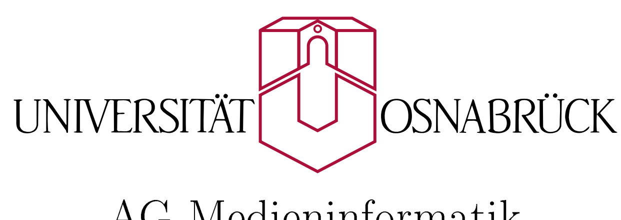


AG Medieninformatik

Bachelorarbeit

Entwurf und Implementation einer Datenbank zur Visualisierung von Verkehrsdaten im Internet

Uwe Hebbelmann

30. September 2009

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachter: Prof. Dr. Elke Pulvermüller

Danksagungen

Ich möchte allen, die mich bei der Realisierung dieser Arbeit unterstützt haben, einen besonderen Dank aussprechen:

- Herrn Prof. Dr. Oliver Vornberger für die Bereitstellung des interessanten Themas und die Tätigkeit als Erstgutachter.
- Frau Prof. Dr. Elke Pulvermüller, die sich für diese Arbeit als Zweitgutachter zur Verfügung gestellt hat.
- Dorothee Langfeld für ihre hervorragende, kompetente, stets freundliche und geduldige Betreuung während der gesamten Zeit meiner Bachelorarbeit.
- Friedhelm Hofmeyer für die Bereitstellung der benötigten Hard- und Software im Institut für Informatik.
- Patrick Fox für seine Bereitschaft, Fragen zu beantworten und hilfreiche Tips und Hinweise zu geben.
- Meiner Mutter, Chris, Dorothee und Inga für das Korrekturlesen dieser Arbeit.
- Ein besonderer Dank gilt meinen Eltern, die mir nicht nur das Studium finanziell ermöglicht, sondern mich darüberhinaus in jeglicher Hinsicht stets bedingungslos unterstützt haben.
- Mein ganz besonderer Dank geht an meine Freundin Inga für die moralische Unterstützung sowie Motivation während meines Studiums, ihrer unendlichen Geduld und ihren unschätzbaren, liebevollen Rückhalt.

Rechtliches

Alle in dieser Arbeit genannten Unternehmens- und Produktbezeichnungen sind in der Regel geschützte Marken- oder Warenzeichen. Auch ohne besondere Kennzeichnung sind diese nicht frei von Rechten Dritter zu betrachten. Alle erwähnten Marken- oder Warenzeichen unterliegen uneingeschränkt den länderspezifischen Schutzbestimmungen und den Besitzrechten der jeweiligen eingetragenen Eigentümern.

Zusammenfassung

Zur Visualisierung von Verkehrsdaten im Internet soll ein Datenbankkonzept entworfen und implementiert werden. Die Ausgangsdaten werden in Dateien im XML-Format geliefert. Diese unterscheiden sich in Basisdaten, die das deutsche Straßennetz beschreiben, und Verkehrsdaten, die Verkehrsdichteangaben für einzelne Straßenabschnitte machen. Es gilt, ein geeignetes Datenbanksystem auszuwählen, die Datenbankstruktur anzulegen und Methoden zu implementieren, um einerseits die aufkommenden Daten in die Datenbank ein zu pflegen und andererseits die Daten aus der Datenbank wieder abfragen zu können. Das Einlesen der Daten soll auf einem Internet-Server automatisiert, die Abfragen ebenfalls über das Internet verfügbar gemacht werden. Neben einem Weiterverarbeitungsformat zur späteren Visualisierung, soll eine einfache Anzeige-Komponente erstellt werden.

Abstract

In order to visualize traffic data in the Internet, a database concept is to be designed and implemented. The original data are supplied in XML files. These vary in basis data, which describe the german road system and traffic data, which depict the traffic density for sections of the road system.
First an eligible database system needs to be selected and the database structure created. On the one hand methods must be implemented to write the received data to the database and on the other hand query the data again from the database. Reading in the data needs to be automated on an internet server. The querys have to be available from the internet as well. Besides a format to further process the data for visualisation a simple possibility to show the data is required.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 "DDG Gesellschaft für Verkehrsdaten mbH (DDG)" 1
1.2 Motivation und Aufgabenstellung 2
1.3 Gliederung der Arbeit 3

2 Einführung in benötigte Grundlagen **5**
2.1 Datenbanksystem 5
2.2 Structured Query Language (SQL) 6
2.3 eXtensible Markup Language (XML) 6
2.3.1 Aufbau von XML-Dokumenten 7
2.3.2 XML-Parser 9
2.4 Scalable Vector Graphics (SVG) 11
2.4.1 Aufbau einer SVG-Datei 12
2.4.2 Attribut `viewBox` 12
2.4.3 Elemente 13
2.5 Java 15
2.6 Apache Tomcat 18

3 Aufbau der Eingangsdaten **19**
3.1 Grundsätzlicher Aufbau 19
3.2 Geocodes 19
3.3 Aufbau der Basisdaten 19
3.4 Aufbau der Verkehrsdichtedaten 21

4 Datenbank **27**
4.1 PostGIS 27
4.1.1 GIS Objekte 27
4.1.2 Grundlegende Funktionen und Operatoren 28
4.2 Konzept und Grundstruktur 30

5 Java-Programm **33**
5.1 Überprüfen und Anlegen der Datenbankstruktur 33
5.2 Einlesen der Daten 34
5.2.1 Einlesen der Basisdaten 34
5.2.2 Einlesen der Verkehrsdichtedaten 40
5.3 Ausführung 44
5.3.1 Ausführung ohne Programmfenster in der Konsole 44
5.3.2 Ausführung mit grafischer Oberfläche 45

6 Web Services **47**
6.1 Java-Servlets 47

6.2 Datenbankabfragen 49
6.3 Rückgabeformate 52

7 Fazit und Ausblick **55**

Anhang **57**

Abkürzungsverzeichnis **59**

Literaturverzeichnis **60**

Abbildungsverzeichnis **63**

Tabellenverzeichnis **65**

Algorithmenverzeichnis **67**

Daten-CD **69**

Erklärung **71**

1 Einleitung

Die Motivation der vorliegenden Arbeit bestand darin, eine sinnvolle Grundlage zu schaffen, um Verkehrsdaten im Internet visualisieren zu können. Die Verkehrsdaten liefern Informationen über die Verkehrsdichte auf Straßenabschnitten des deutschen Straßennetzes und werden von der "DDG Gesellschaft für Verkehrsdaten mbH (DDG)" erhoben.

1.1 "DDG Gesellschaft für Verkehrsdaten mbH (DDG)"

Die DDG liefert die Basis für eine effiziente Verkehrstelematik. Das Unternehmen gehört zu den führenden Content-Providern für Diensteanbieter der Verkehrstelematik. Derzeit nutzt die DDG die folgenden Datenquellen zur Erhebung der Informationen:

- Stationäres Erfassungs-System (SES) mit fest montierten Sensoren an den Autobahnen zur objektiven Messung des Verkehrsflusses
- Floating Car Data-Verfahren (FCD)-Verfahren: Telematikendgeräte in Fahrzeugen, die als mobile Sensoren zu einem aktuellen, exakten Datenbild des Verkehrsgeschehens verhelfen
- Verkehrsdaten der Polizei, die von den Landesmeldestellen weiter verarbeitet werden
- Verkehrsinformationszentralen der einzelnen Bundesländer
- Induktionsschleifen, die in die Fahrbahnen eingelassen sind

Eine höhere Informationsqualität soll durch eine Datenauswertung erreicht werden. Hierzu benutzt die DDG ein selbst entwickeltes Verfahren unter Einsatz von erprobten Verkehrsmodellen. Es verwendet unter anderem Verkehrsflussmodelle, Expertensysteme und neuronale Netze. Im Rechenzentrum der DDG werden alle gewonnenen Daten sorgfältig überprüft und aufbereitet. Die Aufbereitung der für diese Arbeit relevanten Daten geschieht durch das Traffic Map-Verfahren, welches eine grafische Darstellung der Verkehrslage ermöglicht. Dabei werden feingranulare, navigationsfähige Darstellungen unterstützt. Ein Update-Zyklus bis herunter zu einer Minute sorgt für eine notwendige Aktualität [DDG 09]. Die auf diese Weise erhaltenen Daten werden in Form von XML-Dateien zur Verfügung gestellt.

1.2 Motivation und Aufgabenstellung

Wunsch der DDG ist es, die erhaltenen Informationen animiert auf einer Karte darzustellen, um die jeweiligen Verkehrszustände erkennen zu können und mittels Abfragen aus dem Internet zugänglich zu machen. Anhand von historischen Daten können z.B. besondere Vorkommnisse zugeordnet oder Erkenntnisse für Stauprognosen gewonnen werden (Feierabendverkehr, Ferienzeiten, Feiertage, etc.).

Um mit den aufkommenden Datenmengen effektiv und sicher arbeiten zu können, soll ein Datenbanksystem verwendet werden. Es galt, ein geeignetes Datenbanksystem auszuwählen, die Datenbankstruktur anzulegen und Methoden zu implementieren, um einerseits die aufkommenden Daten in die Datenbank einpflegen zu können und andererseits die Daten aus der Datenbank wieder abfragen zu können. Damit sollte eine Grundlage zur späteren Visualisierung der Daten geschaffen werden, die jedoch nicht mehr Bestandteil dieser Arbeit gewesen ist.

Die Aufgabe dieser Arbeit gliederte sich im Groben in drei Teile. Zunächst musste ein Datenbanksystem ausgewählt und auf einem Internetserver eingerichtet werden. Zum Anlegen der Datenbankstruktur wurden entsprechende Methoden in einem Java-Programm implementiert. Die erhobenen Verkehrsdaten in Form von XML-Dateien sollen auf dem Internetserver in einem dafür vorgesehenen Verzeichnis eingestellt werden. Das Java-Programm wurde um Methoden zur Verzeichnisüberwachung erweitert, welche eingehende Dateien automatisch erkennt und in die Datenbank einliest. Zur Anzeige der Daten wurde ein Web Service implementiert, der entsprechende Methoden zur Abfrage der Informationen aus der Datenbank enthält. Dabei sollten Abfragen berücksichtigt werden, die das Straßennetz bzw. Straßennetzausschnitte zurückliefern. Für eine spätere Weiterverarbeitung wurde das XML-Format ausgewählt, welches einer animierten Darstellung dienen soll. Des Weiteren sollte das SVG-Format unterstützt werden. Folgende Abfragen waren zu implementieren:

Das Straßennetz bzw. Straßennetzausschnitte

- ohne Verkehrsdichtedaten
- mit Verkehrsdichtedaten zu einem bestimmten Zeitpunkt

(Eine HTML-Seite zum Testen der implementierten Abfragen wurde unter der Adresse `http://dbs.informatik.uni-osnabrueck.de/trafficdb` zur Verfügung gestellt)

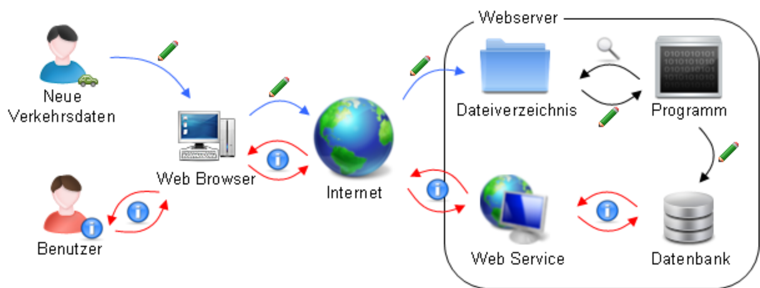


Abb. 1.1: Vorgehensweise

1.3 Gliederung der Arbeit

Kapitel 2 enthält einen allgemeinen Überblick über die in dieser Arbeit verwendeten Basistechnologien. In Kapitel 3 wird der Aufbau der Eingangsdaten erläutert. Kapitel 4 soll eine allgemeine Einführung in das ausgewählte Datenbanksystem geben und die Datenbankstruktur veranschaulichen. In Kapitel 5 wird der Aufbau und die Funktionsweise des implementierten Java-Programms beschrieben. Kapitel 6 zeigt die Umsetzung der Abfragen mit Hilfe von Web Services. In Kapitel 7 soll ein Fazit gezogen und ein Ausblick auf zukünftige Möglichkeiten und Probleme gegeben werden.

2 Einführung in benötigte Grundlagen

2.1 Datenbanksystem

Die wesentliche Aufgabe eines Datenbanksystems besteht darin, große Datenmengen effizient, widerspruchsfrei und dauerhaft zu speichern und benötigte Teilmengen in unterschiedlichen, bedarfsgerechten Darstellungsformen für Benutzer und Anwendungsprogramme bereitzustellen. Es stellt ein computergestütztes System dar, das sich aus einem Datenbankmanagementsystem (DBMS), welches intern die strukturierte Speicherung der Daten organisiert und alle Lese- bzw. Schreibzugriffe auf die Datenbank kontrolliert, und einer zu verwaltenden Datenbasis zur Beschreibung eines Ausschnitts der Realwelt zusammensetzt [Vorn 09, vgl. S. 11].

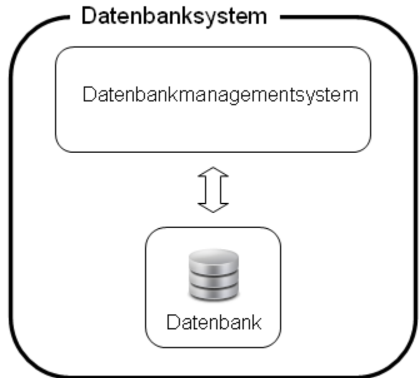


Abb. 2.1: Aufbau eines Datenbanksystems

Wesentliche Merkmale bzw. Anforderungen von DBMS sind:

- die persistente Verwaltung bzw. Speicherung langfristig zu haltender Daten
- die effiziente Verwaltung der Daten, so dass ein Zugriff auf einen bestimmten Datensatz möglichst schnell erfolgen kann
- die Bereitstellung von Operationen (mittels deskriptiver Sprachen) zur Modifikation bzw. Selektion der Daten
- die Vermeidung von Redundanz und Inkonsistenz der Daten
- Anfrageoptimierung
- Sicherstellung der Datenintegrität durch Transaktionen (d.h. zusammengehörende Operationen werden unteilbar entweder als Ganzes oder gar nicht durchgeführt)
- Mehrbenutzerfähigkeit, d.h. die gleichzeitige Benutzung einer Datenbank durch verschiedene Nutzer erfolgt ohne gegenseitige Beeinflussung

- Datensicherheit und Datenschutz durch Benutzerrollen, Rechte, etc.

[Kemp 01, vgl. S. 15 ff.]

Auf das Datenbanksystem, welches im Rahmen dieser Arbeit zum Einsatz kommt, wird in Kapitel 4 näher eingegangen.

2.2 Structured Query Language (SQL)

SQL ist eine Anfragesprache für relationale Datenbanken. Der Ursprung liegt in dem Anfang der 70er Jahre in einem IBM-Forschungslabor entwickelten Prototypen System/R mit der Anfragesprache Sequel [Heue 97, vgl. S. 261]. SQL ist ANSI- und ISO-standardisiert (aktueller Standard SQL-92, auch SQL 2 genannt) und wird von fast allen gängigen Datenbanksystemen unterstützt, wodurch es möglich ist, datenbankunabhängige Anwendungsprogramme zu erstellen.

Grundlage der Sprache bilden die Relationale Algebra und der Relationenkalkül. SQL umfasst dabei die Datenbanksprachen Data Manipulation Language (DML), Data Definition Language (DDL) und Data Control Language (DCL). Die Syntax von SQL ist relativ einfach aufgebaut und semantisch an die englische Sprache angelehnt. Es werden Befehle zur Manipulation und Abfrage der Datenbestände, aber auch zur Definition des relationalen Schemas, zur Formulierung von Integritätsbedingungen, zur Vergabe von Zugriffsrechten und zur Transaktionskontrolle zur Verfügung gestellt [Vorn 09, vgl. S. 82].

In dieser Arbeit kommt SQL bei allen Datenbank-Interaktionen zum Einsatz, d.h. sowohl beim Erstellen der Datenbankstruktur (Kapitel 5.1) als auch beim Einlesen der Daten (Kapitel 5.2) sowie bei den Datenbankabfragen (Kapitel 6.2).

2.3 eXtensible Markup Language (XML)

XML ist eine erweiterbare Auszeichnungssprache und ermöglicht die Strukturierung von Daten. Sie basiert auf der ISO-standardisierten Standard Generalized Markup Language (SGML). XML ist im Gegensatz zum doch recht kompliziertem SGML einfacher gestaltet und anzuwenden. Vor allem wurde Wert darauf gelegt, dass das Erstellen und Bearbeiten von XML-Dokumenten mit einfachen und weit verbreiteten Werkzeugen möglich ist und die maschinelle Verarbeitung und die Transformation von Dokumenten und Daten vereinfacht wird.
Der vom World Wide Web Consortium (W3C) [W3C 09e] im Jahr 1998 vorgeschlagene Dokumentenverarbeitungsstandard definiert eine Metamarkup-Sprache, auf deren Basis durch strukturelle und inhaltliche Einschränkungen anwendungsspezifische Sprachen definiert werden. XML an sich definiert keine Tags oder Elemente, sondern definiert nur, wie dies geschehen soll.
"Die Regeln, nach denen XML-Dokumente gebildet werden, sind einfach, aber streng. Die eine Gruppe von Regeln sorgt für Wohlgeformtheit, die andere für die Gültigkeit des Dokuments" [Vonh 02, S. 45].

Einschränkungen können durch Schemasprachen wie Document Type Definition (DTD) oder XML-Schema ausgedrückt werden. Das Markup beschreibt nur die Struktur eines Dokuments und legt nicht fest, wie das Dokument angezeigt werden soll. Mit Hilfe der eXtensible Stylesheet Language (XSL) ist es möglich, für die einzelnen Tags individuelle Darstellungsweisen festzulegen. Auf diese Weise wird eine Trennung zwischen Struktur, Inhalt und Layout erreicht. In diesem Fall verteilen sich die Angaben zu den Daten auf drei Dateien:

- Beispiel.dtd: DTD mit der Strukturbeschreibung
- Beispiel.xml: XML-Datei mit den durch Tags markierten Daten
- Beispiel.xsl: Stylesheet mit Angaben zum Rendern des Layouts

[Vorn 09, vgl. S. 155]

XML ist ein offenes System und hat den Vorteil, Daten plattformunabhängig speichern und übertragen zu können. Daher wird XML vor allem für den Austausch von Daten zwischen Computersystemen, speziell über das Internet, eingesetzt. XML-Dokumente bestehen aus einfachen Textzeichen und können mit jedem Texteditor geöffnet und bearbeitet werden. Beispiele für XML-Sprachen sind: RSS, XHTML, SVG oder auch XML-Schema.

2.3.1 Aufbau von XML-Dokumenten

Zu Beginn eines XML-Dokuments wird immer die XML-Deklaration aufgeführt, die die XML-Version und die Codierung der XML-Datei angibt. Danach folgt der eigentliche Dokumentinhalt.

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Kommentare werden mit der Zeichenfolge <!- eingeleitet und enden mit der Zeichenfolge ->.

Tags und Elemente

Alle Informationen in einem XML-Dokument bestehen aus Tags bzw. stehen zwischen Tags. Tags sind Textmarken, beginnend mit dem <-Zeichen und endend mit dem >-Zeichen. Zwischen den beiden Zeichen wird der Tagname definiert und optional weitere Attribute gesetzt. Tags dürfen außer dem Tagzeichen, dem /-Zeichen und einer beliebigen Anzahl von Attributen nichts enthalten. Der Tagname und die Attribute werden durch Whitespaces ("Neue Zeile"-, Tabulator- oder Leerzeichen) voneinander getrennt. Das System von Tags ist hierarchisch aufgebaut, d.h. Tags treten immer nur paarweise mit einer öffnenden und einer schließenden Komponente auf:

Beispiel:
öffnende Komponente:

```
<tagname ...>
```

(”tagname” steht als Platzhalter für einen beliebigen Namen; die Punkte für mögliche Attribute)

schließende Komponente:

```
</tagname>
```

Zwischen dem öffnenden und schließenden Tag steht der Inhalt, der z.B. Text enthalten kann. Öffnender Tag, Inhalt und schießender Tag bilden ein Element. Ein Element kann zusätzlich zu Texten auch weitere so genannte ”Kindelemente” beinhalten. Falls alle Elementinformationen in den Attributen enthalten sind, kann eine verkürzte Schreibweise für ein Element verwendet werden:

```
<tagname .../>
```

Hierarchische Regeln

Um eine hierarchische Struktur zu gewährleisten, müssen im Wesentlichen zwei Regeln eingehalten werden. Zum einen dürfen sich Elemente nicht überlappen, es sei denn, eines ist im anderen enthalten und zum anderen muss es immer ein Element als Hierarchiespitze geben, dass alle anderen Elemente, direkt oder indirekt, enthält. Dieses Element wird Wurzelelement (Rootelement) genannt.

