Endbericht der
Projektgruppe
Fahrgemeinschaften
Bericht Nr. 1998/04

Universität
Stuttgart

Endbericht der Projektgruppe Fahrgemeinschaften

Herbert Heid
Daniela Nicklas
Alexander Porrmann
Thomas Schäffer
Volker Scholz

Betreuung

Prof. Dr. Volker Claus
Dipl.-Inf. Wolfgang Reissenberger
Dipl.-Inform. Friedhelm Buchholz
Dipl.-Math. Nicole Weicker

Abteilung Formale Konzepte
Fakultät Informatik
Universität Stuttgart

Prof. Dr. Volker Claus
Abteilung Formale Konzepte
Institut für Informatik
Universität Stuttgart

Breitwiesenstr. 20-22
D-70565 Stuttgart

Telefon:

0711-7816-300 (Prof. Dr. V. Claus)
0711-7816-301 (Sekretariat)
0711-7816-330 (FAX)

E-Mail: claus@informatik.uni-stuttgart.de

Inhaltsverzeichnis

Teil I

Entwicklungsprozeß

Kapitel 1

Einleitung

Das Studium der Informatik vermittelt dem Studierenden zwar einen großen Teil des nötigen Fachwissens, jedoch stellt das Berufsleben noch weitere Anforderungen an Informatikerinnen und Informatiker. Teamfähigkeit und Erfahrung spielen gerade bei der Mitarbeit an großen Software–Projekten eine wichtige Rolle. Hier verfolgt die Idee der Projektgruppe folgende Ausbildungsziele:

- Arbeiten im Team
- Analyse von Problemen, Strukturierung von Lösungen und gemeinsamer Entwurf geeigneter Systeme
- Selbständige Erarbeitung von Lösungsvorschlägen und deren Vorstellung und Verteidigung in einer Gruppe
- Übernahme von Verantwortung für die Lösung von Teilaufgaben und die Erstellung von Modulen
- Mitwirkung an einer umfassenden Dokumentation
- Erstellen eines Software–Produktes, das ein Einzelner innerhalb des vorgegebenen Zeitraumes unmöglich bewältigen kann
- Projekt–Planung und Kosten/Nutzen–Analyse
- Einsatz von Werkzeugen
- Persönlichkeitsbildung (Übernahme von Verantwortung, Selbstvertrauen, Verläßlichkeit, Rücksichtnahme, Durchsetzungsfähigkeit usw.)

Als wir Studenten uns im September 1996 zur ersten Vorbesprechung der Projektgruppe Fahrgemeinschaften trafen, wußte noch keiner von uns, was genau auf ihn zukommen würde. Gemein war uns das Interesse, in einer größeren Gruppe (einem Team) an einem Thema zu arbeiten. Eine Möglichkeit, die das Studium der Informatik an der Universität Stuttgart sonst nicht bietet. Die einzigen Lehrveranstaltungen, bei denen eine Zusammenarbeit mehrerer Studenten vorgesehen ist, nämlich Software-Praktikum und Fachpraktikum, haben eine Gruppengröße von zwei, maximal drei Studenten.

Dieser Bericht ist nicht nur die Dokumentation der nach einem Jahr Projektarbeit erzielten Ergebnisse, sondern schildert das Vorgehen der Projektgruppe Fahrgemeinschaften über verschiedene Zwischenergebnisse bis zum endgültigen Produkt Mobidick (Mobil durch intelligentes computerunterstütztes Kombinieren). Er gliedert sich im Wesentlichen in:

- eine Beschreibung einer Projektgruppe allgemein, sowie der Aufgabe der Projektgruppe Fahrgemeinschaften,
- den Projektablauf,
- die Bedienung von Mobidick,
- Überblick über die Programmierung von Mobidick
- Erweiterungsmöglichkeiten und kritische Betrachtungen,
- Rückblick der Projektgruppe auf Arbeitsorganisation und Ablauf und
- Anhänge zu Litaturhinweisen und zur Programmkodierung.

Die Mitglieder der Projektgruppe Fahrgemeinschaften waren für folgende Aufgaben zuständig:

- Herbert Heid: verantwortlich für Spezifikation, Bewertungsfunktion und Matching-Algorithmus
- Daniela Nicklas: verantwortlich für Anforderungsanalyse, Menüs und optimalen Einteilungsalgorithmus
- Alexander Porrmann: verantwortlich für Entwurf, Fürsorger, Lader und Speicherer und inkrementellen Einteilungsalgorithmus
- Thomas Schäffer: verantwortlich für Endbericht, RCS, Personenverwaltung und Einteilungsverwaltung
- Volker Scholz: verantwortlich für Zwischenbericht, GDF-Daten, Wegsuchealgorithmen

Unser Dank gilt Professor Volker Claus, der durch seinen Einsatz die Projektgruppe erst ermöglicht hat und auch das „unternehmerische Risiko“ trug. Besonders möchten wir unserem Betreuer Wolfgang Reissenberger danken, der sich stark in der Projektgruppenarbeit engagierte. Unserer weiterer Dank gilt Friedhelm Buchholz, der die Rolle des Kunden übernahm und uns bei der Konzeption der Algorithmen zur Seite stand, sowie Nicole Weicker, die unsere Seminarausarbeitungen betreute.

Kapitel 2

Projektgruppe Fahrgemeinschaften

2.1 Problemstellung

Im Rahmen der Projektgruppe soll das Programm Mobidick (Mobil durch intelligentes computerunterstütztes Kombinieren) entstehen, das ausgehend von Personen- und Verkehrsdaten Aufteilungen in Fahrgemeinschaften berechnet. Es handelt sich hierbei um einen Prototyp für ein System, das beispielsweise in einer Mitfahr- oder Mobilitätszentrale eingesetzt werden kann, um für große Personenmengen Fahrgemeinschaften zu bestimmen und zu verwalten.

Die Personendaten enthalten Informationen über Start-, Zielorte, Arbeitszeiten und Eigenschaften der Personen (z.B. Geschlecht, Raucher/Nichtraucher usw.). Außerdem können die Personen angeben, wie ihre Wunschfahrgemeinschaft aussehen sollte, d.h. welche Kriterien ihnen besonders wichtig sind (Umweg, Arbeitszeit und persönliche Zu- bzw. Abneigung gegenüber bestimmten Personen).

Die Verkehrsdaten (Stadtplan) liefern die Grundlage für die Berechnung der besten Routen mit den kürzesten Umwegen. Davon ausgehend soll nun eine Lösung gefunden werden, d.h. eine Einteilung in Fahrgescheinschaften, die optimal ist oder eine bestimmte Mindestgüte besitzt. Hierzu können verschiedene Algorithmen verwendet werden. Die Güte von Fahrgemeinschaften und Einteilungen des Personenstamms kann anhand einer Bewertungsfunktion beurteilt werden. In diese Bewertungsfunktion gehen die o.g. Kriterien Umweg, Arbeitszeiten und Zu- bzw. Abneigungen ein.

Besonderer Wert wird im Projekt auf die Austauschbarkeit der Algorithmen gelegt. Der Prototyp ermöglicht die Untersuchung verschiedener Algorithmen zur Wegsuche und zur Einteilung in Fahrgemeinschaften.

2.2 Projektablauf

2.2.1 Vorgehen

2.2.2 Anforderungsanalyse

Die Anforderungsanalyse dient dazu, das Umfeld der Anwendung zu analysieren, bestehende Anforderungen zu erfassen und noch fehlende zu evaluieren. In der Projektgruppe begann diese Phase damit, daß Friedhelm Buchholz in seiner Rolle als Kunde einen Vortrag über die Anforderungen hielt. In einer zweiten Runde befragte ihn die Projektgruppe dazu. Dann entwickelte sie Szenarien oder Use Cases, aus denen sich dann weitere Anforderungen ergaben. Dabei wurden wichtige Entscheidungen getroffen: Das System wird ein Prototyp werden, der ohne graphische Benutzungsoberfläche auskommt, dafür aber wiederverwendbare Module enthält und um weitere Algorithmen erweitert werden kann. Das Ergebnis dieser Phase ist das Dokument „Anforderungsanalyse“, das sich im Zwischenbericht [4] findet.

2.2.3 Spezifikation

In der Spezifikationsphase wird aus dem Anforderungskatalog das äußere Systemverhalten definiert. Es wird beschrieben, was das System tut, nicht wie es das tut. Für die Spezifikation wurden die Use Cases aus der Anforderungsanalyse verwendet und ausgebaut. Eines der Hauptprobleme war die Datenhaltung und die Konsistenzsicherung zwischen den einzelnen Datenstämmen (Personen, Fahrgemeinschaften, Verkehrsdaten). Das Dokument „Spezifikation“ steht im Kapitel 3 und ist das Ergebnis des gleichnamigen Meilensteins.

2.2.4 Entwurf

Im Entwurf wird entwickelt, wie das System sich intern verhält. Er geht von einem Grobentwurf, in dem die Subsysteme und ihre Schnittstellen identifiziert und definiert werden, in einen Feinentwurf über, bei dem am Ende die genauen Datenstrukturen und Algorithmen beschrieben werden. Das Dokument „Entwurf“ steht im Kapitel 5 und ist das Ergebnis des gleichnamigen Meilensteins.

2.2.5 Zwischenbericht

Der Zwischenbericht erschien etwa nach der Hälfte der Zeit der Projektgruppe. Er enthält die zentralen Dokumente, die bis zu diesem Zeitpunkt erschienen sind. Zusammen mit dem Endbericht stellt er die Studienarbeit der Teilnehmer dar.

2.2.6 Implementierung und Testphase

In der Implementierungsphase werden die im Entwurf entwickelten Schnittstellen, Datenstrukturen und Algorithmen in Programmcode umgesetzt. Allgemein sollte ein Programmierstil verwendet werden, der nicht den Künstler in seiner Einzigartigkeit zeigt, sondern das normgerechte Ingenieurprodukt, dessen Erscheinungsbild jedem kompetenten Leser ein schnelles Verständnis verschafft. Nur solche Programme sind mit akzeptablem Aufwand wartbar.

Für die Implementierung wurden Richtlinien aufgestellt, die folgendes festlegen:

- wie die Bezeichner zu wählen sind
- wie das Layout zu gestalten ist
- welche Sprache verwendet werden soll (Deutsch oder Englisch)
- welche Standardkommentare im Code enthalten sein sollen
- wie die Schnittstellen zwischen den Einheiten zu gestalten sind, z.B. ob globale Variablen zugelassen sind, wie die Parameter zu ordnen sind usw.

Die Testphase dient dazu, die implementierten Komponenten auf semantische Fehler zu überprüfen. Im Zuge der Software-Entwicklung werden Tests in verschiedenen Stadien durchgeführt. Wer eine Komponente fertiggestellt hat, testet diese selbst, ob sie plausible Ergebnisse liefert. Ein Gesamttest kann erst durchgeführt werden, wenn das Programm stabil ist, also weder „abstürzt“ noch offensichtlich Unsinn produziert. Tester und Programmierer verfolgen dieselben Ziele: die Verbesserung des Programms und die Erhöhung des Vertrauens in das Programm [7].

2.2.7 Endbericht und Präsentation

Die Erstellung des Endberichts erfolgte parallel zur Endphase der Implementierung. Teile aus dem Zwischenbericht wurden übernommen und weiterentwickelt. Das Ziel war es, den Endbericht zum Zeitpunkt der Präsentation fertig zu haben. Die Abschlußpräsentation der Projektgruppe erfolgte zu Beginn des Wintersemesteres 97/98.

Kapitel 3

Spezifikation

3.1 Einführung

Dieser Abschnitt enthält die Spezifikation des Softwaresystems Mobidick (Mobil durch intelligentes computerunterstütztes Kombinieren) und wurde nach den IEEE-Richtlinien aus [6] erstellt. Es baut auf der Anforderungsanalyse auf und dient der Beschreibung des äußeren Systemverhaltens. Es ist damit die Grundlage für alle weiteren im Ablauf der Projektgruppe entstehenden Dokumente.

Die Spezifikation gliedert sich wie folgt: Kapitel 3.2 gibt einen allgemeinen Überblick über das System Mobidick. Kapitel 3.3 beschreibt die funktionalen Anforderungen, Kapitel 3.4 die Anforderungen an externe Schnittstellen. In Kapitel 3.5 werden die Leistungsanforderungen beschrieben und in Kapitel 3.6 die zukünftigen Erweiterungen. Im letzten Kapitel (3.7) sind alle Systemmeldungen aufgelistet.

3.2 Allgemeine Beschreibung

3.2.1 Umgebung des Produkts

Das Softwaresystem Mobidick soll unter dem Betriebssystem Solaris 2.5 laufen. Die Ausgabe des Programms erfolgt auf dem Bildschirm oder in eine Datei, die Eingabe über die Tastatur oder eine Maus. Weitere Peripherie wird nicht benötigt.

Zum System gehört ein Tool namens GDF2GRA, um Verkehrsdaten aus dem GDF-Format [2] in das Mobidick-Verkehrsdatenformat zu konvertieren. Das GDF-Format ist in der Dokumentation zu GDF2GRA (siehe Anhang A) beschrieben. Personendaten können über ein spezielles Format aus Dateien eingelesen werden. Dieses Format ist in der Dokumentation zu Mobidick beschrieben.

Eine direkte Schnittstelle zum Drucker ist nicht vorgesehen. Die Programmausgaben auf dem Bilschirm erfolgen rein textuell. Eine Schnittstelle zu einem später zu entwickelnden Fenstersystem ist vorgesehen.

3.2.2 Informelle Beschreibung der Funktionalität

In diesem Abschnitt wird informell beschrieben, welche Funktionen von Mobidick bereitgestellt werden. Eine genaue Beschreibung kann Kapitel 3.3 entnommen werden. Die Interaktion zwischen Benutzer und Programm erfolgt über Menüs.

Zu Programmbeginn erscheint das Hauptmenü mit den Funktionen Dateien, Personen, Vermittlung, Wegsuche, Bewertungsfunktion, Hilfe und Ende. Diese Untermenüs enthalten die folgenden Funktionalitäten:

- Im Untermenü **Dateien** finden sich die notwendigen Funktionen zum Laden und Speichern der Stammdaten.
- Daten einzelner Personen innerhalb der Stammdaten können im Untermenü **Personenverwaltung** eingefügt, verändert oder gelöscht werden. Es besteht die Möglichkeit, zu Testzwecken einen zufälligen Personendatensatz zu generieren.
- Unter **Vermittlung** kann man Einteilungen in Fahrgemeinschaften berechnen. Es stehen verschiedene heuristische und optimale Algorithmen zur Verfügung. Eine Berechnung kann abgebrochen und zu einem späteren Zeitpunkt fortgesetzt werden.
- Die **Wegsuche** bietet die Möglichkeit, kürzeste Wege zwischen zwei Punkten in dem Verkehrsgraphen zu berechnen. Auch hier können verschiedene Algorithmen verwendet und die Rechenzeit gemessen werden.
- Die **Bewertungsfunktion** ist die Grundlage für die Einteilung von Personen in Fahrgemeinschaften. In diesem Menü kann sie verändert, aus einer Datei gelesen oder gespeichert werden.
- Die **Hilfefunktion** zeigt für jedes Menü einen jeweils passenden Hilfetext an.

3.2.3 Charakteristika der Benutzer und Benutzerinnen

In der jetzigen Version kann davon ausgegangen werden, daß die Benutzer und Benutzerinnen von Mobidick über durchschnittliche Erfahrungen im Umgang mit Rechnern verfügen. Für die Erweiterung von Mobidick um Algorithmen werden Erfahrungen mit der Entwicklung in C++ vorausgesetzt.

3.3 Funktionale Anforderungen

Die funktionalen Anforderungen werden in Form von Use Cases oder Szenarien formuliert. Ähnliche oder verwandte Anforderungen werden zu einem Use Case zusammengefaßt. In einem Use Case wird das Zusammenspiel zwischen einem Akteur, in diesem Fall dem Benutzer, und dem System für einen konkreten Anwendungsfall beschrieben.

3.3.1 Start des Fahrgemeinschaftensystems

Das Programm wird von einem Kommandozeileninterpreter aus durch Eintippen des Namens „`mobidick`“ gestartet. Weitere Optionen sind nicht notwendig und werden vom Programm ignoriert. Nach dem Start wird dem Benutzer das Hauptmenü am Bildschirm angezeigt und die in den Voreinstellungen (s. Abschnitt 3.3.11) angegebenen Dateien werden geöffnet. Der Benutzer ist selbst dafür verantwortlich, daß nicht mehrere gleichzeitig gestartete Programme auf die gleiche Personendatei zugreifen.

3.3.2 Menüstruktur

Die Menüstruktur des Systems ist folgendermaßen aufgebaut:

1. Dateien
   (a) Personendateien
      i. Neu
      ii. Laden
      iii. Speichern
      iv. Speichern unter
      v. Schließen
      vi. Importieren
      vii. Zurück
      viii. Hauptmenü
      ix. Hilfe
   (b) Verkehrsgraph laden
   (c) Fahrgemeinschaftseinteilung
      i. Umbenennen
      ii. Duplizieren
      iii. Löschen
      iv. Neu
      v. Zurück
      vi. Hauptmenü
      vii. Hilfe
   (d) Bewertungsfunktionen
      i. Laden
      ii. Speichern
      iii. Speichern unter
      iv. Zurück
      v. Hauptmenü
      vi. Hilfe
   (e) Zurück

   (f) Hauptmenü
   (g) Hilfe
2. Personen
   (a) Neue Person
   (b) Person ändern
   (c) Person löschen
   (d) Personen generieren
   (e) Personen anzeigen
   (f) Zurück
   (g) Hauptmenü
   (h) Hilfe
3. Fahrgemeinschaften
   (a) Neuer Teilnehmer
      i. Manuell eintragen
      ii. Suchsystem
      iii. Zurück
      iv. Hauptmenü
      v. Hilfe
   (b) Teilnehmer fest eintragen
   (c) Teilnehmer löschen
   (d) Fahrgemeinschaft eingeben
   (e) Fahrgemeinschaft ändern
   (f) Fahrgemeinschaft auflösen
   (g) Fahrgemeinschaft bewerten
   (h) Fahrgemeinschaft anzeigen
   (i) Fahrgemeinschaft markieren/unmarkieren
   (j) Zurück
   (k) Hauptmenü
   (l) Hilfe
4. Vermittlung
   (a) Einteilung auswählen
   (b) Systemmeldungen (ein/aus)
   (c) Einteilung berechnen
   (d) Fortfahren mit letzter Berechnung
   (e) Laufzeitmessung (ein/aus)
   (f) Einteilung bewerten
   (g) Einteilung anzeigen

(h) Einteilung auflösen
(i) Zurück
(j) Hauptmenü
(k) Hilfe

5. Bewertungsfunktionen
   (a) Neu
   (b) Ändern
   (c) Auswählen
   (d) Anzeigen
   (e) Zurück
   (f) Hauptmenü
   (g) Hilfe
6. Wegsuche
   (a) Einzelwegsuche
   (b) Wegsuche n-mal
   (c) Auswahl des Algorithmus
   (d) Laufzeit
   (e) Systemmeldungen (ein/aus)
   (f) Zurück
   (g) Hauptmenü
   (h) Hilfe
7. Voreinstellungen
8. Hilfe
9. Ende

3.3.3 Datenmodell

Mit Mobidick können sogenannte Personendateien verwaltet werden. Diese bestehen aus einer Menge P von Personendaten, k Einteilungen M_i ($1 \leq i \leq k$) und den Einteilungen zugeordneten Bewertungsfunktionen f_i (siehe Abbildung 3.1). Die Personendaten einer Person bestehen aus einer eindeutigen Identifikationsnummer, Angaben zur Arbeitszeit und den Eigenschaften der Person. Eine Einteilung besteht aus einer Menge von Fahrgemeinschaften und einer Bewertungsfunktion, mit der diese eingeteilt wurden. Fahrgemeinschaften sind Teilmengen der Personenmenge und alle Fahrgemeinschaften einer Einteilung sind paarweise disjunkt. Die Vereinigung aller Fahrgemeinschaften einer Einteilung ergibt eine Teilmenge der Personenmenge.

Es kann immer nur eine Personendatei geöffnet sein, d.h. vor dem Öffnen einer anderen Personendatei muß die bereits geöffnete geschlossen werden. Alle zu

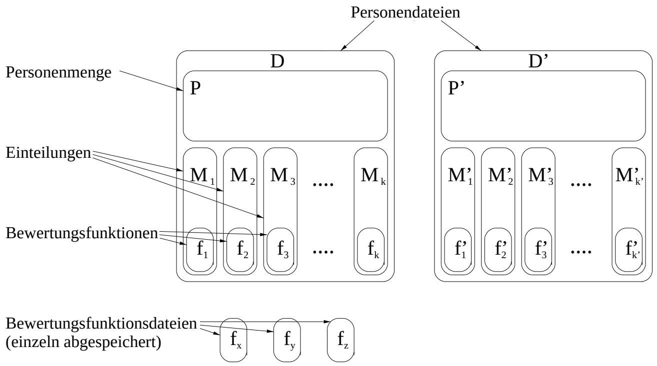

Abb. 3.1: Datenmodell

einer Personendatei gehörenden Daten werden gemeinsam in einer Datei abgespeichert. Um dem Benutzer die Möglichkeit zu geben, eine Bewertungsfunktion aus einer Personendatei D in einer anderen Personendatei D' anzuwenden, kann er Bewertungsfunktionen einzeln in Dateien abspeichern. Solche unabhängig gespeicherten Bewertungsfunktionen können dann zusätzlich zu den in einer Personendatei vorhandenen geöffnet werden (siehe Abbildung 3.1).

Eine in der Personendatei enthaltene und vom Benutzer bestimmte Einteilung wird als aktuelle Einteilung bezeichnet. Genauso ist eine Bewertungsfunktion die aktuelle, es kann hier auch eine zusätzlich geladene sein.

3.3.4 Beenden des Fahrgemeinschaftensystems

Zum Beenden des Programms muß der Befehl `Ende` aus dem Hauptmenü ausgewählt werden. Ist noch eine Personendatei geöffnet, dann wird der Benutzer gefragt „`Personendatei <Name> vor dem Beenden speichern? [J/n]`“. Für die Behandlung noch geöffneter Bewertungsfunktionen siehe Use Case 3.3.5.5.

3.3.5 Dateien

3.3.5.1 Neue Personendatei anlegen

Durch Auswahl des Menüpunkt `Datei-Personendateien-Neu` wird eine leere Personendatei angelegt. Nach Auswahl des Menüpunkts wird dem Benutzer eine Liste mit den im aktuellen Verzeichnis enthaltenen Personendateien angezeigt. Dann wird er aufgefordert den neuen Dateinamen anzugeben durch die Meldung „`Geben Sie einen Namen fuer die zu speichernde Datei (ohne`

`Endung) a fuer akzeptieren oder v fuer Verzeichnis wechseln ein`“. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Meldung „`Name existiert schon, trotzdem abspeichern und vorhandene Datei ueberschreiben? [j/N]`“ angezeigt. Bei Eingabe von „j“ wird die unter diesem Namen existierende Datei überschrieben, bei „n“ wird die Eingabeaufforderung wiederholt.

An den Dateinamen wird vom System die Erweiterung `.per` angehängt, mit der die Datei als Personendatei gekennzeichnet wird.

Das System kann nur eine Personendatei im Hauptspeicher bereithalten. Wenn während der Auswahl von `Neu` schon eine Datei geöffnet ist, wird der Benutzer gefragt „`Personendatei <Name> vor dem Erstellen der neuen Datei speichern? [J/n]`“. Für die Behandlung noch geöffneter Bewertungsfunktionen siehe Use Case 3.3.5.5.

3.3.5.2 Personendatei laden

Nach Auswahl des Menüpunkts `Datei-Personendateien-Laden` wird eine Liste mit den im aktuellen Verzeichnis enthaltenen Personendateien angezeigt. Aus dieser Liste kann die gewünschte Datei ausgewählt werden.

Wenn während der Auswahl von `Laden` schon eine Datei geöffnet ist, wird der Benutzer gefragt „`Personendatei <Name> vor dem Laden der anderen Datei speichern? [J/n]`“. Für die Behandlung noch geöffneter Bewertungsfunktionen siehe Use Case 3.3.5.5.

Enthält die zu ladende Personendatei eine oder mehrere Einteilungen, dann werden nach dem Ladevorgang die erste Einteilung und deren Bewertungsfunktion als aktuell eingestellt.

3.3.5.3 Personendatei speichern

Durch Auswahl des Menüpunkts `Datei-Personendateien-Speichern` wird die aktuelle Personendatendatei unter dem ihr zugewiesenen Dateinamen abgespeichert.

3.3.5.4 Personendatei unter anderem Namen speichern

Nach Auswahl des Menüpunkts `Datei-Personendateien-Speichern unter` wird dem Benutzer eine Liste mit den im aktuellen Verzeichnis enthaltenen Personendateien angezeigt. Dann wird er aufgefordert den neuen Dateinamen anzugeben. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Meldung „`Name existiert schon, trotzdem abspeichern und vorhandene Datei ueberschreiben? [j/N]`“ angezeigt. Bei Eingabe von „j“ wird die unter diesem Namen existierende Datei überschrieben, bei „n“ wird die Eingabeaufforderung wiederholt.

3.3.5.5 Personendatei schließen

Durch Auswahl des Menüpunkts `Datei-Personendateien-Schließen` wird die aktuelle Personendatei geschlossen. Dazu wird der Benutzer gefragt „`Datei vor dem Schließen speichern? [J/n]`“. Wenn er sie unter einem anderen Namen abspeichern will, hat er die Möglichkeit, dies durch Auswahl des Use Case 3.3.5.4 zu tun. Bei Abbruch des Vorgangs bleibt die aktuelle Datei unverändert im Speicher.

Sind noch Bewertungsfunktionen geöffnet, die seit der letzten Änderung nicht gespeichert wurden, so wird der Benutzer zu jeder Bewertungsfunktion gefragt „`Bewertungsfunktion <Name> vor dem Schließen speichern? [J/n]`“. Antwortet der Benutzer mit „j“, so wird die Bewertungsfunktion gespeichert, sonst gehen ihre Werte verloren.

3.3.5.6 Personendatei importieren

Durch das Importieren wird die aktuelle Personendatei um den Inhalt der auf dem Datenträger abgespeicherten Datei erweitert. Sei D die aktuelle Personendatei mit der Personenmenge P und den Fahrgemeinschaftseinteilungen $M_1, \ldots, M_k$ mit den zugehörigen Bewertungsfunktionen $f_1, \ldots, f_k$. D' sei die zu importierende Personendatei mit der Personenmenge P' und D'' die erweiterte Personendatei mit der Personenmenge P'; dann hat die Importierung folgende Auswirkungen:

- Die Personenmenge P'' ist die disjunkte Vereinigung der Personenmengen P und P', wobei die Personen-IDs in P' automatisch angepasst werden, damit keine ID doppelt vorkommt. Personen, die in P und in P' enthalten sind, werden in P'' nur einmal aufgeführt. Personen können durch den Namen und das Geburtsdatum eindeutig identifiziert werden.
- Die Menge der Fahrgemeinschaftseinteilungen in D'' enthält alle Einteilungen aus D und D'. Dabei werden die Personen-IDs der Einteilungen aus D' wie oben automatisch angepasst. Alle Personen aus P (P') sind in den Einteilungen aus D' (D) noch nicht vermittelt. Das bedeutet, daß alle Einteilungen aus D'' nur einen Teil der Personenmenge enthalten können. Will der Benutzer eine Fahrgemeinschaftseinteilung, die alle Personen berücksichtigt, so muß er entweder eine neue Einteilung berechnen oder eine der bestehenden Einteilungen erweitern (siehe 3.3.8.3).

Nach Auswahl des Menüpunkts `Datei-Personendateien-Importieren` wird der Benutzer durch die Systemmeldung „`Name:`“ aufgefordert, den Namen der zu importierenden Datei anzugeben.

Es können nur Dateien, die dem Personendateiformat entsprechen, importiert werden. Dieses Format wird im Enfwurf festgelegt.

3.3.5.7 Verkehrsdaten laden

Nach Auswahl des Menüpunkts `Datei-Verkehrsdaten Laden` wird eine Liste mit den im aktuellen Verzeichnis enthaltenen Verkehrsgraphen angezeigt. Aus dieser Liste kann die gewünschte Datei ausgewählt werden. Das System kann nur einen Verkehrsgraphen im Hauptspeicher bereithalten. Falls zum Zeitpunkt des Lade-Befehls schon ein Graph geöffnet ist, so wird dieser vor Ausführung des Lade-Befehls geschlossen. Es können nur Verkehrsgraphen geladen werden, die dem vom System unterstützten Graphenformat entsprechen.

3.3.5.8 Einteilung umbenennen

Nach Auswahl des Menüpunkts `Datei-Fahrgemeinschaftseinteilung-Umbenennen` wird dem Benutzer eine Liste aller in der Personendatei enthaltenen Einteilungen angezeigt. Dann wird er aufgefordert, den neuen Einteilungsnamen anzugeben. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Meldung „`Name existiert schon, trotzdem abspeichern und vorhandene Datei überschreiben? [j/N]`“ angezeigt. Bei Eingabe von „j“ wird die unter diesem Namen existierende Datei überschrieben, bei „n“ wird die Eingabeaufforderung wiederholt.

3.3.5.9 Einteilung duplizieren

Durch Auswahl des Menüpunkts `Datei-Fahrgemeinschaftseinteilung-Duplizieren` kann eine bestehende Einteilung dupliziert werden. Nach Auswahl des Menüpunkts wird dem Benutzer eine Liste aller in der Personendatei enthaltenen Einteilungen angezeigt und er wird aufgefordert, den Einteilungsnamen des Duplikats anzugeben. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Meldung „`Name existiert schon, trotzdem abspeichern und vorhandene Datei überschreiben? [j/N]`“ angezeigt. Bei Eingabe von „j“ wird die unter diesem Namen existierende Datei überschrieben, bei „n“ wird die Eingabeaufforderung wiederholt. Beim Duplizieren einer Einteilung werden alle Fahrgemeinschaften und die Bewertungsfunktion kopiert.

3.3.5.10 Einteilung löschen

Nach Auswahl des Menüpunkts `Datei-Fahrgemeinschaftseinteilung-Löschen` wird dem Benutzer eine Liste aller in der Personendatei enthaltenen Einteilungen angezeigt. Daraus kann er die zu löschende Einteilung auswählen. Nach der Auswahl wird er gefragt „`Einteilung wirklich löschen? [j/N]`“. Antwortet er mit „j“, so wird die ausgewählte Einteilung aus der Personendatei gelöscht.

3.3.5.11 Bewertungsfunktion laden

Mit diesem Menüpunkt kann eine Bewertungsfunktion, die zuvor in einer Datei abgespeichert wurde, in die Liste der Bewertungsfunktionen aufgenommen werden. Der Benutzer erhält so die Möglichkeit, Bewertungsfunktionen der Personendatei X abzuspeichern und dann auf eine Fahrgemeinschaftseinteilung der Personendatei Y anzuwenden.

Nach Auswahl des Menüpunkts `Datei-Bewertungsfunktionen-Laden` wird eine Liste mit den im aktuellen Verzeichnis enthaltenen Bewertungsfunktionsdateien angezeigt. Aus dieser Liste kann die gewünschte Datei ausgewählt werden.

3.3.5.12 Bewertungsfunktion speichern

Durch Auswahl des Menüpunkts `Datei-Bewertungsfunktionen-Speichern` wird die aktuelle Bewertungsfunktion unter dem ihr zugewiesenen Dateinamen abgespeichert.

3.3.5.13 Bewertungsfunktion unter anderem Namen speichern

Nach Auswahl des Menüpunkts `Datei-Bewertungsfunktionen-Speichern unter` wird dem Benutzer eine Liste mit den im aktuellen Verzeichnis enthaltenen Bewertungsfunktionsdateien angezeigt. Dann wird er aufgefordert, den neuen Dateinamen anzugeben. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Meldung „`Name existiert schon, trotzdem abspeichern und vorhandene Datei ueberschreiben? [j/N]`“ angezeigt. Bei Eingabe von „j“ wird die unter diesem Namen existierende Datei überschrieben, bei „n“ wird die Eingabeaufforderung wiederholt.

3.3.6 Personen

3.3.6.1 Neue Person eintragen

Durch Aufruf der Menüpunkte `Personen-Neue Person` kann der Benutzer eine neue Person anlegen. Das System fragt nacheinander die verschiedenen Daten ab. Dabei bedeutet ein Stern hinter der Frage, daß eine Wertangabe unbedingt notwendig ist. Wird bei solchen Feldern kein Wert angegeben, erfolgt nach Eingabe aller Werte die Meldung „`Fehler : Personendaten nicht vollstaendig!!`“ und mit der Meldung „`Fehlende Daten ergaenzen? [J/n]`“ gefragt, ob der Benutzer die fehlenden Daten ergänzen will. Antwortet er hier mit „j“, so werden die Daten, zu denen er zuvor nichts eingegeben hatte, erneut abgefragt.

Sind die Daten vollständig, so fragt das System `Soll die Person in den Datenbestand uebernommen werden? [J/n]`. Lautet die Antwort „j“, so wird sie übernommen, ansonsten nicht. In jedem Fall befindet man sich nun wieder im Menü `Personen`.

Folgende Daten werden abgefragt:

1. `Name *`
2. `Vorname *`
3. `Geschlecht (m/w) *`
4. `Geburtsdatum (TT.MM.JJ) *`

5. `Strasse *`
6. `Hausnummer *`
7. `PLZ *`
8. `Wohnort *`
9. `Telefon (vorwahl/nummer)`
10. `email`
11. `Raucher (j/n)`
12. `Fahrer (j/n) *`
13. `Komfortklasse des Autos (klein/mittel/gehoben) *`
14. `Baujahr des Autos`
15. `Anzahl Plaetze im Auto *`
16. `Musikgeschmack`
17. `Startort: Strasse (ohne HausNr)`
18. `Startort: Hausnummer`
19. `Startort: PLZ`
20. `Startort: Stadt`
21. `Zielort: Strasse (ohne HausNr)`
22. `Zielort: Hausnummer`
23. `Zielort: PLZ`
24. `Zielort: Stadt`
25. `Startkoordinaten (bitte anklicken)`
26. `Zielkoordinaten (bitte anklicken)`
27. `Ankunftszeit (HH:MM[:SS]-HH:MM[:SS]) *`
28. `Rueckkehrzeit (HH:MM[:SS]-HH:MM[:SS]) *`
29. `Arbeitszeit (HH:MM[:SS]) *`

Bei der Eingabe werden die Felder nacheinander abgefragt. Die Felder 13, 14 und 15 werden nicht gefragt, sofern bei 12. `Fahrer?` „n“ eingegeben wurde. In diesem Fall sind die Angaben zum eigenen Auto unnötig.

Bei 13. `Komfortklasse` wird zwischen Kleinwagen `klein`, Mittelklasse `mittel` und gehobener Klasse `gehoben` unterschieden.

Bei 15. `Anzahl Plaetze` erhält man durch Drücken der Return-Taste den Defaultwert vier.

Bei 16. `Musikgeschmack` können mehrere Musikrichtungen ausgewählt werden (`Klassik, Pop, Rock, Schlager`), außerdem gibt es noch die Alternativen `Ruhe` und `Egal`.

Bei 25. `Startkoordinaten` und 26. `Zielkoordinaten` wird zunächst versucht, die Zahl aus den Daten zum Start- und Zielort zu bestimmen. Ist dies möglich, so wird sie präsentiert und der Benutzer kann sie übernehmen. Ist es nicht möglich oder will der Benutzer sie ändern, so klickt er im Programm `graphdraw` die entsprechende Kante an. Diese wird auf dem Bildschirm ausgegeben und die nächste Frage gestellt.

27. `Ankunftszeit` und 28. `Rückfahrtzeit` werden als Intervall eingegeben, dessen Länge 0 aber nicht negativ sein darf. Macht der Benutzer eine falsche Eingabe, erscheint die Meldung „`Fehler : Fehler in der Eingabe. Bitte wiederholen.`“ und die Frage wird erneut gestellt. Gibt der Benutzer „q“ ein, so erscheint die Meldung „`Keine Angabe.`“ und das Datum bleibt leer. Gleiches gilt für die Arbeitsdauer.

Folgende Daten werden zu den Wünschen in Bezug auf die Mitfahrer eingegeben. Jeder Wunsch trägt einen Gewichtsfaktor von 0 bis 10. 0 steht für völlig unwichtig, 10 für sehr wichtig. Durch Belegung mit dem Gewicht 0 werden sämtliche Wünsche abgeschaltet. Alle Gewichte haben den Defaultwert 0, zu Wünschen mit dem Gewicht 0 muß nichts eingegeben werden.

1. `abgelehnte Musikrichtungen:`
2. `Gewicht:`
3. `Geschlecht:`
4. `Gewicht:`
5. `Raucher:`
6. `Gewicht:`
7. `gewünschte Komfortklasse:`
8. `Gewicht:`
9. `persönliche Abneigung: Person hinzufügen`
10. `persönliche Abneigung: Person löschen`
11. `persönliche Zuneigung: Person hinzufügen`
12. `persönliche Zuneigung: Person löschen`

Bei 1. `abgelehnte Musikrichtungen` können eine oder mehrere Musikrichtungen aus der Menge `Klassik, Pop, Rock, Schlager` angegeben werden. Bei 3. `Geschlecht` kann man angeben, ob man nur mit Männern oder nur mit Frauen fahren will. Bei 5. `Raucher` wird festgelegt, ob man nur mit Rauchern oder nur mit Nichtrauchern fahren will. Bei 7. `gewünschte Komfortklasse` sind die drei oben erwähnten Komfortklassen als Eingabe möglich, die angegebene Komfortklasse ist als Mindestanforderung zu verstehen.

Durch Aufruf von 9. `persönliche Abneigung: Person hinzufügen` wird direkt zur Filterfunktion von Use Case 3.3.6.5 übergegangen. Dort wird über den Index der ausgegebenen Personenliste eine Person ausgewählt, die dann in die Liste der abgelehnten Personen eingefügt wird. Danach befindet man sich wieder in der ursprünglichen Bildschirmmaske, in der diese Liste auch angezeigt wird (Personen-IDs). Bei dem Versuch, eine Person wiederholt einzufügen, erscheint die Fehlermeldung „`Fehler 6: Person bereits vorhanden`". Der Menüpunkt 11. `persönliche Zuneigung: Person hinzufügen` verhält sich analog.

Bei Aufruf von 10. `persönliche Abneigung: Person löschen` kann eine Personen-ID aus der angezeigten Liste eingegeben werden, die betreffende Person wird dann aus der Liste entfernt. Ist die ID nicht in der Liste vorhanden, so erscheint die Fehlermeldung „`Fehler 7: Person nicht in Liste vorhanden`". Der Menüpunkt 12. `persönliche Zuneigung: Person löschen` verhält sich analog.

3.3.6.2 Person ändern

Durch Aufruf der Menüpunkte `Personen-Person ändern` wird direkt zur Filterfunktion aus Use Case 3.3.6.5 übergegangen. Dort kann der Benutzer eine Person über die Personen-ID, Name und Vorname oder ein anderes Kriterium suchen lassen. Aus den gefundenen Personen wählt er über den Index eine aus. Danach erscheinen dieselben Bildschirmmasken wie in Use Case 3.3.6.1 und die Änderungen können vorgenommen werden. Die Änderungen werden erst wirksam, wenn der Menüpunkt `Person in Datenbestand übernehmen` ausgewählt wird. Falls sich Startort, Zielort, Zeiten oder das Feld `Fahrer?` geändert haben, erscheint die Meldung „`Änderungen für Fahrgemeinschaftsbildung relevant`" und anschließend die Frage „`Änderungen vornehmen und Person aus den betroffenen Fahrgemeinschaften löschen? [J/n]`". Falls durch die Änderung die Auflösung einer Fahrgemeinschaft notwendig wird (Person kann nicht mehr fahren), erscheint die Frage „`Änderungen vornehmen und Fahrgemeinschaft auflösen? [J/n]`". Nach Eingabe von „j" wird eine Liste der Personen ausgegeben, die von dieser Auflösung betroffen sind.

3.3.6.3 Person löschen

Durch Aufruf der Menüpunkte `Personen-Person löschen` wird direkt zur Filterfunktion aus Use Case 3.3.6.5 übergegangen. Die Auswahl einer Person geschieht wie in Use Case 3.3.6.2. Falls die Person Fahrer einer Fahrgemeinschaft war, erscheint die Meldung „`Person ist Fahrer, löschen führt zur Auflösung einer Fahrgemeinschaft`". Danach erscheint die Frage „`Person wirklich löschen? [J/n]`". Bei Eingabe von „j" wird die Person aus dem Datenbestand gelöscht, eine Liste der von der Auflösung betroffenen Personen angezeigt und zum Menü `Personenverwaltung` zurückgekehrt, bei „n" befindet man sich sofort wieder im Menü `Personenverwaltung`.

3.3.6.4 Personen generieren

Durch Aufruf der Menüpunkte `Personen-Personen generieren` kann man für Testzwecke eine Personenmenge zufällig erzeugen.

Dazu werden folgende Parameter abgefragt:

1. `Personenzahl *`
2. `Anteil Fahrer (in Prozent) *`
3. `Startort gleichverteilt? (j/n)`
4. `Startort (fest):`
5. `Zielort gleichverteilt? (j/n)`
6. `Zielort (fest):`
7. `Ankunftintervall (HH:MM[:SS]-HH:MM[:SS]) *`
8. `Ankunftszeit (Intervallaengen) *`
9. `Rueckfahrtintervall (HH:MM[:SS]-HH:MM[:SS]) *`
10. `Rückfahrtzeit (Intervallaengen) *`

Bei den Start- und Zielorten kann zwischen einer Gleichverteilung auf dem ganzen Verkehrsgraphen oder einer festen Start- und Zielortadresse gewählt werden.

Die Ankunftszeiten unterliegen einer Gleichverteilung auf dem durch `von` und `bis` gegebenen Intervall. Bei falscher Eingabe erscheint die Fehlermeldung „`Fehler Falsche Eingabe!: .`" In diesem Intervall liegen die einzelnen Ankunftsintervalle der Personen, deren Länge wird in 8. `Ankunftszeit: (Intervallängen)` angegeben.

Für die Rückfahrtzeiten gilt das entsprechende.

Nach Eingabe der Werte wird der Benutzer gefragt „`Personenmenge so generieren? [J/n]`". Beantwortet er die Frage mit „n", so befindet er sich wieder im Personenmenü. Ansonsten wird für jede Person ein Start-, ein Zielort, ein Ankunftsintervall und ein Rückfahrtsintervall generiert, wobei die Intervallängen für alle Personen gleich sind. Alle hier nicht aufgeführten Personeneigenschaften sind nicht vom Benutzer beeinflußbar und werden automatisch generiert. Nach der Berechnung erscheint die Frage „`Personenmenge übernehmen? [J/n]`". Bei Eingabe von „j" wird die generierte Personenmenge als aktuelle Personenmenge übernommen, bei „n" befindet man sich wieder in obigem Menü und kann die Parameter verändern.

3.3.6.5 Personen anzeigen

Durch Aufruf der Menüpunkte `Personen-Personen anzeigen` ist die Personensuche über eine Filterfunktion möglich. Dabei können folgende Suchkriterien angegeben werden:

1. `Name:`
2. `Vorname:`
3. `Personen-ID:`
4. `Startort: (Straßenname)`
5. `Startort: (Radius)`
6. `Zielort: (Straßenname)`
7. `Zielort: (Radius)`
8. `Ankunftszeit: (von)`
9. `Ankunftszeit: (bis)`
10. `Rückfahrtzeit: (von)`
11. `Rückfahrtzeit: (bis)`
12. `Status: (fest/reserviert)`
13. `FGM-ID:`
14. `Fahrer?:`
15. `Zurück`
16. `Hauptmenü`
17. `Hilfe`

Beim Startort kann ein Straßenname und ein Radius angegeben werden, in diesem Bereich soll dann der Startort der Person liegen. Analog beim Zielort. Für Ankunfts- und Rückfahrtszeit können Intervalle angegeben werden, bei falscher Eingabe erscheint die Fehlermeldung „`Fehler 5: Obere Grenze kleiner untere Grenze.`“ In diesem Intervall müssen die tatsächliche Ankunfts- und Rückfahrtszeit der Fahrgemeinschaft liegen, nicht das von der Person angegebene Wunschintervall.

Beim Status wird zwischen folgenden Personengruppen unterschieden: Eine Person ist *vermittelbar*, falls sie bisher in keine Fahrgemeinschaft eingetragen wurde. Sie hat den Status *als reserviert eingetragen*, falls bereits ein Platz in einer Fahrgemeinschaft für sie reserviert wurde. Nimmt sie diesen Platz an, geht der Status über in *fest eingetragen*. Reichen die zu einer Person eingegebenen Daten für eine Vermittlung noch nicht aus, hat sie den Status *Daten unvollständig* (siehe Use Case 3.3.6.1). Das Feld `Fahrer?` entspricht dem gleichnamigen Feld in Use Case 3.3.6.1.

Bei den Suchkriterien können auch einzelne Eingabefelder freigelassen werden. Das System sucht dann nach allen Personen, die alle (UND-Verknüpfung) Kriterien erfüllen und gibt sie als Liste mit Index auf dem Bildschirm aus. Es werden tabellarisch angezeigt:

1. `Personen-ID:`
2. `Name:`
3. `Vorname:`
4. `Startort:`
5. `Zielort:`
6. `Ankunftszeitpunkt:`
7. `Rückfahrtszeitpunkt:`
8. `Status: fest/reserviert`
9. `Fahrgemeinschafts-ID:`

Danach erscheint die Frage „`Tabelle in PostScript-Datei ausgeben? [j/N]`“. Durch Anwahl über die Indexnummer einer Person kann der Benutzer zwischen folgenden weitergehenden Informationen zu einer Person wählen:

1. `weitere Personeneigenschaften` (s. Use Case 3.3.6.1)
2. `Wünsche der Person` (s. Use Case 3.3.6.1)
3. `als Fahrer eingeteilt?`
4. `Datum der Eintragung der Person in das System`

Wird die Filterfunktion zur Auswahl einer bestimmten Person benutzt (z.B. in Use Case 3.3.6.2), führt die Angabe einer Indexnummer nicht zur Anzeige weitergehender Informationen. Der Vorgang ist dann mit der Auswahl abgeschlossen.

3.3.6.6 Personeneigenschaften erweitern

Durch Aufruf der Menüpunkte `Personen-Personeneigenschaften erweitern` ist es möglich, eine weitere Personeneigenschaft hinzuzufügen. Durch Hinzunahme einer weiteren Personeneigenschaft tritt diese auch bei den Wünschen bezüglich der Mitfahrer auf. Für eine bereits vorhandene Personenmenge wird der Wert der neuen Eigenschaft zunächst offengelassen, dies entspricht dem Wert `keine Angabe`.

Es erscheint folgendes Menü:

1. `Name der neuen Eigenschaft:`
2. `Wertebereich:`
3. `eindeutiger Wert`
4. `Mehrfachauswahl`

5. `Eigenschaft hinzufügen`
6. `Zurück`
7. `Hauptmenü`
8. `Hilfe`

Bei 1. kann der Name der neuen Eigenschaft eingegeben werden. Dieser wird daraufhin überprüft, ob er nicht bereits schon für eine andere Eigenschaft verwendet wurde. Bei 2. können die einzelnen Werte in Form von Strings, getrennt durch Kommata eingegeben werden. Bei Mehrfacheingabe eines Wertes muß die Eingabe wiederholt werden. In 3. und 4. wird festgelegt, ob einer Person ein eindeutiger oder mehrere Werte aus dem Wertebereich zugeordnet werden. Diese Einstellung gilt dann auch für die Wünsche bezüglich der Mitfahrer. Durch Auswahl von 5. wird die neue Personeneigenschaft hinzugefügt.

3.3.7 Fahrgemeinschaften

3.3.7.1 Fahrgemeinschaften-Filter

Das Anzeigen von Fahrgemeinschaften kann durch Filter eingeschränkt werden. Dabei werden nach Index die Filterkriterien aufgeführt:

1. Anzahl der freien Plätze (min.)
2. Anzahl der Teilnehmer (max.)
3. Fahrgemeinschaften-ID
4. Komfortklasse des Autos (min.)
5. Startort (Punkt und Radius in km)
6. Zielort (Punkt und Radius in km)
7. Startzeit (Intervall)
8. Zielzeit (Intervall)
9. Markierung
10. Akzeptieren und Weiter

Die Änderungen nimmt man durch Anwählen des Index und Eingeben des neuen Wertes vor, wobei gewisse Werte ignoriert werden. Dies sind negative Werte, nicht existierende Komfortklassen, nicht existierende Punkte und Zeitintervalle. Zusätzlich gibt es den Punkt `Akzeptieren und Weiter`, der die Eingabe der Kriterien beendet.

3.3.7.2 Teilnehmer anzeigen

Die Teilnehmer einer bereits gewählten Fahrgemeinschaft werden gekürzt dargestellt. Dabei werden sie beginnend mit eins aufsteigend numeriert und die Attribute Vorname, Nachname, kann und will fahren, die Markierung (*vermittelbar*, *reserviert eingetragen*, *fest eingetragen*), sowie die ID der Person werden angezeigt.

3.3.7.3 Neuer Teilnehmer

Um einen neuen Teilnehmer in eine Fahrgemeinschaft der aktuellen Fahrgemeinschaftseinteilung aufzunehmen, müssen vorher bereits die Daten des Teilnehmers wie in Use Case 3.3.6.1 eingegeben worden sein. Man kann anschließend den neuen Teilnehmer manuell oder per Suchsystem eintragen lassen. Dazu wählt man den Menüeintrag `Fahrgemeinschaften-Neuer Teilnehmer-Manuell eintragen` oder den Menüeintrag `Fahrgemeinschaften-Neuer Teilnehmer-Suchsystem`.

manuell eintragen

Man möchte eine Person in eine bestehende Fahrgemeinschaft per Hand aufnehmen. Die betreffende Person wird wie in Use Case 3.3.6.5 beschrieben selektiert. Um nun die Fahrgemeinschaft zu finden, werden Fahrgemeinschaften angezeigt (siehe Use Case 3.3.7.10). Nun wählt man über den Index eine der Fahrgemeinschaften aus und beantwortet die Frage des Systems „`Person in Fahrgemeinschaft aufnehmen? [J/n]`“ mit „j“. Auf die neue Fahrgemeinschaft wird dann automatisch die Bewertungsfunktion angewendet und das Ergebnis präsentiert.

Nun beantwortet der Benutzer noch die Frage „`Fahrgemeinschaft übernehmen? [J/n]`“ mit „j“ und die neu entstandene Fahrgemeinschaft wird vom System übernommen. Der neue Teilnehmer wird als *reserviert eingetragen* markiert. Beantwortet man eine der beiden Fragen mit „n“, so wird wieder die Liste der Fahrgemeinschaften angezeigt.

Suchsystem

Man möchte eine Person in eine bestehende Fahrgemeinschaft eintragen und dabei die Hilfe des Systems in Anspruch nehmen. Dazu wird einem nach der Auswahl der Person wie in Use Case 3.3.6.5 beschrieben eine Liste von Fahrgemeinschaften angezeigt, die nach der aktuellen Bewertungsfunktion gut zu der Person passen würden (siehe Use Case 3.3.7.10). Nun wählt man über den Index eine der Fahrgemeinschaften aus und beantwortet die Frage des Systems „`Person in Fahrgemeinschaft aufnehmen? [J/n]`“ mit „j“. Auf die neue Fahrgemeinschaft wird dann automatisch die Bewertungsfunktion angewendet und das Ergebnis präsentiert.

Nun beantwortet der Benutzer noch die Frage „`Fahrgemeinschaft übernehmen? [J/n]`“ mit „j“ und die neu entstandene Fahrgemeinschaft wird vom System übernommen. Der neue Teilnehmer wird als reserviert markiert. Beantwortet man eine der beiden Fragen mit „n“, so wird wieder die Liste der Fahrgemeinschaften angezeigt.

3.3.7.4 Teilnehmer fest eintragen

Ein bereits in eine Fahrgemeinschaft eingetragener Teilnehmer, der noch *reserviert eingetragen* ist, wird nun *fest eingetragen*. Man wählt den Menüpunkt `Fahrgemeinschaften-Teilnehmer fest eintragen`. Es wird eine Liste von Fahrgemeinschaften angezeigt (siehe Use Case 3.3.7.10) die reserviert eingetragene Personen enthalten. Daraus wählt man dann die betreffende Fahrgemeinschaft über den Index aus.

Die Teilnehmer der Fahrgemeinschaft werden nach Use Case 3.3.7.2 angezeigt und man wählt die fest einzutragende Person über ihren Index an. Falls die Person bis jetzt *reserviert eingetragen* war, wird gefragt „`Teilnehmer fest eintragen? [J/n]`“. Sonst wird die Wahl übergangen und die Meldung „`Teilnehmer schon fest eingetragen!`“ ausgegeben. Wird die Frage mit „j“ beantwortet, wird der neue Teilnehmer als *fest eingetragen* markiert. Beantwortet man die Frage mit „n“, werden wieder die Teilnehmer der Fahrgemeinschaften angezeigt.

3.3.7.5 Teilnehmer löschen

Eine Person, die in eine Fahrgemeinschaft eingetragen ist, soll aus ihr gelöscht werden. Bei dieser Person wird dann nur die Markierung *fest eingetragen* oder *reserviert eingetragen* in *vermittelbar* abgeändert und sie wird aus der Fahrgemeinschaft ausgetragen. Dazu wählt man `Fahrgemeinschaften-Teilnehmer löschen`. Es wird eine Liste von Fahrgemeinschaften angezeigt (siehe Use Case 3.3.7.10). Daraus wählt man dann die betreffende Fahrgemeinschaft über den Index aus.

Die Teilnehmer der Fahrgemeinschaft werden nach Use Case 3.3.7.2 angezeigt und man wählt die zu löschende Person über ihren Index an. Wird die Frage des Systems „`Person aus Fahrgemeinschaft löschen? [J/n]`“ mit „j“ beantwortet, wird die Person aus dieser Fahrgemeinschaft gelöscht.

Ist die zu löschende Person der Fahrer, so erscheint die Systemmeldung „`Vorsicht. Durch Loeschen des Fahrers wird die Fahrgemeinschaft aufgelöst.`“. Daraufhin wird gefragt „`Fahrer löschen? [j/N]`“. Wird diese Frage mit „j“ beantwortet, wird die Fahrgemeinschaft aufgelöst (siehe auch Use Case 3.3.7.8) und eine Liste mit den betroffenen Personen wird angezeigt. Ansonsten befindet man sich wieder bei der Liste der Personen.

3.3.7.6 Fahrgemeinschaft eingeben

Fahrgemeinschaften können auch manuell zusammengestellt werden. Die Personen, die hier eingetragen werden sollen, müssen bereits im System erfaßt sein. Der Punkt `Fahrgemeinschaften-Fahrgemeinschaft eingeben` wird gewählt.

Nun wird zuerst der Fahrer nach Use Case 3.3.6.5 selektiert, wobei nur potentielle Fahrer angezeigt werden, die in keiner Fahrgemeinschaft eingetragen sind. Die Person wird über den Index angewählt und die Frage „`Person als Fahrer übernehmen? [J/n]`“ gestellt. Wird die Frage mit „j“ beantwortet, fährt man fort, bei „n“ zeigt man wieder die Liste der potentiellen Fahrer.

Solange freie Plätze vorhanden sind, wird gefragt „`Weiteren Teilnehmer eintragen [J/n]?`“. Wird die Frage mit „j“ beantwortet, wird wie bei der Auswahl des Fahrers eine Liste von Personen angezeigt, die noch keiner Fahrgemeinschaft zugeordnet sind. Man wählt wieder über den Index eine Person aus, die in die Fahrgemeinschaft übernommen wird. Wird die Frage mit „n“ beantwortet, oder sind die freien Plätze erschöpft, wird die Bewertungsfunktion auf die Fahrgemeinschaft angewendet und das Ergebnis präsentiert. Wird die Frage „`Fahrgemeinschaft übernehmen? [J/n]`“ mit „j“ beantwortet, wird sie in das System übernommen, sonst startet man wieder am Anfang dieses Use Case.

3.3.7.7 Fahrgemeinschaft ändern

Die Eigenschaften einer Fahrgemeinschaft sollen geändert werden. Dazu wird nach Auswahl des Menüpunktes `Fahrgemeinschaften-Fahrgemeinschaft ändern` eine Liste der Fahrgemeinschaften angezeigt (siehe Use Case 3.3.7.10). Daraus wählt man dann die betreffende Fahrgemeinschaft über den Index aus. Die änderbaren Eigenschaften der Fahrgemeinschaft werden mit Index angezeigt:

1. Anzahl der freien Plätze
2. Fahrer
3. Akzeptieren und Verlassen

Die zu ändernde Komponente wird über den Index angewählt. Die Anzahl der freien Plätze kann nicht größer werden als Autoplätze minus Teilnehmer. Wird der Fahrer gewählt, werden alle Personen in der Fahrgemeinschaft angezeigt, die potentielle Fahrer sind (siehe Use Case 3.3.7.2). Aus ihnen kann man über den Index einen neuen Fahrer auswählen. Ergibt sich durch die Wahl ein neuer Fahrer, wird die Frage „`Neuer Fahrer: Fahrtroute neu berechnen? [j/N]`“ ausgegeben und eine neue Wegberechnung durchgeführt, falls mit „j“ geantwortet wird. Bei „n“ wird der alte Fahrer beibehalten. Mit `Akzeptieren und Verlassen` werden die Änderungen übernommen.

3.3.7.8 Fahrgemeinschaft auflösen

Eine Fahrgemeinschaft aus der aktuellen Einteilung wird aufgelöst. Die Teilnehmer dieser Fahrgemeinschaft werden dabei nur in ihrer Markierung *fest eingetragen* oder *reserviert eingetragen* geändert, die auf *vermittelbar* gesetzt wird. Man wählt den Menüpunkt `Fahrgemeinschaften-Fahrgemeinschaft auflösen`. Eine Fahrgemeinschaft wird nach dem Anzeigen (Use Case 3.3.7.10) über ihren Index ausgewählt. Der Benutzer wird gefragt „`Fahrgemeinschaft` `auflösen? [J/n]`". Wird die Frage mit „j" beantwortet und war die Fahrgemeinschaft markiert, wird nachgefragt „`Fahrgemeinschaft ist markiert. Wirklich auflösen? [j/N]`". Wird die Frage auch mit „j" beantwortet, werden die Teilnehmer aus der Fahrgemeinschaft entfernt und die Fahrgemeinschaft aus dem System gelöscht. Sonst werden wieder die Fahrgemeinschaften angezeigt.

3.3.7.9 Fahrgemeinschaft bewerten

Die Qualität einer Fahgemeinschaft kann mit der aktuellen Bewertungsfunktion bewertet werden, indem man den Menüpunkt `Fahrgemeinschaften-Fahrgemeinschaft bewerten` aufruft. Eine Liste der Fahrgemeinschaften wird angezeigt (Use Case 3.3.7.10). Die zu bewertende Fahrgemeinschaft wird über ihren Index angewählt. Die Bewertungsfunktion wird auf diese Fahrgemeinschaft angewendet und das Ergebnis präsentiert. Man kann solange aus der Liste auswählen, die wieder angezeigt wird, bis man die Frage „`Weitere Fahrgemeinschaft bewerten? [J/n]`" mit „n" beantwortet.

