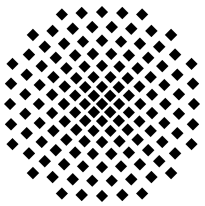
Endbericht der
Projektgruppe
Transportoptimierung
Bericht Nr. 1998/10



Universität Stuttgart

Endbericht der Projektgruppe Transportoptimierung

Jörg Fleischmann
Lars Hermes
Tobias Spribille
Frank Wagner

Betreuung
Prof. Dr. Volker Claus
Dipl.-Inform. Friedhelm Buchholz
Dipl.-Inform. Stefan Lewandowski

Abteilung Formale Konzepte
Fakultät Informatik
Universität Stuttgart

9. November 1998

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

Kapitel 1

Einführung

1.1 Der Bericht

Dieses Dokument ist nicht als alleinstehender Bericht aufzufassen, sondern knüpft direkt an den Zwischenbericht der Projektgruppe Transportoptimierung [Pro98] an. Beide Berichte zusammen ergeben einen kontinuierlichen Überblick über die Arbeit der Projektgruppe von Oktober 1997 bis September 1998. Die Lektüre dieses Endberichts setzt also teilweise die Kenntnis des Zwischenberichts voraus. Begriffe, die bereits im Zwischenbericht ausführlich eingeführt und definiert wurden, werden hier ohne erneute Definition benutzt. Leser, die daran interessiert sind, woraus z.B. die praxisnahe Lehrveranstaltungsform der Projektgruppe im einzelnen besteht, können dies im Zwischenbericht nachlesen.

Der Endbericht setzt dort an, wo der Zwischenbericht endet: Beim Entwurf. Teile des Entwurfs, die erst nach dem Zwischenbericht fertiggestellt wurden, oder deren Notwendigkeit sich gar erst während der Implementierungsphase ergab, sind hier festgehalten. Danach folgt der Bericht über die wichtigsten Aspekte der Projektphasen Implementierung und Test (dem sicherlich nicht die gebührende Aufmerksamkeit geschenkt wurde, aber Zeitprobleme scheinen offensichtlich zum Wesen einer Projektgruppe zu gehören). Eine Übersicht über das entstandene Programm aus Programmierersicht zeigt Kapitel 5 mit einer schematischen Darstellung der Architektur des Systems, die Benutzerseite beschreibt die Bedienungsanleitung in Kapitel 7.

Ein solch langes und aufwendiges Projekt soll natürlich nicht ohne ein abschließendes Fazit bleiben: Möglichkeiten zur Erweiterung des Programms wurden zusammengestellt (möglicherweise als Anregung für folgende Projektgruppen), dem theoretische Zeitplan wird der tatsächliche Ablauf des Projekts gegenübergestellt und schließlich faßt ein Rückblick einige gute und weniger gelungene Aspekte der Projektgruppe zusammen, um daraus Empfehlungen für kommende Projektgruppen (oder ganz allgemein Programmierteams) abzuleiten.

1.2 Das Programm

Eines der Ziele der Projektgruppe war die Erstellung eines Programms zur Verwaltung und Planung der sozialen Fahrdienste des DRK in Bad Cannstatt. Wenn dieses Ziel auch etwas zu hoch gesteckt war (was in diesem Bericht noch aus-

führlicher geschildert werden wird), ist trotzdem ein Programm entstanden, das die wichtigsten Funktionen zur Verwaltung und manuellen Organisation der Daten des DRK zur Verfügung stellt. Über eine graphische Benutzungsschnittstelle läßt es sich komfortabel bedienen.

Das Programm trägt den Namen TROSS, was bereits eine gewisse Bedeutung in sich trägt, vor allem jedoch als Abkürzung zu verstehen ist: TRansport Organisation for Social Services, oder auch Transport-Organisation für Soziale Serviceanbieter.

Kapitel 2

Nachtrag zum Entwurf

2.1 Neue Klassen und Interfaces

2.1.1 Arbeitszeitprofile

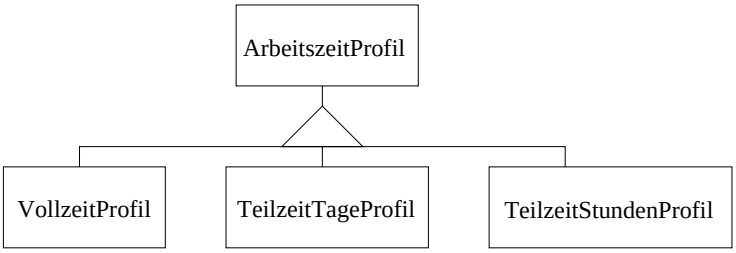
Bei der Einteilung von Mitarbeitern für Touren und Fahrten sollte überprüft werden können, ob der Mitarbeiter zur vorgesehenen Zeit überhaupt eingesetzt werden kann. Dabei soll zum einen berücksichtigt werden, an welchen Tagen bzw. zu welchen Zeiten ein Mitarbeiter prinzipiell im Dienst ist, zum anderen gibt es nicht nur für die wöchentliche, sondern auch für die tägliche Arbeitszeit Grenzen, so daß nicht immer der bestpassende Mitarbeiter eingesetzt werden kann.

Im Gespräch mit dem Benutzer am Ende der Entwurfsphase stellte sich daher die Notwendigkeit heraus, die Mitarbeiter bezüglich ihrer Arbeitszeit zu klassifizieren. Diesem Zweck dienen drei Arten von Arbeitszeitprofilen, deren einzelne Parameter frei definierbar sind und somit unterschiedlichste Einsatzzeiten beschreiben können:

Vollzeitkräfte Diese Mitarbeiter sind prinzipiell „immer“ verfügbar, eine wöchentliche sowie eine maximale tägliche Arbeitszeit sind die Grenzen. Dies sind Soll-Vorgaben, die im Einzelfall durch den Benutzer außer Kraft gesetzt werden können.

Tageweise beschäftigte Teilzeitkräfte Solche Mitarbeiter arbeiten nur an manchen Wochentagen für das DRK und/oder nur zu gewissen Zeiten an diesen Tagen.

Teilzeitkräfte auf Stundenbasis Hierunter fallen Mitarbeiter, die nach Stunden bezahlt werden und meist eine bestimmte Obergrenze nicht überschreiten dürfen (sogenannte „620-Mark-Jobs“). Auch hier muß auf eine Obergrenze für die tägliche Arbeitszeit geachtet werden.



ArbeitszeitProfil: abstrakt

- Attribute

  bezeichnung: String Bezeichner, mit dem verschiedene Arbeitszeitprofile identifiziert und auseinandergehalten werden können.

- Methoden

  abstrakt art: int Die Unterklassen geben hier jeweils ihre Art zurück

VollzeitProfil: ArbeitszeitProfil

- Attribute

  zeitProWoche: int wöchentliche Arbeitszeit in Minuten

  maxZeitProTag: int Höchstgrenze für die tägliche Arbeitszeit in Minuten

TeilzeitTageProfil: ArbeitszeitProfil

- Attribute

  zeitProTag: Zeitspanne[] Für jeden Wochentag die Zeitspanne, an der der Mitarbeiter arbeitet

  maxZeitProTag: int Höchstgrenze für die tägliche Arbeitszeit in Minuten. Eine Zahl, die für alle Wochentage gleichermaßen gilt.

TeilzeitStundenProfil: ArbeitszeitProfil

- Attribute

  zeitProMonat: int monatliche Arbeitszeit in Minuten

  maxZeitProTag: int Höchstgrenze für die tägliche Arbeitszeit in Minuten

Sämtliche Werte sind Mußwerte.

Diese drei Arbeitszeitprofile stellen einen Kompromiß dar zwischen Erfassung aller Möglichkeiten von Teilzeitarbeit und sinnvollem Aufwand bei der programminternen Realisierung (und auch der Dateneingabe). Ein Mitarbeiter, der nur Montags oder Dienstags eingesetzt werden kann, aber eine monatliche Arbeitszeit von 50 Stunden hat, kann mit den vorliegenden Arbeitszeitprofilen nicht dargestellt werden. Eine Annäherung könnte hier durch ein TeilzeitTageProfil erreicht werden, das für Montag und Dienstag eine 5,5 Stunden lange Zeitspanne enthält.

2.1.2 DienstplanErstellbar

Das DienstplanErstellbar-Interface dient als einheitliche Schnittstelle aller Objekte, für die ein Dienstplan erstellt werden kann. Dies sind die Objekte Mitarbeiter, Fahrzeug und Kunde. Mit dem „Dienstplan“ für Kunden kann der DRK-Einsatzplaner schnell feststellen, wann bei einem bestimmten Kunden Dienste verrichtet werden.

Um das DienstplanErstellbar-Interface benutzen zu können, müssen folgende Methoden implementiert werden:

Fahrt[] dienstplanFahrten(Datumsspanne) Liefert alle Fahrten für das Objekt innerhalb der übergebenen Datumsspanne, sortiert nach Datum, zurück.

String dienstplanName() Liefert einen Namen bzw. eine Bezeichnung für das Objekt zurück, für welches der Plan erstellt wird (z.B. Mitarbeitername).

String dienstplanBezeichnung() Gibt die Überschrift zurück, die der Dienstplan haben soll.

2.1.3 Zahler-Interface

Das Zahlerinterface legt eine einheitliche Schnittstelle für alle Objekte fest, die als Rechnungsempfänger in Frage kommen. Dazu stehen folgende Methoden zur Verfügung:

String zahlerName() Gibt den nicht notwendigerweise eindeutigen Namen des Zahlers zurück.

BankVerbindung[] zahlerBankVerbindungen() Gibt alle Bankverbindungen des Zahlers zurück.

2.1.4 Druckbar

Für den mehrseitigen Ausdruck benötigt das druckbare Component einen Printjob, um den Seitenvorschub selbst vornehmen zu können. Wird das Interface Druckbar implementiert, ist das implementierende Component für den Ausdruck völlig selbstverantwortlich, d.h. es bekommt nur einen PrintJob und muß daraus nach Bedarf Graphics-Objekte erzeugen und freigeben. Insbesondere ist darauf zu achten, daß auch das letzte Graphics-Objekt wieder mit `dispose()` freigegeben wird.

Um das DienstplanErstellbar-Interface benutzen zu können, muß folgende Methode implementiert werden:

void drucken(PrintJob) initiert den evtl. mehrseitigen Ausdruck des implementierenden Components.

2.2 Allgemeine Konzepte

2.2.1 Stationen und Anbindungspunkte

Um Adressen für Anfragen an das Verkehrstool benutzen zu können, hat jede Station eine Referenz auf einen Anbindungspunkt. Dies ist eine Adresse im For-

mat des Verkehrstools, die entweder mit der Adresse der Station übereinstimmt oder in deren Nähe liegt, falls die Stationsadresse selbst im Verkehrstool nicht bekannt ist.

Da manche Adressen recht oft benötigt werden (z.B. Schulen, zu denen eine größere Anzahl Kunden befördert werden soll), wurde als Eingabehilfe das Konzept der *benannten Stationen* entwickelt. Jeder Station kann optional ein Name zugewiesen werden, über den dann später ohne erneute Eingabe der gesamten Adresse auf diese Station zurückgegriffen werden kann.

Die im Szenario gespeicherten Stationen lassen sich dadurch in zwei Gruppen einteilen:

1. Stationen ohne Namen stehen für beliebige einmalige Adressen. Dabei können mehrere Stationsobjekte zu einer Adresse existieren, falls z.B. zwei Kunden im selben Haus wohnen oder ein Kunde mehrere Dienstwünsche mit derselben Adresse hat.

2. Benannte Stationen sind über ihren Namen eindeutig unterscheidbar. Haben z.B. mehrere Kunden dasselbe Ziel, verweisen zwei Dienstwünsche auf dieselbe benannte Station.

Die Eingabe von Stationen geht folgendermaßen vor sich:

- Der Benutzer gibt die Adresse einer Station ein.

- Hierzu kann er aus den bereits im System bekannten benannten Stationen auswählen. Name und Adresse der gewählten Station werden ggf. in die Eingabemaske eingetragen.

- Beendet der Benutzer die Eingabe, wird zunächst ein Anbindungspunkt zur Station ermittelt. Ist die Station benannt und bereits im Szenario bekannt, wird diese bekannte Station mitsamt ihrem bereits bekannten Anbindungspunkt benutzt. Ansonsten muß der Anbindungspunkt mit Hilfe des Verkehrsmoduls bestimmt werden.

  Hierzu wird die eingegebene Adresse beim Verkehrstool angefragt, welches einen, keinen oder mehrere alternative Anbindungspunkte zurückgibt. Die letzten beiden Fälle erfordern eine Rückfrage beim Benutzer. Dieser hat die Möglichkeit, aus den angegebenen Alternativen eine zu wählen, oder er gibt eine weitere Adresse für den Anbindungspunkt ein, die wiederum vom Verkehrstool geprüft wird.

  Konnte der Station auf diese Weise ein Anbindungspunkt zugeordnet werden, muß noch unterschieden werden, ob es sich um eine benannte Station handelt:

  - Ist kein Name angegeben, wird eine neue Station erzeugt und im Dienstwunsch gespeichert.
  - Wurde ein Name eingegeben, wird diese Station dem Szenario gemeldet. Sollte ein Konflikt mit einer bereits bekannten Station auftreten, wird der Benutzer darüber informiert und kann seine Eingabe ändern. Sonst wird eine neue benannte Station angelegt, der Dienstwunsch speichert nur eine Referenz darauf.

2.2.2 Untertour

Die Schnittstelle der Untertour hat sich nachträglich geändert. Einzelne Änderungen werden in den folgenden Abschnitten beschrieben.

- Attribute

  nummer : String die Nummer der Untertour (a . . . zz).

  bezeichnung : String Eine frei vergebbare Bezeichnung für die Untertour. Die Bezeichnungen der Untertouren einer Tour müssen verschieden sein.

  tour : Tour ein Verweis auf die Tour

  teilWuensche : Vector (of TeilDienstwunsch) Hier werden alle Teildienstwünsche gespeichert, die in dieser Untertour erfüllt werden sollen.

  anfangsStation : Station Gibt eine Station an, die immer als erste angefahren werden muß, z.B. die Küche bei einer Essen-Tour.

  endStation : Station . Gibt eine Station an, die unbedingt als letzte angefahren werden muß.

  anfangsZeit : Uhrzeit Die Zeit, zu der die Untertour beginnt.

  endZeit : Uhrzeit Die Zeit, zu der die Tour beendet wird.

  halte : Vector (of UntertourHalt) In diesem Vector steht, in welcher Reihenfolge welche Halte anzufahren sind.

  fahrten : Vector (of Fahrt) Die erstellten Fahrten zu der Untertour sortiert nach Datum.

- Methoden

  anfangsUndEndZeitAnpassen() Wenn die `anfangsZeit` nach der `ankunftsZeit` des ersten Haltes liegt, wird sie auf diese `ankunftsZeit` gesetzt. Entsprechend wird mit der `endZeit` und der `abfahrtsZeit` des letzten Haltes verfahren.

  sucheErsteFahrt(Datum) : Fahrt liefert die erste Fahrt, die an oder nach dem angegebenen Datum stattfindet.

  fahrtenInZeitraum(Zeitraum) : Fahrt[] liefert ein Array aller Fahrten im angegebenen Zeitraum. Noch nicht erzeugte Fahrten werden vorher erstellt.

  fahrtAnTag(Datum) : Fahrt liefert die Fahrt an dem angegebenen Tag zurück, wenn es sie schon gibt. Sonst wird null zurückgegeben.

  erstelleFahrten(Zeitraum) : Fahrt[] erstellt noch fehlende Fahrten im angegebenen Zeitraum und gibt diese auch zurück, um sie z.B. dem Benutzer zur Kontrolle anzeigen zu können.

  fahrtenSchreibenUndLoeschen(Datum, PrintWriter) Wenn es eine Fahrt an dem angegebenen Tag gibt, wird eine Info-Zeile in den PrintWriter geschrieben und die Fahrt anschließend gelöscht.

  loescheFahrtenAb(Datum) löscht alle Fahrten ab dem angegebenen Datum, ohne sie zu archivieren.

fahrdauer(int von, int nach) liefert die Fahrzeit in Minuten zwischen den zwei durch ihre Position angegebenen Halten. Dies ist die Differenz von Ankunftszeit bei nach und Abfahrtszeit bei von.

aufenthaltsDauer(int) Liefert die Aufenthaltszeit in Minuten an dem durch seine Position gegebenen Halt.

aendereAufenthaltsDauer(int, int) Ändert die Aufenthaltsdauer an dem angegebenen Halt um das als zweites Argument angegebene Delta.

aendereAufenthaltsDauerAbsolut(int, int) Setzt die Aufenthaltsdauer an dem angegebenen Halt auf die als zweites Argument angegebene Dauer.

pruefeHalteReihenfolge(Vector) : boolean Prüft, ob die Reihenfolge der Stationen im Vector mit der von den Dienstwünschen der Untertour vorgesehenen Reihenfolgeübereinstimmt.

korrigiereFahrzeiten(Vector) Setzt die Fahrzeiten zwischen den Halten im angegebenen Vector auf die vom Verkehrsmodul gelieferten Werte. An- und Abfahrtszeiten werden entsprechend angepaßt. Klappt dies nicht (weil z.B. Map&Guide nicht läuft), wird eineUnbekannteFahrzeitenException, die die fehlenden Strecken enthält, geworfen.

verschiebeHalt(int, int) Verschiebt den Halt an der angegebenen Position (erstes Argument) an eine neue Position (zweites Argument). Alle Dienstwünsche an dem Halt werden mit verschoben. Gibt es an der angegebenen Position schon einen Halt mit der gleichen Station, werden die beiden Halte verschmolzen.

verschiebeDienstwunsch(Dienstwunsch, int, int) Verschiebt einen Dienstwunsch aus dem UntertourHalt an der alten Position (zweites Argument) zur neuen Position (drittes Argument). Gibt es an dieser Position schon einen Halt mit der Station des zu verschiebenden Dienstwunsches, wird der Dienstwunsch an diesem Halt erfüllt, ansonsten wird ein neuer UntertourHalt angelegt.

dienstwuensche() : Dienstwunsch[] Liefert ein Feld mit den Dienstwünschen, die sich aus den Teildienstwünschen ergeben.

teilDienstwunschHinzufuegen(TeilDienstwunsch) Nimmt den TeilDienstwunsch mit seinen Stationen zur Untertour hinzu. Bei einer Rückfahrt wird die Stationenfolge umgedreht.

Um für den Benutzer aufwendige Verschiebungen möglichst zu vermeiden, wird eine gemeinsame Station von Teildienstwunsch und bisheriger Untertour gesucht. Gibt es eine solche Station, werden alle Stationen des Teildienstwunsches vor dieser Station am Anfang der Untertour angefügt, und der Halt mit der gleiche Station wird gemeinsam verwendet. Die restlichen Stationen – alle, wenn es keine gemeinsame Station gibt – werden an das Ende angehängt.

Bei der anschließenden Korrektur der Fahrzeiten kann es zu Fehlermeldungen des Verkehrsmoduls kommen, die weitergereicht werden.

Ob Rhythmen oder Wochentage passen, wird hier nicht geprüft, da das Aufgabe der Konsistenzprüfung ist.

teilDienstwunschEntfernen(TeilDienstwunsch) Entfernt den angegebenen TeilDienstwunsch aus der Untertour. War er der letzte Teil eines Dienstwunsches in der Untertour, so werden auch alle nicht mehr benötigten UntertourHalte entfernt.

wunschEntfernen(Dienstwunsch) Entfernt alle Teildienstwünsche, die zu dem angegebenen Dienstwunsch gehören, aus der Untertour. Die Fahrten bleiben erhalten, sollten aber gelöscht werden (die Benutzungsoberfläche erledigt dies automatisch: Nach Änderungen an einer Tour oder Untertour werden alle noch nicht erfolgten Fahrten (deren Datum in der Zukunft liegt) gelöscht).

laenge() : int Die Länge der Untertour in Metern. Beinhaltet auch Anfangs- und Endstation.

2.2.2.1 TeilDienstwunsch

Im Entwurf waren die Untertouren als einfache Datenklassen gedacht, denen die Stationen mit den Ankunfts- und Abfahrtszeiten einfach übergeben werden. Während der Implementierung kam dann der Wunsch auf, Dienstwünsche in die Untertour einzufügen. Da ein Dienstwunsch aber unterschiedliche Rhythmen haben kann, und unter Umständen auch Rückfahrten enthält, ist nicht klar, wie ein neuer Dienstwunsch die bestehende Stationenfolge verändern soll. Deshalb wurde die Klasse `TeilDienstwunsch` eingeführt. Ein `TeilDienstwunsch` ist ein Teil eines Dienstwunsches, der genau einen Rhythmus hat, an einem Wochentag stattfindet und entweder Hin- oder Rückfahrt ist.

- Attribute

  wunsch : Dienstwunsch der Dienstwunsch, zu dem dieser TeilDienstwunsch gehört

  wochentag : int der Wochentag

  rhythmus : Rhythmus ein Verweis auf den Rhythmus

  hinFahrt : boolean gibt an, ob der TeilDienstwunsch zu einer Hin-Fahrt gehört. Gibt es von einem Dienstwunsch keine Rückfahrten, ist dieses Feld true.

- Methoden

  beginn() : Zeitspanne liefert die Zeitspanne, innerhalb der der zugehörige Dienstwunsch anfangen soll.

  dauer(Station) : int liefert die Aufenthaltsdauer an der angegebenen Station.

2.2.2.2 UntertourHalt

Ein weiteres Problem war die Zuordnung der Teildienstwünsche zu den Stationen. Wenn eine Station doppelt in einer Untertour vorkommt, können die Teildienstwünsche nicht per Algorithmus auf die Stationen verteilt werden, da der Benutzer eventuell eine andere Verteilung möchte. Deshalb wurden die Einträge im Vector `stationen` um die dort behandelten Teildienstwünsche erweitert. Zudem wurde das Attribut umbenannt in `halte` und ist ein Vector mit Elementen vom Typ `UntertourHalt`:

- Erbt von

  StationMitZeiten

- Attribute

  wuensche : Vector enthält die Dienstwünsche, die an dem Halt behandelt werden. Damit sind auch die Teildienstwünsche festgelegt, da alle Teildienstwünsche eines Dienstwunsches in einer Untertour immer gleich behandelt werden.

2.2.2.3 StationMitZeiten

Eine Record-Klasse, von der UntertourHalt erbt.

- Attribute

  station : Station die Station, zu der Zeiten gespeichert werden sollen.

  ankunftsZeit : Uhrzeit Die Ankunftszeit an der Station.

  abfahrtsZeit : Uhrzeit Die Abfahrtszeit an der Station.

2.2.2.4 Fahrten

Während der Implementierung kam die Frage auf, was mit Dienstwünschen geschehen soll, die nicht sofort wirksam werden, oder an denen nur Änderungen vorgenommen wurden. Da die meisten Änderungen vermutlich zu diesen Gruppen gehören, mußte ein einfach aufsetzbares Konzept gesucht werden.

Die beste Möglichkeit wäre gewesen, den Untertouren und unter Umständen auch den Touren eine Datumsspanne zu geben, während der sie gültig sind. Dies hätte jedoch umfangreiche Veränderungen am gesamten System zur Folge gehabt, weshalb eine andere Lösung gesucht wurde.

Das System kennt von einer Untertour immer nur eine Version. Der Benutzer muß sich geplante Änderungen extra aufschreiben und zum gegebenen Zeitpunkt einfügen. Um feststellen zu können, welche Wünsche eines Kunden erfüllt wurden, werden alle Fahrten, bevor sie gelöscht werden, in eine Log-Datei geschrieben, die von TROSS nicht weiter verwendet wird, aber z.B. in eine Tabellenkalkulation geladen werden kann. In dieser Datei werden für jede stattgefundene Fahrt alle Halte mit den jeweils betroffenen Kunden gespeichert. Je nach Dienstart werden auch noch zusätzliche Informationen gespeichert, wie zum Beispiel bei einem Essens-Dienstwunsch Art und Anzahl der Essen. Durch dieses Auslagern wird auch der aktive Datenbestand von TROSS immer wieder reduziert.

Die Fahrt hat nun folgenden Aufbau:

- Attribute

  untertour : Untertour die Untertour, zu der die Fahrt gehört

  datum : Datum

  anfangsZeit : Uhrzeit

  endZeit : Uhrzeit

wuensche : Vector ein Vector mit den tatsächlich erfüllten Dienstwünschen.

fahrzeug : Fahrzeug

ersterMitarbeiter : Mitarbeiter

zweiterMitarbeiter : Mitarbeiter

In der Log-Datei wird für jeden Halt und jeden dort bedienten Dienstwunsch eine Zeile mit folgenden Inhalten angelegt:

- Das Datum (z.B. 24.12.1997)
- Die Nummer der Untertour (z.B. 2b)
- Die Kurzbezeichnung der Dienstart der Tour, und damit aller ihrer Dienstwünsche (z.B. Schule)
- Die Bezeichnung des Fahrzeugs (z.B. 123)
- Der erste Mitarbeiter (z.B. "Zivi, Zacharias; 12")
- Der zweite Mitarbeiter (z.B. "")
- Uhrzeit der Ankunft (z.B. 16:23)
- Die Adresse (z.B. "Die Strasse 12, 70123 Stuttgart")
- Der Kunde (z.B. "Sparwasser, Emma; 11")
- Bei einem Essensdienstwunsch die Anzahl der bestellten Essen je Essensart (z.B. 2*Diät)
- Die Bemerkung vom Tourplan

Die einzelnen Einträge sind durch Tabulatoren getrennt, einige Einträge werden durch Hochkommata vor dem Auftrennen geschützt.

2.2.3 Verkehrsmodul

An der Schnittstelle zum Verkehrsmodul gab es Änderungen, die zu folgendem Aufbau führten:

- Attribute

  anbindungsPunkte : Hashtable in dieser Hashtabelle werden die Anbindungspunkte gespeichert. Schlüssel sind die Map&Guide-Stationen, die Daten sind AnbindungsPunkte.

  entfernungen : EntfernungsTabelle

  MapAndGuidePfad : String der Pfad zu mg.exe.

  MapAndGuideAuftragsVerzeichnis : String Das Verzeichnis, in dem Map&Guide nach Auftrags-Dateien sucht, z.B. `C:\MG41\Jobs`.

  timeOut : int die Zeit in Sekunden, die auf eine Antwort von Map&Guide gewartet werden soll.

- Methoden

  speichern(ObjectOutputStream) speichert alle Daten des Verkehrsmoduls.

  laden(ObjectInputStream) lädt die Daten des Verkehrsmoduls aus dem angegebenen Stream.

  starteVerkehrstool() : MoeglicherAnbindungsPunkt[] startet Map&Guide und gibt ein Feld mit noch nicht korrekt angebundenen Stationen zurück.

  verwendeVerkehrstool : MoeglicherAnbindungsPunkt[] wie `starteVerkehrstool`, nur daß Map&Guide nicht neu gestartet wird.

  anbindungInOrdnung(MoeglicherAnbindungsPunkt) : boolean gibt true zurück, wenn die Anbindung in Ordnung (siehe 3.6) ist oder das Verkehrstool nicht läuft. Ersetzt `sucheMoeglicheAnbindungsPunkte`.

  erstelleAnbindungsPunkt(MoeglicherAnbindungsPunkt) : AnbindungsPunkt Erstellt einen AnbindungsPunkt. Ist der mögliche Anbindungspunkt nicht geprüft worden oder ist er nicht in Ordnung, wird ein Dummy erstellt.

  entferneAnbindungsPunkt(AnbindungsPunkt) macht den angegebenen Anbindungspunkt ungültig und entfernt ihn (wenn er nicht nochmal verwendet wird) aus den Tabellen.

Die Abfrage- und Vorbereitungs-Anfragen haben sich (von der Schnittstelle her) nicht verändert und wurden deshalb nicht wieder aufgeführt.

Um die im Cache vorhandenen Entfernungen berücksichtigen zu können, wurde der Algorithmus der `bereiteVor`-Methode, die alle Entfernungen zwischen den angegebenen Knoten ermittelt und in der Entfernungstabelle speichert, geändert.

