anpassen? Implementieren Sie einen Vergleichsoperator (operator==()), der genau dann *true* liefert, wenn die IDs übereinstimmen.

Um Codeduplizierung zu vermeiden, sollen bei der Ausgabe die entsprechenden vAusgeben-Methoden der übergeordneten Klassen mitbenutzt werden. So soll etwa vAusgeben in *Fahrrad* zunächst die Methode von *Fahrzeug* aufrufen, diese zunächst die Methode von *Simulationsobjekt*. Simulationsobjekt::vAusgeben soll nur die ID und den Namen des Objekts ausgeben. Aufgrund dieses Aufbaus reicht es für den Ausgabeoperator aus, diesen nur in der Klasse *Simulationsobjekt* zu definieren. Dieses Prinzip gilt auch für die Konstruktoren: Die Konstruktoren sollten jeweils die passenden Konstruktoren der Oberklasse aufrufen und nur die eigenen Variablen der Klasse setzen. Der Aufruf des Konstruktors der Oberklasse erfolgt in einer Initialisierungsliste.

2. Richten Sie die Klasse *Weg* als Unterklasse von *Simulationsobjekt* ein. Wege haben zusätzlich zu den geerbten Eigenschaften eine Länge in km (p_dLaenge), eine Liste von Fahrzeugen (p_pFahrzeuge), welche sich aktuell auf dem Weg befinden, und eine maximal zulässige Geschwindigkeit (p_eTempolimit). Die Liste beinhaltet unique_ptr auf Fahrzeuge. Zur Implementierung der Liste benutzen Sie den Container *list* aus der *STL*.

   Es soll für Wege drei unterschiedliche Kategorien (Innerorts, Landstraße und Autobahn) mit unterschiedlichem Tempolimit ($50km/h$, $100km/h$ und Unbegrenzt) geben. Für die Autobahngeschwindigkeit können Sie die Konstante std::numeric_limits<int>::max() verwenden. Definieren Sie dazu in Tempolimit.h einen eigenen Datentyp *Tempolimit* als Aufzählungsklasse (*enumclass*) und eine Konvertierungsfunktion getTempolimit in *Weg*, die für p_eTempolimit die entsprechende Geschwindigkeit als *double* zurückgibt.

   Weg soll einen Standardkonstruktor und einen Konstruktor mit Namen und Länge des Weges, sowie optionalem Tempolimit (default unbegrenzt) als Parameter haben. Außerdem soll die Funktion vSimulieren() so implementiert werden, dass beim Aufruf alle auf dem Weg befindlichen Fahrzeuge simuliert werden. Setzen Sie hierzu eine Range-basierte Schleife ein, die über die gesamte *list* iteriert.

   Beachte: Wenn zwei Klassen jeweils Variablen der anderen als Element enthalten (hier enthält ein Weg Instanzen der Klasse Fahrzeug), können Sie nicht in beiden Headerdateien jeweils die andere Headerdatei inkludieren, da dies zu einer Rekursion führen würde. Es reicht, in den Headerdateien jeweils die andere Klasse zu deklarieren, also einfach class Fahrzeug; bzw. class Weg; einzufügen. In den cpp-Dateien müssen aber dann die entsprechenden Headerdateien eingebunden werden, da dort die Methoden benötigt werden. Um allgemein Probleme mit zirkulären Abhängigkeiten (circular dependencies) in Headerdateien zu vermeiden, kann man meistens folgende Faustregel anwenden: Nur, wenn von einer Klasse geerbt wird, ist es notwending die andere Header-Datei in der Header-Datei einzubinden. Bei allen anderen Klassen reicht in der Headerdatei eine Deklaration. Für cpp-Dateien

gilt das nicht: Dort müssen alle benutzten Headerdateien eingebunden werden, um die Schnittstellen der Funktionen bereitzustellen.

Implementieren Sie eine Funktion vAusgeben für Weg, damit der überladene Ausgabeoperator verwendet werden kann. Die Funktion soll die Implementierung von *Simulationsobjekt* für ID und Name verwenden und selbst die Länge des Weges und in Klammern die Namen der auf dem Weg befindlichen Fahrzeuge ausgeben. Definieren Sie auch hier eine Klassenfunktion vKopf() zur Ausgabe einer Überschrift für Wege wie folgt:

```
ID | Name        | Laenge   | Fahrzeuge
-----------------------------------------------
  0 weg          :      100   ( )
```

3. Testen Sie Ihr altes Hauptprogramm. Es sollte noch unverändert funktionieren. In vAufgabe_4() testen Sie zusätzlich die neue Klasse Weg, indem Sie einen Weg erzeugen und ihn mit dem <<-Operator auf die Standardausgabe ausgeben.

5.4 Parkende und fahrende Fahrzeuge

1. Damit man für ein Fahrzeug verschiedene Verhaltensweisen realisieren kann, wird die Klasse $Fahrzeug$ um eine Membervariable p_pVerhalten erweitert, die eine Instanz der im Folgenden noch zu implementierenden Klasse $Verhalten$ verwaltet. Durch Austausch dieses Objektes kann das Verhalten des Fahrzeugs verändert werden, ohne ein neues Fahrzeug erstellen zu müssen. Überlegen Sie, welche Art von Smartpointer für p_pVerhalten gewählt werden sollte.

   Unter Verhalten verstehen wir, dass zwischen fahrenden und parkenden Fahrzeugen unterschieden werden kann.

