Inhaltsverzeichnis

1 Nachtrag zum Entwurf **2**
- 1.1 Neue Klassen und Interfaces . . . 2
  - 1.1.1 Arbeitszeitprofile . . . 2
  - 1.1.2 DienstplanErstellbar . . . 4
  - 1.1.3 Zahler-Interface . . . 4
  - 1.1.4 Druckbar . . . 4
- 1.2 Allgemeine Konzepte . . . 4
  - 1.2.1 Stationen und Anbindungspunkte . . . 4
  - 1.2.2 Untertour . . . 6
  - 1.2.3 Verkehrsmodul . . . 10

2 Implementierung **15**
- 2.1 Entwurfsentscheidungen auf Programmiersprachenebene . . . 15
  - 2.1.1 Allgemeine Konzepte . . . 15
  - 2.1.2 GUI-Erweiterungen . . . 16
- 2.2 Designentscheidungen . . . 19
  - 2.2.1 Fensteraufbau . . . 19
  - 2.2.2 Destruktoren . . . 20
- 2.3 Umsetzung des Entwurfs in der Implementierung . . . 20
  - 2.3.1 Personenklassen . . . 20
  - 2.3.2 Datenhaltung . . . 25
  - 2.3.3 Konsistenztests . . . 28
  - 2.3.4 Analysedaten . . . 31
  - 2.3.5 Datenausgabe . . . 33
- 2.4 Probleme beim Umsetzen des Entwurfs . . . 33
  - 2.4.1 Unvollständige Details im Entwurf . . . 33
  - 2.4.2 Einfach, aber aufwendig . . . 34
- 2.5 Implementierung ausgehend von einem Prototyp . . . 35
- 2.6 Verwendung externer Programme: Das Verkehrsmodul . . . 35
- 2.7 Erfahrungen mit Java . . . 37
  - 2.7.1 Javas Klassenbibliothek . . . 37
  - 2.7.2 Entwicklungsumgebung . . . 38

Kapitel 1

Nachtrag zum Entwurf

1.1 Neue Klassen und Interfaces

1.1.1 Arbeitszeitprofile

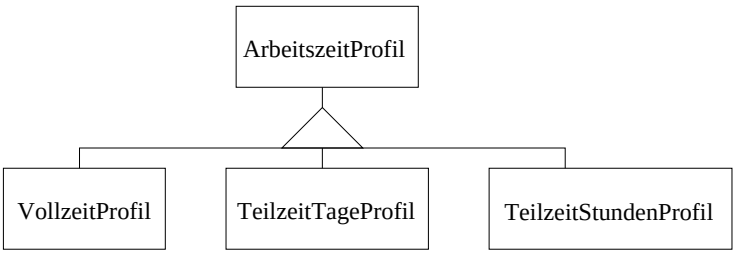
Bei der Einteilung von Mitarbeitern für Touren und Fahrten sollte überprüft werden können, ob der Mitarbeiter zur vorgesehenen Zeit überhaupt eingesetzt werden kann. Dabei soll zum einen berücksichtigt werden, an welchen Tagen bzw. zu welchen Zeiten ein Mitarbeiter prinzipiell im Dienst ist, zum anderen 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.2 Allgemeine Konzepte

1.2.1 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.

1.2.2 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 Teil-Dienstwunsch 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.

1.2.2.1 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.

1.2.2.2 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.

1.2.2.3 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.

1.2.2.4 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.

1.2.3 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 2.6) 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.

1.2.3.1 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.

1.2.3.2 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).

1.2.3.3 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.

1.2.3.4 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 2

Implementierung

2.1 Entwurfsentscheidungen auf Programmiersprachenebene

2.1.1 Allgemeine Konzepte

2.1.1.1 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.

2.1.1.2 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.

2.1.2 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`)

2.1.2.1 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 [**?**]:

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 2.1.2.2 und 2.1.2.5 beschrieben) werden direkt nach deren Bestätigung in den Datenobjekten durchgeführt, da die obige Vorgehensweise hier noch deutlich komplizierter umzusetzen wäre.

2.1.2.2 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.

2.1.2.3 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 2.1.2.1 beschriebenen zweistufigen Mechanismus') und dann den Dialog schließt.

2.1.2.4 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.

2.1.2.5 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.

2.1.2.6 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.

2.2 Designentscheidungen

2.2.1 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.

2.2.2 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.

2.3 Umsetzung des Entwurfs in der Implementierung

2.3.1 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.

2.3.1.1 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.

2.3.1.2 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.

2.3.1.3 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.

2.3.1.4 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.

2.3.1.5 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.

2.3.1.6 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.

2.3.1.7 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.

2.3.1.8 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.

2.3.1.9 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.

2.3.1.10 Sachbearbeiter

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

- Attribute

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

2.3.1.11 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 1.2.1).

- 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.

2.3.1.12 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.

2.3.2 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 2.1 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.