Dabei ist `N` die Menge der Knoten, `|N|` also die Anzahl der Knoten und `N[0]` der erste Knoten. Jeder Knoten wird als Menge von Kanten gedacht, sodaß `|n|` die Anzahl der von Knoten `n ∈ N` ausgehenden Kanten angibt. Diese werden in einer Record-Schreibweise als `n.Kanten` bezeichnet, `n.Kanten[0]` ist also die (in irgendeiner Hinsicht) erste der vom Knoten `n` ausgehenden Kanten.

```
proc bereiteVor(N : Menge von Knoten)
{
  while (n = sucheStartKnoten()) {
    fahreNach(n);
    while (n = naechsterKnoten(n)) {
      fahreNach(n);
    }
  }
  teilAuftragStarten();
}

proc sucheStartKnoten() {
  if (∃ n ∈ N mit |n| ungerade) {
```

```
    return n;
  } else if (|N| == 0) {
    return null;
  } else {
    return N[0];
  }
}

func Knoten naechsterKnoten(Knoten n) {
  if (|n| == 0) return null;
  return n.Kanten[0];
}

proc fahreNach(Knoten n) {
  schreibt den Knoten in die Auftrags-Datei
  wenn schon 20 Knoten geschrieben wurden {
    teilAuftragStarten();
  }
}

proc teilAuftragStarten() {
  uebergibt die Datei an Map&Guide
  warte auf Map&Guide
  werte das Ergebnis aus
}
```

Da die momentane Implementierung dieses Algorithmus' auf naheliegende Optimierungsansätze verzichtet, ergibt sich bereits für die Suche nach dem nächsten Startknoten eine Laufzeit von $\mathcal{O}(|\text{gesuchte Kanten}|^2)$ (Ermittlung der Kantenanzahl per Schleife, erst danach Vergleich mit 0). Deswegen ist dieser Algorithmus mit einem Aufwand von $\mathcal{O}(n^4)$ auch langsamer als der alte, im Kapitel Feinentwurf des Zwischenberichts vorgestellte, mit einem Aufwand von $\mathcal{O}(n^2)$ (Durch kleine Optimierungen könnte dies auf $\mathcal{O}(n \cdot m) \in \mathcal{O}(n^3)$ verbessert werden, n=Anzahl Knoten, m=Anzahl Kanten). In der Praxis werden aber die gesparten Kanten durch den (nur im neuen Algorithmus berücksichtigten) Cache den Ausschlag geben, da die Kommunikation mit Map&Guide lange dauert (nach bisheriger Erfahrung midestens 5 Sekunden).

Für alle 11175 Entfernungen zwischen 150 Knoten, müssen etwa 559 Anfragen an Map&Guide gestellt werden. Die Zeit dafür wurde nicht gestoppt, dürfte aber in der Größenordnung von 2 Stunden liegen.

Allerdings wird diese Methode in der aktuellen Implementierung gar nicht verwendet (eine automatische Optimierung ist nicht Bestandteil des Programms). Das System in seiner momentanen Gestalt benötigt Entfernungen nur zwischen den aufeinanderfolgenden Stationen einer Untertour. Diese Kantenmenge läßt sich für normale Touren (mit nicht mehr als 20 Stationen) in etwa 15 Sekunden bearbeiten.

2.2.3.1 MoeglicherAnbindungsPunkt

Objekte dieser Klasse werden zum Erstellen von Anbindungspunkten für Stationen benötigt.

- Attribute

  station : Station ein Verweis auf die Station.

  strasse : String Der Name der Straße.

  hausNr : String Die Hausnummer.

  postLeitZahl : String Die Postleitzahl.

  ortsname : String Der Name des Ortes.

  MGStation : String Der String, der an Map&Guide übergeben werden soll. Kann null sein.

  korrekturListe : String[] Die von Map&Guide gelieferten „Alternativen“.

- Methoden

  MGStation() : String Gibt den String zurück, der an Map&Guide übergeben werden soll. Wurde MGStation nicht gesetzt wird das Ergebnis von `standardMGStation()` zurückgegeben.

  istInOrdnung() : boolean Gibt true zurück gdw. in der `korrekturListe` genau ein Eintrag enthalten ist.

  standardMGStation() : String Liefert "Orte,"+ postLeitZahl + ',' + ortsName + "„:"+ strasse + ' ' + hausNr.

2.2.3.2 AnbindungsPunkt

Zwischen Paaren von Anbindungspunkten speichert das Verkehrsmodul Entfernungen und Fahrzeiten.

- Attribute

  MGStation : String Der Ortsbezeichner für Map&Guide.

  anzahlVerwendungen : int gibt an, wie oft dieser Anbindungspunkt verwendet wird.

- Methoden

  istDummy() : boolean Gibt true zurück, wenn der Anbindungspunkt kein von Map&Guide geprüfter Anbindungspunkt ist (Verkehrstool wird nicht verwendet).

2.2.3.3 EntfernungsTabelle

In einer Instanz dieser Klasse werden die Entfernungen zwischen den Anbindungspunkten gespeichert.

- Attribute

tabelle : Hashtable In dieser Tabelle werden Paaren von Anbindungspunkten EntfernungsTabellenEinträge zugeordnet. Die Tabelle ist symmetrisch.

- Methoden

setzeEintrag (AnbindungsPunkt von, AnbindungsPunkt nach, int fahrdauer, int entfernung, boolean vonBenutzer) setzt den Eintrag in der Entfernungstabelle.

eintrag(AnbindungsPunkt, AnbindungsPunkt) : EntfernungsTabellenEintrag liefert den Eintrag zu den beiden Anbindungspunkten oder null.

entferneEintraegeMitAnbindungsPunkt(AnbindungsPunkt) Entfernt alle Einträge mit dem angegebenen AnbindungsPunkt aus der Tabelle.

2.2.3.4 EntfernungsTabellenEintrag

Ein Eintrag in der Hashtabelle `tabelle` einer EntfernungsTabelle.

- Attribute

fahrdauer : int die Fahrdauer in Minuten.

entfernung : int die Entfernung in Kilometern.

vonBenutzer : boolean true, wenn der Eintrag vom Benutzer gesetzt wurde und nicht vom Verkehrstool stammt.

Kapitel 3

Implementierung

3.1 Entwurfsentscheidungen auf Programmiersprachenebene

3.1.1 Allgemeine Konzepte

3.1.1.1 Sortieren und Sortierbarkeit

Da Java kein Konzept für sortierbare Objekte bietet, mußte ein solches von der Projektgruppe selbst entworfen und implementiert werden. Um Klassen wie selbstsortierende Listen zu realisieren, wurden zwei Interfaces definiert, die von allen Klassen implementiert werden müssen, deren Instanzen in einer solchen Liste gespeichert bzw. angezeigt werden sollen:

Comparable Vergleichbare Objekte implementieren die Methode `compareTo` nach dem Vorbild des JDK. Leider können die fertigen Java-Klassen dieses Interface nicht mehr implementieren, selbst wenn sie eine entsprechende Methode besitzen.

ListenElement Elemente, die in einer selbstsortierenden Liste von der graphischen Benutzungsoberfläche dargestellt werden sollen, implementieren zusätzlich zum Interface `Comparable` die Methode `listenText`, die eine textuelle Darstellung des Objekts für die Liste liefert.

Im Laufe der Implementierung stellte sich heraus, daß es die Performance von Java nicht zuläßt, umfangreichere Listen vor jedem Anzeigen neu zu sortieren. Deswegen wurden umfangreiche dynamischen Arrays der Klasse Szenario von der Klasse `java.util.Vector` auf eine eigene Klasse `TRO.SortierterVektor` umgestellt. Mit den oben beschrieben Interfaces konnte eine automatische Sortierung (durch binäres Einfügen) leicht erfolgen, und durch die Anpassung der Schnittstelle des selbstsortierenden Arrays an die von `java.util.Vector` waren in den Datenklassen praktisch keine zusätzlichen Veränderungen erforderlich.

3.1.1.2 Datum und Zeit

Die Verwaltung von Kalenderdaten und Uhrzeiten wurde zwischen den Java-Versionen JDK 1.0 und JDK 1.1 erheblich verändert. In der aktuellen Klassenbibliothek existiert eine Klasse, die beide Funktionalitäten vereinen soll. Da

die Konzepte dieser Datums- und Zeitverwaltung der Projektgruppe zum einen nicht gut durchdacht und daher schlecht handhabbar erschienen, zum anderen die vorliegende Version des JDK noch offensichtliche Fehler enthielt, die sich beim Umgang mit Daten und Zeiten bemerkbar machten, wurden eigene Klassen für Uhrzeit und Datum implementiert, die im wesentlichen unabhängig vom JDK arbeiten und nur für einige komplexe Berechnungsfunktionen auf dessen Möglichkeiten zurückgreifen. Ebenso lassen die von Java zur Verfügung gestellten Datums- und Zeitklassen eine Funktion zum Überprüfen, ob ein Datum bzw. eine Uhrzeit sinnvoll eingegeben wurde, vermissen. Da die Implementierung einer solchen Prüffunktion sich als sehr komplex herausgestellt hat, und nur mit erheblichen Aufwand realisiert werden könnte, wurde im Hinblick auf die knappe Zeit darauf verzichtet. Auf falsche Eingaben eines Datums oder einer Uhrzeit wird nun mit der von Java vorgegebenen Methodik reagiert, nach der die Eingabe in eine sinnvolle überführt wird. So wird z.B. auf die Eingabe „35.13.1998“ nicht mit einer Fehlermeldung reagiert, sondern diese *stillschweigend* von Java in das nächste sinnvolle, in diesem Falle den „4.1.1999“ überführt.

3.1.2 GUI-Erweiterungen

Die graphische Benutzungsoberfläche wurde stark modular konzipiert, um den Entwicklungsaufwand möglichst gering zu halten. Alle mehrfach benötigten Oberflächenelemente wurden als eigene Klassen realisiert, die dann zur allgemeinen Verwendung für die restliche Benutzungsoberfläche zur Verfügung stehen. Hier sind insbesondere zu nennen:

- einfache und zusammengesetzte Eingabefelder für häufig benötigte Datentypen. Diese konvertieren selbständig zwischen den Datenobjekten und den in Javas Textfeldern ein- und ausgegebenen Strings und können dabei gleich Typ- und Wertebereichsprüfungen vornehmen.

  Analog zu den Datenklassen setzen sich die zugehörigen Eingabefelder aus anderen Eingabefeldern niedrigerer Komplexität zusammen:

  - ganze Zahlen
  - Datum, Uhrzeit
  - Datumsspanne, Zeitspanne
  - Stationen
  - Personendaten (die wiederum nur einen Teil der Daten eines Kunden oder Mitarbeiters ausmachen)
  - etc.

- Dialogrümpfe für die wichtigsten Grundfunktionen:

  - Zuweisen von Tastendrücken an Buttons und andere Bedienelemente (`TastaturDialog`)
  - Eingabedialoge, die mit „OK“ bestätigt oder mit „Abbruch“ abgebrochen werden können (`OKCancelDialog`)
  - Umfangreiche Eingabemasken, die nach dem Vorbild von Registerkarten zwischen verschiedenen, thematisch zusammengefaßten Eingaberegionen umschalten können (`RegisterDialog`)

3.1.2.1 Typisierte Eingabefelder

Da Javas Klassenbibliothek keinerlei Unterstützung für typisierte Eingabefelder bietet, wurden für das Projekt TROSS zusätzliche Eingabefelder für die Datentypen Integer und Datum geschaffen, die die Konvertierung in Strings selbständig vornehmen. Aus diesen elementaren Eingabefeldern wurden komplexere zusammengestellt (um z.B. Zeiträume einzugeben), die als fertige Module wie einfache Textfelder in die Oberfläche integriert werden können.

Das Auslesen und Setzen von Eingabewerten zusammengesetzter Dialoge soll atomar erfolgen: Kann ein Feld wegen fehlerhafter Eingabe nicht ausgewertet werden, dürfen auch die Werte der anderen Felder noch nicht propagiert werden, sondern es muß ein Rücksprung zur Eingabe erfolgen. Auch für solche kombinierten Konsistenzprüfungen stellt Java keine Unterstützung bereitstellt. Deshalb wurde in die neuen Eingabefeld-Klassen ein Mechanismus integriert, der ohne großen Aufwand für den Aufrufer nicht nur die Atomarität des Auslesevorgangs sicherstellt, sondern bei Fehlern im Eingabeformat auch eine präzise Rückmeldung an den Benutzer erlaubt.

Hierbei wird in der Art des in der Transaktionsverarbeitung eingesetzten Zwei-Phasen-Commit-Protokolls vorgegangen [BN97]:

1. Ein komplexes Eingabeobjekt (Dialog oder zusammengesetztes Eingabefeld) stellt zunächst den „Ready-to-Commit"-Status sicher, indem es alle untergeordneten Eingabefelder zum Auslesen der Eingabewerte auffordert. Jedem Teilfeld wird hierbei eine Bezeichnung mitgegeben, mit der im Falle eines Auslese-Fehlers dem Benutzer genau mitgeteilt werden kann, welche seiner Eingaben unkorrekt war bzw. ein falsches Format hatte.

2. Konnten alle Eingaben in die passenden Datentypen umgesetzt werden, werden in der zweiten Phase tatsächlich die Werte in den Datenstrukturen gesetzt.

Dieses synchronisierte Setzen von Werten gilt allerdings nur für direkt eingegebene Werte. Änderungen in Listen (wie in 3.1.2.2 und 3.1.2.5 beschrieben) werden direkt nach deren Bestätigung in den Datenobjekten durchgeführt, da die obige Vorgehensweise hier noch deutlich komplizierter umzusetzen wäre.

3.1.2.2 Auswahl aus Listen

Eine häufige Aktion bei der Arbeit mit der graphischen Benutzungsoberfläche stellt die Auswahl eines oder mehrerer Elemente aus einer Menge dar.

- Zur Auswahl genau eines Elements dienen herunterklappende Auswahllisten, die jeweils nur das gewählte Element zeigen (Java nennt diese „Choice", unter MS Windows sind sie als „Combobox" bekannt).

  Da nicht in jedem Fall eine Auswahl aus der angebotenen Liste obligatorisch ist (z.B. ist eine Tour auch ohne Angabe der Soll-Mitarbeiter gültig), bietet die Auswahlliste gegebenenfalls ein Dummy-Element „Keine Auswahl" an erster Stelle an, nach dessen Auswahl die zugehörige Variable auf `null` gesetzt wird.

- Die Auswahl mehrerer Elemente, also einer Teilmenge, geschieht je nach Größe auf zwei unterschiedliche Arten:

- Kleinere Listen (z.B. mögliche Qualifikationen der Mitarbeiter), bei denen auch der Überblick über die nicht gewählten Elemente interessant ist, werden komplett dargestellt (bzw. ein mehrelementiger Ausschnitt mit Scrollbalken). Die Auswahl und Abwahl von Elementen erfolgt durch Mausklick, worauf das gewählte Element durch inverse Darstellung gekennzeichnet wird.
- Von langen Listen werden nur die momentan ausgewählten Elemente in einer eigenen Liste präsentiert. Über einen Button kann ein eigenes Dialogfenster geöffnet werden, das die Veränderung der Teilmenge durch Auswahl aus der Gesamtmenge erlaubt.

3.1.2.3 Eingabedialoge

Der typische Eingabedialog besteht aus einer Menge von Eingabefeldern, Auswahllisten und anderen Oberflächenelementen zur Datenerfassung, sowie den zwei Buttons „OK“ zur Bestätigung der Eingabe und Übernahme der Werte und „Abbruch“ zur Beendigung des Dialogs, ohne die eingegebenen Werte zu übernehmen. Die Klasse `TRO.GUI.OKCancelDialog` bietet diese Grundfunktionalität an. Sämtliche Eingabedialoge im Programm TROSS erben von dieser Klasse und besitzen dadurch ohne zusätzliche Programmierung diese zwei grundlegenden Buttons, die auch über die Tasten `Enter` bzw. `Escape` ausgelöst werden können.

Die Aktivierung eines dieser Buttons ruft die passende Methode der Dialogklasse auf, die sich gegebenenfalls um das Auslesen der Werte kümmert (unter Benutzung des in 3.1.2.1 beschriebenen zweistufigen Mechanismus') und dann den Dialog schließt.

3.1.2.4 Unterteilte Dialoge (Register)

Da einige Datenobjekte eine Vielzahl an Eingaben erfordern, wurde im Verlauf der Implementierung bald der Punkt erreicht, an dem die entstehenden Eingabemasken nicht mehr vollständig auf den Bildschirm paßten. Auch aus Gründen der Übersichtlichkeit war hier eine Unterteilung notwendig, die nach dem Prinzip der sogenannten „Registerkarten“ erfolgte:

Eine Reihe von Buttons am oberen Rand des Fensters ermöglicht das Umschalten zwischen verschiedenen „Seiten“ mit thematisch zusammengehörigen Eingabefeldern. Die Gesamtheit all dieser Seiten ergibt den Eingabedialog, die Eingabefelder aller Seiten erfassen das zugehörige Datenobjekt in seiner Gesamtheit.

Diese Grundfunktionalität wurde in der Klasse `TRO.GUI.RegisterDialog` verwirklicht, so daß alle komplexen Dialoge lediglich davon abgeleitet und in passende Seiten aufgeteilt werden mußten.

3.1.2.5 Erweiterbare Listen

Bereits bei der Erstellung des Prototyps der graphischen Benutzungsoberfläche hatte sich gezeigt, daß das Programm TROSS eine Vielzahl von Listen verwaltet, deren Elemente vom Benutzer beliebig geändert, erweitert und gelöscht werden sollen. Um dies sowohl möglichst einfach als auch universell zu unterstützen, wurde die Klasse `TRO.GUI.ObjektListePanel` geschaffen. Diese

präsentiert sich auf dem Bildschirm als eine geordnete Liste von Objekten mit den drei Buttons „Einfügen“, „Ändern“ und „Löschen“. Die jeweils individuelle Funktionalität der Buttons wird vom aufrufenden Programmteil zur Verfügung gestellt, indem sogenannte „Listener“-Objekte definiert werden, wie sie in Javas AWT an vielen Stellen zum Einsatz kommen, um auf gewisse Ereignisse der Benutzungsoberfläche kontextspezifisch reagieren zu können.

3.1.2.6 Eingabe von Hashtabellen

Hashtabellen werden an verschiedenen Stellen im Projekt TROSS eingesetzt, unter anderem zur Speicherung der gewünschten Mengen verschiedener Essensarten. Da es unter den Standardeingabeelementen, die unter Java zur Verfügung stehen, keine Tabellen gibt, sondern nur eindimensionale Listen, mußten die Hashtabellen (bzw. Tabellen aller Art) zur Eingabe in ihre einzelnen Dimensionen zerlegt werden: Eine Liste zeigt alle Essensarten an, bei Auswahl einer Essenart wird die zugehörige Anzahl angezeigt und kann geändert werden. Unerläßlich für die sinnvolle Bedienung war es hier, auf Anfrage eine Übersichtsliste anzuzeigen, die wenigstens in der Ausgabe alle in der Hashtabelle zugeordneten Objekte tabellenartig anzeigt.

3.2 Designentscheidungen

3.2.1 Fensteraufbau

Da der Aufbau eines Fensters unter Java relativ viel Zeit in Anspruch nimmt (weil jedesmal die Anordnung aller Elemente berechnet werden muß), wurde die Möglichkeit diskutiert, einmal geöffnete Fenster nur auf dem Bildschirm unsichtbar zu machen und für weitere Verwendungen zu speichern. Dies hätte zwar einen Vorteil in der Laufzeit gebracht, wurde aber aus zwei Gründen verworfen:

1. Das Umschreiben aller Dialoge wäre mit erheblichem Aufwand verbunden, den die zeitliche Situation der Projektgruppe nicht erlaubt.

2. Die meisten Dialoge bekommen die zu bearbeitenden Objekte im Konstruktor übergeben, da diese immer gesetzt werden müssen. Soll ein einmal erzeugter Dialog nochmals benutzt werden, müßten diese Objekte mit eigenen Methoden gesetzt werden. Das Dialogfenster müßte bei jeder erneuten Benutzung Prüfungen durchführen, ob diese Werte alle gesetzt wurden und gegebenenfalls Exceptions auslösen, die im aufrufenden Objekt abgefangen werden müßten.

   Letztlich müßten also auch alle Aufrufer geändert werden. Die Projektgruppe entschied sich dafür, die übersichtliche Handhabung der Dialogfenster im Programm nicht dafür zu opfern, Probleme der Laufzeitumgebung zu kompensieren.

Außerdem kann das Zeitproblem durch den Einsatz eines Compilers, der Java in reinen Maschinencode für das Zielsystem umsetzt, vermutlich stark relativiert werden.

3.2.2 Destruktoren

In Java arbeitet die Speicherverwaltung automatisch und für den Programmierer weitgehend unsichtbar. Neue Objekte werden mit `new` angefordert, um eine Freigabe braucht sich das Programm nicht zu kümmern. Diese wird vom Laufzeitsystem erledigt, das in gewissen Abständen seinen garbage collector aktiviert, der alle Objekte, die im Programm nicht mehr referenziert werden, aufsammelt und den zugehörigen Speicher freigibt.

Klassen können die Methode `finalize` implementieren, um externe Ressourcen vor dem Löschen des Objekts durch den Garbage collector freizugeben. Im Projekt TROSS ist es nötig, beim Löschen einer Station den zugeordneten Anbindungspunkt beim Verkehrsmodul abzumelden, damit dieses seine internen Tabellen so klein wie möglich halten kann.

Ein selbstverwaltetes Abmelden aller gelöschten Stationen im Programm wäre mit immensem Aufwand verbunden und sehr fehleranfällig. Außerdem widerspräche ein solches Vorgehen dem grundsätzlichen Speicherkonzept der Sprache Java, das oben erläutert wurde.

Leider läßt Javas Dokumentation gewisse Zweifel an der Vollständigkeit und Korrektheit der Speicherverwaltung offen: Der Aufruf des Garbage collectors ist mit folgendem Kommentar versehen:

> Runs the garbage collector.
> Calling this method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made its best effort to recycle all unused objects.

Die Projektgruppe entschied sich dennoch, mit den finalize-Methoden zu arbeiten. Mögliche Fehlfunktion des Java-Laufzeitsystems sollten nicht zu unnötigem Programmieraufwand der Anwendung führen. Für die korrekte Arbeitsweise des Programms TROSS gilt also die Annahme, daß Javas Garbage collector alle nicht mehr referenzierten Objekte erkennt und deren finalize-Methode aufruft.

3.3 Umsetzung des Entwurfs in der Implementierung

3.3.1 Personenklassen

In den folgenden Abschnitten werden die Implementierung und Abweichungen vom Feinentwurf der Personenklassen beschrieben. Alle Klassen besitzen zusätzlich die standardmäßigen setze- und lese-Methoden[1]. Ebenso wurde, wenn nicht anders angegeben, bei allen Klassen ein leerer Konstruktor (der alle referenzierten Objekte mit new initialisiert) und ein vollständiger Konstruktor, der alle Werte mit den übergebenen initialisiert, implementiert.

Zu den Schlüsselattributen ist anzumerken, daß deren Schlüsseleigenschaft von der Benutzungsoberfläche unter Verwendung des Packages Konsistenztests,

[1] Parameter/Rückgabewerte gleichen Typs wie die Attribute, lese-Methoden heißen genauso wie das Attribut, die setze-Methoden heißen „setze“+Attribut-Name

und nicht von den Klassen selbst, sichergestellt wird. Das gleiche gilt für die Mußwerte. Auch hier ist die Angabe beim jeweiligen Attribut nur eine Information für die entsprechende Umsetzung in der Benutzungsoberfläche.

3.3.1.1 Bank

Diese Klasse wurde gegenüber dem Feinentwurf nur marginal verändert.

- Attribute

  bankLeitZahl : int Die Bankleitzahl wurde wegen der einfacheren Handhabung als Integer anstatt als String implementiert. Außerdem wurde die Begrenzung auf acht Stellen aufgehoben.

  bankName: String Name der Bank mit obiger Bankleitzahl.

3.3.1.2 Bankverbindung

- Attribute

  bank : Bank Dient zur Referenzierung des zugehörigen Bankobjektes der Bank, bei dem die Bankverbindung besteht. Damit ersetzt „bank“ das im Feinentwurf vorgesehene Attribut „BLZ“.

  kontoNr : String Die Kontonummer, ohne die im Feinentwurf gemachte Einschränkung auf zehn Ziffern.

  einzug : boolean Ist true, wenn eine Einzugsermächtigung vorliegt. Voreinstellung: true.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.1.3 Bezugsperson

Unterklasse von Person.

- Attribute

  bezugsArt : String Gibt die Art des Bezuges zum Kunden an. Z.B. Familienverhältnis, Arzt, etc.

  bemerkung : String Dieses Attribut wird jetzt aus der Oberklasse Person geerbt.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.1.4 Institution

- Attribute

  name : String Name des Institus, z.B. Krankenkasse X, Sozialamt Y, Versicherung Z.

  adressen : Vector In diesem Attribut, das gegenüber dem Feinentwurf ergänzt wurde, werden alle Adressen einer Institution vermerkt.

bankVerbindungen : Vector In diesem Attribut, das gegenüber dem Feinentwurf ergänzt wurde, werden alle Bankverbindungen einer Institution vermerkt.

kommunikationsVerbindungen : Vector In diesem Attribut, das gegenüber dem Feinentwurf ergänzt wurde, werden alle Kommunikationsverbindungen einer Institution vermerkt.

Zusätzliche wurden die Interfaces Listenelement und Zahler implementiert. Durch das Zahlerinterface kann eine Institution als Rechnungsempfänger bei den Dienstwünschen der Kunden eingetragen werden.

3.3.1.5 Kommunikationsverbindung

Abgesehen von der Implementierung des Listenelements wurde gegenüber dem Feinentwurf keine Änderung vorgenommen.

- Attribute

  art : String (Mußwert) Gibt die Art der Kommunikationsverbindung an (Telefon, Fax, Email, etc.).

  eintrag : String (Mußwert) Zur Art passender Eintrag, z.B. Telefonnummer, Email-Adresse, etc.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.1.6 Kunde

Unterklasse von Person.

- Attribute

  kundenNr : String (Schlüssel) Eindeutige Nummer, welche den Kunden innerhalb des DRK identifiziert.

  hausschluesselZahl : int Anzahl der Haus- bzw. Wohnungsschlüssel, die der Kunde dem DRK überlassen hat. Dieses Attribut wurde wegen der einfacheren Handhabung als Integer (statt als String) implementiert.

  hausSchluesselText : String Beliebige Bemerkungen zu den Schlüsseln, beispielsweise vergebene Schlüsselnummern, für welche Tür, etc.

  maxFahrDauer : int Gibt an, wieviele Minuten ein Kunde maximal in einem Fahrzeug zubringen darf. Die globalen Obergrenzen (z.B. Schulfahrten max. 2h) bleiben davon unberührt. Im Feinentwurf wurde noch der Typ short vorgesehen.

  bemerkung : String wird jetzt aus der Oberklasse Person geerbt.

  bezugsPersonen : Vector Die zum Kunden gehörenden Bezugspersonen wie Hausarzt und Familienangehörige. Institutionen werden jetzt in einem eigenen Vektor berücksichtigt. Kundenbetreuer der Institutionen können nach wie vor als Bezugspersonen hier hinterlegt werden.

institutionen : Vector Hier werden die Institutionen abgelegt, die einen Bezug zum Kunden haben.

bevorzugt : Vector Die vom Kunden bevorzugten Mitarbeiter, diese sollen i.d.R. dessen Dienstwünsche erfüllen.

abgelehnt : Vector Die vom Kunden abgelehnten Mitarbeiter, diese dürfen keine Dienste beim Kunden verrichten.

hilfsmittel : Vector Die vom Kunden benötigten Hilfsmittel wie Sitzkissen u.ä., die über die Attribute des Dienstwunsches hinausgehen. Diese Werte sind nur informativ und werden bei der automatischen Planung nicht berücksichtigt.

zulaessigeFahrzeuge : Vector Fahrzeuge, die für den Kunden vom DRK freigegeben wurden. Ist dieser Vektor leer, dürfen alle Fahrzeuge bei dem Kunden eingesetzt werden.

dienstwuensche : Vector Die Dienstwünsche des Kunden.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.1.7 Mitarbeiter

Unterklasse von Person.

- Attribute

personalNr : String (Schlüssel) Personalnummer des Mitarbeiters beim DRK, die diesen eindeutig identifiziert.

dienstantrittsDatum : Datum Datum, zu dem der Mitarbeiter seinen Dienst beim DRK antritt. Der im Feinentwurf vorgesehene Typ Date wurde durch die eigene Klasse Datum ersetzt.

entlassungsDatum : Datum Datum, zu dem der Mitarbeiter ausscheidet. Bei festangestellten Mitarbeitern ist dieser Eintrag in der Regel leer. Der im Feinentwurf vorgesehene Typ Date wurde durch die eigene Klasse Datum ersetzt.

verfügbarkeitsZeiten : Zeitraum Daten, an denen der Mitarbeiter eingesetzt werden kann.

bemerkung : String Wird jetzt aus der Oberklasse Person geerbt.

erfuellteQualifikationen : Vector Die vom Mitarbeiter erfüllten Qualifikationen.

arbeitszeit : ArbeitszeitProfil Dieses Attribut ist zum Feinentwurf hinzugekommen und nimmt ein Arbeitszeitprofil (Sollarbeitszeit) für den Mitarbeiter auf.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.1.8 Ort

- Attribute

PLZ : String Postleitzahl, ohne Einschränkung auf fünf Stellen. Damit sind auch Postleitzahlen im Ausland (z.B. für private Versicherungen) möglich.

name : String Name des Ortes, der zu obiger Postleitzahl gehört.