   Da das Verhalten u.a. vom jeweiligen Weg abhängt, bekommt die Klasse $Verhalten$ einen Konstruktor, der eine Referenz auf einen Weg als Parameter bekommt und speichert. Weiterhin soll eine Funktion double dStrecke(Fahrzeug& aFzg, double dZeitIntervall) angeboten werden, die ermittelt, wie weit ein Fahrzeug innerhalb des übergebenen Zeitintervalls fahren kann, ohne das Wegende zu überschreiten. Die bisherige Berechnung der aktuellen Teilstrecke in Fahrzeug::vSimulieren wird durch den Aufruf der Funktion dStrecke ersetzt. Zum lesenden Zugriff auf private Variablen von Weg und Fahrzeug müssen ggf. neue Getter-Funktionen geschrieben werden. Beachten Sie, dass dStrecke in jedem Simulationsschritt nur einmal aufgerufen und das Zwischenergebnis lokal zwischengespeichert wird.

   Bei jedem Start eines Fahrzeugs auf einem neuen Weg soll nun eine Instanz von $Verhalten$ erzeugt und in $Fahrzeug$ gespeichert werden.Dies geschieht am besten durch eine neue Memberfunktion Fahrzeug::vNeueStrecke(Weg&), die ein geeignetes Objekt erzeugt und in p_pVerhalten speichert. Was passiert mit der alten Instanz, wenn das Fahrzeug auf einen neuen Weg gesetzt wird?

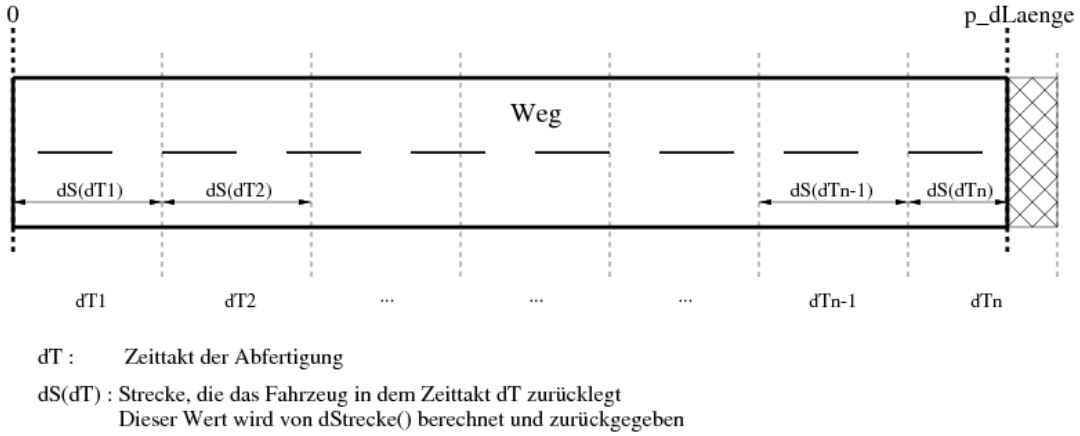


Figure 5.1: Funktionsweise dStrecke()

Da Fahrzeuge jetzt nacheinander auf verschiedenen Wegen fahren sollen, führen wir hier eine zusätzliche Membervariable p_dAbschnittStrecke in Fahrzeug ein. Diese speichert immer nur die auf dem aktuellen Weg zurückgelegte Strecke. Sie wird in gleicher Weise wie bisher p_dGesamtStrecke aktualisiert und beim Betreten des Weges auf 0 gesetzt.Fügen Sie diese Variable Ihren Berechnungen und Ausgaben für $Fahrzeug$ hinzu. Die Variable p_dGesamtStrecke soll weiterhin gepflegt werden.

Da es zurzeit noch keine Einschränkungen für die Fahrzeuge gibt, soll die Funktion dStrecke, wie in Figure 5.1 gezeigt, die auf Grundlage der übergebenen Zeitspanne fahrbare Strecke zurückliefern, falls dadurch die Weglänge noch nicht überschritten wird ($dT_1 \ldots dT_{n-1}$). Im Zeittakt dT_n soll nur die bis zum Wegende verbleibende Strecke zurückgegeben werden, womit das Fahrzeug genau am Ende des Weges ankommt. Im letzen Zeittakt dT_{n+1} wird dann erkannt, dass das Fahrzeug am Ende des Weges steht. Zunächst soll das Programm hier nur eine entsprechende Meldung ausgeben, dass es am Ende des Weges angekommen ist.

2. Schreiben Sie nun eine Funktion Weg::vAnnahme(unique_ptr<Fahrzeug>), die ein Fahrzeug auf dem Weg annimmt. Dazu muss es in die Liste der Fahrzeuge eingetragen werden. Da ein *unique_ptr* nicht kopiert werden kann, muss der Pointer auf das Fahrzeug mit *move* verschoben werden. Damit man die eingetragenen Fahrzeuge auch sehen kann, werden diese in Klammern an die Ausgabe des Weges angehängt. Weiterhin muss dem Fahrzeug signalisiert werden, dass es sich auf einer neuen Strecke befindet.

   Beispiel:

```
ID | Name       | Laenge  | Fahrzeuge
-----------------------------------------------
  0 weg         :     100  ( BMW Audi BMX )
```

3. Testen Sie Ihre neue Klasse in vAufgabe_5(), indem Sie einen Weg und drei Fahrzeuge erzeugen, diese auf den Weg setzen und den Weg simulieren.

4. Der Simulation sollen nun parkende Fahrzeuge hinzugefügt werden. Parkende Fahrzeuge benötigen ein anderes Verhaltensmuster, da diese sich nicht fortbewegen.