Attribute

Ein Attribut stellt in XML immer eine Zuordnung von Schlüssel und Wert dar. Der Schlüssel stellt den eindeutigen Attributnamen innerhalb eines Tags dar und darf damit nur einmal darin vorkommen. Der Attributwert besteht aus Zahlen oder Textzeichen. Attributname und -wert werden durch ein =-Zeichen voneinander getrennt.

Beispiel:

```
attributname=''attributwert''
```

Typografische Regeln

Die typografischen Regeln der Namen für die Bezeichner (Attribut-, Element- oder Tagnamen) halten sich an die gängigen Vorschriften von Programmier- und Metasprachen (in Bezug auf Sonderzeichen, etc.). Sie können alle alphanumerischen Zeichen, Unterstriche, Bindestriche und Punkte enthalten, dürfen jedoch nicht mit einer Zahl oder einem Unterstrich beginnen. Die Namen lassen sich frei vergeben, solange sie wie beschrieben aufgebaut sind. Dadurch wird ein XML-Dokument auch für Menschen relativ leicht lesbar und ohne Hilfe korrekt interpretierbar. Im Gegensatz zu z.B. HTML wird hier zwischen Groß- und Kleinschreibung unterschieden. Des Weiteren müssen Attributwerte immer in Anführungszeichen (einfache oder doppelte) stehen. Alle Regeln sind in der XML-Spezifikation aufgeführt [W3C 09b].

Beispiel einer XML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<interpret>
  <name>Max Mustermann</name>
    <album>
      <titel>Album Nummer 1</titel>
      <jahr>2000</jahr>
    </album>
    <album>
      <titel>Album Nummer 2</titel>
      <jahr>2001</jahr>
    </album>
</interpret>
```

[Bade 04, vgl. S. 87 ff.]

2.3.2 XML-Parser

Mit einem XML-Parser ist es möglich, ein XML-Dokument einzulesen und die enthaltenen Informationen (also Elemente, Attribute usw.) einer Anwendung zur Verfügung zu stellen. XML-Parser unterscheiden sich in zwei Kriterien. Zum einen, ob sie validieren oder nicht, und zum anderen, welche Schnittstelle sie zum Zugriff auf das XML-Dokument anbieten, z.B. Simple API for XML (SAX) oder Document Object Model (DOM). Ein validierender Parser überprüft zusätzlich, ob die XML-Datei wohlgeformt ist, d.h. ob die grundsätzlichen Syntaxregeln eingehalten werden und bei Vorlage einer DTD, ob die XML-Datei gültig ist, d.h. ob ihr Inhalt der Strukturbeschreibung entspricht [Vorn 09, vgl. S. 155].

Simple API for XML (SAX)

SAX ist eine Programm-Schnittstelle (Application Programmers Interface (API)) für die Verarbeitung von XML-Dokumenten mit Hilfe einer objektorientierten Programmiersprache wie z.B. Java [SAX 09]. SAX ist ereignisbasiert, d.h. wenn ein bestimmtes XML-Konstrukt (z.B. ein Element) eingelesen wird, wird ein Ereignis ausgelöst. Bei Fehlern oder Warnungen wird ebenfalls ein Ereignis ausgelöst. Um die Ereignisse individuell zu behandeln, kann ein Eventhandler verwendet werden, der dann beim XML-Parser registriert wird. Dadurch kann eine Applikation je nach Anforderung auf das XML-Dokument reagieren. Da SAX die XML-Elemente nach einander in einem Eingabestrom liefert, eignet es sich besonders für sehr große XML-Dokumente. Auf "vergangene" Ereignisse kann jedoch nicht wieder zugegriffen werden. Vorteil von SAX ist, dass es schnell und einfach ist. Es hat allerdings den großen Nachteil, dass kein "Object Model" benutzt werden kann und es nicht im Speicher zur Verfügung steht [Haro 04, vgl. S. 348 ff.].

Beispiel:
Auszug aus einem XML-Dokument:

```
<beispiel>
  Dies ist ein Beispieltext.
</beispiel>
```

Folgende Ereignisse werden beim Parsen dieses Dokuments ausgelöst:

- öffnendes Element (startElement)
- Text (character)
- schließendes Element (endElement)

Document Object Model (DOM)

DOM ist ein Objektmodell und beschreibt die in einem XML-Dokument enthaltenen Elemente als Objekte für die Verarbeitung mit einer objektorientierten Programmiersprache wie z.B. Java. DOM ist ein Standard des W3C [W3C 09a]. Beim Parsen des XML-Dokuments wird ein Baum aufgebaut, der DOM-Tree. DOM liefert eine komplette Baumstruktur aller Objekte und erstellt damit eine vollständige Darstellung des XML-Dokuments im Speicher. Im Gegensatz zu SAX stellt DOM nicht nur Werkzeuge für den Zugriff, sondern auch zur Manipulation des Baumes zur Verfügung. Allerdings ist es deutlich langsamer und benötigt mehr Speicherkapazität als SAX.

Beispiel:
Folgendes XML-Dokument ergäbe den in Abbildung 2.2 dargestellten DOM-Tree:

```
<vorlesung>
  <titel>Informatik A</titel>
  <dozent>
      <vorname>Oliver</vorname>
      <name>Vornberger</name>
  </dozent>
</vorlesung>
```

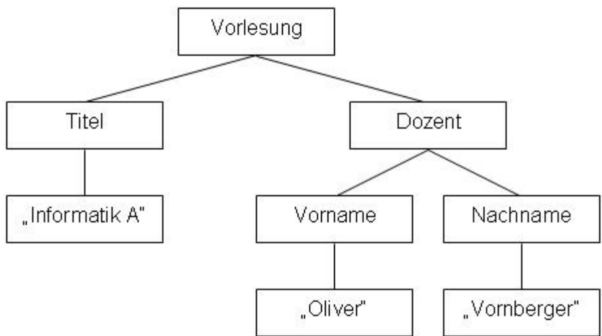


Abb. 2.2: DOM-Tree Beispiel

Die Hierarchie eines DOM-Trees wird durch Knoten (Nodes) aufgebaut. Zum Navigieren im XML-Baum und für den Zugriff auf die einzelnen Knoten werden Interfaces

zur Verfügung gestellt. Es sind sowohl generische als auch spezielle Interfaces vorhanden. Basis Interface ist das generische Interface `Node`, welches für eine Grundfunktionalität sorgt. Vereinfachte Zugriffe erlauben die speziellen Interfaces. Die Wichtigsten sind `Document`, `Element`, `Attr` und `Text`. `Document` ist der hierarchisch oberste Knoten und enthält neben dem Wurzelelement (Rootelement) alle Informationen des XML-Dokuments. `Element` entspricht den XML-Elementen, `Attr` einem Attribut eines XML-Elements. Er enthält den Namen und den Wert des Attributs. Falls ein XML-Element Text enthält, wird er als Text-Knoten dargestellt.
[Haro 04, vgl. S. 324 ff.]

Da alle Daten der DDG in Form von XML-Dateien vorliegen und XML als Weiterverarbeitungsformat zur späteren Visualisierung der Verkehrsdaten gewählt wurde, wird dieses besonders in Kapitel 3, wo die Eingangsdaten beschrieben werden, sowie beim Einlesen der Daten (Kapitel 5.2) und bei den Abfragen mittels Web Services (Kapitel 6) benötigt.

2.4 Scalable Vector Graphics (SVG)

SVG ist eine XML-basierte, textorientierte Auszeichnungssprache, die zweidimensionale, skalierbare Grafiken beschreibt. Es handelt sich dabei um einen Vektorgrafikstandard, der im Jahr 2001 vom W3C veröffentlicht wurde und sich momentan in der Version 1.1 befindet [W3C 09c].
Im Vergleich zu Rastergrafiken (z.B. JPG, PNG oder GIF), bei denen ein Bild durch eine feste Anzahl von Pixeln dargestellt wird (rasterförmige Anordnung), denen jeweils ein Farbwert zugeordnet ist, wird bei Vektorgrafiken ein Bild durch eine Anzahl von geometrischen Primitiven, wie Linien, Rechtecken, Kurven, Kreisen, Polygonen oder allgemeinen Kurven (Splines) beschrieben. Es wird also nicht das Bild "an sich" gespeichert, sondern nur die Anweisungen, wie das Bild zu zeichnen ist. Für eine Linie würden z.B. nicht alle Linienpunkte gespeichert, sondern lediglich Start- und Endpunkt, welche verbunden werden sollen [Zepp 04, vgl. S. 425 ff.].
Da SVG eine Untermenge von XML darstellt, lassen sich textbasiert Grafiken erzeugen, die einfache Grundformen, aber auch Füllungen mit Farbverläufen oder gar aufwendige Filtereffekte beinhalten können. Des Weiteren können animierte Grafiken durch Definition von Ereignissen, die z.B. Farbe, Größe oder Position eines Objekts verändern, erstellt werden oder Elemente einer Grafik mit Hilfe einer Skriptsprache (z.B. JavaScript oder ECMAScript) einfach manipuliert werden.
Vektorgrafiken haben den Vorteil, dass sie ohne Qualitätsverlust (keine so genannte "Treppenbildung") skalierbar sind. Außerdem können sie leicht bearbeitet werden, evtl. vorhandene Texte lassen sich maschinell durchsuchen und die Dateigröße ist unabhängig von der Auflösung [Watt 03, vgl. S. 9 ff.]. Sie kommen immer dann zum Einsatz, wenn Grafiken digital gezeichnet und verändert (z.B. Logos, Konstruktionspläne, usw.) oder wenn vorhandene Rastergrafiken um geometrische Primitive erweitert werden (z.B. Texte oder Markierungen).
SVG Grafiken bestehen wie XML aus ASCII-Code und können mit jedem beliebigen Texteditor erstellt werden. Sie werden noch nicht von allen gängigen Browsern unter-

stützt, können aber ggf. über Plug-Ins, aber auch eigenständigen SVG-Viewern oder SVG-Editoren angezeigt werden.

2.4.1 Aufbau einer SVG-Datei

Benutzt wird die Syntax der eXtensible Markup Language (XML). Zunächst wird das SVG-Dokument mit der XML-Deklaration eingeleitet. Danach folgt die DTD, die den benutzten Namensraum beschreibt, indem ein Verweis auf die entsprechende DTD-Datei eingefügt wird [Watt 03, vgl. S. 52 ff.].
Jedes SVG-Dokument beinhaltet als Wurzelelement das `<svg>`-Element, welches alle anderen Elemente enthält. Hier werden in der Regel auch wichtige Attribute gesetzt. Um Namenskonflikte bei gleichzeitiger Verwendung mehrerer XML-Sprachen innerhalb eines Dokuments zu vermeiden, kann das `xmlns`-Attribut gesetzt werden, welches den Namensraum des Dokuments definiert. Der Namensraum eines validen SVG-Dokuments der Version 1.1 lautet `http://www.w3.org/2000/svg`. Da SVG selbst einige Attribute der XML Linking Language (XLink) [W3C 09d] verwendet, muss dann noch ein weiterer Namensraum festgelegt werden und zwar wird dem Attribut `xmlns:xlink` der Wert `http://www.w3.org/1999/xlink` zugewiesen. [Watt 03, vgl. S. 57 ff.] Optional können hier viele weitere Attribute gesetzt werden, z.B. `width` und `height`, mit denen die Größe der SVG-Grafik festgelegt wird.

Beispiel:
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="500" height="500">

  <!-- Hier steht der Dokumentinhalt -->

</svg>
```

SVG bietet viele weitere Möglichkeiten, wie z.B. das `defs`- oder das `use`-Element. Auf diese soll im Rahmen der Arbeit jedoch nicht näher eingegangen werden, da sie nicht verwendet wurden.

2.4.2 Attribut `viewBox`

In SVG wird ein zweidimensionales kartesisches Koordinatensystem verwendet. Der Ursprung des Koordinatensystems liegt in der linken oberen Ecke, die x-Achse verläuft nach rechts, die y-Achse nach unten. Die Koordinaten werden in Gleitkomma-Zahlen angegeben. Das Attribut `viewBox` definiert innerhalb der SVG-Grafik ein neues Koordinatensystem, indem ein Rechteck aufgespannt wird. Dies erfolgt durch die Angabe der x- bzw. y-Koordinate der linken oberen Ecke des Rechtecks, sowie durch Angabe von Breite und Höhe des Rechtecks. Die eigentliche Funktion des Attributs `viewBox` besteht

darin, dass die Grafik automatisch transformiert wird. Der Teil der SVG-Grafik, der innerhalb des neuen Koordinatensystems liegt, wird beim Rendern automatisch auf die im `<svg>`-Element angegebene Größe der Grafik skaliert [Watt 03, vgl. S. 65 ff.].

2.4.3 Elemente

SVG erlaubt drei verschiedene Arten von Elementen:

- Vektorgrafiken, bestehend aus geometrischen Primitiven
- Text
- Rastergrafiken (z.B. BMP oder PNG)

[Watt 03, vgl. S. 8]

Einbindung von grafischen Primitiven

`<line>`-Element
Das `<line>`-Element repräsentiert eine einfache gerade Linie, durch Angabe der Koordinaten der beiden Endpunkte. Mit dem Attribut `stroke` kann der Linie eine Farbe zugeordnet werden.

```
<line x1="20" y1="10" x2="70" y2="60" stroke="black"/>
```

siehe Abbildung 2.3a

`<rect>`-Element
Ein einfaches Rechteck wird durch die Position (x- und y-Koordinate) der linken oberen Ecke, sowie durch Breite und Höhe (`width` und `height`) mittels des `<rect>`-Elementes spezifiziert. Mit dem Attribut `fill` kann zudem eine Füllfarbe gesetzt werden.

```
<rect x="90" y="10" width="70" height="50" stroke="darkred" fill="red"/>
```

siehe Abbildung 2.3b

`<path>`-Element
Das `<path>`-Element ist das mächtigste Element der grafischen Primitive. Es lassen sich komplexe Formen, Animationsstrecken oder Maskenregionen erstellen. Es enthält ein Attribut `d`, welches analog zum Attribut `points` bei den Polylinien oder Polygonen Stützpunkte enthält. Darüber hinaus können hier aber zusätzlich verschiedene Kommandos angegeben werden, die die Regeln bestimmen, nach denen die Punkte zu interpretieren sind. Folgende Kommandos können verwendet werden:

- `moveTo` (`M`) - setzt den Startpunkt des Pfades
- `lineTo` (`L`, `H` horizontal oder `V` vertikal) - zeichnet eine Linie zum angegebenen Punkt
- `curveTo` (`C`, `S` "Shorthand/Smooth", `Q` quadratisch oder `T` "Shorthand/Smooth" quadratisch) - zeichnet eine Bézier-Kurve

- `arc` (`A`) - zeichnet einen elliptischen oder kreisförmigen Bogen
- `closePath` (`Z`) - schließt einen Pfad

Beispiel einer Bézier-Kurve mit dem `curveTo`-Befehl:

```
<path d="M20 250 C20 200 150 300 150 250" stroke="blue" fill="none"/>
```

siehe Abbildung 2.3g
[Watt 03, vgl. S. 231 ff.]

Des Weiteren werden die Primitive `<circle>` (Kreis siehe Abbildung 2.3c), `<ellipse>` (Ellipse siehe Abbildung 2.3d), `<polyline>` (Polylinie siehe Abbildung 2.3e) sowie `<polygon>` (Polygon siehe Abbildung 2.3f) unterstützt. Da sie aber in dieser Arbeit keine Anwendung finden, soll darauf nicht näher eingegangen werden.

Einbindung von Text

In SVG kann Text nicht nur als grafisches- sondern auch als Text-Element eingefügt werden, wodurch dieser z.B. in die Zwischenablage kopiert oder nach Wörtern durchsucht werden kann. Dadurch können Suchroboter SVG-Grafiken im Internet für Web-Kataloge indizieren. Auszugebender Text wird von den Elementen `<text>` und `</text>` umgeben und durch die Attribute `x` und `y` positioniert. Die Höhe und Breite des Textes hängt von der Schriftart bzw. -größe und anderen Formatierungen ab. Texteigenschaften werden mittels CSS definiert.

```
<text x="20" y="300" style="font-size:10px;font-family:Comic Sans MS;">
  Dies ist ein Beispieltext!
</text>
```

siehe Abbildung 2.3i

Einbindung von Rastergrafiken

Um Rastergrafiken einzubinden, bedient man sich des `<image>`-Elements. Das Bild wird über das Attribut `xlink:href` verknüpft, und die Position durch die Attribute `x` und `y` sowie die Größe durch `width` und `height` gesetzt.

```
<image x="20" y="320" width="32" height="32" xlink:href="smile.png"/>
```

siehe Abbildung 2.3h

In dieser Arbeit wurde das SVG-Format u.a. als mögliches Rückgabeformat der Abfragen festgelegt. Daher findet es in Kapitel 6, ”Web Services“, Verwendung.

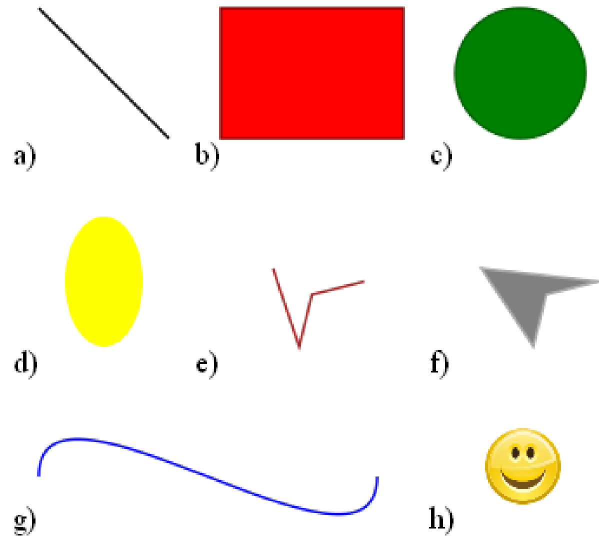


Abb. 2.3: SVG Beispiele

2.5 Java

Java ist eine objektorientierte Programmiersprache, die sich durch einige zentrale Eigenschaften auszeichnet. Entwickler der Sprache ist das Unternehmen "Sun Microsystems", welches die Sprache folgendermaßen beschreibt:

"Java: Eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, hochleistungsfähige, Multithread-fähige und dynamische Sprache" [Flan 98, S. 3].

Diese Eigenschaften machen Java universell einsetzbar und durch das objektorientierte Konzept ist es möglich, moderne und wiederverwendbare Softwarekomponenten zu entwickeln. Im Gegensatz zu herkömmlichen Programmiersprachen, bei denen Maschinencode für eine spezielle Plattform bzw. einen bestimmten Prozessor generiert wird, erzeugt der Java-Compiler plattformunabhängigen Bytecode. Um diesen Bytecode interpretieren zu können, wird eine virtuelle Maschine, die Java Virtual Machine (JVM) benötigt. Diese führt die Programme aus, indem sie den Bytecode interpretiert und bei Bedarf kompiliert [Ulle 09, vgl. S. 55. ff.].

Java-Servlets

Java-Servlets sind Java-Klassen, deren Instanzen innerhalb eines Webservers Anfragen von Clients entgegennehmen und beantworten. Im Gegensatz zu Java-Programmen, die als normale Applikationen auf der Serverseite genutzt werden, liegen Java-Servlets im Kontext des Webservers. Dadurch muss der Webserver (bei Ausführung dieser) nicht erst die JVM aufrufen, bevor das Java-Programm bzw. das Java-Servlet aufgerufen

```
package servlets;

import java.io.IOException;
import javax.servlet.http.*;

public class BeispielServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException
    {
        res.getWriter().println("<html>");
        res.getWriter().println("    <head>");
        res.getWriter().println("        <title>Beispiel Servlet</title>");
        res.getWriter().println("    </head>");
        res.getWriter().println("    <body>");
        res.getWriter().println("        <h1>Dies ist ein Beispiel Servlet</h1>");
        res.getWriter().println("    </body>");
        res.getWriter().println("</html>");
    }
}
```

Abb. 2.4: Beispiel einer einfachen Java-Servlet Klasse

wird. Die Laufzeit wird dadurch verbessert, dass der Webserver eine JVM integriert, die immer läuft, und Objekte einzelne Verbindungen innerhalb der Java-Maschine bedienen (Vergleichbar mit Java-Applets: Ein Java-Applet ist ein "Java-Programm" auf der Clientseite (im Browser), während ein Java-Servlet ein "Java-Programm" auf der Serverseite ist) [Ulle 09, vgl. S. 1231].