- Methoden

boolean gleich(Ort) Gibt true zurück, wenn dieser Ort mit dem Übergebenen in Postleitzahl und Namen übereinstimmt.

3.3.1.9 Person

- Attribute

name : String Nachname der Person.

vorName : String Vorname der Person.

geburtsDatum : Datum Geburtsdatum der Person. Jetzt als Datum statt als Date.

adressen : Vector Die zur Person gehörenden Stationen.

kommunikationsVerbindungen : Vector Die Kommunikationsverbindungen der Person.

bankVerbindungen : Vector Die Bankverbindungen der Person

bemerkung : String Hier kann ein beliebiger Text als Anmerkung eingegeben werden, der vom System nicht weiter verwendet wird.

Zusätzlich wurde das Zahlerinterface implementiert, das es jeder Person erlaubt, Rechnungsempfänger für Dienstwünsche zu sein.

3.3.1.10 Sachbearbeiter

Unterklasse von Person, die gegenüber dem Feinentwurf nicht geändert wurde.

- Attribute

angestelltBei : Institution Arbeitgeber des Sachbearbeiters, beispielsweise Sozialamt oder Krankenkasse.

3.3.1.11 Station

- Attribute

strasse : String Straßen- oder Platzname bzw. Postfach.

hausNr : String In diesem Attribut, das im Feinentwurf noch nicht vorgesehen war, wird die Hausnummer der Station gespeichert. Die Trennung vom Straßennamen wurde wegen der einfacheren Anbindung an das Verkehrstool vorgenommen.

ort : Ort Als Referenz zum Ort der Adresse. Dies ersetzt die im Feinentwurf vorgesehene Postleitzahl.

anbindungspunkt : AnbindungsPunkt Bezeichnung, unter der die Station im Verkehrsmodul vermerkt ist (i.d.R. die Adresse oder ein Punkt, der möglichst nahe an der Station liegt).

name: String (SCHLÜSSEL) Name einer Station als Eingabehilfe für den Anwender (Konzept der benannten Station siehe 2.2.1).

- Methoden

  void entferneAnbindungsPunkt() Veranlaßt die Entfernung des Anbindungspunkts aus dem Verkehrsmodul.

  boolean adressenGleich(Station) Überprüft, ob diese Station mit der Übergebenen in der Adresse übereinstimmt. Dies ist der Fall, wenn sowohl der Straßenname (String.equals) als auch der Ort (Ort.gleich) übereinstimmen.

  void finalize() Sorgt dafür, daß eine unreferenzierte Station durch den Garbage-Collector aus dem Verkehrsmodul abgemeldet wird, bevor sie aus dem Speicher entfernt wird. Um garantiert alle unreferenzierten Stationen aus dem Verkehrsmodul abzumelden, muß der Garbage-Collector vor dem Speichern explizit aufgerufen werden.

Zusätzliche wurde das Interface Listenelement implementiert.

3.3.1.12 Qualifikation

- Attribute

  bezeichnung : String (Schlüssel) Eindeutiger (Kurz-)Bezeichner für die Qualifikation.

  beschreibung : String Nähere Beschreibung der Qualifikation.

  verrechnungsWert : int Kostenfaktor, der bei der Optimierung berücksichtigt wird, falls diese Qualifikation erfüllt, jedoch nicht gefordert wird. Dieser Wert wird automatisch berechnet, falls autoWert true ist und sich Mitarbeiterdaten (bzgl. der Qualifikation) ändern. Im Feinentwurf war noch der Typ byte vorgesehen.

  autoWert : boolean Ist autoWert false, findet keine automatische Berechnung des Verrechnungswertes mehr statt. Wird der Verrechnungswert manuell geändert, wird autoWert auf false gesetzt.

Zusätzlich wurde das Interface Listenelement implementiert.

3.3.2 Datenhaltung

Die Daten werden mit der in Java vorgesehenen Möglichkeit der Objekt-Serialisation in eine Datei geschrieben. Dazu werden die Objekte von Szenarien verwaltet. Wie in Abbildung 3.1 zu sehen, lassen sich folgende zwei Szenarioarten unterscheiden:

1. **Tourszenarien:** Mit den Tourszenarien können, bei gleichbleibenden Stammdaten, verschiedene Tourkonstellationen ausprobiert werden. Da diese logisch zu den gleichen Stammdaten gehören, werden alle Tourszenarien samt den Stammdaten in einer Datei gespeichert.

2. **Allgemeine Szenarien:** Hierbei können auch Stammdaten geändert werden, um beispielsweise zu testen, wie sich eine Verringerung der Fahrzeugflotte auswirken würde. Diese allgemeinen Szenarien werden jeweils in einer eigenen Datei gespeichert und im folgenden kurz als **Szenarien** bezeichnet.

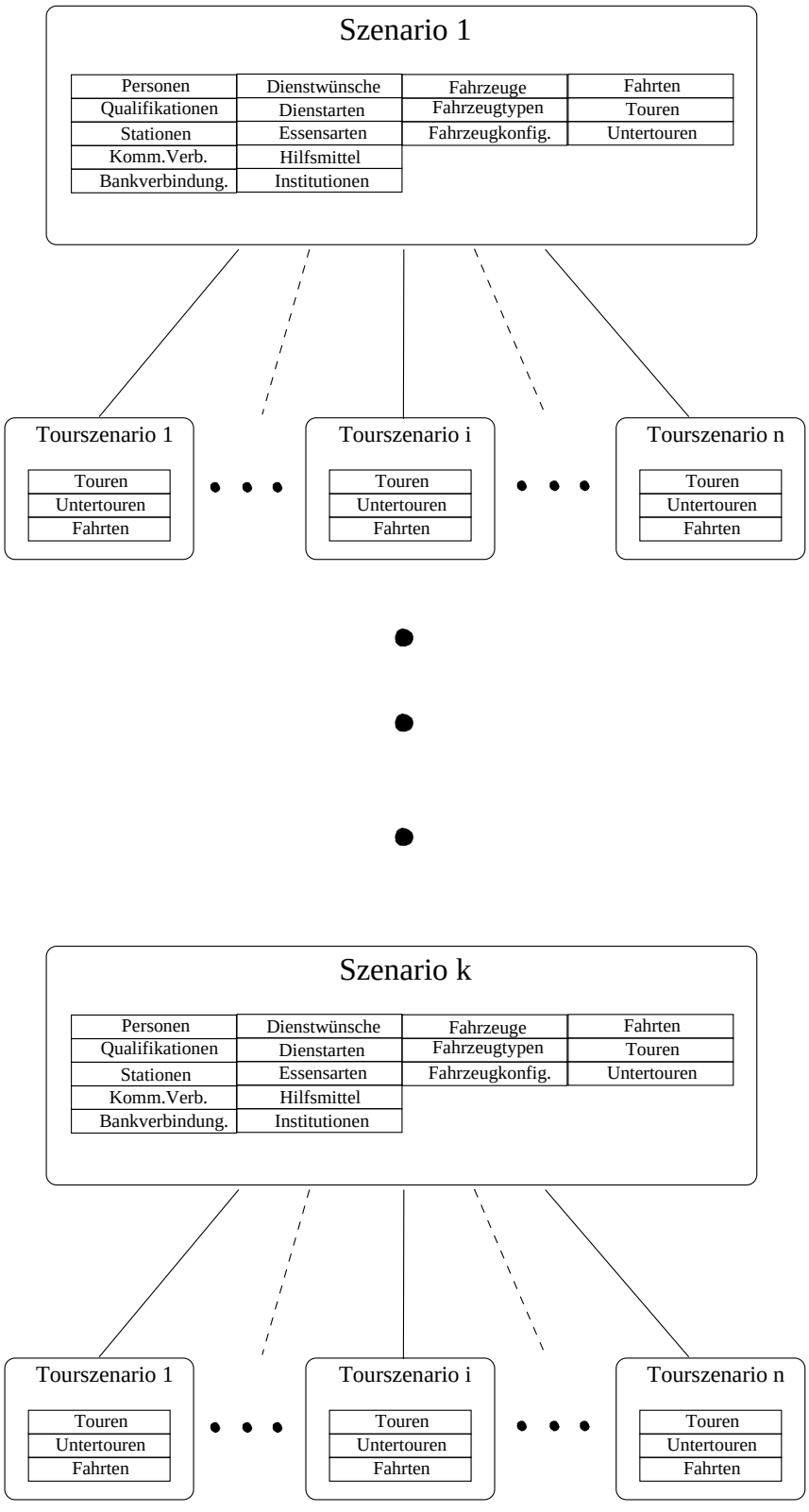


Abbildung 3.1: Datenhaltung in TROSS

Neben dem Dateimanagement bieten die Szenarien auch objektübergreifende Methoden (z.B. Verwaltung benannter Stationen) an.

Im Gegensatz zum Feinentwurf werden nicht alle Objekte direkt von den Szenarien referenziert, sondern nur solche, die nicht von anderen Objekten referenziert werden oder die in globalen Listen vorgehalten werden (z.B. benannte Stationen, Mitarbeiter, Qualifikationen etc.).

Im folgenden nun die Beschreibung zur Implementierung der Verwaltungsklassen.

3.3.2.1 Szenario

- Attribute

  name : String (Schlüssel) Name des Szenarios.

  geaendert : Boolean Ist true, wenn der Datenbestand geändert wurde.

  mitarbeiter : SortierterVektor Referenz auf alle Mitarbeiter des Szenarios.

  kunden : SortierterVector Referenz auf alle Kunden des Szenarios.

  institutionen : SortierterVektor Referenz auf alle Institutionen des Szenarios.

  qualifikationen : SortierterVektor Referenz auf alle Qualifikationen des Szenarios.

  fahrzeuge : SortierterVektor Referenz auf alle Fahrzeuge des Szenarios.

  fahrzeugtypen : SortierterVektor Referenz auf alle Fahrzeugtypen des Szenarios.

  stationen : SortierterVektor Referenz auf alle benannten Stationen des Szenarios.

  essensarten : SortierterVektor Referenz auf alle Essensarten des Szenarios.

  hilfsmittel : SortierterVektor Referenz auf alle Hilfsmittel des Szenarios.

  arbeitszeitProfile : SortierterVektor Referenz auf alle Arbeitszeitprofile des Szenarios.

  tourSzenarien : SortierterVektor Referenz auf alle Tourszenarien des Szenarios.

  aktivesTourszenario : TourSzenario Referenz auf das aktuell aktivierte Tourszenario.

- Methoden

  void wurdeGeaendert() Setzt das Geändert-Flag auf wahr und muß immer aufgerufen werden, wenn sich irgendein Objekt ändert. Diese Methode ersetzt die setzeGeaendert-Methode, da das Geändert-Flag von außen nicht auf false gesetzt werden darf.

  Szenario laden(String pfad, String name) Lädt das angegebene Szenario in den Speicher. Die Methode soll nur aufgerufen werden, wenn sich kein verändertes Szenario im Speicher befindet.

void speichern(String pfad, String name) Speichert das im Hauptspeicher befindliche Szenario unter dem angegebenen Namen und setzt das Veränderungsflag auf false. Stimmt der übergebene Name nicht mit dem des Szenarios überein (speichern als), wird der übergebene Name als neuer Szenarioname übernommen.

boolean istStationBekannt(String name) Überprüft, ob der übergebene Name schon für eine benannte Station verwendet wurde und gibt einen entsprechenden Wahrheitswert zurück.

Station sucheStation(String name) Gibt die Station zum übergebenen Namen zurück.

Die Methoden zum Löschen und Kopieren wurden aus Zeitgründen nicht implementiert, da diese durch entsprechende Dateioperationen auf Betriebssystemebene erreicht werden können. Anstelle der neuen Klasse SortierterVektor (siehe 3.1.1.1) war im Feinentwurf noch die Klasse Vector vorgesehen. Diese Klasse erlaubt die Objekte sortiert abzulegen und damit eine wesentlich effizientere Implementierung der häufig benötigten sortierten Ausgaben.

3.3.2.2 Tourszenario

- Attribute

  name : String (Schlüssel) Name des Tourszenarios.

  touren : Vector Touren des Tourszenarios, die wiederum auf die fahrtenreferenzierenden Untertouren verweisen.

- Methoden

  Fahrt[] objektFahrten(Object, Zeitraum) Gibt alle Fahrten zurück, die das übergebene Objekt innerhalb des übergebenen Zeitraums zu erledigen hat. Dazu werden alle Touren, deren Untertouren und Fahrten durchlaufen. Damit liegt der Aufwand in $\mathcal{O}$(Anz. d. Fahrten).

Zusätzlich wurde das ListenElement-Interface implementiert.

Welches Tourszenario aktiv (Mastertourszenario) ist, wird abweichend vom Feinentwurf jetzt durch das zugehörige Szenario verwaltet.

3.3.3 Konsistenztests

Die Konsistenztests wurden entsprechend dem Entwurf als automatische Prüfungen implementiert: Nach der Eingabe von Daten wird die Einhaltung von Integritätsbedingungen und Vorgaben durch das neu Eingegebene geprüft. Verstoßen die Daten gegen Integritätsbedingungen, muß die Eingabe wiederholt werden. Vorgaben können hingegen explizit außer Kraft gesetzt werden. Um den Überblick über solche explizit akzeptierten Verstöße gegen Vorgaben zu behalten, kann dieser Teil der Konsistenzprüfung auch von Hand aufgerufen werden. Daraufhin bekommt der Benutzer alle Verstöße gegen die Vorgaben gemeldet. Ein gezieltes Fragen nach einzelnen Vorgaben wurde bis auf wenige Ausnahmen nicht implementiert.

3.3.3.1 Referenzintegrität

Die Referenzintegrität wird im Programm TROSS gewahrt, indem das Löschen von Fremdschlüsseln verhindert wird. So darf z.B. eine Qualifikation nicht gelöscht werden, wenn sie einem Mitarbeiter zugeordnet ist oder von einem Kunden gefordert wird.

Da keine Datenwerte als Fremdschlüssel dienen, sondern direkte Referenzen auf die Java-Objekte gespeichert werden, findet eine Veränderung von Fremdschlüsseln durch die Eingabe nicht statt, hier muß also nichts geprüft werden.

3.3.3.2 Vorgaben

„Weiche“ Konsistenzbedingungen, die Bedingungen der realen Welt darstellen, sind für die Touren, Untertouren und Fahrten definiert. Diese werden automatisch aufgerufen, wenn der Benutzer eine Tour, Untertour oder Fahrt im Dialog bearbeitet. Da dieser explizit Verstöße gegen Konsistenzbedingungen zulassen kann, ist es außerdem möglich, die Konsistenzprüfung der Touren manuell zu starten, um einen Überblick über die bewußt akzeptierten Unregelmäßigkeiten zu bekommen.

Für die leicht nachzuvollziehenden Laufzeitabschätzungen der Konsistenzprüfungen werden folgende Bezeichnungen benutzt:

DW Menge aller Dienstwünsche

T Menge aller Touren

K Menge aller Kunden

DW_k Menge der Dienstwünsche des Kunden $k \in K$

DW_t Menge der Dienstwünsche, die auf Tour $t \in T$ erfüllt werden

AZ_m Menge der Ausnahmezeiten für die Verfügbarkeit des Mitarbeiter m

AZ_f Menge der Ausnahmezeiten für die Verfügbarkeit des Fahrzeugs f

MA Menge aller Mitarbeiter

FZ Menge aller Fahrzeuge

QF Menge aller Qualifikationen

Folgende Kriterien werden geprüft:

Mitarbeiter

- Sind erster und zweiter Mitarbeiter unterschiedlich? $\mathcal{O}(1)$
- Ist ein zweiter Mitarbeiter eingeteilt, falls die Dienstwünsche dies verlangen? $\mathcal{O}(|DW_t|)$
- Sind die Mitarbeiter am Tag einer Fahrt verfügbar? $\mathcal{O}(|AZ_m|)$
- Sind die Mitarbeiter für alle Kunden einer Tour zulässig (also nicht bei den Abneigungen der Kunden erfaßt)? $\mathcal{O}(|DW_t| \cdot |MA|)$
- Besitzen die Mitarbeiter alle geforderten Qualifikationen? $\mathcal{O}(|DW_t| \cdot |QF|^2)$

Fahrzeuge

- Ist das Fahrzeug am Tag einer Fahrt verfügbar? $\mathcal{O}(|AZ_f|)$
- Wird die maximale Sitzplatzzahl eingehalten (bei Fahrdiensten)? $\mathcal{O}(1)$
- Ist das Fahrzeug unter den für den Kunden zulässigen (bei Fahrdiensten oder MSD, wenn der Kunde mitfährt)? $\mathcal{O}(|DW_t| \cdot |FZ|)$

Zeiten

- Werden die Zeitvorgaben in den Rhythmen der Dienstwünsche eingehalten? $\mathcal{O}(|DW_t|)$

Vollständigkeit

- Werden alle Dienstwünsche komplett erfüllt (also alle Teildienstwünsche berücksichtigt)? $\mathcal{O}(|DW| \cdot \sum_{t \in T} |DW_t|) \in \mathcal{O}(|T| \cdot |DW|^2)$

Nicht geprüft wird die Arbeitszeit der Mitarbeiter im Vergleich zu den in den Arbeitszeitprofilen definierten Obergrenzen, da dies zum einen nicht ganz einfach zu implementieren ist und zum anderen für die zersplitterte Arbeitszeitverteilung der Zivis beim DRK nur wenig Aussagekraft besitzt.

3.3.3.3 Vorgehensweise

Die Überprüfung auf Konsistenz der eingegebene Daten erfolgt an zwei Stellen der Eingabe:

1. Beim Löschen von Objekten wird die Referenzintegrität geprüft: Es dürfen nur solche Objekte gelöscht werden, die von keinem anderen Objekt referenziert werden.

   Dazu muß z.B. für Qualifikationen jeder Mitarbeiter geprüft werden, ob diesem die Qualifikation zugeordnet ist, sowie jeder Dienstwunsch, ob dieser die Qualifikation fordert. Ist also M die Menge der Mitarbeiter und DW_k die Menge der Dienstwünsche des Kunden k, hat die Prüfung, ob eine Qualifikation gelöscht werden darf, den Aufwand $\mathcal{O}((|M| + \sum_{k \in K} |DW_k|) * |QF|)$.

   Dieser einmalige große Aufwand ließe sich in mehrere kleinere Teile auftrennen, indem jede Qualifikation Rückwärtsreferenzen auf alle Mitarbeiter und Dienstwünsche mitführt, die die Qualifikation referenzieren. Dadurch verlagerten sich allerdings Aufgaben der Konsistenzprüfung in die Datenobjekte, wodurch auch das abgespeicherte Szenario stark wüchse. Da das Löschen von Objekten durch den Benutzer keine zeitkritische Aufgabe darstellt (und auch bei einigen hundert Kunden mit insgesamt einigen tausend Dienstwünschen noch in akzeptabler Zeit durchzuführen sein sollte), wurde hier die einfach und sicher zu implementierende Methode gewählt.

2. Beim Ändern von Objekten (dazu gehört auch das Setzen von Daten in neu erzeugten Objekten) wird kontrolliert, daß allen vorgeschriebenen Datenfeldern auch wirklich ein Wert zugewiesen wurde (Mußwert-Prüfung), außerdem werden die Werte bestimmter Felder auf Einmaligkeit

(Schlüsselwert-Prüfung) und Einhaltung von Wertebereichen bzw. andere Kriterien für sinnvolle Werte geprüft.

Für die Schlüsselwert-Prüfung ist jeweils eine Iteration über alle Objekte derselben Art notwendig, um das Schlüsselattribut zu vergleichen. Ist also K die Menge aller Kunden, hat die Überprüfung einer eingegebenen Kundennummer auf Einmaligkeit den Aufwand $\mathcal{O}(|K|)$.

Eine eventuelle Konsistenzverletzung wird dem Programm durch Exceptions der Prüffunktion mitgeteilt. Zu diesem Zweck wurde eine Menge von hierarchisch voneinander abgeleiteten Exceptions definiert, die für die verschiedenen Arten von Verstößen gegen die Konsistenzregeln stehen. Dadurch kann dem Benutzer eine genaue Rückmeldung gegeben werden, mit deren Hilfe er seine Eingabedaten nochmals durchsehen und berichtigen kann. Die Vererbungshierarchie der Exceptions mit einer gemeinsamen Wurzel in der Klasse `KonsistenzException` ermöglicht die einfache Behandlung solcher Exceptions im Programm, ohne nach jeder Eingabe zwischen diversen möglichen Konsistenzfehlern unterscheiden zu müssen.

Durch diese einheitliche Gestaltung der Konsistenzprüfung besteht auch kein wesentlicher konzeptioneller Unterschied zwischen „harten“ Integritätsbedingungen, die auf jeden Fall erfüllt sein müssen, da sonst die Daten in sich keinen Sinn geben, und „weichen“ semantischen Konsistenzbedingungen (im Entwurf als „Vorgaben“ bezeichnet), die Beziehungen zwischen Objekten der realen Welt beschreiben und vom Benutzer im Einzelfall außer Kraft gesetzt werden können.

Verstöße gegen Integritätsbedingungen werden durch Exceptions angezeigt, die von der Klasse `IntegritätsException` abgeleitet sind. Andere, von `KonsistenzException` abgeleitete Exceptions stehen für semantische Konsistenzverletzungen. Somit kann durch eine minimale Fallunterscheidung bei der Dateneingabe festgestellt werden, ob die Daten im eingegebenen Zustand akzeptiert werden dürfen.

Sämtliche Konsistenzprüfungen liegen in einem eigenen Modul `TRO.Konsistenztests`, das zu jeder zu prüfenden Datenklasse eine zugehörige Prüfklasse definiert. Durch diese Trennung der Konsistenzprüfungen von sowohl Dateneingabe als auch den Daten selbst kann die interne Durchführung der Prüfungen ohne Auswirkungen auf das restliche System geplant und umgesetzt werden. Eine Veränderung der Prüfalgorithmen oder auch die Veränderung der Prüfkriterien und dadurch notwendige Hinzunahme neuer Tests oder Auslassung bestehender kann somit lokal im Modul `TRO.Konsistenztests` erfolgen. Änderungen an anderen Programmteilen sind bis auf eventuelle kleine Veränderungen der Aufrufschnittstelle nicht nötig.

3.3.4 Analysedaten

Aus der Fülle der in der Spezifikation geplanten Analysedaten wurden im Gespräch mit dem Kunden diejenigen ausgewählt, die für den Kunden wichtig und interessant sind:

- Auslastung der Mitarbeiter
- Ressourcenbindung (= Fahrzeugauslastung)
- Anzahl km und Zeit pro Dienstart

- Ausfallzeiten der Mitarbeiter und der Fahrzeuge

3.3.4.1 Darstellung

Für die grafische Darstellung der Auslastungen wurde eine generische Klasse entworfen, die für jeden Mitarbeiter bzw. jedes Fahrzeug nach einer textuellen Beschreibung des Mitarbeiters (Name und Mitarbeiternummer) bzw. des Fahrzeugs (Fahrzeugnummer) einen Balken darstellt, dessen Länge dem prozentualen Verhältnis der Auslastung des Mitarbeiters bzw. des Fahrzeuges entspricht. Den Balken werden dabei zwei Farben zugewiesen: Solange die Auslastung unter 100% bleibt, ist der Balken grün, steigt die Auslastung jedoch über 100% wird der Balken in roter Farbe dargestellt. Die Übergabe der Analysedaten an die Klasse zur grafischen Darstellung (TRO.Analyse.ProzentGrafik) erfolgt mittels einer eigens dafür entworfenen Record-Klasse (TRO.Analyse.ProzentRecord). Die Darstellung der anderen Analysedaten, Zeit- und km-Aufwand bzw. Ausfallzeiten, erfolgt in tabellarischer Form. Dem Benutzer bietet sich auch die Möglichkeit diese Pläne auszudrucken.

3.3.4.2 Vorgehensweise

Auslastung der Mitarbeiter Die Analyse der Mitarbeiterauslastung erfolgt über die geplanten Touren bzw. Untertouren in einem vom Benutzer frei wählbaren Zeitrahmen, in dem die Sollarbeitszeit der Mitarbeiter (im Arbeitszeitprofil festgehalten) mit ihrer geplanten Einsatzzeit ins Verhältnis gesetzt wird. Daraus folgt, daß es dem Benutzer nur möglich ist, seine Planung zu analysieren und gegebenenfalls zu verbessern. Eine Feststellung der tatsächlichen Auslastung ist nicht vorgesehen. Um also ein realistisches Ergebnis zu erhalten, wird eine Zuweisung der Sollmitarbeiter in den Touren erforderlich, d.h. ein Offenlassen dieser Zuordnung, die dem Benutzer in einer frühen Planungsphase möglich ist, führt zu Verfälschungen des Ergebnisses.

Ressourcenbindung Die Analyse der Fahrzeugauslastung erfolgt nur für diejenigen Dienstwünsche, bei denen Gruppen von Kunden transportiert werden (Schulfahrt, Tagespflege und Dialyse). Dabei wird in einem vom Kunden frei zu wählenden Zeitraum die für Fahrgäste verfügbare Sitzplatzzahl eines Fahrzeuges mit der für den gewählten Zeitraum durchschnittlichen Zahl mitfahrender Kunden ins Verhältnis gesetzt.

Zeit- und km-Aufwand pro Dienstart Hierbei werden pro Dienstart und pro Untertour die gefahrenen km bzw. der zeitliche Aufwand für die Untertour aufsummiert.

Ausfallzeiten der Mitarbeiter / Fahrzeuge Bildung einer Summe über die vom Benutzer im System pro Mitarbeiter bzw. pro Fahrzeug eingetragenen Ausfallzeiten. Da im System keine genaue Aufteilung über die Art des Ausfalls der einzelnen Ressourcen erfolgt, kann diese folglich auch nicht in der Analyse dargestellt werden. Dieser Analysepunkt soll dem Benutzer vielmehr mittels eines Überblicks Auffälligkeiten aufzeigen, die er dann verfolgen kann.

3.3.5 Datenausgabe

3.3.5.1 Einzeldienstplan

Der Einzeldienstplan wurde als Balkendiagramm realisiert. Er kann für Objekte, die das DienstplanErstellbar-Interface implementieren, über eine beliebige Datumsspanne erzeugt werden. Zur Auswertung wird die Methode objektFahrten des Tourszenarios aufgerufen. Damit liegt der Aufwand, wie im Abschnitt 3.3.2.2 beschrieben, in O(Anz. d. Fahrten).

Das im Abschnitt 2.1.2 beschriebene DienstplanErstellbar-Interface dient als Schnittstelle, die für beliebige (sinnvolle) Objekte implementiert werden kann. Dadurch ist es möglich, neben Dienstplänen für Mitarbeiter auch Fahrzeugeinsatzpläne zu erzeugen und darzustellen, wann ein Kunde vom DRK betreut wird.

3.3.5.2 Gesamtdienstplan

Der Gesamtdienstplan wurde als Exportfile implementiert. Dieses Exportfile enthält zu allen Mitarbeitern die Fahrten, die sie im angegebenen Zeitraum zu erledigen haben. Dieses Exportfile kann dann mit einem anderen Programm (z.B. einer Tabellenkalkulation) beliebig aufbereitet und gedruckt werden. Auch der Gesamtdiensplan kann, ebenso wie der Diensplan, für alle Objekte erzeugt werden, die das DienstplanErstellbar-Interface implementieren. Dazu wird wieder die in 3.3.2.2 beschriebene Methode objektFahrten des Tourszenarios verwendet, die für jedes Objekt einmal aufgerufen wird. Der Aufwand liegt damit in $\mathcal{O}$(Anz. d. Objekte * Anz. d. Fahrten). Eine effizientere Methode wäre das einmalige Durchlaufen aller Fahrten, wobei für jedes Objekt ein Zähler (z.B. in einer Hashtable) mitgeführt wird. Der Aufwand läge dann, wie bei objektFahrten, in $\mathcal{O}$(Anz. d. Fahrten). Wegen der Code-Wiederverwendung und aus Zeitgründen wurde diese Variante jedoch nicht implementiert.

3.3.5.3 Tourplan

Der Tourplan listet zu allen Touren die zugehörigen Untertouren mit den geplanten Mitarbeitern, dem geplanten Fahrzeug und den anzufahrenden Stationen incl. dortiger Ankunfts- und Abfahrtszeit auf. Zudem wird bei jeder Station angegeben, welche Personen ein- oder aussteigen. Ferner werden für alle im Rahmen der Tour beförderten Kunden die Bezugspersonen ausgegeben, die beim zugehörigen Dienstwunsch angegeben wurden.

3.4 Probleme beim Umsetzen des Entwurfs

3.4.1 Unvollständige Details im Entwurf

Einige Probleme bei der Umsetzung des Entwurfs rührten daher, daß im Entwurf nicht alles bis ins Detail zu Ende gedacht war. Hier wurde insbesondere deutlich, daß sich manche sprachlich einfachen Konstrukte als tückisch bei der Implementierung herausstellten. Auf jeden Fall mußte hier der fehlende Entwurfsschritt nachgeholt werden, was teilweise nicht im ersten Versuch gelang.