   Erweitern Sie dazu die Klasse $Verhalten$ zu einer Klassenhierarchie, wobei Sie zwei Klassen $Fahren$ und $Parken$ von $Verhalten$ ableiten.

   $Verhalten$ soll als *abstrakte Oberklasse* implementiert werden. $Fahren$ soll funktionieren wie vorher $Verhalten$. Implementierten Sie daher für $Fahren$ den Code nicht doppelt, sondern übernehmen diesen. Die Klasse $Parken$ hat einen Konstruktor, der zusätzlich zum Weg den Startzeitpunkt des Fahrzeugs übergeben bekommt. Parken::dStrecke() liefert bis zum Erreichen des Startzeitpunktes den Wert 0.0 zurück. Wenn die Startzeit erreicht wurde, soll das Programm auch hier zunächst eine entsprechende Meldung ausgeben.

   Auf einem Weg sollen sich sowohl parkende als auch fahrende Fahrzeuge befinden können. Um beide zu unterscheiden, soll die Funktion vAnnahme(unique_ptr<Fahrzeug>) durch eine weitere Funktion vAnnahme(unique_ptr<Fahrzeug>, double) überladen werden. Bekommt sie nur einen Zeiger auf Fahrzeug als Argument, dann nimmt sie wie bisher ein fahrendes Fahrzeug an.Wird jedoch ein Zeiger auf Fahrzeug *und* eine Startzeit übergeben, nimmt sie ein parkendes Fahrzeug an. Alle Fahrzeuge sollen weiterhin zusammen in der vorhandenen Liste verwaltet werden.

   Überladen Sie entsprechend auch die Funktion Fahrzeug::vNeueStrecke. Fügen Sie fahrende Fahrzeuge hinten in die Liste an, parkende Fahrzeuge vorne. Diese Eigenschaft werden wir später noch benötigen.

5. Modifizieren Sie vAufgabe_5 mehrfach so, dass das Programm beim Starten bzw. am Streckenende entsprechende Meldungen ausgibt. Alternativ können Sie dies auch mit Hilfe des Debuggers testen.

5.5 Losfahren, Streckenende (Exception Handling)

1. Sie haben bisher an zwei Stellen im Programm nur eine Meldung für das Losfahren und das Streckenende Stattdessen soll nun jeweils eine Ausnahme (*Exception*) geworfen werden (*throw*), die dann in der Simulationsmethode des Weges aufgefangen (*catch*) und abgearbeitet werden kann. Da Sie zwei verschiedene Arten von Ausnahmen werfen, ist es vernünftig, eine Klassenhierarchie für diese Ausnahmefälle zu erstellen.

   Leiten Sie dazu zwei Klassen $Losfahren$ und $Streckenende$ von einer abstrakten Klasse $Fahrausnahme$ ab. Zusätzlich leiten sie die Klasse $Fahrausnahme$ von der Klasse *exception* aus der der C++ Standardbibliothek ab. Überlegen Sie sich was das für Vorteile mitbringt.$Fahrausnahme$ soll eine Referenz auf $Fahrzeug$ und eine Referenz auf Weg als Membervariable besitzen. Diese speichern jeweils das Fahrzeug und den Weg, bei denen die Ausnahme aufgetreten sind.Implementieren Sie auch einen entsprechenden Konstruktor, der die beiden Referenzen setzt. Weiterhin hat die Klasse eine rein virtuelle Funktion vBearbeiten(). Geben Sie in den

beiden Bearbeitungsmethoden der Unterklassen vorerst nur Fahrzeug, Weg und Art der Ausnahme aus.

Beim Auftreten der Ausnahmen (bisher Ausgaben) sollen nun die entsprechenden Objekte geworfen und in der Simulationsroutine des Weges aufgefangen werden. Nachdem ein Ausnahmeobjekt gefangen wurde, wird für dieses einfach nur die Bearbeitungsfunktion vBearbeiten() ausgeführt.

Beachte: Fangen Sie beide Ausnahmen mit nur **einem** *catch*-Block. Wieso ist das möglich?

2. Über die Klasse $Verhalten$ haben Fahrzeuge und die davon abgeleiteten Klassen Kenntnis vom befahrenen Weg. Berücksichtigen Sie die Maximalgeschwindigkeit (Weg::p_eTempolimit), die für den befahrenen Weg, aber nur für PKW gilt, indem Sie die Methode PKW::dGeschwindigkeit() entsprechend implementieren. Definieren Sie zum Testen einen Weg mit Tempolimit.

3. Testen Sie nun mit einer Funktion vAufgabe_6 die gerade implementierte Ausnahmebehandlung und das Tempolimit. Erzeugen Sie dazu zwei Wege (mindestens einer mit Tempolimit) und setzen Sie fahrende und parkende Fahrzeuge auf diese Wege und fertigen beide Wege ab.

   Beachte: Die Ausnahmen *Streckenende* und $Losfahren$ werden beim Erreichen des Wegendes bzw. des Startzeitpunkte bei jedem folgenden Simulationsschritt erneut geworfen. Die entsprechenden Meldungen kommen also mehrfach. Da wir noch keine Fahrzeuge von der Liste entfernen oder umsetzen, ist dieses Verhalten kein Fehler.