Java-Servlets können sowohl Parameter der Anfrage als auch Sitzungsdaten verwenden und Antworten in verschiedenster Form, z.B. als Text (u.a. HTML und XML) oder auch als Bild, zurückliefern. Der Inhalt der Antworten kann dabei dynamisch, also im Moment der Anfrage, erstellt werden und muss nicht bereits statisch (etwa in Form einer HTML-Seite) für den Webserver verfügbar sein. Sie werden oft im Rahmen der Java-EE-Spezifikation [SUN 09a] nach dem "Model-View-Controller (MVC) Pattern" verwendet: Java Server Pages (JSP) repräsentieren die View, Frameworks vervollständigen das MVC-Muster. Der Web-Container (die Laufzeitumgebung) erzeugt bei Bedarf eine Instanz des Java-Servlets. Er kommuniziert mit dem Webserver oder ist bereits integraler Bestandteil dessen.

Im Folgenden wird ein Beispiel für die Implementierung einer dynamischen Webseite unter Verwendung der Servlet-Spezifikation und einer Web-Container-Umgebung (z.B. Apache Tomcat) gegeben. Die Java-Servlet-Klassen müssen immer die Schnittstelle `javax.servlet.Servlet` oder eine davon abgeleitete Klasse implementieren. Häufig wird eine Klasse erstellt, die von der Klasse `javax.servlet.http.HttpServlet` abgeleitet wird, welche wiederum `javax.servlet.Servlet` implementiert. Eine oder beide Methoden `doGet` und `doPost` der Superklasse werden überschrieben, um die beiden wichtigsten HTTP-Methoden `GET` und `POST` verarbeiten zu können. Es kann aber auch nur die Methode `service` überschrieben werden, die unabhängig vom HTTP-Befehl aufgerufen wird.

In der XML-Datei "web.xml" (siehe Abbildung 2.5), dem sogenannten "Deployment Des-

?=? xml	version="1.0" encoding="UTF-8"
web-app	
id	WebApp_ID
version	2.4
xmlns	http://java.sun.com/xml/ns/j2ee
xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation	http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
display-name	BeispielServlet
servlet	
description	Beispiel für ein Servlet
display-name	BeispielServlet
servlet-name	BeispielServlet
servlet-class	servlets.BeispielServlet
servlet-mapping	
servlet-name	BeispielServlet
url-pattern	/BeispielServlet
welcome-file-list	
welcome-file	index.html

Abb. 2.5: Beispiel eines "Deployment Descriptor": XML-Datei "web.xml"

criptor", werden Metainformationen über das Java-Servlet hinterlegt. Diese XML-Datei wird zusammen mit den kompilierten Klassen in einer Archiv-Datei zusammengefügt, welches dem Web-Container zur Verfügung gestellt wird. Der Aufruf des Java-Servlets erfolgt dann über die im "Deployment Descriptor" spezifizierten Angaben.

Beispielaufruf eines Java-Servlets:

```
http://Serveradresse/BeispielArchiv/BeispielServlet
(http://Serveradresse/Name_des_Web-Archives/Servlet_Name)
```

Zusätzlich können etwaige Parameter übergeben werden, indem der URL Angaben folgender Form hinzugefügt werden:

```
...?Param1=Wert1\&Param2=Wert2...
```

Diese können dann innerhalb des Java-Servlets ausgelesen werden. An die Methode `doGet` werden zwei Objekte übergeben, `HttpServletRequest` "request" sowie `HttpServletResponse` "response". Die übergebenen Parameter können über die Funktion `getParameter` aus "request" erhalten werden:

```
String Param1 = request.getParameter("Param1");
```

Im Gegensatz zu "request" ist "response" für die Rückgabe zuständig. Damit der Browser "weiß", wie mit den zurückgegebenen Daten umzugehen ist, kann das Rückgabeformat mit der Funktion `setContentType` z.B. auf "text/xml" für die Rückgabe im XML-Format gesetzt werden. Mittels der Funktion `getWriter` erhält man einen `PrintWriter`, mit dem dann die Rückgabedaten geschrieben werden können:

```
response.setContentType("text/xml");
PrintWriter out = response.getWriter();
out.print(<?xml version="1.0" encoding="UTF-8"?>);
out.print(    ...    );
out.print(XML Inhalt);
out.print(    ...    );
```

```
out.flush();
out.close();
```

Das im Rahmen dieser Arbeit erstellte Programm wurde in Java implementiert (Kapitel 5); die Abfragen verwenden Java-Servlets (Kapitel 6).

2.6 Apache Tomcat

Tomcat ist ein Open Source-Container für Java-basierte Web-Anwendungen und stellt eine Referenzimplementierung der Java-Servlet und JSP-Spezifikation [SUN 09b] dar. Es wurde von der "Jakarta Projektgruppe" entwickelt, welche von der "Apache Software Foundation" unterstützt wird. Die bereitgestellte Umgebung, ein Java-Servlet-Container ("Jetty") mit JSP-Engine ("Jasper"), ermöglicht die Ausführung von Java-Servlet- und JSP-Web-Anwendungen. Zusätzlich bietet Tomcat weitere Funktionen, z.B. Authentifizierungs- und Authorisierungsmöglichkeiten über sogenannte "Realms" (geschützte Bereiche), sowie einen internen Verzeichnisdienst und eröffnet die Möglichkeit gepoolte Datenquellen im Container zu hinterlegen, die von den darin laufenden Applikationen verwendet werden können. Um die Funktionalität von Tomcat mit der eines Apache zu verbinden (PHP, Perl, ...), werden sogenannte "Connector PlugIns" verwendet [Apache 09].

Um das Abfragen der Informationen aus der Datenbank mittels Web Services zu realisieren, kommt Tomcat zum Einsatz (siehe Kapitel 6).

3 Aufbau der Eingangsdaten

3.1 Grundsätzlicher Aufbau

Die Eingangsdaten der DDG unterscheiden sich in Basisdaten, die das Verkehrsnetz Deutschlands beschreiben, sowie in Verkehrsdaten, die für Straßenabschnitte dieses Netzes Verkehrsdichten zu bestimmten Zeitpunkten angeben. Zur Beschreibung von Straßen bzw. Straßenabschnitten verwendet die DDG Geocodes.

3.2 Geocodes

Geocodes dienen der Referenzierung von verkehrstelematisch wichtigen Objekten und Flächen, wodurch für Europa ein einheitlicher Ortsbezug hergestellt wird. Dadurch können geocodierte eindeutige Ortsbeschreibungen von Verkehrsmeldungen vom Benutzer oder in Telematik-Endgeräten entschlüsselt werden. Wichtige verkehrstelematische Objekte, wie z.B. Kreuzungspunkte von Bundesautobahnen oder Bundesstraßen, Verwaltungsgebiete oder "Points of Interest" werden anhand eines Geocodes geocodiert, indem aus den geographischen Koordinaten eines zentralen Punktes des Objektes der Geocode berechnet wird. Die Verkehrsinformationen werden dann auf diesen Punkt referenziert. Auf diese Weise ist es möglich, mit Hilfe der Geocodes die Verkehrsinformationen wieder zu decodieren und einen Ortsbezug herzustellen.
Alternativ besteht die Möglichkeit der Referenzierung durch Traffic Message Channel (TMC)-Locationcodes, die von jedem Land für Gebiete, Punkte oder linearen Ortungen vergeben werden. Im Gegensatz zu den Geocodes, die eine einheitliche Ortsreferenzierung für Europa darstellen, sind TMC-Locationcodes jedoch länderspezifisch [DDG 03, vgl. S. 5].
Die Schnittstelle zu den Benutzern stellt die sogenannte Endgerätetabelle (EGT) dar, in der alle Geocodes aufgelistet sind.

3.3 Aufbau der Basisdaten

Die Basisdaten werden in Form einer XML-Datei zur Verfügung gestellt, die die EGT-Daten enthält. Diese Daten umfassen die erzeugten Geocodes zusammen mit Zusatzinformationen zu den durch die Geocodes referenzierten Orten. Aufgelistet werden alle topographischen Objekte, welche für verkehrstelematische Dienste relevant sein können.

Die Einträge in der EGT werden in drei logische Gruppen eingeteilt:

- Punktobjekte
  Straßenknoten (Geocodetypen 1, 2, 3, 5, 6, 7, 8, 30, 31, 32, 33, 34, 35), "Points of Interest" (Geocodetyp 21) sowie "Straßen-Points of Interest" (Geocodetyp 22) und Richtungsorte (Geocodetyp 23)
- Flächenobjekte
  Verwaltungsgebietseinheiten (Geocodetypen 11, 12, 13, 14, 17, 18, 19) sowie sonstige Gebiete (Geocodetypen 16, 20)
- Linienobjekte
  Straßenabschnitte (Geocodetyp 92), Umleitungsstrecken (Geocodetyp 94), Bundesstraßen (Geocodetyp 98) und Autobahnen (Geocodetyp 99)

[DDG 03, vgl. S. 6]

Im Falle der Straßenknoten können mehrere Einträge den gleichen Geocode besitzen, da sie mehreren Straßen zugeordnet sein können. Eindeutig bestimmt werden, können sie nur in Verbindung mit dem Attribut Straße.

Folgende Attribute bilden die Struktur der EGT:

- `EGT_ID` - beschreibt die fortlaufende Nummer eines Eintrages in der EGT
- `EGT_GEOCODE`* - jedes für verkehrstelematische Dienste relevante topographische Objekt erhält einen Geocode
- `EGT_LFDNR` - die laufende Nummer innerhalb einer Strecke
- `EGT_TYP`* - gibt die Art des Geocodes an (von 0 bis 255)
- `EGT_STRASSE` - die offizielle Bezeichnung der Straße (Kürzel des Straßentyps + Straßennummer z.B. "A1")
- `EGT_NAME`* - die offizielle Benennung des kodierten Objekts
- `EGT_REFGEOCODE` - werden nur bei Nicht-Autobahnknoten, den Verwaltungsgebieten sowie den "Points of Interest" vergeben und dienen einer besseren Beschreibung der kodierten Objekte
- `EGT_NUMMERRICHTUNGNACHORT` - enthält bei Geocodes von Straßenknoten die offiziell für diesen Knoten vergebene Nummer
- `EGT_NUMMERRICHTUNGVONORT` - da diese Nummern von der Fahrtrichtung auf der jeweiligen Straße abhängen können, sind zwei Spalten für die Nummern vorgesehen
- `EGT_VONORT` - die Festlegung der Richtungsorte richtet sich bei Autobahnstrecken nach der vor Ort vorhandenen Beschilderung, bei den untergeordneten Straßen werden als Fernziele / Richtungsorte primär Benennungen von überregional bekannten Ortschaften gewählt
- `EGT_NACHORT` - siehe `EGT_VONORT`
- `EGT_LAENGE1` - die x-Koordinate der Straßenknoten (für den Geocodetyp 92 (Straßenabschnitte) beschreiben `EGT_LAENGE1` und `EGT_BREITE1` den Startpunkt und `EGT_LAENGE2` und `EGT_BREITE2` den Endpunkt des Straßenabschnitts)

- `EGT_BREITE1` - die y-Koordinate der Straßenknoten
- `EGT_LAENGE2` - siehe `EGT_LAENGE1`
- `EGT_BREITE2` - siehe `EGT_LAENGE1`
- `EGT_STRASSEINT` - die internationale numerische Bezeichnung der Straße (Kürzel des Straßentyps (E) + Straßennummer)
- `EGT_NAMEKURZ` - abgekürzter Name (außer für Objekte der Geocodetypen 92, 98 und 99)
- `EGT_VORGAENGER` - bei Straßenknoten werden hier die Geocodes der Vorgänger- bzw. Nachfolger-Knoten angegeben
- `EGT_NACHFOLGER` - siehe `EGT_NACHFOLGER`

(* - Attribut muss bei allen Einträgen in der Endgerätetabelle gesetzt sein)
[DDG 03, vgl. S. 6 ff.]

Die Belegung der einzelnen Attribute ist abhängig vom Geocodetyp. Die Länge der einzelnen Einträge in den Feldern der Tabelle können mit Ausnahme des Feldes abgekürzter Name (`EGT_NAMEKURZ`) beliebig lang sein. Der Primärschlüssel der EGT setzt sich aus den Attributen Geocode (`EGT_GEOCODE`) und Straße (`EGT_STRASSE`) zusammen. Durch die Angabe von Vorgänger- bzw. Nachfolger-Knoten bei Straßenknoten lassen sich die Straßen Deutschlands abbilden. Das Feld Vorgänger-Knoten `EGT_VORGAENGER` wird nicht belegt, wenn der Knoten den Straßenanfang darstellt, also den ersten Knoten einer Straße oder wenn es sich um den ersten Knoten nach einer Unterbrechung der Straße handelt. Analog dazu wird das Feld Nachfolger-Knoten `EGT_NACHFOLGER` nicht belegt, wenn es sich um den letzten Knoten einer Straße, also dem Straßenende oder um den letzten Knoten vor einer Unterbrechung der Straße handelt. Der Knoten sowie die für diesen Knoten angegebenen Vorgänger- bzw. Nachfolger-Knoten gehören generell zur gleichen Straße.
Einen Sonderfall bei der Festlegung von Vorgänger-Nachfolger-Relationen nehmen Ringstraßen ein. Dabei sind zwei Fälle zu unterscheiden. Zum einen Straßen, die auf einem Teilstück einen getrennten Verlauf haben, d.h. sich an einem Knoten in zwei gleichwertige Verzweigungen teilen und sich später wieder zu einem gemeinsamen Verlauf vereinigen (z.B. Ringstraßen um Stadtzentren, siehe Abbildung 3.1) und zum anderen Straßen, die durch ihren gesamten Verlauf einen in sich geschlossenen Ring bilden (siehe Abbildung 3.2). Im ersten Fall werden die Vorgänger- und Nachfolgerknoten, wie in Tabelle 3.1 dargestellt, angegeben. Im Gegensatz dazu haben geschlossene Ringstraßen keinen Anfangs- bzw. Endknoten und werden nach dem Schema in Tabelle 3.2 festgelegt [DDG 03, vgl. S. 16 ff.].

3.4 Aufbau der Verkehrsdichtedaten

Die Verkehrsdichtedaten der DDG, die in diesem Projekt zum Einsatz kommen, basieren auf den Rohdaten, die für die Datengenerierung der Traffic Map Live (TML)

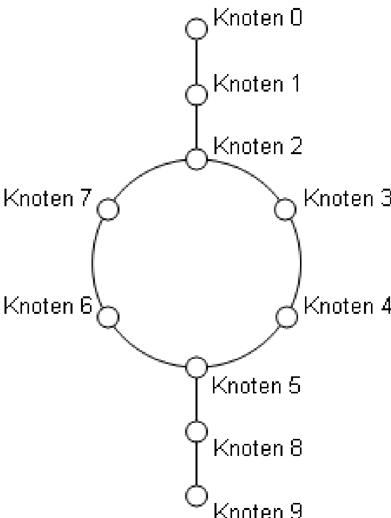


Abb. 3.1: Getrennter Verlauf auf einem Straßenteilstück

Knoten	Vorgänger	Nachfolger
Knoten 1	Knoten 0	Knoten 2
Knoten 2	Knoten 1	Knoten 3
Knoten 3	Knoten 2	Knoten 4
Knoten 4	Knoten 3	Knoten 5
Knoten 5	Knoten 4	Knoten 8
Knoten 6	Knoten 7	Knoten 5
Knoten 7	Knoten 2	Knoten 6
Knoten 8	Knoten 5	Knoten 9

Tabelle 3.1: Vorgänger- und Nachfolgerknotenfestlegung bei getrenntem Verlauf auf einem Straßenteilstück

benutzt werden. TML bezeichnet ein Produkt, bei dem Verkehrsereignisse als animierter Verkehrsfilm dargestellt werden. Die zugrundeliegenden Rohdaten werden von diversen Systemen geliefert (siehe Kapitel 1.1) und mittels spezieller Software in das XML-Format konvertiert [DDG 04, vgl. S. 3]. Die einzelnen Segmente, für die Verkehrsdichten angegeben werden, beziehen sich auf logische Streckenabschnitte. Ein logischer Streckenabschnitt bezeichnet die gerichtete Strecke zwischen zwei Straßenknoten und den Weg durch den Knoten, der die Strecke in stromaufwärtiger Richtung begrenzt.

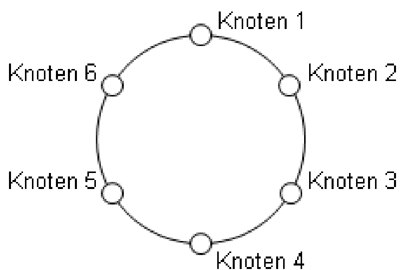
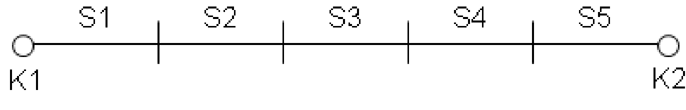


Abb. 3.2: Geschlossene Ringstraße

Knoten	Vorgänger	Nachfolger
Knoten 1	Knoten 6	Knoten 2
Knoten 2	Knoten 1	Knoten 3
Knoten 3	Knoten 2	Knoten 4
Knoten 4	Knoten 3	Knoten 5
Knoten 5	Knoten 4	Knoten 6
Knoten 6	Knoten 5	Knoten 1

Tabelle 3.2: Vorgänger- und Nachfolgerknotenfestlegung bei geschlossenen Ringstraßen

Alle Segmente haben eine einheitliche Länge von 200 Metern und sind Bestandteil eines logischen Streckenabschnitts, wenn der Anfang des Segments innerhalb des Streckenabschnitts liegt. Alle Segmente eines Streckenabschnitts werden fortlaufend, beginnend mit Eins, nummeriert [DDG 04, vgl. S. 5 ff.].



K1, K2: Straßenknoten
S1, ... , S5: Segmente der Strecke $\overline{K1\ K2}$

Abb. 3.3: Beispiel für den Aufbau eines logischen Streckenabschnitts mit fünf Segmenten

Logische Streckenabschnitte, für die Verkehrsinformationen vorliegen, werden in der XML-Datei aufgelistet. Die Struktur gliedert sich dabei wie folgt:

Zum einen werden allgemeine Angaben zum Streckenabschnitt gemacht:

- `ContendProvider` - beinhaltet einen kundenspezifischen Code für Content-Provider
- `TmcVersion` - die TMC Version
- `Country` - das Land
- `RoadName` - der Straßenname des Streckenabschnitts
- `NodeFromGeoCode` - der Geocode des Start-Straßenknotens
- `NodeToGeoCode` - der Geocode des End-Straßenknotens
- `NodeFromTmcCode` - der TMC-Code des Start-Straßenknotens
- `NodeToTmcCode` - der TMC-Code des End-Straßenknotens
- `NodeFromName` - der Name des Start-Straßenknotens
- `NodeToName` - der Name des End-Straßenknotens
- `DirectionFromName` - Bezeichnung der Richtung streckenaufwärts
- `DirectionToName` - Bezeichnung der Richtung streckenabwärts

- `StateFromName` - Bundesland, in dem der Start-Straßenknoten liegt
- `StateToName` - Bundesland, in dem der End-Straßenknoten liegt
- `VehicleClass` - Fahrzeugklasse, für die die Daten erhoben wurden
- `SegmentAmount` - Anzahl der Segmente des Straßenabschnitts
- `SegmentValueAmount` - Anzahl der Segmente, für die Verkehrsdichten vorliegen
- `DateOfGeneration` - Datenerhebungszeitpunkt
- `DateOfService` - Zeitpunkt, ab dem die Verkehrsdichteangaben gültig sind
- `ValidityOfService` - Dauer (in Sekunden) der Gültigkeit der Verkehrsdichteangaben

[DDG 04, vgl. S. 7]

Des Weiteren werden mittels des Attributs `ValueList` Verkehrsdichten für einzelne Segmente angegeben. Dazu beinhaltet `ValueList` für jedes Segment, für das Daten vorliegen, ein "Kindelement" `ValuePair`, welches die Segmentnummer (`SegmentID`) sowie einen Segmentwert (`SegmentValue`) enthält.
Hier kann eines der folgenden Attribute gesetzt werden:

- Fahrzeugdichte (`CarDensity`)
- Reisezeit (`TravelTimeValue`)
- Reisegeschwindigkeit, diskretisierter Wert (`SpeedInterval`)
- Reisegeschwindigkeit, nicht-diskretisierter Wert (`TravelSpeedValue`)
- `Level Of Service`

[DDG 04, vgl. S. 6]

Die Daten, die im Rahmen dieses Projektes zum Einsatz kommen, verwenden die Fahrzeugdichte. Die Fahrzeugdichte (D) kann Werte von "0" bis "255" annehmen. Die Klassifikation kann der Tabelle 3.3 entnommen werden.