3.3.7.10 Fahrgemeinschaft anzeigen

Eine Liste der Fahrgemeinschaften mit Einschränkung durch eine Filterfunktion soll angezeigt werden. Dazu wählt man den Menüpunkt `Fahrgemeinschaften-Fahrgemeinschaften anzeigen` und es wird der Filter wie in Use Case 3.3.7.1 aufgerufen. Anschließend wird eine Liste aller Fahrgemeinschaften zusammengestellt, die den Bedingungen des Filters genügen. Sie werden dabei aufsteigend numeriert und mit ID der Fahrgemeinschaft, einer Liste der Nachnamen der Teilnehmer und ihrer Markierung dargestellt, wobei der Fahrer besonders gekennzeichnet ist. Wählt man eine der Fahrgemeinschaften über ihren Index an, wird sie im Detail dargestellt. Die Liste der Fahrgemeinschaften und die detaillierte Anzeige können nach Use Case 3.4.1.4 durch Drücken der Taste „d" in eine PostScript-Datei ausgegeben werden.

Fahrgemeinschaft im Detail anzeigen

Eine Fahrgemeinschaft wird detailliert angezeigt. Dabei werden folgende Attribute aufgeführt:

- Anzahl der freien Plätze
- Teilnehmer der Fahrgemeinschaft (Vorname, Nachname, Status, Fahrer)
- Komfortklasse des Autos
- Startort
- Zielort
- Entstehungszeitpunkt
- Datum der letzten Änderung

- Markierung (*markiert*/*unmarkiert*)

Desweiteren kann man folgende zwei Punkte anwählen und sich anzeigen lassen:

1. Fahrtroute (graphisch/Straßennamen)
2. Zeitplan

3.3.7.11 Fahrgemeinschaft markieren/unmarkieren

Man möchte Fahrgemeinschaften vor dem Auflösen schützen oder dafür sorgen, daß eine markierte Fahrgemeinschaft doch wieder aufgelöst werden darf. Nach der Auswahl des Menüpunktes `Fahrgemeinschaften-Fahrgemeinschaften markieren/unmarkieren` wird der Benutzer gefragt „`Markieren oder Unmarkieren? [M/u]`". Nach der Wahl der Operation wird dem Benutzer eine Liste aller Fahrgemeinschaften angezeigt (siehe Use Case 3.3.7.10). Die Liste wird durch die Wahl der Operation eingeschränkt. Hat der Benutzer „m" gewählt, sieht er nur unmarkierte Fahrgemeinschaften, sonst nur markierte. Die zu verändernde Fahrgemeinschaft wird über ihren Index angewählt. Je nach Wahl der Operation wird die Frage „`Fahrgemeinschaft markieren? [J/n]`" oder „`Fahrgemeinschaft unmarkieren? [J/n]`" gestellt. Wird „j" gewählt und hatte die Fahrgemeinschaft vorher eine andere Einstellung, wird die neue eingesetzt. Danach sieht man wieder die Liste der Fahrgemeinschaften.

3.3.8 Vermittlung

3.3.8.1 Einteilung auswählen

Durch Auswahl des Menüpunkts `Vermittlung-Einteilung auswählen` kann man die aktuelle Einteilung wechseln. Es werden die Namen der zur aktuellen Personendatei gehörenden Einteilungen in einer numerierten Liste angezeigt. Die aktuelle Einteilung ist dabei voreingestellt als solche markiert. Wird eine der Einteilungen ausgewählt, so wird diese zur aktuellen Einteilung. Wird der Vorgang abgebrochen oder die bisher aktuelle Einteilung gewählt, so bleibt sie auch die aktuelle. Beim Wechseln der aktuellen Einteilung wird die Bewertungsfunktion der neuen aktuellen Einteilung zur aktuellen Bewertungsfunktion.

3.3.8.2 Systemmeldungen: ein/aus

Der Benutzer möchte über den aktuellen Stand der Berechnung durch Bildschirmmeldungen, informiert werden. Dazu muß der Menüpunkt `Vermittlung-Systemmeldungen: ein/aus` aufgerufen werden. Dieser verhält sich wie ein Wechselschalter (an oder aus). Bei jedem Aufruf ändert sich der Status und die aktuelle Einstellung wird am Bildschirm angezeigt. Danach gelangt man automatisch in das Untermenü Vermittlung zurück. Der Status hat keine Auswirkungen auf den Menüpunkt `Wegsuche-Systemmeldungen: ein/aus` (siehe Use Cases 3.3.10.5).

Wenn die Systemmeldungen eingeschaltet sind, werden bei der Berechnung einer Einteilung Informationen über den Stand und den Verlauf der Berechnung auf den Bildschirm und in eine Datei namens Mobidick.log ausgegeben.

3.3.8.3 Einteilung berechnen

Der Benutzer muß zur Berechnung einer Fahrgemeinschaftseinteilung einen von mehreren Algorithmen auswählen. Die Menge von Algorithmen ist in drei Algorithmenklassen eingeteilt. Es gibt heuristische, optimale und inkrementelle Algorithmen zur Berechnung einer Fahgemeinschaftseinteilung.

Bei den heuristischen und den optimalen Algorithmen kann der Benutzer Angaben zur Güte des Ergebnisses machen. Die möglichen Angaben beziehen sich auf die Anzahl der zu berechnenden Fahrgemeinschaften und auf die bestmögliche Bewertung, die eine Einteilung unter der aktuellen Bewertungsfunktion erreichen kann. Zur Eingabe der Güte werden dem Benutzer die Abfragen „Güte bezüglich der Fahrgemeinschaftenanzahl? [<min ♯FGM> - <max ♯FGM>]“ und „Güte bezüglich der Bewertung? [<min Bewertung> - <max Bewertung>]“ angezeigt. Der Benutzer kann dann je einen Wert innerhalb der angegebenen Grenzen eingeben. Gibt er keinen Wert sondern nur Enter ein, so werden die Werte <min ♯FGM> und <max Bewertung> als eingegeben angesehen. Die Kombination dieser Werte ergibt die beste mögliche Güte.

Der Wert <min ♯FGM> gibt an, wieviele Fahrgemeinschaften eine Einteilung mindestens enthalten muß, und wird aus der Größe der Personenmenge und der maximalen Fahrgemeinschaftsgröße berechnet. Die maximale Fahrgemeinschaftsgröße ist vier. Die maximale Anzahl der Fahrgemeinschaften (<min ♯FGM>) in einer Einteilung entspricht der Größe der Personenmenge. Die beste mögliche Bewertung (<max Bewertung>) und die schlechteste mögliche Bewertung (<min Bewertung>) werden passend zur aktuellen Bewertungsfunktion berechnet.

Heuristische Algorithmen sind Näherungsverfahren zur Einteilungsberechnung. Mit ihrer Hilfe soll es möglich sein, in kurzer Zeit eine gute, aber nicht unbedingt optimale Lösung zu finden. Bei der Berechnung einer Einteilung kann ein heuristischer Algorithmus auf die Personendaten, die aktuelle Bewertungsfunktion, eine vom Benutzer anzugebende Güte des Ergebnisses und die kürzesten Wege zwischen den Personen untereinander und zu ihren Arbeitsplätzen zugreifen.

Sobald eine Einteilung gefunden wurde, die der Güte des Benutzers entspricht, wird der Algorithmus beendet und die Einteilung als Ergebnis zurückgegeben. Ist während der Berechnung abzusehen, daß die Heuristik auf den gegebenen Personendaten keine Einteilung mit der vom Benutzer geforderten Güte findet, so kann der Vorgang mit einer entsprechenden Fehlermeldung abgebrochen werden. Am Ende einer erfolgreichen Berechnung wird die gefundene Einteilung mit der aktuellen Bewertungsfunktion bewertet und das Ergebnis dem Benutzer angezeigt.

Optimale Algorithmen sind Verfahren zur Einteilungsberechnung, die die beste Einteilung bezüglich der aktuellen Bewertungsfunktion finden. Optimiert wird nach der Anzahl der Fahrgemeinschaften, wobei die in der Bewertungsfunktion (siehe 3.3.9.1) enthaltenen Randbedingungen eingehalten werden müssen.

Bei der Einteilungsberechnung können die optimalen Algorithmen auf die gleichen Daten wie die heuristischen Algorithmen zugreifen.

Der Benutzer hat auch bei den optimalen Algorithmen die Möglichkeit, eine Güte für das Ergebnis anzugeben. Sobald eine Einteilung der entsprechenden Güte gefunden wurde wird der Algorithmus abgebrochen. Die dabei berechnete Einteilung ist aber nicht die optimale Einteilung sondern nur eine Einteilung mit einer Güte größer oder gleich der vom Benutzer geforderten Güte. Zur Berechnung der optimalen Einteilung muß die beste mögliche Güte vom Benutzer gefordert werden.

Inkrementelle Algorithmen sind Verfahren, die eine bestehende Fahrgemeinschaftseinteilung erweitern. Bei der Berechnung bleiben zunächst alle Fahrgemeinschaften bestehen und es wird versucht, die noch nicht vermittelten Personen in die Fahrgemeinschaften mit freien Plätzen einzufügen. Wenn dies nicht möglich ist, können auch neue Fahrgemeinschaften gebildet werden. Die Inkrementellen Verfahren bieten sich an, wenn die Personenmenge nur um wenige Personen erweitert wurde.

Wie unten beschrieben, kann der Benutzer nach Auswahl der Algorithmenklasse einen Algorithmus aussuchen. In dem Mobidick-System werden zu jeder Algorithmenklasse mindestens je ein Algorithmus implementiert. Die Dokumentation enthält Hinweise, wie ein neuer Algorithmus in den Quellcode des Systems eingefügt werden kann.

Der Menüpunkt Vermittlung-Einteilung berechnen führt in ein Untermenü, wo die Wahl zwischen verschiedenen Arten der Berechnung besteht:

1. heuristisch
2. optimal
3. inkrementell

Bei jeder Auswahl erscheint ein weiteres Menü der Form

1. Die [art] Einteilung berechnen
2. Algorithmus wechseln
3. Zurück
4. Hauptmenü
5. Hilfe

Dabei steht [art] für die vorher gewählte Art der Berechnung, also heuristisch, optimal oder inkrementell.

Mit dem ersten Menüpunkt startet man die entsprechende Berechnung. Nach Auswahl des zweiten Menüpunkts wird eine Liste mit Algorithmen angezeigt, die vom Typ [art] sind. Aus dieser Liste kann der Benutzer einen Algorithmus auswählen, wodurch dieser zum aktuellen Algorithmus seiner Art wird. Zu jeder

Art gibt es einen aktuellen Algorithmus, der ausgeführt wird, wenn der Benutzer mit dem Menüpunkt eins eine Berechnung seiner Art startet. Die übrigen Menüpunkte entsprechen den Erwartungen (siehe 3.4.1).

Nach dem Start eines Algorithmus wird entschieden, ob eine neue Einteilung berechnet werden soll, oder ob eine schon bestehende Einteilung erweitert wird. Ist die gewählte Art inkrementell, so wird die aktuelle Einteilung erweitert. Handelt es sich um eine optimale oder heuristische Berechnung, so wird eine neue Einteilung geöffnet. Dazu wird der Benutzer gefragt „`Aktuelle Einteilung speichern? [J/n]`“. Lautet die Antwort „j“, so wird sie gespeichert, ansonsten wird sie verworfen. Nun wird eine neue Einteilung geöffnet mit der Frage „`Name der neuen Einteilung:`“. Existiert der eingegebene Name schon, so erscheint die Fehlermeldung „`Fehler 2: Dieser Name existiert schon!`“ und die Eingabeaufforderung wird wiederholt.

Alle Algorithmen übernehmen die markierten Fahrgemeinschaften der beim Aufruf des Algorithmus aktuellen Einteilung. Diese Fahrgemeinschaften dürfen bei der Berechnung nicht aufgelöst, sondern nur erweitert werden, falls noch Plätze frei sind. Der Benutzer kann durch manuelle Änderung der freien Plätze die Erweiterbarkeit einer Fahrgemeinschaft einschränken (siehe 3.3.7.7). Durch das Markieren einer Fahrgemeinschaft kann der Benutzer erreichen, daß Fahrgemeinschaften aus der aktuellen Einteilung in der von einem Algorithmus neu berechneten Einteilung wieder enthalten sind.

Zur Einteilungsberechnung werden die kürzesten Wege zwischen allen Start- und Zielorten benötigt. Deshalb wird vor dem Start des Einteilungsalgorithmus überprüft, ob diese bereits berechnet wurden. Falls dies nicht der Fall ist, werden die kürzesten Wege mit dem im Untermenü `Auswahl des Algorithmus` in der Wegsuche (siehe 3.3.10.3) eingestellten Algorithmus berechnet.

Drückt man die Taste Esc bei einer laufenden Berechnung, so erscheint die Meldung „`Die Berechnung kann nicht wieder aufgenommen werden.`“ und die Frage „`Wirklich abbrechen? [j/N]?`“. Wird die Frage mit „j“ beantwortet, so wird die Berechnung abgebrochen (und keine Daten zur Fortsetzung werden gespeichert). Beantwortet man die Frage mit „n“, so wird sie fortgesetzt.

Nun wird die Berechnung gestartet, die durch Drücken der Taste „u“ (wie unterbrechen) unterbrochen werden kann. In diesem Fall werden alle zur Fortsetzung der Berechnung notwendigen Daten bei den zugehörigen Personendaten abgespeichert. Existieren schon Daten dieser Art, so werden sie überschrieben. Es kann somit immer nur die zuletzt abgebrochene Berechnung fortgesetzt werden (siehe Use Case 3.3.8.4). Ist die Laufzeitmessung eingeschaltet, so wird nach Abschluß der Berechnung die Laufzeit angezeigt.

3.3.8.4 Fortfahren mit letzter Berechnung

Durch Auswahl des Menüpunkts `Vermittlung-Fortfahren mit letzter Berechnung` kann man eine unterbrochene Berechnung fortsetzen. Wurde zuvor keine Berechnung unterbrochen, d.h. befindet sich keine Information über eine Berechnung in den Personendaten, so meldet das System „`Es wurde noch keine Berechnung durchgeführt`“ und kehrt ins Menü zurück. Ansonsten lädt

es die notwendigen Daten für die Fortsetzung der Berechnung und fährt mit der letzten Berechnung fort. Diese kann wieder mit der Taste „u“ unterbrochen werden (siehe 3.3.8.3).

3.3.8.5 Laufzeitmessung (ein/aus)

Der Menüpunkt `Vermittlung-Laufzeitmessung (ein/aus)` funktioniert wie ein Wechselschalter. Bei Programmstart ist der Status per Default aus. Da zum Zeitpunkt der Spezifikation noch nicht bekannt ist, wie dieser Punkt realisiert werden kann, gibt es hier mehrere Möglichkeiten. Die erste Möglichkeit hat bei der Entwicklung höchste Priorität:

Volle Integration

Ist die Laufzeitmessung eingeschaltet, so wird bei weiteren Einteilungsberechnungen die Laufzeit nach Abschluß der Berechnung angezeigt. Dieser Punkt hat keine Auswirkungen auf das Verhalten der Laufzeitmessung im Menü „Wegsuche“ (siehe Use Case 3.3.10.4).

Externe Laufzeitmessung

Das System zeigt dem Benutzer einen Informationstext an, in dem detailliert beschrieben wird, mit welchem Werkzeug er eine Laufzeitmessung auf den Algorithmen durchführen kann.

3.3.8.6 Einteilung bewerten

Bei Auswahl des Menüpunkts `Vermittlung-Einteilung bewerten` wird die Bewertungsfunktion auf die aktuelle Einteilung angewendet und das Ergebnis dem Benutzer präsentiert.

3.3.8.7 Einteilung anzeigen

Nach Auswahl des Menüpunkts `Vermittlung-Einteilung anzeigen` erhält der Benutzer eine Liste der Personen mit ihrem Namen, die nach Fahrgemeinschaften sortiert sind. Paßt diese Liste nicht auf den Bildschirm, wird er nach jeder Seite gefragt, ob er noch mehr sehen will oder nicht (siehe 3.4.1.7).

3.3.8.8 Einteilung auflösen

Wählt der Benutzer den Menüpunkt `Vermittlung-Einteilung auflösen`, so wird er gefragt „`Möchten Sie alle bis auf markierte Fahrgemeinschaften auflösen? [j/N]`“. Antwortet er mit „j“, so wird noch gefragt: „`Aktuelle Einteilung speichern? [J/n]`“ Beantwortert er auch diese Abfrage mit „j“, so werden alle bis auf markierte Fahrgemeinschaften gelöscht. Die aktuelle Einteilung besteht nun nur noch aus den markierten Fahrgemeinschaften. Alle anderen Personen sind nun wieder zu vermitteln.

3.3.9 Bewertungsfunktionen

3.3.9.1 Aufbau der Bewertungsfunktion

Eingangsgrößen

Zunächst eine Auflistung sämtlicher Größen, die bei der Bewertung der Güte einer Fahrgemeinschaft eingehen:

1. Umweg des Fahrers (bei mehreren potentiellen Fahrern der kleinste), und zwar relativ zu seinem alten Weg (vor Bildung der Fahrgemeinschaft) (s.u.).
2. Arbeitszeiten: jede Person gibt ein Ankunftsintervall und eine Arbeitsdauer an; in diesem Intervall muß die tatsächliche Ankunftszeit am Arbeitsplatz liegen. Die Arbeitsdauern der Teilnehmer einer Fahrgemeinschaft werden miteinander verglichen und dürfen nicht zu weit auseinanderliegen.
3. Wünsche der Personen (jeweils mit Gewicht):
   - abgelehnte Musikrichtungen
   - Geschlecht
   - Raucher
   - Komfortklasse des Autos
   - weitere

   Bei der Berechnung der Bewertungsfunktion wird für jede Person der Fahrgemeinschaft geprüft, ob ihre Wünsche mit den anderen Personen in Konflikt stehen. Dabei wird nur beachtet, ob ein Konflikt vorliegt oder nicht. Wieviele Personen nicht wunschgemäß sind, wird dabei nicht betrachtet. Jeder Wunsch trägt ein Gewicht zwischen 0 und 10 (unwichtig bzw. sehr wichtig). Wünsche mit dem Gewicht 10 werden als absolut verbindlich betrachtet, d.h. es wird keine Fahrgemeinschaft gebildet, in der diese verletzt würden.
4. Zu- und Abneigungen gegenüber bestimmten Personen:
   - Liste der erwünschten Personen
   - Liste der abgelehnten Personen

Bewertungsfunktion für Fahrgemeinschaften

Sei $P = \{p_1, \ldots, p_n\}$ die betrachtete Personenmenge, $M = \{f_1, \ldots, f_m\}$ eine Einteilung in Fahrgemeinschaften $f_i = \{p_{i_1}, \ldots, p_{i_j}\}$. $W = \{w_1, \ldots, w_k\}$ sei die Menge der Wünsche, $\gamma_1(p), \ldots, \gamma_k(p) \in \{0, 1, \ldots, 10\}$ die zugehörigen Gewichte für jede Person p und $A = \{a_1, \ldots, a_n\}$ eine Menge von Ankunftsintervallen der Personen in P. Die Bewertungsfunktion

$$\psi : M \rightarrow [0, 1]$$

bewertet eine Fahrgemeinschaft f_i mit:

$$\psi(f_i) = \begin{cases} 0, & \text{falls}(a_1 \cap \cdots \cap a_n) = \emptyset \vee \\ , & Z(f_i) = \infty \vee U(f_i) = \infty \vee \\ , & N_a(f_i) = \infty \vee P(f_i) = \infty \vee \\ , & E(f_i) = \infty \\ \psi'(f_i), & sonst \end{cases}$$

$$\psi'(f_i) = \frac{(1-Z(f_i)) \cdot G_Z + (1-U(f_i)) \cdot G_U + (1-N_a(f_i)) \cdot G_{N_a} + (N_e(f_i)) \cdot G_{N_e} + (P(f_i)) \cdot G_P + (1-E(f_i)) \cdot G_E}{G_Z + G_U + G_{N_a} + G_{N_e} + G_P + G_E}$$

wobei $G_Z, G_U, G_{N_a}, G_{N_e}, G_P$ und G_E die Gewichte sind, mit denen die Teilfunktionen Z, U, N_a, N_e, P, E in das Ergebnis von ψ' eingehen.

Die sechs Teilfunktionen nun im einzelnen:

1. $Z : M \rightarrow [0, 1] \cup \infty$ mit

   $$Z(f_i) = \begin{cases} \infty, & \text{falls } Z'(f_i) \geq Z_{max} \\ \frac{Z'(f_i)}{Z_{max}}, & sonst \end{cases}$$

   $$Z'(f_i) = \max_{p \in f_i}(\text{Arbeitsdauer}(p)) - \min_{p\,\mathrm{in} f_i}(\text{Arbeitsdauer}(p))$$

   Z berechnet die Bewertung der Fahrgemeinschaft f_i bezüglich der Arbeitszeiten. Z_{max} gibt an, wie groß der Abstand zwischen den Arbeitsdauern höchstens sein darf. Ist der Abstand größer, so kann die Fahrgemeinschaft nicht gebildet werden.

2. $U : M \rightarrow [0, 1] \cup \infty$ mit

   $$U(f_i) = \begin{cases} \infty, & \text{falls } U'(f_i) \geq U_{max} \\ \frac{U'(f_i)}{U_{max}}, & sonst \end{cases}$$

   $$U'(f_i) = \min_{p \in f_i} \min_{alle\ Wege\ w\ mit\ Fahrer\ p} U(p, w)$$

   , wobei $U(p, w) = d(w) - d(s_p, z_p)$ den Umweg von Fahrer p beim Fahrgemeinschaftsweg w bezeichnet. $d(w)$ ist die Länge des Weges w und $d(s_p, z_p)$ ist die Weglänge zwischen Start- und Zielort von p.

3. $N_a : M \rightarrow [0, 1] \cup \infty$ mit

   $$N_a(f_i) = \begin{cases} \infty, & \text{falls } N'_a(f_i) \geq N_{a\ max} \\ N'_a(f_i), & sonst \end{cases}$$

   $$N'_a(f_i) = \frac{1}{(|f_i| - 1) \cdot |f_i|} \cdot \sum_{p \in f_i} \#\ abgelehnte\ Personen\ aus\ Sicht\ von\ p$$

   Für jede Person p der Fahrgemeinschaft wird die Anzahl der Personen in f_i summiert, die von p abgelehnt werden. Übersteigt die Anzahl $N_{a\ max}$, so soll die Fahrgemeinschaft nicht gebildet werden.

4. $N_e : M \to [0,1]$ mit

$$N_e(f_i) = \frac{1}{(|f_i|-1)\cdot|f_i|}\cdot\sum_{p\in f_i} \#\ \textit{erwünschte Personen aus Sicht von } p$$

Für jede Person p der Fahrgemeinschaft wird die Anzahl der Personen in f_i summiert, die von p erwünscht sind.

5. $P : M \to [0,1] \cup \infty$ mit

$$P(f_i) = \begin{cases} \infty, & \text{falls } |f_i| \geq P_{max} \\ \frac{|f_i|}{P_{max}}, & sonst \end{cases}$$

P bewertet die Auslastung der Fahrzeuge. P_{max} gibt an, wieviele Personen höchstens zu einer Fahgemeinschaft gehören dürfen.

6. $E : M \to [0,1] \cup \infty$ mit

$$E(f_i) = \begin{cases} E'(f_i), & \text{falls } \gamma_j(p)\cdot K(p,j,f_i) \neq 10 \\ , & \forall p \in f_i, \forall j \in \{1,\ldots,k\} \\ , & E'(f_i) \leq E_{max} \\ \infty, & \text{sonst} \end{cases}$$

$$E'(f_i) = (\sum_{p\in f_i}\sum_{j=1}^{k}\frac{\gamma_j(p)}{(|P|-1)\cdot 10}\cdot K(p,j,f_i))\cdot\frac{1}{k\cdot|f_i|}$$

Mit $K(p,j,f_i)$ wird überprüft, ob der Wunsch w_j von Person p mit der Fahrgemeinschaft f_i in Konflikt steht ($K : P \times \{1,\ldots,k\} \times M \to [0,1]$):

$$K(p,j,f_i) = \begin{cases} 1, & \text{falls Wunsch } w_j \text{ von } p \text{ in Konflikt mit } f_i \\ 0, & \text{sonst} \end{cases}$$

Für $E(f_i)$ ergibt sich der Wert ∞ , falls ein Wunsch mit Gewicht 10 verletzt wird oder falls $E'(f_i) \leq E_{max}$. Diese Fahrgemeinschaft kommt dann auf jeden Fall nicht zustande. $E(max)$ gibt an wie schlecht $E'(f_i)$ höchstens sein darf.

Die Parameter einer Bewertungsfunktion sind: Z_{max}, U_{max}, $N_{a\,max}$, P_{max}, E_{max}, G_Z, G_U, G_{N_a}, G_{N_e}, G_P und G_E.

Bewertungsfunktion für Einteilungen

Die Bewertungsfunktion

$$h : M \to [0,1]$$

bewertet eine Einteilung M mit:

$$h(M) = \sum_{f\in M}\psi(f)\cdot\frac{1}{|M|}\cdot\frac{\sum_{f\in M\wedge\psi(f)\neq 0}|f|}{|P|}$$

Die Bewertung einer Einteilung ergibt sich somit aus der Summe der Fahrgemeinschaftsbewertungen normiert auf $[0,1]$ und multipliziert mit $\frac{\sum_{f\in M\wedge\psi(f)\neq 0}|f|}{|P|}$. Die Summe $\sum_{f\in M\wedge\psi(f)\neq 0}|f|$ gibt an, wieviele Personen in gültige Fahrgemeinschaften eingeteilt sind und $|P|$ ist die Anzahl der zu vermittelnden Personen. Gültige Fahrgemeinschaften sind Fahrgemeinschaften, die eine Bewertung $\psi(f) \geq 0$ haben.

3.3.9.2 Neue Bewertungsfunktion anlegen

Durch Auswahl des Menüpunkt `Bewertungsfunktionen-neu` wird eine neue Bewertungsfunktion angelegt. Nach Auswahl des Menüpunkts wird dem Benutzer eine Liste mit allen geöffneten Bewertungsfunktionen angezeigt. Dann wird er aufgefordert den neuen Dateinamen anzugeben durch die Meldung „`Neuer Name:`“. Gibt der Benutzer einen schon existierenden Namen ein, so wird die Fehlermeldung „`Fehler 2: Dieser Name existiert schon!`“ angezeigt und die Eingabeaufforderung wiederholt.

An den Dateinamen wird vom System die Erweiterung `.fkt` angehängt, mit der die Datei als Bewertungsfunktion gekennzeichnet wird. Nach Eingabe des Namens kann der Benutzer wie in 3.3.9.3 die Parameter der neuen Bewertungsfunktion anpassen.

3.3.9.3 Bewertungsfunktion ändern

Nach Auswahl des Menüpunkts `Bewertungsfunktionen-ändern` können die Parameter der aktuellen Bewertungsfunktion verändert werden. Die Parameter werden mit Index präsentiert. Wählt man einen aus, wird man aufgefordert den neuen Wert einzugeben. Nicht korrekte Werte werden nicht angenommen und es wird die Fehlermeldung „`Fehler 1: Ungültige Eingabe!`“ angezeigt. Gibt man einen korrekten Wert ein, wird dieser übernommen. Nach erfoglreicher Änderung wird man gefragt, ob die Eingabe übernommen werden soll. Bei Bestätigung der Frage werden die Änderungen übernommen und der Menüpunkt wird verlassen, sonst erhält man die Möglichkeit weitere Parameter zu ändern.

3.3.9.4 Bewertungsfunktion auswählen

Nach Auswahl des Menüpunkts `Bewertungsfunktionen-auswählen` wird eine Liste aller geöffneten Bewertungsfunktionen angezeigt. Aus dieser Liste kann der Benutzer eine Bewertungsfunktion auswählen, die dann zur aktuellen wird.

3.3.9.5 Bewertungsfunktion anzeigen

Nach Auswahl des Menüpunkts `Bewertungsfunktionen-anzeigen` werden die Parameter der aktuellen Bewertungsfunktion am Bildschirm angezeigt.

3.3.10 Wegsuche

3.3.10.1 Einzelwegsuche

Mit dem Menüpunkt `Wegsuche-Einzelwegsuche` kann der Benutzer eine Wegsuche von Straße A nach Straße B durchführen, ohne dabei eine Fahrgemeinschaft betrachten zu müssen. Nach Auswahl des Menüpunkts wird überpüft, ob überhaupt ein Verkehrsgraph geladen wurde. Wenn dies nicht der Fall ist, erscheint die Fehlermeldung „`Fehler 8: Kein Verkehrsgraph geladen.`“ und der Vorgang wird abgebrochen. Wenn ja, wird eine Bildschirmmaske angezeigt, in der die Start- und Zielstraße eingegeben werden muß. Dabei wird jeweils überprüft, ob der Straßenname mehrmals im System vorkommt. Ist dies der Fall, wird eine Liste aller IDs, die den entsprechenden Namen tragen, anzeigt und der Benutzer muß zum Namen noch zusätzlich die ID angeben. Wird nun die Frage „`Wegsuche starten? [J/n]`“ mit „j“ beantwortet, startet die Suche. Ist die Suche erfolgreich, erscheint der Text „`Wegsuche erfolgreich. Route gefunden`“ und die gefundene Route kann mit Hilfe einer Straßenliste oder einem Verkehrsgraphen angezeigt werden. Möchte man einen Ausdruck von dieser Route haben, so kann durch Drücken der Taste „d“ eine PostScript-Datei erstellt werden.

Wenn sich im System überhaupt kein oder kein gültiger Verkehrsgraph befindet, der zu den Straßen paßt, wird der gesamte Vorgang abgebrochen.

3.3.10.2 n-Wegesuche

Mit dem Menüpunkt `Wegsuche-n-Wegesuche` kann der Benutzer die Zeit anzeigen lassen, die der aktuelle Wegsuchealgorithmus zur Berechnung von allen kürzesten Wegen zwischen n zufällig gewählten Paaren aus Start- und Zielorten benötigt. Nach Auswahl des Menüpunkts wird überpüft, ob überhaupt ein Verkehrsgraph geladen wurde.

Wenn dies nicht der Fall ist, erscheint die Fehlermeldung „`Fehler 8: Kein Verkehrsgraph geladen.`“ und der Vorgang wird abgebrochen. Sonst wird der Benutzer aufgefordert die Zahl n einzugeben. Nach der Eingabe werden n Wege zwischen zufällig gewählten Start- und Zielorten berechnet und die dafür benötigte Laufzeit wird ausgegeben.

3.3.10.3 Auswahl des Algorithmus

Mit dem Menüpunkt `Wegsuche-Auswahl des Algorithmus` kann der Benutzer einen Wegsuchealgorithmus aus einer Liste über einen Index auswählen. Der ausgewählte Algorithmus wird bei der Wegsuche in 3.3.10.1 und in 3.3.8.3 verwendet.

3.3.10.4 Laufzeit

Der Menüpunkt `Wegsuche-Laufzeit` funktioniert wie ein Wechselschalter. Bei Programmstart ist der Status per Default aus. Ist die Laufzeitmessung eingeschaltet, so wird bei der nächsten Wegsuche die Laufzeit nach Abschluß der Berechnung angezeigt. Dieser Punkt hat keine Auswirkungen auf das Verhalten der Laufzeitmessung im Menü `Vermittlung` (siehe Use Case 3.3.8.5).

3.3.10.5 Systemmeldungen: ein/aus

Der Benutzer möchte über den aktuellen Stand der Berechnung, durch Bildschirmmeldungen, informiert werden. Dazu muß der Menüpunkt `Wegsuche-Systemmeldungen: ein/aus` einfach nur aufgerufen werden. Dieser verhält sich wie ein Wechselschalter. Bei jedem Aufruf ändert sich der Status und die aktuelle Einstellung wird am Bildschirm angezeigt. Danach gelangt man automatisch in das Untermenü zurück. Der Status hat keine Auswirkungen auf die 'Systemmeldung: ein/aus' im Menü `Vermittlung` (siehe Use Case 3.3.8.2).

3.3.11 Voreinstellungen

Nach Anwahl des Menüpunkts `Voreinstellungen` hat der Benutzer die Möglichkeit, Pfade und Namen der beim Systemstart zu öffnenden Dateien und Default-Pfade anzugeben. Die möglichen Eingaben sind:

- Default-Pfad für Personendateien
- Name der zu öffnenden Personendatei
- Default-Pfad für Verkehrsgraphen
- Name des zu öffnenden Verkehrsgraphen
- Default-Pfad für Bewertungsfunktionen

Wenn keine Datei geöffnet werden soll, so wird kein Name sondern die Zeichenkette `leer` eingegeben. Die Zeichenkette `leer` als Pfad entspricht dem Pfad \. Beim Beenden der Eingabe wird der Benutzer gefragt: „`Änderungen speichern? [J/n]`“.

Beim Beenden des Mobidick-Systems werden die in den Voreinstellungen angegebenen Daten in der Datei voreinstellungen.mbd abgespeichert.

3.4 Anforderungen an externe Schnittstellen

3.4.1 Benutzungsschnittstelle

3.4.1.1 Menüpunkt Zurück

Wird dieser Menüpunkt gewählt, wird das aktuelle Menü verlassen und man befindet sich eine Ebene höher.

3.4.1.2 Menüpunkt Hauptmenü

Wird dieser Menüpunkt gewählt, wird das aktuelle Menü verlassen und man befindet sich im Hauptmenü.

3.4.1.3 Menüpunkt Hilfe

Wird dieser Menüpunkt aufgerufen, wird dem Benutzer ein Hilfetext präsentiert. Er enthält Informationen darüber, wo man sich in der Menüstruktur befindet, wie man hierher kam, wohin man von hier aus gehen kann, was man in diesem Menü alles anwählen kann und was dies bewirkt.

3.4.1.4 PostScript-Ausgabe

Einige angezeigte Daten kann man zum späteren Drucken in eine PostScript-Datei ausgeben lassen. Dazu drückt man, wenn die entsprechenden Daten angezeigt werden die Taste "d". Nun bekommt man eine Liste aller Dateien mit der Endung ".ps" in dem aktuellen Verzeichnis angezeigt und erhält eine Eingabeaufforderung für den Namen der Datei. Wählt man einen bereits existierenden Namen aus, wird man gefragt „`Datei existiert bereits. Überschreiben? [j/N]`“. Antwortet man mit "j", wird die Datei überschrieben und man sieht wieder die Daten. Antwortet man mit "n", wird die Eingabeaufforderung wiederholt.

3.4.1.5 Markierter Eintrag

Eingestellte Auswahlmenüeinträge werden mit einem $*$ vor der Nummer markiert. Wird bei einer solchen Auswahl anstelle einer Nummer die Return-Taste gedrückt, wird der mit $*$ markierte Eintrag ausgewählt.

3.4.1.6 Escape

Durch Drücken der Escape-Taste wird die aktuelle Bearbeitung einer Bildschirmmaske abgebrochen und der Ausgangszustand wird wieder hergestellt. Danach gelangt man in das Menü zurück, aus dem die Aktion gestartet wurde.

3.4.1.7 Darstellung von Listen

Aufgrund der Länge kann die vollständige Darstellung einer Liste im aktuellen Fenster zu Problemen führen. Daher wird zunächst eine Fensterseite angezeigt. Beanwortet man nun die Frage „`Weiter? [J/n/=]`“ mit „j“, wird die nächste Fensterseite angezeigt. Wählt man die Alternative „n“, wird die Darstellung abgebrochen und bei „=“ werden die restlichen Seiten ohne Unterbrechung ausgegeben.

3.4.1.8 Abfragen

Der Benutzer kann im System alle Abfragen auf zwei Arten beantworten. Zum einen durch Drücken der zugelassenen Zeichen, dabei wird nicht auf die Groß- und Kleinschreibung geachtet und zum anderen durch Drücken der Return-Taste. Die Return-Taste löst dabei die Aktion aus, die dem Großbuchstaben entspricht.

3.4.1.9 Dialoge

Es gibt insgesamt drei verschiedene Arten von Dialogen, die auf unterschiedliche Weise bedient werden.

Menü: Nachdem das System gestartet wird, erscheint eine allgemeine Bildschirmanzeige, die eine Reihe von Auswahleinträgen besitzt. Diese Einträge können in der Kommandozeile über den entsprechenden Index angewählt werden.

Abfrage von Einzeldaten: Ruft man eine Bildschirmanzeige mit Werten zum ersten Mal auf, steht hinter jedem Eintrag ein Default-Wert. Will man diesen Default-Wert ändern, muß der entsprechende Index in der Kommandozeile angewählt werden. Danach kann der Wert eingegeben werden. Entspricht dieser dem Definitionsbereich, so wird der Wert übernommen und in die Bildschirmanzeige eingefügt. Ansonsten muß der Wert neu eingegeben werden.

Abfrage von kompletten Datensätzen: Ruft man eine Bildschirmanzeige auf, in der komplette Datensätze eingegeben werden müssen, so erfolgt eine automatische Abfrage der einzelnen Felder. Am Ende wird der komplette Datensatz präsentiert und durch Beantworten der Frage: „`Alles Korrekt? [J/n]`“ mit „j“, wird der Datensatz ins System aufgenommen. Wenn nein, werden die Abfragen nochmals durchgegangen. Unterschied zu voher ist, daß der Wert vom vorherigen Durchlauf in der Bildschirmanzeige steht und nur noch dort geändert werden muß, wo ein Fehler ist.

3.4.2 Hardwareschnittstellen

3.4.2.1 Drucken

Folgende Daten können in einer PostScript-Datei abgelegt werden:

- Ergebnis der Suche mit Personenfilter
- Ergebnis der Suche mit Fahrgemeinschaftenfilter
- Ergebnis der Wegsuche (textuell)
- Einteilung als Personenliste

3.4.2.2 Sekundärspeicher

Das System Mobidick verwendet folgende Dateien:

- Personendaten mit zugehörigen Einteilungen: Endung `.per`
- Verkehrsgraphen: Endung `.gra`
- Parametereinstellung für Bewertungsfunktion: Endung `.fkt`
- Logfile für Systemmeldungen bei Weg- und Einteilungsberechnung: Endung `.log`
- Die Datei `voreinstellungen.mbd`, in der die Voreinstellungsparameter enthalten sind.

Die entsprechenden Dateiformate werden im Entwurf festgelegt. Bei allen Lade- und Speicheroperationen müssen Name und Pfad frei wählbar sein.

3.4.3 Softwareschnittstellen

Das System Mobidick kommuniziert mit einem externen Graphenviewer, dies ist in 3.5.4 beschrieben.

3.5 Leistungsanforderungen

Das System Mobidick ist für den Einbenutzerbetrieb ausgelegt und läuft auf *einem* Terminal.

3.5.1 Dateien

Das System soll mit beliebig vielen Einteilungs-, Verkehrsgraphen- und Bewertungsfunktionsdateien umgehen können. Bei den Verkehrsgraphen sollte das Stadtgebiet von Stuttgart verarbeitet werden können.

3.5.2 Daten im Hauptspeicher

Es wird immer nur ein Verkehrgraph im Hauptspeicher gehalten. Personen-, Einteilungs- und Bewertungsfunktionsdaten können solange angelegt werden, bis der Speicher voll ist.

3.5.3 Antwortzeiten

Die folgenden Funktionen sollen interaktiv und dementsprechend schnell sein:

- Personenfilter

- Filter für Fahrgemeinschaften
- Bewertung einer Fahrgemeinschaft
- Bewertung einer Einteilung
- Suchen einer Fahrgemeinschaft beim manuellen Einfügen
- Wegsuche auf dem hierarchischen Graphen

Für die optimale, heuristische und inkrementelle Berechnung von Einteilungen muß mit längeren Berechnungszeiten (Stunden, Tage) gerechnet werden.

3.5.4 Entwurfseinschränkungen

Das System Mobidick muß auf dem Rechner tagetes des Rechnerpools der Abteilung Formale Konzepte unter Solaris 5.4 laufen. Als Programmiersprache wird C++ verwendet, und die Version 2.7.2.2 des gnu-Compilers. Einige Funktionen und Datentypen entstammen der Version 3.4.2 von LEDA.

Da bereits ein Graphenviewer zur Anzeige von Verkehrsgraphen in C++ implementiert wurde, muß dieser in die Entwurfsüberlegungen einbezogen werden. Da dieser als eigenständiges Programm bestehen bleiben soll, muß eine Kommunikationsschnittstelle zwischen beiden Programmen festgelegt werden. Zur Anzeige von kürzesten Wegen müssen die entsprechenden Daten an den Graphenviewer übergeben werden.

Zunächst wird nur eine auf Text basierende Benutzungsoberfläche implementiert, eine graphische Oberfläche erscheint für einen Prototypen zu aufwendig.

An Verkehrsdaten liegt uns bisher der Stadtplan von Stuttgart im GDF-Format vor (GDF-Version 2.1). Für die Umwandlung der GDF-Daten in eine Verkehrsgraphendatei wurde ein Perlskript implementiert (siehe Anhang A). Dafür wurde ein vorläufiges Verkehrsgraphenformat festgelegt. Für das Konvertierungsprogramm wird noch eine Dokumentation erstellt, in der Quell-, Zielformat und die Umwandlung beschrieben werden.

3.5.5 Attribute

Da es sich um einen Prototyp handelt, wird nichts zur Verfügbarkeit und zur Sicherheit ausgesagt. Die Wartbarkeit wird durch noch festzulegende Codierungs- und Entwurfsrichtlinien erreicht. Die Portabilität wird durch folgende Maßnahmen erleichtert:

- ausschließliche Verwendung von überall verfügbaren C++-Bibliotheken
- separate Module für die Programmteile, die auf Bildschirm und Dateien zugreifen.

3.6 Zukünftige Erweiterungen

Wochentage: Jeder Teilnehmer kann für jeden Wochentag eine andere Arbeitszeit angeben. Beispiel: Mo 9.00-17.00 Uhr, Di 8.00-16.00 Uhr, Mi 10.00-13.00 Uhr.

Hin- oder Rückfahrt: Der Teilnehmer kann angeben, ob er mit der Fahrgemeinschaft beide Wege oder nur einen Weg fahren will.

Statistische Angaben: Der Benutzer kann statistische Daten über das FGM-System erfahren. Beispielweise die Auslastung der Fahrzeuge, die durchschnittliche Personenzahl einer FGM und weitere mehr.

Zwischenpunkte bei der Wegsuche: Der Teilnehmer kann zusätzliche Straßennamen angeben, die bei der Wegsuche berücksichtigt werden und auf jeden Fall in der Route enthalten sind.

Verknüpfungen: Der Benutzer möchte eine Liste aller reservierten Personen. Danach wählt er eine Person aus. Dabei besteht die Möglichkeit die entsprechende FGM der Person, mit allen anderen Teilnehmern, anzeigen zu lassen.

Sammelpunkte: Man möchte Orte angeben, an denen sich Teilnehmende einer FGM treffen, um von dort loszufahren, oder sich nach dem Ankommen zu zerstreuen.

Graphische Benutzungsoberfläche: Das System besitzt eine graphische Benutzungsoberfläche (Fenster, Pulldown-Menüs usw.) und kann mit Hilfe einer Maus gesteuert werden.

3.7 Systemmeldungen

3.7.1 Meldungen

- „Änderungen für Fahrgemeinschaftsbildung relevant“ (3.3.6.1)
- „Person ist Fahrer, löschen führt zur Auflösung einer Fahrgemeinschaft“ (3.3.6.1)
- „Teilnehmer schon fest eingetragen!“ (3.3.7.4)
- „Vorsicht. Durch Loeschen des Fahrers wird die Fahrgemeinschaft aufgelöst“ (3.3.7.5)
- „Die Berechnung kann nicht wieder aufgenommen werden“ (3.3.8.3)
- „Es wurde noch keine Berechnung durchgeführt“ (3.3.8.4)
- „Wegsuche erfolgreich. Route gefunden“ (3.3.10.1)

3.7.2 Fragen

- „Personendatei <Name> vor dem Beenden speichern? [J/n]“ (3.3.4)
- „Neuer Name:“ (3.3.5.1)
- „Personendatei <Name> vor dem Erstellen der neuen Datei speichern? [J/n]“ (3.3.5.1)
- „Name existiert schon, trotzdem abspeichern und vorhandene Datei überschreiben? [j/N]“ (3.3.5.1, 3.3.5.4, 3.3.5.8, 3.3.5.9, 3.3.5.13)
- „Personendatei <Name> vor dem Laden der anderen Datei speichern? [J/n]“ (3.3.5.2)
- „Datei vor dem Schließen speichern? [J/n]“ (3.3.5.5)
- „Bewertungsfunktion <Name> vor dem Schließen speichern? [J/n]“ (3.3.5.5)
- „Name:“ (3.3.5.6)
- „Einteilung wirklich löschen? [j/N]“ (3.3.5.10)
- „Person wirklich übernehmen? [J/n]“ (3.3.6.1)
- „Änderungen vornehmen und Person aus den betroffenen Fahrgemeinschaften löschen? [J/n]“ (3.3.6.1)
- „Änderungen vornehmen und Fahrgemeinschaft auflösen? [J/n]“ (3.3.6.1)
- „Person wirklich löschen? [J/n]“ (3.3.6.1)
- „Haben Sie die alte Personenmenge gesichert? [j/N]“ (3.3.6.1)
- „Personenmenge übernehmen? [J/n]“ (3.3.6.1)
- „Tabelle in PostScript-Datei ausgeben? [j/N]“ (3.3.6.1)
- „Person in Fahrgemeinschaft aufnehmen? [J/n]“ (3.3.7.3, 3.3.7.3)
- „Fahrgemeinschaft übernehmen? [J/n]“ (3.3.7.3, 3.3.7.3, 3.3.7.6)
- „Teilnehmer fest eintragen? [J/n]“ (3.3.7.4)
- „Person aus Fahrgemeinschaft löschen? [J/n]“ (3.3.7.5)
- „Fahrer löschen? [j/N]“ (3.3.7.5)
- „Person als Fahrer übernehmen? [J/n]“ (3.3.7.6)
- „Weiteren Teilnehmer eintragen? [J/n]“ (3.3.7.6)
- „Neuer Fahrer: Fahrtroute neu berechnen? [j/N]“ (3.3.7.7)
- „Fahrgemeinschaft auflösen? [J/n]“ (3.3.7.8)

- „`Fahrgemeinschaft ist markiert. Wirklich auflösen? [j/N]`“ (3.3.7.8)
- „`Weitere Fahrgemeinschaft bewerten? [J/n]`“ (3.3.7.9)
- „`Markieren oder Unmarkieren? [M/u]`“ (3.3.7.11)
- „`Fahrgemeinschaft markieren? [J/n]`“ (3.3.7.11)
- „`Fahrgemeinschaft unmarkieren? [J/n]`“ (3.3.7.11)
- „`Güte bezüglich der Fahrgemeinschaftenanzahl? [<min ♯FGM> - <max ♯FGM>]`“ (3.3.8.3)
- „`Güte bezüglich der Bewertung? [<min Bewertung> - <max Bewertung>]`“ (3.3.8.3)
- „`Aktuelle Einteilung speichern? [J/n]`“ (3.3.8.3)
- „`Name der neuen Einteilung:`“ (3.3.8.3)
- „`Wirklich abbrechen? [j/N]?`“ (3.3.8.3)
- „`Möchten Sie alle bis auf markierte Fahrgemeinschaften auflösen? [j/N]`“ (3.3.8.8)
- „`Wegsuche starten? [J/n]`“ (3.3.10.1)
- „`Änderungen speichern? [J/n]`“ (3.3.11)
- „`Datei existiert bereits. Überschreiben? [j/N]`“ (3.4.1.4)
- „`Weiter? [J/n/=]`“ (3.4.1.7)
- „`Alles Korrekt? [J/n]`“ (3.4.1.9)

3.7.3 Fehler

- „`Fehler 1: Ungültige Eingabe!`“ (3.3.9.3)
- „`Fehler 2: Dieser Name existiert schon!`“ (3.3.5.1, 3.3.5.4, 3.3.5.8, 3.3.5.9, 3.3.5.13)
- „`Fehler 3: Adresse nicht im Verkehrsgraphen vorhanden`“ (3.3.6.1)
- „`Fehler 4: Kante nicht im Verkehrsgraphen vorhanden`“ (3.3.6.1)
- „`Fehler 5: Obere Grenze kleiner untere Grenze`“ (3.3.6.1)
- „`Fehler 6: Person bereits vorhanden`“ (3.3.6.1)
- „`Fehler 7: Person nicht in Liste vorhanden`“ (3.3.6.1)
- „`Fehler 8: Kein Verkehrsgraph geladen`“ (3.3.10.1)

Kapitel 4

Abweichungen von der Spezifikation

Das Mobidick-System wurde aus Zeitgründen nicht komplett implementiert. Im folgenden Text werden die Punkte aufgeführt, in denen die Implementierung von der Spezifikation abweicht. Außer den aufgeführten Punkten kann es vereinzelt Abweichungen in der Formulierung der Bildschirmmeldungen geben. Es fehlen auch diverse Sicherheitsabfragen.

- Start des Fahrgemeinschaftensystems 3.3.1

  Beim Start werden nicht die in den Voreinstellungen angegebenen Dateien geladen. Sie sind jedoch der Default beim Laden einer neuen Datei.

- Voreinstellungen 3.3.11

  Der Verkehrsgraph kann in den Voreinstellungen nicht eingestellt werden. In der Datei `mobidick.cc` wird explizit der Verkehrsgraph von Stuttgart geladen.

  Die Voreinstellungen können nicht von Mobidick aus verändert werden. Alle Änderungen müssen mit einem Editor vor dem Start von Mobidick direkt in der Datei `voreinstellungen.mbd` eingegeben werden.

  Die Datei voreinstellungen.mbd enthält einen zusätzlichen Parameter, der angibt, wie groß eine Fahrgemeinschaft höchstens sein darf.

- Menüstruktur 3.3.2 und Voreinstellungen 3.3.11

  Der Menüpunkt 7. `Voreinstellungen` existiert nicht mehr. Stattdessen editiert der Benutzer die Datei `voreinstellungen.mbd` selbst.

- Menüstruktur 3.3.2 und Verkehrsdaten laden 3.3.5.7

  Der Menüpunkt 1b. `Verkehrsgraph laden` existiert nicht mehr. Es wird ein Verkehrsgraph mit fest eingestelltem Pfad geladen.

- Personen anzeigen 3.3.6.5

Zur Auswahl einer Person wird lediglich Vorname, Nachname und Wohnort angezeigt. Die Ausgabe in eine PostScript–Datei ist nicht implemetiert. Durch Anwahl der Indexnummer werdem dem Benutzer alle Personeninformationen präsentiert.

- Neue Person eintragen 3.3.6.1

  Zusätzlich zur Adresse einer Person soll sie die Adresse ihres Startorts und ihres Zielorts eingeben. Daraus kann man eventuell die Start- und Zielkoordinaten errechnen. Dies wurde nicht implementiert. Deswegen werden Start- und Zieldaten nicht erfragt, sondern die entsprechenden Koordinaten werden direkt durch Anklicken im Verkehrsgraphen eingegeben.

  Die dynamisch erweiterbaren Personeneigenschaften werden bei der Eingabe der Personendaten nicht abgefragt, da alle Systemteile, die sich auf die dynamischen Eigenschaften beziehen, nicht implementiert wurden. Der Benutzer wird nur zu den zwei statisch implementierten Eigenschaften "Raucher" und "Geschlecht" befragt.

  Die Eingabe von Wünschen mit ihren Gewichten bezüglich der Personeneigenschaften bezieht sich ebenfalls nur auf die beiden implementierten statischen Eigenschaften.

- Personen generieren 3.3.6.4

  Die Rechteckverteilung der Start- und Zielorte wurde nicht implemtiert. Aus diesem Grund wird bei negativer Beantwortung der Frage nach der Gleichverteilung sofort die Frage nach dem festen Start- bzw. Zielort gestellt.

- Personen anzeigen 3.3.6.5

  Der Punkt 12 *Status* wird bei der Abfrage der Filterdaten nicht angeboten.

  Der Status einer Person kann nur reserviert oder fest sein, die Möglichkeit einer Person den Status vermittelbar zu geben ist nicht implementiert.

  Die Ausgabe einer Personenauswahl in eine Postscript-Datei wurde nicht implementiert.

- Personeneigenschaften erweitern 3.3.6.6

  Die Möglichkeit Personeneigenschaften dynamisch zu erweitern wurde nicht implementiert.

- Fahrgemeinschaften-Filter 3.3.7.1

  Das Filterkriterium 9 *Markierung* wird nicht angeboten. Die Daten werden nicht einzeln gesetzt und durch *Akzeptieren und Weiter* bestätigt, sondern nacheinander abgefragt. `Return` bedeutet dabei, daß das Kriterium nicht gesetzt ist.

- Teilnehmer anzeigen 3.3.7.2

  Die Teilnehmer werden nicht numeriert angezeigt. Es werden Vorname, Name und Status angezeigt.

- Neuer Teilnehmer 3.3.7.3

  Das Suchsystem wurde nicht implementiert.

- Fahrgemeinschaft aendern 3.3.7.7

  Die Fahrtroute wird bei Fahreränderung nicht neu berechnet.

- Fahrgemeinschaft anzeigen 3.3.7.10

  Die Fahrtroute wird automatisch angezeigt. Der Zeitplan wird nicht angezeigt.

- Systemmeldungen: ein/aus 3.3.8.2

  Die aktuelle Einstellung wird nicht am Bildschirm angezeigt. Die Logdatei wird nicht erstellt.

- Einteilung berechnen 3.3.8.3 und Fortfahren mit letzter Berechnung 3.3.8.4

  Eine laufende Berechnung kann nicht konsistent unterbrochen werden. Bei Drücken von Ctrl-C wird das gesamte Programm beendet. Unterbrochene Berechnungen können nicht wieder aufgenommen werden.

  Die Möglichkeit, beim Aufruf eines optimalen oder heuristischen Algorithmus Angaben zur Güte und der Anzahl der zu berechnenden Fahrgemeinschaften zu machen, ist nicht immer gegeben. Die Abfrage von Parametern zur Beeinflussung einer Berechnung ist abhängig vom jeweiligen Algorithmus, und kann deshalb für jeden Algorithmus unterschiedlich sein.

- Laufzeitmessung (ein/aus) 3.3.8.5

  Es wurde die Variante *Volle Integration* implementiert.

- Einteilung auflösen 3.3.8.8

  Diese Methode wurde nicht implementiert.

- Laufzeit 3.3.10.4

  Diese Methode wurde nicht implementiert.

- PostScript-Ausgabe 3.4.1.4

  Die Ausgabe von Daten im PostScript-Format wurde nicht implementiert.

- Escape 3.4.1.6

  Es kann nicht jeder Vorgang mit Escape abgebrochen werden.

Kapitel 5

Entwurf

5.1 Einführung

Die dritte der Phasen im Entwicklungsprozeß von Mobidick beschäftigt sich mit dem Entwurf des Systems. Es wird ein Grobentwurf erstellt, in dem die wichtigen Klassen und die Zusammenarbeit der Objekte beschrieben werden. Darauf aufbauend wird ein Feinentwurf angefertigt, der alle Klassen und ihre Schnittstellen im Detail beschreibt.

Für die Klassen wurde ein LaTeX-Makro entworfen, in das ein Kopfteil, Methoden und Attribute eingegeben werden. Bei den Methoden wird zwischen public, private und protected unterschieden. Sie werden in C++-Syntax deklariert und kommentiert. Auch bei den Attributen wird ein Kommentar angegeben.

Passend zu diesem LaTeX-Makro wurde ein perl-Skript geschrieben, das aus den TeX-files ein entsprechendes C++-headerfile generiert. Einige Bemerkungen zu diesem Tool finden sich im Kapitel 5.4.3.

5.1.1 Zweck

Der Zweck dieses Dokumentes ist, den Grobentwurf des Systems Mobidick darzulegen und die Schnittstellen in Form von Methoden und Datenstrukturen festzulegen. Der Kunde und unbeteiligte Dritte sollen durch dieses Dokument einen groben Überblick über das System erhalten.

5.1.2 Überblick

Das Dokument gliedert sich in drei Teilgebiete: den Grobentwurf, die Klassenbeschreibungen und Szenarien.

Im Grobentwurf wird das Gesamtsystem präsentiert und zu jeder wichtigen Klasse eine Beschreibung geliefert, die Kommunikation mit anderen Klassen und eine Motivation für diese Klasse enthält. Die Graphiken orientieren sich am objektorientierten Entwurf nach Booch [1].

Die Klassenbeschreibungen listen Methoden und Attribute aller Klassen des Systems auf. Dieser Teil dient der Abstimmung innerhalb der Projektgruppe und wird gleitend in die Implementierung übergehen. Daher werden die Methoden auch bereits in C++-Syntax deklariert.

Einzelne typische Szenarien werden beschrieben, um das Zusammenspiel der Klassen bei Vorgängen im System zu verdeutlichen. Die Darstellung der Szenarien orientiert sich am objektorientierten Entwurf nach Booch [1].

5.2 Grobentwurf

5.2.1 Systemaufbau

Das System Mobidick gliedert sich in mehrere Verwalterobjekte, eine interaktive Komponente und einen Fürsorger. Die Verwalter stehen untereinander in Verbindung, die interaktive Komponente benutzt den Fürsorger als Schnittstelle zu den Verwaltern. Im weiteren werden die Komponenten vorgestellt und kurz beschrieben. Die Graphik soll den groben Aufbau und die Verbindungen erläutern.

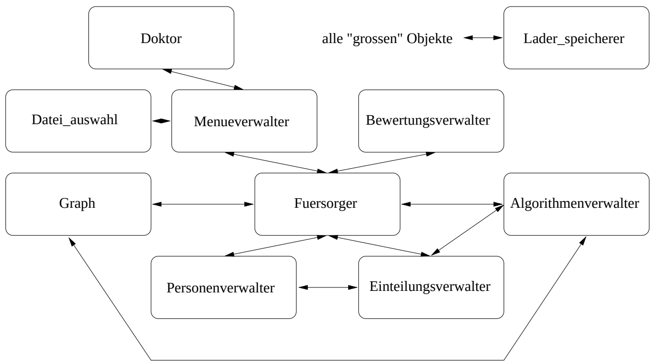

Abb. 5.1: Der Grobaufbau des Systems

5.2.1.1 Menüverwalter und Doktor

Die Kontrolle liegt während des Programmlaufes hauptsächlich beim Menüverwalter. Dies ermöglicht ein leichtes Austauschen der textuellen Oberfläche durch ein graphisches Framework. Der Doktor enthält Warnungen und Fehlermeldungen im Klartext, diese werden von den Menüs auf dem Bildschirm ausgegeben.

Bei einem Fehler wird eine entsprechende Nachricht mit Fehlercode zurückgegeben, die das Menü dann behandelt, indem es den Text aus dem Doktor ausliest, ausgibt und sich dem Fehler entsprechend verhält.

5.2.1.2 Algorithmenverwalter und Bewertungsverwalter

Die Algorithmen zur Wegsuche und zur Berechnung von Einteilungen müssen verwaltet werden und ein Zugriff auf die entsprechenden Daten muß gewährleistet sein. Der Bewertungsverwalter übernimmt die Verwaltung der Bewertungsfunktionen zur Bewertung von Fahrgemeinschaften und Einteilungen.

5.2.1.3 Personenverwalter und Einteilungsverwalter

Diese beiden Verwalter wachen über die Daten des Systems. Abgespeichert werden hauptsächlich Personen- und Einteilungsdaten. Hier werden sehr viele unterschiedliche Objekte verwaltet.