4. Aufgabe zur Nutzung des Debuggers:
   Kontrollieren Sie mit Hilfe des Debuggers, ob das Losfahren immer zum richtigen Zeitpunkt auftritt. Lassen Sie dazu ein Fahrzeug beim Zeitpunkt 3.0 losfahren. Überprüfen Sie den Startzeitpunkt einmal bei einem Zeittakt der globalen Zeit von 0.25 und einmal bei 0.3. Korrigieren Sie ggf. Ihren Code so, dass in beiden Fällen beim Zeitpunkt 3.0 losgefahren wird.

5.6 Grafische Ausgabe

1. Um die Simulation anschaulicher zu machen, soll sie nun grafisch dargestellt werden. Dazu wurde ein Client/Server-Modell entwickelt, bei dem der Server vom Client über TCP/IP Kommandos empfängt und diese dann in eine grafische Darstellung umsetzt.

   Die Grafikschnittstelle wird Ihnen durch die Klassen *SimuClient* und *SimuClientSocket* zur Verfügung gestellt. Der grafische Server wird über die Java-Datei *SimuServer.jar* zur Verfügung gestellt. Um die Grafikschnittstelle nutzen zu können, kopieren Sie zunächst die erforderlichen Dateien (*SimuClient.h*, *Simuclient.cpp*, *SimuClientSocket.h*, *SimuClientSocket.cpp*, *SimuServer.jar*) in Ihr Projektverzeichnis (Verzeichnis

mit cpp/h-Dateien, z.B. *Aufgabenblock_2*) in Ihrem Eclipse-Workspace. Die Dateien finden Sie in den Vorgabedateien in Moodle.

Grafikschnittstelle

Die Funktionen der Grafikbibliothek arbeiten nur mit den übergebenen Werten, kennen also keine anderen Daten Ihres Projektes. Die Werte werden beim Aufruf aber auf syntaktische und semantische Plausibilität geprüft. Das bedeutet:

a) Zahlenwerte müssen in einem sinnvollen Wertebereich liegen.

b) Die Namen dürfen nur Buchstaben, Ziffern und _ enthalten, insbesondere keine Leerzeichen.

c) Fahrzeuge können nur auf Straßen gezeichnet werden, die vorher durch zwei Wege definiert wurden.

Die Grafikschnittstelle stellt folgende Funktionen zur Verfügung:

- bInitialisiereGrafik(int GroesseX, int GroesseY);

  Mit dieser Funktion stellen Sie eine Verbindung zum Grafikserver her und initialiseren Sie die Größe des Fensters. Die Variablen *GroesseX* und *GroesseY* bestimmen die Größe der Grafikdarstellung. Verwenden Sie hier z.B. folgende Werte: GroesseX=800; GroesseY= 500

  ```
  bInitialisiereGrafik(800, 500);
  ```

- vSetzeZeit(double Zeit)

  Mit dieser Funktion können Sie die globale Zeit in der Titelzeile des Ausgabefensters anzeigen lassen.

- bZeichneStrasse(string NameHin, string NameRueck, int Laenge, int AnzahlKoord, int[] Koordinaten)

  Diese Funktion zeichnet eine Straße, die aus den beiden durch ihren Namen identifizierten Wegen besteht. Die Straße soll mit einer Reihe von Koordinaten dargestellt werden. Der Verlauf der Straße wird durch einen Polygonzug mit mindestens 2 Punkten (Gerade) skizziert. Die Koordinaten der Polygonpunkte werden im Array *Koordinaten* übergeben. Das Array enthält *AnzahlKoord* X/Y-Paare. Für eine gerade Straße benutzen Sie für Koordinaten z.B. die Werte

  `{ 700, 250, 100, 250 }.`

  Beachte:

  a) Achten Sie darauf, dass die X-/Y-Koordinatenwerte innerhalb der vorher definierten (bInitialisiereGrafik()) Grenzen liegen.

  b) Das Array muss genau (2 x AnzahlKoord) *int*-Elemente enthalten.

  c) Diese Funktion darf für jede Straße nur einmal aufgerufen werden.

- bZeichnePKW(string PKWName, string WegName, double RelPosition, double KmH, double Tank)
- bZeichneFahrrad(string FahrradName, string WegName, double RelPosition, double KmH)

  Diese Funktionen zeichnen jeweils eine symbolische Darstellung des PKW/Fahrrads auf dem durch seinen Namen identifizierten Weg. Die relativ zur Weglänge zurückgelegte Strecke (Wert zwischen 0 und 1) wird mit $RelPosition$ angegeben. Dem Parameter KmH wird der Wert aus der Funktion dGeschwindigkeit(), dem Parameter $Tank$ der aktuelle Tankinhalt übergeben.
- BeendeGrafik()

  Mit dieser Funktion wird die Verbindung zum Grafikserver getrennt, das Fenster wird automatisch geschlossen.
- void vSleep(int zeit_ms); Mit dieser Funktion wird die weitere Programmausführung um $zeit_ms$ Millisekunden verzögert.

2. Erweitern Sie vAufgabe_6() so, dass die Grafikausgabe getestet werden kann. Fügen sie den Header SimuClient.h in ihrer main.cpp ein. Wählen Sie als Länge der beiden Wege jeweils $500km$ und fassen Sie diese grafisch zu einer Straße zusammen (Hin- und Rückweg).

3. Um beim Zeichen, abhängig vom Fahrzeugobjekt-Typ, die korrekte Zeichenfunktion aufzurufen, soll für PKW und Fahrrad eine Funktion vZeichnen(const Weg&) const implementiert werden. Dazu wird in $Fahrzeug$ die Funktion virtuell deklariert und in der jeweiligen Unterklasse überschrieben. Die Funktion bekommt den Weg, auf dem das Fahrzeug gezeichnet werden soll, als Referenz übergeben und ruft dann die passende Zeichenfunktion (s.o.) auf.

