INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Erweiterung des OTN2SVG Systems um kartografische und TMC-Daten

Bo Quan

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dr. rer. nat., Dipl.-Inform. (Univ.) Bernhard Lorenz

Abgabetermin: 02. Mai 2007

Ziel dieser Projektarbeit ist, eine dynamisch aktualisierte SVG Karte zu generieren, die eingehende TMC Verkehrsnachrichten darstellt.

Kapitel 1 beschreibt die genaue Aufgabenstellung. In Kapitel 2 werden alle verwendeten Techniken vorgestellt, dazu zählen Traffic Message Channel (TMC), Scalable Vector Graphics (SVG), Shapefile, GeoTools und Java Document Object Model (JDOM). In Kapitel 3 wird auf die Implementierung eingegangen. Abschließend folgt in Kapitel 4 die Zusammenfassung und ein Ausblick auf zukünftige Entwicklungen. Kapitel 5 enthält Auszüge aus dem Sourcecode.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 02. Mai 2007

Bo Quan

Inhaltsverzeichnis

Inhaltsverzeichnis i

Abbildungsverzeichnis iii

1 Aufgabenstellung 1

1.1 Zielsetzung 1

1.2 Zu Grunde liegende Projekt- und Diplomarbeiten 1

1.2.1 „Ontologiebasiertes System zur Darstellung von Kartendaten in SVG“, Diplomarbeit 2004 Frank Ipfelkofer 2

1.2.2 „Datensammlung und -speicherung, sowie statistische Aufbereitung und Auswertung von TMC Daten“, Projektarbeit 2005 Markus Krieser, Michael Buschmann 2

1.2.3 „Transformation von TMC Daten in KML und Anbindung an Google Earth“, Projektarbeit 2006 Christian Hänsel 2

2 Technische Grundlagen 3

2.1 Traffic Message Channel (TMC) 3

2.1.1 Eigenschaften vom TMC 3

2.1.2 Geräte für TMC 4

2.1.3 Funktionsprinzip 4

2.2 Scalable Vector Graphics (SVG) 5

2.3 Shapefile 8

2.4 GeoTools 10

2.5 JDOM 10

3 Realisierung 13

3.1 UML Modell 14

3.2 Die TMC-Komponente . . . 15
3.2.1 Aufbau von TMCMessage . . . 15
3.2.2 Anbindung an TMC . . . 16
3.2.3 Datenbankabfrage . . . 17
3.2.4 SVG Karte generieren . . . 18
3.3 Die Shapefile-Komponente . . . 18
3.3.1 Daten extrahieren durch GeoTools . . . 19
3.3.2 Shapefile nach SVG konvertieren . . . 20
3.4 Integration . . . 21
3.5 JDOM . . . 22
3.5.1 XML Datei Laden . . . 22
3.5.2 Generieren einer SVG Datei . . . 23
3.5.3 Generieren einer XML Datei . . . 23
3.5.4 XPath abfragen . . . 23

4 Zusammenfassung und Ausblick 25
4.1 Zusammenfassung . . . 25
4.2 Ausblick . . . 25

5 Anhang - Auszüge aus den Sourcecode 27
5.1 Paket tmc.core . . . 27
5.2 Paket tmc.shp . . . 32
5.3 Paket tmc.util . . . 38

Literaturverzeichnis 39

Abbildungsverzeichnis

2.1 TMC Funktionsprinzip 4
2.2 TMC Geräte 5
2.3 Eine einfache SVG Datei 6
2.4 Sechs Grundelemente 6
2.5 Beispiel für ein SVG text-Element 7
2.6 Beispiel für ein SVG image-Element 7
2.7 Beispiel für ein SVG script-Element 9
2.8 JDOM Modell 11

3.1 Gesamtansicht Deutschlands und Detailansicht München 13
3.2 Detailansicht München mit Straßennetz 13
3.3 UML Modell 14
3.4 Komponenten Überblick 15

1 Aufgabenstellung

1.1 Zielsetzung

Aufgabe dieser Projektarbeit ist, eine dynamisch aktualisierte Scalable Vector Graphics (SVG) Karte zu generieren, die eingehende Traffic Message Channel (TMC) Verkehrsnachrichten darstellt. Es handelt sich um eine Stadt- oder Landkarte, auf der die TMC Verkehrsnachrichten angezeigt werden sollen. Um das zu realisieren, wird diese Projektarbeit auf drei vorhandenen Projekt- bzw. Diplomarbeiten aufgebaut, die am Lehrstuhl entwickelt wurden:

1. „Ontologiebasiertes System zur Darstellung von Kartendaten in SVG“, Diplomarbeit 2004, Frank Ipfelkofer [19]

2. „Datensammlung und -speicherung, sowie statistische Aufbereitung und Auswertung von TMC Daten“, Projektarbeit 2005, Markus Krieser, Michael Buschmann [20]

3. „Transformation von TMC Daten in KML und Anbindung an Google Earth“, Projektarbeit 2006, Christian Hänsel [18]

Das Ziel ist es, eine SVG Karte zu generieren. Zum einen ist das die Landkarte Deutschlands mit Landesgrenzen, Autobahnen und großen Städten, zum anderen die Karte, in der alle aktuellen TMC-Verkehrsnachrichten gekennzeichnet werden. Die beiden generierten Dateien müssen dann in das System von Frank Ipfelkofer eingebaut werden, so dass die Karte richtig dargestellt werden kann.

1.2 Zu Grunde liegende Projekt- und Diplomarbeiten

Die in diesen drei Arbeiten entworfenen Funktionen bzw. Daten werden gebraucht, um das oben genannte Ziel zu erreichen.

1.2.1 „Ontologiebasiertes System zur Darstellung von Kartendaten in SVG“, Diplomarbeit 2004 Frank Ipfelkofer

Es wurde ein System entwickelt, wo der Stadtplan von München in SVG dargestellt wird. Dieses System erweitert den SVG-Standard, so dass die SVG-Datei beliebig zoombar ist, ohne dass die Bildqualität verloren geht. Standardmäßig wird die Zoomstufe einer SVG-Datei auf 16-fach bzw. 1/16 eingeschränkt. Außerdem verfügt dieses System über einen Plugin-Mechanismus, um weitere Erweiterungen zu integrieren und die vorhandenen Funktionen verwenden zu können. Alle in dieser Projektarbeit erstellten SVG-Dateien müssen kompatibel zu dieser Diplomarbeit sein, so dass alles richtig funktionieren kann.

1.2.2 „Datensammlung und -speicherung, sowie statistische Aufbereitung und Auswertung von TMC Daten“, Projektarbeit 2005 Markus Krieser, Michael Buschmann

Bei dieser Projektarbeit wurde eine Datenbank angelegt, die statische Informationen wie z.B. die Koordinaten aller Städte bzw. Straßen Deutschlands enthält. Anhand des Elements „locationCode“ aus den TMC-Daten kann man durch eine Datenbankanfrage die jeweiligen Koordinaten erhalten, um alle Orte auf die Karte zu zeichnen.

1.2.3 „Transformation von TMC Daten in KML und Anbindung an Google Earth“, Projektarbeit 2006 Christian Hänsel

Es steht ein TMC-Cache zur Verfügung, der alle aktuellen TMC-Verkehrsnachrichten liefert. Die Daten liegen im XML-Format vor. Mit einer Socketverbindung kann man auf diesen Cache zugreifen.