5.2.1.4 Fürsorger, Datei-Auswahl und Lader/Speicherer

Der Fürsorger ist die Schnittstelle zwischen dem Menüverwalter und dem Rest des Systems. Er verteilt die Nachrichten und speichert einige Systeminformationen. Die Datei-Auswahl ist ein Hilfswerkzeug für den Dateidialog. Sie ist ein spezielles dynamisches Menü. Der Lader/Speicherer bedient sich der Voreinstellungen und stellt einen konsistenten Anfangszustand beim Programmstart her.

5.2.2 Menüverwalter und Doktor

Der Menüverwalter kapselt die Benutzungsschnittstelle. Mit Ausnahme der Datei_Auswahl darf kein anderes Objekt auf Bildschirm oder Tastatur zugreifen. Damit wird die Modularität gewahrt, die es ermöglicht, zu einem späteren Zeitpunkt an die Stelle des Menüverwalters eine graphische Benutzungsoberfläche zu setzen.

Das Programm präsentiert sich mit Auswahlmenüs, die verschiedene Funktionen zur Verfügung stellen. Diese sind als Klassen modelliert. Wird eine der Funktionen ausgewählt, so wird dies dem Menübaum mitgeteilt, der dann entscheidet, ob es sich um einen Untermenüaufruf oder eine Aktion handelt. Im letzteren Fall gibt er die Kontrolle an den Menüverwalter zurück, der dann eine Methode beim Fürsorger aufruft. Dieser spricht das System an, und das Ergebnis wird über den Menüverwalter und den Menübaum wieder an das Menü geliefert und von diesem auf dem Bildschirm präsentiert.

Der Doktor enthält alle Systemmeldungen und gibt diese auf Anfrage mit Fehlercode heraus. Damit werden sie an zentraler Stelle verwaltet und können leichter gewartet werden.

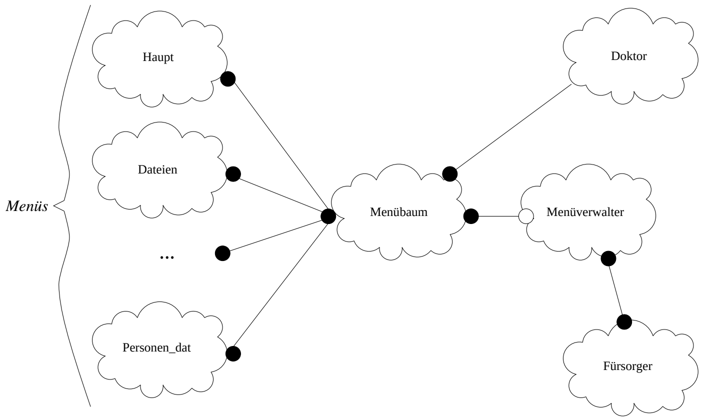

Abb. 5.2: Diagramm der Klasse Menüverwalter

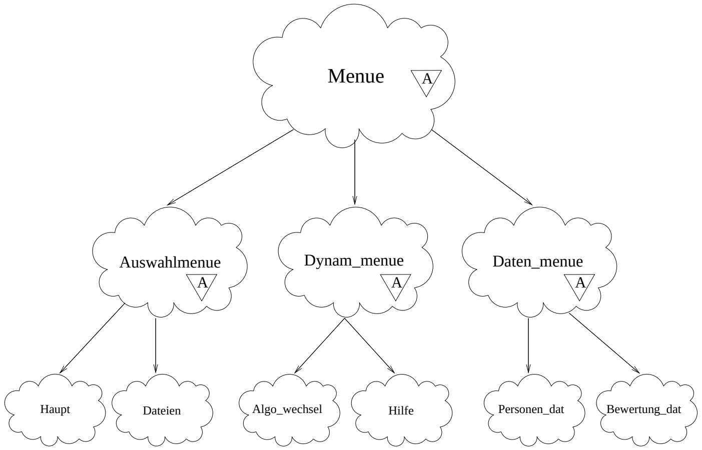

Abb. 5.3: Vererbungshierarchie unter den Menüs

Die drei unterschiedlichen Menüarten werden in einer Vererbungshierarchie modelliert, sie erben von der abstrakten Klasse *Menue*. Ein Auswahlmenü hat fest codierte Menüpunkte, deren Anzahl sich nicht ändert. Bei einem dynamischen

Menü ändert sich das Aussehen zur Laufzeit. Datenmenüs erlauben den Zugriff auf zugehörige Strukturen, so bietet das Menü *Personen_dat* die Möglichkeit, eine Person neu einzugeben, anzuzeigen oder zu ändern. Die konkreten Menüs erben nun von einer dieser drei abstrakten Klassen.

5.2.3 Einteilungsverwalter und Personenverwalter

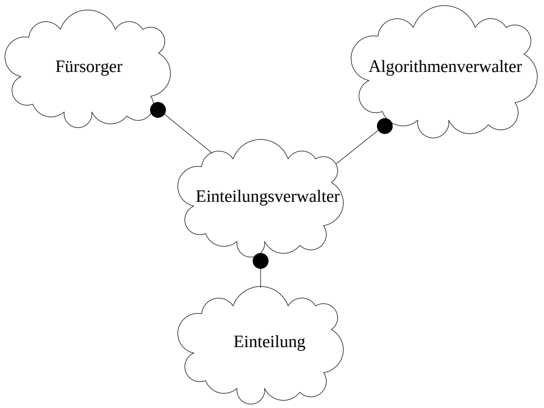

Abb. 5.4: Diagramm der Klasse Einteilungsverwalter

Der Einteilungsverwalter (Abbildung 5.4) organisiert die Datenhaltung aller Einteilungen und Fahrgemeinschaften. Hierzu können Einteilungen angelegt und mit Fahrgemeinschaften gefüllt werden. Soll eine Einteilung bearbeitet werden, muß diese im Einteilungsverwalter zur aktuellen Einteilung gewählt werden, da alle Operationen nur auf der aktuellen Einteilung vorgenommen werden können.

Abb. 5.5: Diagramm der Klasse Einteilung

Eine Einteilung beinhaltet keine, eine oder mehrere Fahrgemeinschaften. Eine Fahrgemeinschaft wiederum enthält Daten der Teilnehmer, z.B. wer Fahrer ist, Fahrtroute der Fahrgemeinschaft und weitere Informationen. Allerdings besitzt jede Fahrgemeinschaft mindestens zwei Teilnehmer, so daß keine Einzelfahrgemeinschaften entstehen können.

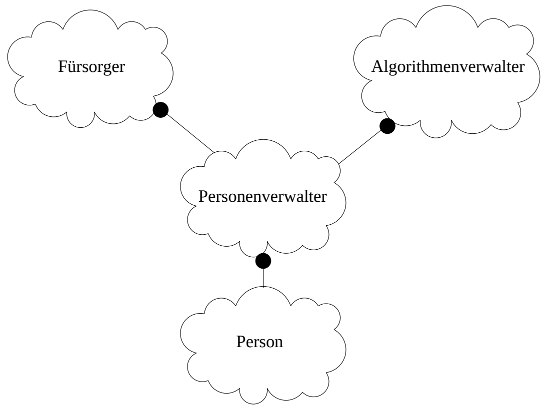

Abb. 5.6: Diagramm der Klasse Personenverwalter

Der Personenverwalter (Abbildung 5.6) ist für die Bereitstellung und Modifikation der Personendaten verantwortlich. Hier werden Nachrichten ausgehend vom Fürsorger und Algorithmenverwalter verarbeitet. Für jede neue Person wird ein neues Objekt der Klasse Person instanziiert und mit den Personendaten gefüllt. Dabei beinhaltet der Personenverwalter selbst nur die Information über das Vorhandensein der Personen, d.h. er besitzt lediglich eine Liste aller Personenobjekte. Spezielle Anfragen an eine Person müssen an den Personenverwalter gerichtet werden, da nicht direkt auf die Datenstruktur einer Person zugegriffen werden kann.

Eine Person (Abbildung 5.7) wird vom Personenverwalter neu angelegt und mit Personendaten gefüllt. Dabei werden nur bestimmte Personendaten abgespeichert (z.B. Name, Vorname, Adresse usw.). Die übrigen Daten werden an Auto, Weginformation und Wunsch weiter geschickt und dort abgespeichert. Für das Anlegen der drei neuen Objekte ist das Objekt Person verantwortlich. Folglich gehört zu jeder Person ein Objekt Auto, Weginformation und Wünsche. Wie bereits schon erwähnt können Anfragen auf Personendaten nur vom Personenverwalter entgegengenommen und an Person weitergeleitet und entsprechend wieder an den Personenverwalter zurückgegeben werden. Zudem kann der Zugriff auf Auto-, Weg- und Wunschdaten nur über das Objekt Person erfolgen.

5.2.4 Algorithmenverwalter

Der Algorithmenverwalter bildet die Schnittstelle zwischen den Einteilungs- und Wegsuchealgorithmen und dem System. Alle Nachrichten zwischen dem System

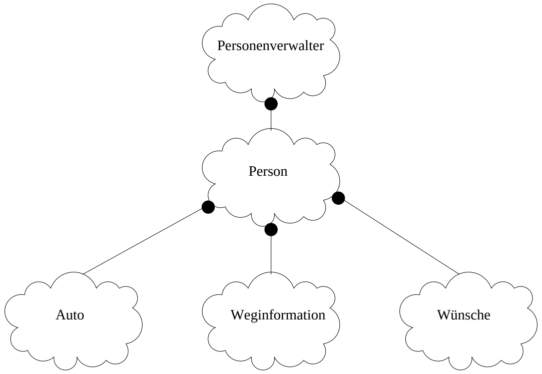

Abb. 5.7: Diagramm der Klasse Person

und der Algorithmen werden an den Algorithmenverwalter geschickt und dieser leitet sie weiter. Der Algorithmenverwalter kennt das kürzeste Wege-Objekt und alle Einteilungs- und Wegsuchealgorithmen. Die Einteilungsalgorithmen erben von der abstrakten Klasse Einteilungsalgorithmus. Die den Algorithmenverwalter betreffenden Klassenbeziehungen sind in Abbildung 5.8 dargestellt.

5.2.5 Algorithmen

Da die Ein- und Ausgabe nicht von den Algorithmen getätigt werden soll, bietet der Menüverwalter die Möglichkeit, integer- double- und Boolean-Werte einzulesen. Dazu stehen Methoden bereit, denen man einen Text und die Variable übergibt, in der der eingegebene Wert gespeichert wird. Der Text wird ausgegeben und die Variable abgefragt.

5.2.5.1 MM_Algorithmus

Der MM-Algorithmus ist eine Heuristik zur Berechnung einer Fahrgemeinschaftseinteilung. Die Buchstaben 'MM' stehen für maximales Matching. Der Algorithmus verwendet bei der Einteilungsberechnung mehrmals ein Verfahren zur Berechnung eines maximalen Matchings. Was ein Matching bzw. ein maximales Matching ist und wie man diese berechnet, kann im Zwischenbericht der Projektgruppe Fahrgemeinschaften [4] nachgelesen werden. Zum Verständnis der nun folgenden Beschreibung des MM-Algorithmus werden die im Zwischenbericht enthaltenen Ausführungen zum maximalen Matching vorausgesetzt.

Mit der einmaligen Berechnung eines maximalen Matchings auf einem Personengraphen, kann eine maximale Menge von 2'er-Fahrgemeinschaften gebildet

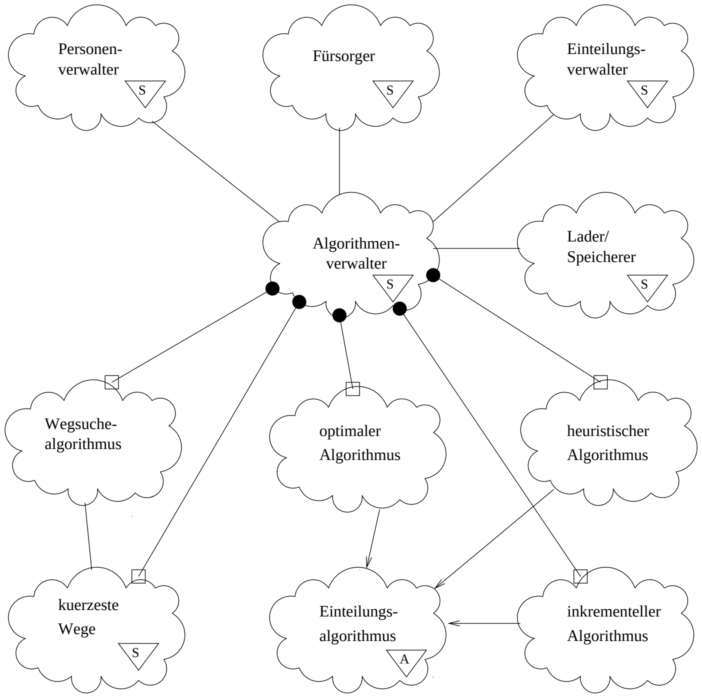

Abb. 5.8: Algorithmenverwalter

werden. Die Heuristik im MM-Algorithmus besteht darin, aus den berechneten 2'er-Fahrgemeinschaften und den übrigen Personen einen neuen Graphen zu erstellen und erneut ein maximales Matching zu berechnen. Dieser Vorgang kann dann so lange fortgesetzt werden, bis Fahrgemeinschaften der gewünschten größe gefunden wurden.

Der MM-Algorithmus kann in folgende Schritte eingeteilt werden:

1. Erstellen eines Graphen. Für jede einzuteilende Person wird in den Graph ein Knoten eingefügt, d.h. jeder Knoten repräsentiert eine Person.

2. Einfügen von Kanten in den Graph. Eine Kante bedeutet, daß die Personen, die durch die Kante verbunden sind, zusammen eine Fahrgemeinschaft bilden könnten. Zur Bestimmung, ob zwischen zwei Knoten eine Kante in den Graphen eingefügt werden soll, wird mit Hilfe der Bewertungsfunktion geprüft, ob sie zusammen fahren könnten.

3. Berechnung des maximalen Matchings auf dem Graph. Zur Berechnung des Matchings wird ein Algorithmus aus der LEDA-Bibliothek verwendet.

4. Bildung von Fahrgemeinschaften. Die Personen, die durch eine Kante des Matchings miteinander verbunden sind, werden zu einer Fahrgemeinschaft zusammengefaßt.

5. Erstellen des neuen Graphen. Die zu den in Fahrgemeinschaften eingeteilten Personen gehörenden Knoten werden aus dem alten Graph gelöscht. Dann wird für jede gebildete Fahrgemeinschaft ein neuer Knoten in den Graph eingefügt. Die neuen Knoten repräsentieren nicht mehr einzelne Personen, sondern je eine Personenmenge.

6. Wiederholung der Schritte 2 bis 5, solange bis das berechnete Matching leer ist. In Schritt 2 wird dabei geprüft, ob die Vereinigung, der durch eine Kante verbundenen Personenmengen, eine Fahrgemeinschaft bilden könnte. Knoten die nur eine Person repräsentieren, werden dabei als Personenmenge mit einem Element betrachtet.

7. Erstellung der neu berechneten Einteilung im Einteilungsverwalter. Die in den Schritten 2 bis 6 gebildeten Personenmengen bilden die Fahrgemeinschaften der zu berechnenden Einteilung. Für jeden Knoten des Graphen, der eine Personenmenge mit mehr als einer Person repräsentiert wird in der im Einteilungsverwalter erstellten Einteilung eine Fahrgemeinschaft gebildet. Die Angaben bezüglich des Fahrers, des Weges und des Zeitplans werden mit Hilfe der Bewertungsfunktion berechnet.

Die Schritte 2 bis 5 werden in einer Schleife zusammengefaßt, die abgebrochen wird, wenn in einem Schleifendurchlauf das leere Matching berechnet wird. Eine spezielle Abbruchbedingung zur Begrenzung der Fahrgemeinschaftsgröße ist nicht nötig, da die Größe in der Bewertungsfunktion berücksichtigt wird. Wenn die Bewertungsfunktion auf eine Personenmenge angewendet wird, deren Personenanzahl das in den Voreinstellungen angegebene Maximum der Fahrgemeinschaftsgröße übersteigt, so ist das Ergebnis der Bewertung null. Dies bedeutet, daß diese Personenmenge keine Fahrgemeinschaft bilden kann und es wird keine

Kante in den Graph eingefügt. Durch diese und die anderen Einschränkungen in der Bewertungsfunktion, wird ein Graph ohne Kanten entstehen, in dem dann keine matchende Kante mehr gefunden werden kann und der MM-Algorithmus terminiert.

5.2.5.2 Inkrementeller Algorithmus

Der inkrementelle Algorithmus arbeitet mit einer Einpersonenumsetzung aus einer Fahrgemeinschaft in eine andere. Es wird in einer Iteration die beste Umsetzung ermittelt und anschließend durchgeführt. Dazu werden auch nicht vermittelte Personen als Fahrgemeinschaften angesehen.

Eine Person wird nicht entnommen, wenn das dazu führt, daß die Fahrgemeinschaft nicht mehr als solche existieren könnte. Das ist zum Beispiel der Fall, wenn der einzige Fahrer entnommen wird, oder die Anzahl der Personen nicht mehr ausreicht, eine Fahrgemeinschaft zu bilden (1 oder 0 Personen).

Die Verbesserung wird lokal für die betroffenen Fahrgemeinschaften ausgerechnet. Es gibt drei Abbruchkriterien:

- Erreichen einer bestimmten Güte
- Anzahl der Iterationen
- Keine Verbesserung erzielt

Der Algorithmus ist in Abbildung 8.4.1 angegeben.

5.2.6 Bewertungsverwalter

Der Bewertungsverwalter bildet die Schnittstelle zwischen den Bewertungsfunktionen und dem System. Auf die Methoden einer Bewertungsfunktion kann nur der Bewertungsverwalter direkt zugreifen. Alle Nachrichten, die eine Bewertungsfunktion betreffen, werden an den Bewertungsverwalter geschickt und dieser leitet sie weiter.

Während eine Bewertungsfunktion eine Fahrgemeinschaft oder eine Einteilung bewertet, erhält sie die benötigten Personen- und Einteilungsdaten von dem Personenverwalter und dem Einteilungsverwalter. Auch die von der Bewertungsfunktion erzeugten Nachrichten werden nicht direkt an den Personen- oder Einteilungsverwalter geschickt, sondern immer an den Bewertungsverwalter, der sie weiterleitet. Die den Bewertungsverwalter betreffenden Klassenbeziehungen sind in der Abbildung 5.9 dargestellt.

5.2.7 Wegsuche

Die Klasse Wegsuche_algorithmus dient als Oberklasse für alle Wegsuchealgorithmen (insgesamt 5 Stück, s. Abb. 5.10), somit kann leicht zwischen den verschiedenen Algorithmen gewechselt werden. In dieser Oberklasse sind die allen

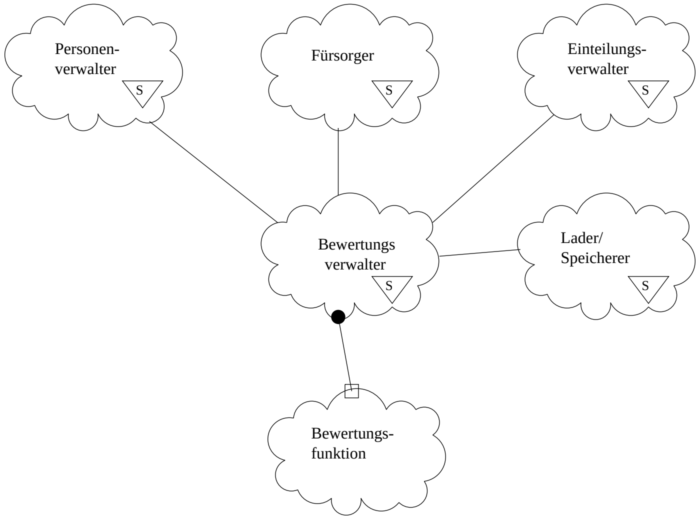

Abb. 5.9: Bewertungsverwalter

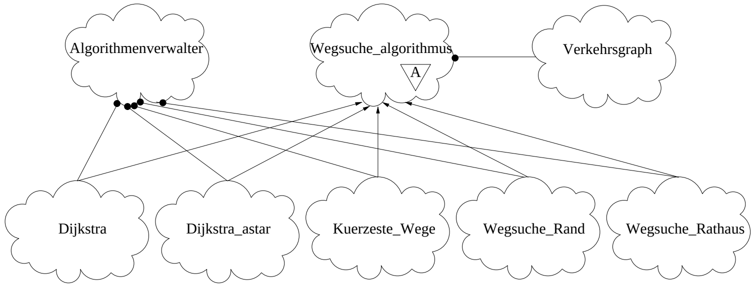

Abb. 5.10: Klassen für die Wegsuche

Algorithmen gemeinsamen Attribute Algorithmusname und -ID enthalten. Für jeden Wegsuchealgorithmus (Dijkstra, Dijkstra_astar, Kuerzeste_Wege, Wegsuche_Rand und Wegsuche_Rathaus) existiert ein Objekt, diese Objekte werden neben den Einteilungsalgorithmen vom Algorithmenverwalter verwaltet. Im Algorithmenverwalter gibt es Methoden zum Wechseln zwischen den verschiedenen Algorithmen und zum Starten der Wegsuche. Bei den Algorithmen Dijkstra und Dijkstra_astar erhält man als Resultat der Wegsuche die Fahrtzeit zwischen Start- und Zielknoten sowie eine Kanten-ID-Liste, die den Weg beschreibt. Bei den restlichen drei Algorithmen wird nur die Fahrtzeit bestimmt.

Dijkstra ist der herkömmliche Dijkstra-Algorithmus mit vorgegebenem Start- und Zielknoten. Bei Dijkstra_astar wird eine A*-Heuristik verwendet, bei der außer der bisherigen Weglänge auch die geschätzte Mindestentfernung zum Zielknoten eingeht. Die Klasse Kuerzeste_Wege kapselt eine bereits berechnete vollständige Entfernungstabelle auf Festplatte, die Wegsuche besteht lediglich im Nachschlagen der zum Knotenpaar zugehörigen Entfernung. Die Algorithmen Wegsuche_Rand und Wegsuche_Rathaus arbeiten auf einem hierarchischen Verkehrsgraphen und sind damit wesentlich schneller als Dijkstra und Dijkstra_astar. Da beide Algorithmen relativ große Tabellen im Hauptspeicher benötigen, werden entweder die Randknoten- oder die Rathaustabellen geladen, niemals aber beide gleichzeitig.

Von der Klasse Verkehrsgraph existiert zur Laufzeit maximal ein Objekt, in dem sämtliche Verkehrsdaten gespeichert sind. Dazu gehört die Knoten- und Kantenmenge des zugrundeliegenden Graphen mit der Kantengewichtsfunktion, ebenso die Knotenkoordinaten. Als Kantengewichtsfunktion wird die geschätzte Fahrzeit verwendet, indem für die verschiedenen Straßenklassen unterschiedliche Geschwindigkeiten festgelegt werden. Dadurch werden schnellere Straßen bei der Wegsuche in einem höheren Maße berücksichtigt. Weiterhin ist im Verkehrsgraphen die Zuordnung zwischen Kantennummern und Straßennamen abgespeichert.

Für die Algorithmen Wegsuche_Rand und Wegsuche_Rathaus wird jeweils ein hierarchischer Verkehrsgraph benötigt, s. Abb. 5.11. Bei der Randknotensu-

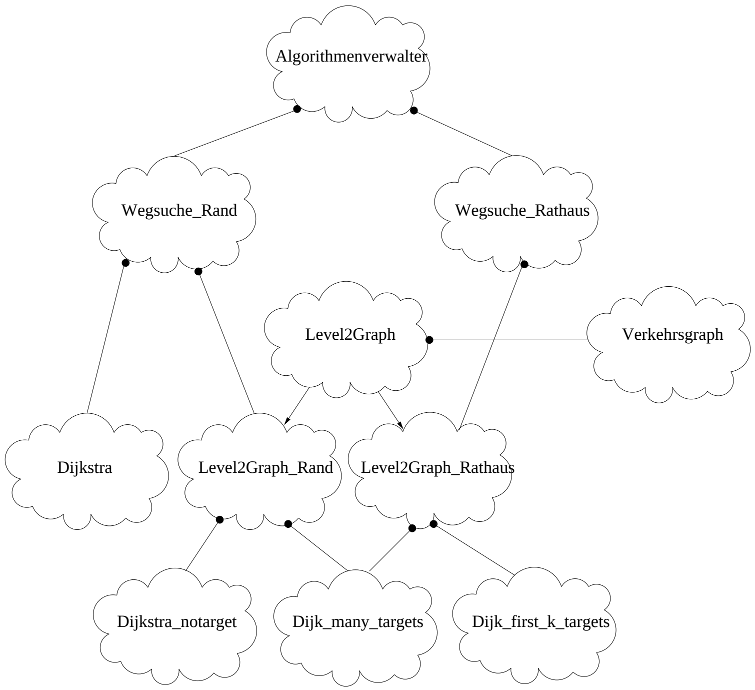

Abb. 5.11: Klassen für die hierarchische Wegsuche

che wird der Verkehrsgraph in Regionen aufgeteilt und die so entstehenden Regionenrandknoten werden zu Knoten des Graphlevels 2, der in der Klasse Verkehrsgraph enthaltene Graph bildet den Level 1. Beim Rathausmodell wählt man wichtige Kreuzungen aus, die zum Level 2 gehören sollen. Die Klasse Level2Graph enthält die den beiden Level-2-Graphen Level2Graph_Rand und Level2Graph_Rathaus gemeinsamen Attribute wie z.B. eine vollständige Entfernungsmatrix. Die Level-2-Graphen enthalten Methoden zum Aufbau des zweiten Levels und der dafür nötigen Tabellen. Für diesen Aufbau werden die Dijkstra-Varianten Dijkstra_notarget, Dijk_many_targets und Dijk_first_k_targets benötigt. Bei der Randknotensuche wird auf den herkömmlichen Dijkstra-Algorithmus zurückgegriffen, falls Start- und Zielknoten in der gleichen Region sind.

5.2.8 Fürsorger und Lader/Speicherer

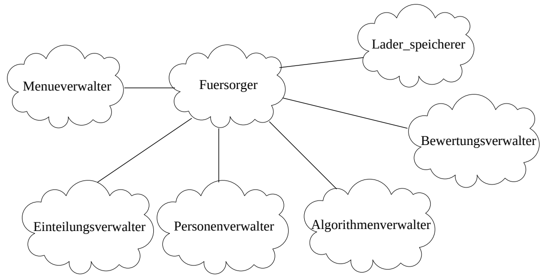

Abb. 5.12: Diagramm der Klasse Fuersorger

Der Fürsorger stellt die Schnittstelle zwischen der Benutzerinteraktion und dem System dar. Hier werden die Nachrichten ausgehend vom Menüverwalter an die Verwalter und den Graphen sowie den Lader/Speicherer verteilt. Der Fürsorger selbst beinhaltet nur Informationen, die das Gesamtsystem betreffen. Durch diese Vorgehensweise kann man die momentan menügesteuerte Interaktion mit dem Benutzer später durch ein Framework ersetzen, ohne am System größere Änderungen vornehmen zu müssen.

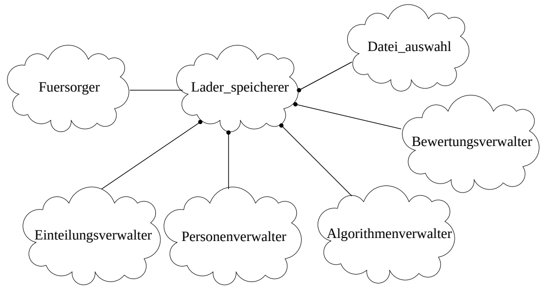

Abb. 5.13: Diagramm der Klasse Lader_speicherer

Der Lader/Speicherer wird vom Fürsorger angestoßen, um Daten zu laden und zu speichern. Er stellt die Schnittstelle zur physikalischen Speicherung der Daten dar. So kann man die Personendaten später zum Beispiel in einer Datenbank halten, ohne an den Zugriffsmethoden im System etwas ändern zu müssen. Der Lader/Speicherer hat die Kontrolle beim Laden und Speichern. Er läßt sich Stück für Stück die zu speichernde Information von den Verwaltern geben und legt sie ab. Beim Laden wird die Information dann auch wieder Stück für Stück weitergereicht. Bei Personendaten wird zum Beispiel jede Person, jede Fahrgemeinschaft und Einteilung einzeln geladen und gespeichert. Die Information zum Fortsetzen einer abgebrochenen Berechnung wird jedoch vom Algorithmenverwalter abgelegt.

Abb. 5.14: Diagramm der Klassen Datei_auswahl und Voreinstellungen

Die Datei-Auswahl übernimmt für den Menüverwalter die Schnittstelle zum Betriebssystem. Hier wird eine Liste der möglichen Dateien angezeigt, die sich im Verzeichnis befinden, das in den Voreinstellungen festgelegt ist. Hier wird auch die Auswahl getroffen und das Überschreiben einer Datei abgefragt.

Die Voreinstellungen enthalten nach dem Systemstart die Pfade und Dateinamen, die in der Datei `voreinstellungen.mbd` in dem Verzeichnis liegen, von dem aus das System gestartet wurde. Wird unter einem neuen Pfad gespeichert, wird dieser übernommen, wird ein neuer Name für eine Datei gewählt, wird dieser übernommen. Diese Änderungen wirken sich allerdings nur zur Laufzeit aus und werden nicht bei Verlassen des Systems abgespeichert. Die Datei `voreinstellungen.mbd` muß von Hand geändert werden.

5.3 Szenarien

Zur Verdeutlichung der Abläufe innerhalb des Systems werden im folgenden einige Szenarien angegeben. Die Darstellung orientiert sich an den Szenarien nach Booch [1]. Zu jedem der Szenarien gibt es einen erläuternden Text. Die Nummern der Szenarien entsprechen den use cases der Spezifikation (Kapitel 3).

5.3.1 Neue Personendatei anlegen

Hier wird aus dem Menü der Punkt `Datei-Personendateien-Neu` gewählt. Die alte Personendatei war noch nicht abgespeichert und muß somit noch abgespeichert werden. Danach kann dann eine neue ausgewählt werden. (Abbildung 5.15)

5.3.2 Neuer Teilnehmer manuell eintragen

Nach use case 3.6.5 wird zuerst eine Person ausgewählt, danach eine Fahrgemeinschaft nach use case 3.7.10. Nun wird der Teilnehmer in die Fahrgemeinschaft eingetragen und diese bewertet. Falls das Ergebnis zufriedenstellend ist, wird der Teilnehmer übernommen. (Abbildung 5.16)

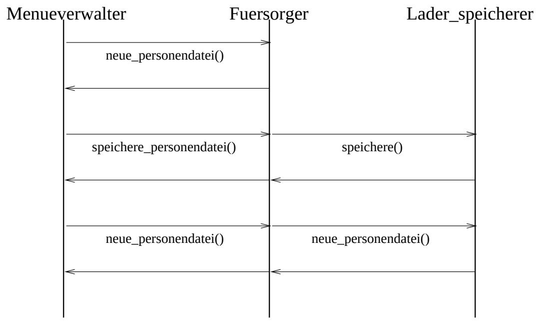

Abb. 5.15: Neue Personendatei anlegen

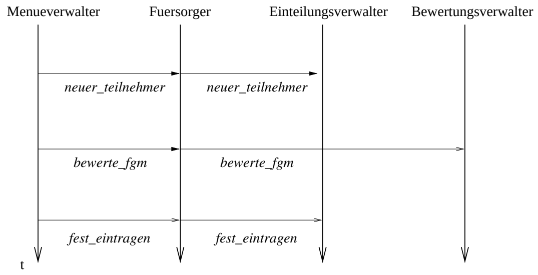

Abb. 5.16: Neuer Teilnehmer manuell eintragen

5.3.3 Einteilung berechnen

Mit dem aktuellen Algorithmus wird die Berechnung einer neuen Einteilung vorgenommen. Der Algorithmenverwalter startet den entsprechenden aktuellen Algorithmus, der an jeder abbrechbaren Stelle nachfragt, ob ein Abbruch vorgenommen werden soll. Hat der Benutzer `u` oder `Esc` gedrückt, wird abgebrochen, sonst fährt der Algorithmus fort. Der Algorithmus nimmt Änderungen an Einteilungen über den Algorithmenverwalter vor. Daten werden auch über den Algorithmenverwalter eingelesen. (Abbildung 5.17)

5.3.4 Neue Bewertungsfunktion anlegen

Bevor eine neue Bewertungsfunktion spezifiziert wird, muss sie angelegt werden. Dazu liest der Menüverwalter alle Bewertungsfunktionen ein und läßt den Benutzer einen neuen Namen angeben. Dem Bewertungsverwalter wird dann

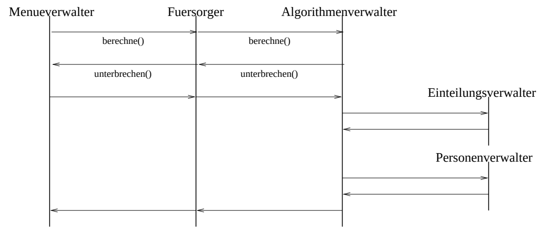

Abb. 5.17: Einteilung berechnen

mitgeteilt, daß er eine neue Bewertungsfunktion mit dem ausgewählten Namen anlegen soll. (Abbildung 5.18)

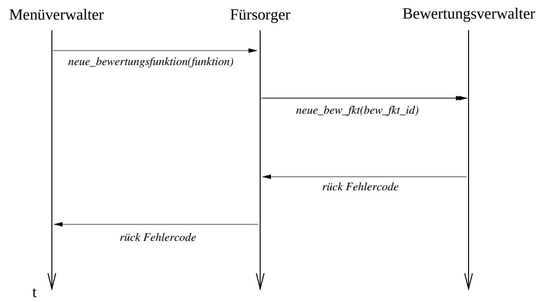

Abb. 5.18: Neue Bewertungsfunktion anlegen

5.3.5 Einzelwegsuche

Zunächst wird überprüft, ob ein Verkehrsgraph geladen ist. Nach Eingabe zweier Straßennamen werden die Start- und Zielkante an den GaraphViewer geschickt, der die Strecke darstellt. Eventuell kann man noch eine Wegliste ausdrucken. (Abbildung 5.19)

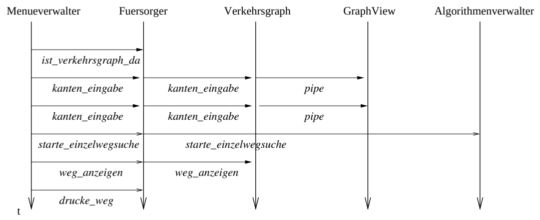

Abb. 5.19: Einzelwegsuche

5.4 Schnittstellen

5.4.1 constants.h

Die Methoden der Klassen benutzen verschiedene globale Datentypen. Diese werden in einer Datei *constants.h* festgelegt, die von allen Modulen des Systems importiert wird. Hier finden sich Aufzählungstypen, Strukturen, die komplexe Datensätze für die Parameterübergabe festlegen (structs) und Konstanten.

Ein wichtiger Aufzählungstyp ist *Fehlercode*. Öffentliche Methoden der Verwalter und der Menüs sowie des Fürsorgers und der Datei-Auswahl geben diesen Wert zurück, um eine einheitliche Fehlerbehandlung zu garantieren. Schlüsselstrukturen sind Personendaten, GenerierDaten, FGMdaten und Bew_fkt_parameter. Der Verkehrsgraph benutzt einige hier festgelegte Pfade.

5.4.2 LaTeX-Makro

Die Eingabe der Klassen sollte möglichst flexibel gehalten werden. Daher wurde in den meisten Fällen auf eine Syntaxprüfung verzichtet und einfach eine Texteingabe erwartet. Als Beispiel für eine entsprechende Datei folgt das *.tex*-file für die Klasse Voreinstellungen.

```
\begin{Klasse}{Voreinstellungen}
  %C: Dateiname:    Voreinstellungen.h
  %C: Autor:        Alexander Porrmann
  %C: erstellt:     ?
  %C: geaendert:    ?
  %C: Zweck:        enthaelt voreingestellte Pfade und Namen und veraendert
  %C:               diese zur Laufzeit in aktuelle Pfade und Namen
  %C: Bemerkungen:  -
  \Includes{}
  %Verzeichnis:  dateien
  \begin{Methodenpublic}
    \Methode{Voreinstellungen();}
    {Im Konstruktor wird die Datei voreinstellungen.mbd eingelesen, falls
      vorhanden; ansonsten werden die Pfadattribute mit dem Aufrufpfad belegt.}
    \Methode{Fehlercode setze\_pfad(const Dateiart art,const string pfad);}
    {Setzt den art entsprechenden Pfad auf pfad.}
```

```
    \Methode{Fehlercode gib\_pfad(const Dateiart art,string\& pfad);}
    {Setzt pfad auf den art entsprechenden Pfad.}
    \Methode{Fehlercode setze\_name(const Dateiart art,const string name);}
    {Setzt den art entsprechenden Namen auf name.}
    \Methode{Fehlercode gib\_name(const Dateiart art,string\& name);}
    {Setzt name auf den art entsprechenden Namen.}
    \Methode{Fehlercode gib\_max\_anzahl\_teilnehmer(int\& anzahl);}
    {Setzt anzahl auf max\_anz\_teilnehmer.}
  \end{Methodenpublic}
  \begin{Attribute}
    \Attribut{string bewertungsname;}{Name der Bewertungsfunktion}
    \Attribut{string bewertungspfad;}{Pfad f"ur Bewertungsfunktionen}
    \Attribut{string personenpfad;}{Pfad f"ur Personen- und Einteilungsdaten}
    \Attribut{string personenname;}{Name der Personendatei}
    \Attribut{string knotenpfad;}{Pfad f"ur Knoten im Verkehrsgraphen}
    \Attribut{string knotenname;}{Name f"ur Knoten im Verkehrsgraphen}
    \Attribut{string kantenpfad;}{Pfad f"ur Kanten im Verkehrsgraphen}
    \Attribut{string kantenname;}{Name f"ur Kanten im Verkehrsgraphen}
    \Attribut{string regionenpfad;}{Pfad f"ur Regionen im Verkehrsgraphen}
    \Attribut{string regionenname;}{Name f"ur Regionen im Verkehrsgraphen}
    \Attribut{int max\_anz\_teilnehmer;}
    {Maximale Anzahl der Teilnehmer einer Fahrgemeinschaft}
  \end{Attribute}
  \begin{Methodenprivate}
    \Methode{Boolean voreinstellungen\_einlesen();}
    {Liest aus der Datei voreinstellungen.mbd im aktuellen Verzeichnis die
      vorhandenen Voreinstellungen in die Attribute ein; gibt FALSE zurueck,
      falls die Datei nicht existiert.}
   \end{Methodenprivate}
\end{Klasse}

%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
```

5.4.3 entwurf2class.pl

Zum Erstellen von C++-headerfiles aus den Dateien, die das LaTeX-Makro benutzen, wurde ein perl-script erstellt. Es arbeitet mit einer oder mehreren dem Format entsprechenden Klassenbeschreibungen in einer Datei. Das Ergebnis wird in einem Verzeichnis abgelegt, das in der Klassenbeschreibung mit angegeben wird. Dabei wird dort nur der relative Pfad ausgehend von einem Entwurfsverzeichnis angegeben. Der voreingestellte Pfad für das Zielverzeichnis steht im Kopf des perl-scripts. Der in der Beschreibung eingetragene header erscheint im Kopf der Zieldatei.

Ist das Zielverzeichnis einer Klassenbeschreibung nicht vorhanden, wird keine Fehlermeldung ausgegeben und kein C++-headerfile erzeugt. Daher sollte die Verzeichnisstruktur vorher überlegt und eingerichtet sein.

Nimmt man Änderungen an den Klassen während der Implementierung vor und führt diese zuerst in den LaTeX-files aus, muß man nur noch das perl-script aufrufen und das *.cc*-file ändern, um den Entwurf und die Implementierung konsisten zu halten.

Das Skript wird mit dem Kommando `entwurf2class.pl` gestartet. Die Option `-d` ermöglicht es, ein anderes Zielverzeichnis einzustellen. Die zu bearbeitenden

Dateien werden als Argumente angegeben.
Bsp: `entwurf2clss.pl -d /home/Mobidick Voreinstellungen.tex`.

5.5 Klassenbeschreibungen

Die Klassen werden mit ihren Methoden und Attributen aufgelistet. Zuerst wird der Name der Klasse genannt. Dann folgen Methoden aufgeschlüsselt nach *public* und *protected*. Das Wort *public* bedeutet, daß diese Methoden von anderen Klassen aufgerufen werden können. Es handelt sich hierbei also um die Schnittstellen nach außen. Die Attribute können also von anderen Objekten aus nur über solche öffentliche Methoden verändert werden.

5.5.1 Menüverwalter und Doktor

Menueverwalter

öffentliche Methoden
`Menueverwalter(Fuersorger* fuersorger);` Konstruktor: Erzeugt den Menueverwalter
`Fehlercode start();` Beginnt das Programm und gibt Kontrolle an den Menuebaum
`Fehlercode systemmeldung(string text);` Gibt eine Systemmeldung im aktuellen Menü auf den Bildschirm aus
`Fehlercode wird_berechnung_unterbrochen(Abbruch& abbruch);` Fuer laufende Berechnung: Wurde U getippt?
`Fehlercode tuwas(Aktionen aktion);` Für den Menuebaum: führt Aktionen aus

Attribute
`Fuersorger* fuersorger;` Der Fuersorger
`Personen_daten* personen_daten;` Das Personen-Datenmenue
`FGM_daten* fgm_daten;` Das Datenmenue fuer Fahrgemeinschaften

Menuebaum

öffentliche Methoden
`Menuebaum();` Konstruktor
`Fehlercode start(Menueverwalter* mv);` Teilt dem Menuebaum mit, wer sein Menueverwalter ist und startet den Menuebaum

Attribute
`Menueverwalter* menueverwalter;` Zeiger auf den Menueverwalter

`Haupt haupt;` Das Hauptmenü
`Dateien dateien;` Das Menü Dateien
`Personen personen;` Das Menü Personen
`Fahrgemeinschaften fahrgemeinschaften;` Das Menü Fahrgemeinschaften
`Vermittlung vermittlung;` Das Menü Vermittlung
`Bewertungfkten bewertungfkten;` Das Menü Bewertungsfunktionen
`Wegsuche wegsuche;` Das Menü Wegsuche
`Person_dat person_dat;` Das Menü Personendateien
`Einteilung_dat einteilung_dat;` Das Menü Einteilungen
`Bewertung_dat bewertung_dat;` Das Menü Dateien-Bewertungsfunktionen
`Neuer_teilnehmer neuer_teilnehmer;` Das Menü Neuer Teilnehmer

Doktor

öffentliche Methoden
`Doktor();` Konstruktor
`Fehlercode gib_Meldung(const Fehlercode fehler, string& meldungstring);` Gibt die Meldung zum Fehlercode zurueck
`Boolean gibt_es_fehler(const Fehlercode fehler);` Ermittelt, ob ein entsprechender Fehler existiert
`Boolean gab_es_fehler(const Fehlercode fehler);` Ermittelt, ob der entsprechende Fehler zur Laufzeit schon auftrat

Attribute
`enum {a_fehler = anzahl_fehler};` Anzahl der Fehlercodes
`char* meldung[a_fehler];` Fehlermeldungen
`Boolean fandstatt[a_fehler];` Flag, ob Fehler schon war

Menue

öffentliche Methoden
`virtual ~Menue(){ };` virtueller Destruktor

Attribute
`char* titel;` Die Ueberschrift des Menues

geschützte Methoden
`Menue(char* titelzeile);` Konstruktor mit Titel
`Menue(){ };` Konstruktor ohne Titel
`virtual Boolean lese_zeichen(char* erlaubte, char& zeichen);` Liest ein Zeichen ein und gibt zurück, ob es in erlaubte enthalten war

Auswahlmenue : public Menue

öffentliche Methoden
`Aktionen warte();` Stellt das Menü dar und gibt Aktion zurück

Attribute
`char* titel;` Die Titelzeile des Menues
`char* erlaubte_zeichen;` Die Zeichen, die man im Menue drücken darf
`int anz_zeilen;` Die wirkliche Anzahl der Zeilen (kleiner M_ZEILEN)
`enum {M_ZEILEN = MAX_ZEILEN };` Mehr Zeilen darf kein Menue haben
`char* zeilen[M_ZEILEN];` Der Text der Menuezeilen
`Aktionen rueck[M_ZEILEN];` Der Rueckgabewert der Menuezeilen
`char drueck[M_ZEILEN];` Die Taste, die man dazu druecken muss

geschützte Methoden
`Auswahlmenue(char* titelzeile);` Konstruktor fuer Auswahlmenues
`Auswahlmenue();` Konstruktor fuer Auswahlmenues
`void stell_dar();`

Stellt das Menue dar
`virtual Aktionen welche_aktion(char eingabe);` Liefert die Aktion zurück, die eingabe entspricht
`virtual Fehlercode belege_zeilen();` Belegt die Menuezeilen mit Text

Haupt : public Auswahlmenue

öffentliche Methoden
`Haupt(char* titelzeile);` Konstruktor für das Hauptmenue
`Haupt();` Konstruktor für das Hauptmenue

geschützte Methoden
`Aktionen welche_aktion(char eingabe);` Liefert die Aktion zurück, die eingabe entspricht
`Fehlercode belege_zeilen();` Belegt die Menuezeilen mit Text

Datenmenue : public Menue

Attribute
`char* titel;` Der Titel des Datensatzes
`int anz_zeilen;` Die wirkliche Anzahl von Datenzeilen (kleiner M_DATEN)
`enum {M_DATEN = MAX_DATEN };` Mehr Daten darf kein Datensatz haben
`char* zeilen[M_DATEN];` Die Beschreibungen der Datenzeilen
`char drueck[M_DATEN];` Das, was man für dieses Datum drücken muss
`Boolean wurde_ausgewaehlt;` Flag, ob bereits ein Datensatz ausgewaehlt wurde

geschützte Methoden
`Datenmenue();` Konstruktor, nicht oeffentlich
`Datenmenue(char* titelzeile);` Konstruktor für Auswahlmenüs
`Fehlercode lese_zahl(int biszu, int& zahl);` Liest eine (mehrstellige) Zahl ein und gibt OK zurueck, wenn sie kleiner als biszu ist. Sonst: ZU_GROSS oder KEINE_ZAHL

5.5.2 Personenverwalter und Einteilungsverwalter

Personenverwalter

öffentliche Methoden
Personenverwalter(); Konstruktor der Klasse Personenverwalter.
~Personenverwalter(); Destruktor der Klasse Personenverwalter.
Fehlercode setze_verkehrsgraph(Verkehrsgraph *graph); Setzt die Referenz auf den Verkehrsgraphen.
Fehlercode loesche_alles(); Loescht alle Personen aus dem Personenverwalter.
Fehlercode neue_person(Personendaten person); Fuegt eine neue Person in den vorhandenen Personenstamm ein und fuellt sie mit den entsprechenden Daten.
Fehlercode aendere_person(const Person_ID per_id,const Personendaten person); Aendert die Daten der Person mit der ID.
Fehlercode loesche_person(const Person_ID per_id); Loescht die Person mit id aus dem Personenstamm.
Fehlercode suche_personen(Per_Filterdaten filter,list<Person_ID> &id); Gibt eine Liste von Personen zurueck, die den Filterkriterien entsprechen. Vorhandene Kriterien siehe constant.h (struct Per_Filterdaten).
Fehlercode existiert_person(const string per_name,const Datum geb_datum,Person_ID &per_id); Gibt die ID der zu ueberpruefenden Person, falls diese existiert, zurueck. Ueberprueft wird der Name und das dazugehoerige Geburtsdatum.
Fehlercode ist_person(const Person_ID per_id); Prueft, od die Person mit der per_id im Personenstamm vorhanden ist.
Fehlercode gib_personen_liste(list<Person_ID> &id); Gibt eine Liste mit allen im Personstamm vorhandenen Personen zurueck.
Fehlercode gib_personendaten(const Person_ID per_id,Personendaten &person); Gibt die Daten der Person mit der ID zurueck.
Fehlercode generiere_personen(GenerierDaten generierungsdaten); Generiert eine Menge von Personen, die den Generierungsdaten entsprechen. Vorhandene Generierungskriterien siehe constants.h (struct GenerierDaten)
Fehlercode gib_name(const Person_ID per_id,string &name); Gibt den Namen der Person mit der ID zurueck.
Fehlercode gib_vorname(const Person_ID per_id,string &name); Gibt den Vornamen der Person mit der ID zurueck.
Fehlercode gib_geschlecht(const Person_ID per_id,Geschlecht &geschlecht); Gibt das Geschlecht der Person mit der ID zurueck.
Fehlercode gib_raucher(const Person_ID per_id,Raucher &raucher); Gibt die Eigenschaft Raucher der Person mit der ID zurueck, d.h. ob sie Raucher ist oder nicht.

Fehlercode gib_musikgeschmack(const Person_ID per_id,list<string> &musikgeschmack); Gibt eine Liste alle Musikrichtungen der Person mit der ID zurueck.
Fehlercode gib_fahrer(const Person_ID per_id,Boolean &fahrer); Gibt die Eigenschaft Fahrer der Person mit der ID zurueck, d.h. ob Fahrer oder nicht.
Fehlercode gib_abneigung_liste(const Person_ID per_id,list <Person_ID> &abneigung); Liefert eine Liste mit Personen zurueck, mit denen die Person nicht zusammenfahren moechte.
Fehlercode loesche_person_aus_abneigung(const Person_ID per_id,const Person_ID per_neigung); Loescht die Person mit der ID aus der Liste der Abneigungen.
Fehlercode fuege_person_abneigung(const Person_ID per_id,const Person_ID per_neigung); Fuegt die Person mit der ID der Liste der Abneigungen hinzu.
Fehlercode gib_zuneigung_liste(const Person_ID per_id,list <Person_ID> &zuneigung); Liefert eine Liste mit Personen zurueck, mit denen die Person zusammenfahren moechte.
Fehlercode loesche_person_aus_zuneigung(const Person_ID per_id,const Person_ID per_neigung); Loescht die Person mit der ID aus der Liste der Zuneigungen.
Fehlercode fuege_person_zuneigung(const Person_ID per_id,const Person_ID per_neigung); Fuegt die Person mit der ID der Liste der Zuneigungen hinzu.
Fehlercode gib_geschlecht_wunsch(const Person_ID per_id,Geschlecht &geschlecht); Gibt das Geschlecht zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_geschlecht_gewicht(const Person_ID per_id,int &gewicht); Gibt die Gewichtung von Geschlecht zurueck.
Fehlercode gib_rauchen_wunsch(const Person_ID per_id,Raucher &raucher); Gibt das Rauchen zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_rauchen_gewicht(const Person_ID per_id,int &gewicht); Gibt die Gewichtung von Rauchen zurueck.
Fehlercode gib_komfortklasse(const Person_ID per_id,Komfortklasse &komfortklasse); Gibt die Eigenschaft Komfortklasse der Person mit der ID zurueck.
Fehlercode gib_anzahl_der_plaetze(const Person_ID per_id,int &anzahl_der_plaetze); Gibt die Anzahl der Plaetze der Person mit der ID zurueck.
Fehlercode gib_startort_kanten_id(const Person_ID per_id,int &startort_kanten_id); Gibt die Kante des Startortes der Person mit der ID zurueck.

`Fehlercode gib_zielort_kanten_id(const Person_ID per_id,int &zielort_kanten_id);` Gibt die Kante des Zielortes der Person mit der ID zurueck.
`Fehlercode gib_ankunftzeit(const Person_ID per_id,Intervall &ankunftzeit);` Gibt das Intervall Ankunftszeit der Person mit der ID zurueck.
`Fehlercode gib_rueckfahrzeit(const Person_ID per_id,Intervall &rueckfahrzeit);` Gibt das Intervall Rueckfahrzeit der Person mit der ID zurueck.
`Fehlercode gib_arbeitsdauer(const Person_ID per_id,Zeit &arbeitsdauer);` Gibt die Arbeitsdauer der Person mit der ID zurueck.

Attribute
`Verkehrsgraph *verkehrsgraph_ref;` Referenz auf Verkehrsgraph
`Person_ID per_id;` Zaehler fuer die id
`Person_ID aktuelle_id;` ID Nummer der aktuellen Person.
`Fehlercode aktueller_fehlercode;` Name des Fehlercode, der zurueckgegeben wird.
`Datum aktuelles_datum;` Enthaelt aktuelles Datum Systemdatum (z.B. 10.03.1997).
`Person * personen_zeiger;` Zeiger auf das Objekt Person.
`Personendaten person_daten;` Datentyp einer Person.
`list<Person_ID> personen_id_liste;` Liste enthaelt die id's der Personen.
`dictionary <Person_ID,Person*> personen_zeiger_dic;` Dictionary enthaelt die Objektzeiger auf die Personen.

private Methoden
`Fehlercode suche_person_dic(const Person_ID per_id,Person* &personen_zeiger);` Gibt zu einer Personen_id den Zeiger auf das Objekt zurueck.
`Boolean pruefe_anzeit_intervall(const Zeitintervall filter_anzeit,const Person_ID per_id);` Prueft, ob anzeit1 und anzeit2 einen gemeinsamen Schnittpunkt haben und liefert dann TRUE zurueck
`Boolean pruefe_rueckzeit_intervall(const Zeitintervall filter_rueckzeit,const Person_ID per_id);` Prueft, ob rueckzeit1 und rueckzeit2 einen gemeinsamen Schnittpunkt haben und liefert dann TRUE zurueck

`Fehlercode gib_strassenname_start(const Person_ID per_id,string &strassenname);` Gibt den Strassenname der Startadresse der Person mit der ID zurueck.
`Fehlercode gib_strassenname_ziel(const Person_ID per_id,string &strassenname);` Gibt den Strassenname der Zieladresse der Person mit der ID zurueck.
`Boolean pruefe_name(const string teil_name,const string voll_name);` Prueft, ob der Teilname mit dem Vollnamen uebereinstimmt.
`Boolean pruefe_start_strasse(const string fix_strassenname,const int radius,const Person_ID per_id);` Prueft, ob die Person im gewuenschten Radius zu einer Strasse ihren Startort hat.
`Boolean pruefe_ziel_strasse(const string fix_strassenname,const int radius,const Person_ID per_id);` Prueft, ob die Person im gewuenschten Radius zu einer Strasse ihren Zielort hat.
`int zufallszahl(const int tief,const int hoch);` Gibt eine Zufallszahl zurueck, die im Bereich der beiden Werte liegt. tief ist der niedrige Wert und hoch der hoehere.

Person

öffentliche Methoden
`Person();` Konstruktor der Klasse Person.
`~Person();` Destruktor der Klasse Person.
`Fehlercode setze_personendaten(const Personendaten personen_daten);` Setzt die Personendaten und die Objekte Auto, Weginformation und Wuensche werden neu angelegt und mit den entsprechenden Daten versorgt.
`void gib_personendaten(Personendaten &personen_daten);` Gibt die Personendaten zurueck.
`void aendere_personendaten(const Personendaten personen_daten);` Aendert die Personendaten und der Daten Auto, Weginformation und Wuensche
`void loesche_personendaten();` Loescht die Personendaten
`Boolean pruefe_personen(const string per_name,const Datum per_geb_datum);` per_name und per_geb_datum wird mit den Daten der Person ueberprueft. TRUE=stimmt FALSE=stimmt nicht
`Boolean pruefe_vollstaendigkeit();` Person wird auf Vollstaendigkeit ueberprueft. TRUE = vollstaendig,FALSE = nicht vollstaendig
`Fehlercode gib_name(string &name);` Gibt den Namen der Person zurueck.

Fehlercode gib_vorname(string &vorname); Gibt den Vornamen der Person zurueck.
Fehlercode gib_geschlecht(Geschlecht &geschlecht); Gibt die Eigenschaft Geschlecht der Person zurueck.
Fehlercode gib_raucher(Raucher &raucher); Gibt die Eigenschaft Raucher der Person mit der ID zurueck, d.h. ob Raucher oder nicht.
Fehlercode gib_fahrer(Boolean &fahrer); Gibt die Eigenschaft Fahrer der Person zurueck.
Fehlercode gib_abneigung_liste(list <Person_ID> &abneigung); Liste mit Personen, mit denen die Person nicht zusammenfahren moechte.
Fehlercode loesche_person_aus_abneigung(const Person_ID per_neigung); Loescht die Person mit der ID aus der Liste der Abneigungen.
Fehlercode fuege_person_abneigung(const Person_ID per_neigung); Fuegt die Person mit der ID der Liste der Abneigungen hinzu.
Fehlercode gib_zuneigung_liste(list <Person_ID> &zuneigung); Liste mit Personen, mit denen die Person zusammenfahren moechte.
Fehlercode loesche_person_aus_zuneigung(const Person_ID per_neigung); Loescht die Person mit der ID aus der Liste der Zuneigungen.
Fehlercode fuege_person_zuneigung(const Person_ID per_neigung); Fuegt die Person mit der ID der Liste der Zuneigungen hinzu.
Fehlercode gib_geschlecht_wunsch(Geschlecht &geschlecht); Gibt das Geschlecht zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_geschlecht_gewicht(int &gewicht); Gibt die Gewichtung von Geschlecht zurueck.
Fehlercode gib_rauchen_wunsch(Raucher &raucher); Gibt das Rauchen zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_rauchen_gewicht(int &gewicht); Gibt die Gewichtung von Rauchen zurueck.
Fehlercode gib_komfortklasse(Komfortklasse &komfortklasse); Gibt die Eigenschaft Komfortklasse der Person zurueck.
Fehlercode gib_anzahl_der_plaetze(int &anzahl_der_plaetze); Gibt die Eigenschaft Anzahl der Plaetze im Auto der Person zurueck.
Fehlercode gib_startort_kanten_id(Kanten_ID &startort_kanten_id); Gibt die ID der Kante des Startortes der Person zurueck.
Fehlercode gib_zielort_kanten_id(Kanten_ID &zielort_kanten_id); Gibt die ID der Kante des Zielortes der Person zurueck.
Fehlercode gib_anzeit(Intervall &anzeit); Gibt das Intervall Ankunftszeit der Person zurueck.
Fehlercode gib_rueckzeit(Intervall &rueckzeit); Gibt das Intervall Rueckfahrzeit der Person zurueck.
Fehlercode gib_arbeitsdauer(Zeit &arbeitsdauer); Gibt die Arbeitsdauer der Person zurueck.
Fehlercode gib_strassenname_start(string &strassenname_start); Gibt den Strassenname der Startadresse der Person zurueck.

Fehlercode gib_strassenname_ziel(string &strassenname_ziel); Gibt den Strassenname der Zieladresse der Person zurueck.

Attribute
Person_ID id_person; Eindeutige Personen_ID, die vom Personenverwalter vergeben wird.
Optional<string> name_person; Nachname einer Person
Optional<string> vorname_person; Vorname der Person
Optional<Geschlecht> geschlecht_person; Geschlecht der Person, wobei entweder maennlich oder weiblich.
Optional<Datum> geburtsdatum_person; Geburtsdatum einer Person: 10.03.1997
Optional<Adresse> wohnort_person; Wohnort der Person: Strasse, Hausnummer, PLZ, Wohnort
Optional<Telefonnummer> telefon_person; Telefonnummer der Person: Vorwahl und Rufnummer.
Optional<string> email_person; Email der Person.
Optional<Raucher> raucher_person; NICHTRAUCHER und RAUCHER moeglich
Optional<Boolean> fahrer_person; true = kann fahren, false = kann nicht fahren
Auto *auto_zeiger; Zeiger auf Datentyp Auto zum Anlegen eines Objekts.
Weginformation *weg_zeiger; Zeiger auf Datentyp Weginformation zum Anlegen eines Objekts.
Wunsch *wunsch_zeiger; Zeiger auf Datentyp Wunsch zum Anlegen eines Objekts.