Als Beispiel soll hier die Speicherung von Essensarten und deren Anzahlen im `EARDienstwunsch` erläutert werden: Der Entwurf läßt mit der Formulierung „Essensart × int“ die genaue Umsetzung offen.

Ein erster Versuch bei der Implementierung benutzte zusätzliche interne Klassen zur Speicherung von Essensarten und Anzahl. Es stellte sich aber heraus, daß diese schlecht zu handhaben sind (eigene Lese- und Setzmethoden sowie Verwaltung der Liste mit Eintragen, Löschen und Suchen wären nötig). Also mußte eine andere Lösung gesucht werden. Die Speicherung der Anzahlen in einer Hashtabelle mit Essensarten als Schlüssel leistet dies auf elegante Weise, ist allerdings nicht ganz trivial in ein gut bedienbares Oberflächenelement zur Eingabe umzusetzen (siehe 3.4.2.1).

3.4.2 Einfach, aber aufwendig

Manche Datenstrukturen lassen sich umgangssprachlich (und auch in der Programmiersprache) leicht ausdrücken, sind aber semantisch recht komplex, so daß ihre Eingabe nicht einfach zu modellieren ist.

3.4.2.1 Hashtabellen

Ein Beispiel dafür sind Hashtabellen, die einer Menge von Schlüsselobjekten je einen Eintrag zuordnen. Da weder unter den gängigen Fenstersystemen, noch unter Java Eingabemöglichkeiten für (in der Größe nicht von vornherein festgelegte) zweidimensionale Tabellen bestehen, sondern nur eindimensionale Listen, mußten die Hashtabellen (bzw. Tabellen aller Art) zur Eingabe in ihre einzelnen Dimensionen zerlegt werden (siehe 3.1.2.6).

3.4.2.2 Teilbare Einheiten

Aufwendig in der Handhabung sind auch Werte mit teilbaren Einheiten: Die Speicherung von Zeitdauern in Minuten ist zweifelsohne sinnvoll, um ohne aufwendige und fehlerträchtige Gleitkommaarithmetik auszukommen. Bei der Eingabe ist es aber in den meisten Fällen nicht zumutbar, alles in Minuten umzurechnen (man denke z.B. an die Erfassung der wöchentlichen Arbeitszeit). Dem Benutzer sollte eine Ausgabe in Form „hh:mm“ (je zwei Stellen für Stunden und Minuten) bzw. getrennte Eingabefelder für Stunden und Minuten angeboten werden.

Hier ist wieder zusätzlicher Aufwand nötig, um einen gegegeben Wert für die Ein- und Ausgabe zu teilen, die Teile einzeln zu editieren und hinterher wieder zusammenzusetzen.

3.4.2.3 Aufwand durch Allgemeingültigkeit

Manche Teile des Entwurfs der Datenklassen wurden bewußt allgemeingültig gehalten, um nicht von vornherein gewisse Eingabemöglichkeiten auszuschließen: Es fand z.B. keine Einschränkung auf zwei Adressen oder Telephonnummern pro Person/Kunde statt, statt dessen kann eine (theoretisch beliebig große) Menge von Adressen, Bank- sowie Kommunikationsverbindungen eingegeben werden.

Dies läßt sich zwar auch in der Benutzungsoberfläche recht einfach umsetzen, ist aber etwas unhandlich in der Bedienung, insbesondere nimmt die Darstellung

solcher Listen einen nicht unerheblichen Teil der doch begrenzten Bildschirmfläche in Anspruch.

3.5 Implementierung ausgehend von einem Prototyp

Um dem DRK schon früh eine anschaulichen Überblick über den Planungsstand geben zu können, wurde bereits parallel zur Anforderungsanalyse ein Prototyp der graphischen Benutzungsoberfläche erstellt (im Akkord über die Weihnachts„ferien“).

Die Existenz dieses Prototyps wurde dann naiverweise als ein bereits beträchtlicher Fortschritt bei der Implementierung gewertet. Daß dem nicht so war, sollte sich recht schnell nach Beginn der tatsächlichen Implementierungsphase herausstellen:

Die Eingabedialoge und Datenfelder waren auf Datenklassen bezogen, die ebenso ad hoc im Kopf entworfen wurden wie der gesamte Prototyp. Da in Spezifikation und Entwurf die Struktur und Zusammenhänge dieser Datenklassen deutlich vom Prototyp abwich, konnte lediglich ein minimaler Teil der alten Oberfläche wiederverwendet werden (um z.B. gewisse Teile von Eingabefenstern aufzubauen). Insbesondere die Anbindung an die Datenklassen (Werte setzen und lesen, siehe 3.1.2.1) war recht aufwendig.

Für eine Anwendung, die so unmittelbar auf der Struktur des Datenmodells aufsetzt, wie dies bei der Datenverwaltung des Programms TROSS der Fall ist, ist es sicher sinnvoller, den Prototyp zu einem späteren Zeitpunkt zu beginnen. Denkbar ist eine Vorgehensweise parallel zum Design des Datenmodells: Zu den bereits spezifizierten Datenobjekten können Eingabemasken entworfen werden, die dann einerseits zur Veranschaulichung des Planungsstandes im Dialog mit dem Anwender dienen, zum anderen aber eine Basis für die endgültige Implementierung bieten, da sicher viele Teile übernommen werden können.

3.6 Verwendung externer Programme: Das Verkehrsmodul

Als wesentliches Problem bei der Implementierung des Verkehrsmoduls hat sich die Batch-Schnittstelle von Map&Guide herausgestellt, die für eine solche Verwendung (zumindest mit der vorhandenen spärlichen Dokumentation) eigentlich nicht geeignet ist. Sie ist dafür ausgelegt, daß die Benutzer ihre Anfragen in einer Auftragsdatei (.asc) an Map&Guide geben, und dieses eine Tourbeschreibung in einer Ergebnisdatei (.lst) speichert, oder die Tour textuell oder graphisch auf einem Drucker ausgibt. Aus der Tourbeschreibung kann zwar prinzipiell die Entfernung zwischen den Stationen herausgelesen werden, aber dafür muß es in dieser Preislage eigentlich besser geeignete Möglichkeiten geben.

Zunächst war die Frage, wie eine Station für Map&Guide aussehen muß. Deshalb wurden einige Beispiele entsprechend dem Handbuch eingegeben. Als erstes Problem stellte sich die Übergabe von Straßennamen heraus, die nicht zu funktionieren schien. Nachdem die Antwort des Supports von Map&Guide nicht funktionierte, fand sich dann doch die Lösung im Handbuch: wenn das erste

Kommentarfeld der Auftragsdatei mit einem Doppelpunkt beginnt, wird der Rest des Feldes als Straße interpretiert. Wer bitte kommt auf solche Ideen?

Als nächstes wurde die Vorschlagsliste (.cor) untersucht, die Map&Guide immer anlegt. Anhand dieser kann eine neue eindeutige Auftragsdatei erstellt werden. Allerdings fehlten dabei stets die Hausnummern, weshalb die gelieferten Stationen zum Teil nicht eindeutig sind. Als erste Lösung wurde die Hausnummer aus der Anfrage einfach wieder an die Straße angehängt. Nach einer Diskussion in einer Sitzung wurde diese Alternative verworfen, da es kaum möglich ist, den Straßennamen algorithmisch von der Hausnummer zu trennen. Deshalb wurde das Attribut Straße in der Station aufgetrennt in zwei Attribute Straße und Hausnummer.

Die nächste Eigenheit von Map&Guide ließ nicht lange auf sich warten: die Postleitzahlen in der Vorschlagsliste waren zum Teil falsch (9xxxx für Stationen in Stuttgart), zum Teil aber auch große negative Zahlen. Diese Vorschläge in eine neue Auftragsdatei gepackt brachten wie erwartet keine sinnvollen Ergebnisse.

Aufgrund dieser Probleme mit der Vorschlagsliste werden die Vorschläge nun nicht mehr zur Erzeugung neuer, gültiger Anbindungspunkte verwendet. Statt dessen muß jeder mögliche Anbindungspunkt geprüft werden (Methode `Verkehrsmodul.anbindungInOrdnung`), und nur wenn in der Vorschlagsliste nur eine Alternative zurückgegeben wird, ist der Anbindungspunkt gültig. Die Alternativen werden dem Benutzer nur noch zur Hilfe angezeigt.

Als nächstes Problem trat in Map&Guide eine falsch kodierte Straße auf: die Sommerrainstraße in Stuttgart Sommerrain ist in zwei Teile aufgeteilt, deren Hausnummern-Mengen nicht disjunkt sind. Deshalb sind Hausnummern größer 41 nie eindeutig. In solchen Fällen gibt es zwei Möglichkeiten: entweder einfach eine andere Hausnummer, oder die geographischen Koordinaten als Anbindungspunkt eingeben (näheres dazu steht im Handbuch von Map&Guide [CAS97]).

Um die Arbeitsweise von Map&Guide zu beeinflussen, gibt es zum einen die Parameterdatei (param.mgb), zum anderen können die Parameter aber auch an den Anfang der Auftragsdatei gestellt werden. Nur funktionierte die zweite Möglichkeit zunächst nicht. Nach etlichen Stunden war die Ursache gefunden: Wenn Map&Guide die Parameterdatei nicht findet, ignoriert es auch die Parameter in der Auftragsdatei. Deshalb wird nun eine Parameterdatei angelegt, wenn noch keine vorhanden ist.

Zu klären war auch die Frage, wie viele Stationen Map&Guide wohl in einer Anfrage schafft. Im Handbuch steht nur, daß bei aktivierter Reihenfolge-Optimierung der Stationen maximal 20 Zwischenstationen möglich sind. Es stellte sich heraus, daß überhaupt nur 20 Stationen, d.h. 19 Kanten möglich sind. Dadurch wird die Anfrage an das Verkehrsmodul zum Vorbereiten aller Entfernungen zwischen je zwei Stationen einer größeren Menge eigentlich unbrauchbar. Ein geringe Verbesserung wird durch einen anderen Algorithmus erreicht (siehe 2.2.3).

Das Erkennen der Fehler von Map&Guide ist nicht einfach, da je nach Art des Fehlers manche Dateien erzeugt werden und andere nicht. Eine Datei, die sich bei einem Fehler immer ändert, ist die Fehler-Datei Error.mgb. Ihr Inhalt ist aber kaum für eine Auswertung per Programm geeignet, da die Anzahl der angehängten Zeilen schwankt. Das Verkehrsmodul meldet nun einen Fehler, wenn sich der Zeitstempel dieser Datei geändert hat, kann aber keine genaueren Angaben zum Fehler machen. Wenn nur Anbindungen geprüft werden, wird dieser Fehler ignoriert.

3.7 Erfahrungen mit Java

3.7.1 Javas Klassenbibliothek

Javas Klassenbibliothek enthält zwar alle grundlegenden Objekte und Funktionen zur Verwaltung von Daten und Erstellung von graphischen Benutzungsoberflächen, allerdings auch nicht mehr. Semantisch höhere Konstrukte sind bis auf Ausnahmen kaum zu finden, vor allem das „Abstract Window Toolkit“ (AWT) für die Programmierung von Oberflächen bietet nur die üblichen Primitive, die von den gängigen Fenstersystemen direkt zur Verfügung gestellt werden.

Manche grundlegenden Konzepte, die in anderen objektorientierten Programmiersystemen zum Standard gehören, sind in Java noch unausgereift oder schlicht nicht vorhanden. Als Beispiel sei hier die Eigenschaft der Vergleichbarkeit genannt, die Voraussetzung für das Sortieren von Objekten ist. Zwar haben manche Java-Klassen bereits eine Methode namens `compareTo` implementiert, die diesen Vergleich durchführt, jedoch ist diese Methode nicht Teil eines Interface, mit dessen Hilfe man die Vergleichbarkeit von Objekten erzwingen könnte (um sie z.B. in eine sortierte Liste aufzunehmen). Folgerichtig gibt es auch keine Listenklassen, die ihren Inhalt automatisch sortieren. Da im Projekt TROSS solche sortierten Listen aber gebraucht wurden, mußten die meisten Datenklassen mit zwei selbsterstellten Interfaces implementiert werden (siehe 3.1.1.1).

Der Verzicht auf Mehrfachvererbung löst zwar gewisse semantische Eindeutigkeitsprobleme, bedeutet aber manchmal zusätzlichen Programmieraufwand durch Verlust an Modularität. Javas Interfaces können zwar wie abstrakte Superklassen gehandhabt werden, bieten aber nicht die Möglichkeit des „code reuse“ durch Vererbung von Methoden.

Im Projekt wäre es z.B. wünschenswert gewesen, sowohl Personen als auch Institutionen in einen Zusammenhang mit Kunden zu bringen. Dies könnte durch eine Superklasse `Bezug` realisiert werden, von der sowohl `BezugsPerson` als auch `Institution` erben. Da `BezugsPerson` aber auf jeden Fall von `Person` erbt, mußte diese Realisierungsmöglichkeit ausscheiden. Wir haben uns für die Verknüpfung von Kunde und Institution über einen Sachbearbeiter entschieden, der eine Bezugsperson ist und von dieser Klasse erbt.

Mehrfachvererbung hätte es auch ermöglicht, dem in 3.1.1.1 erwähnten Interface `Comparable` eine Standard-Implementierung der Vergleichsmethode mitzugeben. In Java mußte der etwas aufwendige Stringvergleich nach länderspezifischen Sortierregeln vielfach redundant codiert werden.

3.7.1.1 Das Abstract Window Toolkit (AWT)

Auch im package `java.awt` zur Erstellung der graphischen Benutzungsoberfläche wurden einige vielverwendete Klassen schmerzlich vermisst. Zum Beispiel mußte ein LayoutManager, der die simple Aufgabe erfüllt, Oberflächenelemente untereinander in einem Dialog anzuordnen, komplett selbst implementiert werden.

Besonders deutlich bemerkbar macht sich die fehlende Unterstützung typisierter Eingabefelder, sowohl beim Setzen als auch beim Auslesen von Werten: Das JDK bietet nur reine Textfelder als fertige Oberflächenelemente an. Um andere Datentypen als Strings zu erfassen, muß sich der Programmierer um die Konvertierung der Größe in einen String genauso kümmern wie um das Umsetzen

der eingegebenen Zeichenkette in das gewünschte Datum. Vor allem letzteres ist mit einem gewissen Aufwand verbunden, da diverse Exceptions behandelt werden müssen. Selbst für elementare Datentypen wie ganze Zahlen (`int`) muß der Umweg über Textfelder von Hand programmiert werden.

3.7.2 Entwicklungsumgebung

Die Entwicklungsumgebung für das Projekt TROSS zeichnete sich vor allem dadurch aus, daß keine Entwicklungsumgebung vorhanden war (Programmierung, Compilieren und Testen mußten von Hand via Editor und Kommandozeilenaufrufe der Java-Tools gemacht werden). Zwei Punkte dürften dazu wesentlich beigetragen haben:

1. Die Entscheidung, welche Programmiersprache eingesetzt wird, wurde erst sehr spät geführt. Das führte dazu, daß der gesamte Prototyp von Hand erstellt werden mußte. Eine komplette Neuerstellung wäre wohl auch mit Entwicklungsumgebung recht zeitaufwendig geworden. Darüberhinaus hätte die Suche nach einer geeigneten Entwicklungsumgebung sowie die Einarbeitung darin einige Projektzeit verschlungen.

   Wäre zu Beginn der Projektgruppe wenigstens dieses Grundhandwerkszeug dagewesen, hätte sicher mancher spätere Aufwand vermieden werden können.

2. Da zu Beginn der Projektgruppe die Maxime ausgegeben wurde, alle verwendeten Programme dürften möglichst nichts kosten, war es kaum möglich, eine leistungsfähige Entwicklungsumgebung zu bekommen.

Wir mußten also auch ohne GUI-Builder auskommen, der beim Erstellen der Benutzungsoberfläche eine beträchtliche Hilfe dargestellt hätte. Viel Handarbeit war angesagt, um zusätzliche Oberflächenelemente zu realisieren und die bestehenden mit ihrer rudimentären Schnittstelle zu bedienen. Insbesondere das Setzen und Auslesen von Eingabewerten (siehe 3.1.2.1) hätte sich mit einem gut konzipierten GUI-Builder ergeblich vereinfacht.

Lange Zeit gab es auch regelmäßige Probleme mit dem Versionsverwaltungssystem CVS, dessen Funktionsweise und Bedienung nicht unmittelbar einleuchtete. Erst nach wiederholten Datenverlusten beim Einchecken neuer Versionen wurde die korrekte Handhabung klar.

Kapitel 4

Test

4.1 Funktionstest des Gesamtprogramms

Eigentlich hätte vor Beginn der Testphase ein Schnitt gemacht werden sollen, so daß nach einem Implementierungsstop eine definierte „Version 0“ des Programms zur Verfügung gestanden hätte. Da dies durch krampfhaftes Festhalten an der Fertigstellung gewisser Programmteile so nicht realisiert werden konnte, gründete der Funktionstest nicht nur auf von Teil zu Teil unterschiedlichen Versionen, sondern stellte teilweise auch Fehler fest, die mittlerweile längst behoben waren. Da es sich hierbei teilweise um wichtige Programmteile handelt, sollen hier kurz diejenigen Punkte wiedergegeben werden, die der Funktionstest als fehlerhaft beschreibt, die aber im Programm zum Zeitpunkt des Endberichts korrigiert waren:

4.1.1 Szenario

Für die folgenden Tests wurden verschiedene Szenarien (u.A. mit DRK-Testdaten, mit generierten Testdaten und mit Daten aus 4.3) verwendet.

Die Menüpunkte *neu*, *öffnen* und *Szenario zum Masterszenario machen* zeigten bei mehrmaligen Aufrufen und auch bei zwischenzeitlichem Verlassen des Programms die gewünschte Wirkung. Das Speichern scheint jedoch Probleme zu machen (oder kam nicht immer der Hinweis „Szenario geändert! - Speichern?“)? Jedenfalls waren nicht immer alle Daten da, die zuvor eingegeben wurden. Allerdings konnte dieser Fehler bisher nicht reproduziert werden.

Auch beim Test der Funktionen des Untermenüs *TourSzenario* gab es Probleme:

wechseln zeigte keine Wirkung.

kopieren schlägt fehl, da die vom System automatisch aufgerufene Methode korrigiereFahrzeiten() zu einer Exception führte.

zum Master machen Funktioniert auf den ersten Blick, allerdings sieht man auf der Konsole, daß das unbekannte Kommando *tourszenario_fahrten_schreiben* aufgerufen wird.

Die übrigen Menüpunkte zeigten bei mehrmaligen Aufrufen und auch bei zwischenzeitlichem Verlassen des Programms die gewünschte Wirkung.

4.1.2 Kunden und Dienstwünsche

Abgesehen von den in Kapitel 4.3 genannten Fehlern fiel nichts mehr auf. Der Menüpunkt *Dienstwünsche erfüllt?* dürfte den Planer nur für kurze Zeit erfreuen: es werden angeblich immer alle Wünsche erfüllt!

4.1.3 Touren, Untertouren und Fahrten

Die Probleme, die sich hinter dem Menüpunkt *Tourenliste* verbergen, werden im Detail in 4.3 beschreiben. Insbesondere ließen sich keine Untertouren bilden. Damit war auch ein Test aller Menüpunkte/Dialoge, die sich auf Untertouren oder Fahrten beziehen, nicht möglich.

Der Konsistenztest entdeckte fehlende Mitarbeiter und Qualifikationen. Ebenso wurden zugewiesene, aber vom Kunden abgelehnte Mitarbeiter korrekt entdeckt. Auf mehrfach verplante Mitarbeiter und Fahrzeuge wurde jedoch nicht hingewiesen.

4.1.4 Test der Menüpunkte Ressourcen, Ausgabe, Analyse und Einstellungen

4.1.4.1 Vorgehensweise

In diesem Testabschnitt sollten die Eingabemasken für Mitarbeiter, Fahrzeuge und die systemweit bekannten Daten (=Einstellungen), sowie die Ausgabe- und Analysefunktionen auf ihre Funktionalität hin überprüft werden. Die Vorgehensweise für den Test der Eingabemasken war folgende: Eingabe neuer Daten (auch falsche Formate), Ändern und Löschen bestehender Daten. Für den Test der Ausgabe- und Analysefunktionen wurde auf schon bestehende Daten zugegriffen.

4.1.4.2 Ressourcen

Sowohl die Masken zur Neueingabe, zum Ändern und zum Löschen von Mitarbeitern, sowie die zur Manipulation der Fahrzeugdaten funktionieren im gewünschten Umfang. Bei der Neueingabe diverser Daten (z.B. Adressen, Kommunikationsverbindungen...) eines für das System neuen Mitarbeiters bzw. Fahrzeugs erscheint im Titel des Fensters nicht der Name des Mitarbeiters bzw. Fahrzeugs, sondern *null null*. Dies ist bedingt durch das Konzept des Zwei-Phasen-Commit, welches sämtliche Eingaben nach Drücken des OK Button erst prüft und danach dem System bekannt macht.

4.1.4.3 Ausgabe

Die Menüpunkte zur Ausgabe waren zum Zeitpunkt an dem der Test durchgeführt wurde, teilweise noch nicht vollständig implementiert und funktionierten dementsprechend noch nicht.

4.1.4.4 Analyse

Die Menüpunkte zur Anzeige der Mitarbeiterauslastung und zur Fahrzeugbesetzung zeigten die gewünschten Daten richtig an. Die weiteren Analysefunktionen

waren zum Testzeitpunkt noch nicht vollständig implementiert und konnten somit nicht getestet werden. Die Funktionen zum Drucken funktionieren mit der Solaris-Version des JDK nicht (X11MOTIF-Exception).

4.1.4.5 Einstellungen

Die Masken zur Eingabe der einzelnen, systemweit bekannten Daten, funktionierten zur vollen Zufriedenheit. Auch falls die Daten in anderen Masken (z.B. Fahrzeugeingabe) verändert oder ergänzt werden. Die Eingaben falscher Datenformate (z.B. Eingabe von Buchstaben statt Zahlen) werden von den Konsistenztests sicher erkannt und erfolgreich abgewiesen.

4.1.4.6 Allgemeine Systemprobleme

Bei manchen Testläufen kam es vor, daß das System ohne Fehlermeldung *abstürzte* oder einfach *hängen* blieb und dadurch von außen beendet werden mußte. Da diese Fehler in unterschiedlichen Bediensituationen auftraten, liegt die Ursache wahrscheinlich in einer gewissen Instabilität der Java-Laufzeitumgebung.

4.1.5 Probleme unter Windows und JDK1.1.5

- Die Eingabe des Zeichens „@“ ist nicht möglich.
- Der Warte-Mauscursor wird erst nach einer Mausbewegung wieder in einen Aktiv-Mauscursor geändert. Spielt man nicht die ganze Zeit mit der Maus, kann man lange warten, bis eine Aktion beendet ist.
- Zeitrahmen werden immer um einen Tag nach vorne geschoben

4.1.6 allgemeine Fehler

- Bilden von Untertouren nicht möglich.
- Es ist nicht möglich, einen Dienstwunsch zu löschen, der von Touren referenziert wird. Es erscheint zwar die Frage „Dienstwunsch aus allen Touren entfernen?“, eine Antwort mit ok wird jedoch nur durch eine Exception quittiert.

4.1.7 Ergänzungen zum Test

Tourszenarien Tourszenarien können jetzt kopiert werden, zwischen verschiedenen Tourszenarien eines Szenarios kann gewechselt werden, und das jeweils aktive Tourszenario kann zum Master-Tourszenario gemacht werden (notwendige Bedingung für das Schreiben von Fahrten).

Untertouren erstellen Untertouren konnten erstellt werden und mit Teildienstwünschen bestückt werden. Das Verschieben einzelner Teildienstwünsche klappte ebenso wie das Verschieben von Halten, wobei die Prüfung auf richtige Reihenfolge der Stationen korrekt arbeitete. Da im Modus ohne Verkehrstool getestet wurde, wurden unbekannte Entfernungen bzw. Fahrzeiten angezeigt und konnten vom Benutzer eingegeben werden.

Fahrten erzeugen Zu den erzeugten Untertouren wurden Fahrten erzeugt. Dabei wurden die Zeiträume der Dienstwünsche beachtet und gegebenenfalls keine neuen Fahrten erzeugt. Die neu erzeugten Fahrten wurden dem Benutzer zur Überprüfung und möglichen Bearbeitung präsentiert. Hier konnten Mitarbeiter und Fahrzeug angepaßt werden, sowie Dienstwünsche entfernt und wieder eingefügt werden.

Das Erstellen von Dienstplänen auf der Basis dieser Fahrten gelang, ebenso wie das Archivieren von Fahrten in eine Log-Datei.

Erfüllung von Dienstwünschen prüfen Es wurden zwei Dienstwünsche überprüft, von denen einer teilweise in Touren und Untertouren enthalten war. Das Programm meldete korrekt alle Teildienstwünsche dieser Dienstwünsche, die nicht in den bestehenden Untertouren erfüllt wurden.

Löschen von Dienstwünschen, die in Touren enthalten sind Nach der Warnmeldung, daß der zu löschende Dienstwunsch noch in Touren steckt, wurde die Option gewählt, diesen automatisch auch dort entfernen zu lassen. Nach Eingabe einer Entfernung (die in der Untertour durch Wegfall des Dienstwunsches benötigt wurde) war der Dienstwunsch sowohl aus der Wunschliste des Kunden als auch aus der Tour und Untertour gelöscht, in denen er zuvor noch referenziert worden war.

Drucken von Analysen und Plänen Pläne und Analysedaten können jetzt auch ausgedruckt werden. Analyselisten werden dabei in Seiten umbrochen, von Dienstplänen wird jeweils nur der Teil ausgedruckt, der auf eine Seite paßt (dies ist etwa eine Woche).

4.2 Grenzen des Systems: Test mit großen Datenmengen

Um die Grenzen des Systems zu erkunden, wurde ein Programm geschrieben, das große Mengen zufälliger Daten anlegt, wobei verschiedene Parameter eingestellt werden können. Die Daten werden in Szenarien gespeichert, die dann von TROSS geladen werden können. Mit diesem Programm wurden zwei Szenarien erzeugt:

1. Das erste Szenario enthält je 50 Qualifikationen, Essensarten, Hilfsmittel und Fahrzeugtypen, je 200 Fahrzeuge und Mitarbeiter, 500 Kunden mit jeweils 0 bis 2 Dienstwünschen, 10 Schulen, 150 Schultouren mit 3 bis 6 Kindern und die Untertouren zu den Schultouren (eine Hin- und eine Rückfahrt).

2. Im zweiten Szenario wurde die Zahl der Kunden auf 1000 erhöht, die Zahl der Schulen auf 20 und die Zahl der Schultouren auf 300.

Diese Szenarien wurden geladen und verschiedene Funktionen von TROSS aufgerufen, um die Schwachstellen zu finden. Für diese wurden dann Zeitmessungen durchgeführt. In Tabelle 4.1 werden die nacheinander aufgerufenen Funktionen und die benötigten (gestoppten) Zeiten in Sekunden auf zwei Rechnern (beide Windows 95 mit JDK 1.1.6) angegeben.

Funktion	Cyrix 166/32MB	P100/24MB
Szenario 1 laden	115	319
Kunden anzeigen	5	12
speichern als Szenario 3	100	152
Szenario 3 laden	110	307
Kunden anzeigen	2,5	8
Szenario 2 laden	305	912
Kunden anzeigen	8	18
Touren anzeigen	2	5

Tabelle 4.1: Zeiten für verschiedene Funktionen

Das große Problem: das Laden (und Speichern) der Daten. In TROSS wird die Serialisation von Java verwendet, wobei die Daten noch komprimiert werden. Auch das Abschalten der Komprimierung änderte kaum etwas. Der Profiler ermittelt für das Laden des kleinen Szenarios die folgenden Top-Rechenzeit-Verbraucher:

59,4% ObjectInputStream.inputClassFields(Object, Class, int[])
20,9% System.gc()
11,2% Object.wait(long)

Alle drei Methoden kommen von Java. Die erste Methode ist nicht dokumentiert, dem Namen nach liest sie die Attribute der Objekte. Die zweite ist der Garbage-Collector und die dritte dient zur Synchronisation paralleler Zugriffe. Eine Laufzeit-Optimierung ist hier also nicht möglich, am einfachsten können die 21% für den Garbage-Collector durch mehr Speicher verringert werden.

Die übrigen gestoppten Zeiten wären auf dem Cyrix-Prozessor gerade noch hinnehmbar, das Laden und Speichern mittels Objekt-Serialisation ist aber nicht akzeptabel, kann aber wegen Zeitmangels nicht mehr geändert werden.