4. Lassen Sie die Fahrzeuge nach jeder Simulation in Weg zeichnen.

5. Führen Sie Ihre Simulation aus. Um die Simulation besser verfolgen zu können, rufen Sie die Funktion $vSleep$ in Ihrer Schleife auf. Je nach Rechenleistung des verwendeten Computers können Sie die Verzögerung anpassen ($100ms$).

5.7 Verzögertes Update (Template)

1. Wenn die Ausnahmesituationen aus der vorigen Teilaufgabe eintreten, soll nun auch die entsprechende Aktion ausgeführt werden:

   - **Fahrzeug startet**: Die parkenden Fahrzeuge sollen vorne, die fahrenden Fahrzeuge hinten in der Liste stehen. Benutzen Sie daher für die Aufnahme der Fahrzeuge entsprechend push_front() bzw. push_back(). Die Liste hat dann folgenden Aufbau:

parkend...amweitestenaufWegfahrend...amWeganfangfahrend

Zum Starten muss das parkende Fahrzeug aus der Liste entfernt und als fahrendes Fahrzeug sofort wieder gespeichert werden. Schreiben Sie zum Löschen der Fahrzeuge aus der Liste eine Funktion unique_ptr<Fahrzeug> Weg::pAbgabe(const Fahrzeug&). In dieser Funktion suchen Sie in der gespeicherten list<unique_ptr<Fahrzeug>> nach dem übergebenen Fahrzeug. Benutzen Sie zum Vergleich der Fahrzeuge den operator==. Beachten Sie den Sonderfall eines Nullpointers.Wenn Sie das Fahrzeug in der Liste gefunden haben, verschieben (*move*) Sie den unique_ptr in eine lokale Variable und löschen dann das Fahrzeug aus der Liste. Anschließend können Sie den lokal gespeicherten Pointer zurückgeben. Mit den Funktionen Weg::pAbgabe und Weg::vAnnahme kann nun die Bearbeitungsfunktion von Losfahren entsprechend angepasst werden.

- **Fahrzeug kommt am Wegende an**: Passen Sie die Bearbeitungsfunktion von *Streckenende* so an, dass am Streckenende ankommende Fahrzeuge aus der Liste entfernt werden.

Testen Sie diese Funktionen: Führen Sie dazu vAufgabe_6() nochmals aus. Je nach Implementierung sollte es nun zu einer Fehlermeldung bei der Simulation des Weges kommen, da der Iterator über die Fahrzeuge nach dem Löschen bzw. Umsetzen eines Fahrzeugs nicht mehr definiert ist und so der Nachfolger nicht mehr bestimmt werden kann. Ebenso ist es möglich, dass Simulationsschritte von Fahrzeugen fehlen, da in der Liste durch Umsetzen von Elementen diese nach hinten oder vorne gerutscht sind.

Um diese Probleme zu vermeiden, soll eine allgemeine Templateklasse $VListe$ im Namensbereich *vertagt* implementiert werden, die das Einfügen und Löschen von Elementen bis zum Aufruf einer Methode vAktualisieren aufschiebt. Zur Vereinfachung geben wir Ihnen das Gerüst der Templateklassen in Form der Dateien vertagt_liste.h und vertagt_aktion.h vor (Die Dateien finden Sie in den Vorgabedateien in Moodle). Fügen Sie diese Dateien ihrem Projekt hinzu und ergänzen Sie in diesen Dateien alle Bereiche, die mit [...] gekennzeichnet sind. Um sicherzugehen, dass Sie alle diese Bereich gefunden haben, können Sie die Dateien in Eclipse mit der Tastenkombination $Strg + F$ durchsuchen.

Die Klasse $VListe$ hat zwei Datenelemente:

a) die Liste mit den eigentlichen Objekten (list<T> p_objekte) mit den zu speichernden Elementen vom Templatetyp T

b) eine Liste zum Zwischenspeichern der noch auszuführenden Aktionen (list<unique_ptr<Aktion>> p_aktionen).

Wir unterscheiden bei der Liste zwischen Lese- und Schreibfunktionen. Leseoperationen können sofort auf der eigentlichen Liste durchgeführt werden, Schreiboperationen müssen als Aktion zwischengespeichert werden.