Auto

öffentliche Methoden
Auto(); Konstruktor der Klasse Auto.
~Auto(); Destruktor der Klasse Auto.
void setze_autodaten(const Autodaten personen_auto); Setzt die Autodaten der Person.
void gib_autodaten(Autodaten &personen_auto); Gibt die Autodaten der Person zurueck.
Boolean ist_vollstaendig_auto(); Autodaten werden auf vollstaendigkeit ueberprueft. TRUE=vollstaendig,FALSE=nicht vollstaendig
Fehlercode gib_komfortklasse(Komfortklasse &komfortklasse); Gibt die Eigenschaft Komfortklasse der Person zurueck.

`Fehlercode gib_anzahl_der_plaetze(int &anzahl_der_plaetze);` Gibt die Eigenschaft Anzahl der Plaetze im Auto der Person zurueck.

Attribute
`Optional<Komfortklasse> komfort_auto;` Komfortklasse des Autos.
`Optional<int> plaetze_auto;` Anzahl der Plaetze im Auto.
`Optional<int> baujahr_auto;` Baujahr des Autos.

Weginformation

öffentliche Methoden
`Weginformation();` Konstruktor der Klasse Weginformation.
`~Weginformation();` Destruktor der Klasse Weginformation.
`void setze_weginformationsdaten(const Wegdaten personen_weg);` Setzt die Weginformationsdaten der Person.
`void gib_weginformationsdaten(Wegdaten &personen_weg);` Gibt die Weginformationsdaten der Person zurueck.
`Boolean ist_vollstaendig_weginformation();` Wegdaten werden auf vollstaendigkeit ueberprueft. TRUE=vollstaendig,FALSE=nicht vollstaendig
`Fehlercode gib_startort_kanten_id(Kanten_ID &startort_kanten_id);` Gibt die ID der Kante des Startortes der Person zurueck.
`Fehlercode gib_zielort_kanten_id(Kanten_ID &zielort_kanten_id);` Gibt die ID der Kante des Zielortes der Person zurueck.
`Fehlercode gib_anzeit(Intervall &anzeit);` Gibt das Intervall Ankunftszeit der Person zurueck.
`Fehlercode gib_rueckzeit(Intervall &rueckzeit);` Gibt das Intervall Rueckfahrzeit der Person zurueck.
`Fehlercode gib_arbeitsdauer(Zeit &arbeitsdauer);` Gibt die Arbeitsdauer der Person zurueck.
`Fehlercode gib_strassenname_start(string &strassenname_start);` Gibt den Strassenname der Startadresse der Person zurueck.
`Fehlercode gib_strassenname_ziel(string &strassenname_ziel);` Gibt den Strassenname der Zieladresse der Person zurueck.

Attribute
`Optional<Adresse> startort_weg;` Adresse des Startortes: Strasse, Hausnummer, PLZ, Wohnort.
`Optional<Kanten_ID> startkoordinaten_weg;` Kanden-ID des Startortes.

`Optional<Adresse> zielort_weg;` Adresse des Zielortes: Strasse, Hausnummer, PLZ, Wohnort.
`Optional<Kanten_ID> zielkoordinaten_weg;` Kanten-ID des Zielortes.
`Optional<Zeitintervall> anzeit_weg;` Intervall der Anfangsfahrzeit: von und bis.
`Optional<Zeitintervall> rueckzeit_weg;` Intervall der Rueckfahrzeit: von und bis.
`Optional<Uhrzeit> arbeitsdauer_weg;` Laenge der Arbeitszeit.
`Intervall anzeit_objekt;` Intervall der Anfangsfahrzeit: von und bis.
`Intervall rueckzeit_objekt;` Intervall der Rueckfahrzeit: von und bis.
`Zeit arbeitsdauer_objekt;` Laenge der Arbeitszeit (wird nur zur Zeitweise verwendet).

Wunsch

öffentliche Methoden
`Wunsch();` Konstruktor der Klasse Wunsch.
`~Wunsch();` Destruktor der Klasse Wunsch.
`void setze_wunschdaten(const Wunschdaten personen_wunsch);` Setzt die Wunschdaten der Person.
`void gib_wunschdaten(Wunschdaten &personen_wunsch);` Gibt die Wunschdaten der Person zurueck.
`Fehlercode gib_abneigung_liste(list <Person_ID> &abneigung);` Liste mit Personen, mit denen die Person nicht zusammenfahren moechte.
`Fehlercode loesche_person_aus_abneigung(const Person_ID per_neigung);` Loescht die Person mit der ID aus der Liste der Abneigungen.
`Fehlercode fuege_person_abneigung(const Person_ID per_neigung);` Fuegt die Person mit der ID der Liste der Abneigungen hinzu.
`Fehlercode gib_zuneigung_liste(list <Person_ID> &zuneigung);` Liste mit Personen, mit denen die Person zusammenfahren moechte.
`Fehlercode loesche_person_aus_zuneigung(const Person_ID per_neigung);` Loescht die Person mit der ID aus der Liste der Zuneigungen.
`Fehlercode fuege_person_zuneigung(const Person_ID per_neigung);` Fuegt die Person mit der ID der Liste der Zuneigungen hinzu.
`Fehlercode gib_geschlecht_wunsch(Geschlecht &geschlecht);` Gibt das Geschlecht zurueck, fuer das eine Gewichtung angegeben wurde.
`Fehlercode gib_geschlecht_gewicht(int &gewicht);` Gibt die Gewichtung von Geschlecht zurueck.
`Fehlercode gib_rauchen_wunsch(Raucher &raucher);` Gibt das Rauchen zurueck, fuer das eine Gewichtung angegeben wurde.

`Fehlercode gib_rauchen_gewicht(int &gewicht);` Gibt die Gewichtung von Rauchen zurueck.

Attribute
`Optional<Personen_liste> abneigung_personen;` Liste mit Personen, mit denen die Person nicht zusammenfahren moechte.
`Optional<Personen_liste> zuneigung_personen;` Liste mit Personen, mit denen die Person zusammenfahren moechte.
`Optional<Geschlecht> geschlecht_wunsch;` Es gibt an, ob sich die Gewichtung auf maennlich oder weiblich bezieht.
`Optional<int> geschlecht_gewichtung;` Wert der Gewichtung fuer das Geschlecht. 0 steht fuer voellig unwichtig und 10 fuer sehr wichtig.
`Optional<Raucher> raucher_wunsch;` Es gibt an, ob sich die Gewichtung auf raucher oder nichtraucher bezieht.
`Optional<int> raucher_gewichtung;` Wert der Gewichtung fuer das Rauchen. 0 steht fuer voellig unwichtig und 10 fuer sehr wichtig.

Einteilungsverwalter

öffentliche Methoden
`Einteilungsverwalter();` Konstruktor der Klasse Einteilungsverwalter.
`~Einteilungsverwalter();` Dekonstruktor der Klasse Einteilungsverwalter.
`Fehlercode setze_personenverwalter(Personenverwalter *person);` Setzt die Referenz auf den Personenverwalter.
`Fehlercode setze_verkehrsgraph(Verkehrsgraph *graph);` Setzt die Referenz auf den Verkehrsgraphen.
`Fehlercode loesche_alles();` Loescht alle Einteilungen aus dem Einteilungsverwalter.
`Fehlercode neue_einteilung(const string name,Einteilungs_ID &ein_id);` Fuegt eine neue Einteilung mit name in das System ein und gibt die zugeordnete ein_id zurueck.
`Fehlercode dupliziere_einteilung(const string name);` Erzeugt eine Kopie der aktuellen Einteilung mit dem Namen name.
`Fehlercode loesche_einteilung(const Einteilungs_ID ein_id);` Loescht die Einteilung mit der ID.
`Fehlercode setze_aktuelle_einteilung(const Einteilungs_ID ein_id);` Setzt die Einteilung mit der ID auf die aktuelle Einteilung.
`Fehlercode gib_aktuelle_einteilung(Einteilungs_ID &ein_id);` Gibt die ID der aktuellen Einteilung zurueck.
`Fehlercode gib_id_einteilung(list <Einteilungs_ID> &ein_id);` Gibt eine Liste mit allen ein_id zurueck.

`Fehlercode pruefe_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze);` Prueft die Anzahl der freien Plaetze der Fahrgemeinschaft mit fgm_id und gib die Anzahl zurueck.
`Fehlercode gib_noch_nicht_vermittelte_personen(list<Person_ID> &per_id);` Gibt eine Liste mit allen per_id zurueck, die noch nicht vermittelt wurden. Das sind Personen, die noch in keiner Fahrgemeinschaft mitfahren.
`Fehlercode gib_startort_fgm(const FGM_ID fgm_id,Kanten_ID &kanten_id);` Gibt den Startort (kanten_id) der Fahrgemeinschaft fgm_id zurück.
`Fehlercode gib_zielort_fgm(const FGM_ID fgm_id,Kanten_ID &kanten_id);` Gibt den Zielort (kanten_id) der Fahrgemeinschaft fgm_id zurück.
`Fehlercode aktuelle_einteilung_mit(list<FGM_ID> &fgm_id);` Fuer jede Personen, die noch nicht vermittelt wurden, wird eine Einzelfahrgemeinschaft angelegt. Die Liste aller nun vorhandenen Fahrgemeinschaften wird zurückgeliefert.
`Fehlercode aktuelle_einteilung_ohne();` Jede Fahrgemeinschaft wird geloescht, die keinen oder einen Teilnehmer besitzt.
`Fehlercode setze_name_einteilung(const string neuer_name);` Setzt den Namen der aktuellen Einteilung neu.
`Fehlercode gib_name_einteilung(string &ein_name);` Gibt den Namen der aktuellen Einteilung zurueck.
`Fehlercode gib_id_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit allen fgm_id der aktuellen Einteilung zurueck.
`Fehlercode loesche_fgm(const FGM_ID fgm_id);` Loescht die Fahrgemeinschaft mit id der aktuellen Einteilung.
`Fehlercode setze_bew_fkt_id(Bewertungs_ID bew_id);` Setzt die ID der Bewertungsfunktion mit der diese Einteilung berechnet wurde.
`Fehlercode gib_bew_fkt_id(Bewertungs_ID &bew_id);` Gibt die ID der Bewertungsfunktion, mit der diese Einteilung berechnet wurde, zurueck.
`Fehlercode gib_vermittelte_personen(list<Person_ID> &per_id);` Gibt eine Liste mit allen per_id zurueck, die vermittelt wurden.
`Fehlercode suche_fgm(const FGM_filter filter,list<FGM_ID> &fgm_id);` Gibt eine Liste von Fahrgemeinschaften zurueck, die die Filterkriterien erfuellen. Filterkriterien siehe in constant.h unter FGM_filter.
`Fehlercode neue_fgm(FGM_ID &fgm_id);` Fuegt eine neue Fahrgemeinschaft in die aktuelle Einteilung ein und gibt die zugeordnete fgm_id zurueck.
`Fehlercode setze_fgmdaten(const FGM_ID fgm_id,const FGMdaten daten_fgm);` Setzt die Fahrgemeinschaftsdaten. Siehe in constant.h unter FGM_daten

`Fehlercode gib_fgmdaten(const FGM_ID fgm_id,FGMdaten &daten_fgm);` Gibt die Fahrgemeinschaftsdaten zurueck.
`Fehlercode neuer_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id);` Fuegt Person mit id in die Fahrgemeinschaft mit fgm_id ein. Hierzu muss allerdings immer zuerst ein Fahrer gesetzt sein.
`Fehlercode neuer_fahrer(const FGM_ID fgm_id,const Person_ID per_id);` Fuegt eine Person und zwar den Fahrer in die Fahrgemeinschaft mit fgm_id ein. Ist bereits ein Fahrer gesetzt, wird eine Fehlermeldung zurückgegeben und der Vorgang wird abgebrochen.
`Fehlercode loesche_teilnehmer_aus_fgm(const FGM_ID fgm_id,const Person_ID per_id);` Loescht die Person mit per_id aus der Fahrgemeinschaft mit fgm_id.
`Fehlercode gib_teilnehmer_fgm(const FGM_ID fgm_id,list<Person_ID> &per_id);` Gibt eine Liste mit per_id zurueck, die in der Fahrgemeinschaft mit fgm_id mitfahren.
`Fehlercode gib_entstehungsdatum_fgm(FGM_ID fgm_id,FGM_Datum &entstehungsdatum);` Gibt das Entstehungsdatum der FGM mit fgm_id zurueck.
`Fehlercode gib_letzte_aenderung_fgm(FGM_ID fgm_id,FGM_Datum &letzte_aenderung);` Gibt das Datum der letzten Aenderung der FGM mit fgm_id zurueck.
`Fehlercode markiere_fgm(const FGM_ID fgm_id);` Markiert die Fahrgemeinschaft mit fgm_id.
`Fehlercode unmarkiere_fgm(const FGM_ID fgm_id);` Macht die Markierung der Fahrgemeinschaft mit fgm_id rueckgaengig.
`Fehlercode ist_fgm_markiert(FGM_ID fgm_id,Boolean &markierung);` Prueft ob die FGM mit fgm_id markiert ist.
`Fehlercode gib_unmarkierte_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit fgm_id zurueck, die unmarkiert sind.
`Fehlercode gib_markierte_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit fgm_id zurueck, die markiert sind.
`Fehlercode setze_status_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id,const Status status);` Setzt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id, wobei Status den Wert reserviert und fest annehmen kann.
`Fehlercode gib_status_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id,Status &status);` Gibt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id, wobei Status den Wert reserviert und fest annehmen kann.
`Fehlercode fest_eintragen(const FGM_ID fgm_id,const Person_ID per_id);` Traegt die Person mit per_id in die Fahrgemeinschaft mit fgm_id fest ein.
`Fehlercode setze_fahrer_fgm(const FGM_ID fgm_id,const Person_ID per_id);`

Setzt den Fahrer der Fahrgemeinschaft mit fgm_id.
`Fehlercode gib_fahrer_fgm(const FGM_ID fgm_id,Person_ID &per_id);` Gibt den Fahrer der Fahrgemeinschaft mit fgm_id zurueck.
`Fehlercode setze_freie_plaetze_fgm(const FGM_ID fgm_id,const int plaetze);` Setzt die Anzahl der freien Plaetze der Fahrgemeinschaft auf plaetze zurueck.
`Fehlercode gib_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze);` Gibt die Anzahl der freien Plaetze der Fahrgemeinschaft zurueck.
`Fehlercode setze_fahrtroute_fgm(const FGM_ID fgm_id,const list<Kanten_ID> kanten_id);` Setzt die Fahrtroute der Fahrgemeinschaft mit fgm_id. Die Fahrroute besteht aus einer Liste mit kanten_id mit allen Start- und Zielorten der beteiligten Personen, in der Reihenfolge, wie gefahren wird.
`Fehlercode gib_fahrtroute_fgm(const FGM_ID fgm_id,list<Kanten_ID> &kanten_id);` Gibt die Fahrtroute der Fahrgemeinschaft mit fgm_id.
`Fehlercode setze_zeitplan_fgm(const FGM_ID fgm_id,const list<Zeit*> zeiten);` Setzt den Zeitplan der Fahrgemeinschaft mit fgm_id. Der Zeitplan enthaelt die Abfahrts- und Ankunfszeiten an allen Start-und Zielorten in der Reihenfolge, wie sie in der Fahrtroute angegeben sind.
`Fehlercode gib_zeitplan_fgm(const FGM_ID fgm_id,list<Zeit*> &zeiten);` Gibt den Zeitplan der Fahrgemeinschaft mit fgm_id.

Attribute
`Personenverwalter *personen_ref;` Referenz auf Personenverwalter
`Verkehrsgraph *verkehrsgraph_ref;` Referenz auf Verkehrsgraph
`Fehlercode aktueller_fehlercode;` Name des Fehlercode, der zurueckgegeben wird.
`Einteilungs_ID einteilungs_id;` Zaehler fuer Einteilungen id, beginnend bei eins.
`Einteilungs_ID aktuelle_ein_id;` Enthaelt die id der aktuellen Einteilung.
`Einteilung *einteilungs_zeiger;` Zeiger auf das Objekt Einteilung.
`list<Einteilungs_ID> einteilungs_id_liste;` Liste enthaelt die id's der Einteilungen.
`dictionary <Einteilungs_ID,Einteilung*> einteilungs_zeiger_dic;` Dictionary enthaelt die Objektzeiger auf die Einteilungen.

private Methoden
`Fehlercode suche_einteilung_dic(const Einteilungs_ID ein_id,Einteilung* &einteilungs_zeiger);` Gibt zu einer Einteilungs_id den Zeiger auf das Objekt zurueck
`Fehlercode differenz_liste(list<Person_ID> liste_eins,const list<Person_ID> liste_zwei, list<Person_ID> &ergebnis);` Berechnet die Differenz zweier Personen_id Listen.

Einteilung

öffentliche Methoden
`Einteilung();` Konstruktor der Klasse Einteilung.
`~Einteilung();` Destruktor der Klasse Einteilung.
`Fehlercode setze_personenverwalter(Personenverwalter *person);` Setzt die Referenz auf den Personenverwalter.
`Fehlercode setze_verkehrsgraph(Verkehrsgraph *graph);` Setzt die Referenz auf den Verkehrsgraphen.
`void neue_einteilung(const string ein_name,const Einteilungs_ID ein_id);` Legt eine neue Einteilung mit dem Namen und der ID an.
`void setze_name_einteilung(const string neuer_name);` Setzt den Namen der aktuellen Einteilung neu.
`void gib_name_einteilung(string &ein_name);` Gibt den Namen der aktuellen Einteilung.
`Fehlercode setze_bew_fkt_id(Bewertungs_ID bew_id);` Setzt die ID der Bewertungsfunktion, mit der diese Einteilung berechnet wurde.
`Fehlercode gib_bew_fkt_id(Bewertungs_ID &bew_id);` Gibt die ID der Bewertungsfunktion, mit der diese Einteilung berechnet wurde, zurueck.
`Fehlercode gib_id_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit allen fgm_id der aktuellen Einteilung zurueck.
`Fehlercode loesche_fgm(const FGM_ID fgm_id);` Loescht die Fahrgemeinschaft mit id der aktuellen Einteilung.
`Fehlercode gib_vermittelte_personen(list<Person_ID> &per_id);` Gibt eine Liste mit allen per_id zurueck, die vermittelt wurden. Das sind Personen, die in einer Fahrgemeinschaft mitfahren.
`Fehlercode suche_fgm(FGM_filter filter,list<FGM_ID> &fgm_id);` Gibt eine Liste von Fahrgemeinschaften zurueck, die die Filterkriterien erfuellen. Filterkriterien siehe in constant.h unter FGM_filter
`void neue_fgm(FGM_ID &fgm_id);` Fuegt eine neue Fahrgemeinschaft in die aktuelle Einteilung ein und gibt die zugeornete fgm_id zurueck.
`Fehlercode setze_fgmdaten(const FGM_ID fgm_id,const FGMdaten daten_fgm);` Setzt die Fahrgemeinschaftsdaten.

`Fehlercode gib_fgmdaten(const FGM_ID fgm_id,FGMdaten &daten_fgm);` Gibt die Fahrgemeinschaftsdaten zurueck.
`void kopie_fgm(const FGM_ID kopie_fgm_id);` Kopiert eine Fahrgemeinschaft mit der fgm_id.
`Fehlercode neuer_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id);` Fuegt Person mit id in die Fahrgemeinschaft mit fgm_id ein. Hierzu muss allerdings immer zuerst ein Fahrer gesetzt sein.
`Fehlercode neuer_fahrer(const FGM_ID fgm_id,const Person_ID per_id);` Fuegt eine Person und zwar den Fahrer in die Fahrgemeinschaft mit fgm_id ein. Ist bereits ein Fahrer gesetzt, wird eine Fehlermeldung zurückgegeben und der Vorgang wird abgebrochen.
`Fehlercode loesche_teilnehmer_aus_fgm(const FGM_ID fgm_id,const Person_ID per_id);` Loescht die Person mit per_id aus der Fahrgemeinschaft mit fgm_id.
`Fehlercode gib_teilnehmer_fgm(const FGM_ID fgm_id,list<Person_ID> &per_id);` Gibt eine Liste mit per_id zurueck, die in der Fahrgemeinschaft mit fgm_id mitfahren.
`Fehlercode gib_entstehungsdatum_fgm(FGM_ID fgm_id,FGM_Datum &entstehungsdatum);` Gibt das Entstehungsdatum der FGM mit fgm_id zurueck.
`Fehlercode gib_letzte_aenderung_fgm(FGM_ID fgm_id,FGM_Datum &letzte_aenderung);` Gibt das Datum der letzten Aenderung der FGM mit fgm_id zurueck.
`Fehlercode markiere_fgm(const FGM_ID fgm_id);` Markiert die Fahrgemeinschaft mit fgm_id.
`Fehlercode unmarkiere_fgm(const FGM_ID fgm_id);` Macht die Markierung der Fahrgemeinschaft mit fgm_id rueckgaengig .
`Fehlercode ist_fgm_markiert(FGM_ID fgm_id,Boolean &markierung);` Prüft ob eine FGM mit fgm_id markiert ist.
`Fehlercode gib_markierte_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit fgm_id zurueck, die markiert sind.
`Fehlercode gib_unmarkierte_fgm(list<FGM_ID> &fgm_id);` Gibt eine Liste mit fgm_id zurueck, die unmarkiert sind.
`Fehlercode setze_status_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id,const Status status);` Setzt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id.
`Fehlercode gib_status_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id,Status &status);` Gibt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id.
`Fehlercode fest_eintragen(const FGM_ID fgm_id,const Person_ID per_id);` Traegt die Person mit per_id in die Fahrgemeinschaft mit fgm_id fest ein.
`Fehlercode setze_fahrer_fgm(const FGM_ID fgm_id,const Person_ID per_id);`

Setzt den Fahrer der Fahrgemeinschaft mit fgm_id.
Fehlercode gib_fahrer_fgm(const FGM_ID fgm_id,Person_ID &per_id); Gibt den Fahrer der Fahrgemeinschaft mit fgm_id zurueck.
Fehlercode setze_freie_plaetze_fgm(const FGM_ID fgm_id,const int &plaetze); Setzt die Anzahl der freien Plaetze der Fahrgemeinschaft auf plätze zurueck.
Fehlercode gib_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze); Gibt die Anzahl der freien Plaetze der Fahrgemeinschaft zurueck.
Fehlercode setze_fahrtroute_fgm(const FGM_ID fgm_id,const list<Kanten_ID> kanten_id); Setzt die Fahrtroute der Fahrgemeinschaft mit fgm_id. Die Fahrroute besteht aus einer Liste mit kanten_id mit allen Start- und Zielorten der beteiligten Personen, in der Reihenfolge, wie gefahren wird.
Fehlercode gib_fahrtroute_fgm(const FGM_ID fgm_id,list<Kanten_ID> &kanten_id); Gibt die Fahrtroute der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_zeitplan_fgm(const FGM_ID fgm_id,const list<Zeit*> zeiten); Setzt den Zeitplan der Fahrgemeinschaft mit fgm_id. Der Zeitplan enthaelt die Abfahrts- und Ankunfszeiten an allen Start-und Zielorten in der Reihenfolge, wie sie in der Fahrtroute angegeben sind.
Fehlercode gib_zeitplan_fgm(const FGM_ID fgm_id,list<Zeit*> &zeiten); Gibt den Zeitplan der Fahrgemeinschaft mit fgm_id.

Attribute
Personenverwalter *personen_ref; Referenz auf Personenverwalter
Verkehrsgraph *verkehrsgraph_ref; Referenz auf Verkehrsgraph
Fehlercode aktueller_fehlercode; Name des Fehlercode, der zurueckgegeben wird.
Einteilungs_ID id_einteilung; Nummer der Einteilung.
string name_einteilung; Name der Einteilung.
Optional<Bewertungs_ID> bew_id_einteilung; Nummer der Bewertungsfunktion.
FGM_ID fahrgemeinschaft_id; Zaehler fuer Fahrgemeinschaften id, beginnend bei eins
FGM_ID aktuelle_fgm_id; Enthaelt die id der aktuellen Fahrgemeinschaft.
Fahrgemeinschaft *fgm_zeiger; Zeiger auf das Objekt Fahrgemeinschaft.
list<FGM_ID> fgm_id_liste;

Liste enthaelt die ID's der Fahrgemeinschaften der aktuellen Einteilungen.
dictionary <FGM_ID,Fahrgemeinschaft*> fgm_zeiger_dic; Dictionary enthaelt die Objektzeiger auf die Einteilungen.

private Methoden
Fehlercode suche_fgm_dic(const FGM_ID fgm_id,Fahrgemeinschaft* &fgm_zeiger); Gibt zu einer Fahrgemeinschafts_id den Zeiger auf das Objekt zurueck.
Boolean pruefe_startzeit(const Zeitintervall filter_zeit,const FGM_ID fgm_id); Prüft, ob startzeit der Fahrgemeinschaft und filter_zeit einen gemeinsamen Schnittpunkt haben und liefert dann TRUE zurueck
Boolean pruefe_zielzeit(const Zeitintervall filter_zeit,const FGM_ID fgm_id); Prüft, ob zielzeit der Fahrgemeinschaft und filter_zeit einen gemeinsamen Schnittpunkt haben und liefert dann TRUE zurueck
Boolean pruefe_start_strasse(const Kanten_ID strassennummer,const int radius,const FGM_ID fgm_id); Prüft, ob die FGM im gewuenschten Radius zu einem Fixort ihre Startstrasse hat.
Boolean pruefe_ziel_strasse(const Kanten_ID strassennummer,const int radius,const FGM_ID fgm_id); Prüft, ob die FGM im gewuenschten Radius zu einem Fixort ihre Zielstrasse hat.

Fahrgemeinschaft

öffentliche Methoden
Fahrgemeinschaft(); Konstruktor der Klasse Fahrgemeinschaft.
~Fahrgemeinschaft(); Dekonstruktor der Klasse Fahrgemeinschaft.
Fehlercode setze_personenverwalter(Personenverwalter *person); Setzt die Referenz auf den Personenverwalter.
void neue_fgm(const FGM_ID fgm_id); Setzt eine neue Fahrgemeinschaft mit der ID.
Fehlercode neuer_teilnehmer(const Person_ID per_id); Fuegt Person mit id in die Fahrgemeinschaft ein.Hierzu muss allerdings immer zuerst ein Fahrer gesetzt sein.
Fehlercode neuer_fahrer(const Person_ID per_id); Fuegt eine Person und zwar den Fahrer in die Fahrgemeinschaft mit fgm_id ein. Ist bereits ein Fahrer gesetzt, wird eine Fehlermeldung zurückgegeben und der Vorgang wird abgebrochen.
Fehlercode loesche_teilnehmer_aus_fgm(const Person_ID per_id); Loescht die Person mit per_id aus der Fahrgemeinschaft.
Fehlercode gib_teilnehmer_fgm(list<Person_ID> &per_id); Gibt eine Liste mit per_id zurueck, die in der Fahrgemeinschaft mitfahren.

`Fehlercode gib_entstehungsdatum_fgm(FGM_Datum &entstehungsdatum);` Gibt das Entstehungsdatum der FGM zurueck.
`Fehlercode setze_aenderung_fgm();` Setzt das Datum der Aenderung der FGM.
`Fehlercode gib_letzte_aenderung_fgm(FGM_Datum &letzte_aenderung);` Gibt das Datum der letzten Aenderung der FGM zurueck.
`Fehlercode setze_fgmdaten(const FGMdaten daten_fgm);` Setzt die Fahrgemeinschaftsdaten.
`Fehlercode gib_fgmdaten(FGMdaten &daten_fgm);` Gibt die Fahrgemeinschaftsdaten zurueck.
`Fehlercode markiere_fgm();` Markiert die Fahrgemeinschaft.
`Fehlercode unmarkiere_fgm();` Macht die Markierung der Fahrgemeinschaft rueckgaengig .
`Fehlercode ist_fgm_markiert(Boolean &markierung);` Prüft ob eine FGM markiert ist.
`Fehlercode setze_status_teilnehmer(const Person_ID per_id,const Status status);` Setzt den Status der Person mit per_id der Fahrgemeinschaft.
`Fehlercode gib_status_teilnehmer(const Person_ID per_id,Status &status);` Gibt den Status der Person mit per_id der Fahrgemeinschaft.
`Fehlercode fest_eintragen(const Person_ID per_id);` Traegt die Person mit per_id in die Fahrgemeinschaft fest ein.
`Fehlercode setze_fahrer_fgm(const Person_ID per_id);` Setzt den Fahrer der Fahrgemeinschaft.
`Fehlercode gib_fahrer_fgm(Person_ID &person_id);` Gibt den Fahrer der Fahrgemeinschaft zurueck.
`Fehlercode setze_freie_plaetze_fgm(const int &plaetze);` Setzt die Anzahl der freien Plaetze der Fahrgemeinschaft auf plaetze zurueck.
`Fehlercode gib_freie_plaetze_fgm(int &plaetze);` Gibt die Anzahl der freien Plaetze der Fahrgemeinschaft zurueck.
`Fehlercode setze_fahrtroute_fgm(const list<Kanten_ID> kanten_id);` Setzt die Fahrtroute der Fahrgemeinschaft mit fgm_id. Die Fahrroute besteht aus einer Liste mit kanten_id mit allen Start- und Zielorten der beteiligten Personen, in der Reihenfolge, wie gefahren wird.
`Fehlercode gib_fahrtroute_fgm(list<Kanten_ID> &kanten_id);` Gibt die Fahrtroute der Fahrgemeinschaft mit fgm_id.
`Fehlercode setze_zeitplan_fgm(const list<Zeit*> zeiten);` Setzt den Zeitplan der Fahrgemeinschaft mit fgm_id. Der Zeitplan enthaelt die Abfahrts- und Ankunfszeiten an allen Start-und Zielorten in der Reihenfolge, wie sie in der Fahrtroute angegeben sind.
`Fehlercode gib_zeitplan_fgm(list<Zeit*> &zeiten);` Gibt den Zeitplan der Fahrgemeinschaft mit fgm_id.

Attribute
`Personenverwalter *personen_ref;` Referenz auf Personenverwalter
`FGM_ID id_fahrgemeinschaft;` Nummer der Fahrgemeinschaft.
`Boolean markierung_fgm;` TRUE = FGM markiert, FALSE = FGM unmarkiert
`Optional<Person_ID> fahrer_fgm;` Fahrer der Fahrgemeinschaft.
`Optional<int> plaetze_fgm;` Freie Plaetze der Fahrgemeinschaft.
`FGM_Datum datum_fgm;` Entstehungsdatum der Fahrgemeinschaft.
`FGM_Datum aenderung_fgm;` Datum der letzten Aenderung.
`list<int> fahrtroute_fgm;` Liste enthaelt die Fahrtroute der Fahrgemeinschaft.
`list<Zeit*> zeitplan_fgm;` Liste enthaelt den Zeitplan der Fahrgemeinschaft.
`list<Person_ID> teilnehmer_liste;` Liste enthaelt die Personen, die in der Fahrgemeinschaften mitfahren.
`dictionary <Person_ID,Status> status_fgm_dic;` Dictionary enthaelt Personen_id und den dazugehoerigen Status der Person.

private Methoden
`void setze_entstehungsdatum_fgm(FGM_Datum & all_datum);` Liefert das aktuelle Datum zurueck.
`Fehlercode suche_status_dic(const Person_ID per_id,Status &per_status);` Gibt zu einer Person_id den Status zurueck.

5.5.3 Algorithmenverwalter

Algorithmenverwalter

öffentliche Methoden
`Algorithmenverwalter(Verkehrsgraph& v);` Konstruktor des Objekts Algorithmenverwalter
`Fehlercode setze_personenverwalter_zeiger(Personenverwalter *zeiger);` Mit der Methode wird der Zeiger auf den Personenverwalter gesetzt.
`Fehlercode setze_bewertungsverwalter_zeiger(Bewertungsverwalter *zeiger);` Mit der Methode wird der Zeiger auf den Bewertungsverwalter gesetzt.
`Fehlercode setze_einteilungsverwalter_zeiger(Einteilungsverwalter *zeiger);` Mit der Methode wird der Zeiger auf den Einteilungsverwalter gesetzt.

Fehlercode setze_fuersorger_zeiger(Fuersorger *zeiger); Mit der Methode wird der Zeiger auf den Fuersorger gesetzt.
Fehlercode gib_optimale_algorithmen(list<Einteilungs_ID> &ein_id_liste, list<string> &ein_namen_liste); Die Methode liefert eine Liste mit den ID's der vorhandenen optimalen Algorithmen und eine Liste mit deren Namen.
Fehlercode gib_heuristische_algorithmen(list<Einteilungs_ID> &ein_id_liste, list<string> &ein_namen_liste); Die Methode liefert eine Liste mit den ID's der vorhandenen heuristischen Algorithmen und eine Liste mit deren Namen.
Fehlercode gib_inkrementelle_algorithmen(list<Einteilungs_ID> &ein_id_liste, list<string> &ein_namen_liste); Die Methode liefert eine Liste mit den ID's der vorhandenen inkrementellen Algorithmen und eine Liste mit deren Namen.
Fehlercode gib_wegsuche_algorithmen(list<Weg_ID> &weg_id_listes, list<string> &weg_namen_liste); Die Methode liefert eine Liste mit den ID's der vorhandenen Wegsuchealgorithmen und eine Liste mit deren Namen.
Fehlercode gib_aktuellen_optimalen_algor(Einteilungs_ID &ein_id); Die Methode gibt die ID des aktuellen optimalen Algorithmus zurück.
Fehlercode gib_aktuellen_heuristischen_algor(Einteilungs_ID &ein_id); Die Methode gibt die ID des aktuellen heuristischen Algorithmus zurück.
Fehlercode gib_aktuellen_inkrementellen_algor(Einteilungs_ID &ein_id); Die Methode gibt die ID des aktuellen inkrementellen Algorithmus zurück.
Fehlercode gib_aktuellen_wegsuche_algor(Weg_ID &weg_id); Die Methode gibt die ID des aktuellen Wegsuchealgorithmus zurück.
Fehlercode wurde_berechnung_unterbrochen(); Gibt im Fehlercode zurück, ob der Benutzer die Berechnung abbrechen will.
Fehlercode setze_aktuellen_optimalen_algor(const Einteilungs_ID ein_id); Der Algorithmus mit der ID ein_id wird der neue aktuelle optimale Algirithmus.
Fehlercode setze_aktuellen_heuristischen_algor(const Einteilungs_ID ein_id); Der Algorithmus mit der ID ein_id wird der neue aktuelle heuristische Algirithmus.
Fehlercode setze_aktuellen_inkrementellen_algor(const Einteilungs_ID ein_id); Der Algorithmus mit der ID ein_id wird der neue aktuelle inkrementelle Algorithmus.
Fehlercode setze_aktuellen_wegsuche_algor(const Weg_ID weg_id); Der Algorithmus mit der ID ein_id wird der neue aktuelle Wegsuchealgorithmus. Fehlercode TABELLEN_NICHT_GELADEN, falls bei Randknoten- oder Rathaussuche noch die Tabellen von Festplatte geladen werden müssen.

Fehlercode lade_randknotentabellen(const char* entf_randknotenfile, const char* start_ziellistfile); Die Tabellen für die Randknotensuche (Wegsuchealg.) wird von Festplatte geladen.
Fehlercode lade_rathaustabellen(const char* entf_rathaeuserfile, const char* start_ziellistfile); Die Tabellen für die Rathaussuche (Wegsuchealg.) wird von Festplatte laden.
Fehlercode randknotentabellen_aufbauen_und_speichern(const char* entf_randknotenfile, const char* start_ziellistfile); Die Methode baut hierarchischen Graphen für Randknotensuche auf und speichert Tabellen in zwei Files: Randknotenentfernung und Start-/Ziellisten. Danach ist das Randknotenmodell aktiv.
Fehlercode rathaustabellen_aufbauen_und_speichern(const char* entf_rathaeuserfile, const char* start_ziellistfile); Die Methode baut hierarchischen Graphen für Rathaussuche auf und speichert Tabellen in zwei Files: Rathausentfernungen und Start-/Zielliste. Danach ist das Rathausmodell aktiv.
Fehlercode starte_optimale_berechnung(float &laufzeit); Der aktuelle optimale Algorithmus zur Einteilungsberechnung wird gestartet.
Fehlercode starte_heuristische_berechnung(float &laufzeit); Der aktuelle heuristische Algorithmus zur Einteilungsberechnung wird gestartet.
Fehlercode starte_inkrementelle_berechnung(float &laufzeit); Der aktuelle inkrementelle Algorithmus zur Einteilungsberechnung wird gestartet.
Fehlercode starte_einzelwegsuche(const Kanten_ID start, const Kanten_ID ziel, list<Kanten_ID> &ergebnisliste, int& fahrtzeit, float& laufzeit); Die Methode liefert als Ergebnis eine Liste mit den Kanten-ID's des kürzesten Weges von start nach ziel. Bei Wegsuchealgorithmen, die nur die Fahrtzeit berechnen, wird eine leere Liste zurückgegeben.
Fehlercode starte_mehrzielsuche(const Kanten_ID start, list<Kanten_ID> zielliste, list<int> &fahrzeitenliste); Die Methode liefert als Ergebnis eine Liste der Fahrtzeiten von der Startkante start zu allen Zielkanten in der Liste zielliste.
Fehlercode letzte_berechnung_fortsetzen(); Wenn eine abgebrochene Berechnung existiert, so wird diese fortgesetzt.
Fehlercode wegsuche_nmal(int n, float& zeit); Die Methode berechnet kürzeste Wege zwischen n zufällig gewählten Paaren von Start- und Zielorten und gibt die Berechnungszeit zurück.
Fehlercode gib_personenliste(list<Person_ID> &per_id_liste); Gibt eine Liste mit Personen-ID's aller Personen der Personendatei zurueck.
Fehlercode gib_fahrer(const Person_ID pid, Boolean &istfahrer); Gibt die Information ueber den Fahrer zurueck
Fehlercode gib_anzahl_der_plaetze(const Person_ID per_id, int &anzahl_der_plaetze); Gibt die Anzahl der Plaetze der Person mit der id zurueck.

Fehlercode gib_startort_kanten_id(const Person_ID per_id, int &startort_kanten_id); Gibt die Kante des Startortes der Person mit id zurueck.
Fehlercode gib_zielort_kanten_id(const Person_ID per_id, int &zielort_kanten_id); Gibt die Kante des Zielortes der Person mit id zurueck.
Fehlercode neue_einteilung(const string name, Einteilungs_ID &ein_id); Fügt eine neue Einteilung mit name in das System ein und gibt die zugeornete ein_id zurück.
Fehlercode berechne_route_fgm(FGM_ID fgm_id, list<Kanten_ID> &kanten_id); Die Methode berechnet aus der Fahrtroute einer FGM (= Kantenliste mit allen Start- und Zielorten der Teilnehmer in der richtigen Reihenfolge) den genauen Weg der FGM. Als Ergebnis wird eine Kantenliste mit allen Kanten der Fahrtroute zurückgegeben.
Fehlercode setze_aktuelle_einteilung(const Einteilungs_ID ein_id); Die Methode setzt die aktuelle Einteilung auf die Einteilung mit ein_id.
Fehlercode gib_markierte_fgm(list<FGM_ID> &liste); Gibt eine Liste mit den fgm_id's der markierten FGM's aus er aktuellen Einteilung zurück.
Fehlercode gib_id_fgm(list<FGM_ID> &liste); Gibt eine Liste mit allen fgm_id der aktuellen Einteilung zurück.
Fehlercode neue_fgm(FGM_ID &fgm_id); Fügt eine neue Fahrgemeinschaft in die aktuelle Einteilung ein und gibt die zugeordnete fgm_id zurück.
Fehlercode gib_noch_nicht_vermittelte_personen(list<Person_ID> &liste); Gibt eine Liste mit allen per_id zurück, die noch nicht vermittelt wurden.
Fehlercode gib_teilnehmer_fgm(const FGM_ID fgm_id, list<Person_ID> &per_id_liste); Gibt eine Liste mit per_id zurück, die in der Fahrgemeinschaft mit fgm_id mitfahren.
Fehlercode neuer_fahrer(const FGM_ID fgm_id,const Person_ID per_id); Fuegt den Fahrer in die Fahrgemeinschaft mit fgm_id ein.
Fehlercode neuer_teilnehmer(const FGM_ID fgm_id, const Person_ID per_id); Fügt Person mit id in die Fahrgemeinschaft mit fgm_id ein.
Fehlercode setze_fahrer_fgm(const FGM_ID fgm_id, const Person_ID per_id); Setzt den Fahrer der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_freie_plaetze_fgm(const FGM_ID fgm_id, const int plaetze); Setzt die Anzahl der freien Plätze der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_fahrtroute_fgm(const FGM_ID fgm_id,const list<int> kanten_id_liste);

Setzt die Fahrtroute der Fahrgemeinschaft mit fgm_id. Die Fahrroute besteht aus einer Liste mit kanten_id mit allen Start- und Zielorten der beteiligten Personen, in der Reihenfolge, wie gefahren wird.
Fehlercode setze_zeitplan_fgm(const FGM_ID fgm_id, const list<Zeit*> &zeiten); Setzt den Zeitplan der Fahrgemeinschaft mit fgm_id. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode loesche_einteilung(const Einteilungs_ID ein_id); Loescht die Einteilung mit der ID.
Fehlercode neue_einteilung_mit_gen_namen(Einteilungs_ID &ein_id); Bildet Einteilung mit generischem Namen
Fehlercode setze_fgmdaten(const FGM_ID fgm_id,const FGMdaten fgm); Setzt die Fahrgemeinschaftsdaten.
Fehlercode gib_fgmdaten(const FGM_ID fgm_id,FGMdaten &fgm); Gibt die Fahrgemeinschaftsdaten zurück.
Fehlercode aktuelle_einteilung_mit(FGMliste &fgm_id_liste); Fuer jede Personen, die noch nicht vermittelt wurden, wird eine Einzelfahrgemeinschaft angelegt. Die Liste aller nun vorhandenen Fahrgemeinschaften wird zurueckgeliefert.
Fehlercode aktuelle_einteilung_ohne(); Jede Fahrgemeinschaft wird geloescht, die keinen oder einen Teilnehmer besitzt.
Fehlercode bewerte_und_bilde(list<Person_ID> teilnehmerliste, Person_ID fahrer); Bewertet eine FGM *und legt sie gleich an*
Fehlercode bewerte_fgm(const FGM_ID fgm_id, Bewertung &wert); Methode bewertet die Güte der Fahrgemeinschaft fgm_id der aktuellen Einteilung und gibt die Bewertung in &wert zurück.
Fehlercode bewerte_aktuelle_einteilung(Bewertung &wert); Methode bewertet die Güte der aktuellen Einteilung mit der aktuellen Bewertungsfunktion und gibt die Bewertung in &wert zurück.
Fehlercode bewerte_fgm_vereinigung(const list<Person_ID> &per_id_liste_fgm1, const list<Person_ID> &per_id_liste_fgm2, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personenmengen von fgm1 und fgm2 wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.

Fehlercode bewerte_fgm_bildung(const list<Person_ID> &per_id_liste, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen der Mengen person_id_liste wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode bewerte_fgm_mit(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen von fgm_id vereint mit der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode bewerte_fgm_ohne(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen von fgm_id ohne der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
void systemmeldung(string text,const Meldungsart art); Gibt eine Systemmeldung an den Menüverwalter, der die Ausgabe übernimmt
void wurde_berechnung_unterbrochen(Abbruch& abbruch); Fragt nach, ob die Berechnung mit "u" oder "Esc" abgebrochen wurde
Fehlercode gib_entfernung(const Kanten_ID start,const Kanten_ID ziel,int &fahrtzeit); Hier wird die Entfernung zwischen Start und Ziel zurueckgeliefert. Allerdings wird hierfuer eine Tabelle im Hauptspeicher benutzt, d.h. ist die Entfernung zweier Kanten in der Tabelle vorhanden, so wird dieser Wert zurueckgegben, ansonsten wird die Entfernung neu berechnet und in die Tabelle eingetragen.
Fehlercode name_fuer_neue_einteilung(string& name); Gibt einen Namen fuer die neue Einteilung zurueck
Fehlercode integer_parameter(const string beschreibung,int& wert); Fragt nach beschreibung und gibt einen Integer-Wert zurueck
Fehlercode double_parameter(const string beschreibung,double& wert); Fragt nach beschreibung und gibt einen Double-Wert zurueck
Fehlercode boolean_parameter(const string beschreibung,Boolean& wert); Fragt nach beschreibung und gibt einen Boolean-Wert zurueck

Attribute
dictionary<Einteilungs_ID,Einteilungs_algorithmus*> optimale_algor; Liste mit den optimalen Algorithmen.
dictionary<Einteilungs_ID,Einteilungs_algorithmus*> heuristische_algor; Liste mit den heuristischen Algorithmen.
dictionary<Einteilungs_ID,Einteilungs_algorithmus*> inkrementelle_algor; Liste mit den inkrementellen Algorithmen.
dictionary<Weg_ID,Wegsuche_algorithmus*> wegsuche_algor; Liste mit den Wegsuchealgorithmen.
list<Weg_ID> weg_id_liste; Liste der Wegsuchealgorithmen-IDs
list<string> weg_namen_liste; Liste der Wegsuchealgorithmennamen
Einteilungs_ID aktueller_optimaler_algor; Einteilungsalgor.-ID des aktuellen optimalen Algorithmus.
Einteilungs_ID aktueller_heuristischer_algor; Einteilungsalgor.-ID des aktuellen heuristischen Algorithmus.
Einteilungs_ID aktueller_inkrementeller_algor; Einteilungsalgor.-ID des aktuellen inkrementellen Algorithmus.
Weg_ID aktueller_wegsuche_algor; Einteilungsalgor.-ID des aktuellen Wegsuchealgorithmus.
Boolean es_ex_abgebr_berechnung; Hinweis daruf, ob eine abgebrochene Berechnung existiert.
Personenverwalter *personenverwalter_zeiger; Zeiger auf den Personenverwalter.
Bewertungsverwalter *bewertungsverwalter_zeiger; Zeiger auf den Bewertungsverwalter (sich selbst).
Einteilungsverwalter *einteilungsverwalter_zeiger; Zeiger auf den Einteilungssverwalter.
Fuersorger *fuersorger_zeiger; Zeiger auf den Fuersorger.
Entfernungstabelle *entfernungstabellen_zeiger; Zeiger auf die Entfernungstabelle.
Verkehrsgraph& vg; Referenz auf Verkehrsgraph
Dijkstra_notarget dijk_notarget; Dijkstra-Algorithmus ohne Zielknoten (komplettes Durchsuchen des Graphen)
Dijk_manytargets dijk_many;

Dijkstra-Algorithmus mit mehreren Zielknoten
Dijk_first_k_targets dijkfirst; Dijkstra-Algorithmus, der kürzesten Weg zu den k nächsten Zielen aus einer Zielknotenmenge berechnet
Level2Graph_Rand vg2_rand; Graph der Randknoten auf Level 2 des Verkehrsgraphen
Level2Graph_Rathaus vg2_rathaus; Graph der Rathäuser auf Level 2 des Verkehrsgraphen

Entfernungstabelle

öffentliche Methoden
Entfernungstabelle(); Konstruktor der Klasse Entfernungstabelle.
~Entfernungstabelle(); Destruktor der Klasse Entfernungstabelle.
Fehlercode setze_algorithmenverwalter(Algorithmenverwalter *verwalter); Setzt die Referenz auf den Algorithmenverwalter.
Fehlercode loesche_tabelle(); Alle Eintraege der Tabelle werden geloescht.
Fehlercode erstelle_tabelle(); Die Methode erstellt eine Tabelle, in der alle Entfernungen zwischen allen Start- und Zielorten, der in der Personendatei enthaltenen Personen, abgelegt werden.
Fehlercode gib_entfernung(const Kanten_ID start,const Kanten_ID ziel,int &fahrtzeit); Die Methode sucht aus der im Hauptspeicher abgelegten Entfernungstabelle die Fahrtzeit zwischen den uebergebenen Kanten heraus und gibt das Ergebnis zurueck.

Attribute
Algorithmenverwalter *algorithmenverwalter_ref; Referenz auf den Algorithmenverwalter.
dictionary<int,dictionary<int,int>*> tabelle; Die Tabelle, in der die Entfernungen abgespeichert werden.

5.5.4 Algorithmen

Einteilungs_algorithmus

öffentliche Methoden
Einteilungs_algorithmus(class Algorithmenverwalter* av, Algorithmus_ID die_id, string name); Konstruktor

virtual Fehlercode gib_id(Algorithmus_ID &eine_id); Gibt die ID des Algorithmus zurück
virtual Fehlercode gib_name(string &ein_name); Gibt den Namen der Einteilungsalgorithmus als Ergebnis zurück.
virtual Fehlercode start(float &laufzeit) = 0; Abstrakte Methode (nicht implementiert) zum Start des Algorithmus.

Attribute
string name; Name des Einteilungsalgorithmus.
Algorithmus_ID id; Die ID des Einteilungsalgorithmus.
class Algorithmenverwalter* algorithmenverwalter; Zeiger auf den algorithmenverwalter

MM_algorithmus : public Einteilungs_algorithmus

öffentliche Methoden
MM_algorithmus(Algorithmenverwalter* av, Algorithmus_ID die_id, string name); Konstruktor
Fehlercode start(float &laufzeit); Die Methode startet den Algorithmus.

private Methoden
void merke(const int knoten_id_i,const int knoten_id_j, dictionary<int,list<int>*>& speicher); Die Methode traegt zwei Knoten-ID's in das Dictionary ein. Mit der Methode schon_geprueft(...) kann dann abgefragt werden, ob merke(...) fuer ein bestimmtes Knoten-ID-Paar schon aufgerufen wurde.
Boolean schon_geprueft(const int knoten_id_i,const int knoten_id_j, dictionary<int,list<int>*>& speicher); Die Methode kann feststellen, on merke(...) fuer das Knoten-ID-Paar (knoten_id_i,knoten_id_j) schon aufgerufen wurde.
void loesche_dict(dictionary<int,list<int>*>& speicher); Die Methode loescht das uebergebene Dictionary.
void vergesse(const int knoten_id, dictionary<int,list<int>*>& speicher); Alle Informationen bezueglich des Knotens mit knoten_id werden aus dem 'speicher' geloescht.

Ink_algorithmus : public Einteilungs_algorithmus

öffentliche Methoden
Ink_algorithmus(Algorithmenverwalter* av, Algorithmus_ID die_id, string name); Konstruktor
Fehlercode start(float &laufzeit); Startet den Algorithmus

Attribute
int iterationen; Anzahl der maximalen Durchläufe des Algorithmus
double guete_obergrenze; Güte bei deren Überschreiten abgebrochen werden darf

5.5.5 Bewertungsverwalter

Bewertungsverwalter

öffentliche Methoden
Bewertungsverwalter(); Konstruktor
~Bewertungsverwalter(); Destruktor
Fehlercode alles_loeschen(); Alle Bewertungsfunktionen werden geloescht.
Fehlercode setze_personenverwalter_zeiger(Personenverwalter *zeiger); Mit der Methode wird der Zeiger auf den Personenverwalter gesetzt.
Fehlercode setze_einteilungsverwalter_zeiger(Einteilungsverwalter *zeiger); Mit der Methode wird der Zeiger auf den Einteilungsverwalter gesetzt.
Fehlercode setze_algorithmenverwalter_zeiger(Algorithmenverwalter *zeiger); Mit der Methode wird der Zeiger auf den Algorithmenverwalter gesetzt.
Fehlercode setze_voreinstellungen_zeiger(Voreinstellungen *zeiger); Mit der Methode wird der Zeiger auf die Voreinstellungen gesetzt.
Fehlercode gib_alle_bew_fkt(list<Bewertungs_ID> &bew_id_liste, list<string> &bew_name_liste); Liefert eine Liste mit allen Bewertungsfunktionen und eine Liste mit deren Namen.
Fehlercode gib_aktuelle_bew_fkt(Bewertungs_ID &bew_id); Die Methode gibt die ID der aktuellen Bewertungsfunktion in bew_id zurück.
Fehlercode setze_aktuelle_bew_fkt(const Bewertungs_ID bew_id); Die Methode setzt aktuelle Bewertungsfunktion auf bew_id.
Fehlercode loesche_bew_fkt(const Bewertungs_ID bew_id); Die Methode löscht die Bewertungsfunktion bew_id.

Fehlercode neue_bew_fkt(const string bew_name, Bewertungs_ID &bew_id); Es wird eine neue Bewertungsfunktion mit dem Namen bew_name angelegt und deren ID zurückgegeben.
Fehlercode gib_parameter_bew_fkt(const Bewertungs_ID bew_id, Bew_fkt_parameter &parametersatz); Die Parameter der aktuellen Bewertungsfunktion werden zurückgegeben.
Fehlercode setze_parameter_bew_fkt(const Bewertungs_ID bew_id, Bew_fkt_parameter &parametersatz); Die Parameter der Bewertungsfunktion mit der ID bew_id werden auf die übergebenen Werte gesetzt.
Fehlercode gib_geaenderte_bew_fkten(list<Bewertungs_ID> &bew_id_liste); Die Methode gibte eine Liste zurück, die die ID's aller Bewertungsfunktionen enthält, die seit der letzten Änderung nicht auf Platte gespeichert wurden.
Fehlercode gib_min_max(int &min_anzahl_fgm, int &max_anzahl_fgm, Bewertung &min_bew, Bewertung &max_bew); Die Methode liefert die Grenzwerte für die Anzahl der FGM'en einer Einteilung und der Bewertung einer Einteilung auf den aktuellen Personendaten mit der aktuellen Bewertungsfunktion.
Fehlercode bewerte_fgm(const FGM_ID fgm_id, Bewertung &wert); Die Methode bewertet die Güte der Fahrgemeinschaft fgm_id der aktuellen Einteilung und gibt die Bewertung in &wert zurück.
Fehlercode bewerte_aktuelle_einteilung(Bewertung &wert); Die Methode bewertet die Güte der aktuellen Einteilung mit der aktuellen Bewertungsfunktion und gibt die Bewertung in &wert zurück.
Fehlercode bewerte_fgm_vereinigung(const list<Person_ID> &per_id_liste_fgm1, const list<Person_ID> &per_id_liste_fgm2, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personenmengen von fgm1 und fgm2 wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode bewerte_fgm_bildung(const list<Person_ID> &per_id_liste, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen der Mengen person_id_liste wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.

Fehlercode bewerte_fgm_mit(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen von fgm_id vereint mit der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode bewerte_fgm_ohne(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer); Die Bildung einer FGM aus den Personen von fgm_id ohne der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
Fehlercode gib_fahrer(const Person_ID per_id,Boolean &fahrer); Gibt die Eigenschaft Fahrer der Person mit id zurueck.
Fehlercode gib_komfortklasse(const Person_ID per_id,Komfortklasse &komfortklasse); Gibt die Eigenschaft Komfortklasse der Person mit id zurueck.
Fehlercode gib_anzahl_der_plaetze(const Person_ID per_id, int &anzahl_der_plaetze); Gibt die Eigenschaft Anzahl der Plaetze der Person mit id zurueck
Fehlercode gib_startort_kanten_id(const Person_ID per_id, int &startort_kanten_id); Gibt die Kante des Startortes der Person mit id zurueck.
Fehlercode gib_zielort_kanten_id(const Person_ID per_id, int &zielort_kanten_id); Gibt die Kante des Zielortes der Person mit id zurueck.
Fehlercode gib_ankunftzeit(const Person_ID per_id, Intervall &ankunftzeit); Gibt das Intervall Ankunftszeit der Person mit id zurueck.
Fehlercode gib_arbeitsdauer(const Person_ID per_id, Zeit &arbeitsdauer); Gibt die Arbeitsdauer der Person mit id zurueck.
Fehlercode ist_person(const Person_ID per_id); Prueft, od die Person mit der per_id vorhanden ist. Wenn sie vorhanden ist wird als Fehlercode OK zurueckgegeben, sonst wird KEINE_PERSON zurueckgegeben.
Fehlercode gib_personen_liste(list<Person_ID> &liste); Gibt eine Liste der ID's aller Personen der Personendatei zurueck.
Fehlercode gib_abneigung_liste(const Person_ID per_id,list <Person_ID> &abneigung);

Liefert eine Liste mit Personen, mit denen die Person nicht zusammenfahren moechte.
Fehlercode gib_zuneigung_liste(const Person_ID per_id,list <Person_ID> &zuneigung); Liefert eine Liste mit Personen, mit denen die Person zusammenfahren moechte.
Fehlercode gib_geschlecht(const Person_ID per_id, Geschlecht &geschlecht); Gibt das Geschlecht der Person mit der id zurueck.
Fehlercode gib_raucher(const Person_ID per_id,Raucher &raucher); Gibt die Eigenschaft Raucher der Person mit der id zurueck, d.h. ob Raucher oder der nicht.
Fehlercode gib_geschlecht_wunsch(const Person_ID per_id, Geschlecht &wunsch); Gibt das Geschlecht zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_geschlecht_gewicht(const Person_ID per_id,int &gewicht); Gibt die Gewichtung von Geschlecht zurueck.
Fehlercode gib_rauchen_wunsch(const Person_ID per_id,Raucher &wunsch); Gibt das Rauchen zurueck, fuer das eine Gewichtung angegeben wurde.
Fehlercode gib_rauchen_gewicht(const Person_ID per_id,int &gewicht); Gibt die Gewichtung von Rauchen zurueck.
Fehlercode gib_id_fgm(list<FGM_ID> &fgm_id); Gibt eine Liste mit allen fgm_id der aktuellen Einteilung zurueck.
Fehlercode pruefe_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze); Prueft die Anzahl der freien Plaetze der Fahrgemeinschaft mit fgm_id und gib die Anzahl zurueck.
Fehlercode gib_teilnehmer_fgm(const FGM_ID fgm_id, list<Person_ID> &per_id); Gibt eine Liste mit per_id zurueck, die in der Fahrgemeinschaft mit fgm_id mitfahren.
Fehlercode gib_fahrer_fgm(const FGM_ID fgm_id,Person_ID &per_id); Gibt den Fahrer der Fahrgemeinschaft mit fgm_id zurueck.
Fehlercode gib_fahrtroute_fgm(const FGM_ID fgm_id, list<Kanten_ID> &kanten_id); Gibt die Fahrtroute der Fahrgemeinschaft mit fgm_id.
Fehlercode starte_einzelwegsuche(const Kanten_ID start, const Kanten_ID ziel, list<Kanten_ID> &ergebnisliste, int& fahrtzeit, float& laufzeit); Die Methode liefert als Ergebnis eine Liste mit den Kanten-ID's des kürzesten Weges von start nach ziel. Bei Wegsuchealgorithmen, die nur die Fahrtzeit berechnen, wird eine leere Liste zurückgegeben.
Fehlercode gib_entfernung(const Kanten_ID start,const Kanten_ID ziel,int &fahrtzeit);

Hier wird die Entfernung zwischen Start und Ziel zurueckgeliefert. Allerdings wird hierfuer eine Tabelle im Hauptspeicher benutzt, d.h. ist die Entfernung zweier Kanten in der Tabelle vorhanden, so wird dieser Wert zurueckgegben, ansonsten wird die Entfernung neu berechnet und in die Tabelle eingetragen.
`Fehlercode gib_p_max(int &p_max);` Methoden zur Abfrage der Voreinstellungen

Attribute
`dictionary<Bewertungs_ID,Bewertungsfunktion*> bewertungsfunktionen;` Assoziativliste der verwalteten Bewertungsfunktionen.
`Bewertungs_ID aktuelle_bew_fkt;` die ID der aktuellen Bewertungsfunktion
`Bewertungs_ID id_zaehler;` der Bewertungs-ID-Zaehler; er wird bei der instanzierung des Bewertungsverwalters auf 0 gesetz und dann jeweils beim erstellen einer neuen Bewertungsfunktion inkrementiert
`Personenverwalter *personenverwalter_zeiger;` Zeiger auf den Personenverwalter
`Algorithmenverwalter *algorithmenverwalter_zeiger;` Zeiger auf den Algorithmenverwalter
`Einteilungsverwalter *einteilungsverwalter_zeiger;` Zeiger auf den Einteilungsverwalter
`Voreinstellungen *voreinstellungen_zeiger;` Zeiger auf die Voreinstellungen

Bewertungsfunktion

öffentliche Methoden
`Fehlercode bewerte_fgm(const FGM_ID fgm_id, Bewertung &wert);` Die Methode bewertet die Güte der Fahrgemeinschaft fgm_id der aktuellen Einteilung und gibt die Bewertung in &wert zurück.
`Fehlercode bewerte_aktuelle_einteilung(Bewertung &wert);` Die Methode bewertet die Güte der aktuellen Einteilung und gibt die Bewertung in &wert zurück.
`Fehlercode bewerte_fgm_vereinigung(const list<Person_ID> &per_id_liste_fgm1, const list<Person_ID> &per_id_liste_fgm2, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer);` Die Bildung einer FGM aus den Personenmengen von fgm1 und fgm2 wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.