Verkehrsstatus	Dichteangabe (D)
freie Fahrt:	$0 <= D < 10$
dichter Verkehr:	$10 <= D < 25$
stockender Verkehr:	$25 <= D < 35$
Stau:	$35 <= D < 255$

Tabelle 3.3: Klassifikation der Fahrzeugdichte

Für Segmente, die nicht aufgelistet werden, gilt "freie Fahrt" [DDG 04, vgl. S. 8].

Ein kurzes, vereinfachtes Beispiel einer XML-Datei für Verkehrsdichtedaten sieht wie folgt aus:

```
<MapElementTML>
  <RoadName>A7</RoadName>
  <NodeFromGeoCode>4110677028</NodeFromGeoCode>
  <NodeToGeoCode>4119589924</NodeToGeoCode>
  <SegmentAmount>5</SegmentAmount>
  <SegmentValueAmount>3</SegmentValueAmount>
  <ValueList>
    <ValuePair>
      <SegmentId>2</SegmentId>
      <SegmentValue>
        <CarDensity>14</CarDensity>
      </SegmentValue>
    </ValuePair>
    <ValuePair>
      <SegmentId>3</SegmentId>
      <SegmentValue>
        <CarDensity>17</CarDensity>
      </SegmentValue>
    </ValuePair>
    <ValuePair>
      <SegmentId>4</SegmentId>
      <SegmentValue>
        <CarDensity>15</CarDensity>
      </SegmentValue>
    </ValuePair>
  </ValueList>
  <DateOfGeneration>2004-08-13T09:41:15</DateOfGeneration>
  <DateOfService>2004-02-22T22:28:00</DateOfService>
</MapElementTML>
```

4 Datenbank

Als Datenbank wurde PostGIS ausgewählt. Im Folgenden wird PostGIS zunächst vorgestellt und grundlegende Eigenschaften sowie Funktionen und Operatoren näher erläutert. Im Anschluss daran wird die in dieser Arbeit erstellte Datenbank, d.h. das Konzept und die Grundstruktur beschrieben. Die Datenbankstruktur ist in einem Entity-Relationship-Modell (ER-Modell) in Abbildung 4.1 dargestellt.

4.1 PostGIS

PostGIS ist eine räumliche Erweiterung für die objektrelationale Datenbank PostgreSQL [PostgreS 09] und erweitert diese um die Unterstützung räumlicher Objekte, wodurch Postgre-SQL als räumliches Datenbank-Backend für Geografische Informations-Systeme (GIS) eingesetzt werden kann [Mitc 08, vgl. S. 311 ff.]. Es wurde von "Refractions Research" entwickelt und ist ebenfalls wie PostgreSQL ein Open Source-Projekt. PostGIS unterliegt der "GNU General Public Licence" [GNU 09] und implementiert die "OpenGIS Simple Features For SQL"-Spezifikationen [OGC 09b] des "Open Geospatial Consortium (OGC)" [OGC 09a]. In Verbindung mit der PostGIS-Erweiterung stellt PostgreSQL ein leistungsstarkes Instrument zur Speicherung und Verwaltung von Geodaten dar. Der UMN MapServer, einer der schnellsten Web Map Services der Welt, unterstützt die Einbindung von PostGIS-Daten [OGC 09c, vgl. S. 11].

PostGIS ermöglicht es, räumliche Daten sowohl zu speichern und abzufragen, als auch zu bearbeiten. Deshalb handelt es sich also nicht nur um einen reinen Datenspeicher, sondern auch um eine Umgebung für die Analyse räumlicher Daten. Großer Vorteil von PostGIS ist die Einhaltung der genannten Standards, wodurch eine Interoperabilität räumlicher Daten erreicht wird und mit Hilfe von standardisierten Funktionen gearbeitet werden kann, ohne auf Umwege und Einschränkungen angewiesen zu sein [Mitc 08, vgl. S. 313]. Des Weiteren können die Geodaten der Datenbank mit denen anderer OGC-standardisierten Datenbanken beliebig kombiniert werden.

4.1.1 GIS Objekte

PostGIS unterstützt alle Objekte, die in der "Simple Features For SQL"-Spezifikation des OGC enthalten sind. Dieser Standard wird um die Unterstützung von `3DZ`-, `3DM`- und `4D`-Koordinaten erweitert.

Die folgenden Geometrietypen können verwendet werden:

- OpenGIS Well-Known Text (WKT)/Binary (WKB):
  z.B. Point, Linestring, Polygon, Multipoint, Multilinestring, Multipolygon und Geometrycollection

- Extended Well-Known Text (EWKT)/Binary (EWKB):
  erweitert den OpenGIS Well-Known Text (WKT)/Binary (WKB) um Höheninformationen und/oder Messwerte

- SQL/Multimedia and Application Packages (MM):
  Oberbegriff für einige Erweiterungen des SQL-Standards, u.a. Circularstring, Curvepolygon, Multicurve

[OGC 09c, vgl. S. 11 ff.]

Geometrien können entweder im Textformat oder im binären Format vorliegen. Vorteil des Textformats ist es, dass mittels SQL-Statements Geometrien erzeugt werden können. Zur Veranschaulichung sind im Folgenden die WKT-Repräsentationen für einen einfachen zweidimensionalen Punkt (`Point`), einem Linienzug (`LineString`) und einem Polygon (`Polygon`) angegeben:

- `POINT (10 10)`
- `LINESTRING (10 10, 20 20, 30 30)`
- `POLYGON (10 10, 10 20, 20 20, 20 15, 10 10)`

Dies sind die drei Basis-Geometrien, von denen weitere Geometrietypen abgeleitet werden: u.a. `MultiPoint`, `MultiLineString` und `MultiPolygon` sowie die `GeometryCollection`, die eine Kombination der anderen Geometrietypen darstellt:

```
GEOMETRYCOLLECTION (
  POINT (10 10)
  LINESTRING (10 10, 20 20, 30 30)
  POLYGON (10 10, 10 20, 20 20, 20 15, 10 10)
)
```

[OGC 09c, vgl. S. 11 ff.]

4.1.2 Grundlegende Funktionen und Operatoren

PostGIS stellt insgesamt über 300 Funktionen zur Verfügung. Diese umfassen räumliche Funktionen (z.B. zur Flächen-, Distanz- oder Verschneidungsberechnung), räumliche Operatoren (z.B. `Overlaps` oder `Contains`) und Funktionen zur Erstellung oder zur Abfrage von Geometrien (in verschiedene Formate z.B. Well-Known Text (WKT)/WKB, GML, SVG oder KML). Ein Großteil der Funktionen basiert auf den OpenGIS Spezifikationen der "OpenGIS Simple Features For SQL". Eine vollständige Beschreibung der Funktionen befindet sich z.B. im PostGIS Manual [OGC 09c].
Anhand eines Beispiels werden hier einige Funktionen beschrieben und erklärt. Möchte man z.B. Großstädte für die Anzeige auf einer vereinfachten Karte Deutschlands speichern, wird zunächst über ein herkömmliches SQL Statement eine Tabelle "staedte" angelegt, in der ein Primärschlüssel (`pkey`) und ein Name (`name`) gespeichert werden kann.

```
CREATE TABLE staedte (pkey INTEGER, name VARCHAR);
```

Dann wird eine PostGIS-Funktion genutzt, um eine Geometriespalte hinzuzufügen, in der die Koordinaten der Stadt gespeichert werden sollen. Die Funktion `AddGeometryColumn` wird mit Hilfe der `SELECT`-Anweisung aufgerufen. Die benötigten Parameter lauten:

- Tabellenname
- Spaltenname
- Spatial Reference System Identifier (SRID)
- Geometrietyp
- Dimension

```
SELECT AddGeometryColumn ('staedte', 'XYCoordinates', -1, 'POINT', 2);
```

Durch die aufgeführten Parameter wird in der Tabelle "staedte" eine Spalte "XYCoordinates" hinzugefügt, die einen zweidimensionalen Punkt repräsentiert [OGC 09c, vgl. S. 15 ff.]. PostGIS verwendet ein "Spatial Reference System (SRS)", welches u.a. für Koordinaten-Projektionen und Transformationen genutzt wird. Dabei stellt der Wert des Parameters `SRID` eine Verknüpfung (Schlüssel) zu der Tabelle "spatial_ref_sys" dar. Wird diese Funktionalität von PostGIS nicht verwendet, wird der Parameter auf "-1" gesetzt [OGC 09c, vgl. S. 13 ff.]. Um nun Daten in der erzeugten Tabelle hinzuzufügen, kann man sich der Funktion `GeometryFromText` bedienen, die mittels `INSERT`-Statement benutzt wird und die Geometriedaten im WKT-Format speichert.

```
INSERT INTO staedte VALUES
  (1, 'Berlin', GeometryFromText('POINT (10 10)', -1));
```

Als Parameter der Funktion `GeometryFromText` wird hier die WKT-Repräsentation der Geometrie, in diesem vereinfachten Beispiel ein zweidimensionaler Punkt mit den Koordinaten "10, 10", sowie "-1" für `SRID` übergeben. Zusammen mit dem `INSERT`-Statement wird ein Datensatz in der Tabelle "staedte" angelegt mit den Werten "1" als Primärschlüssel, "Berlin" als Name und ein zweidimensionaler Punkt mit Koordinaten "10, 10" als Geometrierepräsentation [OGC 09c, vgl. S. 11 ff.].

Um die Koordinateninformationen wieder auszulesen, bieten sich verschiedene Möglichkeiten an. Mittels der Funktion `AsText` wird der Wert der Geometriespalte wieder als WKT-Format zurückgegeben. Die Funktionen `st_x` bzw. `st_y` liefern den Wert der x-Koordinate bzw. y-Koordinate zurück [Mitc 08, vgl. S. 342 ff.]. Möchte man die Geometriespalte im SVG-Format erhalten, wählt man die Funktion `AsSVG`. Alle genannten Funktionen werden mit dem `SELECT`-Statement aufgerufen:

```
SELECT AsSVG(XYCoordinates) FROM staedte;
```

Analog dazu werden auch Analyse-Funktionen aufgerufen. Die Funktion `Distance` gibt z.B. den Abstand zweier Punkte zurück. Mit Hilfe des folgenden Aufrufs werden die Namen aller Städte ausgegeben, deren Abstand zu einem Referenzpunkt (`refX`, `refY`) kleiner als "10" ist:

```
SELECT name FROM staedte
WHERE Distance (POINT(<refX, refY>), xy) <= 10;
```

[Mitc 08, vgl. S. 348 ff.]

4.2 Konzept und Grundstruktur

Um die verschiedenen in Kapitel 3 vorgestellten Eingangsdaten in der PostGIS Datenbank zu verwalten, werden mehrere Tabellen benötigt. `Nodes` wird als Tabelle für Straßenknoten verwendet, die sowohl Knoten der EGT aufnehmen kann als auch Knoten, die die einzelnen Segmente bilden, die zwischen zwei Knoten der EGT liegen. Auf die Berechnung dieser ”Segmentknoten” wird im weiteren Verlauf dieser Arbeit detailliert eingegangen (Kapitel 5.2.2).

Die Tabelle enthält die folgenden Spalten:

- `KEY` - Primärschlüssel bestehend aus den Koordinaten und dem Straßennamen
- `EGT_Name` - Name des Knotens
- `EGT_Geocode` - Geocode des Knotens
- `EGT_Roadname` - Name der Straße, auf der der Knoten liegt
- `Original_Key` - der originale Schlüssel (Geocode + Straßenname) des Knotens
- `EGT_Typ` - der Knotentyp aus der EGT
- `EGT_Von_Ort` - Richtungsort ”von”
- `EGT_Nach_Ort` - Richtungsort ”nach”
- `EGT_Road_Int` - internationale numerische Bezeichnung der Straße
- `EGT_Vorgaenger_From` - der Geocode des EGT-Vorgängerknotens
- `EGT_Nachfolger_To` - der Geocode des EGT-Nachfolgerknotens
- `PGIS_Coordinates` - Geometriespalte, die einen zweidimensionalen Punkt repräsentiert

Im Gegensatz zur DDG, die in ihrer EGT als Primärschlüssel die Kombination aus Geocode und Straßenname verwendet, werden hier die x- und y-Koordinate in Verbindung mit dem Straßennamen genutzt. Da im Rahmen dieser Arbeit Geocodes für neue Knoten berechnet werden mussten, die zum Teil sehr nahe beieinander liegen, ergab dies vereinzelt identische Geocodes für Knoten mit unterschiedlichen Koordinaten und damit Verletzungen des Primärschlüssels. Der ”originale” Schlüssel wird in der Spalte `Original_Key` gespeichert.

Die Straßen, die sich aus der Vorgänger- bzw. Nachfolgerknotenangabe ergeben (siehe Kapitel 5.2.1, Algorithmus 2), werden in der Tabelle ”Roads” gespeichert. Diese enthält neben einem Primärschlüssel (`Key`), der sich aus dem Schlüssel des Start- und Endknotens der entsprechenden Straße ergibt, den Straßennamen (`Roadname`) sowie eine Auflistung der Schlüssel der enthaltenen EGT-Knoten in geordneter Reihenfolge (`Nodes`)

und eine Geometriespalte (`PGIS_Nodes`), die einen Linienzug durch alle enthaltenen EGT-Knoten repräsentiert.

Die Tabelle ”RoadSections” soll die Streckenabschnitte enthalten, für deren Segmente die Verkehrsdichteangaben geliefert werden. Dazu wird der Schlüssel des Start- und End-Straßenknotens (`EGT_NodeFromKey` bzw. `EGT_NodeToKey`), der Straßenname (`Roadname`), sowie die Segmentanzahl (`SegmentAmount`) gespeichert. Des Weiteren wird eine Geometriespalte verwendet, die den Straßenabschnitt als Linienzug repräsentiert (`PGIS_Nodes`). Primärschlüssel der Tabelle sind die Spalten `EGT_NodeFromKey` und `EGT_NodeToKey`.

Für die einzelnen Segmente der Straßenabschnitte ist die Tabelle ”RoadSegments” vorgesehen. Diese enthält neben dem Start- bzw. Endknoten (`NodeFromKey` bzw. `NodeToKey`) die `SegmentID`. Des Weiteren werden die Schlüssel der EGT-Knoten, zwischen denen das Segment liegt, in den Feldern `EGT_NodeFromKey` bzw. `EGT_NodeToKey` gespeichert. Auch die Tabelle ”RoadSegments” verwendet eine Geometriespalte `PGIS_Nodes` zur Repräsentation des Segments als Linienzug. `NodeFromKey` und `NodeToKey` dienen ebenfalls als Primärschlüssel der Tabelle.

Um die Verkehrsdichtedaten der jeweiligen Segmente zu einem bestimmten Zeitpunkt zu speichern, wird die Tabelle ”RoadSegmentData” verwendet. Diese Tabelle enthält die folgenden Spalten:

- `EGT_NodeFromKey` - Schlüssel des EGT-Start-Straßenknotens
- `EGT_NodeToKey` - Schlüssel des EGT-End-Straßenknotens
- `SegmentID` - die Segmentnummer
- `Date` - der Zeitpunkt, zu dem die Verkehrsdichte gültig ist
- `CarDensity` - die Verkehrsdichte des Segments

Um die Daten eindeutig zuordnen zu können, werden als Primärschlüssel die Spalten `EGT_NodeFromKey`, `EGT_NodeToKey`, `SegmentID` sowie `Date` verwendet (vgl. Kapitel 3.3 bzw. 3.4).

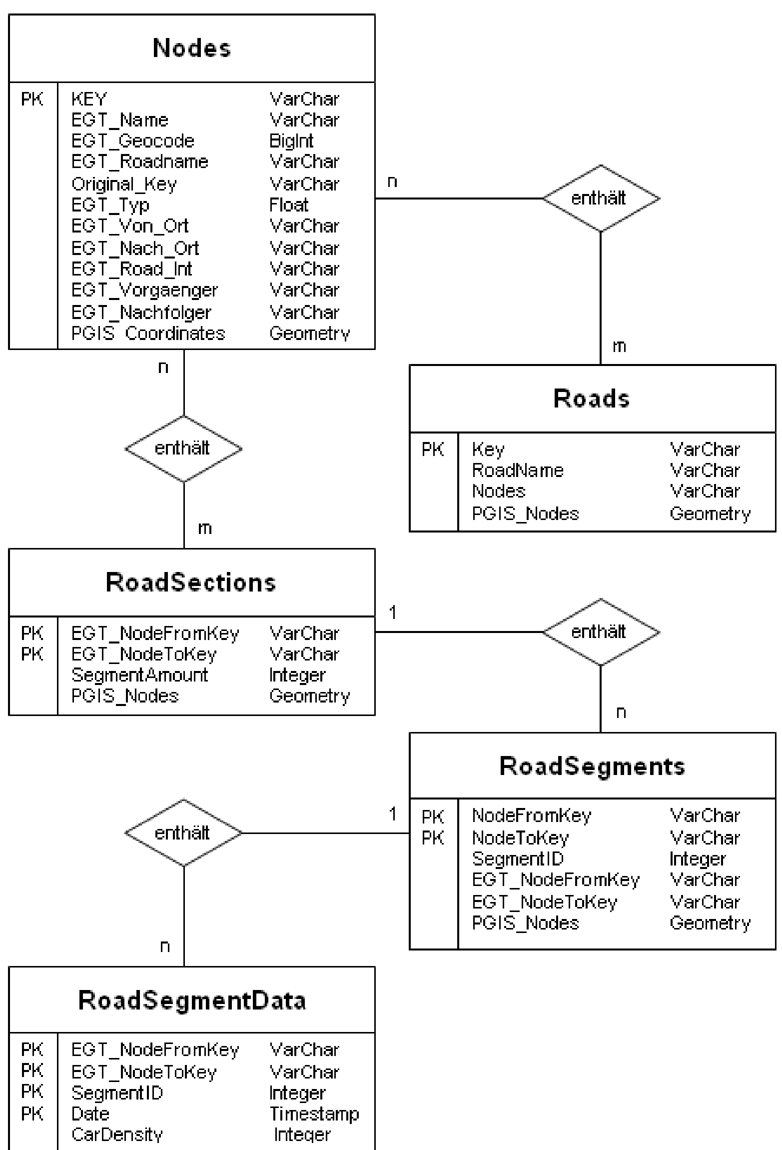


Abb. 4.1: ER-Modell der Datenbankstruktur

5 Java-Programm

Das erstellte Java-Programm übernimmt im Wesentlichen drei Aufgaben. Es überprüft die Datenbankstruktur und legt diese ggf. an. Darüber hinaus pflegt es sowohl die Basisdaten als auch die Verkehrsdichtedaten in die Datenbank ein. Der Quellcode sowie eine Javadoc-Dokumentation aller implementierten Klassen und Methoden befindet sich auf der beigefügten Daten-CD (siehe Anhang: Daten-CD).

5.1 Überprüfen und Anlegen der Datenbankstruktur

Nach dem Programmstart muss zunächst eine Verbindung mit der Datenbank hergestellt und bei Erfolg, die Datenbankstruktur überprüft und ggf. angelegt werden. Dazu wird über ein `SELECT`-Statement jede Programm-relevante Tabelle abgefragt. Nicht vorhandene, für das Programm relevante Tabellen werden automatisch erstellt.

```
/**
 * createTableRoads
 * Erstellt die Tabelle Roads fuer die Strassen
 *
 * @return true wenn erfolgreich erstellt, sonst false
 */
public boolean createTableRoads() {
    boolean result = true;

    System.out.print("Erstelle Tabelle 'roads'..........................");

    String sqlSt = "CREATE TABLE roads (" +
                   "KEY        VARCHAR NOT NULL," +
                   "Roadname   VARCHAR," +
                   "nodes      VARCHAR NOT NULL," +
                   "PRIMARY KEY(KEY)" +
                   ");";
    result &= executeQuery(sqlSt);
    result &= executeQuery("SELECT AddGeometryColumn ('roads', " +
                                                      "'pgis_nodes', " +
                                                      "-1, " +
                                                      "'LINESTRING', " +
                                                      "2);");

    if (result)
        System.out.println("OK!");

    return result;
}
```

Abb. 5.1: Methode zum Anlegen der Tabelle ”Roads” in der Datenbank

Abbildung 5.1 zeigt den Quelltext der Methode `createTableRoads`, die mit Hilfe des enthaltenen SQL-Codes die Tabelle "Roads" anlegt. Anlalog dazu werden auch die übrigen Tabellen erzeugt. Alle Datenbank-relevanten Methoden wurden in der Klasse `DB_All` bzw. in den Klassen implementiert, von denen `DB_All` erbt.