2 Technische Grundlagen

Dieses Kapital bietet einen kurzen Überblick über die verwendeten Techniken:

- Traffic Message Channel (TMC) [17] ist ein Standard zur Übertragung von Verkehrsnachrichten
- Scalable Vector Graphics (SVG) [16] ist ein Dateiformat zur Darstellung von 2D-Vektorgrafiken in der XML-Syntax
- Shapefile [4] ist ein Dateiformat zur Darstellung von Geodaten
- GeoTools [9] ist ein Open-Source Java-API zur Handhabung räumlicher Daten
- JDOM [11] ist ein Java API zur Manipulation von XML-Datei

2.1 Traffic Message Channel (TMC)

TMC ist ein digitaler Radio-Datendienst, der über das Radio Data System (RDS) ausgestrahlt wird und zur Übertragung von Verkehrsinformationen an geeignete Empfangsgeräte verwendet wird. Die Idee ist, Benutzer jederzeit über die aktuellen Verkehrsinformationen zu informieren, um eine Umfahrung von Verkehrsstaus zu ermöglichen.

2.1.1 Eigenschaften vom TMC

- Datenempfang zu jeder Zeit möglich im Gegensatz zu herkömmlichen Verkehrsmeldungen der Radiosender
- Unterstützung verschiedener Sprachen
- Ausstrahlung von TMC innerhalb vieler europäischer Länder von zahlreichen Radiosendern
- Kostenfrei

2.1.2 Geräte für TMC

Inzwischen gibt es einige verschiedene Geräte im Handel, mit denen man TMC-Daten empfangen kann:

- Navigationsgeräte für das Auto, die auf Basis der aktuellen Verkehrslage Navigationsrouten vorschlagen
- Autoradios, die im Display die Meldungen anzeigen
- ein kleines Taschenradio, das man überall mit hinnehmen kann und das die aktuellen Meldungen im Display anzeigt
- eine Radiokarte für PC, die TMC-Daten auswerten und graphisch darstellen kann

2.1.3 Funktionsprinzip

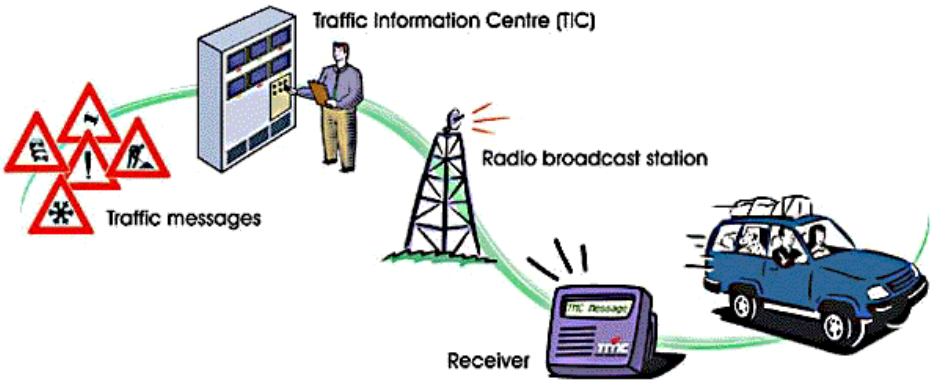

Abbildung 2.1: TMC Funktionsprinzip

Eine TMC-Meldung besteht aus einem „Event Code“ sowie einem „Location Code“, sowie eventuell einer zusätzlichen zeitlichen Beschränkung. Nach Wegfall einer Störung wird diese Störungsmeldung durch eine weitere TMC-Meldung aufgehoben. Der „Location Code“ wird in der Location Code List (LCL) national gepflegt. Jedes Land vergibt die Zahlen für die einzelnen Straßenabschnitte selbst.

Verkehrsinformationen (z.B Staus, Unfälle) werden typischerweise von der Polizei oder Verkehrskameras ermittelt. Die Daten werden dann durch das sogenannte Verkehrsinformationenszentrum mit dem Alert C-Standard kodiert und an Radio Sender gesendet. Der Alert C-Standard beinhaltet eine Liste mit ungefähr 1460 Ereignissen. Mit Hilfe dieser Liste kann die Meldung durch einen TMC De-

kodierer am Auto in eine vom Benutzer verständliche Form umgesetzt werden. Jede TMC-Message, mit einer eindeutigen ID, wird von dem TMC Dekodierer empfangen. Der TMC Dekodierer sucht in dieser Liste nach dieser ID und findet den entsprechenden Eintrag, der ausführliche Informationen enthält. Anschließend werden die Informationen im Klartext dargestellt.

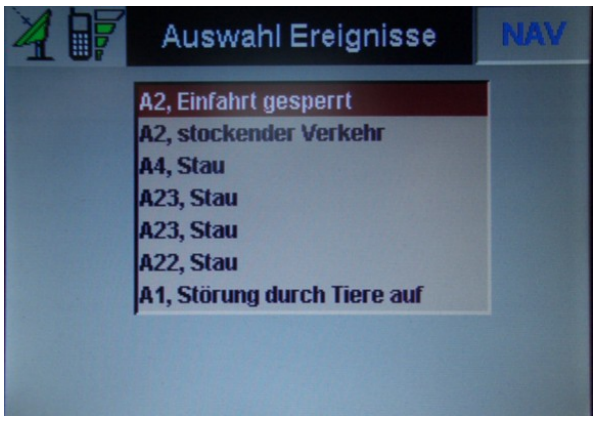

Abbildung 2.2: TMC Geräte

2.2 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) ist ein W3C-Standard zur Beschreibung von 2D Vektorgrafiken in der XML-Syntax. SVG wurde im September 2001 vom W3C als Empfehlung veröffentlicht und wird von einigen namhaften Software-Firmen (wie Adobe, HP, IBM, Macromedia, u.a.) vorangetrieben. Die aktuelleVersion ist 1.1.

SVG ist vollkommen konform zu XML, so dass ein Datenaustausch sehr einfach ist. Es fasst die beiden Vorläufer Precision Graphics Markup Language (PGML) und Vector Markup Language (VML) zusammen.

Eine SVG-Datei besitzt entweder die Dateiendung .svg oder, wenn sie mit gzip komprimiert ist, .svgz. Der MIME-Typ ist image/svg+xml.

Aufbau einer SVG Datei:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!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">
    <circle cx="205" cy="100" r="80" fill="yellow" stroke="red" />
</svg>
```

Die Ausgabe dieser SVG Datei ist in Abbildung 2.3 zu sehen:

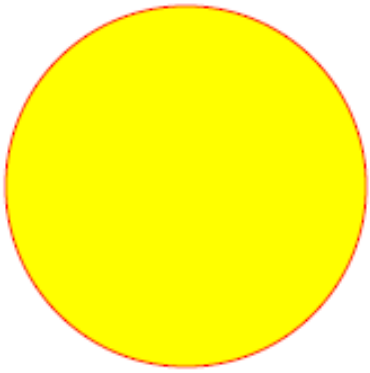

Abbildung 2.3: Eine einfache SVG Datei

Eine SVG-Datei kann in einen gewöhnlichen Browser angezeigt werden, wobei ein Plugin benötigt wird, wie z. B. den SVG-Viewer von Adobe.

SVG stellt 6 Grundelemente zur Darstellung von geometrischen Grundformen zur Verfügung:

Rechteck