`Fehlercode bewerte_fgm_bildung(const list<Person_ID> per_id_liste, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer);` Die Bildung einer FGM aus den Personen der Mengen person_id_liste wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
`Fehlercode bewerte_fgm_mit(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer);` Die Bildung einer FGM aus den Personen von fgm_id vereint mit der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
`Fehlercode bewerte_fgm_ohne(const FGM_ID fgm_id, const Person_ID per_id, Bewertung &wert, list<Kanten_ID> &kanten_id_liste, list<Zeit*> &zeitpunkt_liste, Person_ID &fahrer);` Die Bildung einer FGM aus den Personen von fgm_id ohne der Person per_id wird bewertet. Als Ergebnis wird die Bewertung, die ID des Fahrers, die Fahrtroute und der Zeitplan zurückgegeben. Die Fahrtroute besteht aus den Kanten-ID's aller Start- und Zielorte der an der FGM beteiligten Personen, in der richtigen Reihenfolge. Der Zeitplan enhält die Abfahrtzeit und die Ankunftszeiten an allen Start- und Zielorten in der Reihenfolge wie sie in der Fahrtroute angegeben sind.
`Fehlercode gib_min_max(int &min_anzahl_fgm, int &max_anzahl_fgm, Bewertung &min_bew, Bewertung &max_bew);` Die Methode liefert die Grenzwerte für die Anzahl der FGM'en einer Einteilung und der Bewertung einer Einteilung auf den aktuellen Personendaten.
`Fehlercode setze_bew_verwalter_zeiger(Bewertungsverwalter *zeiger);` Mit der Methode kann der Zeiger auf den Bewertungsverwalter gesetzt werden.
`Fehlercode gib_name(string &bew_name);` Der Name der Bewertungsfunktion wird in bew_fkt_name zurückgegeben.
`Fehlercode setze_name(const string bew_name);` Der Name der Bewertungsfunktion wird auf bew_name gesetzt.
`Fehlercode gib_id(Bewertungs_ID &bew_id);` Die ID der Bewertungsfunktion wird in bew_id zurückgegeben.
`Fehlercode setze_id(const Bewertungs_ID bew_id);` Die ID der Bewertungsfunktion wird auf bew_id gesetzt.
`Fehlercode gib_u_max(int &wert);` Die Methode liefert den Wert von u_max in wert zurueck.

Fehlercode setze_u_max(const int wert); Mit dieser Methode kann der Wert von u_max auf wert gesetzt werden.
Fehlercode gib_z_max(int &wert); Die Methode liefert den Wert von z_max in wert zurueck.
Fehlercode setze_z_max(const int wert); Mit dieser Methode kann der Wert von z_max auf wert gesetzt werden.
Fehlercode gib_na_max(int &wert); Die Methode liefert den Wert von na_max in wert zurueck.
Fehlercode setze_na_max(const int wert); Mit dieser Methode kann der Wert von na_max auf wert gesetzt werden.
Fehlercode gib_e_max(double &wert); Die Methode liefert den Wert von e_max in wert zurueck.
Fehlercode setze_e_max(const double wert); Mit dieser Methode kann der Wert von e_max auf wert gesetzt werden.
Fehlercode gib_gewicht_z(int &wert); Die Methode liefert den Wert von gewicht_z in wert zurueck.
Fehlercode setze_gewicht_z(const int wert); Mit dieser Methode kann der Wert von gewicht_z auf wert gesetzt werden.
Fehlercode gib_gewicht_u(int &wert); Die Methode liefert den Wert von gewicht_u in wert zurueck.
Fehlercode setze_gewicht_u(const int wert); Mit dieser Methode kann der Wert von gewicht_u auf wert gesetzt werden.
Fehlercode gib_gewicht_na(int &wert); Die Methode liefert den Wert von gewicht_na in wert zurueck.
Fehlercode setze_gewicht_na(const int wert); Mit dieser Methode kann der Wert von gewicht_na auf wert gesetzt werden.
Fehlercode gib_gewicht_ne(int &wert); Die Methode liefert den Wert von gewicht_ne in wert zurueck.
Fehlercode setze_gewicht_ne(const int wert); Mit dieser Methode kann der Wert von gewicht_ne auf wert gesetzt werden.
Fehlercode gib_gewicht_e(int &wert); Die Methode liefert den Wert von gewicht_e in wert zurueck.
Fehlercode setze_gewicht_e(const int wert); Mit dieser Methode kann der Wert von gewicht_e auf wert gesetzt werden.
Fehlercode gib_gewicht_p(int &wert); Die Methode liefert den Wert von gewicht_p in wert zurueck.
Fehlercode setze_gewicht_p(const int wert); Mit dieser Methode kann der Wert von gewicht_p auf wert gesetzt werden.
Fehlercode gib_heuristik_verwenden(Boolean &wert); Die Methode liefert den Wert des Parameters heuristik_verwenden in wert zurueck.
Fehlercode setze_heuristik_verwenden(const Boolean wert); Mit dieser Methode kann der Wert von heuristik_verwenden auf wert gesetzt werden.

private Methoden
Fehlercode ankunftszeit(const list<Person_ID> &per_liste,Zeit &zeitpunkt);

Die Methode liefert den Anfangszeitpunkt der Schnittmenge von den Ankunftszeiten der Personen in per_liste. Der Fehlercode ist OK, wenn eine Schnittmenge existiert, sonst ist er KEINE_SCHNITTMENGE.
Fehlercode arbeitslaenge(const list<Person_ID> &per_liste, double &z); Die Methode vergleicht die Arbeitszeiten der Personen in per_liste. Wenn die Arbeitszeiten um mehr als z_max auseinander liegen, wird der Fehlercode Z_MAX_UEBERSCHRITTEN zurueckgegeben, sonst wird die Bewertung z bezueglich der laenge der Arbeitszeiten berechnet.
Fehlercode tausche_listenelemente(int i, int j, list<int> &eine_liste); Die Methode vertauscht die Listenelement i und j.
Fehlercode berechne_fahrtroute(const list<Person_ID> &teilnehmerliste,Person_ID &fahrer,list<Kanten_ID> &bester_weg, list<Zeit*> &zeitplan,double &wert); Die Methode berechnet fuer die Fahrgemeinschaft, die aus den Personen der teilnehmerliste besteht, den Weg, bei dem der Fahrer den kuerzesten Umweg hat. Als Ergebnis werden der beste_weg, der fahrer der zeitplan und die Bewertung bezueglich des Umweges (u_max) zurueckgegeben. Wird kein Weg gefunden, oder ist der kuerzeste Umweg groesser als u_max, so wird der Fehlercode KEIN_WEG_GEFUNDEN zurueckgegebn. Der wert (=Ergebnis der Bewertung) liegt zuwischen 0.0 und 1.0. Wenn der Fahrer keinen Umweg machen muss, so wird wert=0.0, ist der Umweg = u_max so ergibt sich wert=1.0 und ist der Umweg zwischen 0 und u_max, so liegt ergibt sich 0.0 ¡ wert ¡ 1.0. Je kleiner wert ist, desto besser ist die Bewertung bezueglich des Umweges.
void list_perm(const dictionary<int,Kanten_ID> &kantentabelle_mitfahrer,const list<int> &zielortliste,const Kanten_ID startort, const Kanten_ID zielort,list<Kanten_ID> &weg,int &weg_laenge,int x, list<int> &permutationsliste,Fehlercode &fc); Die Methode permutiert die Permutationsliste und ruft fuer jede Permutation, die einem gueltigen Weg entspricht die Methode berechne_kantenliste auf. Die Ergebnisse von berechne_kantenliste werden einfach druchgereicht.
Fehlercode umweg_fuer_fahrer_x(const Person_ID fahrer, const list<Person_ID> &teilnehmerliste, list<Kanten_ID> &weg, int &umweg_laenge); Der kuerzeste Umweg fuer einen Fahrer, der die Personen aus der Teilnehmerliste mitnimmt, wird berechnet. Ergebnisse sind die Kantenliste des Weges und dessen Laenge. Wird kein Weg gefunden, so wird der Fehlercode KEIN_WEG_GEFUNDEN zurueckgegeben.
Fehlercode ist_weg_gueltig(const list<int> &permutationsliste, const list<int> &zielortliste);

Die Methode ueberprueft, ob der durch die permutationsliste beschriebene Weg gueltig ist oder nicht. Die permutationsliste besteht aus Integer-Zahlen, die den Start- und Zielorten der Teilnehmer einer FGM zugeordnet sind. Dabei entsprechen die ungeraden Zahlen den Startorten und die geraden Zahlen den Zielorten. Wenn p den Startort einer Persone repraesentiert, dann ist p+1 der Zielort dieser Person. Ein Weg ist gueltig, wenn alle Startorte vor den zugeordneten Zielorten in der Liste vorkommen. Das bedeutet, dass alle Personen zuerst am Startort abgeholt werden und dann am Zielort wieder abgesetzt werden. Wenn der weg nicht gueltig ist, so wird der Fehlercode KEIN_GUELTIGER_WEG zurueckgegeben, sonst wird OK zurueckgegeben.
`Fehlercode berechne_kantenliste(const dictionary<int,Kanten_ID> &kantentabelle, const Kanten_ID startort,const Kanten_ID zielort, const list<int> &perm_liste,list<Kanten_ID> &weg,int &weg_laenge);` Die Methode berechnet die Kantenliste, die durch die Permutationsliste (perm_liste) bestimmt wird. Nach Erstellen der Kantenliste wird die Laenge des durch sie beschriebenen Weges berechnet und falls dieser kuerzer ist als der zuvor gefundene Weg, so wird der neue Weg und dessen Laenge als Ergebnis zurueckgeliefert. Enthaelt der Weg Kanten, die nicht zum Verkehrsgraph gehoeren, oder Kanten, die im Verkehrsgraph nicht erreichbar sind (=Kanten auf einer Insel im Verkehrsgraph), so wird der Fehlercode KEIN_WEG_GEFUNDEN zurueckgegeben.
`Fehlercode umweg_fuer_fahrer_x_heuristisch(const Person_ID fahrer, const list<Person_ID> &teilnehmerliste, list<Kanten_ID> &weg, int &umweg_laenge);` Der kuerzeste Umweg fuer einen Fahrer, der die Personen aus der Teilnehmerliste mitnimmt, wird berechnet. Ergebnisse sind die Kantenliste des Weges und dessen Laenge. Wird kein Weg gefunden, so wird der Fehlercode KEIN_WEG_GEFUNDEN zurueckgegeben. Zur Umwegberechnung wird eine Heuristik verwendet.
`Fehlercode berechne_p(const list<Person_ID> &teilnehmerliste, const Person_ID fahrer,double &wert);` Die Methode berechnet die Bewertung p bezueglich der belegten Plaetze im Auto des Fahrers. Sind alle Plaetze belegt, so wird wert=1.0 zurueckgegeben, sind nicht alle Plaetze belegt, so wird ein Wert zwischen 0.0 und 1.0 zurueckgegebn.
`Fehlercode berechne_na(const list<Person_ID> &teilnehmerliste, double &na);` Die Methode berechnet die Bewertung na bezueglich der Abneigungen zu Personen. Wenn mehr Abneigungskonflikte bestehen, als in na_max angegeben, so wird der Fehlercode ZU_VIELE_ABNEIGUNGEN zurueckgegeben. Wenn keine Abneigungen zwischen den Personen der FGM bestehen wird na=0.0 als Ergebnis zurueckgegeben. Wenn Abneigungen bestehen, dann ergibt sich als Ergebnis 0.0 ¡ na ¡= 1.0.
`Fehlercode berechne_ne(const list<Person_ID> &teilnehmerliste, double &ne);`

Die Methode berechnet die Bewertung ne bezueglich der Zuneigungen zu Personen. Wenn keine Zuneigungen zwischen den Personen der FGM bestehen wird na=0.0 als Ergebnis zurueckgegeben. Wenn Zuneigungen bestehen, dann ergibt sich als Ergebnis 0.0 ¡ ne ¡= 1.0.
`Fehlercode berechne_u(const list<Kanten_ID> weg, const Person_ID fahrer, double &u);` Die Methode berechnet die Bewertung u bezueglich des Umwegs einer FGM. Ist der zu bewertende Weg kein gueltiger Weg oder ist der Umweg groesser als u_max, so wird der Fehlercode KEIN_WEG_GEFUNDEN zurueckgegebn. Der wert (=Ergebnis der Bewertung) liegt zuwischen 0.0 und 1.0. Wenn der Fahrer keinen Umweg machen muss, so wird wert=0.0, ist der Umweg = u_max so ergibt sich wert=1.0 und ist der Umweg zwischen 0 und u_max, so ergibt sich 0.0 ¡ wert ¡ 1.0. Je kleiner wert ist, desto besser ist die Bewertung bezueglich des Umweges.
`Fehlercode berechne_e(const list<Person_ID> &teilnehmerliste,double &e);` Die Methode berechnet die Bewertung e bezueglich der Eigenschaften und Wuensche der Personen. Wenn ein Wunsch mit Gewicht 10 verletzt ist, wird ein Fehlercode ungleich OK zurueckgegeben, ansonsten ein Wert zwischen 0.0 und 1.0.

Attribute
`string name;` der Name der Bewertungsfunktion
`Bewertungs_ID id;` die ID der Bewertungsfunktion
`Bewertungsverwalter *bewertungsverwalter_zeiger;` Zeiger auf den Bewertungsverwalter.
`Boolean attribute_geaendert;` FGM wurde seit dem letzten Speichern verändert.
`int u_max;` maximaler Umweg einer Fahrgemeinschaft in Kilometern
`int z_max;` maximaler Abstand zwischen den Arbeitszeiten in Minuten
`int na_max;` maximale Anzahl der Abneigungskonflikte pro FGM
`double e_max;` gibt an wie schlecht eine FGM bezueglich der Bewertung der Personeneigenschaften sein darf; Wert zwischen 0 und 1; 0 entspricht einer optimalen FGM (keine Konflikte), 0,99 z.B. entspricht einer sehr schlechten FGM
`int p_max;` maximale Anzahl der Personen pro FGM
`int gewicht_z;` Gewichtung der Bewertung bezueglich Z
`int gewicht_u;` Gewichtung der Bewertung bezueglich U
`int gewicht_na;` Gewichtung der Bewertung bezueglich Na

int gewicht_ne; Gewichtung der Bewertung bezueglich Ne
int gewicht_p; Gewichtung der Bewertung bezueglich P
int gewicht_e; Gewichtung der Bewertung bezueglich E
Boolean heuristik_verwenden; gibt an, ob zur Umwegberechnung die Heuristik oder ein optimaler Algorithmus verwendet werden soll

5.5.6 Wegsuche

Verkehrsgraph

öffentliche Methoden
Verkehrsgraph(); Konstruktor
Verkehrsgraph(); Destruktor
Fehlercode load_graph(const char* nodefile, const char* edgefile); Verkehrsgraph von Platte laden (Knoten- und Kantenfile). Fehler DATEI_NICHT_GEFUNDEN, falls Datei nicht geladen werden konnte.
Fehlercode load_regions(const char* regionfile); Regionenmarkierung der Knoten von Platte laden. Fehler DATEI_NICHT_GEFUNDEN, falls Datei nicht geladen werden konnte.
Fehlercode lade_verkehrsgraph(const char* nodefile, const char* edgefile, const char* regionfile); Knoten-, Kanten- und Regionenfile laden. Fehler DATEI_NICHT_GEFUNDEN, falls Datei nicht geladen werden konnte.
int read_max_node_id() const; Methode gibt größte Knoten-ID zurück.
int read_max_region_id() const; Methode gibt größte Region-ID zurück.
int read_cost(edge e) const; Methode gibt Gewicht der Kante e zurück.
int read_dist(node n) const; Methode gibt Entfernung des Knotens n vom Startknoten der Wegsuche zurück
int read_heur_dist(node n) const; Methode gibt Beschriftung heur_dist (bisherige Kosten + Restkosten bei A*-Suche) des Knotens n zurück.
edge read_pred(node n) const; Methode gibt Vorgängerkante des Knotens n im Baum der kürzesten Wege zurück.
int read_xpos(node n) const; Methode gibt x-Koordinate des Knotens n in Metern zurück.

int read_ypos(node n) const; Methode gibt y-Koordinate des Knotens n in Metern zurück.
int read_node_id(node n) const; Methode gibt ID des Knotens n zurück.
int read_edge_id(edge e) const; Methode gibt ID der Kante e zurück.
bool get_node(int i, node& n); Methode gibt Knotenobjekt mit der ID i zurück. Rückgabewert false, falls ID nicht existiert.
bool get_edge(int i, edge& e); Methode gibt Kantenobjekt mit der ID i zurück. Rückgabewert false, falls ID nicht existiert.
bool read_edge_name(int i, string& name); Methode gibt Straßenname zu einer Kanten-ID leerer String, falls die Kante keinen Namen hat Rückgabewert false, falls ID nicht existiert.
list<int> read_edges_by_name(string name); Methode gibt Kanten-ID-Liste zu einem Straßennamen leere Liste, falls Straßenname im Verkehrsgraphen nicht existiert
int read_region(node n); Methode gibt Regionnummer des Knotens n zurück.
bool read_startlist(int i, list<int>& sl); Methode gibt zur Knoten-ID i die Liste der Entfernungen zu den zugehörigen Randknoten (im Randknotenmodell) bzw. zu den nächsten Rathäusern (im Rathausmodell) zurück. Rückgabewert false, falls Knoten-ID nicht existiert.
bool read_ziellist(int i, list<int>& zl); Methode gibt zu Knoten-ID i die Liste der Entfernungen von den zugehörigen Randknoten zum Knoten i bzw. von den nächsten Rathäusern zurück. Rückgabewert false, falls Knoten-ID nicht existiert.
bool read_startrathaus(int i, list<int>& sr); Methode gibt (im Rathausmodell) zum Knoten i die Knoten-ID-Liste der benachbarten Startrathäuser zurück. Rückgabewert false, falls Knoten-ID nicht existiert.
bool read_zielrathaus(int i, list<int>& zr); Methode gibt (im Rathausmodell) zum Knoten i die Knoten-ID-Liste der benachbarten Zielrathäuser zurück. Rückgabewert false, falls Knoten-ID nicht existiert.
int read_klasse(edge e); Methode gibt Straßenklasse der Kante e zurück.
bool read_target_node(int edge_id, int& node_id); Methode gibt Zielknoten-ID zu Kanten-ID zurück. Rückgabewert false, falls Kanten-ID nicht existiert.
bool sind_rathaustabellen_geladen(); Methode gibt true zurück, falls Rathausmodell aktiv.
bool sind_randknotentabellen_geladen(); Methode gibt true zurück, falls Randknotenmodell aktiv. Entweder ist das Rathaus- oder das Randknotenmodell aktiv.
bool kanten_sind_benachbart(Kanten_ID id1, Kanten_ID id2, int entf);

Methode gibt true zurück, falls die Zielknoten der Kanten id1 und id2 höchstens die Entfernung entf haben.
`void gib_zufaellige_kante(Kanten_ID& id);` Methode gibt eine zufällige Kanten-ID zurück.
`void write_cost(edge e, int i);` Methode setzt Kantengewicht der Kante e
`void write_dist(node n, int i);` Methode setzt Entfernungsbeschriftung des Knotens n.
`void write_heur_dist(node n, int i);` Methode setzt heuristische Entfernung des Knotens n
`void write_pred(node n, edge e);` Methode setzt Vorgängerkante des Knotens n im Baum der kürzesten Wege
`bool write_region(int i, int regionnr);` Methode setzt Regionnummer des Knotens i. Rückgabewert false, falls Knoten-ID nicht existiert.
`bool write_startlist(int i, list<int> sl);` Methode setzt Startliste des Knotens i. Rückgabewert false, falls Knoten-ID nicht existiert.
`bool write_ziellist(int i, list<int> zl);` Methode setzt Zielliste des Knotens i. Rückgabewert false, falls Knoten-ID nicht existiert.
`bool write_startrathaus(int i, list<int> sr);` Methode setzt Liste der Startrathäuser des Knotens i. Rückgabewert false, falls Knoten-ID nicht existiert.
`bool write_zielrathaus(int i, list<int> zr);` Methode setzt Liste der Zielrathäuser des Knotens i. Rückgabewert false, falls Knoten-ID nicht exisitiert.
`void write_klasse(edge e, int i);` Methode setzt Straßenklasse der Kante e.
`void setze_rathaustabellen_geladen();` Methode macht Rathausmodell aktiv.
`void setze_randknotentabellen_geladen();` Methode macht Randknotenmodell aktiv.
`GRAPH<int,int> G;` Dem Verkehrsgraph zugrundeliegender LEDA-Graph (wird für LEDA-Iterationsmakros benötigt)

Attribute
`edge_map<int> cost;` Abbildung Kante auf Kantengewicht
`edge_map<int> klasse;` Abbildung Kante auf Straßenklasse
`node_map<int> dist;` Abbildung Knoten auf Entfernung zum Startknoten
`node_map<int> heur_dist;` Abbildung Knoten auf heuristische Entfernung (A*-Heuristik)
`node_map<edge> pred;` Abbildung Knoten auf Vorgängerkante im Baum der kürzesten Wege
`node_map<int> xpos;` Abbildung Knoten auf x-Koordinate
`node_map<int> ypos;` Abbildung Knoten auf y-Koordinate
`node_map<int> region;` Abbildung Knoten auf Regionnummer des Knotens
`node_map< list<int> > startlist;` Abbildung Knoten auf Startliste (Entfernungsliste zu Randknoten/Rathäusern)
`node_map< list<int> > ziellist;` Abbildung Knoten auf Zielliste
`node_map< list<int> > startrathaus;` Abbildung Knoten auf Liste der Startrathäuser
`node_map< list<int> > zielrathaus;` Abbildung Knoten auf Liste der Zielrathäuser
`dictionary<int,node> node_no;` Abbildung Knoten-ID auf Knotenobjekt zu dieser ID
`dictionary<int,edge> edge_no;` Abbildung Kanten-ID auf Kantenobjekt zu dieser ID
`dictionary<int,string> edge_name;` Abbildung Kanten-ID auf Straßenname
`dictionary< string,list<int> > edges_by_name;` Abbildung Straßenname auf Liste von Kanten-IDs
`int max_node_id;` größte Knoten-ID
`int max_region_id;` größte Regionnummer
`bool rathaustabellen_geladen;` Flag für Rathausmodell
`bool randknotentabellen_geladen;` Flag für Randknotenmodell

private Methoden
`void WandleUmlaute(unsigned char* str);` Methode wandelt Umlaute in den Straßennamen
`void insert_edges_by_name(string name, int e1, int e2 = -1);` Methode fügt die Kanten-IDs e1 und e2 in die Kantenliste zum Straßennamen name ein. e2 kann weggelassen werden.

Wegsuche_algorithmus

öffentliche Methoden
`Wegsuche_algorithmus(Verkehrsgraph& v, string name, int id);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll sowie Algorithmusname und Algorithmus-ID.

`virtual bool setze_parameter(int src, int tgt);` Methode setzt Start- und Zielknoten für die Wegsuche. Rückgabewert false, falls Start- oder Zielknoten nicht existiert.
`virtual list<int> suche(int& fahrzeit) = 0;` Abstrakte Methode zum Starten der Wegsuche, wird in jeder Unterklasse neu definiert. Auf jeden Fall wird Fahrtzeit zurückgegeben, abhängig vom Algorithmus auch eine Kantenliste.
`Fehlercode gib_algo_name(string& name);` Methode gibt Algorithmusnamen zurück
`Fehlercode gib_algo_id(int& id);` Methode gibt Algorithmen-ID zurück

Attribute
`Verkehrsgraph& vg;` der Wegsuche zugrundeliegender Verkehrsgraph
`node s;` Startknoten bei der Wegsuche
`node t;` Zielknoten
`string algo_name;` Algorithmusname
`int algo_id;` Algorithmus-ID

Dijkstra

öffentliche Methoden
`Dijkstra(Verkehrsgraph& v, string name, int id);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll sowie den Algorithmusnamen und Algorithmus-ID.
`list<int> suche(int& fahrzeit);` Methode startet Wegsuche (Start s und Ziel t in Oberklasse). Zurückgegeben wird Fahrtzeit und Liste von Kanten-IDs (Weg).

Dijkstra_astar

öffentliche Methoden
`Dijkstra_astar(Verkehrsgraph& v, string name, int id);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll sowie den Algorithmusnamen und Algorithmus-ID.
`setze_restfaktor(double restfaktor);` Methode setzt Faktor für Gewichtung der Restentfernung bei A*-Heuristik.
`list<int> suche(int& fahrzeit);` Methode startet Wegsuche (Start s und Ziel t in Oberklasse). Zurückgegeben wird Fahrtzeit und Liste von Kanten-IDs (Weg).

Attribute
`double rf;` Faktor für die Restentfernung.

private Methoden
`double euclid_dist(node u, node v);` Funktion zur Berechnung des euklid. Abstands.

Kuerzeste_Wege

öffentliche Methoden
`Kuerzeste_Wege(Verkehrsgraph& v, string name, int id);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll sowie den Algorithmusnamen und Algorithmus-ID.
`Fehlercode files_oeffnen();` Methode zur Initialisierung und Öffnen der Files für den Tabellenzugriff. Fehlercode DATEI_NICHT_GEFUNDEN, falls eine Datei fehlt.
`list<int> suche(int& fahrtzeit);` Fahrtzeit in Tabelle auf Festplatte nachschauen. Als Kantenliste wird leere Liste zurückgegeben. Fahrtzeit = MAXDIST, falls kein Weg zwischen Start und Ziel existiert.

Attribute
`FILE* fp_index;` Filepointer auf Indextabelle
`FILE* fp_tab;` Filepointer auf Tabelle der kürzesten Entfernungen
`int node_number;` Anzahl der Knoten im Verkehrsgraph
`int start_id;` Startknoten-ID
`int ziel_id;` Zielknoten-ID

private Methoden
`int read_index(int node_id);` Hilfsfunktion zum Nachschlagen der Knotenadresse. Knotennummern sind nicht fortlaufend, deswegen muß die Knoten-ID in einen Index umgewandelt werden (Indextabelle).

Wegsuche_Rand

öffentliche Methoden
`Wegsuche_Rand(Verkehrsgraph& v, string name, int id, Dijkstra& d, Level2Graph_Rand& v2);`

Konstruktor, erhält als Parameter den Level-1-Verkehrsgraphen, auf dem gerechnet werden soll sowie Algorithmusname und Algorithmus-ID. Für die Wegsuche im Randknotenmodell wird außerdem der Dijkstra-Algorithmus und Level2Graph_Rand benötigt.
`list<int> suche(int& fahrtzeit);` Methode startet Wegsuche (Start s und Ziel t in Oberklasse). Zurückgegeben wird Fahrtzeit und leere Kanten-ID-Liste.

Attribute
`Dijkstra& dijk;` Dijkstra-Algorithmus für Wegsuche, falls Start- und Zielknoten in der gleichen Region
`Level2Graph_Rand& vg2;` Level-2-Graph beim Randknotenmodell

Wegsuche_Rathaus

öffentliche Methoden
`Wegsuche_Rathaus(Verkehrsgraph& v, string name, int id, Level2Graph_Rathaus& v2);` Konstruktor, erhält als Parameter den Level-1-Verkehrgraphen, auf dem gerechnet werden soll sowie Algorithmusnamen und Algorithmus-ID. Für die Wegsuche im Rathausmodell wird Level2Graph_Rathaus benötigt.
`list<int> suche(int& fahrtzeit);` Methode startet Wegsuche (Start s und Ziel t in Oberklasse). Zurückgegeben wird Fahrtzeit und leere Kanten-ID-Liste.

Attribute
`Level2Graph_Rand& vg2;` Level-2-Graph beim Rathausmodell

Level2Graph

öffentliche Methoden
`Level2Graph(Verkehrsgraph& v);` Konstruktor, erhält als Parameter den zugehörigen Level-1-Verkehrsgraphen.
`bool get_node(int i, node& n);` Methode gibt Knotenobjekt mit der ID i zurück. Rückgabewert false, falls Knoten nicht existiert.
`int read_node_id(node n);` Methode gibt ID eines Knotenobjekts zurück.
`bool read_cost2(int node_id1, int node_id2, int& c2);` Methode gibt Entfernung auf Level 2 zwischen Knoten 1 und 2 in c2 zurück. Rückgabewert false, falls einer der Knoten nicht existiert.
`bool write_cost2(int node_id1, int node_id2, int c2);`

Methode setzt Entfernung auf Level 2 zwischen Knoten 1 und 2 auf c2. Rückgabewert false, falls einer der der Knoten nicht existiert.

Attribute
`Verkehrsgraph& vg;` zugrundeliegender Level-1-Verkehrsgraph
`GRAPH<int,int> G2;` Level-2-Graph
`node_matrix<int> cost2;` Matrix der Knotenentfernungen auf Level 2
`dictionary<int,node> node_no;` Abbildung Knoten-ID auf Knotenobjekt

Level2Graph_Rand

öffentliche Methoden
`Level2Graph_Rand(Verkehrsgraph& v, Dijkstra_notarget& d, Dijk_manytargets& dm);` Konstruktor, erhält als Parameter den zugrundeliegenden Level-1-Verkehrsgraphen und zum Aufbau von Level 2 die Algorithmen Dijkstra_notarget und Dijk_manytargets.
`void aufbauen();` Aufbau des Level-2-Graphen (Randknotenmodell). Bestimmung der Randknoten, Best. der Randknotenentfernungen und Bestimmung der Start- und Ziellisten.
`void speichern(const char* entf_randknotenfile, const char* start_ziellistfile);` Level-2-Graph in zwei Files speichern: Randknotenentfernungen und Start-/Ziellisten.
`Fehlercode laden(const char* entf_randknotenfile, const char* start_ziellistfile);` Level-2-Graph laden
`int read_max_region_id();` Methode gibt größte Regionnummer zurück.
`bool read_randknoten(int region_id, list<int>& rk);` Methode gibt Liste der Randknoten-IDs zu einer Regionennummer zurück. Rückgabewert false, falls Regionennummer nicht existiert.

Attribute
`Dijkstra_notarget& dijk;` Dijkstra-Algorithmus ohne Zielknoten für Bestimmung der Randknotenentfernungen
`Dijk_manytargets& dijkmany;` Dijkstra-Algorithmus mit mehreren Zielen für Bestimmung der Start- und Ziellisten
`int max_region_id;`

größte Regionennummer
`dictionary<int, list<int> > randknoten;` Abbildung Regionennummer auf Liste der zugehörigen Randknoten

private Methoden
`bool in_randknotenliste(int u_id, int region);` Methode prüft, ob Knoten in Randknotenliste einer best. Region vorhanden. Rückgabewert true, falls ja.
`void bestimme_randknoten();` Bestimmung der Randknoten beim Aufbau des Level-2-Graphen
`void bestimme_entf_rand();` Bestimmung der Randknotenentfernungen beim Aufbau des Level-2-Graphen
`void bestimme_start_ziel();` Bestimmung der Start-/Ziellisten beim Aufbau des Level-2-Graphen
`void randknoten_eintragen(int v_node_id, int u_region, int v_region);` Methode trägt Knoten in die Randknotenlisten zweier Regionen ein.

Level2Graph_Rathaus

öffentliche Methoden
`Level2Graph_Rathaus(Verkehrsgraph& v, Dijkstra_notarget& d, Dijkstra_first_k_targets& df);` Konstruktor, erhält als Parameter den zugrundeliegenden Level-1-Verkehrsgraphen und zum Aufbau von Level 2 die Algorithmen Dijkstra_notarget und Dijk_first_k_targets.
`void aufbauen();` Aufbau des Level-2-Graphen (Rathausmodell). Bestimmung der Rathaeuser, Best. der Rathausentfernungen und Bestimmung der Start- und Ziellisten.
`void speichern(const char* entf_rathaeuserfile, const char* start_ziellistfile);` Level-2-Graph in zwei Files speichern: Rathäuserentfernungen und Start-/Ziellisten.
`Fehlercode laden(const char* entf_rathaeuserfile, const char* start_ziellistfile);` Level-2-Graph laden

Attribute
`Dijkstra_notarget& dijk;` Dijkstra-Algorithmus ohne Zielknoten für Bestimmung der Rathausentfernungen
`Dijk_manytargets& dijkmany;` Dijkstra-Algorithmus für die nächsten k Nachbarn zur Bestimmung der Start- und Ziellisten
`int rathaeuser_in_zelle[100][100];` Rathäuseranzahl in den Gitterzellen

`int xmin, ymin, xmax, ymax;` min. und max. Knotenkoordinaten

private Methoden
`bool zelle_nicht_voll(int xpos, int ypos);` Methode überprüft, ob Rathaus mit Koordinaten xpos, ypos noch in zugehörige Gitterzelle passt.
`bool zelle_leer(int xpos, int ypos);` Methode überprüft, ob die zu den Koordinaten xpos, ypos zugehörige Gitterzelle leer ist.
`void bestimme_rathaeuser();` Bestimmung der Rathäuser beim Aufbau des Level-2-Graphen
`void bestimme_entf_rathaeuser();` Bestimmung der Rathausentfernungen beim Aufbau des Level-2-Graphen
`void bestimme_start_ziel();` Bestimmung der Start-/Ziellisten beim Aufbau des Level-2-Graphen
`void liste_abspeichern(FILE* fp, list<int> l);` Methode speichert Liste auf Platte ab.
`void liste_laden(FILE* fp, list<int>& l);` Methode lädt Liste von Platte.

Dijkstra_notarget

öffentliche Methoden
`Dijkstra_notarget(Verkehrsgraph& v);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll.
`Dijkstra_notarget();` Destruktor.
`setze_parameter(int src);` Methode setzt Startknoten für die Suche.
`void suche();` Methode startet Wegsuche vom Startknoten aus, kein Zielknoten (komplettes Durchsuchen des Graphen). Berechnete Entfernungen und Wege stehen nachher im Verkehrsgraphen.

Attribute
`Verkehrsgraph& vg;` Verkehrsgraph, auf dem gerechnet wird.
`node s;` Startknoten

Dijk_manytargets

öffentliche Methoden
`Dijk_manytargets(Verkehrsgraph& v);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll.
`Dijk_manytargets();` Destruktor
`void setze_parameter(int src, list<int> targets);` Methode setzt Startknoten-ID und Liste der Zielknoten-IDs, zu denen die Entfernung vom Startknoten aus berechnet werden soll.
`list<int> suche();` Methode starte Wegsuche zu den Zielen in der Zielliste.

Attribute
`Verkehrsgraph& vg;` Verkehrsgraph, auf dem gerechnet wird.
`node s;` Startknoten.
`set<node> targetset;` Menge der Zielknoten.
`list<int> target_id_list;` Liste der Zielknoten-IDs.

Dijk_first_k_targets

öffentliche Methoden
`Dijk_first_k_targets(Verkehrsgraph& v);` Konstruktor, erhält als Parameter den Verkehrsgraphen, auf dem gerechnet werden soll.
`Dijk_first_k_targets();` Destruktor
`void setze_parameter(int src, list<int> targets, int kk);` Methode setzt Startknoten-ID, Liste der Zielknoten-IDs und Anzahl kk der Ziele, zu denen die Entfernung vom Startknoten aus berechnet werden soll.
`void suche(list<int>& nachbarn, list<int>& entf);` Methode starte Wegsuche zu den k nächsten Zielen, nachbarn enthält danach die Knoten-ID-Liste der k nächsten Ziele, entf die zugehörige Entfernungsliste.

Attribute
`Verkehrsgraph& vg;` Verkehrsgraph, auf dem gerechnet wird.
`node s;` Startknoten.
`set<node> targetset;` Menge der Zielknoten.
`int k;`

Anzahl der Nachbarn.

Pipe2GraphDraw

öffentliche Methoden
`Pipe2GraphDraw(int mobi2graph_pipe[2], int graph2mobi_pipe[2]);` Konstruktor, erhält jeweils zwei Filedeskriptoren für Pipeline in Richtung GraphDraw und Pipeline von GraphDraw.
`Fehlercode weg_anzeigen(list<int> kantenliste, Kanten_ID start, Kanten_ID ziel);` Methode schickt Kantenliste an GraphDraw und zeigt Weg in Stadtplan an. Start- und Zielkante müssen auch angegeben werden.
`Fehlercode kantenliste_anzeigen(list<int> kantenliste);` Methode schickt Kantenliste an GraphDraw, die dann angezeigt wird. Gleiche Funktion wie weg_anzeigen, aber ohne Start- und Zielkante.
`Fehlercode strasse_anzeigen(string strassenname);` Methode schickt Straßennamen an GraphDraw und zeigt Straße in Stadtplan an. Substringsuche, d.h. es werden alle Straßen angezeigt, die in ihrem Namen den Suchstring als Substring enthalten. Groß-/Kleinschreibung beim Suchstring wird ignoriert.
`Fehlercode eine_kante_auswaehlen(Kanten_ID& kanten_id);` Methode holt von GraphDraw per Mausklick ausgewählte Kanten-ID.
`Fehlercode zwei_kanten_auswaehlen(Kanten_ID& start_id, Kanten_ID& ziel_id);` Methode holt von GraphDraw zwei per Mausklick ausgewählte Kanten-IDs. Zuerst Auswahl der Start-, dann der Zielkante.

Attribute
`int to_graph_pipe[2];` Filedeskriptoren für Pipe Richtung GraphDraw, 0: lesen, 1: schreiben.
`int to_mobi_pipe[2];` Filedeskriptoren für Pipe von GraphDraw, 0: lesen, 1: schreiben.
`char inbuf[100];` Puffer für von GraphDraw empfangene Strings.
`char msg[100];` Puffer für an GraphDraw gesendete Strings.

private Methoden
`void receive_string(char* inbuf);` Methode empfängt String von GraphDraw bis zum Trennzeichen "." und gibt String ohne "." in inbuf zurück.
`void send_string(char* s);` Methode sendet String an GraphDraw.

5.5.7 Fürsorger und Lader/Speicherer

Fuersorger

öffentliche Methoden
`Fuersorger();`
`Fuersorger(Einteilungsverwalter* einteilungen,Personenverwalter* personen,Algorithmenverwalter* algorithmen,Bewertungsverwalter* bewertungen,Menueverwalter* menues,Lader_speicherer* dateien, Verkehrsgraph* graph);` Konstruktor, der den Attributen des Objekts die entsprechenden Zeiger zuweist.
`void setze_personenverwalter(Personenverwalter* personen);`
`void setze_einteilungsverwalter(Einteilungsverwalter* einteilungen);`
`void setze_algorithmenverwalter(Algorithmenverwalter* algorithmen);`
`void setze_bewertungsverwalter(Bewertungsverwalter* bewertungen);`
`void setze_menueverwalter(Menueverwalter* menues);`
`void setze_verkehrsgraph(Verkehrsgraph* graph);`
`void setze_lader_speicherer(Lader_speicherer* dateien);`
`Fehlercode neue_personendatei();` Legt eine neue Personendatei mit einem von der Datei_auswahl zu wählenden Namen an.
`Fehlercode lade_personendatei();` Lädt eine Personendatei und wirft die alte weg.
`Fehlercode speichere_personendatei();` Speichert die Personendatei unter ihrem aktuellen Namen.
`Fehlercode speichere_personendatei_unter();` Speichert die Personendatei unter einem neuen Namen und nimmt diesen dann als aktuellen Namen.
`Fehlercode schliesse_personendatei();` Schließt die aktuelle Personendatei – es kann wieder eine neue angelegt oder geladen werden.
`Fehlercode importiere_personendaten();` Aus einer zweiten Personendatei werden die Daten zusätzlich zu den aktuellen geladen, es wird auf doppeltes Vorkommen getestet.
`Fehlercode lade_bewertungsfunktion();` Lädt eine neue Bewertungsfunktion, auszuwählen durch die Datei_auswahl.
`Fehlercode speichere_bewertungsfunktion();`

Speichert die aktuelle Bewertungsfunktion.
`Fehlercode speichere_bewertungsfunktion_unter();` Speichert die aktuelle Bewertungsfunktion unter einem neuen Namen und aktualisiert die aktuelle Bewertungsfunktion, sie steht dann auf der mit dem neuen Namen.
`Fehlercode neue_person(const Personendaten person);` Fügt eine neue Person ein.
`Fehlercode aendere_person(const Person_ID per_id,const Personendaten person);` Ändert die Daten der Person mit id.
`Fehlercode loesche_person(const Person_ID per_id);` Löscht die Person mit id aus den Personendaten.
`Fehlercode suche_personen(const Per_Filterdaten filter,list<Person_ID> &id);` Gibt eine Liste von Personen zurück, die die Filterkriterien erfüllen.
`Fehlercode existiert_person(const string per_name,const Datum geb_datum,Person_ID &per_id);` Existiert die Person mit name und geb_datum, dann wird die entsprechende Personen_id zurückgegeben, ansonsten der Wert 0.
`Fehlercode gib_personen_liste(list<Person_ID> &id);` Gibt eine Liste von Personenid's zurück.
`Fehlercode gib_personendaten(const Person_ID per_id,Personendaten &person);` Gibt die Daten der Person mit id zurück.
`Fehlercode generiere_personen(const GenerierDaten gendat);` Generiert eine Anzahl von Personen.
`Fehlercode gib_geschlecht(const Person_ID per_id,Geschlecht &geschlecht);` Gibt die Eigenschaft Geschlecht der Person mit id zurück.
`Fehlercode gib_raucher(const Person_ID per_id,Raucher &raucher);` Gibt die Eigenschaft Raucher der Person mit id zurück.
`Fehlercode gib_musikgeschmack(const Person_ID per_id,list<string> &musikgeschmack);` Gibt eine Liste zurück, in der alle Musikrichtungen der Person mit id enthalten sind.
`Fehlercode gib_fahrer(const Person_ID per_id,Boolean &fahrer);` Gibt die Eigenschaft Fahrer der Person mit id zurück.
`Fehlercode gib_komfortklasse(const Person_ID per_id,Komfortklasse &komfortklasse);` Gibt die Eigenschaft Komfortklasse der Person mit id zurück.
`Fehlercode gib_anzahl_der_plaetze(const Person_ID per_id,int &anzahl_der_plaetze);` Gibt die Eigenschaft Anzahl der Plätze der Person mit id zurück.
`Fehlercode neue_einteilung(const string name,Einteilungs_ID &ein_id);` Fügt eine neue Einteilung mit name in das System ein und gibt die zugeornete ein_id zurück.
`Fehlercode dupliziere_einteilung(const string name);`

Erzeugt eine Kopie der aktuellen Einteilung mit dem Namen name.
Fehlercode loesche_einteilung(const Einteilungs_ID ein_id); Löscht die Einteilung mit der ID ein_id.
Fehlercode setze_aktuelle_einteilung(const Einteilungs_ID ein_id); Setzt die aktuelle Einteilung auf die Einteilung mit ein_id.
Fehlercode gib_aktuelle_einteilung(Einteilungs_ID &ein_id); Gibt die aktuelle ein_id zurück.
Fehlercode gib_id_einteilung(list <Einteilungs_ID> &ein_id); Gibt eine Liste mit allen Einteilungs-IDs zurück.
Fehlercode neue_fgm(FGM_ID &fgm_id); Fügt eine neue Fahrgemeinschaft in die aktuelle Einteilung ein und gibt die zugeordnete fgm_id zurück.
Fehlercode loesche_fgm(const FGM_ID fgm_id); Löscht die Fahrgemeinschaft mit id der aktuellen Einteilung.
Fehlercode suche_fgm(const FGM_filter filter,list<FGM_ID> &fgm_id); Gibt eine Liste von Fahrgemeinschaften zurück, die die Filterkriterien erfüllen.
Fehlercode setze_name_einteilung(const string ein_name); Setzt den Namen der aktuellen Einteilung auf ein_name.
Fehlercode setze_bew_fkt_id(Bewertungs_ID bew_id); Setzt die ID der Bewertungsfunktion mit der diese Einteilung berechnet wurde auf bew_id.
Fehlercode gib_name_einteilung(string &ein_name); Gibt den Namen der aktuellen Einteilung zurück.
Fehlercode gib_bew_fkt_id(Bewertungs_ID &bew_id); Gibt die ID der Bewertungsfunktion, mit der diese Einteilung berechnet wurde, zurück.
Fehlercode gib_id_fgm(list<FGM_ID> &fgm_id); Gibt eine Liste mit allen Fahrgemeinschafts-IDs der aktuellen Einteilung zurück.
Fehlercode gib_vermittelte_personen(list<Person_ID> &per_id); Gibt eine Liste mit allen Personen-IDs zurück, die vermittelt wurden.
Fehlercode gib_noch_nicht_vermittelte_personen(list<Person_ID> &per_id); Gibt eine Liste mit allen Personen-IDs zurück, die noch nicht vermittelt wurden.
Fehlercode gib_markierte_fgm(list<FGM_ID> &fgm_id); Gibt eine Liste mit Fahrgemeinschafts-IDs zurück, die markiert sind.
Fehlercode gib_unmarkierte_fgm(list<FGM_ID> &fgm_id); Gibt eine Liste mit Fahrgemeinschafts-IDs zurück, die unmarkiert sind.
Fehlercode pruefe_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze); Gibt die Anzahl der freien Plaetze in der Fahrgemeinschaft fgm_id zurueck
Fehlercode neuer_fahrer(const FGM_ID fgm_id,const Person_ID per_id);

Fügt den Fahrer der FGM ein, diese Methode muß vor neuer_teilnehmer aufgerufen werden.
Fehlercode neuer_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id); Fügt Person mit per_id in die Fahrgemeinschaft mit fgm_id ein.
Fehlercode loesche_teilnehmer_aus_fgm(const FGM_ID fgm_id,const Person_ID per_id); Löscht die Person mit per_id aus der Fahrgemeinschaft mit fgm_id.
Fehlercode fest_eintragen(const FGM_ID fgm_id,const Person_ID per_id); Trägt die Person mit per_id in die Fahrgemeinschaft mit fgm_id fest ein.
Fehlercode markiere_fgm(const FGM_ID fgm_id); Markiert die Fahrgemeinschaft mit fgm_id.
Fehlercode unmarkiere_fgm(const FGM_ID fgm_id); Macht die Markierung der Fahrgemeinschaft mit fgm_id rückgängig.
Fehlercode ist_fgm_markiert(FGM_ID fgm_id,Boolean &flag); Prüft, ob die FGM mit fgm_id markiert ist, flag wird auf den Wert gesetzt.
Fehlercode setze_status_teilnehmer(const FGM_ID fgm_id,const Person_ID per_id,const Status status); Setzt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_fahrer_fgm(const FGM_ID fgm_id,const Person_ID per_id); Setzt den Fahrer der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_freie_plaetze_fgm(const FGM_ID fgm_id,const int plaetze); Setzt die Anzahl der freien Plätze der Fahrgemeinschaft mit fgm_id.
Fehlercode setze_fahrtroute_fgm(const FGM_ID fgm_id,const list<Kanten_ID> kanten_id); Setzt die Fahrtroute der Fahrgemeinschaft mit fgm_id. Die Fahrroute besteht aus einer Liste mit Kanten-IDs mit allen Start- und Zielorten der beteiligten Personen, in der Reihenfolge, wie gefahren wird.
Fehlercode setze_fgmdaten(const FGM_ID fgm_id,const FGMdaten fgm); Setzt die Fahrgemeinschaftsdaten.
Fehlercode gib_fgmdaten(const FGM_ID fgm_id,FGMdaten &fgm); Gibt die Fahrgemeinschaftsdaten zurück.
Fehlercode gib_status_teilnehmer(const Person_ID per_id,const FGM_ID fgm_id,Status &status); Gibt den Status der Person mit per_id der Fahrgemeinschaft mit fgm_id zurück.
Fehlercode gib_fahrer_fgm(const FGM_ID fgm_id,Person_ID &per_id); Gibt den Fahrer der Fahrgemeinschaft mit fgm_id zurück.
Fehlercode gib_freie_plaetze_fgm(const FGM_ID fgm_id,int &plaetze); Gibt die Anzahl der freien Plätze der Fahrgemeinschaft mit fgm_id zurück.
Fehlercode gib_fahrtroute_fgm(const FGM_ID fgm_id, list<Kanten_ID> &kanten_id);

Gibt die Fahrtroute der Fahrgemeinschaft mit fgm_id zurück.
`Fehlercode setze_zeitplan_fgm(const FGM_ID fgm_id,const list<Zeit*> zeiten);` Setzt den Zeitplan der Fahrgemeinschaft mit fgm_id. Der Zeitplan enthält die Abfahrts- und Ankunfszeiten an allen Start- und Zielorten in der Reihenfolge, wie sie in der Fahrtroute angegeben sind.
`Fehlercode gibt_zeitplan_fgm(const FGM_ID fgm_id,list<Zeit*> &zeiten);` Gibt den Zeitplan der Fahrgemeinschaft mit fgm_id zurück.
`Fehlercode gib_startort_fgm(const FGM_ID fgm_id,Kanten_ID &kanten_id);` Gibt den Startort (kanten_id) der Fahrgemeinschaft fgm_id zurück.
`Fehlercode gib_zielort_fgm(const FGM_ID fgm_id,Kanten_ID &kanten_id);` Gibt den Zielort (kanten_id) der Fahrgemeinschaft fgm_id zurück.
`Fehlercode gib_entstehungsdatum_fgm(FGM_ID fgm_id,FGM_Datum &entstehungsdatum);` Gibt das Entstehungsdatum der FGM mit fgm_id zurück.
`Fehlercode gib_letzte_aenderung_fgm(FGM_ID fgm_id,FGM_Datum &letzte_aenderung);` Gibt das Datum der letzten Änderung der FGM mit fgm_id zurück.
`Fehlercode gib_teilnehmer_fgm(const FGM_ID fgm_id,list<Person_ID> &per_id);` Gibt eine Liste mit Personen-IDs zurück, die in der Fahrgemeinschaft mit fgm_id mitfahren.
`Fehlercode lade_verkehrsgraph();` Lädt einen neuen Verkehrsgraphen und wirft den alten weg.
`Fehlercode ist_verkehrsgraph_da(Boolean& janein);` Liefert das Vorhandensein eines Verkehrsgraphen im System.
`Fehlercode ist_mehrfach(const Kanten_ID id,Boolean& janein);` Gibt TRUE zurück, falls die Kante mit id mehrfach im Verkehrsgraph enthalten ist, sonst FALSE.
`Fehlercode gib_strassen(const Kanten_ID id,Strassenliste& liste);` Gibt alle gleichen Strassen zurück.
`Fehlercode gib_alle_bew_fkt(list<Bewertungs_ID> &bew_id_liste, list<string> &bew_name_liste);` Liefert eine Liste mit allen Bewertungsfunktionen und eine Liste mit deren Namen.
`Fehlercode gib_aktuelle_bew_fkt(Bewertungs_ID &bew_id);` Gibt die ID der aktuellen Bewertungsfunktion in bew_id zurück.
`Fehlercode setze_aktuelle_bew_fkt(const Bewertungs_ID bew_id);` Setzt aktuelle Bewertungsfunktion auf bew_id.
`Fehlercode loesche_bew_fkt(const Bewertungs_ID bew_id);` Löscht die Bewertungsfunktion bew_id.
`Fehlercode neue_bew_fkt(const string bew_name, Bewertungs_ID &bew_id);`

Es wird eine neue Bewertungsfunktion mit dem Namen bew_name angelegt und deren ID zurückgegeben.
`Fehlercode gib_parameter_bew_fkt(const Bewertungs_ID bew_id, Bew_fkt_parameter &parametersatz);` Die Parameter der aktuellen Bewertungsfunktion werden zurückgegeben.
`Fehlercode setze_parameter_bew_fkt(const Bewertungs_ID bew_id, Bew_fkt_parameter &parametersatz);` Die Parameter der Bewertungsfunktion mit der ID bew_id werden auf die übergebenen Werte gesetzt.
`Fehlercode gib_geaenderte_bew_fkten(list<Bewertungs_ID> &bew_id_liste);` Die Methode gibte eine Liste zurück, die die IDs aller Bewertungsfunktionen enthält, die seit der letzten Änderung nicht auf Platte gespeichert wurden.
`Fehlercode gib_min_max(int &min_anzahl_fgm, int &max_anzahl_fgm, Bewertung &min_bew, Bewertung &max_bew);` Die Methode liefert die Grenzwerte für die Anzahl der FGMen einer Einteilung und der Bewertung einer Einteilung auf den aktuellen Personendaten mit der aktuellen Bewertungsfunktion.
`Fehlercode bewerte_fgm(const FGM_ID fgm_id, Bewertung &wert);` Methode bewertet die Güte der Fahrgemeinschaft fgm_id der aktuellen Einteilung und gibt die Bewertung in &wert zurück.
`Fehlercode bewerte_aktuelle_einteilung(Bewertung &wert);` Methode bewertet die Güte der aktuellen Einteilung mit der aktuellen Bewertungsfunktion und gibt die Bewertung in &wert zurück.
`Fehlercode gib_optimale_algorithmen(list<Algorithmus_ID> &algo_id_liste, list<string> &algo_namen_liste);` Die Methode liefert eine Liste mit den IDs der vorhandenen optimalen Algorithmen und eine Liste mit deren Namen.
`Fehlercode gib_heuristische_algorithmen(list<Algorithmus_ID> &algo_id_liste, list<string> &algo_namen_liste);` Die Methode liefert eine Liste mit den IDs der vorhandenen heuristischen Algorithmen und eine Liste mit deren Namen.
`Fehlercode gib_inkrementelle_algorithmen(list<Algorithmus_ID> &algo_id_liste, list<string> &algo_namen_liste);` Die Methode liefert eine Liste mit den IDs der vorhandenen inkrementellen Algorithmen und eine Liste mit deren Namen.
`Fehlercode gib_wegsuche_algorithmen(list<Weg_ID> &weg_id_listes, list<string> &weg_namen_liste);` Die Methode liefert eine Liste mit den IDs der vorhandenen Wegsuchealgorithmen und eine Liste mit deren Namen.
`Fehlercode gib_aktuellen_optimalen_algor(Algorithmus_ID &algo_id);` Die Methode gibt die ID des aktuellen optimalen Algorithmus zurück.
`Fehlercode setze_aktuellen_optimalen_algor(const Algorithmus_ID algo_id);` Der Algorithmus mit der ID algo_id wird der neue aktuelle optimale Algorithmus.

`Fehlercode gib_aktuellen_heuristischen_algor(Algorithmus_ID &algo_id);` Die Methode gibt die ID des aktuellen heuristischen Algorithmus zurück.
`Fehlercode setze_aktuellen_heuristischen_algor(const Algorithmus_ID algo_id);` Der Algorithmus mit der ID algo_id wird der neue aktuelle heuristische Algorithmus.
`Fehlercode gib_aktuellen_inkrementellen_algor(Algorithmus_ID &algo_id);` Die Methode gibt die ID des aktuellen inkrementellen Algorithmus zurück.
`Fehlercode setze_aktuellen_inkrementellen_algor(const Algorithmus_ID algo_id);` Der Algorithmus mit der ID algo_id wird der neue aktuelle inkrementelle Algorithmus.
`Fehlercode gib_aktuellen_wegsuche_algor(Weg_ID &weg_id);` Die Methode gibt die ID des aktuellen Wegsuchealgorithmus zurück.
`Fehlercode setze_aktuellen_wegsuche_algor(const Weg_ID weg_id);` Der Algorithmus mit der ID algo_id wird der neue aktuelle Wegsuchealgorithmus.
`Fehlercode starte_optimale_berechnung();` Der aktuelle optimale Algorithmus zur Einteilungsberechnung wird gestartet.
`Fehlercode starte_heuristische_berechnung();` Der aktuelle heuristische Algorithmus zur Einteilungsberechnung wird gestartet.
`Fehlercode starte_inkrementelle_berechnung();` Der aktuelle inkrementelle Algorithmus zur Einteilungsberechnung wird gestartet.
`Fehlercode starte_einzelwegsuche(const Kanten_ID start,const Kanten_ID ziel, list<Kanten_ID> &ergebnisliste, int& fahrtzeit,float& laufzeit);` Die Methode liefert als Ergebnis eine Liste mit den Kanten-ID's des kürzesten Weges von start nach ziel.
`Fehlercode wegsuche_nmal(int n, float &zeit);` Die Methode berechnet kürzeste Wege zwischen n zufällig gewählten Paaren von Start- und Zielorten und gibt die Berechnungszeit zurück.
`void systemmeldung(string text,const Meldungsart art);` Gibt eine Systemmeldung an den Menüverwalter, der die Ausgabe übernimmt.
`void wurde_berechnung_unterbrochen(Abbruch& abbruch);` Fragt nach, ob die Berechnung mit "u" oder "Esc" abgebrochen wurde.
`Fehlercode name_fuer_neue_einteilung(string& name);` Gibt einen Namen fuer die neue Einteilung zurück.
`Fehlercode integer_parameter(const string beschreibung,int& wert);` Fragt nach beschreibung und gibt einen Integer-Wert zurück.
`Fehlercode double_parameter(const string beschreibung,double& wert);` Fragt nach beschreibung und gibt einen Double-Wert zurück.

`Fehlercode boolean_parameter(const string beschreibung,Boolean& wert);` Fragt nach beschreibung und gibt einen Boolean-Wert zurück.
`Fehlercode letzte_berechnung_fortsetzen();` Die zuletzt durchgeführte Berechnung wird fortgesetzt.
`Fehlercode ea_laufzeitmessung(const Meldungsart art);` Schaltet das Laufzeitmessungsattribut für art um (toggle).
`Fehlercode ea_systemmeldung(const Meldungsart art);` Schaltet das Systemmeldungsattribut für art um (toggle).
`Fehlercode drucke_weg(const Weg weg);` Gibt das Ergebnis einer Wegsuche in Postscript-Format aus. Nicht implementiert!