Für die folgenden Versuche wurde ein viertes Szenario mit vom zukünftigen Benutzer von TROSS gelieferten Daten erstellt. Dieses enthält 46 Fahrzeuge, 63 Mitarbeiter, 94 Kunden mit Dienstwünschen (Essen auf Rädern und Schulfahrten) und 11 Schultouren.

Bei ausgeschaltenem Verkehrstool wurden die Dienstwünsche angelegt. Danach wurde das Verkehrstool eingeschaltet. Für das Anbinden der 95 Stationen (94 und eine Schule) wurden 95 Sekunden benötigt. Danach wurden für alle 11 Schultouren die Entfernungen zwischen den einzelnen Stationen ermittelt. Das dauerte 160 Sekunden. Hier wurden etwa $160/11 = 15$ Sekunden pro Map&Guide Auftrag.

Der benötigte Speicherplatz auf der Festplatte hält sich dank der Kompression in Grenzen. Tabelle 4.2 zeigt die Dateigrößen in Bytes verschiedener Szenarien. Weshalb das Szenario 3 kleiner ist als Szenario 1, ist nicht erklärbar.

Die Szenarien in den letzten drei Zeilen enthalten jeweils noch die Fahrten der Schultouren für 1, 2 bzw. 3 Wochen. Pro Fahrt werden etwa $(17905 - 17187)/220 = 3,2$ Bytes benötigt. Dank der Kompression gibt es hier also keine Engpässe.

Szenario	Größe in Bytes
Szenario 1	134.472
Szenario 2	254.954
Szenario 3	134.423
Szenario 4	16.292
Szenario 4 + 1 Woche	17.187
Szenario 4 + 2 Wochen	17.513
Szenario 4 + 3 Wochen	17.905

Tabelle 4.2: Größen verschiedener Szenarien

4.3 Abgleich mit den Anforderungen: Durchspielen der Szenarien

Ein Teil der Testphase bestand darin, die Daten aus den Szenarien der Anforderungsanalyse mit TROSS zu verarbeiten. Im folgenden soll dargestellt werden, welche Teile mit TROSS verarbeitet werden konnten, welche sich aufgrund von Fehlern nicht verarbeiten ließen und welche aus konzeptionellen Gründen nicht mit TROSS verarbeitet werden konnten.

4.3.0.1 Schulfahrten

Szenario 1 Die Qualifikationen „Lastenheben“ und „Epilepsie-Umgang“ wurden neu angelegt. Daraufhin wurden die Zivildienstleistenden Gustav Häberle und Karl Tremmle gemäß den Vorgaben des Szenarios erfaßt. Danach wurden die vier vorgegebenen Kunden erfaßt und zu einer Tour zusammengestellt, die wie gefordert für die automatische Optimierung gesperrt wurde. Da die Schülerin Karin Klein krank ist, wurde in ihrem Dienstwunsch die voraussichtliche Dauer der Krankheit vom Zeitrahmen ausgenommen.

Probleme

- Da für Touren keine Bemerkung vorgesehen ist, muß der Auftrag, die Schüler in die Klassen zu bringen, bei jedem Dienstwunsch angegeben werden.
- Der Zivi Karl Tremmle machte zu Beginn seiner Dienstzeit einen Lehrgang. Dies sollte in TROSS durch eine Datumsspanne für die Zeit des Lehrgangs dargestellt werden. Dazu wäre es wünschenswert gewesen, bei den Ausnahemzeiträumen auch Bemerkungen eingeben zu können, damit man weiß, warum ein Mitarbeiter fehlt. Bisher kann nur dargestellt werden, daß ein Mitarbeiter nicht verfügbar ist, aber nicht, warum er fehlt.
- Daß der Klassenlehrer von Karin Klein über deren Krankheit unterrichtet werden soll, ist im System nicht sinnvoll darstellbar, da der entsprechende Dienstwunsch nicht mehr in der Fahrt auftaucht.
- Die Zeitangabe ohne Minuten („16: “) führte zu der verwirrenden Fehlermeldung „ganze Zahl eingeben“.

Fehler

- Beim Erstellen von Untertouren trat eine Exception auf. Daher konnte auch nicht festgestellt werden, ob es möglich ist, daß die Schülerin Susanne Schlecht auf der Hinfahrt garantiert als letzte abgeholt und bei der Rückfahrt als erste abgeliefert wird.

Szenario 2 Das Szenario konnte mit dem System komplett umgesetzt werden. Ob die Änderungen der Dienstwünsche in den Touren durch Umsetzen der Kunden in den Untertouren und Fahrten richtig nachvollzogen wird, konnte nicht festgestellt werden, da es nicht möglich war, Untertouren zu bilden.

4.3.0.2 MSD / Pflegedienst

Szenario 1 Das Szenario konnte mit TROSS nachvollzogen werden. Die Vorgabe, beide Dienstanforderungen als einen Dienstwunsch zu erfassen, kann nachvollzogen werden, hat im System jedoch folgendes Problem: Auf der Tour muß immer ein teurer Pfleger mitgehen, auch wenn nur Einkäufe zu erledigen sind.

Szenario 2 Bei der Umsetzung des zweiten MSD-Szenarios traten zwei Fehler auf. Zum einen war gefordert, einen bevorzugten Mitarbeiter festzulegen. Dies geschah bei der Eingabe des Dienstwunsches, allerdings war die Eingabe beim nächsten Aufruf der Dienstwunschmaske wieder verschwunden.

Das zweite Problem machte die Ausnahmezeiterfassung beim Kunden. Es sollten zwei Zeiträume eingegeben werden, in denen die Kundin nicht bedient werden soll. Allerdings wurde die zweite Eingabe ignoriert und dafür der erste Zeitraum (unter Windows) um einen Tag nach vorne verschoben.

4.3.0.3 Essen auf Rädern

Szenario 1 Das Szenario konnte erfolgreich nachvollzogen werden.

Szenario 2 Bei diesem Szenario traten folgende Probleme auf:

Zunächst sollte eine Email-Adresse eingegeben werden, da der Kunde per Email bestellte. Dabei fiel auf, daß unter Windows95 die Eingabe des Zeichens „@“ nicht möglich ist, obwohl dies unter UNIX funktioniert.

Das nächste Problem war die nachträgliche Adreßerfassung in der Dienstwunschmaske. Da zunächst keine Lieferadresse angegeben war, erschien korrekterweise eine entsprechende Fehlermeldung. Daraufhin wurde die Lieferadresse ergänzt. Diese konnte vom Verkehrsmodul nicht angebunden werden, weshalb ein entsprechender Hinweis kam, der mit Abbrechen beendet werden sollte. Leider blieb daraufhin auch TROSS stehen und mußte von außen verlassen werden.

Nachdem der Dienstwunsch wegen obigem Fehler nochmals erfaßt war, sollte sichergestellt werden, daß er am Ende einer Essenstour erledigt wird. Eine Reihenfolge kann jedoch erst bei Untertouren festgelegt werden, die wie schon erwähnt nicht funktionierte. Damit bleibt dieser Punkt ungeprüft. Als Alternative wurde eine späte Lieferzeit eingegeben, die jedoch nichts garantiert.

Die Verquickung mit einem MSD-Dienstwunsch ist aus konzeptionellen Gründen nicht direkt möglich. Dies muß bei der Tourplanung manuell berücksichtigt werden (wobei die beiden Dienstwünsche wegen der unterschiedlichen Dienstarten nicht in einer Tour erledigt werden dürfen!). Eine andere Möglichkeit wäre

ein MSD-Dienstwunsch mit der Bemerkung „Essen mitnehmen“ oder ein EAR-Dienstwunsch mit einer längeren Aufenthaltszeit. Dies bringt aber Probleme für die (in dieser Version nicht realisierte) automatische Abrechnung mit sich, da sich diese auf die Dienstart stützt und somit entweder ein Essen oder MSD-Stunden in Rechnung stellen könnte.

Szenario 3 Beim Szenario 3 ergeben sich durch die geforderte Verbindung von EAR- und MSD-Dienst die schon oben genannten Probleme. Weitere Probleme traten in Verbindung mit der Erfassung von Institutionen auf:

- Obwohl die Klasse Kunde einen Vektor für die Aufnahme von Institutionen besitzt, ist es ist nicht möglich, dem Kunden direkt, d.h. ohne bekannten Sachbearbeiter, eine Institution zuzuweisen. Ist der Sachbearbeiter unbekannt, muß die Institution über eine Dummy-Bezugsperson referenziert werden.
- Es ist keine Angabe der Versicherten-Nummer vorgesehen. Diese kann zwar im Feld Bemerkung untergebracht werden, verhindert aber (z.B. für automatische Rechnungsstellung) eine Auswertung durch das Systems.

Szenario 4 Das Szenario konnte mit TROSS erfaßt werden. Ein Problem trat unter Windows95, jedoch nicht bei UNIX auf: Der Zeitrahmen wird beim Verlassen der Zeitrahmeneingabe um einen Tag nach vorne verschoben.

Szenario 5 Das Szenario 5 konnte problemlos nachvollzogen werden.

Szenario 6 Der entsprechende Ausnahmezeitraum wurde erfaßt (und nach Szenario 4 erwartungsgemäß verschoben). Auswirkungen auf die Fahrten konnten nicht getestet werden, da es nicht möglich war, Fahrten zu erzeugen.

Szenario 7 Der nicht mehr benötigte Dienstwunsch wurde durch Angabe eines Enddatums in seiner Zeitspanne suspendiert. Die Entfernung aus der entsprechenden Tour muß manuell erfolgen, Auswirkungen auf die Untertouren und Fahrten konnten nicht getestet werden.

Szenario 8 Solche kurzfristigen Änderungen werden vermutlich nicht über das System laufen. Falls doch, müssen die entsprechenden Fahrten geändert werden (2 Fahrten zu einer Untertour, wobei jede einen Teil erledigt). Dies konnte nicht getestet werden, da es nicht möglich war, Fahrten zu erzeugen.

4.3.0.4 Individualfahrten

Szenario 1 Dieses Szenario (Kundin fährt zum Arzt und wieder zurück) kann problemlos nachvollzogen werden.

Szenario 2 Bei der Eingabe dieses Szenarios einer Individualfahrt mit einer Zwischenstation blieb die Frage offen, wann die Zwischenstation im System berücksichtigt wird (auf der Hinfahrt, auf der Rückfahrt, immer?). Außerdem kann nur angeben werden, ob der Mitarbeiter zwischen der Hin- und Rückfahrt

anwesend sein muß, für Zwischenstationen ist keine Angabe möglich. Eine andere Möglichkeit dieses Szenario so einzugeben, daß der Mitarbeiter dem DRK zwischen allen Stationen zur Verfügung steht, wäre die Aufspaltung des Dienstwunsches auf drei Individualfahrten (Wohnung, Friedhof), (Friedhof, Gemeindehaus), (Gemeindehaus, Wohnung).

Außerdem war es wegen Exceptions nicht möglich, überhaupt Zwischenstationen anzugeben.

Szenario 3 Entspricht, abgesehen von der Anfangszeit, dem Szenario 2.

Szenario 4 Ist für die Dateneingabe völlig uninteressant, da die Fahrt abgelehnt wird. Diese Entscheidung kann auf Grundlage der Datenausgabe getroffen werden, deren Testergebnisse in 4.1.4.3 zu finden sind.

Kapitel 5

Architektur des Programms TROSS

5.1 Architektur des System TROSS

Abbildung 5.1 bietet einen Überblick über die gesamte Architektur des System TROSS.

5.2 Graphische Benutzungsoberfläche

Eine schematische Übersicht bietet Abbildung 5.2

5.3 Verkehrsmodul

Abbildung 5.3 zeigt den prinzipiellen Aufbau und die Verwendung des Verkehrsmoduls.