```
<rect x="100" y="100" width="100" height="200" fill="green"/>
```

Kreis

```
<circle cx="100" cy="100" r="50" fill="green"/>
```

Ellipse

```
<ellipse cx="100" cy="100" rx="50" ry="20" fill="green"/>
```

Linie

```
<line x1="100" y1="100" x2="200" y2="200" stroke="green"/>
```

Polyline

```
<polyline points="100 100 200 200 300 40" stroke="green"
fill="none"/>
```

Polygon

```
<polygon points="100 100 100 200 150 200" fill="green"/>
```

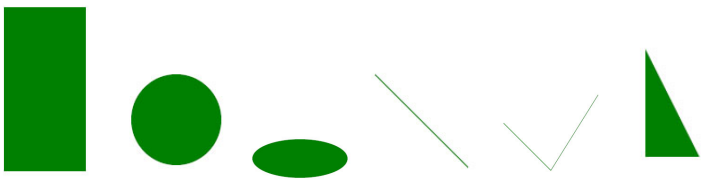

Abbildung 2.4: Sechs Grundelemente

Weitere SVG Elemente

Text

```
<text x="100" y="100" font-size="40px">SVG</text>
```

SVG

Abbildung 2.5: Beispiel für ein SVG text-Element

Hyperlink

```
<a xlink:href="http://svg.org/" xlink:title="http://svg.org/">
    <text x="410" y="300" font-size="40px">
    Hier steht die Text...
    </text>
</a>
```

Grafiken

```
<image x="50" y="50" width="244" height="249"
xlink:href="lmu.png"></image>
```

Abbildung 2.6: Beispiel für ein SVG image-Element

Transformationen

```
<rect x="0" y="0" width="100" height="30" rx="5" ry="5"
transform="translate(20,150) scale(.5)"/>
```

In SVG können Elemente oder Objekte (Gruppierungen oder Symbole) transformiert werden. Mit Hilfe des Transform-Attributs sind folgende Transformationen möglich:

- Verschieben
- Skalieren
- Rotieren
- Verzerren

Die verschiedenen Transformationen werden durch unterschiedliche Wertzuweisungen für das Attribut transform realisiert. Dabei ist es auch möglich mehrere Transformantionsanweisungen

hintereinander anzugeben.

Scripting und CSS Sie können in SVG Dokumenten Skriptsprachen wie ECMAScript, JavaScript, JScript oder VBScript verwenden, um Dynamik und Interaktivität zu verwirklichen. SVG Dokumente verwenden standardmäßig ECMAScript um die Funktionalität von Event-Attributen (wie z.B. onclick, onmouseover, etc.) zu gewährleisten.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!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">
    <defs>
        <!-- Script, hier wird Javascript verwendet-->
        <script type="text/javascript">
        <![CDATA[
            function meldung(){
                alert("Javascript in SVG");
            }
        ]]>
        </script>

        <!-- CSS Style -->
        <style type="text/css">
        <![CDATA[
            text {
            font-family:Verdana;
            font-size:16px;
            font-weight:bold;}
            rect {fill:#33cc33; stroke:black;}
        ]]>
        </style>

        <!-- button -->
        <g id="button">
            <rect x="0" y="0" width="100" height="30"
            rx="5" ry="5" />
            <text x="18" y="20">click me</text>
        </g>
    </defs>
    <!-- Methodaufruf durch Buttonklick -->
    <use xlink:href="#button" x="15" y="15"
    onclick="meldung();" />
</svg>
```

Wenn man auf die Schaltfläche klicken, erscheint ein Dialogfenster mit dem Text „Javascript in SVG“. Alle in HTML unterstützten Javascript Funktionen werden hier ebenfalls unterstützt.

2.3 Shapefile

Das Dateiformat Shapefile ist ein von Environmental Systems Research Institute (ESRI) ursprünglich für ArcView entwickeltes Format für Geodaten, wobei ArcView ein Geoinformationssystem (GIS) ist, mit dem sich Geodaten erzeugen, bearbeiten, analysieren und darstellen lassen. Es hat sich mittlerwei-

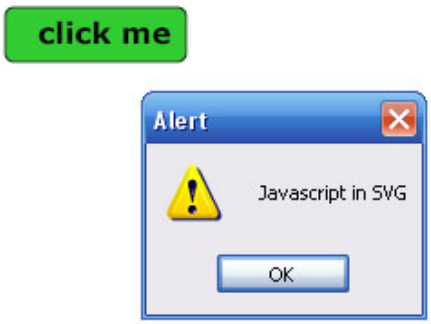

Abbildung 2.7: Beispiel für ein SVG script-Element

le zu einer Art Quasi-Standard im GIS-Umfeld verbreitet. Es ist ein Format, in dem viele Kartendaten zur Verfügung stehen. Die Unterstützung von freien sowie kommerziellen Tools & Bibliotheken (z.B. GeoTools [9]) ist bei keinem Format so gut wie bei dem Shapefile-Format. Es ist mit vielen Open Source Tools möglich, Daten von Shapefiles in eine „Spatial Datenbanken“ umzuwandeln.

Das Shape-Format besteht aus 3 Dateien:

.shp dient zur Speicherung der Geometriedaten

.shx dient als Index der Geometrie zur Verknüpfung der Sachdaten (auch Attributdaten genannt)

.dbf Sachdaten im dBase-Format. dBase war die erste weitgenutzte Datenbank-Applikation für Mikrocomputer und wurde von der Firma Ashton-Tate für das Betriebssystem CP/M vertrieben. Später wurde die Datenbank auf den IBM-PC unter MS-DOS portiert. [3]

In einem Shapefile können jeweils nur die Basis-Elemente (Simple-Features) eines GIS

- Punkte
- Linien
- Polygone

gespeichert werden. Dabei können nicht nur 2D-Objekte, sondern auch 3D-Geometrien erfasst werden.

2.4 GeoTools

GeoTools ist ein Open-Source Java-API zur Handhabung räumlicher Daten. Es wird von „Open Geospatial Consortium (OGC)“ in enger Zusammenarbeit mit den „GeoAPI“ [5] und „GeoWidgets“ [10] Projekten implementiert.

Geotools unerstützt verschiedenene Datenformate. Die Schnittstelle DataStore wird verwendet, um auf Vektordaten zuzugreifen, während die Schnittstelle GridCoverageExchange für den Zugriff auf die Rasterbilder zuständig ist.

Unterstützte Vektorformate:

Shapefile ESRI shapefile (R/W) [4]

GML Geography Markup Language (R) [6]

PostGIS geometrische Objekte für PostgreSQL (R) [15]

Oracle Spatial Oracle Extension für räumliche Daten (R) [14]

ArcSDE ESRI Middleware für räumliche Datenbanken (R) [4]

MySQL Unterstützung eines neuen geometrischen Typs [13]

GeoMedia ein Intergraph-Format (R) [7]

Unterstützte Rasterbildformate:

ArcGrid ArcInfo ASCII Grid format und GRASS ASCII Grid format (optionally compressed)[1]

GeoTIFF eine spezielle Form eines TIFF-Bildes (R) [8]

2.5 JDOM

JDOM steht für „Java Document Object Model“ und ist ein speziell für Java konzipiertes API zum Bearbeiten von XML. JDOM integriert die Vorteile von SAX und DOM.

Ähnlich wie beim Document Object Model (DOM) wird ein XML-Dokument als Baum im Hauptspeicher repräsentiert. Jedoch wird bei JDOM ein XML-Knoten nicht als „node“ repräsentiert, d.h. es gibt keine sogenannte Superklasse, von der sich alle anderen Knoten ableiten. Damit sind alle Knoten

(z.B Attribute, Kommentar, Document, Element) unabhängig voneinander, was dazu führt, dass die Suche schneller ist. JDOM selbst hat keinen eigenen Parser, stattdessen verwendet JDOM den SAX Parser. JDOM kann im Gegensatz zu SAX zu jeder Zeit auf jedes Element zugreifen.