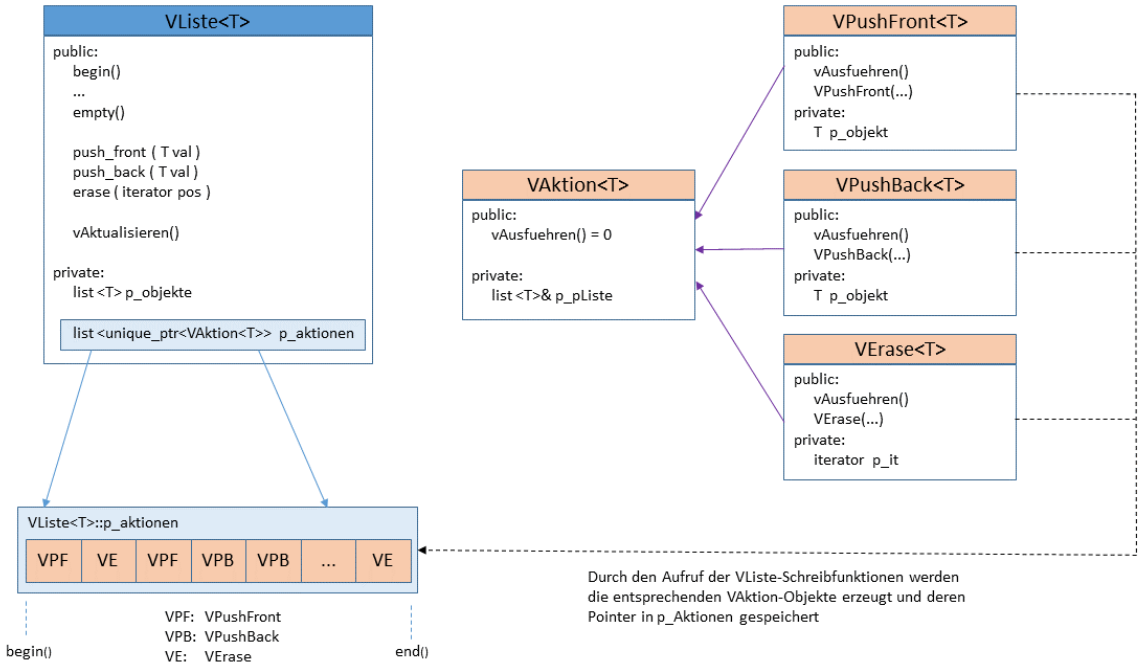


Figure 5.2: Prinzipielle Funktionsweise von *VListe*

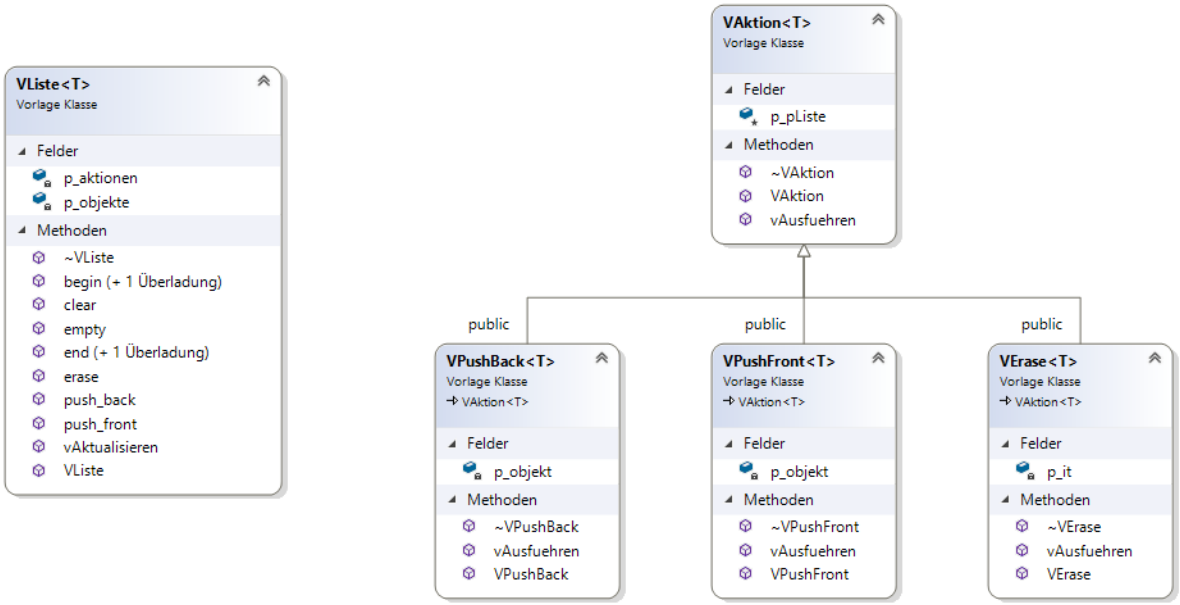


Figure 5.3: Klassenhierarchie von *VListe*

Für die Schreib-Aktionen wird eine Klassenhierarchie mit einer abstrakten Oberklasse *VAktion* angelegt, siehe Figure 5.3, die lediglich die Funktion vAusfuehren() und eine Referenz auf die zu bearbeitende Liste (p_pListe) beinhaltet. Für jede Schreibfunktion wird eine zugehörige Unterklasse von *VAktion*, also *VPushFront*, *VPushBack* und *VErase* abgeleitet. Dem Konstruktor der Unterklassen wird der jeweilige Parameter der Schreibfunktion und eine Referenz auf die eigentliche Liste übergeben, da sonst kein Zugriff auf die Liste möglich wäre. Die in den Unterklassen überschriebene Funktion vAusfuehren() führt dann die eigentliche Operation aus.

Die Funktion vAktualisieren() der *VListe* durchläuft die Liste der anstehenden Aktionen und arbeitet jedes Element mit der Methode vAusfuehren() ab. Beachten Sie, dass die benutzten Objekte aus der Aktionsliste entfernt werden. Die Funktionsweise von *VListe* ist in Figure 5.2 nochmal schematisch dargestellt.