5.2 Einlesen der Daten

Sowohl Basisdaten als auch Verkehrsdichtedaten werden in Form von XML-Dateien geliefert (siehe Kapitel 3). Um die darin enthaltenen Informationen auszulesen, wurde die Klasse `XML_Tool` implementiert. Diese Klasse enthält Methoden, um mittels DOM-Parser auf XML-Dateien zuzugreifen und enthaltene Informationen auslesen zu können.

5.2.1 Einlesen der Basisdaten

Beim Einlesen der Basisdaten soll zum einen die Tabelle "Nodes" mit den Straßenknoten aus den EGT-Daten gefüllt werden und zum anderen das Straßennetz bzw. die darin enthaltenen Straßen gebildet und in der Tabelle "Roads" gespeichert werden. Die Straßen, die sich aus der Vorgänger- bzw. Nachfolgerknotenbeziehung aus den EGT-Daten ergeben, stellen im Grunde eine Liste von Straßenknoten dar, die miteinander verbunden werden. Diese Straßen werden in den EGT-Daten noch fahrbahnunabhängig betrachtet, d.h. Fahrbahn und Gegenfahrbahn werden nicht getrennt voneinander behandelt. Um eine Trennung zu gewährleisten, wird beim Auslesen jede Straße nochmal in umgekehrter Knotenreihenfolge gespeichert und somit die Gegenrichtung bzw. die Gegenfahrbahn der jeweiligen Straße erzeugt. Hierbei stellt sich das Problem, dass die Fahrbahnen bei einfacher Anzeige übereinander liegen und nicht voneinander zu unterscheiden sind. Deshalb wurde zudem die Möglichkeit implementiert, die Fahrbahnen einer Straße über einen sogenannten "Splitfaktor" voneinander zu trennen. Dieses Verfahren wird im weiteren Verlauf der Arbeit noch detailliert vorgestellt (siehe Algorithmus 3).

Die Klasse `EGT_MapGenerator`, die von der Klasse `XML_Tool` abgeleitet wurde, übernimmt das Auslesen der Basisdaten aus der XML-Datei. An den Konstruktor der Klasse wird der Dateiname der Quelldatei übergeben. Optional kann auch der Splitfaktor gesetzt werden. Um die Daten nun aus der Quelldatei auslesen zu können, wurde die Funktion `getRoads` implementiert. Zunächst werden die Straßenknoten mittels der implementierten Methode `getEGT_Nodes` in eine Knotenliste eingelesen. Diese Knotenliste wird an die Methode `getSimpleRoads` übergeben, die daraus einfache Straßen bildet und in eine Straßenliste speichert. Falls die Fahrbahntrennung der Straßen aktiviert wurde, wird die Methode `getSplitRoads` aufgerufen, andernfalls `getBidirectionalRoads` und die gebildete Straßenliste übergeben. In beiden Fällen werden die Gegenrichtungen der Straßen erzeugt, indem die Knotenreihenfolge einer Straße erneut gespiegelt gespeichert und zusammen mit den bereits erstellten Straßen zurückgegeben wird. `getSplitRoads` trennt zusätzlich jeweils die beiden Fahrbahnen einer Straße.

Die Algorithmen, die dafür implementiert wurden, sollen nun näher erläutert werden. `getEGT_Nodes` liest die in der XML-Datei aufgelisteten Straßenknoten ein. Zur Speicherung wird eine `Map` bzw. eine `HashMap` verwendet. Eine `Map` ist ein assoziativer Speicher und im Prinzip eine Liste, in der jeder Eintrag aus einem Schlüssel-Objekt (`Key`) und dem dazugehörigen Wert-Objekt (`Value`) besteht, vergleichbar mit einem Telefonbuch, in dem Namen mit Telefonnummern verbunden sind. Dabei ist jeder `Key` eindeutig und wird auf genau einen `Value` gemappt. Eine `Map` eignet sich ideal dazu, viele Elemente unsortiert zu speichern und sie über die Keys schnell wieder verfügbar zu machen:

```
Map<KeyType, ValueType> beispiel = new HashMap<KeyType, ValueType> ();
Map<String, EGT_Node> nodes = new HashMap<String, EGT_Node>();
```
[Ulle 09, vgl. S. 643. ff.]

Grundlage dieser `Map` ist der dafür implementierte Datentyp `Node` bzw. `EGT_Node`. `Node` repräsentiert einen einfachen Straßenknoten inklusive aller nötigen Eigenschaften und Methoden; `EGT_Node` erbt von `Node`, enthält aber zusätzlich noch die Vorgänger- bzw. Nachfolgerbeziehung zu den anderen Straßenknoten, die das Straßennetz ergeben.

Algorithmus 1 getEGT_Nodes: Generieren der Straßenknotenliste (als Map) aus XML-Datei *D*

Lege Map<String, EGT_Node> *Map* zum Speichern der Straßenknoten an
Lege Liste *List* zum Einlesen der XML-Elemente an
Lese alle XML-Elemente aus der Datei *D* in die Liste *List* ein
for all XML-Elemente aus Liste *List* **do**
5: Bestimme Straßenknotenelement aus XML-Element
Lege `EGT_Node` *Node* an und bestimme Attribute aus Straßenknotenelement
if Wert des Attributs `EGT_Typ` ist relevant (ist ein Straßenknoten) **then**
Speichere *Node* mit Geocode + Straßennamen als Key in *Map*
end if
10: **end for**
Liefere *Map* zurück

Nachdem jeder Knoten, wie in Algorithmus 1 dargestellt, zunächst als `EGT_Node` mit allen relevanten Attributen eingelesen und mit einem Key (Geocode + Straßenname) in der `Map` gespeichert wurde, übernimmt die Methode `getSimpleRoads` das Bilden der Straßen. Dazu wurde der Datentyp `Road` bzw. `RoadOfNodes` implementiert. `Road` stellt eine Liste von Objekten in einer geordneten Reihenfolge dar, d.h. der Datentyp beschreibt eine einfache Straße. Hierzu wird eine `ArrayList` verwendet, vereinfacht gesagt, ein Array variabler Länge mit wählbarem Datentyp, in diesem Fall `Object` [Ulle 09, vgl. S. 651. ff.]. `RoadOfNodes` ist von `Road` abgeleitet, besteht aber nur aus Straßenknoten (`Node`). Als Rückgabewert der Methode wurde ebenfalls ein neuer Datentyp implementiert, nämlich `Roads`. `Roads` definiert eine Liste von Straßen (in diesem Fall `RoadOfNodes`). Algorithmus 2 zeigt, wie eine einfache Straßenliste aus der übergebenen Knotenliste über die Vorgänger- bzw. Nachfolgerknotenbeziehung erstellt wird.

Algorithmus 2 getSimpleRoads: Generieren der Straßenliste (Roads) aus Straßenknotenliste *Nodes*

Lege eine Straßenliste (Roads) *Result* für die Rückgabe an
Lege eine Map *Streets* an
(KeyType: Straßenname, ValueType: Knotenliste ArrayList<EGT_Node>)
for all Knoten (EGT_Nodes) aus *Nodes* **do**
5: Bestimme den Straßennamen vom aktuellen Knoten
if Straßenname nicht in *Streets* **then**
Speichere leere ArrayList<EGT_Node> mit Straßenname als Key in *Streets*
end if
Speichere Knoten in der ArrayList von *Streets* mit Key = Straßenname
10: **end for**
Lege Knoten (EGT_Node) *now*, *next* und *first* an
Lege Boolean *Kreis* an
for all Knotenliste (List<EGT_Node>) *Street* aus *Streets* **do**
while *Street* nicht leer **do**
15: Lege neue RoadOfNodes *Road* an
Füge *now* = erster Knoten aus *Street Road* hinzu, entferne ihn aus *Street*
Setze *Kreis* auf "false"
while Finde *next* = Nachfolger von *now* in *Nodes* und nicht *Kreis* **do**
if *Street* enthält *next* **then**
20: Setze *Kreis* auf "true"
end if
Füge *next* an das Ende von *Road* ein
Entferne *next* aus *Street*
Setze *now* = *next*
25: **end while**
Setze *now* = *first*
Setze *Kreis* auf "false"
while Finde *next* = Vorgänger von *now* in *Nodes* und nicht *Kreis* **do**
if *Street* enthält *next* **then**
30: Setze *Kreis* auf "true"
end if
Füge *next* am Anfang von *Road* ein
Entferne *next* aus *Street*
Setze *now* = *next*
35: **end while**
Füge *Road* zu *Result* hinzu
end while
end for
Liefere *Result* zurück

Zu Beginn sollen alle Knoten ihrer Straße zugeordnet werden. Dazu wird eine Map angelegt, die den Straßennamen als KeyType und eine Knotenliste in Form einer ArrayList von Knoten (EGT_Node) als ValueType verwendet. Die übergebene Knotenliste wird durchlaufen und der Straßenname des aktuellen Knotens bestimmt. Ist dieser noch

nicht in der neuen `Map` enthalten, wird für ihn ein neuer Eintrag mit einer leeren `ArrayList` erstellt. Der aktuelle Knoten wird über den Straßennamen der entsprechenden `ArrayList` hinzugefügt. Dadurch werden alle Knoten einer Straße in der selben Knotenliste (`ArrayList`) gespeichert, sind jedoch noch in ungeordneter Reihenfolge und repräsentieren daher noch nicht die Straße.
Um die korrekte Reihenfolge herzustellen, wird die erstellte `Map` nun durchlaufen und jede Straße einzeln betrachtet. Der erste Knoten wird einem neuen `RoadOfNodes`-Objekt hinzugefügt und aus der aktuell betrachteten Straße entfernt. Danach wird aus der `Map` der Straßenknoten über das Attribut `EGT_Nachfolger` der Nachfolgerknoten bestimmt, hinter dem ersten Knoten eingefügt und ebenfalls aus der aktuellen Straße entfernt. Dieselbe Prozedur wird solange für die nachfolgenden Knoten durchgeführt, bis kein Nachfolgerknoten mehr gefunden wird oder der betrachtete Knoten nicht mehr in der aktuellen Straße enthalten ist, d.h. bereits entfernt wurde. Das impliziert, dass der Straßenverlauf einen Kreis darstellt. Analog zur Nachfolgerknotenbestimmung werden alle Vorgängerknoten in korrekter Reihenfolge dem `RoadOfNodes`-Objekt hinzugefügt. Die auf diese Weise erzeugte Straße wird dem Rückgabeobjekt hinzugefügt und die nächste Straße betrachtet.

Je nach Einstellung wird danach entweder die Methode `getSplitRoads` oder `getBidirectionalRoads` mit der erzeugten, einfachen Straßenliste aufgerufen. Ziel bei beiden ist es, für jede Straße der Liste, die jeweilige Gegenfahrbahn zu generieren und ebenfalls der Liste hinzuzufügen. `getBidirectionalRoads` durchläuft dazu jede Straße der Liste von hinten nach vorne und erstellt dabei die Gegenfahrbahn durch Spiegelung der Knotenreihenfolge. Ergebnis der Methode ist ein Objekt des Datentyps `Roads`, dem zusätzlich zu den Fahrbahnen der einfachen Straßenliste noch die generierten Gegenfahrbahnen hinzugefügt werden.

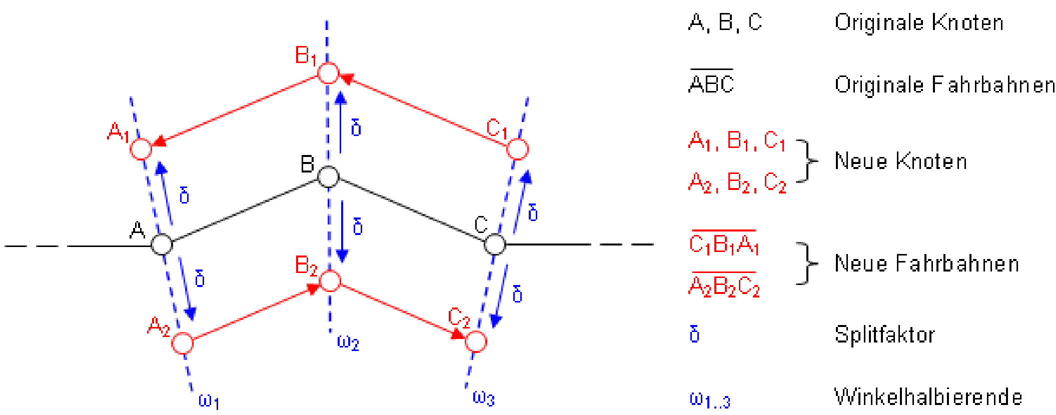


Abb. 5.2: Generierung der neuen Knoten beim "Straßensplitting" mit Hilfe von Winkelhalbierenden

`getSplitRoads` erstellt nicht nur Fahrbahn und Gegenfahrbahn, sondern "trennt" diese zusätzlich. Damit die Fahrbahnen nicht übereinander verlaufend dargestellt werden, wird jeder Knoten einer Straße in zwei Knoten aufgetrennt, indem er mit dem sogenannten "Splitfaktor" auf der Winkelhalbierenden am Knoten, die sich durch den

Straßenverlauf ergibt, in die jeweils beiden möglichen Richtungen verschoben wird (siehe Abbildung 5.2). Dadurch, dass die neuerzeugten Knoten zum einen immer an den Anfang und zum anderen an das Ende der jeweiligen Straße eingefügt werden, erfolgt automatisch die unterschiedliche Richtungsgenerierung. Die Vorgehensweise wird in Algorithmus 3 veranschaulicht.

Algorithmus 3 getSplitRoads: Generieren der bidirektionalen Straßenliste (`Roads`) aus einfacher Straßenliste (*SimpleRoads*) mit Fahrbahntrennung

```
    Lege Roads LeftAndRight für die Rückgabe an
    for all RoadOfNodes Road aus SimpleRoads do
      Lege RoadOfNodes für rechte (Right) und linke (Left) Fahrbahn an
      if Anzahl Knoten in Road größer "2" then
 5:     Setze Startknotenverbindung p0 und p1 (EGT_Node)
        Bestimme Vektor p0->p1 und berechne dessen Normale
        Berechne rechten Knoten (siehe Alg. 4) und füge ihn am Ende von Right ein
        Berechne linken Knoten (siehe Alg. 5) und füge ihn vorne in Left ein
        {Durchlaufe Liste vom zweiten bis zum vorletzten Knoten}
10:     for i = 1 to "Anzahl Knoten in Road - 1" do
          Setze Knoten p0 ("i-1"), p1 ("i") und p2 ("i+1") aus Road
          (Vorgänger-, aktueller und Nachfolgerknoten)
          Berechne Winkelhalbierende zwischen den Vektoren p0->p1 und p1->p2
          Berechne Normale der Winkelhalbierenden
15:       Berechne rechten Knoten und füge ihn am Ende von Right ein
          Berechne linken Knoten und füge ihn vorne in Left ein
        end for
        Setze Endknotenverbindung p0 und p1 (EGT_Node)
        Bestimme Vektor p0->p1 und berechne dessen Normale
20:     Berechne rechten Knoten und füge ihn am Ende von Right ein
        Berechne linken Knoten und füge ihn vorne in Left ein
      else if Anzahl Knoten in Road ist gleich "2" then
        Setze Knotenverbindung p0 und p1 (EGT_Node)
        Bestimme Vektor p0->p1 und berechne dessen Normale
25:     Berechne rechten Knoten von p0 und füge ihn am Ende von Right ein
        Berechne linken Knoten von p0 und füge ihn vorne in Left ein
        Berechne rechten Knoten von p1 und füge ihn am Ende von Right ein
        Berechne linken Knoten von p1 und füge ihn vorne in Left ein
      end if
30:   if Anzahl Knoten in Right größer 1 then
        Füge Right in LeftAndRight hinzu
      end if
      if Anzahl Knoten in Left größer 1 then
        Füge Left in LeftAndRight hinzu
35:   end if
    end for
    Liefere LeftAndRight zurück
```

In `getSplitRoads` werden die aus `getSimpleRoads` erhaltenen Straßen nacheinander betrachtet. Jeder Knoten soll in zwei aufgetrennt werden, um nebeneinander laufende linke und rechte Fahrbahnen zu generieren. Dazu werden Winkelhalbierende verwendet. Einen Sonderfall stellen der erste und der letzte Knoten einer Straße dar. Hier muss der Knoten auf der zur Startknotenverbindung senkrechten Strecke durch den betrachteten Knoten verschoben werden. Dazu wird der Richtungsvektor der Knotenverbindung aus erstem und zweitem bzw. vorletztem und letztem Knoten und dessen Normale berechnet. Zu den Koordinaten des Knotens wird jeweils das Produkt aus Normale und "Splitfaktor" addiert bzw. substrahiert und dadurch der Knoten für die rechte und linke Fahrbahn bestimmt. Die Erstellung der neuen Knoten wird in den Funktionen `getRightNode` (Algorithmus 4) bzw. `getLeftNode` (Algorithmus. 5) gezeigt. Damit die enthaltenen Richtungsangaben der neuerstellten Gegenfahrbahn korrekt bleiben, werden die Attribute "EGT_Von_Ort" bzw. "EGT_Nach_Ort" sowie "EGT_Vorgaenger" bzw. "EGT_Nachfolger" der Knoten getauscht. Des Weiteren müssen die Geocodes der neuerstellten Knoten beim "Straßensplitting" neu berechnet werden. Dazu wurde ein "Geocoder" implementiert, der für x- und y-Koordinate den entsprechenden Geocode bzw. für einen Geocode die entsprechenden Koordinaten berechnet. Für alle anderen Knoten wird zum Verschieben der Koordinaten eine Winkelhalbierende am betrachteten Knoten benötigt. Diese wird anhand der Richtungsvektoren aus vorhergegangenem und aktuellem Knoten und aktuellem und nachfolgendem Knoten berechnet. Über die Normale der Winkelhalbierenden werden unter Verwendung der Methoden `getRightNode` bzw. `getLeftNode` wieder die Knoten für die linke und rechte Fahrbahn gewonnen. Die auf diese Weise erstellten Fahrbahnen, werden dem Rückgabeobjekt hinzugefügt. Straßen, bestehend aus nur einem Knoten, sowie leere Straßen werden nicht übernommen. Dieses Verfahren stellt eine einfache Möglichkeit dar, um die Fahrbahnen einer Straße getrennt voneinander anzuzeigen. Allerdings kommt es z.B. bei sehr "engen" Kurven zu Darstellungsproblemen, die an dieser Stelle aber vernachlässigt wurden, da die Anzeige selbst nicht Bestandteil dieser Arbeit gewesen ist. Die aus `getBidirectionalRoads` bzw. `getSplitRoads` erhaltene Liste von Straßen wird ebenfalls als Rückgabewert der Methode `getRoads` verwendet.