Betrachten wir zunächst die Erzeugung, dann ist besonders die Klasse `SAXBuilder` hervorzuheben, die zur Erzeugung eines JDOM-Dokuments eine SAX-Implementierung bemüht. Die Klasse `DOMBuilder` wird für die gleiche Aufgabe einen DOM-Parser benutzt. Da alle zur Zeit existierenden DOM-Parser hinter den Kulissen ebenfalls SAX zum Parsen nutzen, macht die Verwendung dieser Klasse zum Parsen von XML keinen Sinn, da dies immer ineffizienter sein muss, als die zuvor besprochene Alternative.

Die Klasse `XMLOutputter` bietet die Standardmöglichkeit ein JDOM-Dokument als XML in einen beliebigen Ausgabestrom zu pumpen. Besonders interessant ist außerdem die Variante mit Hilfe der Klasse SAXOutputter entsprechend der Struktur des Dokuments SAX-Ereignisse auszulösen, die von einer anderen Komponente interpretiert werden können. Die vorangegangene Beschreibung soll durch die Abbildung 2.8 [12] verdeutlicht werden [2]:

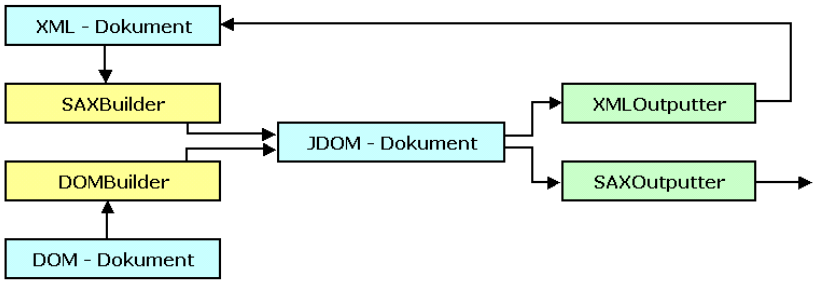

Abbildung 2.8: JDOM Modell

JDOM besteht aus 5 Paketen mit den Klassen zur Repräsentation des Dokuments und zum Einlesen und Ausgeben.

Das Paket org.jdom In diesem Paket sind alle Klassen zusammengefasst, damit ein XML-Dokument im Speicher repräsentiert werden kann. Dazu gehören zum Beispiel die Klassen Attribute, Comment, CDATA für Text, DocType, Document, Element, Entity und Processing-Instruction.

Das Paket org.jdom.input Im Input-Paket befindet sich dafür die Klasse DOMBuilder, die einen JDOM-Baum mit DOM erzeugt, und SAXBuilder, die dafür SAX verwendet. Beide Klassen

basieren auf der abstrakten Oberklasse AbstractBuilder, die Builder implementiert und drei build()-Funktionen angibt. Damit kann das Dokument aus einer Datei, einem Stream oder einer URL erzeugt werden. Nach dem Einlesen sind die Daten unabhängig von dem konkreten Parser des Herstellers und sie können weiterverarbeitet werden.

Das Paket org.jdom.output XMLOutputter übernimmt die interne Repräsentation und erzeugt eine XML-Ausgabe in einen PrintWriter. Daneben werden die unterschiedlichen Verarbeitungsstrategien DOM und SAX durch die Ausgabeklassen SAXOutputter und DOMOutputter berücksichtigt.

Das Paket org.jdom.transform Mit diesem Paket wird das JAXP-TraX-Modell in JDOM integriert. Dies ermöglicht JDOM die Unterstützung für XSLT-Transformationen von XML-Dokumenten.

Das Paket org.jdom.xpath Mit diesem Paket können XPath-Abfragen evaluiert werden, um die gewünschte Knoten zu finden.

3 Realisierung

Die komplette Applikation besteht aus 3 Komponenten: der TMC-Komponente, der Shapefile-Komponente und dem System von Frank Ipfelkofer. Zuerst werfen wir einen Blick auf das UML Modell. Dann gehen wir auf die Details jeder Komponente ein. Folgende zwei Abbildungen stellen verschiedene Ansichten dieser Applikation dar.

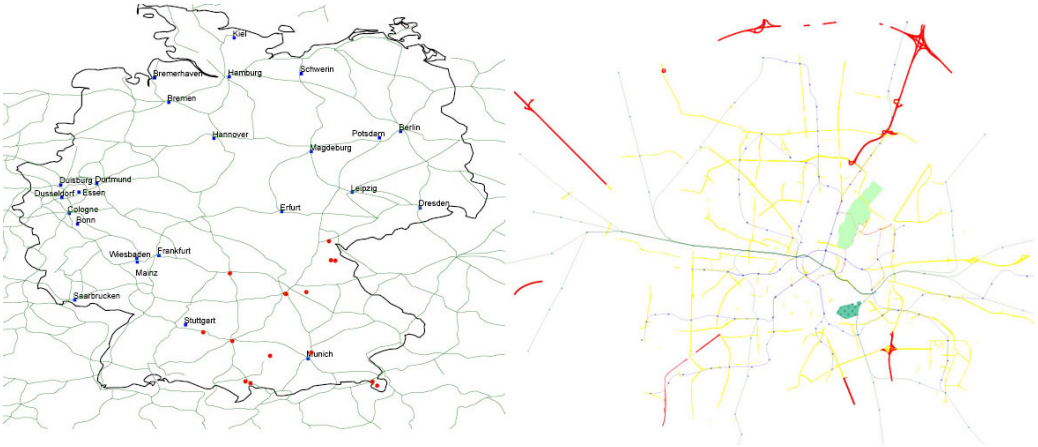

Abbildung 3.1: Gesamtansicht Deutschlands und Detailansicht München

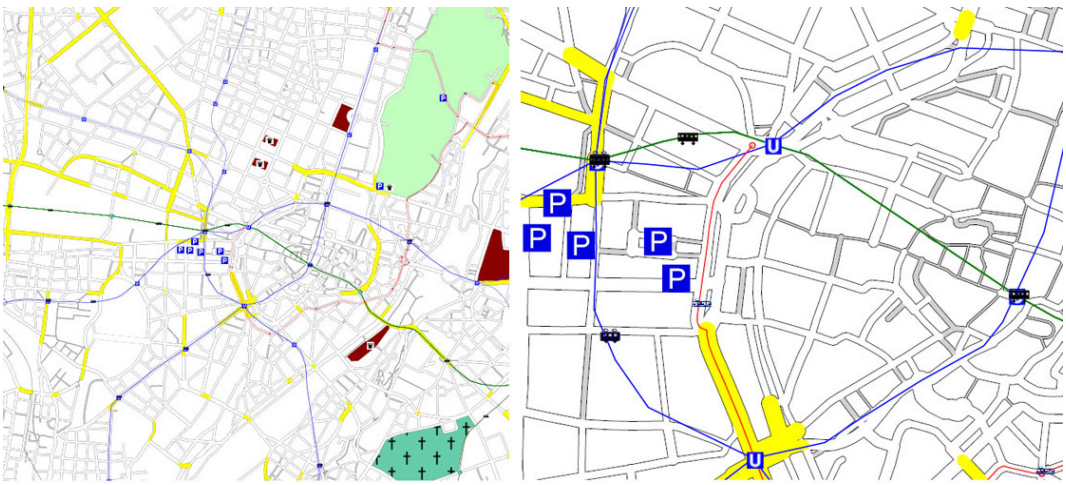

Abbildung 3.2: Detailansicht München mit Straßennetz

3.1 UML Modell

Der komplette Sourcecode wird in 3 Pakete unterteilt:

- tmc.core
- tmc.shp
- tmc.util

`tmc.core` ist für die TMC-Anbindung und Datenbankanfrage zuständig. Hier werden die eigentlichen TMC-Daten empfangen und bearbeitet. `tmc.shp` ist für den Zugriff auf das Shapefile zuständig. Die relevanten Daten werden in XML-Format gespeichert. Da es viele Dateizugriffsoperationen gibt, sollte es vermieden werden, die Pfade im Quellcode anzugeben. `tmc.util` bietet die Möglichkeit, die Hartcodierung zu vermeiden. Eine Datei `conf.properties` wurde angelegt, in der alle Pfade angegeben werden können. Damit es ist möglich, generierte Dateien sowohl unter Windows als auch unter Linux zu erstellen.

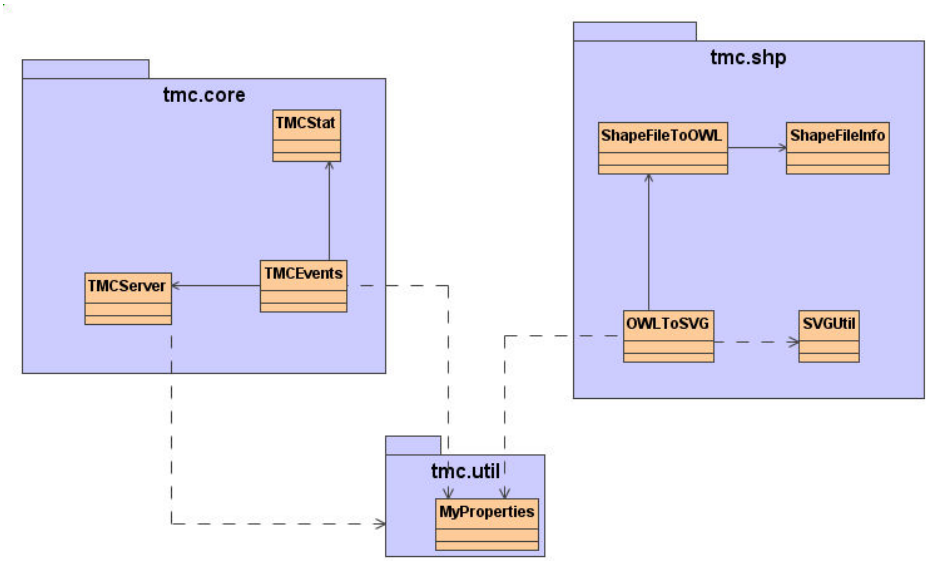

Abbildung 3.3: UML Modell

Die komplette Applikation besteht aus 3 Komponenten: der TMC-Komponente, der Shapefile-Komponente und dem System von Frank Ipfelkofer, wobei die ersten zwei unabhängig voneinander sind und die letzte angepasst werden muss.

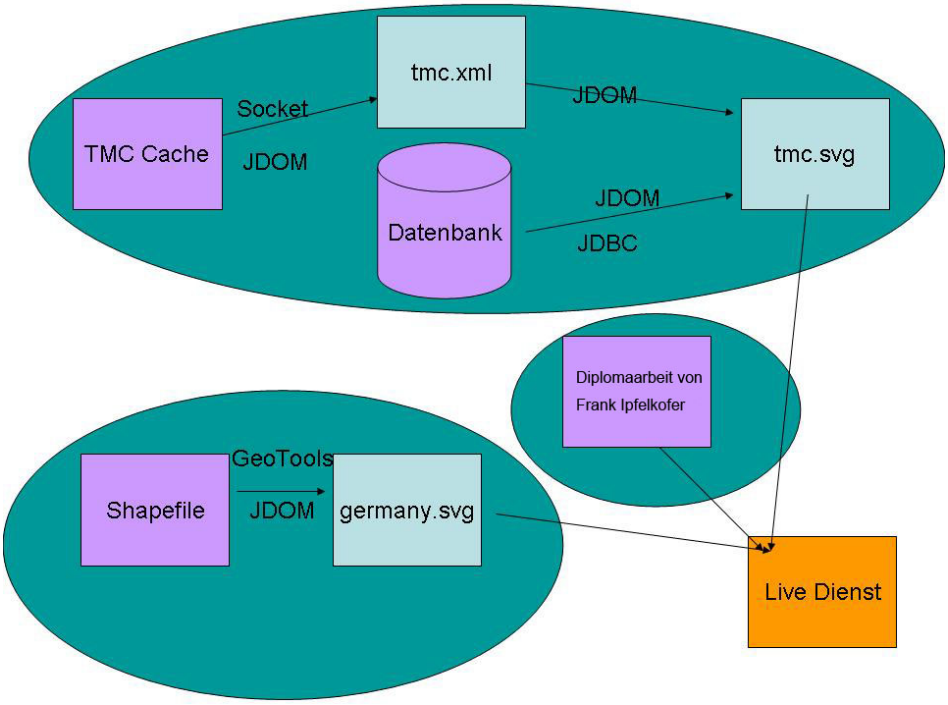

Abbildung 3.4: Komponenten Überblick

3.2 Die TMC-Komponente

Diese Komponente ist für den TMC-Empfang und die Generierung der SVG-Karte, in der die TMC-Verkehrsnachrichten dargestellt werden, zuständig. Das entsprechende Paket ist `tmc.core`.

Die aktuellen Verkehrsmeldungen befinden sich im TMC-Cache, der von Christian Hänsel [18] entwickelt wurde.

3.2.1 Aufbau von TMCMessage

Die im TMC-Cache erhaltenen Daten haben folgenden Aufbau:

```
<TMCCache>
  <TMCMessage>
    <meta>
      <id>3713</id>
      <timestamp>2006-06-18T20:05:55+0200</timestamp>
      <frequency>91.65</frequency>
    </meta>
    <primary>
      <event>
        <eventCode>101</eventCode>
        <eventName>Stau</eventName>
        ...
      </event>
```

```
      <location>
        <locationCode>34196</locationCode>
        <type>A1.0</type>
        <typeName lang="de">Kontinent</typeName>
        <location>
          <locationCode>1</locationCode>
          <locationName lang="de">Deutschland</locationName>
          <type>A3.0</type>
          <typeName lang="de">Land</typeName>
            <location>
              <locationCode>7210</locationCode>
              <type>L3.0</type>
              <typeName lang="de">Segment 1. Ordnung</typeName>
              <roadNumber>A8</roadNumber>
              <positiveEndName lang="de">München</positiveEndName>
              <negativeEndName lang="de">Salzburg</negativeEndName>
              <direction>positive</direction>

            </location>
        </location>
      </location>
    </primary>
  </TMCMessage>
  ...
</TMCCache>
```

Jede TMCMessage besteht aus zwei Teilen und zwar „meta“ und „primary“. „meta“ besteht aus „id“, „timestamp“, und „frequency“. „id“ kennzeichnet diese TMCMessage. „timestamp“ sagt aus, wann sich dieses Ereignis ereignete. „frequency“ steht für den Radiosender, der die TMCMessage ausstrahlt.