Folgende Funktionen sollen für die vertagte Liste implementiert werden:

- iterator begin(): gibt einen Iterator zurück, der auf das erste Element zeigt
- iterator end(): gibt einen Iterator zurück, der hinter das letzte Element zeigt
- bool empty(): gibt zurück, ob das Objekt keine Elemente enthält
- void push_front(T obj): fügt *obj* vor dem ersten Element ein
- void push_back(T obj): fügt *obj* am Ende ein
- void erase(iterator it): löscht das Element an Position *it*
- void vAktualisieren(): aktualisiert p_objekte
- void clear(): aktualisiert die Liste und löscht dann alle Elemente in p_objekte

2. Testen Sie in vAufgabe_6a() Ihre neue VListe, indem Sie eine vertagt::VListe von ganzzahligen Zufallszahlen zwischen 1 und 10 erzeugen. Folgende Aktionen sollen nacheinander auf der Liste ausgeführt werden:

   - Liste ausgeben
   - innerhalb einer Schleife alle Elemente > 5 mit erase() löschen
   - Liste wieder ausgeben (da vAktualisieren() noch nicht ausgeführt wurde, sollte hier dieselbe Ausgabe erfolgen)
   - vAktualisieren() auf die *Liste* anwenden
   - Liste nochmal ausgeben (jetzt sollte sich die *Liste* geändert haben).
   - Zum Schluss fügen Sie am Anfang und am Ende der Liste noch zwei beliebige (verschiedene) Zahlen ein und geben die Liste zur Kontrolle nochmal aus.

   Tipp: Eine ganzzahlige Zufallszahl zwischen a und b ermitteln Sie wie folgt:

```
#include <random>
```

```
static std::mt19937 device(seed);
std::uniform_int_distribution<int> dist(a, b);
int zuf = dist(device);
```

Kurz: Wir setzen einen pseudo-random Generator ein, um Zufallszahlen zu erzeugen. In Zeile 1 erzeugen wir eine statische Variable (device), die den benutzten Algorithmus (mt19937, d.h. Mersenne_Twister_Engine) und den Initialwert (seed) festlegt. In Zeile 2 erzeugen wir eine Verteilung (Gleichverteilung mit Integerzahlen) und legen das gewünschte Intervall [a,b] fest. In Zeile 3 wird mit Hilfe des Generators und der Verteilung eine entsprechende Zufallszahl bestimmt. Mehrere Zufallszahlen werden durch wiederholten Aufruf von dist(device) erzeugt. Für Details schauen Sie sich die Bibliothek <random> an. **Beachte**: Wir wollen reproduzierbare Ergebnisse erhalten, daher wählen wir einen festen Wert für seed (0).

3. Ersetzen Sie nun bei der Fahrzeugliste in Weg die einfache Liste durch eine entsprechende vertagt::VListe. Sie sollten vor und nach jeder Simulation von Weg die Liste aktualisieren. Beachten Sie, dass Sie gegenüber der Nutzung des Templates für Integer bei der Nutzung mit unique_ptr noch Anpassungen zum Besitzwechsel durchführen müssen. Testen Sie nun nochmal vAufgabe_6(). Die eventuell aufgetretene Speicherschutzverletzung sollte nun nicht mehr auftreten. Achten Sie darauf, ob die Fahrzeuge in der Liste richtig umgesetzt und am Ende des Weges aus der Liste gelöscht werden. Kontrollieren Sie, ob auch die entsprechenden Fahrzeugobjekte automatisch gelöscht werden.

5.8 Aufbau des Verkehrssystems

1. Bisher besteht das Verkehrsnetz nur aus isolierten Wegen und darauf fahrenden Fahrzeugen. Die Wege sollen nun mittels Kreuzungen verbunden werden. Da die Infrastruktur gut ausgebaut ist, soll es keine Einbahnstraßen geben und eine Straße jeweils aus Hin- und Rückweg bestehen.

   Erweitern Sie die Klassenhierarchie um die Klasse $Kreuzung$ die von $Simulationsobjekt$ abgeleitet wird. Die Klasse $Kreuzung$ speichert in einer Liste p_pWege alle von ihr wegführenden Wege und bekommt eine Membervariable p_dTankstelle. Die Variable speichert das Volumen, das einer Kreuzung zum Auftanken zur Verfügung steht. Überfährt ein PKW eine Kreuzung mit Tankstelle (p_dTankstelle > 0.0), wird er vollgetankt und p_dTankstelle um die entsprechende Menge reduziert, so lange, bis die Tankstelle leer ist. Auch hier gibt es zur Vereinfachung eine Reserve, so dass auch der letzte PKW volltanken kann.

   Schreiben Sie eine statische Methode Kreuzung::vVerbinde(...), welcher der Namen des Hin- und Rückweges, die Weglänge, die Start und die Zielkreuzung sowie die gültige Geschwindigkeitsbegrenzung als Parameter übergeben wird. Diese Funktion muss statisch sein und beide Kreuzungen als Parameter übergeben bekommen.

Um die Kreuzungen verbinden zu können, müssen die Wege erzeugt und untereinander bekannt gemacht werden, d.h. ein Weg kennt seinen direkten Rückweg und er weiß, auf welche Kreuzung er führt. Welche Art von Smartpointer können Sie für die Elemente von p_pWege wählen?

Da Wege und Kreuzungen ggf. zyklisch wieder auf sich selbst verweisen, sollten die Variablen zur Speicherung der Zielkreuzung und des Rückweges vom Typ *weak_ptr* sein. Fügen Sie der Klasse *Weg* entsprechende Membervariablen hinzu. Die Zielkreuzung kann *const* gewählt werden, da sie sich nicht verändert. Dazu müssen Sie den Konstruktor von Weg dementsprechend anpassen und die Zielkreuzung mit *nullptr* initialisieren. Warum können Sie die Variable für den Rückweg nicht *const* setzen? Schreiben Sie für beide Variablen *Getter*, die einen *shared_ptr* auf das Objekt zurückgeben. Wichtig hierbei ist zu beachten, dass Sie die Funktion *lock* der Smartpointer benutzen.