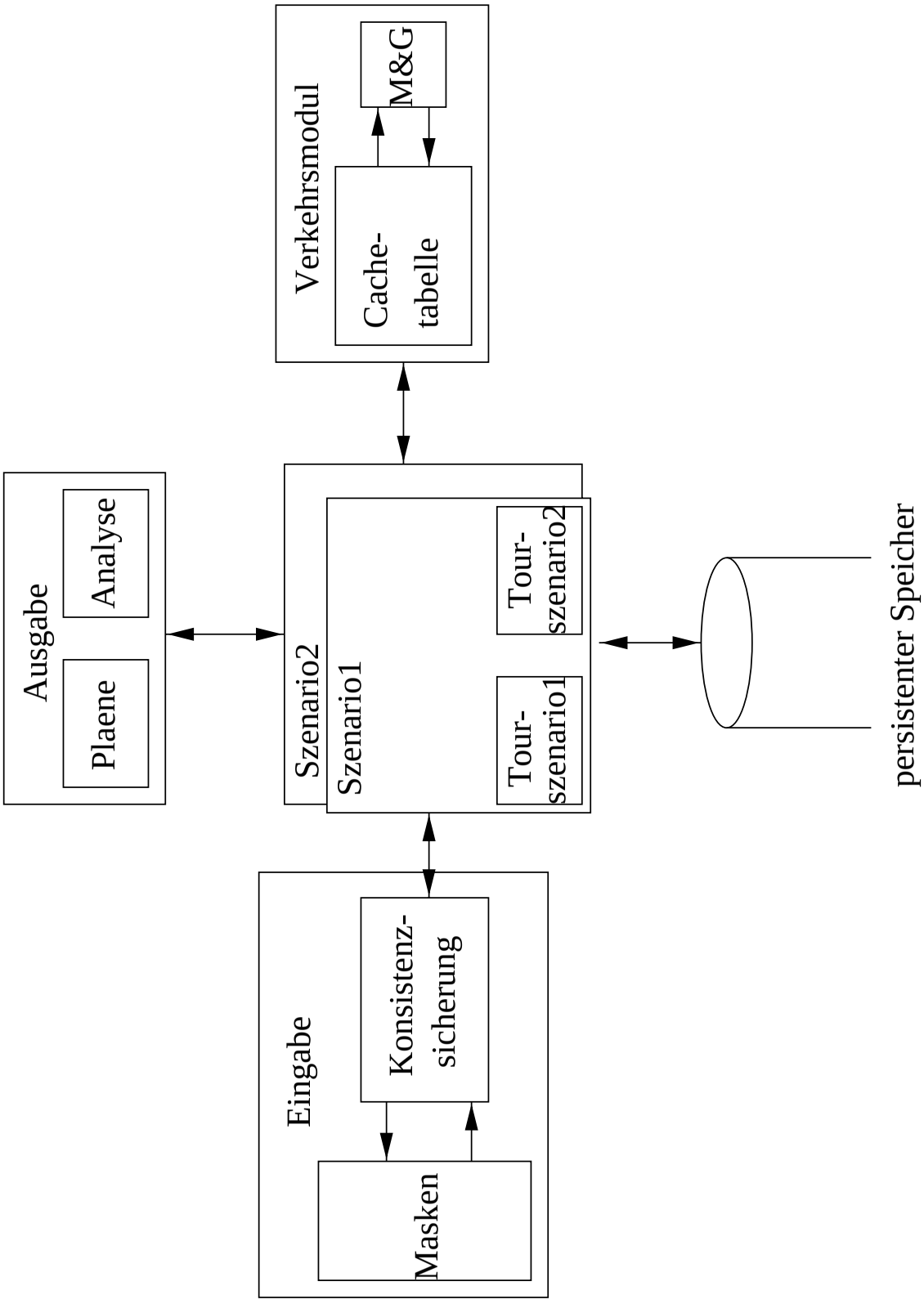


Abbildung 5.1: Architektur des System TROSS

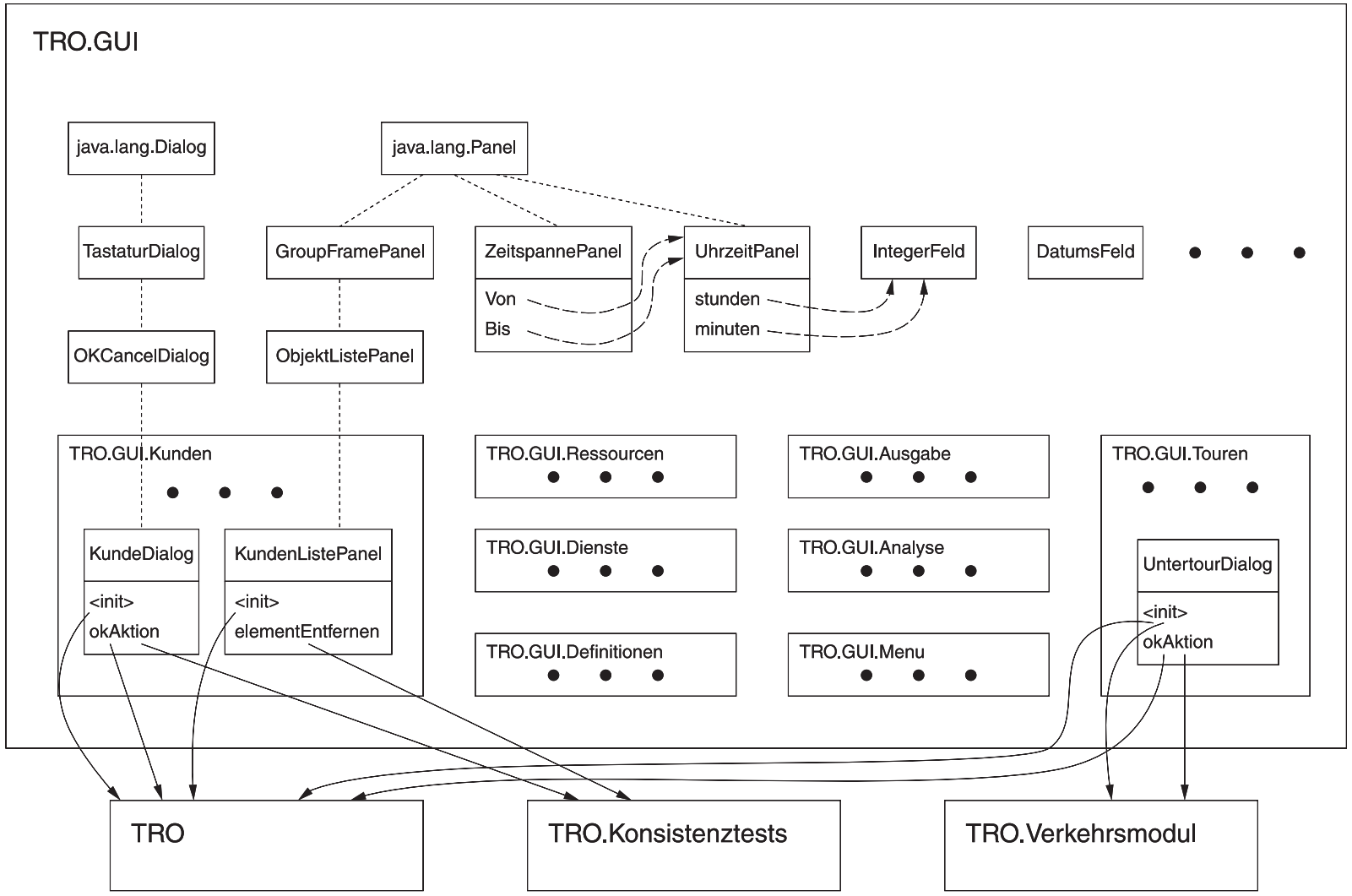


Abbildung 5.2: Architektur der graphischen Benutzungsoberfläche

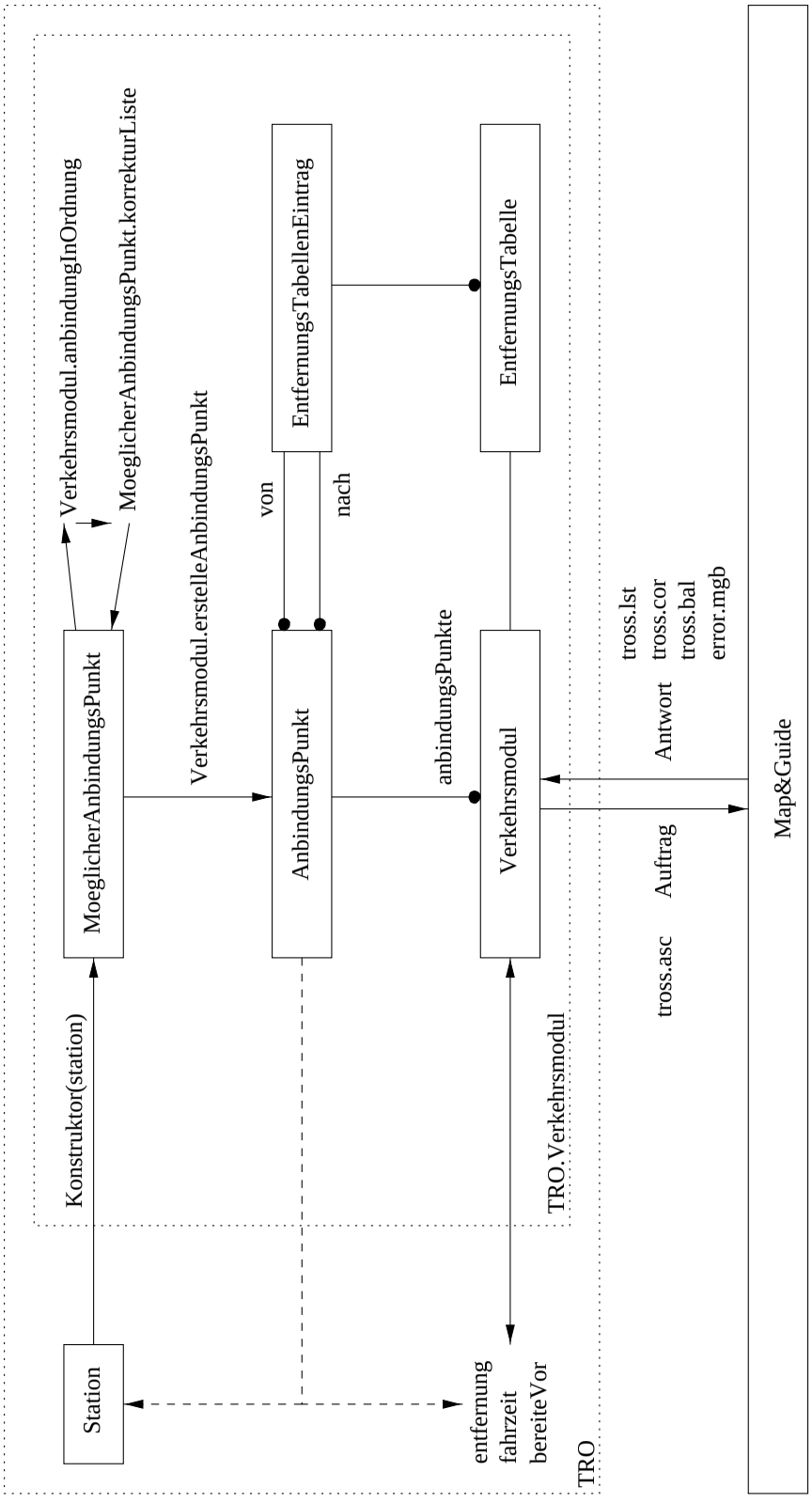


Abbildung 5.3: Prinzipskizze des Verkehrsmoduls

5.4 Klassenhierarchie

```
* class java.lang.Object
     o class TRO.Verkehrsmodul.AnbindungsPunkt (implements
       java.io.Serializable)
     o class TRO.ArbeitszeitProfil (implements java.io.Serializable,
       TRO.ListenElement)
        + class TRO.TeilzeitStundenProfil (implements
          java.io.Serializable)
        + class TRO.TeilzeitTageProfil (implements
          java.io.Serializable)
        + class TRO.VollzeitProfil (implements java.io.Serializable)
     o class TRO.Bank (implements java.io.Serializable)
     o class TRO.BankVerbindung (implements java.io.Serializable,
       TRO.ListenElement)
     o class TRO.GUI.ColumnConstraints (implements java.lang.Cloneable,
       java.io.Serializable)
     o class TRO.GUI.ColumnLayout (implements java.awt.LayoutManager2,
       java.io.Serializable)
     o interface TRO.Comparable
     o class java.awt.Component (implements java.awt.image.ImageObserver,
       java.awt.MenuContainer, java.io.Serializable)
        + class java.awt.Choice (implements java.awt.ItemSelectable)
             + class TRO.GUI.ObjektAuswahl
        + class java.awt.Container
             + class java.awt.Panel
                  + class TRO.GUI.Definitionen.ArbeitszeitPanel
                       + class
                         TRO.GUI.Definitionen.TeilzeitStundenPanel
                       + class TRO.GUI.Definitionen.TeilzeitTagePanel
                       + class TRO.GUI.Definitionen.VollzeitPanel
                  + class TRO.GUI.DatumsspannePanel
                  + class TRO.GUI.GroupFramePanel
                       + class TRO.GUI.AuswahlListePanel
                       + class TRO.GUI.HashPanel
                       + class TRO.GUI.Dienste.KonfigurationsPanel
                       + class TRO.GUI.ObjektListePanel
                            + class TRO.GUI.Dienste.BezugspersonenPanel
                              (implements TRO.GUI.ObjektListePanel.
                              ElementListener)
                            + class
                              TRO.GUI.Dienste.DienstwunschListePanel
                              (implements TRO.GUI.ObjektListePanel.
                              ElementListener)
                            + class TRO.GUI.Touren.FahrtenListePanel
                              (implements TRO.GUI.ObjektListePanel.
                              ElementListener)
                            + class
                              TRO.GUI.Ressourcen.FahrzeugListePanel
```

```
                    (implements TRO.GUI.ObjektListePanel.
                    ElementListener)
                  + class TRO.GUI.Kunden.KundenListePanel
                    (implements TRO.GUI.ObjektListePanel.
                    ElementListener)
                  + class
                    TRO.GUI.Ressourcen.MitarbeiterListePanel
                    (implements TRO.GUI.ObjektListePanel.
                    ElementListener)
                  + class TRO.GUI.Touren.StationenListePanel
                    (implements
                    java.awt.event.ActionListener,
                    TRO.GUI.ObjektListePanel.
                    InformationListener)
                  + class TRO.GUI.Dienste.StationenPanel
                    (implements TRO.GUI.ObjektListePanel.
                    ElementListener,
                    TRO.GUI.ObjektListePanel.
                    InformationListener,
                    java.awt.event.ActionListener)
                  + class TRO.GUI.Dienste.ZahlerPanel
                    (implements TRO.GUI.ObjektListePanel.
                    ElementListener,
                    TRO.GUI.ObjektListePanel.
                    InformationListener)
            + class TRO.GUI.Kunden.PersonPanel
            + class TRO.GUI.Dienste.QualifikationsPanel
            + class TRO.GUI.StationPanel (implements
              java.awt.event.ActionListener)
            + class TRO.GUI.Dienste.TerminPanel (implements
              TRO.GUI.ObjektListePanel. ElementListener)
            + class TRO.GUI.Touren.TourQualifikationsPanel
            + class TRO.GUI.Touren.TourVorliebenPanel
            + class TRO.GUI.Kunden.VorliebenPanel
            + class TRO.GUI.ZeitraumPanel (implements
              TRO.GUI.ObjektListePanel. ElementListener)
      + class TRO.GUI.Touren.TourenListePanel
      + class TRO.UeberTrossPanel
      + class TRO.GUI.UhrzeitPanel
      + class TRO.GUI.ZeitdauerPanel
      + class TRO.GUI.ZeitspannePanel
+ class java.awt.Window
      + class java.awt.Dialog
            + class TRO.GUI.AnbindungspunktWahlDialog
              (implements java.awt.event.ActionListener)
            + class
              TRO.GUI.Definitionen.ArbeitszeitArtDialog
              (implements java.awt.event.ActionListener)
            + class TRO.GUI.Dienste.DienstartDialog
              (implements java.awt.event.ActionListener)
```

```
+ class TRO.GUI.TastaturDialog (implements
  java.awt.event.KeyListener,
  java.awt.event.ContainerListener)
     + class TRO.GUI.JaNeinDialog
     + class TRO.GUI.OKCancelDialog
          + class TRO.GUI.AdresseDialog
          + class
            TRO.GUI.Definitionen.ArbeitszeitProfilDialog
          + class
            TRO.GUI.Kunden.BankverbindungsDialog
          + class
            TRO.GUI.Definitionen.BenannteStationDialog
          + class
            TRO.GUI.Kunden.BezugsPersonDialog
          + class TRO.GUI.DatumsspanneDialog
          + class TRO.GUI.DienstwunschWahlDialog
          + class TRO.GUI.ElementWahlDialog
          + class
            TRO.GUI.Definitionen.EntfernungDialog
          + class
            TRO.GUI.Definitionen.EssensartDialog
          + class TRO.GUI.Touren.FahrtDialog
            (implements
            java.awt.event.ActionListener)
          + class
            TRO.GUI.Definitionen.FahrzeugkonfigurationsDialo
            (implements TRO.GUI.HashPanel.
            HashListener)
          + class
            TRO.GUI.Definitionen.FahrzeugtypDialog
            (implements
            TRO.GUI.ObjektListePanel.
            ElementListener)
          + class
            TRO.GUI.Definitionen.FeiertagDialog
          + class
            TRO.GUI.Definitionen.HilfsmittelDialog
          + class
            TRO.GUI.Definitionen.InstitutionDialog
          + class
            TRO.GUI.Kunden.KommunikationsverbindungsDialog
          + class TRO.GUI.KonsistenzDialog
          + class
            TRO.GUI.Definitionen.QualifikationDialog
          + class TRO.GUI.RegisterDialog
               + class
                 TRO.GUI.Dienste.EssenDialog
                 (implements TRO.GUI.HashPanel.
                 HashListener)
               + class
```

```
                    TRO.GUI.Ressourcen.FahrzeugDialog
                    (implements TRO.GUI.HashPanel.
                    HashListener)
                  + class
                    TRO.GUI.Dienste.IndividualfahrtDialog
                  + class
                    TRO.GUI.Kunden.KundeDialog
                  + class TRO.GUI.Dienste.MSDDialog
                  + class
                    TRO.GUI.Ressourcen.MitarbeiterDialog
                  + class
                    TRO.GUI.Dienste.SchulfahrtDialog
              + class TRO.GUI.Dienste.RhythmusDialog
              + class
                TRO.GUI.Kunden.SachbearbeiterDialog
              + class
                TRO.GUI.Dienste.StationZeitDialog
              + class
                TRO.GUI.TeilDienstwunschWahlDialog
              + class TRO.GUI.TeilListeWahlDialog
              + class TRO.GUI.TextEingabeDialog
              + class TRO.GUI.Touren.TourDialog
                (implements
                TRO.GUI.ObjektListePanel.
                ElementListener)
              + class TRO.GUI.UhrzeitDialog
              + class TRO.GUI.Touren.UntertourDialog
              + class
                TRO.GUI.Definitionen.VerkehrstoolDialog
              + class TRO.GUI.ZahlEingabeDialog
              + class TRO.GUI.ZeitdauerDialog
        + class TRO.GUI.OKDialog
              + class
                TRO.GUI.Definitionen.ArbeitszeitProfileDialog
                (implements
                TRO.GUI.ObjektListePanel.
                ElementListener)
              + class
                TRO.GUI.Definitionen.BenannteStationenDialog
                (implements
                TRO.GUI.ObjektListePanel.
                ElementListener)
              + class
                TRO.GUI.Definitionen.EssensartenDialog
                (implements
                TRO.GUI.ObjektListePanel.
                ElementListener,
                TRO.GUI.ObjektListePanel.
                InformationListener)
              + class
```

```
                                    TRO.GUI.Touren.FahrtenListeDialog
                                  + class
                                    TRO.GUI.Definitionen.FahrzeugtypenDialog
                                    (implements
                                    TRO.GUI.ObjektListePanel.
                                    ElementListener)
                                  + class
                                    TRO.GUI.Definitionen.FeiertageDialog
                                    (implements
                                    TRO.GUI.ObjektListePanel.
                                    ElementListener)
                                  + class
                                    TRO.GUI.Definitionen.HilfsmittelListeDialog
                                    (implements
                                    TRO.GUI.ObjektListePanel.
                                    ElementListener,
                                    TRO.GUI.ObjektListePanel.
                                    InformationListener)
                                  + class
                                    TRO.GUI.Definitionen.InstitutionenDialog
                                    (implements
                                    TRO.GUI.ObjektListePanel.
                                    ElementListener)
                                  + class TRO.GUI.MeldungDialog
                                      + class
                                        TRO.GUI.VerkehrstoolLaeuftNichtDialog
                                  + class
                                    TRO.GUI.Definitionen.QualifikationenDialog
                                    (implements
                                    TRO.GUI.ObjektListePanel.
                                    ElementListener,
                                    TRO.GUI.ObjektListePanel.
                                    InformationListener)
                  + class java.awt.Frame (implements
                    java.awt.MenuContainer)
                       + class TRO.Tross (implements
                         java.awt.event.ActionListener)
     + class java.awt.List (implements java.awt.ItemSelectable)
          + class TRO.GUI.ObjektListe
     + class java.awt.TextComponent
          + class java.awt.TextField
               + class TRO.GUI.DatumsFeld
               + class TRO.GUI.IntegerFeld
o class TRO.Datum (implements java.lang.Cloneable,
  java.io.Serializable)
o class TRO.Datumsspanne (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Dienstwunsch (implements java.io.Serializable,
  TRO.ListenElement)
     + class TRO.EARDienstwunsch (implements java.io.Serializable)
```

```
      + class TRO.MSDDienstwunsch (implements java.io.Serializable)
      + class TRO.TransportDienstwunsch (implements
        java.io.Serializable)
o interface TRO.Druckbar
o class TRO.Essensart (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Fahrt (implements java.lang.Cloneable,
  java.io.Serializable, TRO.ListenElement)
o class TRO.Fahrzeug (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Fahrzeugkonfiguration (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Fahrzeugtyp (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Feiertag (implements TRO.ListenElement,
  java.io.Serializable)
o class TRO.GUI.GUIHelfer
o class TRO.Helfer
o class TRO.Hilfsmittel (implements TRO.ListenElement,
  java.io.Serializable)
o class TRO.Institution (implements java.io.Serializable,
  TRO.Zahler)
o class TRO.KommunikationsVerbindung (implements
  java.io.Serializable, TRO.ListenElement)
o class TRO.KontaktPerson (implements TRO.ListenElement)
o interface TRO.ListenElement (extends TRO.Comparable)
o class TRO.Verkehrsmodul.MoeglicherAnbindungsPunkt (implements
  TRO.ListenElement, java.io.Serializable)
o class TRO.Ort (implements java.io.Serializable)
o class TRO.Person (implements java.io.Serializable, TRO.Zahler)
      + class TRO.BezugsPerson (implements TRO.ListenElement)
           + class TRO.Sachbearbeiter
      + class TRO.Kunde (implements TRO.ListenElement,
        java.io.Serializable)
      + class TRO.Mitarbeiter (implements TRO.ListenElement,
        java.io.Serializable)
o class TRO.Qualifikation (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Rhythmus (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.SortierterVektor (implements java.lang.Cloneable,
  java.io.Serializable)
o class TRO.Station (implements java.io.Serializable,
  java.lang.Cloneable, TRO.ListenElement)
o class TRO.StationMitZeiten (implements java.io.Serializable,
  java.lang.Cloneable, TRO.ListenElement)
      + class TRO.UntertourHalt (implements java.io.Serializable,
        java.lang.Cloneable, TRO.ListenElement)
o class TRO.Strecke
o class TRO.Verkehrsmodul.Strecke (implements java.io.Serializable)
```

```
o class TRO.Szenario (implements java.io.Serializable)
o class TRO.TeilDienstwunsch (implements java.io.Serializable,
  TRO.ListenElement)
o class TRO.Termin (implements java.io.Serializable)
o class java.lang.Throwable (implements java.io.Serializable)
     + class java.lang.Exception
          + class TRO.GUI.EingabeFormatException
          + class TRO.IstMasterSzenario
          + class TRO.UnbekannteFahrzeitenException
          + class TRO.Verkehrsmodul.VerkehrstoolLaeuftNicht
          + class TRO.VerschiebungNichtErlaubtException
          + class TRO.Verkehrsmodul.WertNichtBekanntException
o class TRO.Tour (implements java.lang.Cloneable,
  java.io.Serializable, TRO.ListenElement)
o class TRO.TourSzenario (implements java.lang.Cloneable,
  java.io.Serializable, TRO.ListenElement)
o class TRO.Uhrzeit (implements java.io.Serializable,
  java.lang.Cloneable)
o class TRO.Untertour (implements java.lang.Cloneable,
  java.io.Serializable, TRO.ListenElement)
o class TRO.Verkehrsmodul.Verkehrsmodul
o interface TRO.Zahler (extends TRO.ListenElement)
o class TRO.Zeit (implements java.io.Serializable)
     + class TRO.Aufenthaltszeit (implements java.io.Serializable)
     + class TRO.Lieferzeit
     + class TRO.Transportzeit
o class TRO.Zeitraum (implements java.io.Serializable)
o class TRO.Zeitspanne (implements java.io.Serializable,
  TRO.ListenElement)
```

Kapitel 6

Erweiterungsmöglichkeiten

6.1 Mögliche Verbesserungen am Programm

- Bei der Kundeneingabe erscheint bei den dienstbezogenen Daten kein Kundenname, es wäre jedoch hilfreich zu wissen, welcher Kunde gerade bearbeitet wird. Bei der Dienstwunscheingabe wird zwar ein Kundenname angezeigt, dieser ist jedoch bei Kundenneuanlage null, null.
- Bei der Tourplanung erhält der Benutzer keinen Hinweis, wenn Mitarbeiter oder Fahrzeuge mehrfach verplant werden, obwohl das System diese Informationen schon hat.
- Die Dialogbox Dienstart wählen kann nicht abgebrochen werden.
- Vermißt wurde eine globale Zeitgrenze für Essenstouren, die in der Regel zwischen 10:45 und 12:30 erledigt werden müssen.

6.2 Hilfestellungen für den Benutzer

- Die Einführung benannter Rhythmen könnte die Eingabe von häufig auftretenden Rhythmen (z.B. Schulfahrten) erheblich vereinfachen.
- An allen Stellen, an denen aus globalen Listen (z.B. Hilfsmittel) ausgewählt werden kann, sollte auch eine Modifizierung der Liste möglich sein.
- Wird eine Rückfahrzeit angegeben, sollte das Rückfahrt-Flag automatisch gesetzt werden, anstatt die eingegebenen Rückfahrtzeiten zu ignorieren.
- Es wäre schön, wenn auch Kunden Ausnahme-Datumsspannen hätten, um alle Dienstwünsche des Kunden auf einfache Weise zu suspendieren.
- Bei der Untertourzusammenstellung wären die Funktionen alle „Hin- bzw. Rückfahrten wählen/löschen“ sehr hilfreich.
- Die Unterstützung des direkten Umsetzens (Dienstwunsch in einer Tour wählen und in eine andere Tour verschieben) wäre wünschenswert.

- Bei Dienstwünschen können nur Bezugspersonen angegeben werden, die schon beim Kunden definiert wurden. Es wäre wesentlich einfacher, auch hier die Neuanlage von Bezugspersonen zu ermöglichen.

6.3 Erweiterungsmöglichkeiten

Aufgrund der modularen Programmierweise können einzelne Teile leicht erweitert oder gar ganz ausgetauscht werden, um leistungsfähigeren Konzepten Platz zu machen.

- Weitere Konsistenztests können fast ohne Änderungen am restlichen Programm implementiert werden.
- Für eine Zusammenarbeit mit einem anderen Verkehrstool als Map&Guide (vielleicht mit einem Programm, das als Bibliothek vorliegt und direkt vom Programm aus ohne Umwege über Batchdateien aufgerufen werden kann; oder gar eine Online-Abfrage über ein geeignetes Netzwerk) muß nur das Verkehrsmodul intern angepaßt werden. Über die definierten Schnittstellen kann die Datenverwaltung wie bisher damit kommunizieren.
- Da Map&Guide beim Einsatz im System TROSS doch einige Schwächen aufzeigt, wäre es vielleicht vorteilhaft dieses durch ein *geeignetes* Verkehrstools bzw. sogar durch *nackte* Verkehrsdaten zu ersetzen.
- Einer Umsetzung der Datenverwaltung auf eine Datenbank kommt das Datenmodell durch eine gewisse Normierung (siehe Klassen Ort und Bank) entgegen. Die in der Oberfläche benutzte Vorgehensweise zum Überprüfen und Setzen von Werten eignet durch ihre Anlehnung an das Zwei-Phasen-Commit-Konzept gut für die Umsetzung auf ein Datenbanksystem.
- Aufsetzen des System TROSS auf ein DBMS: Dies könnte zum einen evtl. die schlechte Lade- und Speicher-Performance ausgleichen, und zum anderen könnte dem Benutzer die Möglichkeit geboten werden, die Analyse- und Ausgabefunktionen dynamisch zu erweitern (z.B. Heraussuchen wieviele Kunden im Plz-Bereich 7xxxx (= best. Stadteil (z.B. Stuttgart West)) wohnen).
- Da TROSS mit sehr großer Wahrscheinlichkeit auf einem Windows95-System zum Einsatz kommt, bestünde die Möglichkeit, das System anstatt in plattformunabhängigen Bytecode, in einen plattformabhängigen Microcode zu übersetzen, um so das Laufzeitverhalten zu verbessern.
- Da es sich bei Java um eine *sehr junge* Programmiersprache handelt, die sich aber trotzdem oder gerade deswegen sehr dynamisch entwickelt, wäre es wünschenswert, das System TROSS an neue Entwicklungen der Sprache Java anzupassen. Insbesondere bei den noch recht spartanischen Möglichkeiten der Benutzungsoberflächenprogrammierung der Version 1.1.
- Eine weitere Anpassungsmöglichkeit bestünde beim Einsatz der generischen Containerbibliothek *JGL* (=Java Generic Library) die Möglichkeiten zur Listenbehandlung, Sortierung u.ä. bietet.

Kapitel 7

Bedienungsanleitung

Der Aufbau dieser Bedienungsanleitung entspricht der Anordnung der Menupunkte im Programm TROSS, so daß sie vor allem zum schnellen Nachschlagen bei Unklarheiten im Umgang mit dem Programm geeignet ist.

Trotzdem sollte jeder Benutzer vor der Arbeit mit dem Programm TROSS alle Punkte durchgehen, um die Konzepte und Arbeitsweisen zu verstehen, die z.B. eine gewisse Reihenfolge mancher Bedienungsschritte erfordern.

7.1 Systemvoraussetzungen

Das Programm benötigt eine Rechnerplattform, auf der Java läuft (entwickelt und getestet wurde es unter Sun Solaris 2.3 sowie Microsoft Windows NT 4.0). Soll das Verkehrstool „Map&Guide“ benutzt werden (siehe 7.10.9), ist Microsoft Windows als Betriebssystem notwendig.

7.2 Installation

Das Programm besteht aus drei Dateien:

`Tross.jar` Diese Datei enthält alle ausführbare Teile des Programms TROSS

`Tross.gif` Das Titelbild

`Tross.ini` Hier werden die globalen Einstellungen des Programms gespeichert, z.B. die Einstellungen zum Verkehrstool.

Diese Dateien müssen in ein Verzeichnis auf der Festplatte kopiert werden. Im folgenden nehmen wir an, daß die Programmdateien im Verzeichnis `\tross` liegen, und das Java Development Kit unter `\java` zu finden ist (auf einem PC unter MS Windows).

Dann wird das Programm mit folgendem Aufruf gestartet:

```
jre -cp \java\lib\classes.zip;\tross\tross.jar TRO.Tross
```

7.3 Grundlegende Konzepte

7.3.1 Erweiterbare Listen

An vielen Stellen des Programms werden Listen angezeigt, die vom Benutzer beliebig erweitert oder verkleinert werden können. Dazu dienen die Buttons „Einfügen“, „Ändern“ und „Löschen“ rechts neben der Liste. Diese haben folgende Bedeutung:

Einfügen Öffnet einen Eingabedialog, um ein neues Element einzugeben. Schließt der Benutzer seine Eingabe mit „OK“ ab, wird das neue Element an der passenden Stelle in der Liste eingefügt.

Ändern Dieser Button hat nur dann eine Funktion, wenn ein Element der Liste durch Anklicken ausgewählt wurde. Dann wird ein Dialogfenster geöffnet, um die Daten des gewählten Elements zu ändern. Nach Bestätigung der Eingabe erscheint das Element im geänderten Zustand in der Liste.

Löschen Auch hierzu muß ein Element ausgewählt sein. Dieses wird aus der Liste gelöscht, falls keine Konsistenzbedingungen dies verbieten.

7.3.2 Eingabedialoge

Jeder Eingabedialog (egal, ob zur Erfassung eines neuen Datenobjekts oder zur Änderung eines bestehenden) hat am unteren Rand zwei Buttons „OK“ und „Abbruch“. Mit „OK“ wird die Eingabe bestätigt und alle Änderungen übernommen. Bei „Abbruch“ werden alle direkten Änderungen verworfen und das Datenobjekt bleibt im ursprünglichen Zustand. Dies gilt nicht für Änderungen an Listen (wie in 7.3.1 beschrieben), die unmittelbar nach Eingabe wirksam werden. Beispielsweise ist die Erfassung einer Kundenadresse bereits dann gespeichert, wenn die Eingabe der Adresse bestätigt wurde und diese in der Liste erscheint. Ein späteres Abbrechen des Dialogs zur Kundeneingabe hat hierauf keinen Einfluß.

Die Buttons „OK“ und „Abbruch“ können auch direkt per Tastaturkürzel aktiviert werden: <Enter> (auch als <Return> oder <Eingabe> bezeichnet) bestätigt die Eingabe und beendet den Dialog, mit <Escape> wird die Eingabe verworfen und der Dialog abgebrochen.

Dialoge für umfangreichere Eingaben sind in mehrere „Register“ unterteilt. Dazu bietet ein solcher Dialog eine waagrechte Reihe von Buttons am oberen Rand, mit denen zwischen den verschiedenen Teilen des Dialogs umgeschaltet werden kann. Trotzdem gehören alle Eingabefelder zusammen, auch wenn sie verschiedenen Registern zugeordnet sind. Der Button „OK“ bestätigt also auch hier alle Eingaben und schließt den Dialog, nicht nur die jeweilige Registerseite.

7.4 Szenario

Zur Unterstützung manueller Planung und Optimierung bietet TROSS das Konzept der *Szenarien* an. Ein Szenario besteht aus den Stammdaten (Kunden mit ihren Dienstwünschen, Ressourcen und allen Einstellungen) einerseits und einer Menge von Tourszenarien andererseits. Ein *Tourszenario* wiederum enthält alle

Tourdaten (Touren, Untertouren und Fahrten) zu den Stammdaten des Szenarios. Jedes Szenario kann in eine Datei gespeichert und wieder geladen werden.

Dadurch können Planungsansätze mit der Methode „Was wäre, wenn?“ leicht durchgeführt werden:

Sollen unterschiedliche Touren ausprobiert werden, um z.B. verschiedene Verteilungen der Fahrdienstkunden auf die Fahrzeuge zu vergleichen, werden innerhalb eines Szenarios verschiedene Tourszenarien angelegt, die verschiedene Möglichkeiten darstellen, denselben Kundenkreis mit denselben Ressourcen zu bedienen. Von den Tourszenarien eines Szenarios ist immer eines aktiv, d.h. alle Änderungen an Touren, Untertouren und Fahrten beziehen sich auf das momentan aktive Tourszenario.

Noch weitergehende Planspiele bieten verschiedene Szenarien: Hier ist z.B. ein Kapazitätsvergleich möglich, wenn nach dem Ausscheiden eines Zivildienstleistenden im einen Fall ein Nachfolger eingestellt wird, im anderen Fall mit einem Mitarbeiter weniger gearbeitet werden muß.

Um den Überblick bei vielen Szenarien und Tourszenarien zu behalten, hat jedes einen eindeutigen Namen. Die Titelzeile des Hauptfensters gibt Auskunft über das momentan geladene Szenario und dessen aktives Tourszenario. Überdies ist ein Szenario als sogenanntes „Master-Szenario“ sowie eines seiner Tourszenarien als „Master-Tourszenario“ ausgezeichnet. Diese Kennzeichnung ist für die real gültigen Daten gedacht, also die tatsächlich vorhandenen Ressourcen des Benutzers und die wirklich durchgeführten Fahrten. Erweist sich eine andere Planungsvariante als so gut, daß sie anstelle der bisherigen benutzt werden soll, kann diese jederzeit zum Masterszenario gemacht werden. Der Benutzer sollte also darauf achten, daß das Masterszenario und das Master-Tourszenario stets mit der Realität übereinstimmen (also auch regelmäßig mitgeführt werden).

7.4.1 Neu

Erstellt ein neues, leeres Szenario. Dessen Name wird vom Benutzer definiert.

7.4.2 Laden

Lädt ein gespeichertes Szenario.

7.4.3 Speichern

Speichert das aktuelle Szenario in einer Datei.

7.4.4 Szenario zum Masterszenario machen

Macht das aktuelle Szenario zum Masterszenario. Zukünftig sollten also alle Änderungen am Datenbestand an diesem Szenario durchgeführt werden.

7.4.5 Untermenu Tourszenario

7.4.5.1 Neu

Erstellt im aktuellen Szenario ein weiteres, leeres Tourszenario.

7.4.5.2 Wechseln

Wechselt das aktive Tourszenario.

7.4.5.3 Kopieren

Erstellt eine Kopie des aktiven Tourszenarios unter anderem Namen. Dadurch können, ausgehend vom momentanen Datenbestand, beliebige Veränderungen auf der Kopie ausprobiert werden, ohne das ursprüngliche Tourszenario zu verändern.

7.4.5.4 Löschen

Löscht ein Tourszenario mit allen seinen Daten aus dem Szenario.

7.4.5.5 Umbenennen

Gibt einem Tourszenario einen anderen Namen.

7.4.5.6 Zum Master-Tourszenario machen

Macht das aktive Tourszenario zum Master-Tourszenario, falls das momentan geladene Szenario das Master-Szenario ist.

7.4.6 Programm beenden

Beendet das Programm. Wurde das Szenario seit der letzten Änderung nicht gespeichert, erfolgt eine Sicherheitsabfrage.

7.5 Kunden

7.5.1 Kundenliste

Zeigt eine änderbare Liste mit allen Kunden des Szenarios an. Diese sind alphabetisch nach Nachnamen sortiert.

7.5.1.1 Eingabedialog für Kunden

Persönliche Daten Hier werden folgende Daten erfaßt:

Kundennummer Jeder Kunden braucht zur eindeutigen Identifikation eine Kundennummer, da Namen mehrfach vorkommen können.

Maximale Fahrdauer Zeitdauer, die der Kunde bei einer Fahrt maximal im Fahrzeug verbringen darf. Falls hier ein Wert angegeben wird, hat dieser Vorrang vor dem global definierten. Fehlt eine Angabe, gilt die globale Obergrenze (siehe 7.10.11).

Anzahl Schlüssel Anzahl der Hausschlüssel des Kunden, die dieser zur Durchführung von Diensten zur Verfügung gestellt hat.

Bemerkungen (Schlüssel) Beliebige Bemerkungen zu den Hausschlüsseln. Hier könnte z.B. die Nummer stehen, unter der dieser Hausschlüssel im Schlüsselschrank zu finden ist.

Allgemeine Personendaten

Name Der Nachname der Person. Dieser Wert muß immer angegeben werden.

Vorname Der Vorname (optional).

Geburtsdatum Das Geburtsdatum (optional).

Adressen Jeder Person können mehrere Adressen zugeordnet werden. Die erste dieser Adressen gilt als Wohnort des Kunden, der als Vorgabewert für Dienstwünsche benutzt wird (z.B. Abholadresse für Fahrten).

Kommunikationsverbindungen Eine Kommunikationsverbindung besteht aus der Art (z.B. „Telephon“) und dem Eintrag (z.B. „0815/4711“);

Bankverbindungen Bankverbindungen bestehen aus Bankname, Bankleitzahl, Kontonummer und der Angabe, ob für dieses Konto eine Einzugsermächtigung vorliegt.

Dienstbezogene Daten

Vorlieben und Abneigungen Definiert Mitarbeiter, die der Kunde gerne für seine Dienstwünsche eingesetzt hätte (dies ist eine Soll-Vorgabe) und solche, die auf keinen Fall bei dem Kunden eingesetzt werden dürfen (dies ist eine Muß-Vorgabe). Der Button „Ändern“ unter der jeweiligen Liste öffnet ein Fenster, worin diese Liste verändert werden kann, indem Mitarbeiter entfernt oder weitere aus der Gesamtliste eingefügt werden.

Benötigte Hilfsmittel Hier können aus der Liste der definierten Hilfsmittel (siehe 7.10.6) diejenigen ausgewählt werden, die der Kunde benötigt. Hiermit kann z.B. ein Abgleich zwischen geforderten Kindersitzen und tatsächlich im Fahrzeug vorhandenen stattfinden. Das Programm macht dies jedoch nicht.

Zulässige Fahrzeuge Darf ein Kunde nur mit bestimmten Fahrzeugen befördert werden, können diese hier ausgewählt werden. Ist die Liste leer, kommt prinzipiell jedes Fahrzeug in Frage.

Bezugspersonen Hier können beliebige Personen erfaßt werden, deren Daten für die Ausführung von Diensten bei diesem Kunden relevant sind, z.B. Eltern eines behinderten Kindes, der Hausarzt etc. Zusätzlich zu den allgemeinen Personendaten (siehe 7.5.1.1) sind diese durch die Art des Bezuges charakterisiert (hier könnte „Hausarzt“ oder „Mutter“ eingetragen werden). Ein gewisser Sonderfall sind Sachbearbeiter einer Institution, z.B. der Krankenkasse des Kunden. Diese haben als Bezug generell „Sachbearbeiter“, sind aber zur näheren Identifikation einer Institution zugeordnet (zur Eingabe von Institutionen siehe 7.10.4).

Dienstwünsche Hier werden die Dienstwünsche des Kunden erfaßt. Diese stehen nach Dienstart sortiert in der Liste.

7.5.2 Eingabedialoge für Dienstwünsche

Zur Eingabe eines neuen Dienstwunsches muß zunächst die zugehörige Dienstart gewählt werden, bevor der passende Dienstwunschdialog geöffnet wird. Die Dienstwunschdialoge sind zweigeteilt:

Anforderungen Hier können verschiedene dienstwunschspezifische Anforderungen des Kunden erfaßt werden, wie z.B. Art des benötigten Sitzplatzes oder die Art des zu liefernden Essens.

Allen Dienstwünschen gemeinsam ist eine Liste von Rechnungsempfängern für den Dienst (dies können der Kunde selbst, Bezugspersonen oder Institutionen sein; eine der angegebenen Bankverbindungen muß für diesen Dienstwunsch ausgewählt werden) sowie eine Liste mit Bezugspersonen, deren Daten mit auf dem Tourplan abgedurckt werden sollen (dies könnte z.B. der Hausarzt von anfallsgefährdeten Kunden sein).

Ort und Zeit Erfaßt einen oder mehrere Orte (je nach Dienstart) sowie den Termin des Dienstwunsches. Ein *Termin* enthält einen Zeitraum sowie eine Menge von Rhythmen.

Der *Zeitraum* gibt den Datumsbereich an, innerhalb dessen der Dienst ausgeführt werden soll. Angegeben werden kann ein Rahmen (z.B. ein Schuljahr) sowie eine Menge von Ausnahmezeiten (z.B. Ferien), während derer der Dienst nicht erwünscht ist. Ist der Rahmen leer, hat dies die Bedeutung „Immer“.

Ein *Rhythmus* gibt an, wie oft und regelmäßig ein Dienst stattfinden soll. Die gewünschten Wochentage können bestimmt werden, ebenso, ob der Dienst nur an Werktagen stattfinden soll (also ausfällt, falls der angegebene Wochentag ein Feiertag ist). Ist das Kästchen „Nachricht an Feiertagen“ angekreuzt, erscheint beim Laden des Szenarios eine Warnmeldung, falls der Dienst wegen eines Feiertages ausfallen würde. Dadurch kann der Dienst in der kritischen Woche rechtzeitig vorher auf einen anderen Tag verlegt werden. Für seltener stattfindende Dienste kann angegeben werden, alle wieviel Wochen die Ausführung erwünscht ist.

Außerdem enthält ein Rhythmus die Uhrzeit, zu der ein Dienst an den angegebenen Tagen stattfinden soll. Die hierfür benötigten Angaben sind von der Dienstart abhängig und werden in Zusammenhang mit den dienstspezifischen Anforderungen erläutert.