„primary“ enthält die eigentlichen Daten und besteht aus „event“ und „location“. „event“ beschreibt die Ereignisse, wie z.B. Stau oder Unfall. „location“ sagt aus, wo die Ereignisse sich ereigneten. Jede „location“ wird durch ein `locationCode`-Element gekennzeichnet, das der Datenbankabfrage zur Verfügung steht, um die tatsächlichen Koordinaten zu erhalten.

Die „location“ hat eine verschachtelte Struktur. Die oberste Ebene ist der Kontinent, die zweite ist das Land, in unserem Fall immer „Deutschland“. Es folgt die Autobahn, ein Segment der Autobahn und schließlich die Autobahnanbindung. z.B. Europa - Deutschland - A9 - Strecke von München nach Erding - Ausfahrt Erding.

3.2.2 Anbindung an TMC

Der erste Schritt ist, eine Verbindung zum TMC-Cache herzustellen.

Um darauf zuzugreifen, muss eine Socket-Verbindung aufgebaut werden. In unserem Fall hat die Socket-Verbindung den Host `localhost` und den Port `9779`. Wenn die Verbindung erfolgreich ist, werden die Daten in einem InputStream-Objekt gespeichert.

```
Socket cacheServer = new Socket("localhost", 9779);
InputStream input = cacheServer.getInputStream();
```

Anschließend kann man mit Hilfe von der Klasse SAXBuilder aus JDOM den Datenstrom in einem JDOM `Document` - Objekt speichern.

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(input);
```

Jetzt liegen die Daten im XML-Format vor, man kann sie bequem mit JDOM bearbeiten und verschiedene Operationen durchführen.

3.2.3 Datenbankabfrage

Der zweite Schritt ist, die Koordinaten aller in den TMC-Daten vorkommenden Orte festzustellen.

Diese TMC-Daten enthalten mehrere `locationCode`-Elemente von Ereignisorten. Um die tatsächlichen Koordinaten zu gewinnen, muss eine Datenbankabfrage durchgeführt werden. Bei der Projektarbeit von Markus Krieser und Michael Buschmann wurde eine Datenbank angelegt, in der statische Informationen wie z.B. die Koordinaten aller Städte bzw. Strassen Deutschlands gespeichert wurden. Wir führen folgende SQL-Abfrage durch, um die Koordinaten aller in TMC-Daten vorkommenden Orte festzustellen.

```
String sql="select x_coord, 5490560-y_coord from location where
locationcode = " + locationcode + " and x_coord >0 and y_coord >0";
```

In dieser Abfrage kennzeichnet `locationcode` jeden Ereignisort. Alle Koordinaten in der Datenbank beziehen sich auf den Bezugspunkt 0 Grad geographischer Länge und 0 Grad geographischer Breite. Dies entspricht dem mathematischen Koordinatensystem. Da sich Deutschland nördlich des Äquators und östlich des Nullmeridians (0 Grad Länge) befindet, sind alle x- und y-Koordinaten größer als Null. Da sich allerdings der Bezugspunkt des Koordinatensystems für SVG links oben befindet, müssen die y-Koordinaten entsprechend durch „5490560 - y“ umgerechnet werden. Die x-Koordinaten bleiben unverändert.

Durch die Bedingungen „x_coord $>$ 0 and y_coord $>$ 0“ wird die Darstellung auf den Bereich des minimalen umgebenden Rechteckes beschränkt, andernfalls würde der gesamte Kartenbereich rechts und oberhalb des Nullpunktes ausgegeben.

3.2.4 SVG Karte generieren

Der dritte Schritt ist, die SVG-Datei zu erstellen.

Mit der Klasse `XMLOutputter` kann man formatierte XML-Dateien erstellen. Eine saubere Formatierung der XML Datei wird durch `Format.getPrettyFormat()` veranlasst. Der Parameter `doc` ist das Objekt, das die Methode `build()` aus der Klasse `SAXBuilder` zurückgibt. Der Parameter `outputFile` ist der Pfad der Ausgabedatei.

```
XMLOutputter outp = new XMLOutputter();
outp.setFormat(Format.getPrettyFormat());
outp.output(doc, new FileOutputStream(outputFile));
```

Der letzte Schritt ist, die SVG-Datei in einem bestimmten Zeitintervall zu aktualisieren.

Das erfolgt in zwei Schritten. Zuerst müssen die TMC-Daten aus dem TMC-Cache erneut eingelesen werden, danach wird die SVG-Datei nochmal generiert. Auf diese Weise bleiben die Informationen auf der Karte immer aktuell.

```
while (true) {
        try {
            tmcserver.loadData();
            outputFile(tmcSVG, "590430 4738710 910430 751850");
            Thread.sleep(frequence);
        }
        catch (InterruptedException e){
            e.printStackTrace();
        }
    }
```

`loadData()` holt die Daten laufend aus dem TMC-Cache und schreibt sie in eine XML-Datei. `outputFile()` liest den Inhalt aus dieser XML-Datei und generiert dann die SVG Karte (siehe Anhang: Methode `getTMCMessage` in der Klasse `TMCEvents`). Dieser Vorgang wiederholt sich in einem bestimmten Zeitintervall, so dass die Karte immer die aktuellen Nachrichten anzeigen kann. `frequence` ist das Zeitintervall, dessen Wert man ändern kann, indem man die Datei `conf.properties` bearbeitet.

3.3 Die Shapefile-Komponente

Diese Komponente ist für das Erstellen der Karte Deutschlands zuständig. Das entsprechende Paket ist `tmc.shp`. Die eigentlichen Kartendaten stammen aus drei Shapefiles, jeweils für Landesgrenzen, Autobahnen und Städte. Um auf die Shapefile-Datei zuzugreifen, werden geeignete Werkzeuge

benötigt. In dieser Projektarbeit wird „GeoTools“ verwendet. Das Konvertieren der Shapefiles nach SVG erfolgt grundsätzlich in drei Schritten:

1. Datenentnahme mit Hilfe von „GeoTools“ aus den Shapefile-Dateien — ShapeFileInfo.java
2. Konvertieren ins XML-Format mit Hilfe von JDOM — ShapeFileToOWL.java
3. Konvertieren aus dem XML-Format ins SVG-Format mit Hilfe von JDOM — OWLToSVG.java

3.3.1 Daten extrahieren durch GeoTools

Mit Hilfe von GeoTools kann man in Shapefile enthaltene Daten extrahieren. Das erfolgt grundsätzlich in drei Schritten:

Shapefile laden

```
URL shapeURL = null;
try{
    shapeURL = (new URI(shapeFile)).toURL();
}
catch(URISyntaxException e){
    e.printStackTrace();
}
```

Zurerst wurde ein URL-Objekt anhand des Pfades des Shapefiles initialisiert. Die Variable `shapeFile` stellt hierbei den Pfad dar.

DataStore erzeugen Anhand der URL wird ein Shapefile DataStore-Objekt erzeugt, mit dem man auf das „feature“ Objekt zugreifen kann.