Algorithmus 4 getRightNode: Berechne rechten Knoten zur Fahrbahntrennung (erhält Knoten p_1 und Normale $Norm$)

Lege `EGT_Node` $Result = p_1$ für die Rückgabe an
Setze Koordinaten von *Result* (verschoben mit Splitfaktor * *Norm* nach rechts)
Berechne Geocode von *Result* neu
Liefere *Result* zurück

Algorithmus 5 getLeftNode: Berechne linken Knoten zur Fahrbahntrennung (erhält Knoten p_1 und Normale $Norm$)

Lege `EGT_Node` $Result = p_1$ für die Rückgabe an
Setze Koordinaten von *Result* (verschoben mit Splitfaktor * *Norm* nach links)
Berechne Geocode von *Result* neu
Tausche Richtungsangaben (Spiegelung der Richtung dieser Fahrbahn)
5: Liefere *Result* zurück

Nachdem die Straßenliste auf die gezeigte Weise erstellt wurde, müssen die Informationen in der Datenbank gespeichert werden. Auch hier muss unterschieden werden, ob "Straßensplitting" verwendet wurde oder nicht. Falls es aktiviert wurde, sind alle in den Straßen enthaltenen Knoten eindeutig und müssen in die Tabelle "Nodes" aufgenommen werden. Wenn es nicht aktiviert wurde, wurde zur Generierung der beiden Fahrbahnen lediglich die Knotenreihenfolge gespiegelt. Deshalb müssen hier nur die Knoten einer Fahrbahn gespeichert werden. Dazu wurden die Funktionen `writeToNodes` und `writeToRoads` sowie `writeToRoadsAndNodes` für das "Straßensplitting" in `DB_All` implementiert. `writeToRoadsAndNodes` erhält die Straßenliste (`Roads`) und schreibt die Straßenknoten der aktuellen Straße in der Tabelle "Nodes" sowie die Straßen selbst in der Tabelle "Roads". Die Methoden `writeToNodes` und `writeToRoads` können mit verschiedenen Übergabeparametern aufgerufen werden (z.B. einzelne Knoten bzw. einzelne Straßen aber auch Knoten- oder Straßenlisten). Abbildung 5.3 zeigt beispielhaft den Quelltext der Methode, die eine Straße in der Tabelle "Roads" speichert.

```
/**
 * writeToRoads
 * Speichert eine Liste von Knoten (Strasse) in die Tabelle Roads
 *
 * @param nl - die Liste der Knoten (Strasse)
 * @return true wenn erfolgreich gespeichert, sonst false
 */
public boolean writeToRoads(RoadOfNodes nl) {
    if ((nl == null) || (nl.isEmpty())) return false;

    StringBuffer sqlSt = new StringBuffer("");

    sqlSt.append("INSERT INTO roads ");
    sqlSt.append("VALUES(");
    sqlSt.append("'" + nl.getKey() + "', ");
    sqlSt.append("'" + nl.getRoadname() + "', ");
    sqlSt.append("'" + nl.getNodeKeyList() + "', ");
    sqlSt.append(nl.getWellKnownText());
    sqlSt.append(");");

    return executeQuery(sqlSt.toString());
}
```

Abb. 5.3: Methode zum Speichern einer Straße in der Tabelle "Roads"

5.2.2 Einlesen der Verkehrsdichtedaten

Wie in Kapitel 3.4 detailliert beschrieben, werden die Verkehrsdichtedaten in Form von XML-Dateien geliefert, die eine Liste von Knotenverbindungen (Sektionen) enthält. Eine Knotenverbindung wird dabei nochmals in Segmente gleicher Länge unterteilt, für die jeweils Verkehrsdichten eines bestimmten Zeitpunkts angegeben werden. Die angegebenen Sektionen sowie deren Segmente sollen zusammen mit den Verkehrsdichten in der Datenbank gespeichert werden. Um die Informationen später visualisieren zu können, müssen die Segmente einer Sektion zunächst erstellt und berechnet werden. Segmente setzen sich aus einem Anfangs- und einem Endknoten zusammen. Diese Knoten werden jedoch nicht für alle Segmente angegeben, sondern nur Anfangs- und Endknoten einer

Sektion sowie die Anzahl der Segmente, in die diese unterteilt wird. Da alle Segmente die gleiche Länge haben, können die Koordinaten, und damit auch der Geocode, der sogenannten "Segmentknoten", aus diesen Informationen berechnet werden. Dazu wird, wie in Abbildung 5.4 zu sehen, der Richtungsvektor von Start- zu Endpunkt mit dem Quotienten aus Segment-ID des zu bestimmenden Segmentknotens und der Segmentanzahl multipliziert und das Ergebnis zum Startpunkt addiert. Es ergibt sich folgende Formel zur Berechnung der Segmentknoten: $S_i = \vec{AB} * \frac{i}{\alpha} + A$ (A, B - Straßenknoten der Sektion) bzw. $S_i\left((X_b - X_a) * \frac{i}{\alpha} + X_a, (Y_b - Y_a) * \frac{i}{\alpha} + Y_a\right)$ (X, Y - x- bzw. y-Koordinaten der Knoten).

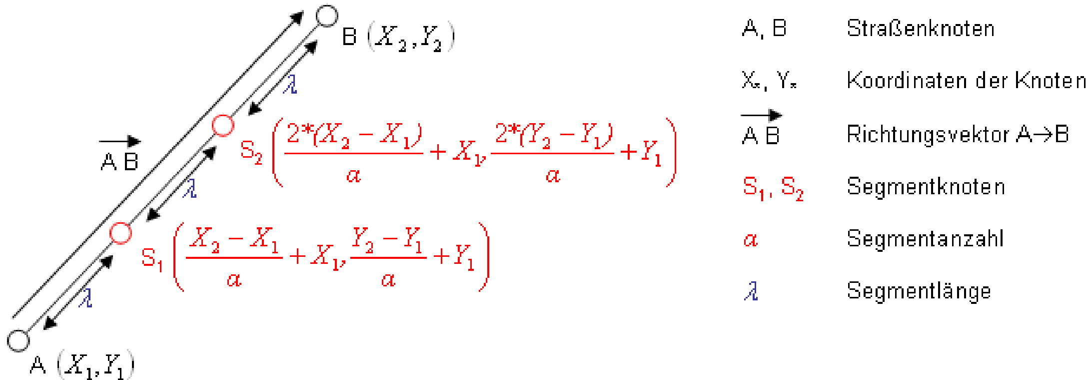


Abb. 5.4: Berechnung der Segmentknoten

Der Aufbau einer Sektion bleibt konstant. Da die Berechnung der Segmentknoten und die Speicherung von Sektionen und Segmenten deshalb nur beim jeweiligen ersten Auftreten erfolgen muss, wird beim Einlesen der Verkehrsdichtedaten zunächst überprüft, ob die aktuell betrachtete Sektion bereits in der Datenbank vorhanden ist. Ist dies nicht der Fall, werden alle nötigen Informationen berechnet und gespeichert, ansonsten werden die Verkehrsdichten der einzelnen Segmente direkt ausgelesen und gespeichert.
Im Gegensatz zum Einlesen der Basisdaten werden hier nicht erst alle Daten ausgelesen und im Anschluss gespeichert, sondern während die Knotenverbindungsliste der XML-Datei durchlaufen wird, werden die relevanten Informationen in die Datenbank eingefügt. Um die relevanten Daten auszulesen und zu speichern, wurden ebenfalls neue Datentypen implementiert. `TML_RoadSection` repräsentiert eine Sektion. Neben den Attributen Anfangsknoten (`EGT_Node`), Endknotens (`EGT_Node`), Segmentanzahl ("tml_segmentAmount") sowie des Gültigkeitszeitpunkts ("tml_dateOfService") ist eine Liste der Segmente enthalten. Dafür wurde der Datentyp `TML_RoadSegment` erstellt. Dieser besteht aus einem Start-Segmentknoten ("from"), einem End-Segmentknoten ("to") sowie der SegmentID ("tml_segmentId"). Der Datentyp des Segmentknotens lautet `TML_RoadSegmentNode`. Dieser wurde von `Node` abgeleitet und um den Sektionsanfangs- sowie Endknoten erweitert. Um einen Segmentknoten in der Datenbank von einem Straßenknoten unterscheiden zu können, wird als "EGT_Typ" der bisher nicht verwendete Wert "100" benutzt. Alle neuimplementierten Datentypen enthalten ebenfalls Methoden, um alle Informationen in benötigter Weise setzen und abfragen zu können.
Zum Einlesen der XML-Dateien wurde die Klasse `TML_MapGenerator` erstellt, die eine aktive Instanz der Datenbankklasse `DB_All` erhält sowie die Information, ob beim

Einlesen der Basisdaten ”Straßensplitting” verwendet wurde oder nicht. Um eine Datei einzulesen, wird die Methode `writeTMLDataToDataBase` mit dem Dateinamen der einzulesenden Datei aufgerufen. Algorithmus 6 zeigt, auf welche Weise die Verkehrsdichtedaten eingelesen werden.

Algorithmus 6 Einlesen der Verkehrsdichtedaten aus XML-Datei D

```
    Lese alle XML-Elemente aus der Datei D in eine Liste L1 ein
    for all XML-Elemente aus Liste L1 do
      Bestimme Straßenname, Geocode von Start- und Endknoten aus XML-Element
      Berechne Koordinaten von Start- und Endknoten
 5:   if Sektion noch nicht in Datenbank vorhanden then
        Lege neue TML_RoadSection Section an
        Bestimme relevante Eigenschaften aus XML-Element, speichere sie in Section
        for i = 1 to Segmentanzahl do
          Berechne Start- und Endknoten des aktuellen Segments
10:       (TML_RoadSegmentNode)
          Lege TML_RoadSegment an und füge es Section hinzu
        end for
        Speichere Section in Datenbank
        (Sektion, enthaltene Segmente sowie Segmentknoten)
15:     Lege Liste L2 für Verkehrsdichtedaten an und lese aus XML-Element aus
        for all XML-Elemente aus Liste L2 do
          Bestimme Segment-ID und Verkehrsdichte aus XML-Element
          Speichere Verkehrsdichte in Datenbank
        end for
20:   end if
    end for
```

Um die Segementknoten berechnen zu können, müssen die Koordinaten des Start- und Endknotens der Sektion bestimmt werden, da nicht diese, sondern die Geocodes in der XML-Datei enthalten sind. Ist das ”Straßensplitting” deaktiviert, können die Koordinaten mit Hilfe des implementierten Geocoders schnell berechnet werden. Im anderen Fall sind die Geocodes für die Knoten nicht mehr aktuell. Deshalb müssen die Koordinaten mit Hilfe der Spalte ”Original_Key” aus der Tabelle ”Nodes” ausgelesen werden. Um viele Datenbankzugriffe und damit eine höhere Bearbeitungsdauer zu vermeiden, wird über die Methode `getEGT_Nodes` einmalig eine `Map` gefüllt, die den originalen Key als Keytype und den Knoten (`EGT_Node`) als Valuetype verwendet. Auf diese Weise können die Koordinaten dann schnell aus der erstellten `Map` bestimmt werden.
Die Methode `containsRoadSection` überprüft mittels Straßenname sowie Koordinaten von Start- und Endknoten, ob eine Sektion bereits in der Datenbank vorliegt. Ist dies der Fall, kann davon ausgegangen werden, dass auch die entsprechenden Segmente sowie Segmentknoten bereits in der Datenbank vorhanden sind, da der Aufbau einer Sektion konstant bleibt. Ist es nicht der Fall, übernimmt die Methode `generateCoordinates` das Berechnen der Koordinaten sowie des Geocodes des aktuellen Knotens. Als Rückgabewert der Methode wurde der Datentyp `Point` implementiert, der einen Punkt mit x- bzw. y-Koordinate und Geocode darstellt. Es wurden Konstruktoren erstellt, die je nach Parameter die fehlenden Eigenschaften automatisch (unter Verwendung des Geo-

coders) berechnen.
Zum Speichern der Informationen wurde die Methode `writeRoadSection` erstellt, die sowohl die Sektion als auch die enthaltenen Segmente und Segmentknoten in der Datenbank speichert (siehe Algorithmus 7). Die Methoden `writeToRoadSections`, `writeToRoadSegments`, `writeToNodes` sowie `writeToRoadSegmentData` speichern analog zur Methode `writeToRoads` (siehe Abbildung 5.3) die Instanz des jeweiligen Datentyps in die Datenbank. Zuletzt werden die Verkehrsdichteangaben aus der XML-Datei ausgelesen. Dies geschieht auch dann, wenn die Sektion bereits in der Datenbank vorhanden gewesen ist. Die Segmentnummer und die zugehörige Verkehrsdichte werden ausgelesen und mit Hilfe der Methode `writeToRoadSegmentData` mit dem Datum, zu dem die Daten gültig sind, in die Datenbank geschrieben.

Algorithmus 7 writeRoadSection: Speichert die übergebene Sektion (`TML_RoadSection`) *Section* in die Datenbank

```
    Speichere Section in "RoadSections" (writeToRoadSections)
    for all Segmente Segment aus Section do
      Speichere Segment in "RoadSegments" (writeToRoadSegments)
      if Startknoten Node von Segment ist nicht Startknoten von Section then
5:      Speichere Node in "Nodes" (writeToNodes)
      end if
    end for
```

Die XML-Dateien mit den Verkehrsdichtedaten können im Maximalfall in einem minütlichen Zyklus geliefert werden. Um diese Dateien automatisch einzulesen, wurde eine "Verzeichnisüberwachung" implementiert. Dazu läßt sich in den Programmeinstellungen ein Verzeichnispfad setzen, in dem die Daten eingestellt werden sollen. Das Java-Programm überprüft dieses in einem einstellbaren Intervall auf neue Eingangsdaten und liest diese ggf. ein und verschiebt sie in ein dafür vorgesehenes Unterverzeichnis.

Algorithmus 8 run: Einlesen und Verschieben neuer XML-Dateien

```
    if run noch aktiv then
      Return
    end if
    Setze run auf "aktiv"
5:  Bestimme Liste Files von XML-Dateien aus Quellverzeichnis
    for all File aus Files do
      Lese Daten aus File ein (writeTMLDataToDataBase)
      Verschiebe File in Zielverzeichnis
    end for
10: Setze run auf "nicht aktiv"
```

Grundlage bildet die erstellte Klasse `DirectoryWatcher`, die von `TimerTask` abgeleitet wurde. Der Konstruktor der Klasse erhält sowohl den Quell- sowie Zielverzeichnispfad, als auch eine aktive Instanz der Datenbankklasse `DB_All` und die Einstellung, ob "Straßensplitting" verwendet worden ist und erstellt eine Instanz der Klasse

`TML_MapGenerator`. Die Methode `run` wird in jedem Intervall aufgerufen und übernimmt das Einlesen und Verschieben der Dateien. Die Arbeitsweise ist in Algorithmus 8 dargestellt. Um den Zyklus zu starten, wird die Instanz von `DirectoryWatcher` an ein `Timer`-Objekt zusammen mit dem Intervall übergeben und durch die Methode `schedule` aktiviert.

5.3 Ausführung

Das Programm kann grundsätzlich auf zwei verschiedene Arten ausgeführt werden. Es kann manuell bedienbar mit grafischer Oberfläche oder mit einer über Parameter festgelegten Ausführung in der Konsole ohne Programmfenster gestartet werden. Alle Parameter sowie Programmoptionen können in der XML-Datei "TrafficDBConfig.xml" gesetzt und bei Programmstart ausgelesen werden. Bei Ausführung mit grafischer Oberfläche werden alle Einstellungen bei Programmende automatisch gespeichert. Zusätzlich kann der Programmmodus bzw. die Ausführung auch über einen Kommandozeilenparameter bestimmt werden:

- `-v` - Manuelle Ausführung mit Programmfenster
- `-b` - Einlesen der Basisdaten ohne Programmfenster
- `-t` - Einlesen der Verkehrsdichtedaten ohne Programmfenster

5.3.1 Ausführung ohne Programmfenster in der Konsole

```
TrafficDB gestartet!

Verbinde zur Datenbank............................OK!

Ueberpruefe Datenbank.............................OK!

Generiere EGT_Knoten aus XML Datei................OK!
Generiere einfache Strassenliste..................OK!
Generiere bidirektionale Strassen.................2 generiert!

Schreibe 2 Strassen in 'roads':
..................................................100%

Programm beendet!
```

Abb. 5.5: Ausführung des Java-Programms ohne Programmfenster in der Konsole

Bei der Ausführung des Java-Programms in der Konsole muss die Programmabfolge vor dem Start festgelegt werden, d.h. dass alle Programmparameter bereits in der XML-Datei "TrafficDBConfig.xml" gesetzt worden sein müssen und dass bereits ausgewählt werden muss, ob Basisdaten oder Verkehrsdaten eingelesen werden sollen.
Nach dem Programmstart wird automatisch eine Verbindung mit der Datenbank hergestellt und die Datenbankstruktur überprüft und ggf. angelegt. Danach werden je nach Einstellung entweder die Basisdaten eingelesen oder die Verzeichnisüberwachung zum

Einlesen der Verkehrsdichtedaten gestartet. Ein Beispiel für eine Ausführung ohne Programmfenster ist in Abbildung 5.5 dargestellt.

5.3.2 Ausführung mit grafischer Oberfläche

Bei Ausführung des Java-Programms mit grafischer Oberfläche muss zunächst manuell über Schaltflächen die Datenbankverbindung hergestellt und die Struktur überprüft werden. Alle dafür nötigen Parameter können über Textfelder gesetzt werden (siehe Abbildung 5.6). Danach können entweder Basisdaten eingelesen oder die Verzeichnisüberwachung gestartet und ggf. wieder beendet werden. Alle relevanten Parameter lassen sich ebenfalls auf den jeweiligen Reitern der Programmoptionen einstellen (siehe Abbildungen 5.7 und 5.8). Die Programmausgaben werden in einer Protokoll-Box, dargestellt in Abbildung 5.9, ausgegeben.

Abb. 5.6: Java-Programm: Datenbankoptionen

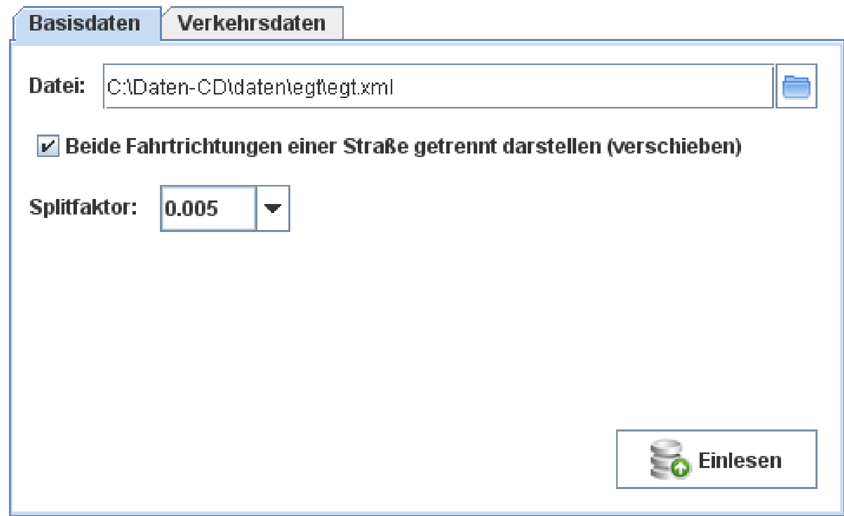


Abb. 5.7: Java-Programm: Basisdaten-Reiter

Abb. 5.8: Java-Programm: Verkehrsdaten-Reiter

Log

TrafficDB gestartet!

Verbinde zur Datenbank...........................OK!

Ueberpruefe Datenbank............................OK!

Verzeichnisueberwachung gestartet!

Speicher Log in Datei (../log.txt)

Abb. 5.9: Java-Programm: Protokoll-Box

6 Web Services

Um den Datenbankinhalt aus dem Internet zugänglich zu machen, werden Web Services verwendet. Diese werden durch Java-Servlets (siehe Kapitel 2.5) realisiert, die durch Apache Tomcat verfügbar gemacht werden. Die Java-Servlets werden in einer ”Web Application Archive (WAR)-Datei” u.a. zusammen mit dem Deployment Descriptor ”web.xml” nach der Java-Servlet-Spezifikation komprimiert und Tomcat hinzugefügt. Der Quellcode sowie eine Javadoc-Dokumentation aller implementierten Klassen und Methoden befindet sich auf der beigefügten Daten-CD (siehe Anhang: Daten-CD).