Attribute
`Einteilungsverwalter* einteilungsv;`
`Personenverwalter* personenv;`
`Bewertungsverwalter* bewertungsv;`
`Algorithmenverwalter* algorithmenv;`
`Menueverwalter* menuev;`
`Verkehrsgraph* graphv;`
`Lader_speicherer* dateiv;`
`Boolean systemmeldungen_algo;` Quasselstatus des Systems bei Algorithmen zur Berechnung von Einteilungen
`Boolean systemmeldungen_weg;` Quasselstatus des Systems bei einfachen Wegberechnungen
`Boolean personendatei_geoeffnet;` ist eine Personendatei geöffnet
`Boolean personendatei_geaendert;` wurde die Personendatei geändert
`Boolean laufzeitmessung_algo;` Laufzeitmessung für Algorithmen
`Boolean laufzeitmessung_weg;` Laufzeitmessung für Wegalgorithmen

geschützte Methoden
`void double_zu_string(const double zahl,string& zeichenkette);` Wandelt einen double-Wert in einen LEDA-string um.

Lader_speicherer

öffentliche Methoden
`Lader_speicherer(Datei_auswahl* dateiauswahl,Personenverwalter* personen, Einteilungsverwalter* einteilungen,Bewertungsverwalter* bewertungen, Verkehrsgraph* graph);` Konstruktor, belegt die entsprechenden Attribute mit Zeigern auf die Objekte.
`Fehlercode neue_datei(const Dateiart art);` Legt eine neue Datei der Art art mit einem zu wählenden Namen an.
`Fehlercode schliesse(const Dateiart art);` Schließt die aktuelle Datei der Art art.
`Fehlercode lade(const Dateiart art);` Lädt die der art entsprechende Dateiart, alte Daten werden verworfen.
`Fehlercode speichere(const Dateiart art);` Speichert die der art entsprechende Dateiart.
`Fehlercode speichere_unter(const Dateiart art);` Speichert die der art entsprechende Dateiart und selektiert einen neuen Namen.
`Fehlercode importiere(const Dateiart art);` Importiert zusätzlich Daten der art entsprechenden Dateiart.

Attribute
`Datei_auswahl* datei_auswahl;` die Datei_auswahl
`Personenverwalter* personenv;` der Personenverwalter
`Einteilungsverwalter* einteilungsv;` der Einteilungsverwalter
`Bewertungsverwalter* bewertungsv;` der Bewertungsverwalter
`Verkehrsgraph* graphv;` der Verkehrsgraph
`string akt_persdatei_name;` der Name der aktuellen Personendatei
`string akt_persdatei_pfad;` der Pfad der aktuellen Personendatei
`string akt_bewfunktion_name;` der Name der aktuellen Bewertungsfunktion
`string akt_bewfunktion_pfad;` der Pfad der aktuellen Bewertungsfunktion
`dictionary<Person_ID,Person_ID> per_id_merker;` hier werden die Paare von Personenids für das Laden und Importieren abgelegt

private Methoden
`Boolean lade_personendatei(ifstream& stream);`

Lädt eine Personendatei, benutzt dabei lade_personen und lade_einteilungen.
`Boolean speichere_personendatei(ofstream& stream);` Speichert eine Personendatei, benutzt dabei speichere_personen und speichere_einteilungen.
`Boolean lade_personen(ifstream& stream);` Lädt die Personendaten.
`Boolean speichere_personen(ofstream& stream);` Speichert die Personendaten.
`Boolean lade_einteilungen(ifstream& stream);` Lädt die Einteilungen, benutzt dabei lade_einteilung.
`Boolean speichere_einteilungen(ofstream& stream);` Speichert die Einteilungen, benutzt dabei speichere_einteilung.
`Boolean lade_einteilung(ifstream& stream);` Lädt eine Einteilung, benutzt dabei lade_fgmen.
`Boolean speichere_einteilung(ofstream& stream);` Speichert eine Einteilung, benutzt dabei speichere_fgmen.
`Boolean lade_fgmen(ifstream& stream);` Lädt die Fahrgemeinschaften, benutzt dabei lade_fgm.
`Boolean speichere_fgmen(ofstream& stream);` Speichert die Fahrgemeinschaften, benutzt dabei speichere_fgm.
`Boolean lade_fgm(ifstream& stream,FGMdaten& fgm);` Lädt eine Fahrgemeinschaft.
`Boolean speichere_fgm(ofstream& stream,FGMdaten fgm);` Speichert eine Fahrgemeinschaft.
`Boolean lade_person(ifstream& stream,Personendaten& person);` Lädt eine Person aus einer Datei und gibt zurück, ob Laden erfolgreich war
`Boolean speichere_person(ofstream& stream,Personendaten person);` Speichert eine Person in eine Datei und gibt zurück, ob Speichern erfolgreich war
`Boolean lade_bewertungsfunktion(ifstream& stream,Bew_fkt_parameter& bew_fkt_param);` Lädt eine Bewertungsfunktion.
`Boolean speichere_bewertungsfunktion(ofstream& stream,Bew_fkt_parameter bew_fkt_param);` Speichert eine Bewertungsfunktion.
`void merke(const Person_ID id_alt,const Person_ID id_neu);` Merkt sich in einem Dictionary das Zahlenpaar id_alt, id_neu, wobei id_alt der Schlüssel ist.
`Person_ID ersetze_id(const Person_ID per_id);` Gibt den Inhalt von per_id_merker mit dem Schlüssel per_id zurück, falls ein Eintrag existiert, sonst per_id.

Datei_auswahl

öffentliche Methoden
`Datei_auswahl(Voreinstellungen* voreinst);`

`Fehlercode selektiere(const Dateiart art,const Dateizugriff zugriffsart);` Selektiert Name und Pfad einer Datei der Dateiart art.
`Fehlercode gib_name_und_pfad(const Dateiart art,string& name,string& pfad);` Liefert den selektierten Namen und Pfad.
`Fehlercode gib_pfad_name(const Dateiart art, string& pfadname);` Liefert den selektierten Pfad und Namen für den direkten Zugriff.

Attribute
`Voreinstellungen* voreinstellungen;` die Voreinstellungen
`string aktuelles_verzeichnis;` das aktuelle Verzeichnis
`string akt_per_name;` der aktuelle Name der Personendatei
`string akt_per_pfad;` der aktuelle Pfad der Personendatei
`string akt_bew_name;` der aktuelle Name der Bewertungsfunktion
`string akt_bew_pfad;` der aktuelle Pfad der Bewertungsfunktion
`string akt_kno_name;` der aktuelle Name des files für die Knoten des Verkehrsgraphen
`string akt_kno_pfad;` der aktuelle Pfad des files für die Knoten des Verkehrsgraphen
`string akt_kan_name;` der aktuelle Name des files für die Kanten des Verkehrsgraphen
`string akt_kan_pfad;` der aktuelle Pfad des files für die Kanten des Verkehrsgraphen
`string akt_reg_name;` der aktuelle Name des files für die Regionen des Verkehrsgraphen
`string akt_reg_pfad;` der aktuelle Pfad des files für die Regionen des Verkehrsgraphen
`dictionary<string,string> dateien;` Dateiliste für die Auswahl beim lesenden Zugriff

private Methoden
`void strings_ausgeben(list<string> string_liste);` Gibt die Liste von Strings auf dem Bildschirm aus.
`void strings_numeriert_ausgeben(list<string> string_liste);` Gibt die Liste von Strings auf dem Bildschirm aus und versieht sie mit aufsteigenden Nummern, ausgehend von der 1.
`void datei_ausgeben(const Dateiart art);` Gibt den Namen und Pfad der aktuellen Datei der gewaehlten art aus.

`Boolean datei_einlesen(const Dateiart art,const Dateizugriff zugriffsart);` Liest den Namen und Pfad einer Datei der Art art ein, setzt den aktuellen Pfad und Namen auf die eingelesenen Werte, gibt TRUE zurueck falls die Datei existiert.
`Boolean wert_einlesen(string& ergebnis);` Liest eine Zahl, die einer Datei entspricht, ein “a” oder ein “v” ein, sonst Rückgabewert=FALSE.
`Boolean name_einlesen(string & ergebnis);` Liest den Namen einer Datei ohne Endung oder ein “v” ein, sonst Rückgabewert=FALSE.
`void einschraenken(const list<string> quelle,const Dateiart art,list<string>& ziel);` Schränkt eine Liste auf Namen ein, die eine Endung entsprechend der art haben.
`Boolean lese_janein(const string defaultwert,string& ergebnis);` Liest “j” oder “n” ein, falls nichts angegeben wird, wird dem ergebnis der defaultwert zugewiesen, bei einer falschen Eingabe wird FALSE zurückgegeben.
`Boolean verzeichnis_wechseln(Dateiart dateiart, Dateizugriff zugriffsart);` Wechselt in ein einzugebendes Verzeichnis. Gibt zurück, ob das Verzeichnis existiert. Gewechselt wird nur, falls das Verzeichnis existiert.

Voreinstellungen

öffentliche Methoden
`Voreinstellungen();` Im Konstruktor wird die Datei voreinstellungen.mbd eingelesen, falls vorhanden; ansonsten werden die Pfadattribute mit dem Aufrufpfad belegt.
`Fehlercode setze_pfad(const Dateiart art,const string pfad);` Setzt den art entsprechenden Pfad auf pfad.
`Fehlercode gib_pfad(const Dateiart art,string& pfad);` Setzt pfad auf den art entsprechenden Pfad.
`Fehlercode setze_name(const Dateiart art,const string name);` Setzt den art entsprechenden Namen auf name.
`Fehlercode gib_name(const Dateiart art,string& name);` Setzt name auf den art entsprechenden Namen.
`Fehlercode gib_max_anzahl_teilnehmer(int& anzahl);` Setzt anzahl auf max_anz_teilnehmer.

Attribute
`string bewertungsname;` Name der Bewertungsfunktion
`string bewertungspfad;` Pfad für Bewertungsfunktionen
`string personenpfad;`

Pfad für Personen- und Einteilungsdaten
`string personenname;` Name der Personendatei
`string knotenpfad;` Pfad für Knoten im Verkehrsgraphen
`string knotenname;` Name für Knoten im Verkehrsgraphen
`string kantenpfad;` Pfad für Kanten im Verkehrsgraphen
`string kantenname;` Name für Kanten im Verkehrsgraphen
`string regionenpfad;` Pfad für Regionen im Verkehrsgraphen
`string regionenname;` Name für Regionen im Verkehrsgraphen
`int max_anz_teilnehmer;` Maximale Anzahl der Teilnehmer einer Fahrgemeinschaft

private Methoden
`Boolean voreinstellungen_einlesen();` Liest aus der Datei voreinstellungen.mbd im aktuellen Verzeichnis die vorhandenen Voreinstellungen in die Attribute ein; gibt FALSE zurueck, falls die Datei nicht existiert.

5.5.8 Ergänzungsklassen

Optional

öffentliche Methoden
`Optional() validity_flag = false;` Konstruktor der Klasse Optional.
`Optional(Type argument) x = argument;validity_flag = true;` Variablenzuweisen. Optional<int> zahl; zahl = 5;
`void nichts() validity_flag = false;` Hier kann der Wert einer Variable geloescht werden.
`bool ist_gesetzt() return (validity_flag);` Hier kann ueberprueft werden, ob ein Wert gesetzt wurde. Ist dies der Fall, so wird True zurueckgegeben, ansonsten false.
`void setze_wert (Type argument) x = argument;validity_flag = true;` Hier kann der Wert der Variable explizit gesetzt werden.
`Type gib_wert () if (validity_flag) return (x);` Hier wird der Wert der Variable zurueckgegeben.

Attribute
`Type x;` Wertvariable
`bool validity_flag;` Attribut, ob ein Wert gesetzt wurde: true = gesetzt und false = nicht gesetzt.

Zeit

öffentliche Methoden
`Zeit();` Konstruktor der Klasse Zeit.
`Zeit();` Destruktor der Klasse Zeit.
`friend int compare(const Zeit &z1, const Zeit &z2);` Compare zwischen zwei Zeiten.
`void setze_zeit(const int stunden,const int minuten,const int sekunden);` Setzt die Zeit: Stunden, Minuten, Sekunden.
`void setze_zeit_in_sekunden(const int sekunden);` Setzt die Zeit (Sekunden,Minuten,Stunden), die als Sekundenzahl uebergeben wurde.
`void gib_zeit(int &stunden,int &minuten,int &sekunden);` Gibt die Zeit, aufgesplittet in Stunden, Minuten und Sekunden, zurueck.
`int gib_uhrzeit_in_sekunden();` Gibt die Uhrzeit in Sekunden zurueck.
`Boolean ist_zeit_kleiner(const Zeit &eine_zeit);` Die Methode prueft, ob eine_zeit kleiner ist als die eigene Zeit. Wenn eine_zeit kleiner ist wird TRUE zurueckgegeben, wenn eine_zeit groesser oder gleich ist, wird FALSE zurueckgegeben.
`void gib_uhrzeit_als_intervall(Zeitintervall &zeiten);` Gibt die ueberlieferte Zeit als Zeitintervall zurueck, wobei zeit_von und zeit_bis gleich sind.

Attribute
`int stunden_zeit;` Stunden
`int minuten_zeit;` Minuten
`int sekunden_zeit;` Sekunden

Intervall

öffentliche Methoden
`Intervall();`

Konstruktor der Klasse Intervall.
`Intervall();` Destruktor der Klasse Intervall.
`void setze_intervall(const Zeitintervall zeiten);` Setzt das Intervall zweier Zeiten.
`void setze_von_zeit(const Zeit &eine_zeit);` Setzt den Anfangszeitpunkt auf eine_zeit.
`void setze_bis_zeit(const Zeit &eine_zeit);` Setzt den Endzeitpunkt auf eine_zeit.
`void gib_intervall(Zeitintervall &zeiten);` Gibt das Intervall zweier Zeiten zurueck.
`Boolean gib_intervall_vorhanden();` Gibt zurueck, ob ein Intervall vorhanden ist (TRUE) oder nicht (FALSE)
`Boolean pruefe_intervall(Intervall fix_zeit);` Prueft, ob die Zeit anzeit1 und die vorhandene Zeit anzeit2 im Objekt einen gemeinsamen Schnittpunkt haben und liefert dann TRUE = JA und FALSE = Keinen zurueck.
`Boolean schnittmenge(Intervall &inter2, Intervall &ergebnis);` Wenn eine Schnittmenge existiert, wird TRUE und die Schnittmenge in ergebnis zurueckgegeben, sonst FALSE.
`void gib_anfang_intervall(Zeit &zeitpunkt);` Liefert den Anfangszeitpunkt eines Intervalls.
`void gib_ende_intervall(Zeit &zeitpunkt);` Liefert den Endzeitpunkt eines Intervalls.

Attribute
`Zeit intervall_von;` Anfang vom Intervall
`Zeit intervall_bis;` Ende vom Intervall
`Boolean intervall_vorhanden;` Es gibt ein Schnittintervall

Kapitel 6

Testphase

Die letzte Phase im Entwicklungsprozeß von Mobidick ist die Testphase. Dabei sei von den in Eigenverantwortung durchgeführten Modultests in der Programmierphase und der Tests gegen die Spezifikation, die mehrfach durchgeführt wurden nicht die Rede. Die hier vorgestellten Tests wurden auf einem Sparc-Prozessor unter SunOS 5.5.1 durchgeführt und als C++-Compiler wurde der gnu-Compiler Version 2.7.2.1 verwendet. Die Ergebnisse entstanden durch Anwenden von verschiedenen Einteilungs- und Wegsuchealgorithmen auf bestimmten Personendaten, die nun im folgenden vorgestellt werden.

Die Dateien für das Testen der Algorithmen haben folgenden Aufbau:
Stglv und Ziglv stehen für Startort gleichverteilt bzw. Zielort gleichverteilt, was bedeutet, daß die Koordinaten aus dem Verkehrsgarphen zufällig ausgewählt wurden. Der Name ist der Name der Testdatei. Jede Datei enthält eine Einteilung, die inkrementell erstellt wurde, eine mit dem MM-Algorithmus, eine mit dem optimalen Algorithmus (falls möglich) berechnete und eine mit dem MM-Algorithmus erstellte und dem inkrementellen Algorithmus verbesserte Einteilung. Die Anzahl entspricht der Anzahl der Personen in der Personenmenge. Der Wert Fahrer gibt an, wieviel Prozent der Personen Fahrer sind. Die Ankunft und Rückfahrt gibt das Zeitintervall an, in dem die Person abfährt oder zur Rückfahrt bereit ist. Die Längen geben ein Intervall an, das um die Abfahrt- oder Ankunftszeit herum eine zusätzliche "Reservezeit" gibt.

Name	Anzahl	Fahrer	Stglv	Ziglv	Ankunft	Länge	Rückfahrt	Länge
Test_10.per	10	50%	j	j	08:00-09:00	1	18:00-19:00	1
Test_20.per	20	50%	j	j	08:00-09:00	1	18:00-19:00	1
Test_50.per	50	50%	j	j	08:00-09:00	1	18:00-19:00	1
Test_100.per	100	50%	j	j	08:00-09:00	1	18:00-19:00	1
Test_200.per	200	50%	j	j	08:00-09:00	1	18:00-19:00	1
Test_1000.per	1000	50%	j	j	08:00-09:00	1	18:00-19:00	1

Nun folgen die Arten von Algorithmen mit einer Angabe der benutzten Bewertungsfunktion, der Laufzeit in Sekunden, der erzielten Bewertung der Einteilung, dem Namen der Einteilung und den Fahrgemeinschaften. Hier werden die Anzahlen der unterschiedlichen Fahrgemeinschaften aufgeführt, sowie die unvermittelten Personen (uv.). 4 2er bedeutet also zum Beispiel, daß die Einteilung vier Fahrgemeinschaften mit je zwei Teilnehmern enthält. Der optimale Algorithmus ist durch die Anzahl der Personen und die Prozentzahl der Fahrer eingeschränkt. Bei zu großen Zahlen (mind. ab 20) ist keine Berechnung mehr möglich, da der Speicherplatz zu knapp wird.

MM_Algorithmus (siehe auch Abschnitt 5.2.5.1)

PersDatei	BewFkt	Zeit	Wert	Einteilung	FGMen
Test_10.per	bewertung.fkt	0,099998	0,466481	MM_alg1	4 2er, 2 uv.
Test_20.per	bewertung.fkt	0,566666	0,481512	MM_alg1	1 3er, 7 2er, 3 uv.
Test_50.per	bewertung.fkt	3,833313	0,54213	MM_alg2	1 4er, 1 3er, 21 2er, 1 uv.
Test_100.per	bewertung.fkt	18,0	0,475253	MM1_alg	46 2er, 8 uv.
Test_200.per	bewertung.fkt	90,816650	0,510423	MM1_alg	4 4er, 88 2er, 8 uv.
Test_1000.per	bewertung.fkt	7148,399902	0,509607	MM_alg1	26 4er, 1 3er, 436 2er, 21 uv.

Inkrementeller_Algorithmus

PersDatei	BewFkt	Zeit	Wert	Einteilung	FGMen
Test_10.per	bewertung.fkt	0,450005	0,355535	Ink_alg1	3 2er, 4 uv.
Test_20.per	bewertung.fkt	3,516663	0,426336	Ink_alg1	7 2er, 6 uv.
Test_50.per	bewertung.fkt	57,383347	0,473089	Ink_alg1	18 2er, 14 uv.
Test_100.per	bewertung.fkt	557,216675	0,445647	Ink_alg1	34 2er, 32 uv.
Test_200.per	bewertung.fkt	3843,916504	0,475834	Ink_alg1	69 2er, 62 uv.

Optimaler_Algorithmus

PersDatei	BewFkt	Zeit	Wert	Einteilung	FGMen
Test_10.per	bewertung.fkt	708,466614	0,466481	Opt_alg1	4 2er, 2 uv.

Inkrementeller_Algorithmus auf Ergebnis von MM_Algorithmus angewandt

PersDatei	BewFkt	Zeit	Wert	Einteilung	FGMen
Test_10.per	bewertung.fkt	0,016663	0,466481	Ink_MM1	4 2er, 2 uv.
Test_20.per	bewertung.fkt	0,333313	0,484869	Ink_MM1	1 3er, 7 2er, 3 uv.
Test_50.per	bewertung.fkt	1,516663	0,542595	Ink_MM1	3 3er, 20 2er, 1 uv.
Test_100.per	bewertung.fkt	1,666626	0.475253	Ink_MM1	46 2er, 8 uv.
Test_200.per	bewertung.fkt	93,649902	0,517245	Ink_MM2	1 4er, 6 3er, 85 2er, 8 uv.

Die Testergebnisse belegen den Nutzen des MM-Algorithmus für das Berechnen einer neuen Einteilung mit allen Personen und den Nutzen des Inklrementellen Algorithmus für das Verbessern von Einteilungen (vom MM-Algorithmus berechnete oder durch Hinzufügen neuer Personen entstandene).

Teil II

Mobidick

Kapitel 7

Bedienung

7.1 Erste Schritte

Das Programm wird aus einem Xterminal heraus mit dem Befehl *mobidick* gestartet. Es erscheinen einige Meldungen über den Zustand des Systemaufbaus. Der Graphenviewer wird geladen. Man bekommt das Hauptmenü angezeigt und der Graphenviewer erscheint in einem eigenen Fenster.

Nun kann man sich durch Auswahl der Menüpunkte durch das System bewegen. Um arbeiten zu können, benötigt man eine Personendatei. Dazu lädt man entweder eine bereits vorhandene oder erzeugt eine neue. Hat man eine neue Personendatei erzeugt, muß man eine Bewertungsfunktion anlegen, um eine Einteilung berechnen zu lassen. Die vorhandene Personendatei kann man nun durch Hinzufügen weiterer Personen, Anlegen weiterer Einteilungen oder Veränderung bestehender Daten erweitern oder ändern. Geänderte Personendateien sollte man vor dem Verlassen des Systems speichern. Auch beim Laden einer neuen Personendatei wird bei einer geänderten Personendatei eine Speicheranfrage gestartet.

Lädt man zu Beginn eine bereits erstellte Personendatei, sollte man noch einmal einen Algorithmus laufen lassen, da sonst keine Entfernungstabelle erstellt wird und somit eine Bewertung nicht möglich ist; das Ergebnis einer Bewertung wird immer 0 ergeben. Näheres zur Entfernungstabelle kann man dem Kapitel 8.6 entnehmen.

7.2 Allgemeines

7.2.1 Menüs

Das System präsentiert sich dem Benutzer in einem hierarchischen Menüsystem, aus dem heraus die verschiedenen Funktionen aufgerufen werden können. Alle Auswahlmenüs enthalten die Punkte *Zurück*, *Hauptmenü* und *Hilfe*, wo-

bei *Zurück* ins aufrufende, darüberliegende Menü wechselt, *Hauptmenü* in das Haupmenü und *Hilfe* einen Hilfetext für das Menü anzeigt.

Folgende Auswahlmenüs befinden sich im System:

Das Hauptmenue

1. Dateien
2. Personen
3. Fahrgemeinschaften
4. Vermittlung
5. Bewertungsfunktion
6. Wegsuche

Von diesem Menü aus werden die aufgeführten Menüs aufgerufen (Dateien, Personen, Fahrgemeinschaften,...). Statt den Punkten *Zurück* und *Hauptmenü* gibt es hier den Punkt *Ende*, mit dem das Programm verlassen wird.

Dateien

1. Personendateien
2. Fahrgemeinschaftseinteilung
3. Bewertungsfunktionen

Das Menü *Dateien* ist für die Datenverwaltung zuständig. Zwar werden sämtliche zu einem Personenstamm gehörende Daten - Einteilungen und Bewertungsfunktionen - zusammen in einer Datei abgespeichert, trotzdem ist es möglich, Bewertungsfunktionen vom System aus gesondert zu laden und zu speichern.

Personen

1. Neue Person
2. Person aendern
3. Person loeschen
4. Person generieren
5. Personen anzeigen

Im Menü *Personen* kann man die Funktionen aufrufen, die für das Verwalten der Personendaten nötig sind, zum Anlegen einer Person, zum Ändern oder zum Generieren einer zufälligen Personenmenge, mit deren Hilfe man Algorithmen testen kann.

Fahrgemeinschaften

1. Neuer Teilnehmer
2. Teilnehmer fest eintragen
3. Teilnehmer loeschen
4. Fahrgemeinschaft eingeben
5. Fahrgemeinschaft aendern
6. Fahrgemeinschaft aufloesen
7. Fahrgemeinschaft bewerten
8. Fahrgemeinschaft anzeigen
9. Fahrgemeinschaft markieren|unmarkieren

Das Menü *Fahrgemeinschaften* dient der manuellen Verwaltung von Fahrgemeinschaften. Hier kann man unter anderem Personen, die sich schon im System befinden, aus Fahrgemeinschaften nehmen, sie von *reserviert* auf *festeingetragen* ändern und einzelne Fahrgemeinschaften bewerten lassen.

Vermittlung

1. Einteilung auswaehlen
2. Systemmeldungen (ein|aus)
3. Einteilung berechnen
4. Fortfahren mit letzter Berechnung
5. Laufzeitmessung (ein|aus)
6. Einteilung bewerten
7. Einteilung anzeigen
8. Einteilung aufloesen

Das ist das Herzstück des Systems. Hier werden Fahrgemeinschaften automatisch aus dem Personenstamm gebildet. Dazu werden verschiedene Algorithmen angeboten.

Bewertungsfunktion

1. Neu
2. Ändern
3. Auswählen
4. Anzeigen

Mit Hilfe der Bewertungsfunktion wird die Güte einer Einteilung festgestellt. Hier kann der Benutzer selbst solche Funktionen anlegen oder bei bestehenden die Parameter ändern, um noch bessere Einteilungen zu erhalten.

Wegsuche

1. Einzelwegsuche
2. Wegsuche n-mal
3. Auswahl des Algorithmus
4. Laufzeit
5. Systemmeldungen (ein|aus)

Die Wegsuche ist ein kleines Bonbon. Wegsuchealgorithmen werden bei der Umwegberechnung der Algorithmen eingesetzt. In diesem Menü kann man sie ausgekoppelt benützen, um einfach den kürzesten Weg zwischen zwei Punkten im Verkehrsgraphen zu bestimmen. Auch hier stehen verschiedene Algorithmen zur Auswahl. Hat man einen besonders schnellen ausgewählt, so kann man mit *Wegsuche n-mal* diesen n Male über zufällig gewählte Punkte laufen lassen und die Laufzeit bestimmen.

7.2.2 Graphenviewer

Das Programm *graphdraw* wird zusammen mit Mobidick gestartet. Es ermöglicht die graphische Darstellung des Verkehrsgraphen. Bei bestimmten Funktionen, wie z.B. der Wegsuche, wird der Benutzer aufgefordert, Straßenkanten über den Graphen auszuwählen. Dies geht durch Anklicken der entsprechenden Kanten im Anzeigefenster.

Mit *graphdraw* hat man außerhalb des Systems die Möglichkeit, nach Straßen zu suchen, sich die Namen von Kanten anzeigen zu lassen, sofern sie in den GDF-Daten vorhanden sind und den Bereich der Kanten auf bestimmte Straßenklassen einzuschränken.

Innerhalb des Systems dient *graphdraw* der Eingabe von Kanten. Diese kann man zur Eingabe suchen lassen, den Bereich vergrößern oder verkleinern und dann die entsprechende Kante durch Mausklick auswählen.

Abb. 7.1: Das Programm *graphdraw*

Wie in Abbilung 7.1 zu sehen, besitzt *graphdraw* zwei Bereiche. Im größeren Bereich ist der aktuelle Verkehrsgraph zu sehen. Die Straßen haben unterschiedliche Farben, die ihre Ebene symbolisieren (Autobahn, Bundesstraße, Nebenstraße, ...). Der obere Bereich besteht aus zwei Zeilen. Im oberen sind Steuerfelder untergebracht. Den Knopf *Quit* sollte man nur benutzen, wenn man auch Mobidick verläßt, um Inkonsistenzen zu vermeiden. In der zweiten Zeile sind die Informationen zu der Straße zu sehen, über der sich gerade der Mauszeiger befindet: ihr Name, ihre Kantennummer und eventuell zusätzliche Angaben (z.B. B27).

7.2.3 Systemmeldungen

Mobidick informiert den Benutzer über relevante Informationen des Systemablaufs oder wenn ein Fehler aufgetreten ist. In Abschnitt 7.10 befindet sich eine Tabelle mit den wichtigsten Meldungen, ihren Ursachen und eventuellen Lösungsvorschlägen.

7.2.4 Voreinstellungen

Voreinstellungen sucht Mobidick in der Datei *voreinstellungen.mbd* im Verzeichnis, aus dem heraus Mobidick gestartet wurde. Die Datei kann mit jedem Texteditor geändert werden. Hier werden die Pfade definiert, unter denen sich die

Personendatei und die Bewertungsfunktionen befinden. Bisher wird jedoch von einem festen Verkehrsgraphen ausgegangen, der in einem Verzeichnis liegt, das in *constants.h* angegeben ist. Sind die Angaben für die Personendatei spezifiziert, so wird sie beim Systemstart geladen. Die Angaben für die Personendatei und die Bewertungsfunktionen werden von der Datei-Auswahl benutzt.

7.3 Importieren und Exportieren von Daten

Mobidick kennt Personendateien (*.per*) und Bewertungsfunktionen (*.fkt*). Das Speichern erfolgt ohne weitere Eingabeaufforderung. Mit Hilfe der Datei-Auswahl werden die entsprechenden Dateiarten, wenn sie den im Kapitel 8 beschriebenen Dateiformaten entsprechen, selektiert und anschließend geladen oder unter neuem Namen gespeichert.

7.3.1 Datei-Auswahl

Die Datei-Auswahl dient dem Selektieren von zu ladenden oder unter neuem Namen zu speichernden Dateien. Man hat immer die Möglichkeit, das Verzeichnis mit Eingabe von “`v`” zu wechseln. Eine Eingabeaufforderung erwartet den absoluten Pfad ohne abschließenden “`/`”, den relativen Pfad ausgehend vom aktuellen Verzeichnis ohne abschließenden “`/`” oder ein “`..`”. Außerdem kann man durch Eingabe von “`a`” eine voreingestellte Datei auswählen. Ist eine Datei voreingestellt, wird ihr Pfad und Name nach “`Datei :`” angezeigt. Sind die Daten unvollständig, wird die Datei nicht geladen.

Beim Laden präsentiert sich zum Beispiel folgendes:

```
Dateiauswahl: selektieren

1 personen.per
2 personen1.per
3 personen2.per
4 personen3.per
5 personen4.per
Datei :/home/fgm/implementierung/work/personen.mbd
Waehlen Sie eine der angezeigten Dateien ueber ihre Nummer,
‘a’ fuer akzeptieren oder ‘v’ fuer Verzeichnis wechseln
Ihre Wahl:
```

Man hat nun die Möglichkeit, eine der Personendateien über ihre Nummer anzuwählen. Diese wird dann direkt geladen. Man kann das Verzeichnis wechseln, oder die voreingestellte Datei wählen.

Beim Speichern unter einem neuen Namen präsentiert sich zum Beispiel folgendes:

```
Dateiauswahl: selektieren
```

```
stuttgart_west.per
stuttgart_ost.per
esslingen.per
Datei :/home/porrmaar/implementierung/stuttgart_ost.per
Geben Sie einen Namen fuer die zu speichernde Datei (ohne Endung),
'a' fuer akzeptieren oder 'v' fuer Verzeichnis wechseln ein
Ihre Wahl:
```

Hier gibt man nun den Namen ein, unter dem die aktuelle Personendatei gespeichert werden soll. Die Endung wird automatisch angehängt. Ist er mit einem der angezeigten identisch, wird nachgefragt "`Name existiert schon, trotzdem abspeichern und vorhandene Datei ueberschreiben? [j/N]`". Bei Eingabe von "`j`" wird die Datei überschrieben, sonst landet man wieder bei der Datei-Auswahl.

7.3.2 Personendatei

Personendateien liegen in dem im Kapitel 8 beschriebenen Format vor. Sie haben die typische Endung *.per*. Man begibt sich in das Menü *Dateien-Personendateien*.

Möchte man eine neue Personendatei anlegen wählt man den Punkt *Neu*. Man bekommt nun die gleiche Eingabeaufforderung wie beim Speichern unter neuem Namen.

Zum Laden einer Personendatei wählt man den Punkt *Laden*. Wählt man nun über die Datei-Auswahl eine Personendatei, werden die im System vorhandenen Daten verworfen und durch jene aus der ausgewählten Datei ersetzt.

Zum Speichern von Personendateien wählt man den Punkt *Speichern*. Es erfolgt automatisch unter dem aktuellen Namen und Pfad der Datei. Dabei wird eine möglicherweise bereits vorhandene Datei überschrieben.

Soll die aktuelle Personendatei unter anderem Namen gespeichert werden, wählt man *Speichern unter*. Nach Eingabe eines Namens über die Datei-Auswahl wird die aktuelle Personendatei abgespeichert.

Falls eine geänderte Personendatei verworfen werden soll, wählt man den Menüpunkt *Schliessen*.

Das Importieren von Daten erfolgt durch den Punkt *Importieren*. Dabei wird eine Prüfung auf das Vorhandensein einer Person in beiden Personendateien durchgeführt und bei Erfolg die zu importierenden Daten entsprechend angepasst.

7.4 Anlegen eines Personenstammes

Zum Anlegen eines Personenstammes gib es generell zwei Möglichkeiten. Zum einen lassen sich Personen von Hand eintragen, wie es in Mitfahrzentralen der Fall ist. Zum anderen können Personen, vor allem zu Testzwecken, automatisch generiert werden. Im folgenden wird der Umgang mit diesen beiden Möglichkeiten vorgestellt und erklärt.

7.4.1 Personen manuell eintragen

Eine neue Person soll in den Personenstamm mitaufgenommen werden. Dazu muß im Hauptmenü der Menüpunkt *Personen* und dort der Menüpunkt *Neue Person* ausgewählt werden. Nun können die Daten der Person eingetragen werden. Damit das Fahrgemeinschaftensystem korrekt arbeiten kann, müssen die Daten einer Person vollständig sein. Vollständigkeit bedeutet, daß zu den nun aufgeführten Abfragen eine syntaktisch korrekte Angabe gemacht wird (siehe Use Case 3.3.6.1):

1. `Name *`
2. `Geburtsdatum (TT.MM.JJ) *`
3. `Fahrer (j/n) *`
4. `Anzahl Plaetze im Auto *`
5. `Startkoordinaten (bitte anklicken) *`
6. `Zielkoordinaten (bitte anklicken) *`
7. `Ankunftszeit (HH:MM[:SS]-HH:MM[:SS]) *`
8. `Rueckkehrzeit (HH:MM[:SS]-HH:MM[:SS]) *`
9. `Arbeitszeit (HH:MM[:SS]) *`

Die notwendigen Angaben erkennt man einem einen Stern `"*"` hinter dem Abfragenamen. Bei allen anderen Abfragen ist eine Eingabe nicht zwingend erforderlich, da diese Daten für die Bildung von Fahrgemeinschaften nicht gebraucht werden. Allerdings können diese für die Büroorganisation sehr nützlich sein.

Generell ist bei der Eingabe der Daten auf das richtige Format zu achten, z.B. Geburtstag (TT.MM.JJ). Für das Berechnen einer Fahrgemeinschaft ist der Start- und Zielort einer Person von großer Bedeutung. Aus diesem Grund können hier über den Graphenviewer, der zur selben Zeit wie Mobidick gestartet wird, die Koordinaten bestimmt werden. Dazu muß lediglich die gewünschte Straße gefunden werden und anschließend durch Drücken der linken Maustaste als Start- bzw. Zielkoordinate bestätigt werden (siehe Punkt 7.2.2).

Nachdem die Eigenschaften der Person eingegeben sind, werden die Wünsche der Person abgefragt. Wünsche sind Kriterien, die diese Person an andere Personen, die mit ihr in einer Fahrgemeinschaft zusammenfahren sollen, stellt. Hier kann in der aktuellen Version auf drei Wünsche eingegangen werden:

- Geschlecht (m,w)
- Raucher (ja/nein)

- Personen mit Zu- bzw. Abneigung

Bei den ersten beiden Wünsche kann der entsprechende Typ gewählt werden, über dem eine Aussage gemacht werden soll, d.h. `"Moechte nur mit Frauen/Maennern fahren (w/m)"` bzw. `"Moechte nur mit Rauchern/Nichtrauchern fahren (r/nr)"` . Anschließend muß diesem Typ eine Gewichtung gegeben werden, wie wichtig das für die neue Person ist. Dabei trägt jeder Wunsch einen Gewichtungsfaktor von 0 bis 10. 0 steht für völlig unwichtig und 10 für sehr wichtig. Durch die Belegung mit dem Gewicht 0 werden Wünsche unerheblich für die Bewertung.

Im letzten Punkt kann die neue Person angeben, mit welchen weiteren Personen sie auf jeden Fall zusammenfahren möchte (Zuneigung) und mit welchen nicht (Abneigung). Hierfür wird eine Liste aller Personen angezeigt, die sich aktuell im Personenstamm befinden. Nun kann eine entsprechende Person ausgesucht und durch Angabe ihrer ID in die Liste der Ab- bzw. Zuneigung übernommen werden. Will man weitere Personen in die Liste mitaufnehmen, so muß die Frage: `"Weitere Personen eintragen?"` mit `"J"` beantwortet werden. Bei `"n"` muß durch die Frage `"Personen zur Zuneigungsliste hinzufuegen?"` endgültig bestätigt werden, ob nun die Listen mit den ausgewählten Personen erweitert werden sollen.

Nachdem nun alle Daten der Person korrekt eingegeben sind, können sie durch positives Bestätigen der Frage `"Soll die Person in den Datenbestand uebernommen werden?"` in den Personenstamm mitaufgenommen werden.

7.4.2 Personen generieren

Um für Testzwecke eine große Anzahl von Personendaten zu Verfügung zu haben, kann im Hauptmenü der Menüpunkt *Personen* und im Untermenü der Menüpunkt *Personen generieren* ausgewählt werden. Hier können Personen nach bestimmten Kriterien automatisch generiert werden. Es läßt sich die Zahl der Personen und davon der Anteil der Fahrer in Prozent angeben. Weiterhin kann der Start- und Zielort durch Zufall oder durch Angabe einer festen Straßenkante, die mit Hilfe des Graphenviewers ausgewählt wird, bestimmt werden. Bei einer festen Straßenkante besitzen nach der Generierung alle Personen den gleichen Start- bzw. Zielort. Das Rück- und Ankunftsintervall gibt den gesamten Bereich des Intervalls an, aus dem zufällig ein Startzeitpunkt bestimmt wird. Durch die Intervallänge kann nun die Länge des Intervalls, allerdings für jede Person gleich, entsprechend angegeben werden. Daraus entstehen für die Personen unterschiedliche Rück-und Ankunftszeiten. Wird am Ende die Frage `"Personenmenge so generieren?"` mit `"J"` beantwortet, werden die generierten Personen zu den bisherigen Personen hinzugefügt.

7.5 Anlegen einer Einteilung

Einteilungen bestehen aus Fahrgemeinschaften und können manuell oder automatisch angelegen werden. Im folgenden wird nur der Vorgang für das manuelle

Anlegen beschrieben, die andere Möglichkeit wird unter Einteilung berechnen lassen (siehe Punkt 7.7) dargestellt.

Damit Fahrgemeinschaften angelegt werden können, muß eine Einteilung angelegt sein. Dies geschieht durch Auswahl der Menüpunkte *Dateien* im Hauptmenü, *Fahrgemeinschaftseinteilung* im Untermenü und dort im Untermenü der Menüpunkt *Neu*. Nach Eingabe des Namens der neuen Einteilung wird die Einteilung angelegt und als aktuelle Einteilung ausgewählt. Das bedeutet, daß sich alle nachfolgenden Operationen, wie Umbenennen, Duplizieren und Löschen auf diese Einteilung auswirken. Will man eine andere Einteilung zur aktuellen auswählen, muß im Hauptmenü *Vermittlung* der Menüpunkt *Einteilung auswaehlen* gewählt werden. Die im System vorhandenen Einteilungen werden angezeigt und durch Angabe der ID der neuen aktuellen Einteilung wird die nun aktuelle Einteilung bestimmt.

Nachdem eine Einteilung vorliegt, kann mit der manuellen Einteilung der Personen in Fahrgemeinschaften begonnen werden (siehe Punkt 7.6.2.5).

7.6 Arbeiten mit den Datensätzen

Im folgenden wird der Umgang mit Personen- und Fahrgemeinschaftsdaten beschrieben.

7.6.1 Personendaten

Wählt man im Hauptmenü den Menüpunkt *Personen*, erscheint folgendes Untermenü:

```
---------------------
 MOBIDICK - Personen
---------------------
1. Neue Person
2. Person aendern
3. Person loeschen
4. Person generieren
5. Person anzeigen
```

7.6.1.1 Neue Person

Die Daten einer neuen Person können eingegeben und anschließend dem Personenstamm hinzugefügt werden (siehe Punkt 7.4.1).

7.6.1.2 Person ändern

Die Daten einer Person können hier verändert werden. Zum schnelleren Auffinden der richtigen Person hilft der Personenfilter. Dieser sollte zunächst mit

Daten versorgt werden. Das bedeutet, man kann zu den verschiedenen Filterkriterien Angaben machen. Soll keine Angabe zu einem Filterkriterium gemacht werden, muß nur `RETURN` gedrückt werden, d.h. keine Einschränkung. Gibt es überhaupt keine Einschränkungen, werden zur Auswahl alle Personen angezeigt, ansonsten nur solche, welche die Filterkriterien erfüllen. Die angezeigte Liste enthält die Information über Name, Vorname und Wohnort der gefundenen Personen. Wählt man nun eine Person aus der Liste aus, können die Daten geändert werden (siehe Punkt 7.6.1.1). Der Unterschied zu *Neue Person* ist folgender: Die bereits gesetzten Daten der Person werden in eckigen Klammern angezeigt und nur bei Abweichung dieser Daten müssen neue Daten eingegeben werden, ansonsten kann mit `RETURN` einfach weitergesprungen werden. Wurde eine Änderung vorgenommen, können am Ende durch Beantworten der Frage `"Aenderungen noch einmal anzeigen?"` mit `"J"` die gesamten Daten nochmals angezeigt werden. Erst durch Bestätigen der Frage `"Person in Datenbestand uebernehmen?"` mit `"J"` wird die geänderte Person übernommen.

7.6.1.3 Person löschen

Eine Person kann aus dem Personenstamm gelöscht werden. Zum schnelleren Auffinden der richtigen Person hilft auch hier der Personenfilter (siehe Punkt 7.6.1.2). Wählt man nun eine Person aus der Liste aus, wird diese ohne weitere Abfragen aus dem Personenstamm gelöscht.

7.6.1.4 Person generieren

Eine Anzahl von Personen kann nach verschieden Kriterien generiert und zu den bisherigen Personen hinzugefügt werden (siehe Punkt 7.4.2).

7.6.1.5 Person anzeigen

Die Daten einer Person können angezeigt werden. Auch hier hilft der Personenfilter zum Auffinden der richtigen Person (siehe Punkt 7.6.1.2). Wählt man nun eine Person aus der Liste aus, werden die Daten der Person angezeigt.

7.6.2 Fahrgemeinschaftsdaten

Wählt man im Hauptmenü den Menüpunkt Fahrgemeinschaften, erscheint folgendes Untermenü:

```
-------------------------------
 MOBIDICK - Fahrgemeinschaften
-------------------------------
1. Neuer Teilnehmer
2. Teilnehmer fest eintragen
3. Teilnehmer loeschen
```



```
4. Fahrgemeinschaft eingeben
5. Fahrgemeinschaft aendern
6. Fahrgemeinschaft aufloesen
7. Fahrgemeinschaft bewerten
8. Fahrgemeinschaft anzeigen
9. Fahrgemeinschaft markieren|unmarkieren
```

7.6.2.1 Neuer Teilnehmer

Ein neuer Teilnehmer kann in eine bereits bestehende Fahrgemeinschaft mitaufgenommen werden. Hierzu gibt es die Möglichkeit, den Teilnehmer manuell einzutragen. Hierfür wird zunächst über den Fahrgemeinschaftenfilter eine vorausgehende Einschränkung der anzuzeigenden Fahrgemeinschaften getroffen. Aus den gefundenen Fahrgemeinschaften muß nun eine Fahrgemeinschaft ausgewählt werden, in die der Teilnehmer eingetragen werden soll, wobei der neu einzutragende Teilnehmer ebenfalls aus einer Liste ausgewählt wird. Beantwortet man die Frage `"Person aufnehmen?"` mit `"J"`, so wird dieser Teilnehmer in die Fahrgemeinschaft mitaufgenommen.

7.6.2.2 Teilnehmer fest eintragen

Es besteht die Möglichkeit, die Teilnehmer einer Fahrgemeinschaft fest einzutragen. Fest eintragen bedeutet, daß die bisherigen Teilnehmer einer Fahrgemeinschaft und eine reserviert eingetragene Person einverstanden sind, daß die reserviert eingetragene Person in Zukunft bei der Fahrgemeinschaft mitfährt. Dies wird dadurch vermerkt, daß sie ihren Zustand in fest eingetragen ändert.

7.6.2.3 Teilnehmer löschen

Ein Teilnehmer kann aus einer bestehenden Fahrgemeinschaft gelöscht werden. Dazu muß auch wieder eine Fahrgemeinschaft aus einer Liste ausgewählt werden. Dann wird der zu löschende Teilnehmer nach Bestätigung der Frage `"Person aus Fahrgemeinschaft loeschen?"` mit `"J"` gelöscht.

7.6.2.4 Fahrgemeinschaftendaten

Eine Fahrgemeinschaft kann auch manuell zusammengestellt werden. Dazu muß im Hauptmenü *Fahrgemeinschaften* ausgewählt werden und die folgenden Menüpunkte werden angezeigt:

```
1. Neuer Teilnehmer
2. Teilnehmer fest eintragen
3. Teilnehmer loeschen
4. Fahrgemeinschaft eingeben
5. Fahrgemeinschaft aendern
6. Fahrgemeinschaft aufloesen
```

```
7. Fahrgemeinschaft bewerten
8. Fahrgemeinschaft anzeigen
9. Fahrgemeinschaft markieren|unmarkieren
```

7.6.2.5 Fahrgemeinschaften eingeben

Zur manuellen Eingabe einer Fahrgemeinschaft beginnt man mit dem Menüpunkt *Fahrgemeinschaft eingeben*. Zunächst wird eine Liste der Fahrer angezeigt, aus der man den Fahrer dieser Fahrgemeinschaft ermittelt. Nach Bestätigung der Frage `"Person als FahrerIn uebernehmen?"` mit `"J"` können weitere Teilnehmer für diese Fahrgemeinschaft ausgewählt werden. Wird die Frage `"Fahrgemeinschaft uebernehmen?"` mit `"J"` bestätigt, wird die Fahrgemeinschaft in die Einteilung mitaufgenommen.

7.6.2.6 Fahrgemeinschaften ändern

Es können verschiedene Attribute der Fahrgemeinschaft nachträglich verändert werden (z.B. Anzahl der freien Plätze, Fahrer).

7.6.2.7 Fahrgemeinschaften auflösen

Eine Fahrgemeinschaft kann aufgelöst werden. Das bedeutet, daß diese Fahrgemeinschaft aus der Einteilung gelöscht wird. Dazu wird über eine Liste die entsprechende Fahrgemeinschaft ausgewählt und nach Bestätigung der Frage `"Fahrgemeinschaft aufloesen?"` mit `"J"` wird die Fahrgemeinschaft gelöscht. Die Teilnehmer (Personen) werden nicht gelöscht, sondern haben den Status unvermittelt.

7.6.2.8 Fahrgemeinschaften bewerten

Die Bewertung einer ausgewählten Fahrgemeinschaft wird mit Hilfe der aktuellen Bewertungsfunktion berechnet. Das Ergebnis der Bewertung wird ausgeben.

7.6.2.9 Fahrgemeinschaften ausgeben

Die gesamten Informationen einer Fahrgemeinschaft werden angezeigt: Fahrer, Mitfahrer, Freie Plätze, Automobil, Start- und Zielort, Fahrtroute.

7.6.2.10 Fahrgemeinschaften markieren bzw. unmarkieren

Eine ausgewählte Fahrgemeinschaft kann markiert bzw. unmarkiert werden. Zunächst muß ausgewählt werden ob markiert oder unmarkiert, danach die entsprechende Fahrgemeinschaft. Wird die Frage `"Fahrgemeinschaft markieren?"` mit `"J"` beantwortet, ist die Fahrgemeinschaft markiert (analog mit unmarkieren). Prinzipiell ist eine Fahrgemeinschaft,

wenn sie neu angelegt wird, unmarkiert. Das bedeutet, falls eine neue Einteilungsberechnung stattfindet, können Teilnehmer aus dieser Fahrgemeinschaft herausgenommen und neu eingeteilt werden. Will man dies vermeiden, so muß diese Fahrgemeinschaft markiert werden, um diese Fahrgemeinschaft in ihrem bisherigen Zustand zu belassen.

7.7 Einteilung berechnen lassen

Zur Berechnung einer Einteilung muß man sich zunächst in das Menü *Vermittlung* begeben und dort den Punkt *Einteilung* auswählen. Dann muß man sich entscheiden, ob die Einteilung mit einem heuristischen, einem optimalen oder einem inkrementellen Algorithmus berechnet werden soll. Hat man die Entscheidung durch Auswahl des entsprechenden Menüpunkts eingestellt, wird ein Menü angezeigt, in dem es die Möglichkeit gibt, eine Berechnung zu starten oder den Algorithmus zur Einteilungsberechnung zu wechseln. Wählt man den Punkt *Algorithmus wechseln*, so wird eine Liste der verfügbaren Algorithmen angezeigt. Diese sind entsprechend der oben getroffenen Entscheidung nur heuristische, optimale oder inkrementelle Algorithmen. Nach Auswahl des gewünschten Algorithmus gelangt man wieder in das aufrufende Menü. Dort kann man mit dem Punkt *Algorithmus starten* die Berechnung einer Einteilung mit dem aktuell eingestellten Algorithmus beginnen. Abhängig vom gewählten Algorithmus werden noch verschiedene Fragen gestellt, die sich auf Parameter des Algorithmus, Angaben zur Güte, der zu berechnenden Einteilung oder den Namen der neuen Einteilung beziehen können. Bei inkrementellen Algorithmen wird keine neue Einteilung angelegt, sondern die aktuelle Einteilung verändert. Wurde bei der Berechnung mit einem optimalen oder heuristischen Algorithmus keine Einteilung gefunden, so wird der Benutzer mit einer Systemmeldung darüber informiert.

7.8 Wegsuche

7.8.1 Einzelwegsuche

Das System bietet die Möglichkeit, einen kürzesten Weg zwischen zwei Kanten suchen zu lassen. Diese Einzelwegsuche steht in keinem Zusammenhang zu der Einteilungsberechnung oder der Datenverwaltung von Mobidick. Nach Auswahl des Menüpunkts *Einzelwegsuche* im Menü *Wegsuche* kann im Graphenviewer mit der Maus die Start- und Zielkante der Wegsuche ausgewählt werden. Zu der nun folgenden Wegberechnung wird der im Menüpunkt *Auswahl des Algorithmus* eingestellte Wegsuchealgorithmus verwendet. Am Ende der Berechnung wird der berechnete Weg als Kantenliste am Bildschirm angezeigt.

7.8.2 Wegsuche n-mal

Zum Laufzeitvergleich verschiedener Wegsuchealgorithmen bietet Mobidick die Möglichkeit, n Wege mit zufällig gewählten Start- und Zielorten zu berechnen und die dafür benötigte Laufzeit auszugeben. Zunächst kann mit dem Menüpunkt *Auswahl des Algorithmus* der Wegsuchealgorithmus eingestellt werden, mit dem die Wegsuche durchgeführt werden soll. Dann wird nach Auswahl des Menüpunkts *Wegsuche n-mal* die Anzahl der zufälligen Wege abgefragt und die Wegsuche gestartet. Nachdem die Berechnungen durchgeführt wurden, wird die benötigte Laufzeit in Sekunden angezeigt.

7.9 Bewertungsfunktion

Die Bewertungsfunktion besitzt verschiedene Parameter, die im Untermenü *Bewertungsfunktion* des Hauptmenüs eingegeben und verändert werden können. Die einzelnen Parameter haben folgende Bedeutungen:

max. Arbeitszeitabstand: gibt an, wie groß der Abstand in Minuten zwischen der längsten und der kürzesten Arbeitszeit der Teilnehmer höchstens sein darf.

max. Umweg: gibt an, wieviele Minuten der Weg des Fahrers länger werden darf, wenn er die Teilnehmer abholt und zu ihrem Arbeitsplatz bringt.

max. Abneigungskonflikte: eine Schranke für die Anzahl der Konflikte zwischen den Teilnehmern einer Fahrgemeinschaft und ihren Abneigungslisten. Wenn es mehr Konflikte gibt, wird die Bewertung 0, was bedeutet, daß diese Fahrgemeinschaft während der Einteilungsberechnung nicht gebildet wird.

max. Bewertung bezüglich Eigenschaften: ist ein reeller Wert zwischen 0 und 1, der angibt, wie schlecht die Bewertung bezüglich der Personeneigenschaften höchstens sein darf.

Heuristische Umwegberechnung: gibt an, ob bei der Berechnung des kürzesten Umwegs eines Fahrers die optimale Methode oder eine Heuristik angewendet werden soll.

Weitere Parameter sind die Gewichte, mit denen die Teilergebnisse der Fahrgemeinschaftsberechnung in das Endergebnis eingehen sollen. Die Gewichte sind ganze Zahlen und müssen im Bereich zwischen 0 und $\frac{MaxInt}{\#Gewichte}$ liegen.

7.10 Systemmeldungen

Fehlercode	**Beschreibung** ***mögliche Lösung***
ALLGEMEINER_FEHLER	Es ist ein Fehler aufgetreten. *Der aufgetretene Fehler wurde nicht näher spezifiziert. Versuchen Sie es noch einmal oder starten Sie das System neu.*
ZU_GROSS	Die Zahl ist zu groß. *Geben Sie eine kleinere Zahl ein.*
KEINE_ZAHL	Das Ergebnis ist keine Zahl. *Geben Sie eine Zahl ein.*
KEINE_AUSGEWAEHLT	Es wurden keine Daten ausgewählt. *Wählen Sie die entsprechenden Datensätze zunächst aus.*
KEINE_LISTE	Es gibt keine Personen im Personenverwalter. *Geben Sie Personen ein oder laden Sie eine Personendatei.*
NICHT_VOLL	Die Personendaten sind nicht vollständig. *Füllen Sie alle Datenfelder, die mit einen Stern (*) gekennzeichnet sind.*
NICHT_IM_BEREICH	Die Gewichtung des Wunschs ist nicht im Bereich. *Eine Gewichtung eines Wunschs muß zwischen 0 und 10 liegen.*
KEINE_EINTEILUNG	Die Einteilung ist nicht vorhanden. *Erzeugen Sie zunächst eine Einteilung oder laden Sie eine Datei, in der eine vorhanden ist.*
KEINE_EIN_LISTE	Es ist keine Einteilungsliste mit Einteilungen vorhanden. *Erzeugen Sie zunächst eine Einteilung oder laden Sie eine Datei, in der eine vorhanden ist.*
KEINE_FGM	Es ist keine Fahrgemeinschaft vorhanden. *Erzeugen Sie zunächst eine Einteilung mit Fahrgemeinschaften oder laden Sie eine Datei, in der eine vorhanden ist.*
KEINE_FGM_LISTE	Es ist keine Fahrgemeinschaftenliste vorhanden. *Erzeugen Sie zunächst eine Einteilung mit Fahrgemeinschaften oder laden Sie eine Datei, in der eine vorhanden ist.*
KEINE_PER_LISTE	Es sind keine vermittelten Personen vorhanden. *Erzeugen Sie zunächst eine Einteilung mit Fahrgemeinschaften oder laden Sie eine Datei, in der eine vorhanden ist.*
KEIN_TEILNEHMER	Es ist kein Teilnehmer vorhanden. *Setzen Sie einen Teilnehmer in die Fahrgemeinschaft.*
KEINE_TEILNEHMER_LISTE	Es ist keine Teilnehmerliste vorhanden. *Setzen Sie einen Teilnehmer in die Fahrgemeinschaft.*

TEILNEMER_DA	Der Teilnehmer ist in der aktuellen Einteilung schon vorhanden. *Löschen Sie zunächst den Teilnehmer aus der Einteilung, bevor Sie ihn wieder einsetzen.*
KEINE_MARK_FGM	Es sind keine markierten Fahrgemeinschaften vorhanden. *Sie können keine Fahrgemeinschaften unmarkieren, wenn keine markiert sind.*
KEINE_UNMARK_FGM	Es sind keine unmarkierten Fahrgemeinschaften vorhanden. *Sie können keine Fahrgemeinschaften markieren, wenn keine unmarkiert sind.*
KEIN_FAHRER	Es gibt keine Angaben über den Fahrer. *Tragen Sie einen Fahrer in die Fahrgemeinschaft ein.*
KEIN_FREIER_PLATZ	Es gibt keinen freien Platz in der Fahrgemeinschaft. *Löschen Sie zunächst einen anderen Teilnehmer, bevor Sie einen hineinsetzen.*
KEINE_FAHRTROUTE	Es ist keine Fahrtroute vorhanden. *Lassen Sie die Fahrgemeinschaft zunächst bewerten.*
KEIN_ZEITPLAN	Es ist kein Zeitplan vorhanden. *Lassen Sie die Fahrgemeinschaft zunächst bewerten.*
KEINE_PLAETZE	Es sind keine Plätze vorhanden. *Die Fahrgemeinschaft ist voll.*
KEINE_PERSONENDATEI	Es ist keine Personendatei vorhanden. *Erstellen oder laden Sie eine Personendatei.*
NICHT_GEOEFFNET	Es ist keine Personendatei geöffnet. *Öffnen Sie eine Personendatei.*
NICHT_GESPEICHERT	Die Personendatei wurde geändert aber noch nicht gespeichert. *Speichern Sie die Personendatei.*
FALSCHE_DATEIART	Die Methode arbeitet nicht mit dieser Dateiart. *Versuchen Sie es mit einer anderen Datei.*
KEIN_SCHNITTPUNKT	Die Zeitintervalle der Personen haben keinen Schnittpunkt. *Verwenden Sie eine andere Personenmenge.*
KEINE_AKT_BEW	Es wurde noch keine aktuelle Bewertungsfunktion ausgewählt. *Wählen Sie eine Bewertungsfunktion als aktuelle aus.*
DATEI_NICHT_GEFUNDEN	Die Datei existiert nicht. *Versuchen Sie es mit einer anderen Datei.*

BEW_EXISTIERT_NICHT	Es wurde eine Bewertungsfunktions-ID übergeben, die in der Bewertungsliste nicht enthalten ist. *Sie versuchen, eine Bewertungsfunktion zu löschen, die nicht existiert.*
PERS_MENGE_ZU_KLEIN	Die zu bewertende Personenmenge ist kleiner als 2. *Bewerten Sie eine größere Personenmenge.*
KEIN_AKT_ALGOR	Es wurde entweder kein aktueller optimaler, inkrementeller oder heuristischer Algorithmus ausgewählt. *Wählen sie einen Algorithmus aus.*
TABELLEN_NICHT_GELADEN	Die Tabellen für Randknoten- oder Rathaussuche wurden noch nicht geladen. *Starten Sie das System neu.*
STARTKANTE _EXISTIERT_NICHT	Die Startkante für die Wegsuche existiert nicht. *Sie haben es geschafft, eine ungültige Startkante einzugeben. Versuchen Sie es erneut.*
ZIELKANTE _EXISTIERT_NICHT	Die Zielkante für die Wegsuche existiert nicht. *Sie haben es geschafft, eine ungültige Zielkante einzugeben. Versuchen Sie es erneut.*
KEIN_WEG	Es wurde kein Weg zwischen Start- und Zielkante gefunden. *Die Verkehrsdaten sind unvollständig oder eine der Kanten „hängt in der Luft“. Versuchen sie zwei andere Kanten.*
UNDEFINIERT	Dieser Fehler wurde nicht definiert. *Der aufgetretene Fehler existiert nicht. Dies ist ein Fehler im System.*
NICHT_FERTIG	Diese Methode ist noch nicht implementiert. *Die ausgewählte Funktion ist im Lieferumfang nicht enthalten.*

Kapitel 8

Programmierung

8.1 Installation

8.1.1 Installation

Folgende Schritte führen bei fehlerfreier Ausführung zu einem funktionierenden Mobidick-System.