```
ShapefileDataStore store = new ShapefileDataStore(shapeURL);
String name = store.getTypeNames()[0];
FeatureSource source = store.getFeatureSource(name);
FeatureCollection fsShape = source.getFeatures();
```

Feature holen

```
// alle features holen
FeatureIterator reader = fsShape.features();
ArrayList al = new ArrayList();
// alle features durchgehen
while (reader.hasNext()) {
    Feature feature = reader.next();
    for (int i = 0; i < feature.getNumberOfAttributes(); i++) {
        Object attribute = feature.getAttribute( i );
        // nur non-Geometry Objekt
        if (!(attribute instanceof Geometry)){
            // nur Deutschland
            if(attribute.toString().equals("Germany")){
                if(!al.contains(feature))
                    al.add(feature);
            }
            fw.write(attribute.toString()+" ### \n");
        }
    }
}
```

Die Intstanz `reader` von der Klasse `FeatureIterator` enthält alle „features“ eines Shapefiles, d.h. die konkreten Eigenschaften aller geometrischen Objekte. Dazu zählen `Geometry` Objekte (Punkt, Linie usw.) und `non-Geometry` Objekte (Textbeschreibungen). Wir schauen jedes Feature-Objekt an, ob es ein `non-Geometry` Objekt ist. Wenn ja, vergleichen wir den Namen dieses Feature-Objekts, ob es `Germany` ist. Dazu brauchen wir hier lediglich die Geodaten von Deutschland.

Auszug der Ausgabe für die Landesgrenzen (`attribute.toString()`):

- Country Boundaries 3.6
- MULTIPOLYGON (((16.429721999999998 56.20888500000001, ...)),((...)), ...)
- Germany
- Weitere Ausgaben, die hier nicht verwendet werden (z.B. die Einwohnerzahl des Landes)

`MULTIPOLYGON (((...)),(( ... )), ... )` enthält die Koordinaten der Landesgrenzen, die wir auf die Karte zeichnen sollen. Das gilt analog für Städte und Autobahnen.

3.3.2 Shapefile nach SVG konvertieren

Nachdem wir alle Koordinaten aus Shapefiles entnommen haben, können wir die SVG-Datei erstellen.

In dem Paket `tmc.shp` befinden sich vier Dateien: `ShapeFileInfo.java`, `ShapeFileToOWL.java`, `OWLToSVG.java` und `SVGUtil.java`.

`ShapeFileInfo.java` liest die Daten aus der Shapefile-Datei und gibt ein ArrayList-Objekt zurück, das `ShapeFileToOWL.java` zur Verfügung steht. In der Datei `ShapeFileToOWL.java` werden drei Dateien erstellt, jeweils für Landesgrenzen, Städte und Autobahnen. Anschließend greift `OWLToSVG.java` auf die XML-Dateien zu und generiert die Landkarte. In `SVGUtil.java` werden viele Hilfsfunktionen definiert, die zur Manipulation von XML verwendet werden.

3.4 Integration

Das Plugin-Konzept der Diplomarbeit „Ontologiebasiertes System zur Darstellung von Kartendaten in SVG“ ermöglicht eine einfache Integration von weiteren Komponenten in das vorhande System. Die beiden generierten Dateien `germany.svg` und `tmc.svg` müssen dann in dieses System integriert werden, so dass die Karte richtig dargestellt werden kann.

```
<g minDetail="40000" maxDetail="2000000" module="TMC">
    <g transform="translate(-680000,-3100000) scale(60000)"
    loadUrl="maps/TMC/germany.svg"/>
    <g transform="translate(-680090,-393430) scale(0.6)"
    updatePeriod="20" loadUrl="maps/TMC/tmc.svg"/>
</g>
```

Es wird ein SVG Element `g` hinzugefügt, das wiederum zwei `g` Elemente enthält, und zwar für die Landkarte und die TMC Meldungen. `module` gibt dem Plug-In den Namen.

Die Attribute „minDetail“ und „maxDetail“ dienen dazu, die Ansicht zwischen Landkarte und Stadtplan zu wechseln, indem man die Karte vergrößert oder verkleinert.

Gewisse Objekte sind zu klein um in einem großen Ausschnitt wahrgenommen zu werden, (z.B. kleine Straßen, Straßennamen) und so ist es oft nicht sinnvoll, diese anzuzeigen. Auf einer Deutschlandkarte bietet es sich an, eine Stadt nur als einen Punkt mit einem Namen aufzuführen und erst ab einer gewissen Detailstufe den Punkt durch das genaue Straßennetz zu ersetzen. Hier besteht nun die Möglichkeit, bei jeder Art von Objekt anzugeben, ab welcher Detailstufe („minDetail“) und bis zu welcher Detailstufe („maxDetail“) es sichtbar sein soll.

Die Detailstufe ist die minimale Anzahl an Koordinateneinheiten, die gerade dargestellt werden. Wenn eine Karte den Bereich von (200, 400) bis (500, 1000) anzeigt, dann ist der sichtbare Bereich in x-Richtung 500 - 200 = 300 Einheiten und in y-Richtung 1000 - 400 = 600 Einheiten. Der minimale Bereich und damit die Detailstufe entsprecht dann dem Wert 300. (Siehe 4.2.3 in [19])

Hinzufügen des neuen TMC-Moduls:

```
<g minDetail="40000" maxDetail="2000000" module="TMC">
    <g transform="translate(-680000,-3100000) scale(60000)"
    loadUrl="maps/TMC/germany.svg"/>
    <g transform="translate(-680090,-393430) scale(0.6)"
    updatePeriod="20" loadUrl="maps/TMC/tmc.svg"/>
</g>
```

Wenn die sichtbaren Koordinateneinheiten dieser SVG-Datei kleiner als 40000 sind, wird diese Ansicht nicht mehr sichtbar. Andererseits wenn die sichtbaren Koordinateneinheiten größer als 2000000

sind, wird diese Ansicht ebenfalls nicht mehr sichtbar.

Detailstufe für Stadtansicht anpassen:

```
<g minDetail="0" maxDetail="40000">
    ...
</g>
```

Am Anfang ist die Landkarte sichtbar. Nach viermaligen Einzoomen ist die Landkarte unsichtbar, während der Stadtplan angezeigt wird. Auf diese Weise kann die Karte beliebig eingezoomt und ausgezoomt werden.

Darüber hinaus soll die viewBox von dem Wurzelelement SVG angepasst werden.

```
<svg preserveAspectRatio='xMinYMin meet' viewBox='0 0 267750 220710'>
```

Der Wert von viewBox hat folgende Bedeutung: Der erste Wert `0` steht für die x-Koordinate. Der zweite Wert `0` steht für die y-Koordinate. Der dritte Wert `267750` steht für die Breite der SVG-Datei. Der vierte Wert `220710` steht für die Höhe der SVG-Datei.

Mit diesen Werten kann die neue Karte richtig und vollständig im Browser angezeigt werden.

3.5 JDOM

JDOM wird an vielen Stellen verwendet, um neue Dateien anzulegen oder auf vorhandene Dateien zuzugreifen.

3.5.1 XML Datei Laden

Um mit JDOM auf eine XML-Datei zuzugreifen, muss zuerst das „Document“ Objekt erzeugt werden. Erst dann kann man auf Knoten, Text und Attribute zugreifen

```
public static Document loadDocument(String url){
    SAXBuilder builder = new SAXBuilder();
    Document doc = null;
    try{
        doc = builder.build(url);
    }
    catch(FileNotFoundException f){
        System.err.println(f);
    }
    return doc;
}
```

`url` ist der Pfad der zu ladenden XML-Datei.

3.5.2 Generieren einer SVG Datei

Hier wird das Gerüst einer SVG-Datei generiert und gibt ein JDOM-Document Objekt zurück.