6.1 Java-Servlets

Es wurden zwei Java-Servlets erstellt, die aus dem Internet aufgerufen werden können: `HS_RoadsAsSVG` für die Abfrage des Straßennetzes im SVG-Format und `HS_RoadsAsXML` für die Abfrage des Straßennetzes im XML-Format. Beide Klassen erben von `HS_Basic`, welches als Basis-Servlet implementiert wurde und von `HttpServlet` abgeleitet ist. Es stellt die gesamte Funktionalität für alle Abfragen zur Verfügung. `HS_RoadsAsSVG` bzw. `HS_RoadsAsXML` sind praktisch leere Klassen und setzen beim Aufruf nur durch den unterschiedlichen Konstruktor ein Attribut in `HS_Basic`, welches das Ausgabeformat bestimmt. Für differenzierte Abfragen können weitere Parameter an die Java-Servlets übergeben werden. Um nicht das gesamte Straßennetz, sondern nur einen Teil abzufragen, kann ein Rechteck definiert werden, welches den gewünschten Ausschnitt darstellt. Dazu können entweder die x- bzw. y-Koordinaten oder die Geocodes der ”linken, oberen Ecke” und der ”rechten, unteren Ecke” des Rechtecks übergeben werden (siehe Abbildung 6.1). Des Weiteren kann ein Datum (mit Uhrzeit) benutzt werden, welches angibt, zu welchem Zeitpunkt die angezeigten Verkehrsdaten gültig sein sollen. Wird kein Datum übergeben, wird das Straßennetz bzw. der Straßennetzausschnitt ohne Verkehrsdichteangaben ausgegeben. Damit ergeben sich folgende mögliche Parameter:

- `xfrom` - x-Koordinate der ”linken, oberen Ecke” des Straßennetzausschnitts
- `yfrom` - y-Koordinate der ”linken, oberen Ecke” des Straßennetzausschnitts
- `xto` - x-Koordinate der ”rechten, unteren Ecke” des Straßennetzausschnitts
- `yto` - y-Koordinate der ”rechten, unteren Ecke” des Straßennetzausschnitts
- `date` - Verkehrsdichte des Segments
- `geocodefrom` - Geocode der ”linken, oberen Ecke” des Straßennetzausschnitts
- `geocodeto` - Geocode der ”rechten, unteren Ecke” des Straßennetzausschnitts

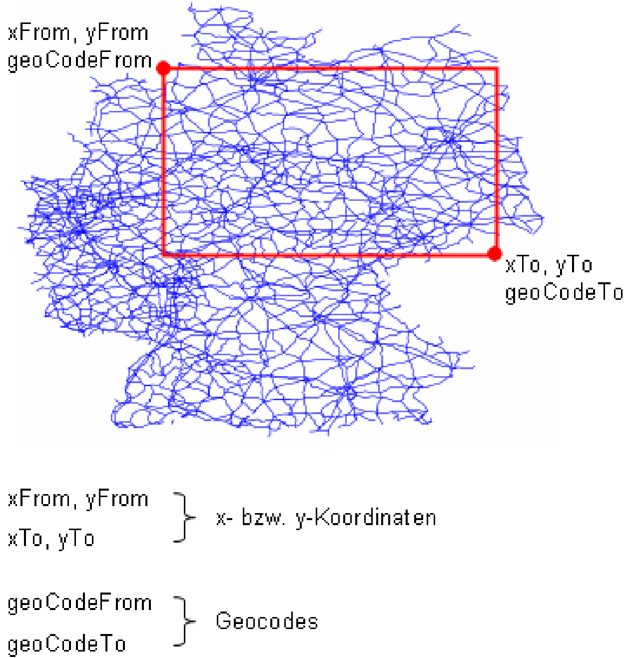


Abb. 6.1: Bilden eines Straßennetzausschnitts durch Rechteckdefinition

In der `doGet`-Methode(siehe Kapitel 2.5) werden zunächst die Parameter aus dem `HttpServletRequest`-Objekt ausgelesen oder mit einem Standardwert initialisiert, falls sie nicht übergeben wurden oder keinen Sinn ergeben. Die Methode `setParamMode` erhält alle Parameter, wertet diese aus und setzt den Abfragemodus. Danach wird die Datenbankverbindung hergestellt. Nun müssen die Informationen aus der Datenbank ausgelesen und in das gewünschte Ausgabeformat konvertiert werden. Um dies zu realisieren, wurden die Klassen `Displaymethods` sowie `FileOperations` implementiert. `FileOperations` stellt Methoden bereit, um die Rückgabewerte der Datenbankabfragen in gültige SVG- oder XML-Dateien zu konvertieren. `Displaymethods` erbt von `FileOperations` und verbindet sozusagen die Datenbankabfragen mit der Formatkonvertierung. Dazu wird an den Konstruktor der Klasse eine aktive Instanz der Datenbankklasse `DB_All` übergeben, in der die Datenbankabfragen angelegt wurden. Je nach Parameterübergabe, sprich Abfragemodus, wird die Methode `getRoadsAsSVG` bzw. `getRoadsAsXML` mit den entsprechenden Parametern aufgerufen. Der Rückgabewert der Methoden ist vom Typ `StringBuffer`. Das `HttpServletResponse`-Objekt der `doGet`-Methode wird mittels der Funktion `setContentType` auf das entsprechende Ausgabeformat gesetzt (`image/svg+xml` für SVG oder `text/xml` für XML), damit der Browser erkennt, wie die zurückgelieferten Daten behandelt werden sollen. Über die Funktion `getWriter` des `HttpServletResponse`-Objekts wird ein `PrintWriter` erhalten, der die erhaltene `StringBuffer`-Instanz für die Rückgabe schreibt. Zuletzt wird die Datenbankverbindung wieder freigegeben.

6.2 Datenbankabfragen

Die Datenbankabfragen unterscheiden sich hauptsächlich in zwei Punkten. Zum einen im Ausgabeformat und zum anderen darin, ob Verkehrsdichteangaben enthalten sein sollen oder nicht. Im Gegensatz zu SVG, welches für die Anzeige der Straßen nur die Koordinaten der Straßenknoten sowie die Knotenbeziehung untereinander benötigt, sollen bei der XML-Ausgabe detaillierte Informationen zu den Straßen bzw. Straßenknoten geliefert werden, da die Daten später weiterverarbeitet werden sollen.

Betrachten wir zunächst die Abfragen ohne Verkehrsdichteangaben. Diese sind recht einfach und beschränken sich auf die Tabelle "Roads", für XML außerdem auf die Tabelle "Nodes", die für zusätzliche Knoteninformationen benötigt wird. Bei SVG kommen vordefinierte Funktionen von PostGIS zum Einsatz. Um die Abfrageresultate im SVG-Format zu erhalten, wird die Funktion `AsSVG` verwendet. Die Abfrage sieht dann folgendermaßen aus:

```
SELECT AsSVG(PGIS_Nodes) FROM Roads;
```

Soll nur ein Straßennetzausschnitt ausgegeben werden, wird die Funktion `Intersection` benutzt. Neben der auszulesenen Spalte "PGIS_Nodes" erhält sie ein Rechteck, welches mit der WKT-Repräsentation eines Polygons den Ausschnitt darstellt und durch die Methode `GeometryFromText` und die übergebenen Parameter definiert wird.

```
SELECT AsSVG(Intersection(PGIS_Nodes,
                          GeometryFromText('POLYGON(xFrom yFrom,
                                                    xTo   yFrom,
                                                    xTo   yTo,
                                                    xFrom yTo,
                                                    xFrom yFrom)', -1)
                          )
            ) FROM Roads;
```

Werden zur Definition des Straßennetzausschnitts Geocodes übergeben, so werden diese zunächst mit Hilfe des Geocoders in entsprechende x- und y-Koordinaten umgewandelt. Das Ergebnis wird in einer String-Liste zurückgeliefert und von der Funktion `writeStringsToSVG` in eine SVG-Datei geschrieben, die abermals in Form eines `StringBuffers` zurückgeliefert wird.

Dient XML als Rückgabeformat, sollen zusätzliche Knoteninformationen mitzurückgeliefert werden. Deshalb müssen die erforderlichen Knoten aus der Tabelle "Nodes" bestimmt werden. Dazu bieten sich zwei Möglichkeiten an: erstens ein einzelnes Auslesen jedes Knotens bei Bedarf, d.h. die fehlenden Eigenschaften eines Knotens werden nur dann aus der Datenbank ausgelesen, wenn sie auch benötigt werden; zweitens können zuerst alle Knoten in eine `Map` eingelesen werden, die den Key der Knoten als "KeyType" und den Knoten selbst (`Node`) als "ValueType" verwendet. Danach kann der Knoten bei Bedarf aus der `Map` bestimmt werden. Im Gegensatz zur ersten Methode, bei der ein Datenbanklesezugriff pro benötigten Knoten erfolgt, werden bei der zweiten Methode die Informationen zu allen Knoten in nur einem Datenbanklesezugriff bestimmt. Die Bearbeitungszeit zum späteren Auslesen des Knotens aus der `Map` ist so gering, dass sie

```
/**
 * checkRange
 * Ueberprueft, ob die Punktkoordinaten in den
 * uebergebenen Grenzen liegen
 *
 * @param Point p - zu ueberpruefender Punkt
 * @param xFrom   - Grenzkoordinaten
 * @param yFrom
 * @param xTo
 * @param yTo
 * @return true, wenn Punktkoordinaten in den
 *         uebergebenen Grenzen liegen sonst false
 */
private boolean checkRange(Point p, double xFrom, double yFrom,
                                     double xTo, double yTo) {
    return (((xFrom == -1) && (yFrom == -1) &&
              (xTo == -1)   && (yTo == -1) ) ? true :
             ((xFrom < p.getX()) && (yFrom < p.getY()) &&
              (xTo   > p.getX()) && (yTo   > p.getY())));
}
```

Abb. 6.2: checkRange: Überprüft, ob die Koordinaten eines Knotens innerhalb des Straßennetzauschnitts liegen

vernachlässigt werden kann. Die erste Methode ist dann schneller als die zweite, wenn nur ein kleiner Teil der Gesamtknoten benötigt wird, z.B. bei einem Straßenausschnitt. Werden viele Knoten benötigt, ist die zweite Methode die schnellere. Mehrere Tests haben gezeigt, dass auch bei kleinen Straßenausschnitten so viele Knoten benötigt werden, dass die zweite Methode schneller arbeitet als die erste. Daher wurde nur diese verwendet und zwar in der Methode `getEGT_Nodes`.
Die `Map` wird zunächst durch `getEGT_Nodes` gefüllt. Danach wird mit Hilfe des SQL-Statements `SELECT Nodes FROM Roads` die Straßenrepräsentationen (geordnete Auflistung der Keys der enthaltenen Straßenknoten) in einer Liste bestimmt. Für jede enthaltene Straße wird die Liste der Keys der enthaltenen Knoten durchlaufen und für jeden Key der entsprechende Knoten aus der `Map` bestimmt. Mit der Methode `checkRange` (Quellcode siehe Abbildung 6.2) wird überprüft, ob die Koordinaten des ermittelten Knotens innerhalb des Straßennetzausschnitts liegen. Wird das gesamte Straßennetz abgefragt, so ist diese Prüfung in jedem Fall korrekt. Rückgabewert ist `Roads`. Jeder Knoten, der innerhalb der Grenzen liegt, wird der aktuellen Straße (im Format `RoadOfNodes`) und jede Straße, die mehr als einen Knoten enthält, dem `Roads`-Objekt hinzugefügt. Dieses Objekt wird im Anschluss mit Hilfe der Methode `writeRoadsToXML` in das XML-Format umgewandelt.

Sollen auch die Verkehrsdichteangaben enthalten sein, wird sowohl für das SVG- als auch für das XML-Format die Methode `getRoadsWithData` benutzt, die ebenfalls `Roads` als Rückgabeformat verwendet. Dieses wird danach von den implementierten Methoden `writeRoadsToXML` bzw. `writeRoadsToSVG` in das entsprechende Format konvertiert.

In `getRoadsWithData` werden zunächst wieder alle Knoten aus der Tabelle "Nodes" in eine `Map` eingelesen. Analog dazu wird mit Hilfe der Funktion `getSegments` eine weitere `Map` gefüllt, die den Key einer Straßensektion als "KeyType" und eine Straße als "ValueType" verwendet. Dabei kommt nicht wie bisher der Datentyp `RoadOfNodes` zum

Einsatz, sondern der Datentyp `RoadOfSections`, welcher eine Straße, bestehend aus Segmenten (`Segment`) darstellt, die eine Straßensektion bilden. Diese werden genauso wie die Knoten bei `RoadOfNodes` in geordneter Reihenfolge in einer Liste gespeichert. Die Sektionen werden quasi als Straße, bestehend aus Segmenten beschrieben. `Segment` wurde speziell für die Ausgabe mit Verkehrsdaten erstellt und beinhaltet neben einem Start- und Endknoten (`Node`) nur die Verkehrsdichte des Straßenteilstücks. Die `Map` wird mit allen Sektionen bzw. Segmenten gefüllt, für die zum übergebenen Zeitpunkt Verkehrsdichtedaten vorliegen. Dazu wird folgendes SQL-Statement verwendet:

```
SELECT rs.egt_nodefromkey, rs.egt_nodetokey,
       rs.nodefromkey, rs.nodetokey,
       rd.cardensity, rd.segmentid
  FROM roadsegments rs, roadsegmentdata rd
  WHERE rs.egt_nodefromkey = rd.egt_nodefromkey
  AND   rs.egt_nodetokey   = rd.egt_nodetokey
  AND   rs.segmentid       = rd.segmentid
  AND   date               = 'date'
  ORDER BY rs.egt_nodefromkey, rs.egt_nodetokey, rd.segmentid
```

Die Ergebnisliste besteht aus Segmenten, definiert durch den Start- bzw. Endknoten (`nodefromkey` und `nodetokey`) und der Segmentnummer (`segmentid`), für die zusätzlich die Verkehrsdichte (`cardensity`) sowie die Knoten der Sektion, in der sich das jeweilige Segment befindet, bestimmt worden sind (`egt_nodefromkey` und `egt_nodetokey`). Die Liste wird aufsteigend nach den Keys der Sektionsknoten sowie der Segmentnummer sortiert, wodurch alle Sektionen in richtiger Reihenfolge aufeinander folgen. Dann wird die zurückgelieferte Liste durchlaufen und für jeden Eintrag ein `Segment` erstellt, für das mit Hilfe von `nodefromkey` und `nodetokey` aus der `Map` der Straßenknoten die jeweiligen Knoten geliefert werden. Die erstellten Segmente werden einer `RoadOfSections` hinzugefügt. Da nur Segmente aus der Ergbenisliste erhalten werden für die auch Verkehrsdichteangaben vorliegen, müssen die übrigen Segmente durch ein Teilstück mit Verkehrsdichte "0" ergänzt werden, damit die Konsistenz der Sektion sichergestellt ist. Sobald eine Sektion abgeschlossen ist, wird die aktuelle `RoadOfSections` in die Rückgabe-`Map` aufgenommen. Dies kennzeichnet eine Änderung von `egt_nodefromkey` und `egt_nodetokey`.

`getRoadsWithData` arbeitet im Prinzip nach folgendem Schema: Nachdem sowohl eine `Map` für die Straßenknoten als auch für die Sektionen erstellt und gefüllt wurde, werden mit Hilfe des SQL-Statements `SELECT Nodes FROM Roads` die Straßenrepräsentationen (geordnete Auflistung der Keys der enthaltenen Straßenknoten) aus der Datenbank ausgelesen. Die enthaltenen Straßen werden durchlaufen und für jede Knotenverbindung, sprich Sektion, aus der `Map` der Straßenknoten mit Hilfe der Keys die Knoten bestimmt. Mit der Funktion `checkRange` wird überprüft, ob die Knoten ggf. im Straßenausschnitt liegen. Falls nicht, wird die nächste Sektion betrachtet. Ansonsten wird in der `Map` der Sektionen nachgesehen, ob ein Eintrag für die Sektion vorliegt, d.h. ob Verkehrsdichteinformationen zu dem übergebenen Zeitpunkt vorhanden sind. Ist dies der Fall, wird die Sektion der aktuellen `RoadOfSections` hinzugefügt, ist es nicht der Fall, wird der aktuellen `RoadOfSections` eine neue Sektion mit Verkehrsdichte "0" angehängt, da nach

Vorgabe der DDG davon ausgegangen wird, dass für Sektionen, für die keine Angaben gemacht wurden, zu diesem Zeitpunkt "freie Fahrt" gilt.

6.3 Rückgabeformate

Als Rückgabeformat für die Weiterverarbeitung, sprich Visualisierung, der Daten wurde das XML-Format vereinbart. Des Weiteren sollte es, u.a. zur eigenen Kontrolle, möglich sein, mittels einfacher Abfragen die Daten anzuzeigen. Dafür wurde das SVG-Format unterstützt. Der Aufbau der Formate wird im Folgenden erläutert. Eine HTML-Seite zum Testen der implementierten Abfragen wurde unter der folgenden Adresse zur Verfügung gestellt: `http://dbs.informatik.uni-osnabrueck.de/trafficdb`.

SVG-Rückgabe

Bei der Rückgabe in SVG erfolgt zunächst, wie in Kapitel 2.4.1 beschrieben, der Dateikopf mit XML-Deklaration und DTD-Angabe. Danach beginnt das Wurzelelement `<svg>`, für das einige Attribute gesetzt werden. Zum einen das `xmlns`- sowie `xmlns:xlink`-Attribut, zum anderen die Höhe und Breite des Dokuments (standardmäßig auf "800x800"). Außerdem wird eine `viewBox` angegeben. Falls nur ein Straßenausschnitt abgefragt wird, werden die Grenzen des Ausschnitts für die `viewBox` umgerechnet, ansonsten wird eine Einstellung für das komplette Straßennetz Deutschlands verwendet. Danach werden die Straßen mit Hilfe des `<path>`-Elements aufgeführt.

Beispiel für den Aufbau einer SVG-Rückgabe:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="800" height="800"
     viewBox="5.5 -56 12 12">
  <path fill="none" stroke-width="0.0010" stroke="green"
        d="M 6.06829 -50.77158 L 6.07524 -50.78644"/>

  ...

</svg>
```

Werden keine Verkehrsdichtedaten mit ausgegeben, wird als Straßenfarbe "blau" benutzt. Andernfalls richtet sich die Farbe jedes Segments nach der entsprechenden Verkehrsdichte. Die gewählte Farbverteilung wird in Tabelle 6.1 gezeigt. Beispiele für SVG-Abfragen ohne bzw. mit Verkehrsdichteangaben, jeweils ohne bzw. mit Verwendung des "Straßensplittings" werden in Abbildung 6.3 dargestellt.

Verkehrsstatus	Dichteangabe (D)	Farbbezeichnung	Farbe
freie Fahrt	D < 3	green	
	3 <= D < 7	seagreen	
	7 <= D < 10	yellowgreen	
dichter Verkehr	10 <= D < 15	yellow	
	15 <= D < 20	gold	
	20 <= D < 25	orange	
stockender Verkehr	25 <= D < 28	darkorange	
	28 <= D < 31	orangered	
	31 <= D < 35	red	
Stau	35 <= D	darkred	

Tabelle 6.1: Farbverteilung der Straßensegmente anhand der vorliegenden Verkehrsdichte

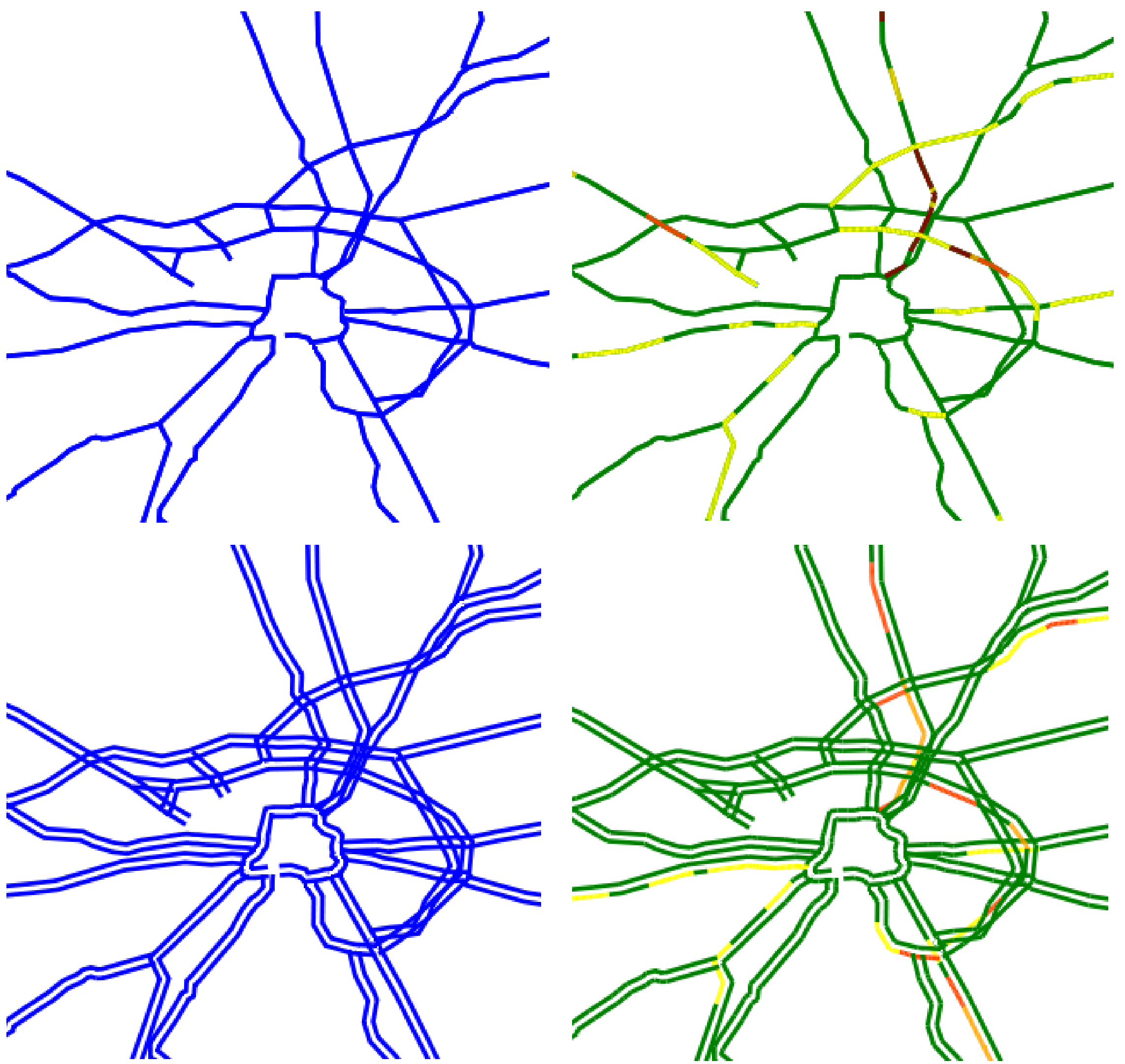


Abb. 6.3: Beispiele für SVG-Abfragen ohne bzw. mit Verkehrsdichteangaben, jeweils ohne bzw. mit Verwendung des "Straßensplittings"

XML-Rückgabe

Bei der Rückgabe in XML erfolgt zunächst die XML-Deklaration. Danach wird das Wurzelelement `<bidirectionalStreets>` begonnen. Dieses enthält nacheinander alle Straßen, entweder als `RoadOfNodes` oder `RoadOfSections`, welche mit den Attributen `key` und Straßenname (`name`) angegeben werden. Für jede Straße werden die enthaltenen Knoten (`Node` bzw. `EGT_Node`) bzw. Sektionen (`Section`) mit den entsprechenden Attributen angegeben.

Beispiel für den Aufbau einer XML-Rückgabe:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bidirectionalStreets>
    <RoadOfNodes key="" name="A1">
        <Node geocode="" strasse="A1" x="" y="" ... />
        <Node geocode="" strasse="A1" x="" y="" ... />

        ...

        <Node geocode="" strasse="A1" x="" y="" ... />
    </RoadOfNodes>
    <RoadOfSections key="" name="A30">
        &lt;Segment key="" strasse="A30" cardensity="25"&gt;
            <NodeFrom geocode="" strasse="A30" x="" y="" ... />
            <NodeTo   geocode="" strasse="A30" x="" y="" ... />
        &lt;/Segment&gt;

        ...

        &lt;Segment key="" strasse="A30" cardensity="40"&gt;
            <NodeFrom geocode="" strasse="A30" x="" y="" ... />
            <NodeTo   geocode="" strasse="A30" x="" y="" ... />
        &lt;/Segment&gt;
    </RoadOfSections>

    ...

</bidirectionalStreets>
```