Weiterhin soll in *Kreuzung* die Funktion vTanken(Fahrzeug&) implementiert werden, die ggf. das übergebene Fahrzeug volltankt und den Inhalt der Tankstelle aktualisiert.

Implementieren Sie eine Methode Kreuzung::vAnnahme(unique_ptr<Fahrzeug>, double), die Fahrzeuge annimmt und diese parkend auf den ersten abgehenden Weg stellt. Die Fahrzeuge sollen dabei ggf. aufgetankt werden. Nun implementieren Sie eine Funktion Kreuzung::vSimulieren(), die alle von dieser Kreuzung abgehenden Wege simuliert.

2. Beim Weiterleiten von Fahrzeugen sollen aus den wegführenden Wegen der Kreuzung zufällig einer ausgewählt werden. Dabei soll das Fahrzeug aber nicht dieselbe Straße zurückfahren, die es gekommen sind. Implementieren Sie dazu eine Funktion shared_ptr<Weg> Kreuzung::pZufaelligerWeg(Weg&), die als Parameter eine Referenz auf den Weg enthält, über den die Kreuzung erreicht wurde. Der Rückgabewert soll der ausgewählte Weg für das Fahrzeug sein. Bei einer "'Sackgasse"' muss natürlich der zurückführende Weg genommen werden.

   Bauen Sie diese Funktion nun in die Bearbeitungsfunktion von *Streckenende* ein, damit ein Fahrzeug, das am Ende des Weges angekommen ist, $fahrend$ auf einen so gefundenen Weg umgesetzt wird. Dabei soll auch getankt werden. Um die Bewegungen der Fahrzeuge besser verfolgen zu können, soll beim Umsetzen folgende Ausgabe erfolgen:

   ZEIT : [Zeitpunkt der Umsetzung]
   KREUZUNG : [Name der Kreuzung] [Inhalt der Tankstelle]
   WECHSEL : [Name alter Weg] → [Name neuer Weg]
   FAHRZEUG : [Daten des Fahrzeugs]

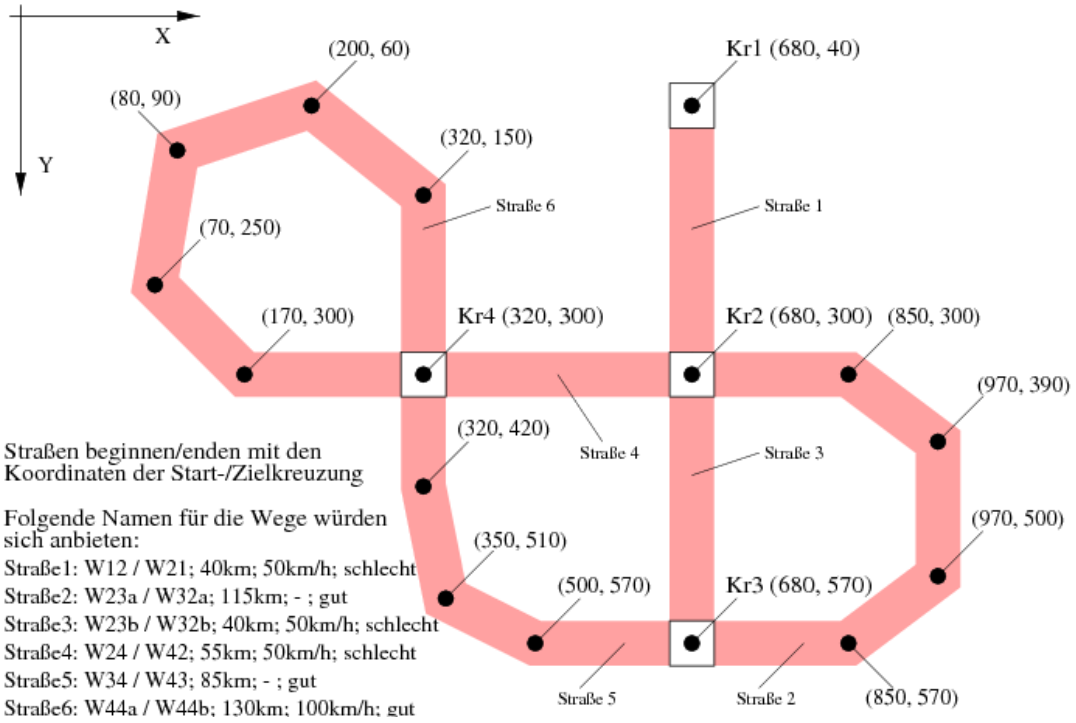


Figure 5.4: Koordinaten des Verkehrssystems

3. Testen Sie die bisherige Klasse $Kreuzung$ in vAufgabe_7(), indem Sie ein Verkehrsnetz entsprechend Figure 5.4 aufbauen und darin Fahrzeuge über die Kreuzung $Kr1$ annehmen. Für die grafische Darstellung der Kreuzung steht folgende Methode zur Verfügung:

   ```
   void bZeichneKreuzung(int posX, int posY);
   ```

   Diese Funktion zeichnet eine Kreuzung an den Koordinaten $posX$ und $posY$.

   Setzen Sie dann die Tankkapazität für Kreuzung $Kr2$ auf $1000l$ und simulieren Sie die Kreuzungen. Alle anderen Kreuzungen haben keine Tankstelle (Tankkapazität $= 0$).