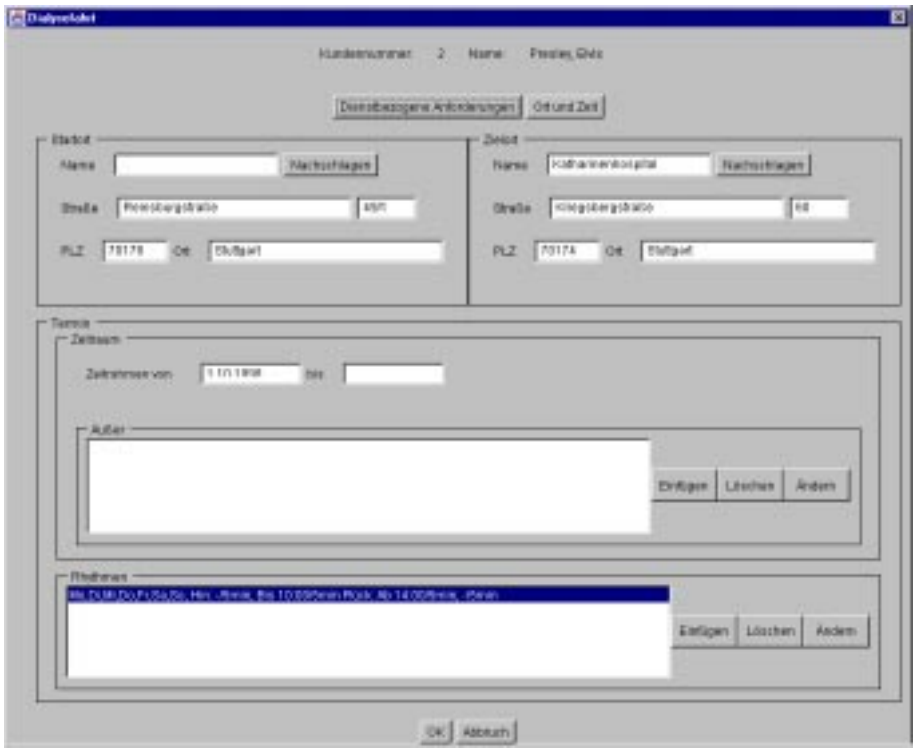


Abbildung 7.1: Eingabedialog für Ort und Zeit

Schulfahrten, Dialysefahrten und Fahrten zur Tagespflege Hier können folgende Anforderungen erfaßt werden:

- Die Art des benötigten Platzes im Fahrzeug: Sitzplatz, Rollstuhlplatz oder Sitzplatz mit fest montiertem Hilfsmittel einer bestimmten Art
- Die Forderung, daß ein zweiter Mitarbeiter bei der Aufführung des Dienstes dabei sein muß
- die erforderlichen Qualifikationen des oder der Mitarbeiter
- Die Vorlieben und Abneigungen des Kunden sind hier zur Information nochmals angezeigt, eine Änderung wirkt sich aber stets auf die Kundendaten aus, gilt also für alle Dienstwünsche.

Für die Fahrt muß ein Startort angegeben werden, an dem der Kunde abgeholt werden soll, und ein Zielort, zu dem der Kunde gebracht wird.

Die Angaben zur Zeit sind in Daten für die Hin- und Rückfahrt eingeteilt. Die Zeiten für die Rückfahrt kommen allerdings nur dann zur Geltung, wenn die Checkbox „Rückfahrt“ angekreuzt ist. Für jede dieser Teilfahrten können Abholzeit und Ankunftszeit vorgegeben werden, sowie die benötigten Zeitdauern zum Ein- und Aussteigen in das befördernde Fahrzeug. Die Zeiten sind jeweils als Zeitspanne definiert, so daß eine gewünschte Toleranz explizit angegeben

werden kann. Von den vier möglichen Uhrzeitangaben muß nur eine angegeben werden, z.B. bedeutet der Eintrag „8:00 Uhr“ bei „Ankunft bis“, daß der Kunde spätestens um 8 Uhr am Zielort ankommen muß. Die Abholzeit kann abhängig hiervon beliebig gewählt werden.

Individualfahrten Für Individualfahrten können dieselben Anforderungen erfaßt werden wie für Schulfahrten. Zusätzlich kann gefordert werden, daß die Mitarbeiter zwischen Hin- und Rückfahrt am Zielort bleiben, um z.B. mit einem behinderten Kunden ein Konzert zu besuchen. Außerdem können beliebige Zwischenstationen mit Aufenthaltsdauern angegeben werden, die auf der Fahrt zum Ziel angefahren werden sollen.

Essen auf Rädern Hier werden unter Anforderungen die Anzahl der bestellten Essen erfaßt. Zu jeder vorrätigen Essensart (Eingabe siehe 7.10.7) kann eine Anzahl zu liefernder Essen angegeben werden.

Als Ort wird die Adresse erfaßt, an die die Essen gebracht werden sollen. Die Uhrzeit besteht aus Zeitspanne, innerhalb derer das Essen geliefert werden muß, sowie einer Dauer für das Abgeben des Essens.

MSD / APD Die Anforderungen umfassen neben den Ansprüchen an Mitarbeiter (Qualifikationen und Vorlieben/Abneigungen) die Angabe, ob der Kunde bei der Verrichtung des Dienstes im Fahrzeug mitfahren soll (z.B. zum Einkaufen). Ist dies angekreuzt, kann die Art des erforderlichen Platzes angegeben werden.

Es muß ein Dienstort angegeben werden, an dem der Dienst stattfindet, sowie die Uhrzeit (wieder als Zeitspanne) des Beginns und die Dauer des Dienstes.

7.5.3 Dienstwünsche erfüllt?

Ermöglicht das gezielte Überprüfen einer Menge von Dienstwünschen. Der Benutzer wählt eine Menge von Dienstwünschen, die mit den bestehenden Touren und Untertouren abgeglichen werden. Sämtliche Teildienstwünsche (siehe 7.6.1.2), die nicht durch das aktive Tourszenario erfüllt werden, werden dem Benutzer gemeldet. Daraus kann z.B. ganz gezielt erkannt werden, daß für einen Dienstwunsch noch eine Rückfahrt am Freitag fehlt.

7.6 Touren

Zur Beschreibung der geplanten und tatsächlich stattfindenden Fahrten, auf denen die diversen Dienstwünsche der Kunden erfüllt werden, benutzt das Programm TROSS folgende Bezeichnungen:

Tour Eine Tour besteht aus einer Menge von Dienstwünschen, die nacheinander von denselben Mitarbeitern mit demselben Fahrzeug erfüllt werden. Bei Fahrdiensten entspricht dies also einer „Fahrgemeinschaft“ von Kunden. Die Angabe von Mitarbeitern und Fahrzeug ist optional, eine Tour beschreibt lediglich die Planung, d.h. sie hat kein Datum, sondern beschreibt ein regelmäßig wiederkehrendes Ereignis.

Untertour Jede Tour besteht aus einer Menge von Untertouren, die diese weiter konkretisieren. Eine Untertour findet an gewissen Wochentagen mit einem gewissen Rhythmus statt und kann entweder eine Hinfahrt oder eine Rückfahrt der Kunden zu einem Ziel sein. In der Untertour werden die verschiedenen anzufahrenden Stationen in eine Reihenfolge gebracht. Auch die Untertouren beschreiben die Planung.

Fahrt Eine Fahrt ist eine an einem konkreten Datum stattfindende Untertour. Gegenüber Tour und Untertour kann sie gewisse Abweichungen aufweisen, die angeben, wie die tatsächliche Dienstausführung von der Planung abwich.

Da die Fahrten die tatsächliche Durchführung von Diensten abbilden, können sie die Grundlage einer Buchführung über diese Dienstleistungen bilden. Dazu kann jede Fahrt, falls nötig, gegenüber der geplanten Untertour verändert werden, um z.B. festzuhalten, daß ein anderer Mitarbeiter den Dienst erledigt hat oder einer der Kunden an diesem Tag nicht mitfuhr.

In den Eingabedialogen für Touren und Untertouren sind noch verschiedene Eingabefelder zum Sperren einzelner Dienstwünsche sowie ganzer Touren für die Optimierung vorhanden. Da diese automatische Optimierung von der Projektgruppe TRO (Wintersemester 1997/98 und Sommersemester 1998) aus Zeitgründen nicht mehr implementiert werden konnte, haben diese Eingaben keinerlei Wirkung. Sie wurden jedoch im Programm belassen, um für eine mögliche spätere Erweiterung zur Verfügung zu stehen.

7.6.1 Tourenliste

Zeigt eine änderbare Liste mit allen Touren und Untertouren des aktiven Tourszenarios an. Nach Auswahl einer Tour durch Mausklick in der linken Liste werden in der rechten Liste alle Untertouren dieser Tour angezeigt. Sowohl Touren als auch Untertouren können beliebig neu angelegt, verändert, sowie gelöscht werden. Beim Verändern von Touren/Untertouren muß allerdings Rücksicht auf die zugehörigen Fahrten genommen werden (siehe 7.6.4).

7.6.1.1 Eingabedialog für Touren

Jede Tour wird durch eine Nummer eindeutig gekennzeichnet. Zusätzlich kann eine Bezeichnung angegeben werden, um die Touren für den Benutzer besser identifizierbar zu machen.

Jeder Tour können ein oder zwei Mitarbeiter sowie ein Fahrzeug zugewiesen werden, die diese Tour normalerweise fahren. Es ist sinnvoll, diese Einträge zu machen, da sonst alle Fahrten von Hand bearbeitet werden müssen (da für sie diese Angaben obligatorisch sind) und außerdem die Analysefunktionen sonst keine sinnvollen Werte liefern.

In der Liste „Dienstwünsche“ werden alle Dienstwünsche angezeigt, die auf der Tour erfüllt werden (damit enthält die Liste alle Kunden, die diese Tour bedient). Der Button „Einfügen“ öffnet ein Dialogfenster zur Auswahl eines Dienstwunsches: Zunächst muß in der linken Liste ein Kunde ausgewählt werden, draufhin erscheinen in der rechten Liste dessen Dienstwünsche, von denen einer gewählt werden kann.

Im unteren Teil des Tourdialoges werden die vereinigten Anforderungen aller Dienstwünsche der Tour angezeigt. Diese Angaben dienen nur der Information und Übersicht und sollten daher vom Benutzer nicht verändert werden.

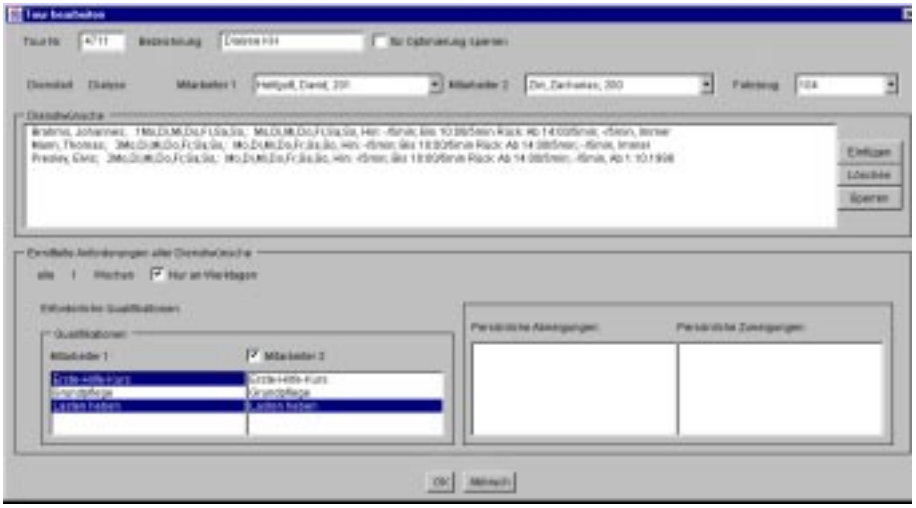


Abbildung 7.2: Eingabedialog für Touren

7.6.1.2 Eingabedialog für Untertouren

Da ein Dienstwunsch ein relativ komplexes Gebilde mit verschiedenen Zeiten an verschiedenen Wochentagen sein kann, wurde für die Untertour der Begriff des Teildienstwunsches eingeführt. Ein *Teildienstwunsch* ist ein Dienstwunsch, eingeschränkt auf einen Wochentag und entweder Hin- oder Rückfahrt. Demnach besteht jeder Dienstwunsch aus einem oder mehreren Teildienstwünschen. Genauso, wie eine Tour aus verschiedenen Dienstwünschen besteht, beinhaltet eine Untertour eine Menge von Teildienstwünschen.

Um eine neue Untertour zu definieren, können deshalb zunächst Teildienstwünsche gewählt werden, die die Untertour erfüllen soll.

Die Untertour wird durch einen alphanumerischen Zusatz zur Tournummer gekennzeichnet. Diese Untertournummer kann Werte von „a“ bis „z“ und „aa“ bis „zz“ annehmen (für neu erstellte Untertouren wird jeweils ein passender Wert vorgeschlagen). Zur leichteren Unterscheidbarkeit kann jeder Untertour eine Bezeichnung gegeben werden.

Dienstart, Mitarbeiter und Fahrzeug der Tour werden zur Information hier nochmals angezeigt.

Der wesentliche Teil der Untertour besteht in den diversen Stationen, die bei ihrer Ausführung angefahren werden sollen. Dies sind neben den Stationen, an denen Kunden abgeholt, beliefert oder anderweitig bedient werden, eine unabhängige Anfangs- und Endstation mit zugehörigen Zeitangaben. Die Untertour beginnt an der Anfangsstation zur angegebenen Zeit (dies kann die Einsatzzentrale sein, oder auch die Endstation einer unmittelbar vorausgegangenen Untertour) und endet an der Endstation zur angegebenen Zeit. Anfangs- und

Endzeit werden automatisch angepaßt, falls sie im Widerspruch zu den Zeiten der Stationenliste stehen.

Diese Stationenliste enthält alle Untertourhalte, die aufgrund der in der Untertour enthaltenen (Teil-)Dienstwünsche angefahren werden sollen. Ein *Untertourhalt* enthält neben der Station, an der der Halt stattfindet, die Ankunfts- und Wiederabfahrtszeit. Da vor allem bei Schulfahrten die Dienstwünsche vieler Kunden dasselbe Ziel haben (das aber nur einmal in der Liste auftauchen soll), sind jedem Halt auch die Dienstwünsche zugeordnet, die diese Station fordern.

Die Liste mit den Untertourhalten läßt sich mit den rechts daneben angeordneten Buttons auf vielfältige Weise verändern (die Operationen „auf“ und „ab“ beziehen sich hierbei immer auf den ausgewählten Halt):

Dienstwunsch Auf Verschiebt einen in einem Untertourhalt enthaltenen Dienstwunsch eine Position nach oben. Dadurch entsteht ein neuer Halt mit diesem einen Dienstwunsch an der zugehörigen Station.

Dienstwunsch Ab Verschiebt einen Dienstwunsch aus dem gewählten Halt auf dieselbe Art nach unten.

Halt Auf Verschiebt einen Halt mit allen enthaltenen Dienstwünschen eine Position nach oben.

Halt Ab Verschiebt einen Halt mit allen enthaltenen Dienstwünschen eine Position nach unten.

Fahrzeit ändern Ändert die Fahrzeit zwischen der Station des gewählten Haltes und der des vorausgehenden. Ist der erste Halt in der Liste gewählt, kann die dortige Ankunftszeit gesetzt werden.

Aufenthaltszeit ändern Ändert die Aufenthaltszeit an einem Halt. Standardmäßig entspricht die Aufenthaltszeit der im Dienstwunsch geforderten (z.B. Einsteigezeit der Kunden, die dort abgeholt werden), so daß Korrekturen zunächst einmal dort gemacht werden sollten. Trotzdem kann es im Einzelfall sinnvoll sein, hier die gesamte Aufenthaltszeit explizit anzugeben.

Dienstwunsch komplett entfernen Entfernt einen Dienstwunsch mit allen Teildienstwünschen aus der Untertour. Dadurch können auch einzelne Halte verschwinden.

Teildienstwunsch einfügen Fügt einen weiteren Teildienstwunsch zur Untertour hinzu und hängt die neuen Halte an die Liste an.

Teildienstwunsch entfernen Entfernt einen Teildienstwunsch und gegebenenfalls die zugehörigen Halte aus der Liste.

Verschieben von Dienstwünschen und Halten in der Liste ist nur dann möglich, wenn die Reihenfolge der Stationen innerhalb der einzelnen Dienstwünsche gewahrt bleibt. Da es z.B. wenig Sinn hat, den Zielort einer Dialysefahrt zu erreichen, bevor man alle zu transportierenden Kunden abgeholt hat, wird in einem solchen Fall mit einer Warnmeldung das Verschieben verweigert.

Nach Änderungen an einer Untertour werden alle zugehörigen Fahrten gelöscht, deren Datum in der Zukunft liegt. Ältere, die also bereits erfolgt sind,

bleiben bestehen, können sich aber möglicherweise auch verändern. Daher sollten vor Änderungen gegebenenfalls erst die alten Fahrten archiviert werden (siehe 7.6.4).

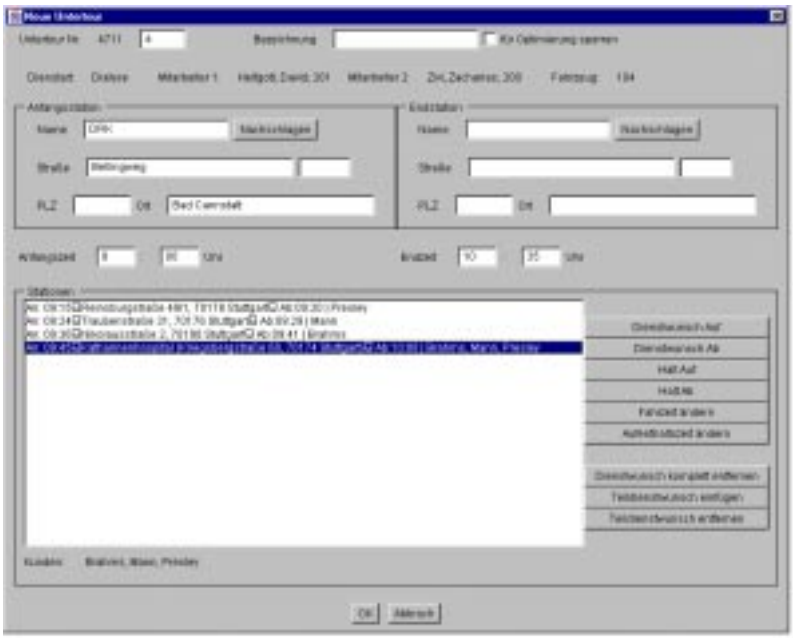


Abbildung 7.3: Eingabedialog für Untertouren

7.6.2 Fahrtenliste

Zeigt alle Fahrten des aktiven Tourszenarios an. Diese werden zunächst nach Tour- und Untertournummer, dann nach Datum sortiert. Ein „*“ vor der Fahrt zeigt an, daß diese gegenüber den Planungsdaten in Tour und Untertour verändert wurde (indem Mitarbeiter bzw. Fahrzeug geändert oder Dienstwünsche entfernt wurden). Diese können einzeln geändert werden, um Abweichungen der tatsächlich durchgeführten Fahrten von den geplanten Untertouren zu dokumentieren.

7.6.2.1 Dialog zur Änderung von Fahrten

Der Dialog für Fahrten zeigt Tournummer, Datum und Dienstart zur Information mit an.

Mitarbeiter und Fahrzeuge sind bei neu erzeugten Fahrten zunächst auf die Werte der Tour gesetzt (ist dort nichts angegeben, wird automatisch der erste mögliche Eintrag in der Liste gewählt), können aber geändert werden. Ebenso können die Anfangs- und Endstation mit den zugehörigen Zeiten abweichend von der Untertour eingetragen werden.

Die anzufahrenden Stationen werden in einer Liste angezeigt. Werden bei einer Fahrt einzelne Dienstwünsche der Untertour nicht erfüllt, können diese mit dem Button „Dienstwunsch entfernen“ aus der Fahrt entfernt werden (mit „Dienstwunsch einfügen“ kann dies wieder rückgängig gemacht werden). Die Liste wird dann dementsprechend verändert, so daß nur diejenigen Stationen angezeigt werden, die sich auf Dienstwünsche beziehen, die der Fahrt noch zugeordnet sind.

Die Zeitangaben in der Liste sind die in der Untertour ermittelten. Beim Löschen von Dienstwünschen aus einer Fahrt werden sie nicht angepaßt (eine Erfassung dieser Zeiten wäre sicher auch nicht sinnvoll, da sonst alle „Etappenzeiten“ von real durchgeführten Fahrten notiert und im Programm gegebenenfalls angepaßt werden müßten).

7.6.3 Fahrten erzeugen

Erzeugt aus den Untertouren Fahrten für einen vom Benutzer angegebenen Zeitraum. Bereits bestehende Fahrten werden dadurch nicht beeinflußt. Die neu erzeugten Fahrten werden zum Durchsehen und Bearbeiten aufgelistet.

7.6.4 Fahrten archivieren

Um erfolgte Fahrten archivieren zu können, bietet TROSS die Möglichkeit, diese in einem tabellarischen Format abzuspeichern (siehe 2.2.2.4). Die entstehende Datei kann in eine Textverarbeitung, Tabellenkalkulation oder Datenbank geladen und den Bedürfnissen des Anwenders gemäß aufbereitet werden. Nach dem Speichern werden die archivierten Fahrten gelöscht.

Achtung: TROSS bietet kein Konzept zur Verwaltung der Historie von Touren, Untertouren und Fahrten. Fahrten im Tourszenario beziehen sich direkt auf den status quo der zugeordneten Untertour, so daß sich Änderungen an einer Tour oder Untertour direkt auf die Fahrten auswirken (z.B. Entfernen eines Dienstwunsches).

Vor Änderungen an einer Tour oder Untertour des Master-Tourszenarios ist es daher sinnvoll, die bereits erfolgten Fahrten (die also vor dem jeweiligen Tagesdatum liegen) zu archivieren.

7.6.5 Konsistenzprüfung

Überprüft alle Touren, Untertouren und Fahrten des aktiven Tourszenarios auf ihre Konsistenz mit den Vorgaben der Dienstwünsche. Verstöße gegen diese Regeln werden dem Benutzer in einer Liste angezeigt.

Folgende Kriterien werden abgeprüft:

Mitarbeiter

- Sind erster und zweiter Mitarbeiter unterschiedlich?
- Ist ein zweiter Mitarbeiter eingeteilt, falls die Dienstwünsche dies verlangen?
- Sind die Mitarbeiter am Tag einer Fahrt verfügbar?
- Sind die Mitarbeiter für den Dienst an den Kunden der Tour zulässig (werden also von diesen Kunden nicht abgelehnt)?

- Haben die Mitarbeiter alle geforderten Qualifikationen?

Fahrzeuge

- Ist das Fahrzeug am Tag einer Fahrt verfügbar?
- Wir die maximale Sitzplatzzahl eingehalten (bei Fahrdiensten)?
- Ist das Fahrzeug für alle Kunden als zulässig eingetragen (bei Fahrdiensten oder MSD+Kunde fährt mit)?

Zeiten

- Entsprechen die Zeiten der Untertour den Zeitvorgaben der Dienstwünsche?

7.7 Ressourcen

Hier werden die wichtigsten Ressourcen verwaltet, die zur Verfügung stehen, um Dienstwünsche der Kunden zu erfüllen: Mitarbeiter und Fahrzeuge.

7.7.1 Mitarbeiter

Zeigt eine änderbare Liste aller Mitarbeiter an.

7.7.1.1 Eingabedialog für Mitarbeiter

Neben den allgemeinen Personendaten (siehe 7.5.1.1) hat jeder Mitarbeiter eine Personalnummer, die ihn eindeutig kennzeichnet. Diese kann aus beliebigen Zeichen bestehen (ist also nicht nur auf Ziffern beschränkt).

Ein Dienstantrittsdatum sowie ein Entlassungsdatum kann hier erfaßt werden (was insbesondere bei Zivildienstleistenden eine wichtige Information für die Einsatzplanung ist). Durch Auswahl eines Arbeitszeitprofils (siehe 7.10.2) wird festgelegt, zu welchen Zeiten bzw. wie lang ein Mitarbeiter im Dienst ist. Diese Information wird zur Darstellung der Mitarbeiterauslastung benötigt.

Die Verfügbarkeit eines Mitarbeiters wird durch einen Zeitrahmen (z.B. das laufende Kalenderjahr oder den Rest der Dienstzeit) sowie durch Ausnahmezeiten gekennzeichnet (z.B. Krankheit, Urlaub oder Schulung). Werden bekannte Ausnahmezeiten rechtzeitig vorher hier eingetragen (z.B. Urlaub), wird dies bei der Konsistenzüberprüfung beachtet und gegebenenfalls gewarnt, falls ein momentan nicht verfügbarer Mitarbeiter für Fahrten eingeteilt ist.

Aus den definierten Qualifikationen (siehe 7.10.1) können durch Auswahl in der Liste diejenigen bestimmt werden, die der Mitarbeiter besitzt. Diese Information ist wichtig, um die Fähigkeiten von Mitarbeitern mit den Anforderungen der Kunden in ihren Dienstwünschen abgleichen zu können.

7.7.2 Fahrzeuge

Zeigt eine änderbare Liste aller Fahrzeuge an.

7.7.2.1 Eingabedialog für Fahrzeuge

Ein Fahrzeug wird durch Angabe von Modell, amtlichem Kennzeichen sowie einer internen Nummer identifiziert (im Programm TROSS dient die interne Nummer als Unterscheidungsmerkmal, z.B. in der Liste aller Fahrzeuge).

Drei wichtige Termine können für jedes Fahrzeug erfaßt werden: Die nächste Hauptuntersuchung beim TÜV, die nächste ASU, sowie der nächste Kundendienst. Beim Laden eines Szenarios werden diese Werte mit dem Tagesdatum verglichen und dem Benutzer gemeldet, falls sie innerhalb der nächsten zwei Wochen fällig werden.

Die Austattung eines Fahrzeugs wird durch die Eingabefelder „spezielle Daten“ definiert: Für jedes definierte Hilfsmittel kann die Anzahl der davon im Fahrzeug mitgeführten Exemplare angegeben werden (gemeint sind hier nur lose Hilfsmittel wie z.B. spezielle Sitzkissen, die einfach auf einem Sitzplatz angebracht und wieder von dort entfernt werden können; fest montiere Hilfsmittel werden in der Fahrzeugkonfiguration erfaßt).

Durch Auswahl eines Fahrzeugtyps und einer diesem Typ zugeordneten Fahrzeugkonfiguration wird der mögliche sowie momentane „Umbauzustand“ des Fahrzeugs beschrieben (siehe hierzu auch 7.10.3 und 7.10.3.2).

7.8 Ausgabe

7.8.1 Tourplan

Zeigt eine Tour mit allen Untertouren schematisch an. Hieraus können sowohl der Benutzer als auch die ausführenden Mitarbeiter erkennen, welcher Dienst wann erledigt werden soll.

7.8.2 Dienstplan

Datengrundlage für Dienstpläne sind die Fahrten eines Zeitraum, den der Benutzer vorgibt. Dadurch werden einzelne Änderungen an Fahrten (z.B. kurzzeitiger Austausch eines Mitarbeiters wegen Krankheit) in die Pläne übernommen.

Der Plan zeigt eine graphische Übersicht über den gewünschten Zeitraum. Darin werden alle Fahrten, die der gewählte Mitarbeiter ausführen soll, als Balken dargestellt.

7.8.3 Gesamtdienstplan

7.8.4 Angezeigten Plan drucken

Gibt den gerade im Hauptfenster angezeigten Plan auf den Drucker aus.

7.8.5 Untermenu Pläne drucken

7.8.5.1 Dienstpläne

Druckt eine Menge von Dienstplänen, ohne diese einzeln auf dem Bildschirm anzuzeigen. Der Benutzer kann eine beliebige Teilmenge aller Mitarbeiter angeben, für die die Pläne gedruckt werden sollen.

Abbildung 7.4: Beispiel für einen Tourplan

7.8.5.2 Tourpläne

Druckt eine Menge von Tourplänen, ohne diese einzeln anzuzeigen. Der Benutzer wählt die Touren aus.

7.9 Analyse

Alle Analysedaten basieren auf den Plandaten, also Touren und Untertouren.

7.9.1 Auslastung Mitarbeiter

Zeigt die Zeit, die die Mitarbeiter auf Touren beschäftigt sind, im Verhältnis zur Sollarbeitszeit, die durch die Arbeitsprofile vorgegeben wird.

7.9.2 Ausfallzeiten

7.9.2.1 Ausfallzeiten Mitarbeiter

Zeigt die Anzahl der Tage, die die Mitarbeiter für einen gegebenen Zeitraum ausfällt.

7.9.2.2 Ausfallzeiten Fahrzeuge

Zeigt die Anzahl der Tage, die die Fahrzeuge für einen gegebenen Zeitraum ausfällt.

7.9.3 Fahrzeugbesetzung

Zeigt die Anzahl der Passagiere pro Fahrzeug im Verhältnis zur maximalen Sitzplatzzahl (im Fahrzeugtyp definiert).

7.9.4 Auslastung Mitarbeiter drucken

Druckt die Auslastungsgraphik für alle Mitarbeiter, mit automatischem Seitenumbruch.

7.9.5 Auslastung Fahrzeuge drucken

Druckt die Auslastungsgraphik für alle Fahrzeuge, mit automatischem Seitenumbruch.

7.9.6 Ausfall Mitarbeiter drucken

Druckt die Ausfallzeiten für alle Mitarbeiter, mit automatischem Seitenumbruch.

7.9.7 Ausfall Fahrzeuge drucken

Druckt die Ausfallzeiten für alle Fahrzeuge, mit automatischem Seitenumbruch.