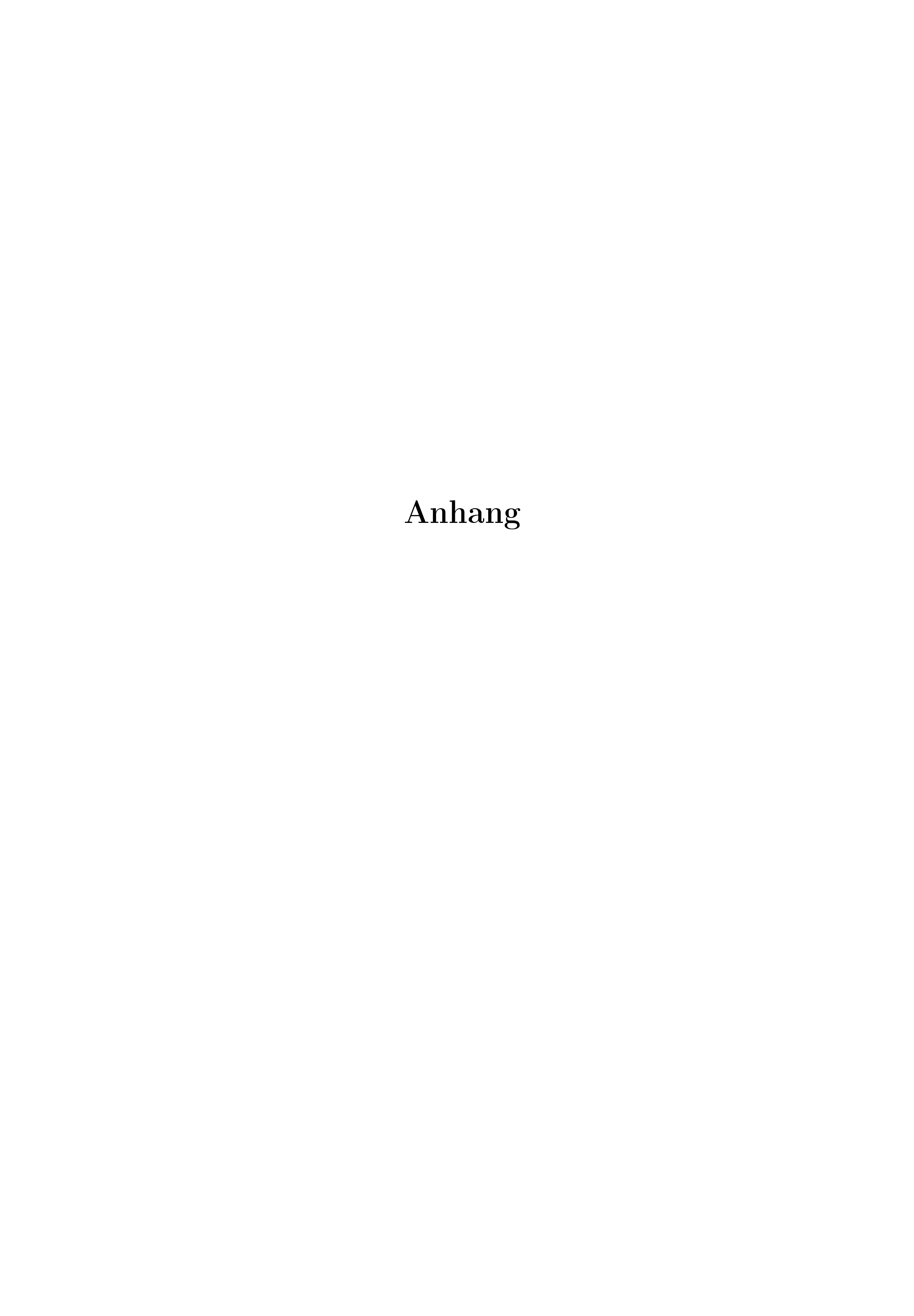
7 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde eine Datenbankgrundlage geschaffen, um die von der DDG erhaltenen Daten effizient und dauerhaft speichern zu können. Die Datenbankstruktur kann von dem implementierten Java-Programm angelegt und Basisdaten eingepflegt werden. Ständig neu anfallende Verkehrsdichtedaten können automatisiert über eine "Verzeichnisüberwachung" in die Datenbank eingelesen werden. Darüber hinaus wurde es ermöglicht, diese Daten über das Internet abzufragen. Dazu wurde ein Web Service erstellt, der die Funktionalität mittels Java-Servlets bereitstellt und mit Hilfe von Tomcat auf Webservern verfügbar gemacht werden kann. Bisher läßt sich das Straßennetz bzw. Straßennetzausschnitte mit oder ohne Verkehrsdichtedaten, die zu einem wählbaren Zeitpunkt gültig sind, in den Formaten XML sowie SVG zurückgeben.

Diese Abfragen stellen nur eine Grundfunktionalität dar und können in Zukunft erweitert werden. Anhand von gesammelten historischen Daten können nicht nur Verkehrszustände zu einem bestimmten Zeitpunkt, sondern auch über einen bestimmten Zeitraum abgefragt und Erkenntnisse für Stauprognosen gewonnen werden (Feierabendverkehr, Ferienzeiten, Feiertage, etc.). Des Weiteren könnten sich Stauverlaufstendenzen (zunehmend, abnehmend, gleichbleibend), Verzögerungszeiten oder Restgeschwindigkeiten im Stau ableiten und dadurch Reisezeiten für eine Routenplanung schätzen lassen. Es sind Verkehrsmeldungen mit Orts-, Zeit- und Längenangaben von Staus denkbar.
Da das verwendete Datenbanksystem PostGIS OGC-standardisiert ist [OGC 09b], können die Daten mit denen anderer im Internet zugänglichen standardisierten Datenbanken beliebig kombiniert werden. Informationen über Baustellen, besondere Vorkommnisse und Wetter könnten gewonnen und Gefahrenmeldungen und Umleitungsempfehlungen erstellt werden. Außerdem sind eine Vielzahl von Abfragen zur Statistikerhebung denkbar.

Die Absprache mit der DDG sieht bisher eine Speicherung aller erhaltenen Daten in der Datenbank vor. Diese können im Maximalfall in einem minütlichen Zyklus für das gesamte deutsche Straßennetz anfallen. Ein Problem, dass sich daraus ergibt, ist, dass der Datenbestand und damit die Datenbank selbst sehr groß wird. Eine Verbesserung würde sich dadurch ergeben, dass Verkehrszustände nur bei Änderung gespeichert werden. Dies würde jedoch wiederum mehr Aufwand beim Auslesen der Daten aus der Datenbank bedeuten. Des Weiteren könnten sehr alte Daten, die inzwischen an Relevanz verloren haben (z.B. aufgrund von Modifikationen in der Verkehrsführung), exportiert und aus der Datenbank entfernt werden.

Die implementierten Abfragen können im SVG-Format zurückgeliefert werden. Dieses stellt eine einfache Datenanzeige dar, die hauptsächlich zur eigenen Kontrolle diente. Von der DDG wird eine Visualisierung der Daten angestrebt, die dem Benutzer eine größere Funktionalität zur individuellen Darstellung ermöglicht. Zu diesem Zweck sollen zukünftig die Abfragen im XML-Format verwendet werden. Die Visualisierung war jedoch nicht mehr Teil dieser Arbeit.



Anhang

Abkürzungsverzeichnis

ANSI	American National Standards Institute
API	Application Programmers Interface
ASCII	American Standard Code for Information Interchange
BMP	Windows Bitmap
CSS	Cascading Style Sheets
DBMS	Datenbankmanagementsystem
DCL	Data Control Language
DDG	DDG Gesellschaft für Verkehrsdaten mbH
DDL	Data Definition Language
DML	Data Manipulation Language
DOM	Document Object Model
DTD	Document Type Definition
EGT	Endgerätetabelle
ER-Modell	Entity-Relationship-Modell
EWKB	Extended Well-Known Binary
EWKT	Extended Well-Known Text
FCD	Floating Car Data-Verfahren
GIF	Graphics Interchange Format
GIS	Geografische Informations-Systeme
GML	Geography Markup Language
GNU	rekursives Akronym von GNU's Not Unix.
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocolf
IBM	International Business Machines Corporation
ISO	Internationale Organisation für Normung
JAR	Java Archiv-Datei
JPG	Joint Photographic Experts Group
JSP	Java Server Pages
JVM	Java Virtual Machine
KML	Keyhole Markup Language
MM	Multimedia and Application Packages
MVC	Model-View-Controller
OGC	Open Geospatial Consortium
PDF	Portable Document Format
PHP	rekursives Akronym für "Hypertext Preprocessor", Backronym aus "Personal Home Page Tools"
PNG	Portable Network Graphics
RSS	Rich Site Summary
SAX	Simple API for XML
SDK	Software Development Kit

SES	Stationäres Erfassungs-System
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
SRID	Spatial Reference System Identifier
SRS	Spatial Reference System
SVG	Scalable Vector Graphics
TMC	Traffic Message Channel
TML	Traffic Map Live
UMN	University of Minnesota
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAR	Web Application Archive
WKB	Well-Known Binary
WKT	Well-Known Text
XHTML	eXtensible HyperText Markup Language
XLink	XML Linking Language
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language

Literaturverzeichnis

[Apache 09] "Apache Tomcat". http://tomcat.apache.org, 2009.

[Bade 04] H. Bader. *SVG Reporting - Vektorgrafiken im Web einsetzen*. Vol. 1, Software & Support Verlag GmbH, Frankfurt, 2004.

[DDG 03] "Richtlinie für die Vergabe von Geocodes". Tech. Rep. 1.1, DDG Gesellschaft für Verkehrsdaten mbH (DDG), Düsseldorf, 2003.

[DDG 04] "TrafficMap Live - Rohdatenformat". Tech. Rep. 1.2, DDG Gesellschaft für Verkehrsdaten mbH (DDG), Düsseldorf, 2004.

[DDG 09] "DDG Gesellschaft für Verkehrsdaten mbH (DDG)". http://www.ddg.de, 2009.

[Flan 98] D. Flanagan. *Java In a Nutshell*. Vol. 2, O'Reilly Media, Köln, 1998.

[GNU 09] "GNU General Public License". http://www.gnu.org/copyleft/gpl.html, 2009.

[Haro 04] E. R. Harold and W. S. Means. *XML in a Nutshell - A Desktop Quick Reference*. Vol. 3, O'Reilly Media, Köln, 2004.

[Heue 97] A. Heuer and G. Saake. *Datenbanken - Konzepte und Sprachen*. Vol. 1. korrigierter Nachdruck, International Thomson Publishing, Bonn, 1997.

[Kemp 01] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung*. Vol. 4, Oldenbourg, München, 2001.

[Mitc 08] T. Mitchell, A. Christl, and A. Emde. *Web Mapping mit Open Source-GIS-Tools*. Vol. 1, O'Reilly Media, Köln, 2008.

[OGC 09a] "OEG Open Geospatial Consortium". http://www.opengeospatial.org, 2009.

[OGC 09b] "OGC OpenGIS Simple Features For SQL Specifications". http://www.opengeospatial.org/standards/sfs, 2009.

[OGC 09c] "Open Geospatial Consortium: PostGIS 1.3.5 Manual". http://postgis.refractions.net/download/postgis-1.3.5.pdf, 2009.

[PostgreS 09] "PostgreSQL". http://www.postgresql.org, 2009.

[SAX 09] "Simple API for XML". http://sax.sourceforge.net, 2009.

[SUN 09a] "Java Plattform, Enterprise Edition (Java EE) 5 Spezifikationen". http://java.sun.com/javaee/technologies/javaee5.jsp, 2009.

[SUN 09b] "JavaServer Pages Technology". http://java.sun.com/products/jsp/index.jsp, 2009.

[Ulle 09] C. Ullenboom. *Java ist auch eine Insel.* Vol. 8, Galileo Press GmbH, Bonn, 2009.

[Vonh 02] H. Vonhoegen. *Einstieg in XML.* Vol. 2, Galileo Press GmbH, Bonn, 2002.

[Vorn 09] O. Vornberger. *Datenbanksysteme (Vorlesungsskript SS 2005).* Vol. 9, Selbstverlag der Universität Osnabrück, Osnabrück, 2009.

[W3C 09a] "W3C Recommendation - Document Object Model (DOM) Level 3 Core Specification". `http://www.w3.org/TR/DOM-Level-3-Core`, 2009.

[W3C 09b] "W3C Recommendation - Extensible Markup Language (XML)". `http://www.w3.org/TR/REC-xml`, 2009.

[W3C 09c] "W3C Recommendation - Scalable Vector Graphics (SVG) 1.1 Specification". `http://www.w3c.org/TR/SVG11`, 2009.

[W3C 09d] "W3C Recommendation - XML Linking Language (XLink) Version 1.0". `http://www.w3.org/TR/xlink`, 2009.

[W3C 09e] "World Wide Web Consortium". `http://www.w3c.org`, 2009.

[Watt 03] A. Watt, C. Lilley, D. J. Ayers, R. George, C. Wenz, T. Hauser, K. Lindsey, and N. Gustavsson. *SVG Unleashed.* Vol. 1, Sams Publishing, Indianapolis, 2003.

[Zepp 04] K. Zeppenfeld. *Lehrbuch der Grafikprogrammierung - Grundlagen, Programmierung, Anwendung.* Vol. 1, Spektrum Akademischer Verlag, Heidelberg, 2004.

Die Literaturquellen beziehen sich nicht nur auf gedruckte Literatur, sondern auch auf Internetseiten. Letztere bieten einen aktuelleren Zugang zu Informationen, vor allem in der recht schnelllebigen Informatik, sind aber auch gerade deshalb schnell veraltet, so dass sie gelöscht werden oder sich die URL durch Umstrukturierungen auf dem Server ändert. Zu dem Zeitpunkt der Abgabe dieser Bachelorarbeit waren alle aufgeführten Adressen erreichbar.

Abbildungsverzeichnis

1.1 Vorgehensweise 2

2.1 Aufbau eines Datenbanksystems 5
2.2 DOM-Tree Beispiel 10
2.3 SVG Beispiele 15
2.4 Beispiel einer einfachen Java-Servlet Klasse 16
2.5 Beispiel eines "Deployment Descriptor": XML-Datei "web.xml" 17

3.1 Getrennter Verlauf auf einem Straßenteilstück 22
3.2 Geschlossene Ringstraße 22
3.3 Beispiel für den Aufbau eines logischen Streckenabschnitts mit fünf Segmenten 23

4.1 ER-Modell der Datenbankstruktur 32

5.1 Methode zum Anlegen der Tabelle "Roads" in der Datenbank 33
5.2 Generierung der neuen Knoten beim "Straßensplitting" mit Hilfe von Winkelhalbierenden 37
5.3 Methode zum Speichern einer Straße in der Tabelle "Roads" 40
5.4 Berechnung der Segmentknoten 41
5.5 Ausführung des Java-Programms ohne Programmfenster in der Konsole . 44
5.6 Java-Programm: Datenbankoptionen 45
5.7 Java-Programm: Basisdaten-Reiter 45
5.8 Java-Programm: Verkehrsdaten-Reiter 46
5.9 Java-Programm: Protokoll-Box 46

6.1 Bilden eines Straßennetzausschnitts durch Rechteckdefinition 48
6.2 checkRange: Überprüft, ob die Koordinaten eines Knotens innerhalb des Straßennetzauschnitts liegen 50
6.3 Beispiele für SVG-Abfragen ohne bzw. mit Verkehrsdichteangaben, jeweils ohne bzw. mit Verwendung des "Straßensplittings" 53

Tabellenverzeichnis

3.1 Vorgänger- und Nachfolgerknotenfestlegung bei getrenntem Verlauf auf einem Straßenteilstück 22
3.2 Vorgänger- und Nachfolgerknotenfestlegung bei geschlossenen Ringstraßen 23
3.3 Klassifikation der Fahrzeugdichte 24

6.1 Farbverteilung der Straßensegmente anhand der vorliegenden Verkehrsdichte 53

Algorithmenverzeichnis

1 getEGT_Nodes: Generieren der Straßenknotenliste (als Map) aus XML-Datei D . . . 35
2 getSimpleRoads: Generieren der Straßenliste (`Roads`) aus Straßenknotenliste $Nodes$ . . . 36
3 getSplitRoads: Generieren der bidirektionalen Straßenliste (`Roads`) aus einfacher Straßenliste ($SimpleRoads$) mit Fahrbahntrennung . . . 38
4 getRightNode: Berechne rechten Knoten zur Fahrbahntrennung (erhält Knoten p_1 und Normale $Norm$) . . . 39
5 getLeftNode: Berechne linken Knoten zur Fahrbahntrennung (erhält Knoten p_1 und Normale $Norm$) . . . 39
6 Einlesen der Verkehrsdichtedaten aus XML-Datei D . . . 42
7 writeRoadSection: Speichert die übergebene Sektion (`TML_RoadSection`) $Section$ in die Datenbank . . . 43
8 run: Einlesen und Verschieben neuer XML-Dateien . . . 43

Daten-CD

Inhalt der Daten-CD:

arbeit	**Ausarbeitung der Bachelorarbeit**
arbeit/pfd	PDF-Version
daten	**Beispieldaten**
daten/egt	Basisdaten
daten/tml	Verkehrsdichtedaten
programm	**Java-Programm**
programm/	"TrafficDBConfig.xml" Konfigurationsdatei
programm/build	Compilierten Klassen des Java-Programms sowie Programmicons
programm/doc	Javadoc des Java-Programms
programm/external_jars	Externe Java Archiv-Datei (JAR)-Dateien
programm/src	Quellcode des Java-Programms
programm/WebContent	Daten für den Web-Container
software	**benötigte Software**
software/apache	Apache Tomcat
software/java	Java-SDK und Rundime-Umgebung
software/eclipse	Eclipse Ganymede incl. Web-Standard-Tools
web	**Beispiel-HTML-Seite für die Internetabfragen**
programm/html	HTML-Seiten
programm/war	"WAR-Datei" für Apache Tomcat

Die Beispieldaten stammen von der DDG und dürfen nicht vervielfältigt oder an Dritte weitergegeben werden.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Osnabrück, 2009