1. Besorgen Sie sich das Programmpaket Mobidick-aktuelle_Version.tar.gz.

2. Erzeugen Sie ein Verzeichnis und kopieren Sie es dort hin.

3. Packen Sie das Paket mit gunzip und tar aus.

4. Stellen Sie sicher, daß die LEDA-Bibliothek in einem der folgenden Verzeichnisse zu finden ist.

   ```
   /usr/local/include/LEDA-3.4.2
   /usr/local/gnu/lib/g++-include/LEDA-3.4.2
   ```

   Fall dies nicht der Fall ist, teilen Sie *configure* mit, wo sie zu finden ist.

5. Rufen Sie *configure* auf.

6. Rufen Sie *make* auf.

7. Im Verzeichnis befindet sich nun eine ausführbare Datei namens „mobidick“. Rufen Sie diese auf.

8. Willkommen im System.

8.1.2 LEDA-Bibliothek

LEDA (Library of Efficient Data types and Algorithms) ist eine Bibliothek mit Datentypen und Algorithmen, die von Kurt Mehlhorn, Stefan Näher und Christian Uhrig entwickelt wurde. Diese drei hatten die Absicht, für Nichtexperten einen Zugang zu schwierigen Algorithmen und gebräuchlichen Datentypen so einfach wie möglich zu machen.

Die Haupteigenschaften der LEDA-Bibliothek sind folgende :

- LEDA bietet eine beträchtliche Sammlung von Datentypen und Algorithmen, die sehr leicht von Nichtexperten genutzen werden können.
- LEDA bietet eine wissenschaftliche und lesbare Spezifikation für jeden erwähnten Datentyp und Algorithmus, der sich auf das Wesentliche konzentiert.
- Für viele effizienten Datenstrukturen ist der Zugriff auf eine gewünschte Information sehr wichtig. In LEDA wird dafür das item-Konzept verwendet. Das item repräsentiert die Position der Information, mit dem dann der Inhalt gelesen werden kann.
- LEDA beinhaltet eine effiziente Implementierung für jeden Datentyp.
- LEDA bestitzt einen komfortablen Graphendatentyp. Dieser bietet Standardberechnungen, wie z.B. „for all nodes v of graph G do“ oder „for all neighbours w of v do“. Weiterhin können Arrays und Matrizen mit Knoten und Kanten angelegt werden und vieles mehr. Mit Hilfe dieses Graphendatentyps können Graphenprobleme sehr einfach programmiert werden.
- LEDA wurde mit einer C++ Klassenbibliothek implementiert und kann mit fast jedem C++ Compiler benutzt werden.
- LEDA ist nicht public domain, ist aber für universitäre Forschung und Entwicklung frei verfügbar.
- LEDA ist erhältlich auf folgendem FTP-Server: ftp.mpi-sb.mpg.de im Verzeichnis /pub/LEDA Weitere Informationen auf der Homepage: www.mpi-sb.mpg.de .

Für das Projekt Fahrgemeinschaft wird die LEDA-Version R 3.4.2 benutzt. Die allgemeine Benutzung der Datentypen und Algorithmen setzt voraus, daß entsprechende Bibliotheken in das System hinzugefügt werden. Aus diesem Grund werden die Bibliotheken list.h, string.h und dictionary.h in constant.h eingefügt und sind allgemein verfügbar. Spezielle Bibliotheken werden nur in den entsprechenden Headerfiles hinzugefügt.

Weitere Informationen zur Benutzung der einzelnen Bibliotheken können aus dem Handbuch „The LEDA User Manual“ entnommen werden [8].

8.1.3 Voraussetzungen

Mobidick läuft unter Solaris 2.5 und X11R5. Die Eingabe erfolgt über Tastatur und Maus, die Ausgabe auf den Bildschirm. Weitere Peripherie wird nicht benötigt. Der Benutzer sollte mit dem Umgang von Rechnern hinlänglich vertraut sein.

8.2 Programmierrichtlinien

Die hier verwendeten Richtlinien orientieren sich am Dokument von ELLEMTEL [5] und werden nun im einzelnen vorgestellt:

1. Jedesmal, wenn eine Regel verletzt wird, muß es klar dokumentiert werden.
2. Jede Implementierungsdatei muß einen Header besitzen, welcher nähere Informationen über den Code angibt.
3. Die Namen von Variablen, Konstanten und Funktionen beginnen mit Kleinbuchstaben.
4. Die Namen von abstrakten Datentypen, Strukturen und `typedef` beginnen mit Großbuchstaben.
5. Namen, die sich aus mehreren Wörtern zusammensetzen, werden mit _ getrennt und nur der erste Buchstabe des ersten Wortes wird groß geschrieben.
6. Die public, protected und private Sektionen einer Klasse werden in dieser Reihenfolge angegeben.
7. Ein Include-File sollte nie mehr als eine Klasse beinhalten.
8. `//` werden für Kommentare benutzt.
9. `#include "filename.h"` wird für benutzereigene Include Files benutzt.
10. `#include <filename.h>` wird für Bibliotheks Include Files benutzt.
11. Variablennamen sollen nach ihrem Nutzen gewählt werden.
12. Globale Variablen, Konstanten und Typdefinitionen sollten in einer eigenen Klasse gekapselt werden.
13. Funktionen mit vielen Argumenten sollten vermieden werden.
14. Lange und komplexe Funktionen sollten vermieden werden.
15. Jeder Variablen muß zunächst ein Wert zugeordnet werden, bevor sie benutzt werden kann.

8.3 Schnittstellen

8.3.1 Strukturen

Unter einer Struktur versteht man i. A. die Zusammenfassung von einer oder mehreren Variablen zu einer Einheit, die untereinander auch unterschiedliche Datentypen besitzen dürfen. Sie sind in der Datei *constants.h* definiert. Hier gibt es für jedes Datenobjekt mindestens eine Struktur. Meist gibt es noch eine zweite, falls man einen Filter über die Datensätze legen kann. Viele der Einträge einer Struktur sind vom Typ *Optional*. Dieses Template enthält einen Eintrag vom übergebenen Typ oder definiert Nichts. Man kann mit den Methoden `typ.nichts()` und `typ.setze_wert(wert)` den Wert setzen oder als “Nichts” definieren. Mit `typ.ist_gesetzt()` kann man anfragen, ob ein Wert ungleich Nichts gesetzt ist. Mit `typ.gib_wert()` bekommt man den gesetzten Wert. Falls kein Wert gesetzt ist, ist das Ergebnis undefiniert.

8.3.2 Klassen

Namen für Klassen beginnen mit einem Großbuchstaben. Sollte ihr Name aus mehreren Wörtern bestehen, werden diese mit “_” aneinandergehängt. Alle nachfolgenden Wörter fangen klein an.

8.3.3 Methoden

Im Grobentwurf werden die einzelnen Verwalter, der Fürsorger, die Verkehrsgraphen, die Datei-Auswahl und die Menüs festgelegt. Kommunikation, die zwischen diesen Teilen stattfindet, ist vereinheitlicht. Innerhalb der Blöcke darf von den Regeln abgewichen werden. Generell entsprechen die öffentlichen Methoden von Klassen den folgenden Regeln, auch wenn sie innerhalb von Blöcken liegen.

Einheitliche Methoden entsprechen folgenden Regeln:

1. Es gibt einen Rückgabewert vom Typ *Fehlercode*.
2. Wertparameter stehen am Anfang, Referenzparameter am Ende.
3. Attribute von Objekten werden in Form von structs übergeben (siehe constants.h).

Die Namen von Methoden sind deutsch, falls mehrere Wörter miteinander verknüpft werden, werden diese alle kleingeschrieben und durch “_” getrennt.

8.3.4 constants.h

Typen und Konstanten, die als Parameter übergeben werden, oder in mehreren Klassen Verwendung finden, liegen in dieser Datei. Sie wird von jedem Modul importiert.

```
// Dateiname:    constants.h
// Autor:        Projektgruppe FGM
// erstellt:     weissnichmehr97
// geaendert:    siehe RCS
// Zweck:        Sammlung der globalen Typen,Konstanten und
//               Variablen
// Bemerkungen:  wird in den header-files standardmaessig
//               eingebunden, wenn das perl-script benutzt
//               wird


#include <stdio.h>
#include <LEDA/list.h>
#include <LEDA/string.h>
#include <LEDA/dictionary.h>
#include <LEDA/integer.h>
#include "Optional.h"
#include "/home/fgm/implementierung/work/personen/Zeit.h"

#ifndef CONSTANTS_H
#define CONSTANTS_H


#define name_der_voreinstellungen "voreinstellungen.mbd"

typedef int Person_ID;
typedef int FGM_ID;
typedef int Einteilungs_ID;
typedef int Bewertungs_ID;
typedef int Kanten_ID;
typedef int Weg_ID;
typedef int Algorithmus_ID;
typedef double Bewertung;
typedef list<Person_ID> Personen_liste;
typedef list<Person_ID> Personenliste;
typedef list<Kanten_ID> Strassenliste;
typedef list<Kanten_ID> Weg;  // darf nicht sortiert sein!
typedef list<Algorithmus_ID> Algorithmenliste;
typedef list<Bewertungs_ID> Bewertungsliste;
typedef list<Einteilungs_ID> Einteilungsliste;
typedef list<FGM_ID> FGMliste;
typedef list<Zeit *> Zeitliste;

// Kein Auswahlmenue darf mehr als 60 Zeilen haben
const int MAX_ZEILEN = 60;
// Kein Datenmenue darf mehr als 60 Daten haben
const int MAX_DATEN  = 60;
// Defaultgroesse fuer Fahrgemeinschaften (maximal)
const int MAX_TEILNEHMER_DEFAULT = 4;
enum Aktionen {     // Menue-Aktionen
```



```
  // Person anzeigen
  PERS_ANZEIGEN,
  // Person anlegen
  NEUE_PERSON,
  // Hilfe wurde ausgewaehlt
  HILFE,
  // Programmende
  ENDE,
  // Hauptmenue
  HAUPT,
  // Zurueck ins letzte Menue
  ZURUECK,
  // Fehlerhafte Eingabe
  TUNIX,
  // Menuepunkte Hauptmenue
  HAUPT_1, HAUPT_2, HAUPT_3, HAUPT_4, HAUPT_5, HAUPT_6,
  // Menuepunkte Dateien
  DATEIEN_1, DATEIEN_2, DATEIEN_3, DATEIEN_4,
  // Menuepunkte Dateien->Personendateien
  PERSON_DAT_1, PERSON_DAT_2, PERSON_DAT_3, PERSON_DAT_4,
  PERSON_DAT_5, PERSON_DAT_6,
  // Menuepunkte Dateien->Fahrgemeinschaftseinteilung
  EINTEILUNG_DAT_1, EINTEILUNG_DAT_2, EINTEILUNG_DAT_3,
  EINTEILUNG_DAT_4,
  // Menuepunkte Dateien->Bewertungsfunktionen
  BEWERTUNG_DAT_1, BEWERTUNG_DAT_2, BEWERTUNG_DAT_3,
  // Menuepunkte Personen
  PERSON_1, PERSON_2, PERSON_3, PERSON_4, PERSON_5,
  // Menuepunkte Fahrgemeinschaften
  FGM_1, FGM_2, FGM_3, FGM_4, FGM_5, FGM_6, FGM_7, FGM_8,
  FGM_9,
  // Menuepunkte Fahrgemeinschaften->Neuer Teilnehmer
  NEUER_TEILNEHMER_1, NEUER_TEILNEHMER_2,
  NEUER_TEILNEHMER_3,
  // Menuepunkte Vermittlung
  VERMITTLUNG_1, VERMITTLUNG_2, VERMITTLUNG_3,
  VERMITTLUNG_4, VERMITTLUNG_5, VERMITTLUNG_6,
  VERMITTLUNG_7, VERMITTLUNG_8, VERMITTLUNG_9,
  // Menuepunkte Algorithmenauswahl
  ALGO_AUSWAHL_1, ALGO_AUSWAHL_2, ALGO_AUSWAHL_3,
  // Menuepunkte Algorithmenstart
  ALGO_START_1, ALGO_START_2,
  ALGO_OPTIMAL,ALGO_HEURISTISCH,ALGO_INKREMENTELL,
  WECHSEL_OPTIMAL, WECHSEL_HEURISTISCH, WECHSEL_INKREMENTELL,
  // Menuepunkte Bewertungsfunktionen (Hauptmenue)
  BEWERTUNGFKTEN_1, BEWERTUNGFKTEN_2, BEWERTUNGFKTEN_3,
  BEWERTUNGFKTEN_4,
  // Menuepunkte Wegsuche
  WEGE_1, WEGE_2, WEGE_3, WEGE_4, WEGE_5
};
```

```
enum Meldungsart    {VERMITTLUNG,WEGSUCHE};
enum Algorithmenart {HEURISTISCH,INKREMENTELL,OPTIMAL};
enum Dateiart       {BEWERTUNGSFUNKTION,PERSONENDATEI,
     VERKEHRSGRAPH,KNOTEN,KANTEN,
     REGIONEN,VOREINSTELLUNGEN};
enum Komfortklasse  {KLEINWAGEN,MITTELKLASSE,
     GEHOBENE_KLASSE};
enum Geschlecht     {MAENNLICH,WEIBLICH};
enum Raucher        {NICHTRAUCHER,RAUCHER};
enum Status         {RESERVIERT,FEST};
enum Abbruch        {JA,NEIN,SPEICHERN};
enum Dateizugriff   {LESEND,SCHREIBEND};

enum Boolean {FALSE,TRUE};

enum Fehlercode {
  //  Alle: Alles hat geklappt
  OK=0,
  //  Wenn eine Methode noch nicht implementiert ist
  NICHT_FERTIG,
  //  Ein noch nicht naeher behandelter Fehler
  ALLGEMEINER_FEHLER,
  //  MV intern: Die Zahl ist zu gross
  ZU_GROSS,
  //  MV intern: Es wurde keine Zahl eingegeben
  KEINE_ZAHL,
  //  MV intern: Es wurden keine Daten ausgewaehlt
  KEINE_AUSGEWAEHLT,
  //  PV: Keine Person
  KEINE_PERSON,
  //  PV: Keine Liste
  KEINE_LISTE,
  //  PV: Personendaten nicht vollstaendig
  NICHT_VOLL,
  //  PV: Attribute wurden nicht gesetzt
  //  LS: Name oder Pfad einer Dateiart sind nicht
  //      gesetzt
  NICHT_IM_BEREICH,
  //  PV: Bei Wunsch, Gewichtung nicht im Bereich
  NICHT_GESETZT,
  //  EV: Einteilung nicht vorhanden
  //  PV: Bei Wunsch, Eigenschaft nicht vorhanden
  KEINE_EINTEILUNG,
  //  EV: Keine Einteilungsliste mit Einteilungen
  //      vorhanden
  KEINE_EIN_LISTE,
  //  EV: Keine Fahrgemeinschaft vorhanden
  KEINE_FGM,
  //  EV: Keine FGM Liste nicht vorhanden
```



```
KEINE_FGM_LISTE,
//  EV: Keine vermittelten Personen vorhanden
KEINE_PER_LISTE,
//  EV: Kein Teilnehmer vorhanden
KEIN_TEILNEHMER,
//  EV: Keine Teilnehmerliste vorhanden
KEINE_TEILNEHMER_LISTE,
//  EV: Keine Bewertungs_id vorhanden
KEINE_BEW_ID,
//  EV: Teilnehmer ist in der aktuellen Einteilung
//      schon vorhanden
TEILNEMER_DA,
//  EV: Keine markierten FGMs vorhanden
KEINE_MARK_FGM,
//  EV: Keine unmarkierten FGMs vorhanden
KEINE_UNMARK_FGM,
//  EV: Keine Angabe ueber Fahrer
KEIN_FAHRER,
//  EV: Kein freier Platz in der FGM
KEIN_FREIER_PLATZ,
//  EV: Keine Fahrtroute vorhanden
KEINE_FAHRTROUTE,
//  EV: Kein Zeitplan vorhanden
KEIN_ZEITPLAN,
//  EV: Keine Plaetze vorhanden
KEINE_PLAETZE,
//  LS: Keine Personendatei geladen
KEINE_PERSONENDATEI,
//  LS: Eine Datei, auf die zugegriffen werden
//      soll, ist nicht geoeffnet
NICHT_GEOEFFNET,
//  LS: Personendatei geaendert, aber nicht
//      gespeichert
NICHT_GESPEICHERT,
//  LS: Methode arbeitet nicht mit dieser Dateiart
FALSCHE_DATEIART,
//  AV: Startkante fuer Wegsuche ex. nicht
STARTKANTE_EXISTIERT_NICHT,
//  AV: analog
ZIELKANTE_EXISTIERT_NICHT,
//  AV: es ex. kein Wegsuchealgorithmus mit
//      dieser ID
FALSCHE_WEGALGO_ID,
//  AV: Datei existiert nicht
DATEI_NICHT_GEFUNDEN,
//  AV: es wurde kein Weg zwischen Start- und
//      Zielkante gefunden
KEIN_WEG,
//  AV: Tabellen fuer Randknoten- oder
//      Rathaussuche nicht geladen
```

```
  TABELLEN_NICHT_GELADEN,
  //  AV: es wurde kein aktueller Algorithmus
  //      festgelegt
  KEINE_NEUE_EINTEILUNG,
  //      Wird von den Einteilungsalgorithmen
  //      zurueckgegeben, wenn sie nach der
  //      Berechnung keine neue Einteilung
  //      angelegt haben.
  KEIN_AKT_ALGOR,
  //  BV: Es wurde noch keine aktuelle
  //      Bewertungsfunktion ausgewaehlt.
  KEINE_AKT_BEW,
  //  BF: Bei der Berechnung der Ankunftszeit am
  //      Arbeitsort wird festgestellt, dass die
  //      Zeitintervalle der Personen keinen
  //      Schnittpunkt haben.
  KEIN_SCHNITTPUNKT,
  //  BF: Die zu bewertende Personenmenge ist
  //      kleiner als 2.
  PERS_MENGE_ZU_KLEIN,
  //  BF:
  Z_MAX_UEBERSCHRITTEN,
  //  BF: der Weg besucht Zielorte vor den
  //      zugehoerigen Startorten
  KEIN_GUELTIGER_WEG,
  //  BF: der beschriebene Weg existiert im
  //      Verkehrsgraphen nicht
  KEIN_WEG_GEFUNDEN,
  //  BV: Es wurde eine Bewertungsfunktions_ID
  //      uebergeben, die in der Bewertungsliste
  //      nicht enthalten ist.
  BEW_EXISTIERT_NICHT,
  //  BF:
  ZU_VIELE_ABNEIGUNGEN,
  //  OPTI-Algo: Es gibt keine Fahrer in der Personenmenge
  KEINE_FAHRER,
  //  OPTI-Algo intern: Die Fahrgemeinschaft konnte nicht
  //                    gebildet werden
  FGM_PUTT,
  //  Vorletzter Fehlercode: fuer Doktor
  UNDEFINIERT,
  //  Immer der letzte Fehlercode! (fuer Doktor)
  anzahl_fehler
};

//Entstehungsdatum einer Fahrgemeinschaft
struct FGM_Datum
{
  int sekunden;
  int minuten;
```



```
  int stunden;
  int tag;
  int monat;
  int jahr;
};

//Daten einer Fahrgemeinschaft
struct FGMdaten
{
  // die ID
  FGM_ID id_fgm;
  // Markierung fuer Aufloesen
  Boolean markierung_fgm;
  // der Fahrer
  Optional<Person_ID> fahrer;
  // Anzahl der noch freien Plaetze
  // 0<=freie_plaetze<=autoplaetze-#teilnehmer
  Optional<int> frei_plaetze;
  // Entstehungsdatum
  FGM_Datum datum_fgm;
  // Datum der letzten Aenderung
  // bei Entstehung=datum_fgm
  FGM_Datum aenderung_fgm;
  // die Fahrtroute
  Weg fahrtroute;
  // die Teilnehmer
  Personenliste teilnehmer;
  // der Status der Teilnehmer
  dictionary <Person_ID,Status> status_teilnehmer;
  // die wichtigen Zeitpunkte auf der Fahrtstrecke
  Zeitliste fahrzeiten;
};

struct Uhrzeit
{
  int stunden;
  int minuten;
  int sekunden;
};

// Arbeitszeiten, Ankunfts- und Rueckfahrtzeiten, ...
struct Zeitintervall
{
  Uhrzeit zeit_von;
  Uhrzeit zeit_bis;
};

// Filterdaten fuer Fahrgemeinschaften
struct FGM_filter
{
```

```
  Optional<FGM_ID> fgm_id;
  Optional<int> min_plaetze;
  Optional<int> max_teilnehmer;
  Optional<Komfortklasse> minkomfort;
  Optional<int> startort;
  Optional<int> radius_startort;
  Optional<int> zielort;
  Optional<int> radius_zielort;
  Optional<Zeitintervall> startzeit;
  Optional<Zeitintervall> zielzeit;
};

// Adressteil der Personendaten
struct Adresse
{
   string strasse;
   string hausnummer;
   int postleitzahl;
   string wohnort;
};




// Personen enthalten Informationen zu ihren Autos
struct Autodaten
{
  Optional<Komfortklasse> komfort;
  Optional<int> baujahr;
  Optional<int> plaetze;
};

// Daten der Person, ihren Fahrtweg betreffend
struct Wegdaten
{
  Optional<struct Adresse> startort;
  Optional<int> startkoordinaten;
  Optional<struct Adresse> zielort;
  Optional<int> zielkoordinaten;
  Optional<struct Zeitintervall> anzeit;
  Optional<struct Zeitintervall> rueckzeit;
  Optional<struct Uhrzeit> arbeitsdauer;
};

// Daten der Person, ihren Fahrtweg betreffend
struct Wunschdaten
{
  Optional<Personenliste> abneigung_zu_personen;
  Optional<Personenliste> zuneigung_zu_personen;
  Optional<Geschlecht> geschlecht_wunsch;
```



```
  Optional<int> geschlecht_gewichtung;
  Optional<Raucher> raucher_wunsch;
  Optional<int> raucher_gewichtung;
};

struct Datum
{
  int tag;
  int monat;
  int jahr;
};

struct Telefonnummer
{
  string vorwahl; // z.B.: 0711 int reicht hier nicht
  string rufnummer;
};

// die Systemform der Personendaten, dieser struct
// wird hin und hergeschickt...
struct Personendaten
{
  struct Autodaten automobil;
  struct Wegdaten  weginfo;
  struct Wunschdaten wunschinfo;
  Person_ID  personen_id;
  Optional<string> name;
  Optional<string> vorname;
  Optional<Geschlecht> geschlecht;
  Optional<struct Datum> geburtsdatum;
  Optional<struct Adresse> wohnsitz;
  Optional<struct Telefonnummer> telefon;
  Optional<struct Telefonnummer> fax;
  Optional<string> email;
  Optional<Boolean> fahrer;
  Optional<Raucher> raucher;
};

// die Systemform der Personenfilterdaten, dieser struct
// wird hin und hergeschickt...
struct Per_Filterdaten
{
  Optional<int> personen_id;
  Optional<string> name;
  Optional<string> vorname;
  Optional<struct Zeitintervall> anzeit;
  Optional<struct Zeitintervall> rueckzeit;
  Optional<string> startstrasse;
  int startkoordinaten;
  int startradius;
```

```
  Optional<string> zielstrasse;
  int zielkoordinaten;
  int zielradius;
  Optional<Boolean> fahrer;
};

// Dieser struct wird zur Generierung einer neuen
// Personenmenge benutzt.
struct GenerierDaten
{
  Optional<int> personen_anzahl;
  Optional<int> anteil_fahrer;
  Optional<Boolean> startort_gleichverteilt;
  //Optional<struct rechteck> startort_rechteck;
  Optional<int> startort_fest;
  Optional<Boolean> zielort_gleichverteilt;
  //Optional<struct rechteck> zielort_rechteck;
  Optional<int> zielort_fest;
  Optional<struct Zeitintervall> ankunftszeit;
  // In Sekunden und werden im Menue von
  // Stunden auf Sekunden umgewandelt
  Optional<int> an_intervall_laenge;
  Optional<struct Zeitintervall> rueckfahrtzeit;
  // In Sekunden und werden im Menue von
  // Stunden auf Sekunden umgewandelt
  Optional<int> ziel_intervall_laenge;
};

// Dieser struct wird zur uebermittlung der
// Bewertungsfunktionsparameter zum
// Menue und zum Lader/Speicher benutzt.
struct Bew_fkt_parameter
{
  int z_max;
  int u_max;
  int na_max;
  double e_max;
  int gewicht_z;
  int gewicht_u;
  int gewicht_na;
  int gewicht_ne;
  int gewicht_e;
  int gewicht_p;
  string name;
  Boolean heuristik_verwenden;
};


//Dieser struct wird zur Zwischenspeicherung von FGM-Daten
//bei der Einteilungsberechnung durch den MM-Algorithmus
```



```
//verwendet.
struct MM_FGM_daten
{
  Person_ID fahrer;
  Personenliste teilnehmerliste;
  Weg weg;
  Zeitliste zeitplan;
};


/*************************************/
/* Konstanten fuer das Modul Wegsuche */
/*************************************/

/* groesster Tabelleneintrag in kompletter Wegetabelle */
/* fuer Entfernung unendlich                            */
const unsigned short MAXDIST    = 65535;

/* Eintrag in Indextabelle zur Wegetabelle fuer den    */
/* Fall, dass Knotennr. nicht existiert                 */
const unsigned short NODE_UNDEF = 65535;

/* Parameter fuer A*-Suche */
const double RESTFAKTOR = 3.6/100;

/* Parameter fuer Rathaussuche */
const int NACHBAR_RATHAEUSER = 3;
const int MAX_RATHAEUSER_PRO_ZELLE = 2;
const int ZELLEN_PRO_ZEILE = 25;

/* Geschwindigkeiten der Strassenklassen 0..8 in km/h */
const int geschw_klasse[] = {100, 60, 60, 40, 40,
     25, 20, 20, 20};

/* Pfade fuer die Verkehrsdaten */
// Knoten-, Kanten-, Regionenfile und Tabellen
// fuer hierarchische Graphen
#define VERKEHRSDATENPFAD \
"/home/fgm/implementierung/work/verkehrsdaten/"

// vollstaendige Entfernungstabelle
#define WEGETABPFAD "/home/fgm_halde/"

// Programm GraphDraw
#define GRAPHDRAWPFAD \
"/home/fgm/implementierung/work/graphdraw/GraphDraw/graphdraw"

const int BILDSCHIRMZEILEN = 24;
#endif
```

8.4 Algorithmen

8.4.1 Inkrementeller Algorithmus

Es wird eine Person gesucht, deren Umsetzung aus einer Fahrgemeinschaft in eine andere die größtmögliche Verbesserung bewirkt. Anschließend wird der Vorgang so lange wiederholt, bis eines der Abbruchkriterien erfüllt ist. Seien zwei Einteilungen benachbart, wenn durch eine Personenumsetzung von der einen in die andere übergegangen werden kann. Dann findet der inkrementelle Algorithmus ein lokales Maximum, die Bewertung der Einteilungen betreffend. Die Abbruchkriterien sind

- Überschreiten einer bestimmten Güte (max_guete)
- Anzahl der Iterationen (iterator)
- Keine Verbesserung erzielt (verbesserung)

Der Einteilungsverwalter erhält am Anfang die Nachricht `aktuelle_einteilung_mit(fahrgemeinschaften)`, woraufhin er die aktuelle Einteilung um die unvermittelten Personen als Einzelfahrgemeinschaften erweitert. Es gibt zwei Arten von Fahrgemeinschaften markierte und unmarkierte. Den markierten Fahrgemeinschaften darf kein Teilnehmer entnommen werden, so daß diese für den Algorithmus nur zum Einfügen interessant sind. Der Algorithmus läßt nun Veränderungen an der Einteilung durch den Einteilungsverwalter vornehmen, bis es zum Abbruch kommt. Anschließend löst der Algorithmenverwalter die Einzelfahrgemeinschaften wieder durch die Nachricht `aktuelle_einteilung_ohne()` auf.

Der Algorithmus geht in einer Iteration alle unmarkierten Fahrgemeinschaften durch und sucht für jede der dort enthaltenen Personen eine Fahrgemeinschaft, in die sie eingefügt werden könnte. Dabei merkt man sich nur die beste Umsetzung und am Ende des Durchlaufes auch durchgeführt, falls es eine Verbesserung gab. Auch wird die gesamte Einteilung bewertet, um festzustellen, ob die maximale Güte vielleicht schon überschritten wurde.

In einer Iteration hat der Algorithmus eine Laufzeit von $O(n^2) * O(Bewertungsfunktion)$, wobei n gleich der Anzahl der Personen ist, über die der Personenstamm verfügt. Die Anzahl der Iterationen kann im worst case vermutlich exponentiell in n sein. Bei unseren Tests hat der Algorithmus in weniger als n Iterationen gestoppt. Der Pseudocode des Algorithmus findet sich in der Abbildung 8.4.1.

8.4.2 MM_Algorithmus

Im Entwurf unter Punkt 5.2.5.1 wurde beschrieben, wie der MM_Algorithmus funktioniert, im folgenden Abschnitt wird auf die Implementierung und die Laufzeit des Algorithmus eingegangen.

Bei der Implementierung wurden Klassen und Algorithmen aus der LEDA-Bibliothek (Abschnitt 8.1.3) verwendet. Zur Repräsentation des Graphen


```
inkrementelle_verbesserung(int it, double guete_obergrenze)
≡
  list < FGM_ID > fahrgemeinschaften, unmarkierte_fahrgemeinschaften,
    personen_fgmen;
  FGM_ID fgm_plus, fgm_minus, neu_fgm_plus, neu_fgm_minus;
  Person_ID person, neue_person;
  double bewertungs_differenz, max_bewertung;
  int zaehler = 1;
  Boolean heuristisch;
  boolean verbesserung, kleiner_als_guete_obergrenze;

  einteilung_mit(fahrgemeinschaften);
  gib_unmarkierte_fgm(unmarkierte_fahrgemeinschaften);
  fgm_plus = 0; fgm_minus = 0; neu_fgm_plus = 0;
  neu_fgm_minus = 0; person = 0; neu_person = 0;
  bewertungs_differenz = 0.0; max_bewertung = 0.0;
  verbesserung = TRUE; kleiner_als_max_obergrenze = TRUE;

  while (zaehler <= it && verbesserung && kleiner_als_guete_obergrenze) do
    foreach fgm_minus in unmarkierte_fahrgemeinschaften do
      foreach person in gib_personen(fgm_minus) do
        if heuristisch then
          personen_fgmen = schraenke_fgmen_ein(fahrgemeinschaften);
        else
          personen_fgmen = kopiere_liste(fahrgemeinschaften);
        end if
        foreach fgm_plus in personen_fgmen do
          if fgm_plus ≠ fgm_minus then
            bewertungs_differenz = (bewerte_fgm_ohne(fgm_minus, person) +
              bewerte_fgm_mit(fgm_plus, person)) −
              (bewerte_fgm(fgm_minus) + bewerte_fgm(fgm_plus));
              if bewertungs_differenz > max_bewertung then
                max_bewertung = bewertungs_differenz;
                neu_fgm_plus = fgm_plus;
                neu_fgm_minus = fgm_minus;
                neu_person = person;
              end if
        end foreach
      end foreach
    end foreach
    if max_bewertung > 0 then
      neu_fgm_plus = fuege_hinzu(neu_fgm_plus, person);
      neu_fgm_minus = nehme_heraus(neu_fgm_minus, person);
      verbesserung = TRUE;
      kleiner_als_max_obergrenze =
        (bewerte_aktuelle_einteilung() < guete_obergrenze);
    else
      verbesserung = FALSE;
    end if
  end while
```

Abb. 8.1: Der Pseudocode des inkrementellen Algorithmus wurde ein *parameterized Ugraph* aus LEDA eingesetzt und die Berechnung des maximalen Matchings auf diesem Graph wird mit Hilfe des *MAX_CARD_MATCHING(...)*-Algorithmus aus LEDA durchgeführt.

Die Laufzeit des LEDA-Matching-Algorithmus liegt in $O(n^3)$ und die Erstellung des Graphen benötigt jeweils höchstens n^2 Schritte, da für jedes Knotenpaar geprüft wird, ob eine Fahrgemeinschaft möglich ist. Daraus ergibt sich, daß die Gesamtlaufzeit des MM_Algorithmus in $O(n^3)$ liegt.

8.4.3 Wegsuche auf dem flachen Verkehrsgraphen

Im folgenden sind Implementierung und Experimente zur Wegsuche auf dem normalen, nicht hierarchischen Verkehrgraphen beschrieben. Dabei wurde die Graphenbibliothek von LEDA benutzt.

8.4.3.1 Verkehrsgraph

Die Algorithmen zur Wegsuche arbeiten auf dem im Zwischenbericht beschriebenen Graphenformat. Die Kanten des Graphen sind hier mit einer Längenangabe in Metern beschriftet. Nach ersten Experimenten bei der Wegsuche zeigte sich, daß der Dijkstra-Algorithmus (s.u.) bei dieser Kantengewichtung zwar den kürzesten Weg findet, allerdings führen die Wege manchmal mitten durch Wohngebiete. Um die schnellen Straßen (niedrige Strassenklassen) stärker zu berücksichtigen, wurde jeder Strassenklasse eine Geschwindigkeit zugeordnet und die Kanten mit der Durchfahrtszeit in Sekunden beschriftet:

- Klasse 0: 100 km/h (Autobahn)
- Klasse 1+2: 60 km/h (Bundes-, Landstraßen, fast immer innerorts)
- Klasse 3+4: 40 km/h (Verbindungs-, Hauptstraßen)
- Klasse 5: 25 km/h (Siedlungsstraßen)
- Klassen 6-8: 20 km/h (Siedlungsstraßen, aber auch Feld- und Gehwege)

Die Zuordnung der Geschwindigkeiten ist durchaus problematisch, da die Straßenklasse in GDF eigentlich nur von der Straßenbreite im zugrundeliegenden Kartenmaterial abhängt. Diese hat mit der erlaubten Höchstgeschwindigkeit zunächst einmal nichts zu tun. Vor allem die Klassen 6-8 machen Probleme, da sie durchaus wichtige Siedlungsstraßen enthalten, andererseits aber auch Feldwege und Fußgängerzonen. Damit diese Straßen möglichst nicht durchfahren werden, bekommen sie eine niedrigere Geschwindigkeit als die der Klasse 5. Im GDF-Standard ist eine Attributierung mit Höchstgeschwindigkeiten zwar vorgesehen, in unserem Datensatz sind aber keine solche Angaben vorhanden. Insgesamt wurden eher niedrige Geschwindigkeiten angesetzt, schließlich wird der Aufenthalt an Kreuzungen und die Verkehrslage überhaupt nicht berücksichtigt.

Der Graph, auf dem die Wegsuche nun durchgeführt wird, enthält ca. 15500 Knoten. Einbahnstraßen werden durch gerichtete Kanten (ca. 38500 Kanten) modelliert. Die in GDF ebenfalls vorhandenen Abbiegeverbote werden nicht berücksichtigt.

8.4.3.2 Algorithmen

Zur Wegsuche im Stuttgarter Straßennetz wurden drei Algorithmen implementiert. Der Dijkstra-Algorithmus (mit oder ohne A*-Heuristik) berechnet den kürzesten Weg zwischen einem vorgegeben Start- und Zielknoten, während der Floyd-Algorithmus die kürzesten Wege zwischen allen Knotenpaaren berechnet.

Dijkstra-Algorithmus

Beim Dijkstra-Algorithmus wird die Knotenmenge des Graphen während der Suche in drei Mengen aufgeteilt: die Knoten, für die die kürzeste Entfernung zum Startknoten schon bekannt ist, die Menge der bisher unerreichten Knoten und die Menge der Randknoten, für die eine Entfernungsschätzung vorliegt. Jeder Knoten `v` trägt die Markierungen `pred[v]` (letzte Kante auf dem kürzesten Weg zu `v`) und `dist[v]` (Entfernung von `v` zum Startknoten). Jede Kante ist mit einer Länge `cost[e]` beschriftet. Die Menge der Randknoten wird in einer Prioritätswarteschlange verwaltet. Diese Datenstruktur erlaubt einen schnellen Zugriff auf den Knoten mit der kleinsten Entfernung `dist[v]` zum Startknoten.

Nach einer Initialisierungsphase löscht der Algorithmus immer den kleinsten Knoten `u` aus der Prioritätswarteschlange (für diesen ist die kürzeste Entfernung nun bekannt), betrachtet alle von `u` ausgehenden Kanten und ändert die Beschriftung `dist[v]` der über diese Kanten erreichten Knoten `v`, falls ein kürzerer Weg nach `v` über `u` gefunden wurde. Falls ein Knoten `v` hierbei zum ersten Mal besucht wurde, wird er in die Prioritätswarteschlange eingefügt. Der Algorithmus erhält als Eingabe den Startknoten `s` und den Zielknoten `t`.

```
void Dijkstra(node s, node t)
{
  priority_queue<int,node> PQ;
  forall_nodes(node v,graph G)
  {
    pred[v] = nil;
    dist[v] = MAXINT;         // Entfernung unendlich
  }
  dist[s] = 0;
  PQ.insert(0,s);
  while (!PQ.empty())         // solange priority-queue nicht leer
  {
    node u = PQ.delete_min(); // kleinstes Element aus PQ loeschen
    int du = dist[u];
    forall_adj_edges(edge e, node u)  // betrachte alle von u aus-
    {                                 // gehenden Kanten
      v = target(e);                  // Kante e=(u,v)
      int c = du + cost[e];
      if (c < dist[v])
      {
        if (dist[v] == MAXINT)        // v wurde noch nie besucht
          PQ.insert(int c,node v);
```

```
        else                          // kuerzerer Weg zu v wurde
        {                             // gefunden
          PQ.decrease(node v,int c);
        }
        dist[v] = c;
        pred[v] = e;
      }
    }
    if (u == t) break;   // Abbruch, wenn Zielknoten
  }                      // kleinstes Element war
}
```

Läßt man die Abfrage `u == t` weg, werden alle kürzesten Wege mit Startknoten `s` berechnet. Für diesen Fall wird weiter unten eine Laufzeitabschätzung angegeben.

Für die Prioritätswarteschlange kann man den in LEDA schon vorhandenen Fibonacci-Heap verwenden. Sei n die Anzahl der Elemente in der Warteschlange, dann ist die Operation `delete_min` von der Ordnung $O(\log n)$, während die Operationen `decrease`, `empty` und `insert` in konstanter amortisierter Laufzeit erledigt werden können ($O(1)$).

Nun zur Laufzeitabschätzung. Sei der betrachtete Graph durch $G = (V, E)$ gegeben, dann braucht man für die Initialisierungsphase die Zeit $O(|V|)$. Die Operationen `insert` und `decrease` werden $|E|$-mal ausgeführt, Zeitbedarf also $O(|E|)$. Die maximale Größe des Fibonacci-Heaps ist $|V|$. Die Operation `delete_min` wird $|V|$-mal ausgeführt, also ergibt sich eine Gesamtlaufzeit von $O(|E| + |V| \cdot \log |V|)$. Da Stadtpläne fast planar ist, kann man $O(|E|) = O(|V|)$ annehmen und erhält für die Laufzeit ($n = |V|$): $O(n \cdot \log n)$

Dijkstra-Algorithmus mit A*-Heuristik

Im folgenden betrachten wir einen euklidischen Graph $G = (V, E)$, d.h. die Knoten tragen x- und y-Koordinaten und das Gewicht einer Kante (u, v) entspricht dem euklidischen Abstand zwischen u und v. Bei der A*-Heuristik werden zu den bisher aufgelaufenen Kosten `dist[v]` noch die geschätzten Restkosten zum Ziel aufaddiert. Eine untere Schranke für den Restweg zum Ziel `t` ist einfach der euklidische Abstand zu `t` (Luftlinie). Jeder Knoten erhält eine zusätzliche Markierung `heurist_dist`, die gerade dieser Summe entspricht. In der Prioritätswarteschlange sind die Knoten ebenfalls nach dieser neuen Größe geordnet, bei jedem Schleifendurchlauf wird also der Knoten ausgewählt, bei der die Summe aus bisheriger Weglänge und geschätzter Restentfernung am kleinsten ist. Dadurch werden die Knoten in der Randmenge bevorzugt, die näher am Zielknoten liegen und die isotrope Ausbreitungsrichtung des einfachen Dijkstra-Algorithmus wird auf den Zielknoten fokussiert. Insgesamt werden bei der A*-Suche also weniger Knoten besucht.

Im weiter unten angegebenen Algorithmus taucht ausserdem noch ein sog. Restfaktor `rf` auf, bei euklidischen Graphen kann man diesen gleich eins setzen und erhält die oben beschriebene A*-Suche. Da es sich beim Verkehrsgraphen

aber nicht um einen euklidischen Graphen handelt, sondern die Kanten mit der Durchfahrtszeit bewertet sind, muß man die Restfahrtzeit zum Ziel abschätzen. Eine untere Schranke erhält man sicher dann, wenn man für die restliche Fahrtstrecke die Höchstgeschwindigkeit annimmt (100 km/h). Durch den Faktor `rf` läßt sich diese Geschwindigkeit einstellen und eventuell auch noch absenken, um die Zahl der besuchten Knoten möglichst klein zu halten. (s. Experimente).

```
void Dijkstra_astar(node s, node t, double rf)
{                              // rf ist der Restfaktor
  priority_queue<int,node> PQ;
  forall_nodes(node v,graph G)
  {
    pred[v] = nil;
    dist[v] = MAXINT;
    heur_dist[v] = MAXINT;   // heur_dist: dist
  }                          //       + geschaetzte Restentfernung
  dist[s] = 0;
  int rest_dist = rf*euclid_dist(s,t);
  heur_dist[s] = rest_dist;    // Luftlinie zwischen s und t
  PQ.insert(rest_dist,s);      // Eintraege in PQ: dist + Rest
  while (!PQ.empty())
  {
    node u = PQ.delete_min();
    int du = dist[u];
    forall_adj_edges(edge e, node u)
    {
      v = target(e);
      int c = du + cost[e];
      if (c < dist[v])
      {
        int rest_dist = rf*euclid_dist(v,t);
        if (dist[v] == MAXINT)            // Luftlinie zw. v und t
          PQ.insert(c+rest_dist,node v);
        else
        {
          PQ.decrease(node v,c+rest_dist);
        }
        dist[v] = c;
        heur_dist[v] = c+rest_dist;
        pred[v] = e;
      }
    }
    if (u == t) break;
  }
}
```

Floyd-Algorithmus

Beim Floyd-Algorithmus werden die kürzesten Wege zwischen allen Knotenpaaren berechnet. Dazu legt man eine $|V| \times |V|$-Matrix `a` an, wobei der Eintrag `a[i][j]` nachher die Entfernung zwischen Knoten `i` und Knoten `j` enthalten soll. In einer Initialisierungsphase wird für die Knotenpaare, für die eine Kante im Graphen existiert, das Kantengewicht `cost[e]` eingetragen. Alle restlichen Einträge werden auf unendlich (`MAX_DIST`) gesetzt. In den folgenden drei geschachtelten `for`-Schleifen werden nun sukzessive für wachsendes `k` die kürzesten Weglängen zwischen allen Knoten `i` und `j` berechnet, wobei auf diesen Wegen nur Knoten mit einer Nummer kleinergleich `k` besucht werden. Nach $|V|$ Durchläufen der äußersten Schleife enthält `a` die kürzesten Entfernungen zwischen allen Knotenpaaren. Für den Zeitaufwand erhält man $O(|V|^3)$.

```
void Floyd()
{
  // alle Eintraege auf MAX_DIST setzen
  for (int i = 1; i <= NODE_NO; i++)
     for (int j = 1; j <= NODE_NO; j++)
       a[i][j] = MAX_DIST;

  // Kantengewichte in Matrix a eintragen
  for (int i = 1; i <= NODE_NO; i++)
    for (int j = 1; j <= NODE_NO; j++)
    {
      edge e = (node i, node j);
      a[i][j] = cost[e];
    }

  // Hauptschleife
  for (int k = 1; k <= NODE_NO; k++)
    for (int i = 1; i <= NODE_NO; i++)
      for (int j = 1; j <= NODE_NO; j++)
      {
       int neu_entf = a[i][k]+a[k][j];
       if (a[i][j] > neu_entf)
          a[i][j] = neu_entf;
      }
}
```

8.4.3.3 Zusammenfassung der Experimente

Sämtliche Messungen wurden auf einem Sparc-Prozessor unter SunOS 5.5.1 durchgeführt, als C++-Compiler wurde der gnu-Compiler Version 2.7.2.1 verwendet.

Vollständige Entfernungstabelle

Für spätere Vergleichsmessungen mit den anderen Algorithmen wurde eine vollständige Entfernungstabelle zwischen allen Knotenpaaren auf Festplatte angelegt (15500x15500x2 Byte = 480 MB). Dazu wurde der Dijkstra-Algorithmus auf alle $|V|$ Startknoten angewandt (kein Zielknoten) und die in Abschnitt 8.4.3.1 beschriebene Kantengewichtung verwendet. Der Zeitaufwand beträgt hierfür $O(|V|^2 \cdot \log |V|)$, die reale Laufzeit betrug 4.2 Stunden. Für die Einzelwegsuche ergibt sich also ein Durchschnittswert von 0.98 s. Außerdem wurde die Laufzeit für 10000 zufällige Tabellenzugriffe gemessen (Gleichverteilung auf den Knotennummern). Für einen Tabellenzugriff ergibt sich dann ein Durchschnittswert von 28 ms. Da die Knoten nicht fortlaufend numeriert sind, wurde eine zusätzliche Indextabelle zum Nachschlagen der Adresse angelegt. Für einen Tabellenzugriff sind also zwei Plattenzugriffe nötig.

Im Vergleich dazu wurde die Laufzeit des Floyd-Algorithmus (Aufwand $O(|V|^3)$) bestimmt. Da für ein Array von 480 MB der virtuelle Speicher nicht ausreicht, wurden nur 13000 der 15500 Knoten betrachtet (340 MB). Mittels einer Hochrechnung erhält man für die Laufzeit einen Wert von 35.800 Stunden für 13000 Knoten. Dieser hohe Wert läßt sich dadurch erklären, daß bei einem Array der Größe 340 MB und einem Hauptspeicher von 120 MB der Rechner vermutlich nur noch mit Paging beschäftigt ist. Man könnte das Array auch komplett auf Platte ablegen, das bringt aber wahrscheinlich auch keinen grossen Geschwindigkeitsgewinn.

Einfluß der A*-Heuristik

Hier wurde der Restfaktor `rf` bestimmt, so daß die A*-Heuristik immer noch den optimalen Weg findet und die geschätzte Restzeit zum Ziel möglichst klein ist, also die Anzahl der besuchten Knoten minimiert wird. Bei der Untersuchung wurden die mit der A*-Heuristik berechneten Entfernungen mit den Einträgen der Entfernungstabelle verglichen.

Die Experimente wurden auf einer kleineren Version des Verkehrsgraphen mit ca. 13000 Knoten durchgeführt (die Straßenklassen 6-8 wurden weggelassen). Für die verschiedenen Straßenklassen wurden abweichend von Abschnitt 8.4.3.1 folgende Geschwindigkeiten gewählt:

- Klasse 0: 100 km/h
- Klasse 1: 60 km/h
- Klasse 2-4: 40 km/h
- Klasse 5: 25 km/h

Für `rf` wurde mittels binärer Suche ein optimaler Wert von 0.42 ermittelt, das entspricht einer Geschwindigkeit von ca. 60 km/h (mit 100 km/h wäre man auf jeden Fall auf der sicheren Seite).

Bei der Ermittlung von `rf` wurde ein Startknoten fest gewählt und die Entfernungen zu allen $|V|-1$ Zielknoten berechnet. Beim optimalen Wert von `rf` erhält man eine Laufzeit von 53 Min. für die Berechnung des kürzesten Weges zwischen $|V|-1$ Knotenpaaren, der einfache Dijkstra-Algorithmus benötigt auf der kleineren Version des Verkehrsgraphen 64 Min.. Gegenüber dem Dijkstra-Algorithmus werden nur 52% der Knoten besucht. Dieser Vorteil wird aber durch die aufwendige Entfernungsberechnung praktisch wieder zunichte gemacht. Während der einfache Dijkstra-Algorithmus mit Integer-Additionen auskommt, werden für die Berechnung des euklidischen Abstands Floating-Point-Operationen benötigt (Quadrieren und Wurzelziehen). Diese sind wesentlich langsamer als eine einfache Integer-Addition.

8.4.3.4 Zusammenfassung

Der gewöhnliche Dijkstra-Algorithmus bestimmt den kürzesten Weg auf dem vorliegenden Verkehrsgraph in etwa 1s und scheint von den drei untersuchten Algorithmen immer noch der geeignetste zu sein. Die A*-Heuristik bringt wegen der aufwendigen Entfernungsberechnung keinen Laufzeitvorteil. Der Floyd-Algorithmus ist wegen seines hohen Speicherbedarfs ebenfalls nicht brauchbar.

8.4.4 Wegsuche auf dem hierarchischen Verkehrsgraphen

8.4.4.1 Randknotenmodell

Zur Beschleunigung der Wegsuche wurde die Wegsuche auf hierarchischen Verkehrsgraphen implementiert. Dabei soll nur die Entfernung zwischen Start- und Zielknoten bestimmt werden, der Verlauf des Weges wird dabei außer acht gelassen.

Beim Randknotenmodell besteht der Graph aus zwei Leveln, Level eins entspricht dem alten flachen Verkehrsgraphen $G_1 = (V, E, \mu, \delta)$ mit den Knotenkoordinaten $\mu : V \to \mathbb{R}^2$ und der kürzesten Entfernung zwischen zwei Knoten $\delta : V \times V \to \mathbb{R}^+$. Nun teilt man den Graphen in k geographische Regionen ein, d.h. es wird eine Partition $V = \bigcup_{i=1}^{k} V_i$ der Knotenmenge gebildet. Jeder Knoten v erhält eine Regionennummer $R(v)$ mit $R(v) = i$, falls $v \in V_i$. Durch die Regionengrenzen werden einige Kanten zerschnitten, die sog. Randkanten

$$E_R = \{(u,v) \in E \mid R(u) \neq R(v)\}$$

$E_R(i)$ bezeichne die Randkanten der Region i:

$$E_R(i) = \{(u,v) \in E_R \mid R(u) = i \vee R(v) = i\}$$

Liegen Startknoten u und Zielknoten v in verschiedenen Regionen ($R(u) \neq R(v)$), so müssen alle Pfade von u nach v auch über Randkanten führen. Sei $u = u_0 u_1 \dots u_{n-1} u_n = v$ mit $(u_i, u_{i+1}) \in E \quad \forall i \in \{0 \dots n-1\}$ ein solcher Pfad, dann existieren also Knoten w_1, w_2, w_3 und $w_4 \in \{u_i \mid 1 \leq i \leq n\}$ mit $(w_1, w_2) \in E_R(R(u))$, $(w_3, w_4) \in E_R(R(v))$ und $R(w_1) = R(u)$ sowie $R(w_4) = R(v)$. Da ein Weg aus einer Region über eine Randkante der Region führen muß, kann man jeweils einen der beiden zu einer Randkante inzidenten Knoten in die Randknotenmenge V_R aufnehmen, z.B. immer den Startknoten:

$$V_R = \{u \in V \mid \exists v \in V : (u,v) \in E_R\}$$

Wege zwischen verschiedenen Regionen müssen über solche Randknoten führen, da man die zugehörigen Randkanten betreten muß. Die Randknotenmenge einer Region i geht aus der Randkantenmenge $E_R(i)$ hervor:

$$V_R(i) = \{u \in V_R \mid \exists v \in V : (u,v) \in E_R(i)\}$$

Die Randknoten bilden den zweiten Graphlevel: $G_2 = (V_R, V_R \times V_R, \delta_2)$. Für alle Randknotenpaare $(u,v) \in V_R \times V_R$ wird die Entfernung $\delta_2(u,v) = \delta(u,v)$ auf dem Level-1-Graphen berechnet und in einer Tabelle gespeichert. Die Wegsuche zwischen Startknoten u und Zielknoten v mit $R(u) \neq R(v)$ kann dann in folgenden Schritten ablaufen:

1. Bestimmung der kürzesten Wege $\delta(u, u_R)$ vom Startknoten u zu allen Randknoten $u_R \in V_R(R(u))$ seiner Region.
2. Bestimmung der kürzesten Wege $\delta(v_R, v)$ von allen Randknoten $v_R \in V_R(R(v))$ der Zielregion zum Zielknoten v.
3. Bestimmung der kürzesten Entfernung $\delta(u,v)$ zwischen Start- und Zielknoten durch Minimierung der Summe $\delta(u,u_R)+\delta(u_R,v_R)+\delta(v_R,v)$ über alle Randknotenpaare $(u_R, v_R) \in V_R(R(u)) \times V_R(R(v))$.

Für die Schritte 1 und 2 werden in der Preprocessing-Phase ebenfalls Tabellen angelegt, die sog. Start- und Ziellisten. Die Wegsuche besteht also lediglich aus Tabellenzugriffen und der Minimierung über den Randknotenpaaren. Liegen Start- und Zielknoten in derselben Region ($R(u) = R(v)$), wird die Entfernung mit dem gewöhnlichen Dijkstra-Algorithmus auf dem Level-1-Graphen bestimmt.

Bestimmung der Randknoten

Mit der in Abschnitt 8.4.4.4 gewonnenen Regionenmarkierung der Knoten werden die Randknoten mit folgender Routine ausgewählt:

```
forall_edges(edge e, G)
  {
    node u = G.target(e);
    node v = G.source(e);

    int u_region = G.read_region(u);  // Regionennummern
    int v_region = G.read_region(v);

    // zerschnittene Kante
    if (u_region != v_region)
      {
        if ((!in_randknotenliste(u, u_region) ||
             !in_randknotenliste(u, v_region)) &&
            (!in_randknotenliste(v, u_region) ||
             !in_randknotenliste(v, v_region)))
```

```
          {
            randknoten_eintragen(v, u_region, v_region);
          }
      }
  }
```

Eine zerschnittene Kante liegt dann vor, wenn die Regionennummern der inzidenten Knoten `u` und `v` unterschiedlich sind. Für jede Region wird eine Liste der zugehörigen Randknoten geführt, ein Randknoten ist also in mindestens zwei solcher Listen enthalten. In der zweiten `if`-Abfrage wird überprüft, ob `u` und `v` beide noch nicht als Randknoten zwischen den Regionen `region_u` und `region_v` eingetragen sind. Ist dies der Fall, so wird `v` als neuer Randknoten in beide Randknotenlisten eingetragen. Durch diese Bedingung wird die Randknotenmenge V_R gegenüber der Definition in Abschnitt 8.4.4.1 verkleinert. Wenn zwei entgegengesetzt orientierte Randkanten $(u, v), (v, u) \in E_R$ vorliegen, wird jetzt entweder u oder v ausgewählt, aber nicht beide. Man erhält so eine kleinere Randknotenmenge. Welcher der beiden zum Randkantenpaar inzidenten Knoten ausgewählt wird, hängt davon ab, in welcher Reihenfolge die Randkanten in obigem Algorithmus durchlaufen werden. Beim Stuttgarter Graphen erhält man mit dieser Methode 558 Randknoten.

Bestimmung der Randknotentabelle

Die Entfernung aller Randknotenpaare $\delta_2(u, v) = \delta(u, v) \quad \forall u, v \in V_R$ werden mit dem Dijkstra-Algorithmus auf dem Level-1-Graphen G_1 bestimmt. Dabei wird jeder Randknoten $u \in V_R$ als Startknoten ausgewählt und die Entfernungen zu allen anderen Randknoten in einem Durchlauf bestimmt. Die Randknotentabelle benötigt etwa 1.2 MB im Hauptspeicher.

Bestimmung der Start- und Ziellisten

Für die Entfernungen jedes Regionenknotens $u \in V_i$ zu seinen Randknoten $u_R \in V_R(i)$ werden Listen angelegt, für jeden Knoten eine Start- und eine Zielliste. Die Startliste enthält die kürzesten Entfernungen $\delta(u, u_R)$ vom Regionenknoten zu seinen Randknoten, die Zielliste die kürzesten Entfernungen $\delta(u_R, u)$ von den Randknoten zum Regionenknoten. Zur Berechnung der Zielliste wird auf dem umgekehrt orientierten Graphen G_1^{-1} die kürzeste Entfernung $\delta(u, u_R)$ von $u \in V_i$ zu seinen Randknoten $u_R \in V_R(i)$ bestimmt.

```
// zur Berechnung von ziellist Graphen komplett umdrehen

G.rev_all_edges();

for (int regionnr = 1; regionnr <= max_regionnr; regionnr++)
  {
    list<node> randlist;    // Liste der Randknoten
    list<int>  ziellist;    // Liste der Entfernungen zu den Randknoten
```



```
    randlist = randknoten(regionnr);

    forall_nodes(u, G)
      {
        if (region(u) == regionnr)
           {
              ziellist(u) = dijkstra_many_targets(u, randlist);
           }
      }
  }

// zur Berechnung von startlist Graph wieder zurueckdrehen

G.rev_all_edges();

  ...
```

Die Startliste wird wieder auf dem ursprünglichen Graphen G_1 berechnet. Bei der kürzesten Wegesuche zu den Randknoten einer Region wird die Variante `dijkstra_many_targets` des Dijkstra-Algorithmus verwendet. Bei jedem in die Baummenge aufgenommenen Knoten wird überprüft, ob er zur Randknotenmenge der Region gehört. Sind alle Randknoten einer Region in die Baummenge aufgenommen worden, kann die Suche abgebrochen werden.

Damit sind die Preprocessing-Schritte der Abschnitte 8.4.4.1, 8.4.4.1 und 8.4.4.1 abgeschlossen. Die Laufzeit dafür beträgt etwa 6.8 h. Start- und Ziellisten benötigen etwa 20 MB Hauptspeicher.