7.10 Einstellungen

7.10.1 Qualifikationen

Alle Qualifikationen, die Mitarbeitern zugewiesen werden sollen, müssen zuvor in dieser Liste definiert werden.

7.10.1.1 Eingabedialog für Qualifikationen

Eine Qualifikation hat eine kurze Bezeichnung (die im Programm als Identifikation dient und z.B. in Listen angezeigt wird), sowie eine ausführlichere Beschreibung.

Die Felder für Verrechnungswerte beziehen sich auf eine (nicht implementierte) automatische Optimierung, sind hier also bedeutungslos.

7.10.2 Arbeitszeitprofile

Alle Arbeitszeitprofile, die Mitarbeitern zugewiesen werden sollen, müssen zuvor in dieser Liste definiert werden. Ein Arbeitszeitprofil gibt an, wann und wie lange ein Mitarbeiter im Dienst ist.

7.10.2.1 Eingabedialog für Arbeitszeitprofile

Es gibt drei Arten von Arbeitszeitprofilen, mit denen die meisten tatsächlichen Arbeitszeitverhältnis dargestellt werden können:

Vollzeit Mitarbeiter mit einer bestimmten Arbeitszeit pro Woche.

Teilzeit (tageweise) Mitarbeiter, die je nach Wochentag zu bestimmten Uhrzeiten arbeiten.

Teilzeit (stundenweise) Mitarbeiter mit einer bestimmten Arbeitszeit pro Monat.

Allen Arbeitszeitprofilen gemeinsam ist die Angabe einer maximalen Arbeitszeit pro Tag (wie sie z.B. von Arbeitsschutzrichtlinien vorgeschrieben wird).

7.10.3 Fahrzeugtypen

Alle Typen der vorhandenen Fahrzeuge müssen zuvor in dieser Liste definiert werden.

7.10.3.1 Eingabedialog für Fahrzeugtypen

Ein Fahrzeugtyp hat neben einer kurzen Bezeichnung und einer ausführlichen Beschreibung eine maximal zulässige Sitzplatzzahl sowie einen Kilometerpreis. Die verschiedenen Möglichkeiten, ein Fahrzeug umzubauen (z.B. zur Schaffung von Rollstuhlplätzen durch Ausbau von Sitzbänken) werden durch eine Menge von Fahrzeugkonfigurationen beschrieben. Für jeden tatsächlich vorhandenen Zustand eines Fahrzeuges muß in dessen Typ die passende Konfiguration definiert werden.

7.10.3.2 Eingabedialog für Fahrzeugkonfigurationen

Die Konfiguration wird durch Angabe der Anzahl vorhandener Plätze pro Platzart angegeben: Anzahl Sitzplätze, Anzahl Rollstuhlplätze, sowie zu jedem definierten Hilfsmittel die Anzahl Plätze, an denen ein solches Hilfsmittel fest montiert ist. Falls die Fahrzeugkonfiguration über eine Kühlmöglichkeit verfügt (z.B. Schienen für einen Kühlcontainer und ausreichend Platz für dessen Transport), kann dies angegeben werden.

7.10.4 Institutionen

Hier können alle Institutionen erfaßt werden, die mit einem Kunden in Beziehung stehen, z.B. Krankenkassen als Rechnungsempfänger.

7.10.4.1 Eingabedialog für Institutionen

Eine Insitution hat außer einem eindeutigen Namen eine Menge von Adressen, Kommunikations- und Bankverbindungen.

7.10.5 Stationen

Manche Stationen werden sehr oft verwendet, da sie z.B. das gemeinsame Ziel vieler Fahrten darstellen (Schulen oder Krankenhäuser). Deshalb können sie unter einem kennzeichnenden Namen hier erfaßt werden und stehen dann bei jeder Eingabe einer Station über den Button „Nachschlagen“ zur Verfügung. Wird bei der Eingabe einer Station, z.B. in einem Dienstwunsch, diese mit einem Namen versehen, wird sie ebenfalls in die globale Liste eingetragen.

7.10.5.1 Eingabedialog für benannte Stationen

Zusätzlich zur Adresse muß hier auch ein Name für die Station eingegeben werden, unter dem sie später gefunden werden kann.

Ist das Verkehrstool angeschaltet (siehe 7.10.9), wird ermittelt, ob die Adresse dort bekannt ist. Konnte sie nicht eindeutig identifiziert werden, erscheint ein Dialog mit Alternativvorschlägen des Verkehrstools. In den Eingabefeldern kann jetzt entweder einer dieser Vorschläge übernommen werden, oder man gibt eine Adresse an, die in der Nähe der ursprünglich gewünschten liegt (z.B. die nächstgrößere Straße). Ist diese Adresse dem Verkehrstool bekannt, wird sie als Grundlage für Fahrzeitberechnungen benutzt.

7.10.6 Hilfsmittel

Hier können alle Hilfsmittel erfaßt werden, die Kunden benötigen und deshalb in ihren Dienstwünschen fordern.

7.10.6.1 Eingabedialog für Hilfsmittel

Hilfsmittel werden durch eine kurze Bezeichnung sowie eine ausführlichere Beschreibung definiert.

7.10.7 Essensarten

Alle Essensarten, die der Dienst „Essen auf Rädern“ anbietet, müssen hier erfaßt werden, damit sie in den Dienstwünschen gewählt werden können.

7.10.7.1 Eingabedialog für Essensarten

Essensarten werden durch eine kurze Bezeichnung sowie eine ausführlichere Beschreibung definiert.

7.10.8 Feiertage

Damit das Programm weiß, welche Tage Feiertage sind (was sich auf das Stattfinden von Diensten auswirkt), müssen hier deren Daten eingegeben werden. Es empfiehlt sich, die Feiertage immer einige Monate im voraus zu erfassen, damit eventuelle Auswirkungen auf die Planung rechtzeitig berücksichtigt werden können.

7.10.8.1 Eingabedialog für Feiertage

Ein Feiertag wird durch Namen und Datum charakterisiert.

7.10.9 Verkehrstool

Um Entfernungen und Fahrzeiten zwischen den diversen Stationen automatisch ermitteln zu können, ist das Programm TROSS in der Lage, mit dem Routenplanungsprogramm „Map&Guide“ zusammenzuarbeiten. Dieses Programm wird hier als *Verkehrstool* bezeichnet.

7.10.9.1 Einstellungsdialog für das Verkehrstool

Hier kann angegeben werden, ob Map&Guide als Verkehrstool verwendet werden soll. Für den Einsatz von Map&Guide muß der Pfad zu diesem Programm sowie ein Verzeichnis angegeben werden, in das die Auftragsdateien für dessen Batch-Schnittstelle abgelegt werden (nähere Informationen hierzu sind dem Handbuch zu Map&Guide zu entnehmen sein).

Je nach Wahl des Benutzers arbeitet TROSS also in einem von zwei Modi:

mit Verkehrstool Hier werden sämtliche Entfernungen und Fahrzeiten vom Verkehrstool erfragt. Ebenso werden alle Stationen nach ihrer Eingabe dahingehend überprüft, ob sie dem Verkehrstool bekannt sind.

Da diese Anfragen über die langsame Batch-Schnittstelle von Map&Guide laufen, kann es jeweils zu gewissen Wartezeiten kommen.

Sollte der Modus mit Verkehrstool eingestellt sein, dieses aber aus irgendwelchen Gründen nicht korrekt ansprechbar sein, wird die momentane Operation mit einer entsprechenden Fehlermeldung abgebrochen.

ohne Verkehrstool Ist die Verwendung des Verkehrstool nicht vorgesehen, müssen alle im Programm notwendigen Entfernungen und Fahrzeiten von Hand eingegeben werden. Dazu wird dem Benutzer zur gegebenen Zeit ein Eingabedialog präsentiert, wo er die fehlenden Daten eintragen kann.

Achtung: Da im Modus ohne Verkehrstool keine Stationen geprüft werden können, werden diese zunächst als ungeprüft gekennzeichnet. Bei einem Wechsel in den Modus mit Verkehrstool müssen zunächst all diese Stationen vom Verkehrstool überprüft und gegebenenfalls vom Benutzer noch korrigiert werden. Beim Umschalten in den Modus mit Verkehrstool muß daher mit einer längeren Wartezeit gerechnet werden.

7.10.10 Entfernungen korrigieren

Ermöglicht die manuelle Eingabe von Entfernungen und Fahrzeiten zwischen zwei Stationen. Die hier eingegebenen Werte haben Vorrang vor denen des Verkehrstools und können somit benutzt werden, um dessen Werte zu korrigieren.

7.10.11 Maximale Fahrzeit

Hier wird der Standardwert für die Zeit erfaßt, die ein Kunde maximal im Fahrzeug verbringen darf. Dieser Wert gilt für alle Kunden, für die keine individuelle Grenze angegeben wurde.

Kapitel 8

Projektplanung

8.1 Planung für das Projekt Transportoptimierung

Da die Seminare nur indirekt mit dem Projekt bzw. dem Ziel der Erstellung eines lauffähigen Softwaresystems in Verbindung zu bringen sind, wurde diese Phase bei der Planung nicht berücksichtigt. Die ursprüngliche Zahl von fünf Projektgruppenmitgliedern verringerte sich auf vier, nachdem der fünfte Mann die Projektgruppe kurz nach dem Ende der Seminarphase verlassen hatte. Da dieser für die Projektgruppe prinzipiell keine verwendbaren Ergebnisse hinterlassen hatte, taucht er in der weiteren Planung nicht mehr auf.

8.1.1 Planung des Zeit- und Kostenaufwandes

Die Planung beruht auf den oben genannten Annahmen, sowie einem fikitiven Plan, der im Rahmen eines Vortrages über Projektplanung von Anke Drappa, einer Mitarbeiterin der Abteilung Software Engineering der Fakultät für Informatik der Universität Stuttgart, vorgestellt wurde. Der von den Betreuern vorgegebene Rahmen für die einzelnen Phasen der Projektgruppe sah folgendermaßen aus:

- 17.11.97 - 21.12.97 Anforderungsanalyse (5 Wochen)
- 22.12.97 - 31.01.98 Spezifikation (6 Wochen)
- 01.02.98 - 15.03.98 Entwurf (5 Wochen)
- 16.03.98 - 29.03.98 Zwischenbericht (2 Wochen)
- 30.03.98 - 24.05.98 Implementierung (8 Wochen)
- 25.05.98 - 28.06.98 Test (5 Wochen)
- 29.06.98 - 15.07.98 Enddokumentation (2,5 Wochen)

Dies entspricht einem Gesamtaufwand von **33,5 Wochen** für das gesamte Softwareprojekt. Abweichend vom Vorschlag von Anke Drappa, wurden die beiden Berichtsphasen mit einbezogen, da es sich bei diesen nicht um eine Erstellung

von Berichten, sondern vielmehr um eine Zusammenfügung bestehender Dokumente handelte, die im Rahmen der Projektarbeit erstellt wurden.
Betrachtet man die Kapazität eines einzelnen Mitarbeiters, kommt man auf folgenden Aufwand pro Projektgruppenmitglied:
33,5 Wochen · 2 Mitarbeitertage = 67 Mitarbeitertage = 502,5 Mitarbeiterstunden. Wobei von einem wöchentlichen Aufwand pro Mitarbeiter von ungefähr $15 \frac{Stunden}{Woche} (\cong 2 Tage)$ ausgegangen wurde. Umgerechnet auf alle Mitarbeiter ergibt sich daraus folgender Gesamtaufwand für das Projekt:
33,5 Wochen · 4 Mitarbeiter = 134 Mitarbeiterwochen = 268 Mitarbeitertage.
Werden pro Mitarbeiterstunde die fiktiven Kosten von **150,- DM** angesetzt, die wahrscheinlich weit unter einem realistischen, in der freien Wirtschaft veranlagten Wert liegen, ergeben sich folgende Kosten:
268 Mitarbeitertage · 7,5 Stunden · 150,- DM = 301 500,- DM.

8.1.2 Meilensteine

Bei einem Meilenstein handelt es sich um einen ausgezeichnten Zeitpunkt während des Projektverlaufes, an diesem ein vorher festgelegtes Ergebnis erwartet wird. Folgende Meilensteine waren während des Projekts zu erreichen:

- Beendigung der Anforderungsanalyse
  Ergebnis: angenommenes Dokument
  geplant : 21.12.97
  Abnahme : durch Review

- Beendigung der Spezifikation
  Ergebnis: angenommenes Dokument
  geplant : 30.01.98
  Abnahme : durch Review

- Vorführung Prototyp Oberfläche
  Ergebnis: Prototyp für die Benutzungsschnittstelle
  geplant : 06.02.98
  Abnahme : durch Kunden

- Beendigung des Entwurfs
  Ergebnis: angenommenes Dokument
  geplant : 13.03.98
  Abnahme : durch Review

- Beendigung des Zwischenberichts
  Ergebnis: angenommenes Dokument
  geplant : 27.03.98
  Abnahme : durch Review

- Beendigung der Implementierung
  Ergebnis: angenommenes Dokument
  geplant : 22.05.98
  Abnahme : durch Review/Kunden

- Beendigung des Tests
  Ergebnis: angenommenes Dokument

geplant : 26.06.98
Abnahme : durch Review

- Beendigung des Endberichtes
  Ergebnis: angenommenes Dokument
  geplant : 15.07.98
  Abnahme : durch Review

Hinzu kamen noch verschieden Besprechungstermine, um die Anforderungen des Kunden herauszuarbeiten und um offene Fragen bezüglich der Realisierung zu klären.

8.1.3 Projektverlauf

Die Zusammenfassung des Projektes 'ahnlich erfolgt, wie bei der Vorstellung der Meilensteine (8.1.2), in tabellarischer Form. Die einzelnen Daten werden doppelt dargestellt: Auf der einen Seite das Datum, an dem der Projektabschnitt tatsächlich begonnen hat, und auf der anderen Seite in Klammern der geplante Beginn. Eine Überschneidung der einzelnen Phasen kommt dadurch zustande, daß die einzelnen Phasen an ihren Schnittstellen teilweise parallel bearbeitet wurden, d.h. der Abschluß der einen Phase (Korrektur der Berichte, kleinere Änderungen, ...), fiel in den Beginn der neuen Phase.

- Anforderungsanalyse
  Beginn: 17.11.1997 (17.11.1997)
  was wurde getan:
  - einige Vorträge über Werkzeuge und Programmiersprachen, die für das Projekt nützlich sein könnten:
    * Genetische Algorithmen und das System Genom
      (Nicole Weickert und Alexander Leonardi)
    * Java
      (Fritz Hohl)
    * Projektplanung und MS-Project
      (Anke Drappa)
    * C++ und wxwin
      (Stefan Lewandowski)
    * Smalltalk und Visual Works
      (Tobias Spribille)
  - Vortrag des Kunden (Herr Schroff) über die Anforderungen an das System und die aktuelle Abwicklung der Fahrdienstplanung des DRK Stuttgart
  - Analyse des Ist-Zustandes beim Kunden
  - Erfassung der notwendigen Eingabedaten
  - Mitfahrt bei einzelnen Diensten zur Erfahrungssammlung
  - Erstellung einzelner Szenarien

  Review: 16.1.1998 (21.12.1997)

- Spezifikation
  Beginn: 19.1.1998 (22.12.1997)
  was wurde getan:
  - Architektur des Gesamtsystems
  - Erstellung eines Datenmodells
  - Aufbau und Funktionsweise des Verkehrsmoduls
  - Zusammenstellung der Datenausgabe
  - Möglichkeiten der Optimierung
  - Aussehen der Benutzungsoberfläche
  - Erstellung eines Prototyps für die Benutzungsoberfläche

  Review: 27.3.1998 (13.3.1998)

- Entwurf
  Beginn: 2.3.1998 (2.3.1998)
  was wurde getan:
  - Umsetzung der Spezifikation in passende Klassenstruktur
  - Konzepte der Datenhaltung
  - Erstellung der Menüstruktur

  Review: 27.3.1998 (13.3.1998)

- Zwischenbericht
  Beginn: 30.3.1998 (16.3.1998)
  was wurde getan:
  - Zusammenstellung bisher erstellter Dokumente
  - korrekturlesen

  Dokument fertig: 29.4.1998 (27.3.1998)

- Implementierung
  Beginn: 3.4.1998 (30.3.1998)
  was wurde getan:
  - Ausprogrammierung des Enturfs
    (aufgrund des großen Zeitaufwandes fiel die Optimierung weg)
  - Vorführung einer Version 0

  Review über Teile 20.7.1998 (22.5.1998)
  vollständige Abgabe Version 1: 13.8.1998
  Übergabe an den Kunden: 21.9.1998

- Test
  Beginn 9.7.1998 (1.6.1998)
  was wurde getan:
  → Test erfolgte über eine unvollständige Implementierung!
  - Funktionalitätstest der einzelnen Masken

  - Test des Laufzeitverhaltens
  - Abgleich mit den Szenarien

  Review: 23.7.1998 (26.6.1998)

- Endbericht
  Beginn: 2.7.1998 (29.6.1998)
  was wurde getan:

  - Zusammenstellung bisher erstellter Dokumente
  - korrekturlesen

  Abgabe Rohfassung: 18.8.1998 Dokument fertig: 5.10.1998 (15.7.1998)

8.1.4 Tatsächlicher Zeit- und Kostenaufwand

Zum Abschliessen des Projektes wurde als Stichtag der Abgabetag des Endberichtes (18.08.98) genommen. Gegenüber dem geplanten Aufwand von **33,5 Wochen** steht ein tatsächlicher Aufwand von **39 Wochen**. Dieser Rahmen konnte aber nur eingehalten werden, da – vor allem in der Schlußphase der Projektgruppe – vieles parallel bearbeitet wurde. Den Arbeitsaufwand pro Projektgruppenmitglied läßt zeigt Abbildung 8.1. Somit ergibt sich ein Gesamtaufwand für das Projekt von **2186 Stunden** bzw. ungefähr **292 Tagen**. Werden die unter 8.1.1 angesetzten fiktiven Kosten von **150,- DM** pro Mitarbeiterstunde angesetzt, ergeben sich folgende Kosten:
2186 Mitarbeiterstunden · 150,- DM = 327 900,- DM. Das geplante Budget wurde also um **26400,- DM** überschritten. Wäre das System wie geplant ausprogrammiert worden, wären der Kosten- und Zeitüberhang noch deutlicher ausgefallen.

	Arbeitszeitaufwand pro Mitarbeiter			
Woche	Frank	Jörg	Lars	Tobias
47	7,50	7,50	5,00	6,50
48	7,50	7,50	6,00	5,50
49	7,50	7,50	6,00	8,50
50	7,50	7,50	7,00	11,50
51	7,50	7,50	6,00	13,00
52	7,50	7,50	20,50	18,50
1	7,50	7,50	16,00	31,00
2	7,50	7,50	11,00	8,50
3	7,50	17,25	8,50	12,50
4	19,75	22,00	14,50	17,00
5	14,00	17,00	18,50	22,00
6	4,25	23,25	17,50	22,00
7	22,00	24,25	18,00	19,00
8	10,00	21,75	14,00	15,50
9	22,25	9,00	13,50	10,00
10	8,75	9,50	12,00	13,50
11	13,50	20,00	18,50	4,50
12	6,00	18,00	9,00	6,50
13	0,00	16,25	13,00	9,50
14	12,00	12,00	13,00	0,00
15	9,50	10,00	18,00	0,00
16	16,00	15,75	0,00	7,50
17	17,00	14,00	17,00	11,00
18	9,00	17,00	16,50	14,50
19	15,00	10,75	21,00	28,50
20	26,00	15,50	23,00	35,00
21	17,25	21,50	19,50	19,00
22	26,25	20,75	23,00	26,00
23	25,75	19,75	21,00	22,50
24	40,75	17,00	28,00	19,00
25	7,50	14,75	16,50	23,50
26	37,50	11,00	7,00	15,50
27	15,25	10,50	5,00	21,00
28	35,50	19,25	5,00	25,00
29	17,75	22,00	18,00	11,00
30	26,50	31,00	17,00	0,00
31	0,00	11,00	12,00	0,00
32	0,00	21,25	12,00	0,00
33	0,00	0,00	9,50	0,00
Gesamt	542,50	573,00	536,50	534,00

Abbildung 8.1: Stundenzahl pro Projektgruppenmitglied

Kapitel 9

Rückblick

9.1 Zeitplanung

Der grobe Zeitrahmen für die einzelnen Phasen der Projektgruppe war bereits zu Anfang von den Betreuern fest vorgegeben. Da er seltsamerweise zwei Monate über das Ende der Projektgruppenzeit hinausging, mußte er gekürzt werden, um alle vorgesehenen Phasen bis zum Ende des Vorlesungszeitraums im Juli 1998 unterzubringen. Die beiden letzten Phasen, nämlich Implementierung und Test, wurden radikal gekürzt. Im Nachhinein trägt diese Maßnahme entscheidend dazu bei, daß nicht alles im Programm umgesetzt werden konnte und selbst die wichtigsten Bestandteile nur durch enorme Mehrarbeit der Projektgruppe in der Implementierungsphase überhaupt bis zur vollen Funktionalität gebracht werden konnten.

Ein weiteres zeitliches Problem war natürlich die Aufgabenstellung, die insgesamt einfach zu komplex war, um sie mit vier Personen in dieser Zeit komplett zu bewältigen (siehe 9.2). Sowohl dem Verlauf als auch dem Ergebnis der Projektgruppe wäre es sicher zugute gekommen, nach der Anforderungsanalyse, als die Überdimensionierung der Aufgabe durchaus schon zu sehen war, einen sauberen Schnitt zu machen. Spätestens aber nach dem Entwurf hätte man sich dazu durchringen müssen, im Interesse eines lauffähigen Gesamtsystems von vornherein gewisse Teile nicht zu implementieren. So wurden zunächst nahezu alle Module parallel begonnen, und am Schluß fehlte die Zeit, die Arbeit wirklich zu Ende zu führen.

Ebenso wurde der Zeitaufwand für die Reviews nach jeder Phase bei der Planung nicht extra eingeplant, so daß letztlich jede Phase um eine bis zwei Wochen in die nächste überhing, während derer das Review vorbereitet und durchgeführt werden mußte.

Besonders realitätsfern war das Vorhaben, die gesamten vorlesungsfreie Zeit zu verplanen. Daß in dieser Zeit mit Prüfungen zu rechnen war, sollte jedem, der selbst einmal studiert hat, ebenso klar sein, wie die Tatsache, daß intensive Prüfungsvorbereitung eine gewisse Zeit erfordert, in der für umfangreiche sonstige Aufgaben kein Platz ist, weder zeitlich noch gedanklich. Zeit für Urlaub oder sonstige Erholung (die in der freien Wirtschaft längst als grundlegend wichtig für die Motivation und die Leistungsfähigkeit der Mitarbeiter erkannt wurde) wurde den Studenten der Projektgruppe im Zeitplan nicht zugestanden.

9.2 Umfang der Aufgabenstellung

Mit einem echten Kunden konnte bisher noch keine Projektgruppe der Uni Stuttgart arbeiten, so daß zunächst eine gewisse „Wie im richtigen Leben"-Euphorie vorhanden war. Daß allerdings eine detaillgetreue Modellierung der Realität einen enormen Aufwand erfordert, wurde schnell klar. Bereits die halbwegs schematisch geordnete Erfassung und Dokumentation der Anforderungen zeigte die besonders komplizierten (oder auch nur komplexen) Begriffe auf. Während Spezifikation und Entwurf, beim Umsetzen in Datenstrukturen und Algorithmen, Verfahren und Schnittstellen kam deutlich zutage, daß mit vier Personen in acht Wochen an eine komplette Umsetzung der bisher entwickelten Ideen in ein laufendes Programm nicht zu denken war.

9.3 Zuständigkeiten und Kompetenzen in der Projektgruppe

Ein Grundproblem bei der praktischen Durchführung dieser Projektgruppe war die weder eindeutig noch sinnvoll festgelegte Verteilung von Entscheidungskompetenz einerseits und Verantwortung für Entscheidungen andererseits:

So wurden sowohl Zeitplan als auch die Aufgabe von den Betreuern vorgegeben. Versuche von Seiten der Projektgruppe, die Aufgabenstellung auf das zeitlich Mögliche einzuschränken, wurden meist abgeblockt mit Auflistung von Funktionen, die noch „gemacht werden *müssen*". Der „für den Zeitplan zuständige" Student konnte diese Vorgaben lediglich mit dem Programm „MS Project" verwalten, und hatte die undankbare Aufgabe, die regelmäßigen Kassandra-Rufe zum zeitlichen Stand des Projekts zu verkünden.

Die Beschaffung des Verkehrstools (Map&Guide) erfolgte als verblüffender Schnellschuß: Lange Zeit ruhte man sich auf der scheinbaren Gewißheit aus, daß passende Verkehrsdaten zur Verfügung stehen würden. Nachdem sich dies als falsch herausstellte (da die an der Universität vorhandenen Verkehrsdaten nicht an einen Fremdkunden der Projektgruppe weitergegeben werden durften), mußte auch das Verkehrstool selbst beschafft werden. Während der zuständige Projektgruppenteilnehmer noch Anbieter von Streckenplanungsprogrammen anschrieb, um eine möglichst kostenlose Überlassung der Daten zu Studienzwecken zu erbetteln, war plötzlich das Programm Map&Guide schon gekauft (zu einem Betrag, den wir nie für Hilfsmittel einzuplanen gewagt hätten). Daß das Programm eine externe Schnittstelle anbietet, war die einzige Information, die vor dessen Eintreffen an der Uni zur Verfügung stand. Später stellte sich dann heraus, daß die Batch-Schnittstelle weder von der Geschwindigkeit noch vom möglichen Umfang der Anfragen für die Anforderungen des Programms TROSS geeignet war.

9.4 Empfehlungen an zukünftige Projektgruppen

Auch wenn es bisher in diesem Dokument (wie auch schon im Zwischenbericht) fast immer nur um die zu modellierende Aufgabenstellung und das dafür geschriebene Programm ging, sind das Hauptziel einer Projektgruppe die vielfältigen Lerneffekte. Um die diversen Erfahrungen dieser Projektgruppe, die oft

genug durch mühsames „Learning by doing“ erzielt wurden, nicht nur für uns zu behalten, sollen hier einige Punkte genannt werden, die künftige Projektgruppen unserer Meinung nach beachten sollten. Dadurch kann hoffentlich (bis zu einem gewissen Grad) vermieden werden, daß jede Projektgruppe wieder – wie wir – von Null auf beginnt, sondern durch Nutzung dieser Erkenntnisse die Projektzeit sinnvoller für die eigentlich interessanten Problemstellungen verwendet. Das schlägt sich bestimmt auch positiv im Ergebnis nieder (also in einem guten, lauffähigen Programm).

- Der Zeitplan sollte die verfügbare Zeit nicht bis auf den letzten Tag verplanen. Der Verlauf mehrerer Projektgruppen hat gezeigt, daß die Zeit immer überschätzt (bzw. die Aufgabe unterschätzt) wird. Ein Pufferzeitraum am Ende wäre sicherlich kein Fehler.

- Reviews müssen als feste Größen im Zeitplan vorgesehen werden. Dazu muß nach der eigentlichen Durchführung jeder Phase mindestens eine Woche ausschließlich für Vorbereitung, Durchführung und Nachbereitung des Reviews zur Verfügung stehen.

- Die Aufgabenstellung erforderlichenfalls anpassen. Lieber ein kleines, lauffähiges Programm, als eine große, universelle Planungsruine.

- Die Entscheidung über die benutzte Programmiersprache sollte möglichst früh fallen. Dann kann man sich rechtzeitig nach Entwicklungsumgebungen, Bibliotheken mit Hilfsfunktionen etc. umschauen und in die Syntax und Philosophie der Sprache einarbeiten.

- Ein Prototyp sollte nicht zu früh erstellt werden, da man sonst nicht nur viel zusätzliche Arbeit hat, sondern sich auch in selbsterdachten Datenstrukturen verrennt, die dem tatsächlichen Datenmodell dann in die Quere kommen (siehe 3.5).

Literaturverzeichnis

[Bal96] Balzert, Helmut: *Lehrbuch der Softwaretechnik*, Band 1: Software-Entwicklung. Spektrum-Verlag, 1996.

[BN97] Bernstein, Philip A. und Eric Newcomer: *Principles of transaction processing.* The Morgan Kaufmann series in data management systems. Morgan Kaufmann, San Francisco, Calif., 1997. XXIV, 358 S.

[CAS97] CAS Software GmbH: *Map&Guide Benutzerhandbuch*, 1997.

[Fla97] Flanagan, David: *Java in a Nutshell.* O'Reilly, Zweite Auflage, Mai 1997.

[Pre92] Pressman, Roger S.: *Software engineering : a practitioner's approach.* McGraw-Hill international editions : computer science series. McGraw-Hill, New York [u.a.], Dritte Auflage, 1992. 793 S.

[Pro98] Projektgruppe Transportoptimierung: *Zwischenbericht – Bericht 1998/06.* Institut für Informatik der Universität Stuttgart, 1998.