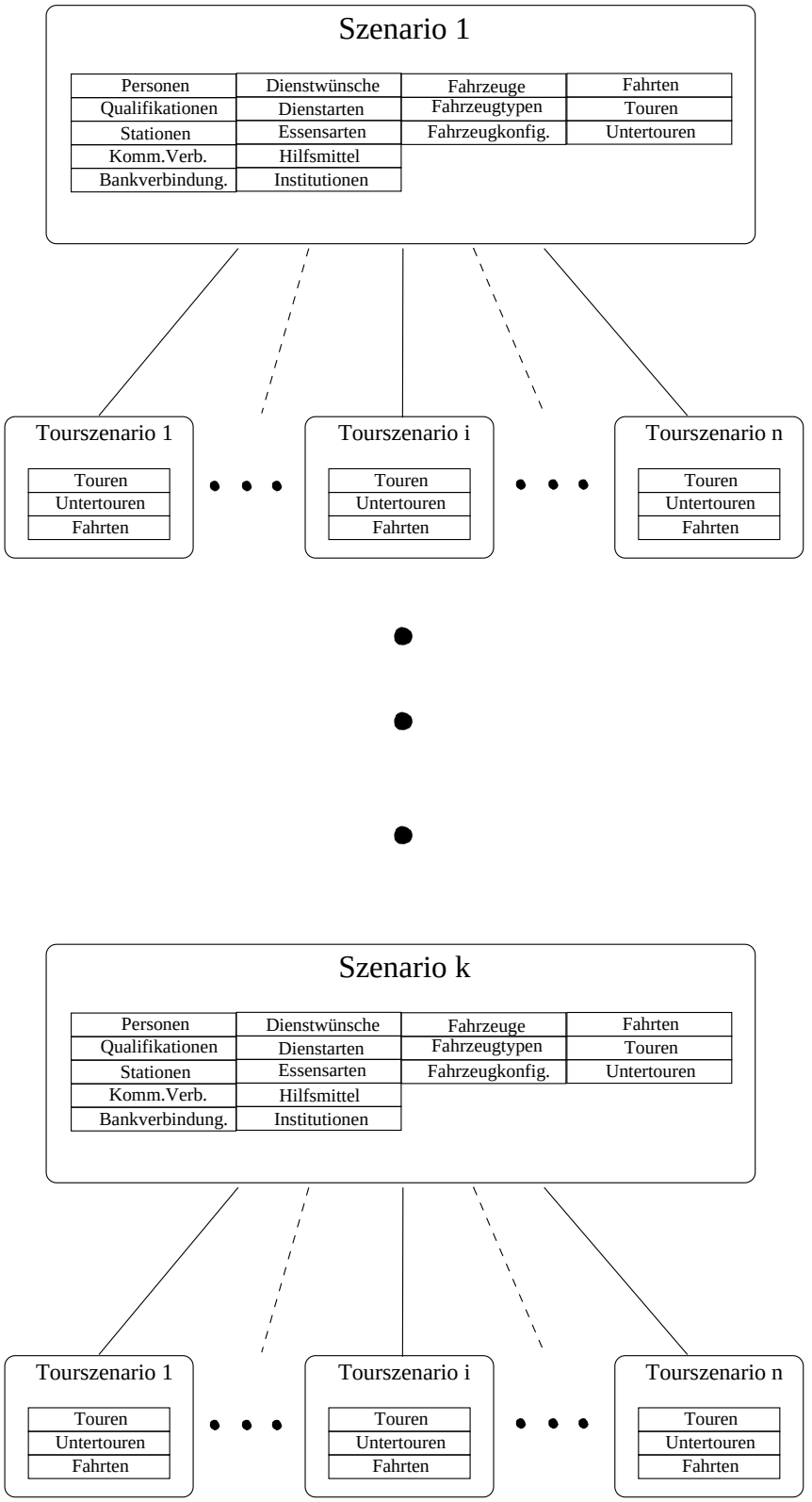


Abbildung 2.1: Datenhaltung in TROSS

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.

2.3.2.1 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 2.1.1.1) 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.

2.3.2.2 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.

2.3.3 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.

2.3.3.1 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.

2.3.3.2 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.

2.3.3.3 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.

2.3.4 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

2.3.4.1 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.

2.3.4.2 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.

2.3.5 Datenausgabe

2.3.5.1 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 2.3.2.2 beschrieben, in O(Anz. d. Fahrten).

Das im Abschnitt 1.1.2 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.

2.3.5.2 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 2.3.2.2 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.

2.3.5.3 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.

2.4 Probleme beim Umsetzen des Entwurfs

2.4.1 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 2.4.2.1).

2.4.2 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.

2.4.2.1 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 2.1.2.6).

2.4.2.2 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.

2.4.2.3 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.

2.5 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 2.1.2.1) 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.

2.6 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 [?]).

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 1.2.3).

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.

2.7 Erfahrungen mit Java

2.7.1 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 2.1.1.1).

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 2.1.1.1 erwähnten Interface `Comparable` eine Standard-Implementierung der Vergleichsmethode mitzugeben. In Java mußte der etwas aufwendige Stringvergleich nach länderspezifischen Sortierregeln vielfach redundant codiert werden.

2.7.1.1 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.

2.7.2 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 2.1.2.1) 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.