Wegsuche

Bei der Wegsuche von u nach v mit $R(u) = i \neq j = R(v)$ seien $u_R \in V_R(i)$ und $v_R \in V_R(j)$ wiederum die Randknoten der beiden beteiligten Regionen. Nun werden die Entfernungen $\delta(u, u_R)$, $\delta(u_R, v_R)$ und $\delta(v_R, v)$ in den Tabellen nachgeschlagen und die Summe $\delta(u, u_R) + \delta(u_R, v_R) + \delta(v_R, v)$ über alle Randknotenpaare $(u_R, v_R) \in V_R(i) \times V_R(j)$ minimiert:

```
int suche(node s, node t)
{
  int region_s = G.read_region(s);
  int region_t = G.read_region(t);

  if (region_s == region_t)
    {
      int entf = dijkstra.suche(s, t);
      return entf;
    }
  else
```

```
    {
      // Entfernung von t zu seinen Randknoten

      list<node> randlist_t = randknoten(region_t);
      list<int>  entflist_t = ziellist(t);

      // Entfernung von s zu seinen Randknoten

      list<node> randlist_s = randkonten(region_s);
      list<int>  entflist_s = startlist(s);

      // Minimierung ueber alle Randknotenpaare

      int mindist = MAXINT;

      for (int i = 0; i <  randlist_s.length(); i++)
        {
          node rand_s = randlist_s.item(i);
          int  dist_s = entflist_s.item(i);

          for (int j = 0; j < randlist_t.length(); j++)
            {
              node rand_t = randlist_t.item(j);
              int  dist_t = entflist_t.item(j);

              // Randknotenentfernungen stehen in cost2
              int dist2 = cost2(rand_s, rand_t);
              int dist  = dist_s + dist2 + dist_t;

              if (dist < mindist)
                {
                  mindist = dist;
                }
            }
        }
      return mindist;
    }
}
```

Liegen Start- und Zielknoten in der gleichen Region ($R(u) \neq R(v)$), so wird die Entfernung mit dem Dijkstra-Algorithmus auf dem Level-1-Graphen G_1 bestimmt. Für die Zeitmessung wurden für 50 Startknoten die Entfernungen zu allen Zielknoten bestimmt. Die Gesamtlaufzeit betrug 7.0 h, für eine Wegsuche ergibt sich damit eine durchschnittliche Laufzeit von 34 ms. Liegen beide Knoten in derselben Region, kommt der langsamere Dijkstra-Algorithmus (Sekundenbereich) auf G_1 zum Einsatz. Deswegen liegt der Zeitaufwand bei Knoten in verschiedenen Regionen wohl eher unter dem Durchschnittswert.

8.4.4.2 Rathausmodell

Beim Rathausmodell geht man davon aus, daß bei längeren Wegen bestimmte Knoten (Rathäuser) auf jeden Fall besucht werden. Es liegt wiederum der Level-1-Graph $G_1 = (V, E, \mu, \delta)$ vor. Die Rathäuser $R \subseteq V$ bilden die Knotenmenge von Level 2: $G_2 = (R, R \times R, \delta_2)$. Für die Rathausentfernungen legt man eine vollständige Tabelle $\delta_2(u, v) = \delta(u, v)$ an, die anhand des Level-1-Graphen G_1 berechnet wird. Bei der Wegsuche bestimmt man jeweils die drei nächsten Rathäuser zum Start- und Zielknoten und minimiert über die Rathauspaare. Diese Heuristik garantiert natürlich nicht, daß man hierbei den optimalen Weg findet. Unterschreitet der euklidische Abstand von Start- und Zielknoten eine bestimmte Schranke (mittlere Rathausentfernung), so wird man mit dem herkömmlichen Dijkstra-Algorithmus den kürzesten Weg bestimmen, da der Weg zum nächsten Rathaus dann meistens ein Umweg ist.

Man kann das Rathausmodell als eine Verallgemeinerung des Randknotenmodells ansehen, da die Auswahl der Rathäuser zunächst einmal beliebig ist. Beim Randknotenmodell sind die Randknoten durch die Regionengrenzen festgelegt.

Auswahl der Rathäuser

Als Rathäuser werden zunächst alle größeren Straßenkreuzungen ausgewählt, d.h. alle Knoten mit Mindestgrad 3 (ungerichtete Version von G_1), bei denen mindestens eine Straße der Klassen 0-4 beteiligt ist. Man erhält so etwa 2100 Rathäuser. Um diese Zahl weiter zu reduzieren, wird ein quadratisches Gitter über das Stuttgarter Stadtgebiet gelegt, die Gitterzellenbreite beträgt 755 m ($\frac{1}{25}$ der horizontalen Ausdehnung). Pro Gitterzelle werden jetzt maximal 2 Rathäuser zugelassen, die restlichen werden vernachlässigt. Bei den nun noch leeren Gitterzellen wird überprüft, ob in ihnen überhaupt Knoten vorhanden sind. Ist dies der Fall, so wird einer davon zufällig als Rathaus ausgewählt. Damit ist gewährleistet, daß sich in jeder nichtleeren Gitterzelle mindestens ein Rathaus befindet. Mit dieser Methode erhält man schließlich 661 Rathäuser.

Nun läßt sich eine Abschätzung für den maximalen Fehler bei der Wegsuche im Rathausmodell angeben. Im schlimmsten Fall liegen Startknoten und Startrathaus in gegenüberliegenden Ecken einer Gitterzelle und der kürzeste Weg vom Start- zum Zielrathaus führt wiederum über den Start- und den Zielknoten (siehe Abb. 8.2). Nimmt man nun an, daß der kürzeste Weg zwischen zwei Knoten in gegenüberliegenden Ecken einer Zelle durch die Diagonalenlänge beschränkt ist, so ist der maximale Gesamtfehler die vierfache Diagonalenlänge, also 4270 m. Bei der Niedrigstgeschwindigkeit von 20 km/h entspricht das 13 Min. Fahrtzeit.

Bei der Wahl der Gitterzellenbreite müssen zwei Faktoren beachtet werden: Macht man das Gitter zu fein und fordert weiterhin, daß in jeder nichtleeren Zelle mindestens ein Rathaus liegt, so erhält man zuviele Rathäuser. Ist das Gitter zu grob, so gilt dies für die Fehlerabschätzung ebenfalls.

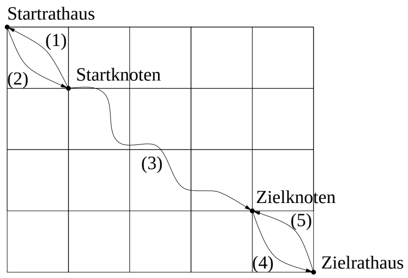

Abb. 8.2: maximaler Fehler beim Rathausmodell

Bestimmung der Rathausentfernungen

Die Tabelle der Rathausentfernungen wird mit der gleichen Methode wie in Abschnitt 8.4.4.1 berechnet. Die Tabelle benötigt etwa 1.7 MB im Hauptspeicher.

Bestimmung der Start- und Ziellisten

Die Entfernungen aller Knoten zu den Rathäusern wird analog zur Vorgehensweise in Abschnitt 8.4.4.1 bestimmt. Die Entfernung zu den drei nächsten Rathäusern wird mit einem modifizierten Dijkstra-Algorithmus bestimmt, der die Liste aller Rathäuser zu Beginn erhält und die Suche beendet, wenn drei Rathäuser in die Baummenge aufgenommen worden sind. Da die Kantengewichte positiv sind, sind dies dann auch die drei Rathäuser mit den geringsten Entfernungen zum Startknoten. Für jeden Knoten wird eine Start- und eine Zielliste zu den drei nächsten Rathäusern berechnet.

Für die Preprocessing-Schritte der Abschnitte 8.4.4.2, 8.4.4.2 und 8.4.4.2 wird eine Laufzeit von 43 Min. benötigt. Die Start- und Ziellisten benötigen etwa 10 MB Hauptspeicher.

Wegsuche

Wie oben schon erwähnt, werden bei der Wegsuche die Entfernungen in den Tabellen nachgeschlagen und über alle neun Rathauspaare minimiert. Für die Zeitmessung wurden für 100 Startknoten die Entfernungen zu allen Zielknoten bestimmt. Die Gesamtlaufzeit betrug 5.7 Min., für eine Wegsuche ergibt sich eine durchschnittliche Zeit von 0.4 ms. Zur Bestimmung des Fehlers bei der Weglänge wurden für ca. 150.000 Knotenpaare die Abweichung vom kürzesten Weg (Fahrtzeit) gemessen. Die durchschnittliche Abweichung beträgt 10.8 s mit Standardabweichung 20.9 s, der Maximalfehler beträgt 300 s. Bei 45% der Knotenpaare wird sogar genau der kürzeste Weg gefunden. Die Wege wurden auch im Programm Graphdraw visuell beurteilt. Liegen Start- und Zielknoten weit

auseinander, so ist der gefundene Weg oft mit dem kürzesten Weg identisch. Wie schon in Abschnitt 8.4.4.2 erwähnt, ist bei geringem euklidischem Abstand zwischen Start- und Zielknoten das Rathausmodell weniger geeignet, da zum Erreichen des nächsten Rathauses meist ein Umweg gemacht werden muß.

8.4.4.3 Zusammenfassung der Experimente

In diesem Abschnitt werden die Meßergebnisse nocheinmal kurz zusammengefaßt. Der vorliegende Stuttgarter Straßengraph besitzt 15.500 Knoten und 38.500 Kanten. Die Laufzeitmessungen wurden auf einer SPARCStation xx mit 120 MB Hauptspeicher durchgeführt.

Randknotenmodell

- Anzahl Randknoten: 558 in 37 Regionen
- Größe der vollständigen Randknotenentfernungstabelle: 1.2 MB
- Größe der Start- und Ziellisten: 20 MB
- Laufzeit für Preprocessing (Auswahl Randknoten, Berechnung Randknotenentfernungen, Berechnung der Start- und Ziellisten): 6.8 h
- durchschnittliche Laufzeit für eine Wegsuche: 34 ms

Rathausmodell

- Anzahl Rathäuser: 661
- Größe der vollständigen Rathausentfernungstabelle: 1.7 MB
- Größe der Start- und Ziellisten: 10 MB
- Laufzeit für Preprocessing (Auswahl Rathäuser, Berechnung Rathausentfernungen, Berechnung der Start- und Ziellisten): 43 Min.
- durchschnittliche Laufzeit für eine Wegsuche: 0.4 ms
- Fehler der berechneten Entfernungen: $\mu = 10.8$ s, $\sigma = 20.9$ s, Maximalfehler 300 s (bei 150.000 Knotenpaaren). Bei 45% der Knotenpaare Fehler 0.

Der Preprocessing- und Speicheraufwand ist beim Randknotenmodell wesentlich höher, allerdings erhält man hier immer den kürzesten Weg. Der Fehler beim Rathausmodell ist aber geringfügig und könnte durch Einsatz des normalen Dijkstra-Algorithmus bei Start- und Zielknoten, deren euklidische Entfernung klein ist, noch verbessert werden.

8.4.4.4 Regionenbildung

Da in den uns zur Verfügung gestellten GDF-Daten keine Regioneneinteilung vorliegt, werden die Regionen mit Algorithmen aus der Bildverarbeitung bestimmt.

Originalbild

Als Ausgangspunkt aller weiteren Bildverarbeitungsschritte wird der Stuttgarter Stadtplan in ein 512x512-Binärbild (nur schwarz und weiß, keine Zwischengrauwerte) eingezeichnet (s. Abb. 8.6). Dabei werden nur die Straßenklassen 0-5 verwendet, so daß die einzelnen Stadtteile noch gut voneinander abgrenzbar sind. Das Ziel der weiteren Schritte ist es nun, das gesamte Stadtgebiet so in Regionen zu zerlegen, daß möglichst wenig Straßen oder Graphkanten zerschnitten werden. Dabei sollen die Regionen ungefähr Stadtteilen entsprechen.

Morphologische Operatoren

Morphologische Operatoren dienen in der Bildverarbeitung zur Analyse der Form und Struktur von Bildobjekten. Im folgenden wird immer mit Binärbildern gearbeitet, diese enthalten nur zwei Pixelwerte: schwarz und weiß. Ein Binärbild läßt sich durch eine Menge $A \subseteq \mathbb{Z}^2$ beschreiben, die alle schwarzen Pixel des Bildes enthält [3]:

$$A = \{(x, y) \in \mathbb{Z}^2 \mid \text{Pixel an Position } (x, y) \text{ schwarz}\}$$

Die Dilatation eines Bilds A durch ein Strukturelement $B \subseteq \mathbb{Z}^2$ wird folgendermaßen definiert:

Definition 1 (Dilatation) $A \oplus B = \{c \mid c = a + b, \forall a \in A, \forall b \in B\}$

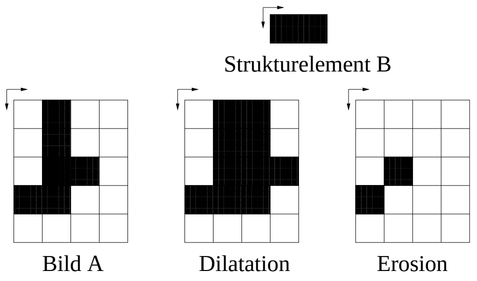

Abb. 8.3: Erosion und Dilatation

Man setzt also den Nullpunkt des Strukturelements B auf jedes schwarze Pixel in A und bildet die Vereinigung über alle so erhaltenen Bilder (vgl. Abb. 8.3). Die Dilatation bewirkt eine Vergrößerung des Randes von A (Expansion).

Die dazu komplementäre Erosion wird wie folgt definiert:

Definition 2 (Erosion) $A \ominus B = \{c \mid c + b \in A, \forall b \in B\}$

Wiederum wird das Strukturelement B über das ganze Bild A verschoben. Der Nullpunkt von B gehört genau dann zum erodierten Bild, wenn B ganz in A enthalten ist (vgl. Abb. 8.3). Die Erosion führt zur Verkleinerung des Randes von A (Kontraktion). Die Operation $(A \oplus B) \ominus B$ (Dilatation gefolgt von Erosion)

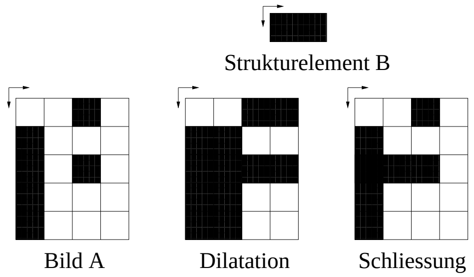

Abb. 8.4: Schließung

nennt man Schließung. Der Name rührt daher, daß dabei Objekte verschmolzen werden, die vorher nur durch schmale Brücken oder garnicht miteinander verbunden waren (siehe Abb. 8.4). Außerdem werden dabei die Objektkonturen geglättet. Die komplementäre Operation $(A \ominus B) \oplus B$ ist die Öffnung, hierbei

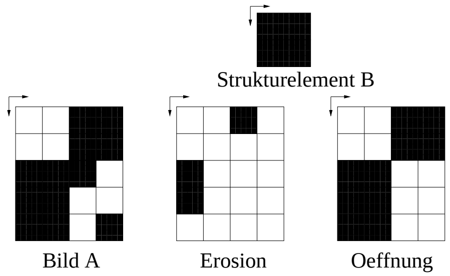

Abb. 8.5: Öffnung

werden kleine Landengen getrennt und kleine Inseln eliminiert (vgl. Abb. 8.5).

Führt man Schließung und Öffnung hintereinander aus, so entspricht dies einem morphologischen Filter, d.h. Bilddetails unterhalb einer bestimmten Größe werden unterdrückt (Tiefpaßwirkung). Diese Eigenschaft wird zum Auffinden der Stadtteile genützt.

Morphologische Einzelschritte

Zunächst wird die Schließung mit einem kreisförmigen 5x5-Strukturelement auf dem Originalbild ausgeführt (siehe Abb. 8.7). Gebiete mit einer hohen Straßendichte werden zu kompakten schwarzen Gebieten. Um die Gebiete voneinander zu trennen, wird eine Öffnung mit dem gleichen Strukturelement durchgeführt (siehe Abb. 8.8). Um die jetzt noch vorhandenen kleinen Inseln wegzufiltern, wird nochmal mit einem größeren 13x13-Strukturelement geöffnet. Man erhält für jede Region ein kompaktes schwarzes Objekt, insgesamt 37 Stück (siehe Abb. 8.9).

Zusammenhangskomponenten

Im nächsten Schritt werden die Zusammenhangskomponenten im Bild bestimmt, d.h. jedes Bildpixel wird mit einer Marke für die entsprechende Komponente versehen. Dabei wird das Bild einmal von links oben nach rechts unten durchlaufen und jedes Pixel C mit dem linken Nachbar L und dem darüberliegenden Pixel U verglichen [11]. C, L und U stehen für die zugehörigen Pixelwerte. Die entsprechenden Marken seien mit C', L', U' bezeichnet. Beim Vergleich der Pixelwerte können vier Fälle auftreten (C ist dabei ein schwarzes Pixel, nur diese erhalten eine Marke):

1. $C = U \wedge C \neq L$: wähle $C' = U'$.
2. $C \neq U \wedge C = L$: wähle $C' = L'$.
3. $C = L \wedge C = U$: wähle willkürlich $C' = L'$. U' wäre genauso möglich. Auf jeden Fall wird die Äquivalenz der Marken U' und L' in einer Äquivalenzliste vermerkt, denn C, U und L gehören zur selben Zusammenhangskomponente.
4. $C \neq U \wedge C \neq U$: bei zwei weißen Nachbarn liegt ein neuer Bereich vor, d.h. C erhält eine neue Marke C' (Inkrementierung eines Markenzählers).

In einem zweiten Durchlauf des Bildes werden die Marken dann noch gemäß der vermerkten Markenäquivalenzen korrigiert, genaueres siehe [11]. Die Marken werden wieder in ein Bild eingezeichnet (siehe Abb. 8.10). Zur Verdeutlichung wurden die Graustufen hierbei einfach zufällig gewählt.

Regionenwachstum

Um eine Unterteilung der gesamten Bildfläche in Regionen zu erhalten, läßt man nun die Komponenten bis zum Zusammenstoß mit einer anderen Komponente wachsen. Dabei wird nach folgendem Algorithmus vorgegangen:


```
void Grow_regions(int b1[MAX_WIDTH][MAX_HEIGHT],
                  int b2[MAX_WIDTH][MAX_HEIGHT])
{
  bool aenderung = true;

  while (aenderung == true)
  {
    aenderung = false;

    // nach oben wachsen
    for (int i = 0; i < max_width; i++)
      for (int j = 0; j < max_height; j++)
        if (j != 0 && b1[i][j] != 0 && b1[i][j-1] == 0)
        {
          b2[i][j-1] = b1[i][j];
          aenderung = true;
        }

    // nach unten
    for (int i = 0; i < max_width; i++)
      for (int j = 0; j < max_height; j++)
        if (j != max_height-1 && b1[i][j] != 0 && b1[i][j+1] == 0)
        {
          b2[i][j+1] = b1[i][j];
          aenderung = true;
        }

    // nach links

       ...

    // nach rechts

       ...
  }
}
```

`b1` entspricht dabei dem Eingabebild, in `b2` steht nachher das Ausgabebild. Die äußere Schleife wird abgebrochen, wenn keine Veränderungen im Bild mehr auftreten. Für jedes Pixel werden die vier Nachbarn überprüft. Falls ein Nachbar schwarz ist (Pixelwert 0), kann die Region in die entsprechende Richtung wachsen. Das Verfahren terminiert, wenn keine schwarzen Hintergrundpixel mehr im Bild vorhanden sind. Die Anwendung des Regionenwachstums auf das Komponentenbild ergibt Abb. 8.11. Zur Verdeutlichung der Regionen ist außerdem der Stadtplan aus dem ersten Bild eingezeichnet. Man kann durchaus einzelne Stadtteile erkennen, obwohl die Unterteilung manchmal etwas zu fein ist. Andererseits gibt es auch eine sehr große Region im Stadtkern.

Regionenzuordnung der Knoten

Die Markierung der Graphknoten mit den entsprechenden Regionennummern kann nun einfach über die Knotenkoordinaten vorgenommen werden. Man erhält damit 1213 zerschnittene Kanten.

Regionenbildung mit Voronoi-Diagramm

Ausgehend von den in Abb. 8.10 dargestellten Zusammenhangskomponenten kann man auch eine Regioneneinteilung mit Hilfe von Voronoi-Diagrammen berechnen. Sei die Halbebene $H(p_1|p_2) = \{p \in \mathbb{R}^2 \mid d(p_1,p) \leq d(p_2,p)\}$ die Menge aller Punkte, die näher beim Punkt p_1 als beim Punkt p_2 liegen [10]. Die Voronoi-Region $V(p)$ eines Punkts $p \in P$ einer Punktmenge P besteht dann aus allen Punkten, die näher bei p liegen als bei irgendeinem anderen Punkt aus P:

$$V(p) = \bigcap_{p' \in P \setminus \{p\}} H(p|p')$$

Wählt man als Punktmenge P nun die Schwerpunkte der Zusammenhangskomponenten (siehe Abb. 8.10), so erhält man das in Abb. 8.12 dargestellte Voronoi-Diagramm. Offensichtlich lassen sich die Stadtteile nicht so leicht durch Polygone abgrenzen, deswegen wurde dieser Ansatz nicht weiter verfolgt.

Bilder

Abb. 8.6: Originalbild

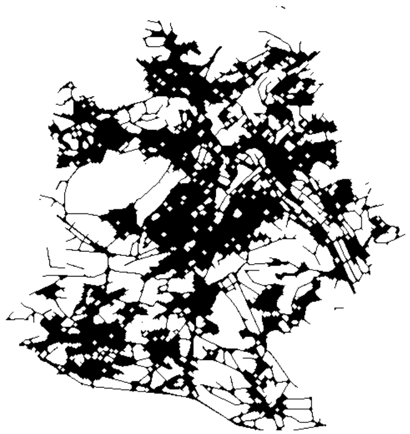

Abb. 8.7: Schließung mit kreisförmigen Strukturelement

Abb. 8.8: Öffnung

Abb. 8.9: Öffnung mit größerem Strukturelement

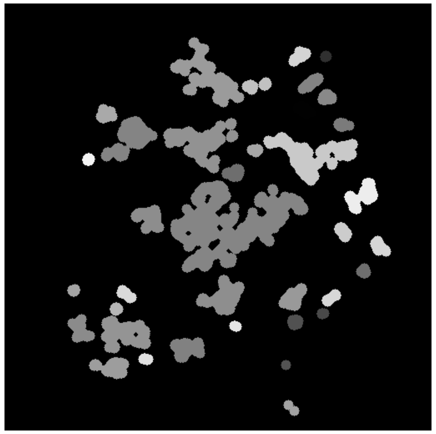

Abb. 8.10: Markierung der Zusammenhangskomponenten (Pseudograustufen)

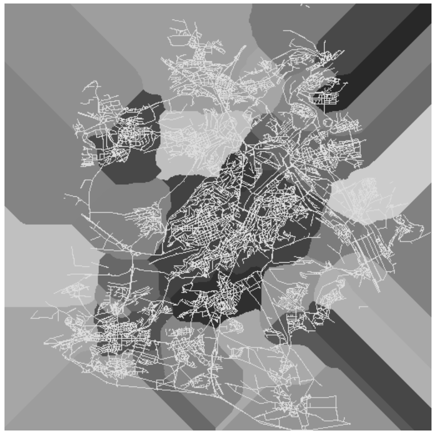

Abb. 8.11: Regioneneinteilung (Pseudograustufen)

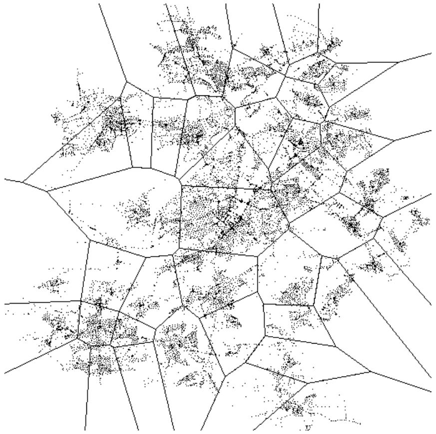

Abb. 8.12: Einteilung mit Voronoi-Diagramm

8.5 Bewertungsfunktion

In der Spezifikation wurde zur Bestimmung des kürzesten Umwegs für den Fahrer eine optimale Berechnung vorgesehen. Diese optimale Berechnung dauert aber wegen ihrer exponentiellen Laufzeit sehr lange. Deshalb wurde als Alternative zur Bestimmung des Umwegs folgende Heuristik implementiert.

Die Heuristik zur Berechnung eines Umwegs für einen Fahrer berechnet je Fahrer genau einen Umweg. Ausgehend vom Startort des Fahrers wird der nächstgelegene sinnvolle Ort besucht. Der nächste sinnvolle Ort nach dem Startort des Fahrers ist einer der Startorte der Mitfahrer, da die Mitfahrer zuerst abgeholt werden müssen, bevor sie an ihrem Arbeitsplatz abgesetzt werden können. Nach der Auswahl eines Startorts der Mitfahrer wird bei der weiteren Suche auch der zugehörige Zielort einbezogen. Dieses Verfahren wird solange wiederholt, bis alle Start- und Zielorte der Mitfahrer besucht wurden. Dann wird zum bisher berechneten Weg noch der kürzeste Weg zum Zielort des Fahrers hinzugefügt. Von der Länge des nun berechneten Wegs wird die Länge des direkten Wegs vom Startort zum Zielort des Fahrers subtrahiert und als Ergebnis erhält man den Umweg, der für den Fahrer entsteht, wenn er die anderen Teilnehmer mitnimmt.

Auf diese Weise wird für jeden Teilnehmer, der fahren kann, ein Umweg berechnet. Der Fahrer, der den kleinsten Umweg benötigt, um alle Teilnehmer absetzen zu können, wird als Fahrer in der Fahrgemeinschaft festgelegt.

8.6 Entfernungstabelle

Während der Einteilungsberechnung durch einen Algorithmus wird sehr oft die Bewertungsfunktion aufgerufen. Sie benötigt zur Bewertung von Einteilungen und Fahrgemeinschaften die Entfernungen zwischen Start- und Zielorten der einzuteilenden Personen. Diese Entfernungen können mit Hilfe der Wegsuchealgorithmen berechnet werden. Damit die Wegberechnung die Laufzeit der Einteilungsalgorithmen nicht beeinträchtigt, wird vor dem Start eines Algorithmus eine Entfernungstabelle erstellt.

Der Algorithmenverwalter ruft vor dem Start eines Algorithmus zur Einteilungsberechnung eine Methode des Objekts Entfernungstabelle auf. Diese Methode erstellt eine Tabelle, in der die Entfernungen zwischen allen Start- und Zielorten aller Personen der aktuellen Personendatei enthalten sind. Auf diese Tabelle, die sich im Hauptspeicher befindet, greift die Bewertungsfunktion während der Bewertung zu.

Zum Aufbau der Tabelle wird eine Methode verwendet, die zu einem Startort die Entfernungen zu allen als Parameter übergebenen Zielorten berechnet. Zur Wegsuche wird dabei der Algorithmus von Dijkstra eingesetzt.

8.7 Dateiverwaltung

8.7.1 Formate

Personendateien und Bewertungsfunktionen sowie Voreinstellungen müssen einem bestimmten Format entsprechen. Haben sie ein anderes Format, so tritt beim Laden ein Fehler auf. Im folgenden werden die Formate aufgeführt. Die GROSS geschriebenen Wörter entsprechen den Einträgen in der Datei. Die Zeilen mit zwei Einträgen geben den Typ und den Namen des Attributes an, wobei der Name in spitzen Klammern steht. Wörter mit und ohne "\" öffnen und schließen eine logische Umgebung (Bsp. REGIONEN ... \REGIONEN).

```
Voreinstellungen:

VOREINSTELLUNGEN
PERSONENDATEI
NAME
string <name>
PFAD
string <pfad>
\PERSONENDATEI
BEWERTUNGSFUNKTION
NAME
string <name>
PFAD
string <pfad>
\BEWERTUNGSFUNKTION
KNOTEN
NAME
string <name>
PFAD
string <pfad>
\KNOTEN
KANTEN
NAME
string <name>
PFAD
string <pfad>
\KANTEN
REGIONEN
NAME
string <name>
PFAD
string <pfad>
\REGIONEN
MAXTEILNEHMERANZAHL
int <maxanzteil>
\VOREINSTELLUNGEN
```

Bei den Voreinstellungen können der Name und/oder der Pfad weggelassen werden. Dafür muß aber pro weggelassenem Eintrag eine Leerzeile stehen. Auf die Knoten, Kanten und Regionen wird im Moment noch nicht zurückgegriffen, da das Laden von Verkehrsgraphen nicht mehr im System vorkommt.

```
Bewertungsfunktion:

BEWERTUNGSFUNKTION
int    <name>
int    <z_max>
int    <u_max>
int    <na_max>
double <e_max>
int    <gewicht_z>
int    <gewicht_u>
int    <gewicht_na>
int    <gewicht_ne>
int    <gewicht_e>
int    <gewicht_p>
\BEWERTUNGSFUNKTION
```

Bei der Bewertungsfunktion müssen alle Parameter angegeben werden.

```
Personendatei:

PERSONENDATEI
PERSONENDATEN
PERSON
int    <id>
string <name>
string <vorname>
string <geschlecht>
int    <tag>              |
int    <monat>            |Geburtsdatum
int    <jahr>             |geburtsdatum
// keine Leerzeile
string <strasse>          |
string <hausnummer>       |Adresse
int    <postleitzahl>     |wohnsitz
string <wohnort>          |
// keine Leerzeile
string <vorwahl>          |Telefonnummer
string <rufnummer>        |telefon
// keine Leerzeile
string <vorwahl>          |Telefonnummer
string <rufnummer>        |fax
// keine Leerzeile
string <email>
string <raucher>
string <fahrer>
```



```
// falls fahrer=TRUE :
string <komfort>          |
int    <plaetze>          |Autodaten
int    <baujahr>          |automobil
// sonst direkt weiter
string <strasse>          |
string <hausnummer>       |Adresse
int    <postleitzahl>     |weginfo.startort
string <wohnort>          |
int    <startkoordinaten>
string <strasse>          |
string <hausnummer>       |Adresse
int    <postleitzahl>     |weginfo.zielort
string <wohnort>          |
int    <zielkoordinaten>
int    <stunden>          |
int    <minuten>          |
int    <sekunden>         |Zeitintervall
int    <stunden>          |weginfo.anzeit
int    <minuten>          |
int    <sekunden>         |
// keine Leerzeile
int    <stunden>          |
int    <minuten>          |
int    <sekunden>         |Zeitintervall
int    <stunden>          |weginfo.rueckzeit
int    <minuten>          |
int    <sekunden>         |
// keine Leerzeile
int    <stunden>          |
int    <minuten>          |Uhrzeit
int    <sekunden>         |weginfo.arbeitsdauer
\PERSON
.
.
.
\PERSONENDATEN
EINTEILUNGEN
EINTEILUNG
FAHRGEMEINSCHAFTEN
FGM
int    <id>
string <markierung>
int    <fahrer>
int    <frei_plaetze>
int    <sekunden>         |
int    <minuten>          |
int    <stunden>          |FGM_Datum
int    <tag>              |datum_fgm
int    <monat>            |
```

```
int    <jahr>            |
// keine Leerzeile
int    <sekunden>        |
int    <minuten>         |
int    <stunden>         |FGM_Datum
int    <tag>             |aenderung_fgm
int    <monat>           |
int    <jahr>            |
ROUTE
// Liste von Kanten_IDs
int    <startkante>
.
.
.
int    <zielkante>
\ROUTE
TEILNEHMER
// Liste von Person_IDs
int    <teilnehmer1>
.
.
.

\TEILNEHMER
STATUS
// Liste von Paaren aus Person_ID und Status
int    <teilnehmer1>
string <status_teilnehmer1>
.
.
.
\STATUS
\FGM
.
.
.
\FAHRGEMEINSCHAFTEN
BEWERTUNGSFUNKTION
// siehe Format fuer Bewertungsfunktion
\BEWERTUNGSFUNKTION
\EINTEILUNG
.
.
.
\EINTEILUNGEN
\PERSONENDATEI
```

Die mit | markierten Umgebungen wie zum Beispiel *Geburtsdatum* können weggelassen werden. An ihrer Stelle muß dann jedoch eine Leerzeile stehen. Alle nicht markierten Attribute können weggelassen werden. An ihrer Stelle muß

dann jedoch eine Leerzeile stehen.

8.7.2 Laden und Speichern

Zum Laden und Speichern wurden hauptsächlich die Operatoren << und >> verwendet. Nachteil dabei ist, daß >> beim Einlesen eines Strings mit einem Blank-Symbol einen Fehler gibt. Außerdem muß man nach jeder >>-Operation das Newline am Zeilenende einlesen. Dazu wird die Funktion *get()* benutzt. Bei Strings, die Leerzeichen enthalten können, wird die LEDA-Methode string.read_line(stream) benutzt, die in einen string aus einem stream eine Zeile ohne abschließenden Zeilenumbruch einliest. Der Zeilenumbruch wird dann übersprungen.

Aufzählungstypen werden von >> zwar geschrieben, können aber von << nicht gelesen werden. Im file steht die Position des Eintrags, die beim Einlesen als Zeichen interpretiert wird. Dann kommt es zu einer inkompatiblen Zuweisung.

Daher werden sämtliche Aufzählungstypen als strings geschrieben und gelesen. Das bereitet keine Probleme.

Hat ein Eintrag keinen Inhalt (*eintrag.ist_gesetzt()==false*), wird eine Leerzeile geschrieben. Dementsprechend wird beim Lesen einer Zeile der Eintrag auf leer gesetzt (*eintrag.nichts()*).

Die Reihenfolge der Attribute entspricht in den meisten Fällen der Reihenfolge in constants.h.

Für den Zugriff auf die Verzeichnisse, den Verzeichniswechsel und das Herausfinden des aktuellen Verzeichnisses werden C-Funktionen benutzt, die im Modul functions.cc gekapselt sind.

Der Code, der verwendet wird, um eine Datei zu speichern wird auch verwendet, um eine Datei unter anderem Namen zu speichern.

Das Laden und Speichern ist in einige Unterfunktionen gegliedert, die sich wieder aufgliedern, bis man beim Laden oder Speichern eines Datensatzes angelangt ist. Dies sind Personen, Fahrgemeinschaften und Bewertungsfunktionen. Dabei wird der Ein- oder Ausgabestream immer mit übergeben. Er muß schreibbar sein, da sich ein Zeiger merkt, an welcher Position man sich innerhalb des streams befindet.

Ein Fehler beim Laden oder Speichern, egal wo er auftritt führt zu einem ALLGEMEINEN FEHLER. Hier wird nicht genauer nach der Ursache gegliedert, da diese sehr unterschiedlich sein kann.

8.8 Erweiterungen

8.8.1 Einhängen von Algorithmen in das System

Das Mobidick-System ist so entworfen worden, daß man mit wenigen Änderungen einen neuen Einteilungsalgorithmus oder einen neuen Wegsuchealgorithmus einfügen kann. Im Folgenden wird beschrieben, worin diese Änderungen bestehen.

8.8.1.1 Einteilungsalgorithmen

Alle Einteilungsalgorithmen erben von der abstrakten Oberklasse *Einteilungs_algorithmus* und alle benötigten Informationen zur Einteilungsberechnung können über die Methoden der Klasse *Algorithmenverwalter* erfragt werden. Der Algorithmenverwalter bildet die Schnittstelle zwischen den Algorithmen und dem System, d.h. alle Anfragen vom System an die Algorithmen werden über Methoden des Algorithmenverwalters gestellt.

Ein Einteilungsalgorithmus sollte mindestens folgende Includes enthalten:

```
#include <stdio.h>
#include <iostream.h>
#include "../constants.h"
#include "Einteilungs_algorithmus.h"
#include "../algorithmenverwalter/Algorithmenverwalter.h"
```

Die abstrakte Klasse *Einteilungs_algorithmus* enthält die virtuelle Methode start(float &laufzeit). Diese Methode muß in jedem neuen Algorithmus implementiert werden. Das System benutzt diese Methode, um einen Algorithmus zu starten. In dem Parameter *laufzeit* gibt der Algorithmus die benötigte Laufzeit zur Berechnung der neuen Einteilung zurück. Zur Laufzeitbestimmung wird die LEDA-Methode used_time() verwendet. Der folgende Programmausschnitt zeigt, wie die Methode eingesetzt wird:

```
//Deklarationen
  ...
  ...
  float zeitpunkt;
  ...
  ...

//Methodenkoerper
  ...
  ...
  zeitpunkt = used_time();          //Beginn der Zeitmessung
  ...
  ...
  //In diesem Abschnitt fuehrt der Algorithmus Berechnungen
  //durch.
  ...
  ...
  laufzeit = used_time(zeitpunkt);  //Ende der Laufzeitmessung
  ...
  ...
```


Die Instanziierung der Einteilungsalgorithmen wird im Konstruktor des Algorithmenverwalters durchgeführt. Das System unterscheidet zwischen drei Grundtypen: heuristische, optimale und inkrementelle Einteilungsalgorithmen. Der Algorithmenverwalter enthält für jeden dieser Algorithmentypen ein LEDA-dictionary, in dem die Algorithmen-ID's und Zeiger auf die Algorithmenobjekte enthalten sind. Im Konstruktor des Algorithmenverwalters werden die instanziierten Algorithmen in das jeweilige LEDA-dictionary eingetragen.

Um einen Einteilungsalgorithmus in das System einzubinden sind drei Schritte notwendig:

1. Ein Zeiger, der auf das Objekt mit dem neuen Algorithmus zeigt, muß deklariert werden.
2. Das Objekt mit dem neuen Algorithmus muß instanziiert werden. Dabei werden als Parameter ein Zeiger auf den Algorithmenverwalter, die ID und der Name des neuen Algorithmus übergeben.
3. Die ID und der Zeiger auf den Algorithmus müssen in das entsprechende LEDA-dictionary eingefügt werden.

Der folgende Programmcode zeigt beispielhaft, wie der MM_Algorithmus in das System eingefügt wurde. In den Kommentaren ist jeweils angegeben, welcher der drei Schritte in der beschriebenen Programmzeile realisiert wird. Der MM_Algorithmus ist ein heuristischer Algorithmus und wird deshalb im Schritt drei in das LEDA-dictionary für die heuristischen Algorithmen eingetragen.

```
Algorithmenverwalter::Algorithmenverwalter(...) : ...
{
  //Schritt 1:
  //Deklaration des Zeigers fuer den neuen Algorithmus
  Einteilungs_algorithmus* eint_algor_zeiger;

  ...
  ...
  ...

  //////////////////////////////////////////
  //heuristische Algorithmen

  //Schritt 2:
  //Instanziierung des MM-Algorithmus
  //            this = Zeiger auf den Algorithmenverwalter
  //               1 = die ID des Algorithmus
  // "MM-Algorithmus" = der Name des Algorithmus
  eint_algor_zeiger = new MM_algorithmus(this,1,"MM-Algorithmus");

  //Schritt 3:
  //MM-Algorithmus mit der ID 1 in das dictionary der
  //heuristischen Algorithmen eintragen
```

```
  heuristische_algor.insert(1, eint_algor_zeiger);

  ...
  ...
  ...
}
```

Wenn ein Algorithmus zusätzliche Parameter wie zum Beispiel die Angabe der Güte der zu berechnenden Einteilung oder den Namen der neu berechneten Einteilung benötigt, so kann der Benutzer mit folgenden speziellen Methoden befragt werden:

`name_fuer_neue_einteilung(...)` Methode zur Abfrage des Namens für die neu berechnete Einteilung.

`integer_parameter(...)` Methode zur Abfrage eines Parameters vom Typ `integer`.

`double_parameter(...)` Methode zur Abfrage eines Parameters vom Typ `double`.

`boolean_parameter(...)` Methode zur Abfrage eines Parameters vom Typ `boolean`.

Die Methoden werden vom Algorithmenverwalter bereitgestellt. Beim Aufruf der Methoden für die `integer`-, `double`- und `boolean`-Parameter kann ein String zur Beschreibung des geforderten Wertes übergeben werden. Diese Beschreibung wird dann dem Benutzer angezeigt, damit dieser den benötigten Wert eingeben kann.

Die Einteilungsalgorithmen werden vom Algorithmenverwalter aufgerufen. Um dem System mitzuteilen, ob die Berechnung erfolgreich war, oder ob ein Fehler aufgetreten ist, gibt jeder Einteilungsalgorithmus einen Fehlercode zurück. Wurde von dem Einteilungsalgorithmus eine neue Einteilung berechnet und angelegt, so müssen in dieser auch Informationen über die verwendete Bewertungsfunktion abgespeichert werden. Diese Aufgabe wird vom Algorithmenverwalter übernommen und muß deshalb nicht vom Einteilungsalgorithmus erledigt werden. Damit der Einteilungsalgorithmus entscheiden kann, ob eine neue Einteilung angelegt wurde, werden folgende Fehlercodes erwartet. Nach erfolgreicher Durchführung einer Berechnung und Bildung einer neuen Einteilung, wird der Fehlercode *OK* zurückgegeben. Wurde die Berechnung erfolgreich durchgeführt, aber keine neue Einteilung gebildet, so ist der geforderte Fehlercode *KEINE_NEUE_EINTEILUNG*, ist bei der Berechnung ein Fehler aufgetreten, dann wird der dazu passende Fehlercode zurückgegeben.

8.8.1.2 Wegsuchealgorithmen

Das Einhängen von Wegsuchealgorithmen funktioniert im Wesentlichen genauso, wie das in Abschnitt 8.8.1.1 beschriebene Einhängen von Einteilungsalgorithmen. Die Unterschiede sind:

- Alle Wegsuchealgorithmen erben von der abstrakten Oberklasse *Wegsuche_algorithmus*.
- Die minimalen Includes enthalten nur die Oberklasse.
- Die virtuelle Methode ist `list<int> suche(int& fahrtzeit)`, `list<` $int>$ ist eine LEAD-Liste mit den Kanten-ID's des berechneten Weges und fahrtzeit gibt an, wieviele Minuten benötigt werden, um den Weg zurückzulegen.
- Es gibt keine Unterteilung der Wegsuchealgorithmen in Grundtypen, aber zusätzlich zum Eintrag in das LEDA-dictionary der Wegsuchealgorithmen ist ein Eintrag in eine ID-Liste und eine Namensliste nötig. Das folgende Programmstück zeigt beispielhaft, wie das Einhängen eines Wegsuchealgorithmus aussehen kann:

```
Algorithmenverwalter::Algorithmenverwalter(...) : ...
{
  ...
  ...
  ...
  ///////////////////////////
  // Dijkstra-Algorithmus

  //Den Algorithmus anlegen
  Dijkstra* dijk_zeiger =
                new Dijkstra(vg, "Dijkstra-Algorithmus", 1);

  //Den Algorithmus mit der ID 1 in das dictionary eintragen
  wegsuche_algor.insert(1, dijk_zeiger);

  //Die ID des Algorithmus in die Liste der ID's eintragen
  weg_id_liste.append(1);

  //Den Namen des Algorithmus in die Liste der Namen
  //eingragen
  weg_namen_liste.append("Dijkstra-Algorithmus");

  ...
  ...
  ...
}
```

Kapitel 9

Weiterentwicklung

9.1 Erweiterungsmöglichkeiten

Das System Mobidick wurde so entwickelt, daß Erweiterungen leicht möglich sind. In erster Linie wurde daran gedacht, Algorithmen zur Wegsuche und zur Einteilungsberechnung zu entwickeln und diese dann ins System zu stellen. Dazu wird eine abstrakte Klasse angeboten, von denen man einen neuen Algorithmus erben lassen kann.

Als Erweiterungen des bestehenden Systems bieten sich natürlich die Funktionen an, die in der Kürze des Projekts nicht haben realisiert werden können, in der Spezifikation jedoch vorgesehen sind. Dies sind die Postscript-Ausgabe und die Druckfunktion sowie diverse Sicherheitsabfragen, die in einem System im realen Einsatz sicherlich vorhanden sein sollten.

Darüber hinaus ist es recht einfach möglich, die textuellen Menüs durch eine graphische Benutzungsoberfläche zu ersetzen: Das Modul *Menüverwalter* kapselt sämtliche Ausgaben und hat eine definierte Schnittstelle zum *Fürsorger*, der dann den Rest des System kennt.

Ebenso ist es denkbar und bei großen Datenmengen auch sinnvoll, die Datenverwaltung, die momentan noch mit typisierten Dateien auf der Festplatte arbeitet, in eine Datenbank auszulagern. Dazu müsste das Modul *Lader/Speicherer* mit einer Datenbankschnittstelle gekoppelt werden.

In späteren Versionen von Mobidick ist die Ausgabe relevanter Informationen in Postscript-Dateien und eventuell auch direkt auf einen Postscriptdrucker geplant. Die vorliegende Version benötigt keinen Drucker.

9.2 Kritische Betrachtungen

Der Prototyp Mobidick zeigt auf, welche Grenzen dem Einsatz eines solchen Systems gesetzt sind. Die Bedienung des Programms ist durch die rein textuellen Menüs recht mühsam. Hier wäre eine graphische Oberfläche sicherlich

wünschenswert. Die zu berechnenden Probleme sind hochgradig komplex, so daß mit den verwendeten Algorithmen bei einem Einsatz mit 1000 Personen mit mehrstündigen Laufzeiten zu rechnen ist. Da für einen systematischen Test die Zeit fehlte, ist die Qualität des Systems nicht gesichert.

Das System benötigt auch sehr viel Hauptspeicher, besonders durch die Konzentrierung auf schnelle Laufzeit beim Entwurf der Algorithmen.

Als positiv ist die modulare Architektur zu bewerten. Sie erleichterte die Integration der einzeln entwickelten Subsysteme („Verwalter“). Neue Algorithmen können schnell ins System eingehängt werden. Auch die Bewertungfunktion ist als Dienstleister für die Algorithmen gut ausgereift.

Kapitel 10

Rückblick

Im folgenden wird ein kritischer Rückblick in Bezug auf die einzelnen Phasen der Projektgruppe vorgenommen. Er macht deutlich, welche Erfahrungen und Lerneffekte stattgefunden haben und wird deswegen nachfolgenden Projektgruppen ans Herz gelegt.

10.1 Seminarphase und Vorträge

Die Seminarphase hatte zum Ziel, die Teilnehmer der Projektgruppe auf das anstehende Problem vorzubereiten. Durch den späten Beginn und die lange Einarbeitungszeit dauerte sie zu lange. Außerdem war die Auswahl der Themen zu wenig problemspezifisch. Fachvorträge zum Thema Programmiersprache, Software Engineering und Entwicklungsumgebung wären wünschenswert gewesen.

10.2 Anforderungsanalyse und Spezifikation

Diese frühen Phasen wurden sehr ausführlich und sorgfältig gemacht, wovon beim Zwischenbericht profitiert wurden. Es verzögerte sich jedoch der Zeitplan. Die Entscheidung über die Programmiersprache fiel erst spät. Es wurde kein Prototyp mit dem Benutzungsinterface erstellt. Dadurch fiel diese Arbeit in die Implemtierung und verzögerte diese weiter.

10.3 Zwischenbericht

Der Zwischenbericht entstand in der Anfangsphase des Entwurfs. Die bisher entstandenen Dokumente - Anforderungsanalyse und Spezifikation - gingen ein, so daß dieser ohne große Komplikationen fertig gestellt werden konnte.

10.4 Entwurf

In der Entwurfsphase wurde viel Zeit am Anfang verloren. Es fehlten detailliertes Wissen über die Vorgehensweise. Die Schnittstellen im Grobentwurf wurden zum Großteil eingehalten und wirkten sich positiv auf die Integration aus.

10.5 Implementierung

Da die Sprachentscheidung sehr spät getroffen wurde, begann die Implementierungsphase mit einer relativ langen Einarbeitungszeit in die Sprache und die verwendete Entwicklungsumgebung, deren Möglichkeiten nur teilweise genutzt, deren Unzulänglichkeiten jedoch als störend empfunden wurde. Hier empfiehlt die Projektgruppe die rechtzeitige Auswahl einer Sprache mit einer integrierten Entwicklungsumgebung und einer gemeinsamen Einarbeitung (z.B. durch einen entsprechenden Seminarvortrag).

Richtlinien für die Kodierung wurden nicht immer genau eingehalten.

Eine schnelle Implementierung eines Prototyps mit Benutzungsoberfläche war in früheren Phasen versäumt worden. Dadurch wurde die Oberfläche parallel zur Funktionalität entwickelt, was zu Integrations- und Zeitproblemen führte.

10.6 Test und Integration

Die Integration funktionierte erstaunlich gut, nicht zuletzt durch die im Grobentwurf geklärten Schnittstellen. Kleinere Probleme wurden entweder in der Projektgruppe oder zwischen den Betroffenen gelöst. Ein systematischer Test hat nicht stattgefunden.

10.7 Allgemeines

Die Abwicklung des Projekts in starren Phasen erwies sich als ungeeignet. In den frühen Phasen wurden viele Fehler durch Unkenntnis der späteren gemacht.

Für die einzelnen Phasen gab es zwar einen Verantwortlichen. Jedoch war er zum einen selbst in die Ausführung der Phase verstrickt, zum anderen fehlte es zum Teil an der Koordination zur Projektleitung. Dadurch ergaben sich Verzögerungen im Zeitplan.

Zu empfehlen ist eine gemeinsame Urlaubsplanung. Ein gewisses Zeitkontigent einiger Wochen für jeden Teilnehmer, das er dann im Laufe der Projektgruppe für Urlaub und Prüfungen in Anspruch nehmen kann, bringt Klarheit in den erwarteten Arbeitsaufwand. Wenn von Anfang an klar ist, daß die Projektgruppe nicht Ende Juli endet, sollte dies auch nicht in der Ankündigung stehen.

Für den Fall, daß die gestellte Aufgabe zu Beginn des nächsten Wintersemesters nicht vollständig gelöst ist, sollten von Anfang an klare Überlegungen existieren. Wird in dem Fall abgebrochen oder weitergearbeitet?

Anhang A

Bedienung des Tools gdf2graph

Hier soll kurz beschrieben werden, wie mit Hilfe der Perlskripte in `gdf2graph/perlskripte` des Tools gdf2graph ein GDF-Datensatz in das von uns verwendete Graphenformat konvertiert werden kann. Dazu muß das `.gdf`-File ausgepackt im gleichen Verzeichnis wie die Perlskripte liegen (z.B. `stuttg.gdf`). Danach werden vier Perlskripte in folgender Reihenfolge aufgerufen:

```
perl one_line_records.pl
perl gauss_krueger.pl
perl graphnodes.pl
perl graphedges.pl
```

Neben einigen Zwischenfiles mit den Endungen `.gdf` und `.gk` entstehen die Files `edges.graph` und `nodes.graph` mit den Kanten- und Knotendaten des Verkehrsgraphen.

Anhang B

Programmkodierung wichtiger Methoden

Im folgenden werden einige wichtige Methoden in Ausschnitten präsentiert. Hier ersieht man typische Vorgehensweisen und kann sich ein Bild von groben Strukturen und Zusammenhängen machen. Die Texte sind aus dem Quellcode kopiert.

Über den Fürsorger läuft die Kommunikation zwischen den Menüs und dem Rest des Systems. Wird am Anfang eine neue Personendatei angelegt, ergeben sich die Aufrufe, die im folgenden aufgeführt sind. Der Pfeil `->` gibt an, daß der Rumpf der aufgerufenen Methode im weiteren aufgelistet wird. Nach dem Wort *Fehlercode* steht der Name der Klasse, die die Methode zur Verfügung stellt.

```
  do{
    switch (aktion){
    case HAUPT_1:
      aktion=dateien.warte();
      do{
        switch (aktion) {
        case DATEIEN_1:
          aktion= person_dat.warte();
          do{
            switch (aktion) {
            case PERSON_DAT_1:
->            fehlercode=menueverwalter->tuwas(aktion);
              aktion= person_dat.warte();
              break;
...

Fehlercode Menueverwalter::tuwas(Aktionen aktion){
  Fehlercode fehler;
  Fehlercode F; // nur fuer doktor
  string fehlerstring;
  Doktor doktor;
```

```
  switch (aktion){
  case PERSON_DAT_1:
->  fehler = fuersorger->neue_personendatei();
    break;
...

Fehlercode Fuersorger::neue_personendatei()
{
  if (personendatei_geaendert==TRUE)
    return NICHT_GESPEICHERT;
->Fehlercode fehlercode=dateiv->neue_datei(PERSONENDATEI);
  if (fehlercode==OK)
    personendatei_geoeffnet=TRUE;
  // personendatei_geaendert ist FALSE
  return fehlercode;
}

Fehlercode Lader_speicherer::neue_datei(const Dateiart art)
{
  Fehlercode fehlercode;
  string     datei;
  ofstream   stream;

->fehlercode=datei_auswahl->selektiere(art,SCHREIBEND);
  if (fehlercode!=OK)
    return fehlercode;
...

Fehlercode Datei_auswahl::selektiere(const Dateiart art,const Dateizugriff zugri
ffsart)
{
  list<string> inhalt,inhalt_eingeschraenkt;
  string       verzeichnis;
  Fehlercode   fehlercode;
  Boolean      nochmal=TRUE;

  inhalt.clear(); inhalt_eingeschraenkt.clear();

  switch (art)
    {
    case PERSONENDATEI:
      verzeichnis = akt_per_pfad;
      break;
    case BEWERTUNGSFUNKTION:
      verzeichnis = akt_bew_pfad;
      break;
    default:
      return FALSCHE_DATEIART;
      break;
```



```
    }

  if (list_contents_dir(verzeichnis,inhalt)==FALSE)
    return ALLGEMEINER_FEHLER;
  einschraenken(inhalt,art,inhalt_eingeschraenkt);

  cout << endl << "Dateiauswahl: selektieren" << endl << endl;

  switch (zugriffsart)
    {
    case LESEND:
      strings_numeriert_ausgeben(inhalt_eingeschraenkt);
      break;
    case SCHREIBEND:
      strings_ausgeben(inhalt_eingeschraenkt);
      break;
    default:
      return FALSCHE_DATEIART;
      break;
    }
  while (nochmal==TRUE)
    {
      datei_ausgeben(art);
      if (datei_einlesen(art,zugriffsart)==TRUE)
        nochmal=FALSE;
    }
  return OK;
}
```

Ein Großteil der Verwaltungsarbeit liegt beim Einteilungs- und Personenverwalter. Nun ein Beispiel für das Zusammenspiel von Menü und Verwalter mit Hilfe von Strukturen. Eine neue Person wurde eingelesen und wird jetzt dem Personenverwalter in Form einer Struktur *Personendaten* übergeben.

```
//Methode zum Anlegen einer Person
Fehlercode Personen_daten::anlegen(){

  struct Personendaten neueperson, person;
  Fehlercode fehler;

  cout << "Geben Sie nun die Daten fuer einen neue Person ein."
       << endl;
  cout << "Bitte beachten Sie: ein * hinter der Frage bedeutet, "
       << endl;
  cout << "dass ein Wert eingegeben werden muss." << endl;

  neueperson = lese_Personendaten(person, FALSE);


  // Uebergeben der Person an den Fuersorger
```

```
->fehler = fuersorger->neue_person(neueperson);
...

//Fuegt eine neue Person ein.
Fehlercode Personenverwalter::neue_person(Personendaten person)
{
  dic_item di;
  Fehlercode muell_voll;

  //Objekt Person wird angelegt
  personen_zeiger = new Person();
  //ID der Person wird berechnet
  per_id++;
  //Vergabe der ID
  person.personen_id = per_id;
  //Personendaten werden gesetzt, dabei wird geprueft, ob die Daten
  //vollstaendig sind
->muell_voll = personen_zeiger->setze_personendaten(person);

  if (muell_voll == NICHT_VOLL)
    {
      //Personenobjekt wird wieder geloescht
      personen_zeiger->loesche_personendaten();
      delete personen_zeiger;
      return NICHT_VOLL;
    }
  else
    {
      //Die id der Person wird in die Liste personen_id_liste eingefuegt.
      personen_id_liste.push(per_id);

      //Der Zeiger auf ein Personobjekt wird in Dictionary
      //personen_zeiger_dic eingefuegt.
      di = personen_zeiger_dic.insert(per_id,personen_zeiger);

      return OK;
    }
}

//Setzt die Personendaten und die Objekte Auto, Weginformation
//und Wuensche werden neu angelegt und mit den entsprechenden
//Daten versorgt.
Fehlercode Person::setze_personendaten(Personendaten personen_daten)
{
  Boolean ist_vollstaendig;

  id_person = personen_daten.personen_id;

  name_person = personen_daten.name;
  vorname_person = personen_daten.vorname;
```



```
  geschlecht_person = personen_daten.geschlecht;
  geburtsdatum_person = personen_daten.geburtsdatum;
  wohnort_person = personen_daten.wohnsitz;
  telefon_person = personen_daten.telefon;
  email_person = personen_daten.email;
  fahrer_person = personen_daten.fahrer;
  raucher_person = personen_daten.raucher;

  //*** solange bis das Menue es unterstuetzt, dann muss oben auch
  //const wieder eingefuegt werden.
  Optional<Personenliste> abneigung;
  Optional<Personenliste> zuneigung;
  abneigung = personen_daten.abneigung_zu_personen;
  zuneigung = personen_daten.zuneigung_zu_personen;
  personen_daten.wunschinfo.abneigung_zu_personen = abneigung;
  personen_daten.wunschinfo.zuneigung_zu_personen = zuneigung;
  //*** solange bis das Menue es unterstuetzt

  //Objekt Auto wird angelegt und Daten werden gesetzt
  auto_zeiger = new Auto();
  auto_zeiger->setze_autodaten(personen_daten.automobil);
  //Objekt Weginformation wird angelegt und Daten werden gesetzt
  weg_zeiger = new Weginformation();
  weg_zeiger->setze_weginformationsdaten(personen_daten.weginfo);
  //Objekt Wunsch wird angelegt und Daten werden gesetzt
  wunsch_zeiger = new Wunsch();
  wunsch_zeiger->setze_wunschdaten(personen_daten.wunschinfo);

  ist_vollstaendig = pruefe_vollstaendigkeit();

  if (ist_vollstaendig == TRUE)
      return OK;
  else
    return NICHT_VOLL;
}
```

Literaturverzeichnis

[1] Grady Booch. *Objektorientierte Analyse und Design.* Addison Wesley, 1994.

[2] H. Claussen et al. GDF 2.1 Draft Standard. unpublished, 1992.

[3] Robert M. Haralick and Linda G. Shapiro. *Computer and robot vision 1+2.* Addison-Wesley, 1992.

[4] Herbert Heid, Daniela Nicklas, Alexander Porrmann, Thomas Schäffer, and Volker Scholz. *Zwischenbericht der Projektgruppe Fahrgemeinschaften (FGM).* Universität Stuttgart Institut für Informatik, 1997.

[5] FN/MAts Henricson and Erik Nyquist. *Programming in C++, Rules and Recommendations.* ELLEMTEL, 04 1992.

[6] IEEE Computer Society (Hrsg.). IEEE guide for software requirements specification. *IEEE Std 830-1984*, 1984.

[7] Jochen Ludewig. Grundlagen des Software Engineerings. Fachschaft Informatik, 1997. Skript zur Vorlesung im Sommersemester 1997.

[8] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. *The LEDA User Manual.* Max-Planck-Institut für Informatik, Saarbrücken, Martin-Luther Universität Halle-Wittenberg, 1997. Version R 3.4.

[9] Ralf Melchisedech. Spezifikation für das Software-System SA-Mini. Institut für Informatik, 1997.

[10] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*, pages 232–242. Reihe Informatik Band 70. BI-Wissenschaftsverlag, 1993.

[11] H. Bässmann und W. Besslich. *Bildverarbeitung Ad Oculos.* Springer-Verlag, 1993. 54-63.