```
public static Document createSVGFile(String viewBox){
    Element svgroot = new Element("svg");
    svgroot.setAttribute("viewBox", viewBox);
    Document doc = new Document(svgroot);
    doc.setDocType(new DocType("svg","-//W3C//DTD SVG 1.1//EN",
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"));
    return doc;
}
```

Das Gerüst einer SVG-Datei sieht so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!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">
    <!-- Inhalt -->
</svg>
```

Der Wert von `viewBox` wurde hier festgelegt, in unserem Fall ist „`0 0 267750 220710`“ (siehe Ende Kap. 3.4), damit die komplette Karte in der Mitte des Browsers angezeigt werden kann. Der `DocType` für SVG wurde auch hier explizit angegeben. Das ist notwendig, sonst wird diese Datei als normale XML Datei behandelt.

3.5.3 Generieren einer XML Datei

Letztlich wird die SVG-Datei auf Festplatte angelegt.

```
public static void save(Document doc, String file){
    XMLOutputter outp = new XMLOutputter();
    Format format = Format.getPrettyFormat();
    format.setEncoding("ISO-8859-1");
    outp.setFormat(format);
    outp.output(doc, new FileOutputStream(file));
}
```

Diese Methode erwartet zwei Parameter `doc` und `file`. Der Erste ist `doc`, der die Methode `createSVGFile()` zurückgibt. Der Pfad der Ausgabedatei wird durch `file` festgelegt. Eine saubere Formatierung der XML Datei wird durch `Format.getPrettyFormat()` veranlasst.

3.5.4 XPath abfragen

Um die gewünschten Knoten zu finden, die ein bestimmtes Kriterium erfüllen, kann man eine XPath-Anfrage durchführen.

JDOM besitzt eine Klasse XPath, mit der XPath-Ausdrücke ausgewertet werden können. Die Klasse wird über die statische Methode `newInstance()` instanziiert, der als String der XPath-Ausdruck übergeben wird. Mit der Methode `selectNodes()` kann dann ein Dokument oder Element als String übergeben werden. Das bzw. die Ergebnisse werden wieder in einem List-Objekt zurückgegeben.

```
public static Iterator evaluateXPath(Document doc, String path){
    XPath xpath = XPath.newInstance(path);
    List nodes = null;
    Iterator iter = null;
    try{
        nodes = xpath.selectNodes(doc);
        iter = nodes.iterator();
    }
    catch(NullPointerException f){
        System.err.println(f);
    }
    return iter;
}
```

Eine XPath Instanz wird anhand eines XPath-Strings `path` (z.B. `//locationCode`) initialisiert. Durch den Aufruf von `selectNodes()` wird die Anfrage durchgeführt und gibt eine Menge von Knoten zurück, oder `null`, wenn keine Knoten gefunden wurden.

Die vorangegangenen Beispiele verdeutlichen, wie man mit JDOM neue XML Dateien erzeugen bzw. existierende XML Dateien bearbeiten kann.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Das Ziel dieser Projektarbeit ist, eine SVG-Karte zu erstellen, die sowohl statische Geodaten als auch dynamische TMC Verkehrsnachrichten darstellt. Es bietet sich eine Stadtplanansicht und eine Landkartenansicht. Die Stadtplanansicht kommt aus der Diplomarbeit von Frank Ipfelkofer. Ich habe diese Arbeit um drei Funktionen erweitert:

- Erstellen einer Landkarte Deutschlands
- Darstellung von TMC Verkehrsnachrichten auf der Landkarte
- Umschalten zwischen den beiden Ansichten

Die TMC-Daten bleiben immer aktuell. Man kann das Programm sowohl unter Linux als auch unter Windows laufen lassen.

Die Darstellung in SVG ist sehr geeignet für eine Landkarte bzw. einen Stadtplan, da man die Karte beliebig auszoomen und einzoomen kann, ohne dass die Bildqualität verloren geht. Im Gegensatz zum Rasterbild ist die Dateigröße von SVG wesentlich kleiner. Die ursprünglichen Geodaten stammen aus dem Shapefile-Format, das ein im GIS-Umfeld weit verbreitetes Datenformat darstellt. Mit Hilfe von GeoTools kann man leicht auf das Shapefile-Format zugreifen und in andere Formate (z.B. XML) umwandeln. JDOM bietet effektive Bearbeitung von XML in Java.

4.2 Ausblick

Die Web-Oberfläche ist zur Zeit nicht besonders benutzerfreundlich. Folgende Verbesserungen wären denkbar:

- ein elegantes Aussehen der graphischen Oberfläche
- eine konfigurationsmöglichkeit der Update-Frequenz

- Anzeigen der Dauer aller Ereignisse
- Anzeigen der gegenwärtigen Zoomstufe

Mit Hilfe des Plugin-Konzepts der Diplomarbeit von Frank Ipfelkofer können weitere Module hinzugefügt werden. Da die Ausstrahlung von TMC inzwischen in vielen europäischen Ländern von zahlreichen Radiosendern erfolgt, ist es auch ohne weiteres möglich, den TMC-Datenempfang auf ganz Europa zu erweitern. Man kann selbst entscheiden, welche Meldungen aus welchen Ländern angezeigt werden sollen.

Es kann eine „History“-Funktion eingebaut werden. Damit hat man die Möglichkeit, einen Überblick innerhalb eines bestimmten Zeitraumes zu schaffen. Zum Beispiel es wäre sehr nützlich, die häufigste Unfallstelle oder Staustelle eines Monats herauszufiltern. Diese Funktion soll in der Lage sein, die Auswertung in einem Diagramm darzustellen, je nach Autobahnen oder Zeiträumen.

Obwohl wir in dieser Arbeit das Shapefile Format eingesetzt haben, ist es aber auch kein Problem, andere Geodatenformate zu verwenden.

5 Anhang - Auszüge aus den Sourcecode

5.1 Paket tmc.core

Klasse TMCServer

```
package tmc.core;

/**
 *This class builds up connection to the tmc server and
 *saves all lastest traffic information on local place
 */

public class TMCServer {
    private String tmcOutputFile;

    /**
     *Constructor
     *
     */
    public TMCServer() throws IOException,JDOMException, Exception{
        tmcOutputFile = MyProperties.read("tmcOutputFile");
    }

    /**
     *invoke the tmc service and save all information in a xml-file on a hard disk
     */
    public void loadData() throws IOException,JDOMException, Exception{
        System.out.println("TMC Meldungen werden empfangen ...");
        Document tmc = getTMCGeoData();
        outputStrom(tmc, tmcOutputFile);
    }

    /**
     *convert the xml-file in pretty format
     *@param doc   a JDOM <i>Document</i> object, which contains all tmc messages
     *@param outputFile   the name of the output file
     */
    public void outputStrom(Document doc, String outputFile) throws IOException{
        XMLOutputter outp = new XMLOutputter();
        outp.setFormat(Format.getPrettyFormat());
        outp.output(doc, new FileOutputStream(outputFile));
    }

    /**
     *building up connection to the tmc server and the read the datastream.
     *Finally save all in a <i>Document</i> object.
     *@return a JDOM <i>Document</i> object, which contains all tmc messages
     */
    public Document getTMCGeoData() {
        Document doc = null;
        try {
            Socket cacheServer = new Socket("localhost", 9779);
            InputStream input = cacheServer.getInputStream();
            SAXBuilder builder = new SAXBuilder();
```

```
            doc = builder.build(input);
            cacheServer.close();
            return doc;
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return doc;
    }
}
```

Klasse TMCEvents

```
/**
 *This class read all tmc traffic information and convert to svg-files
 *@author bo
 */
public class TMCEvents {

    private TMCStat tmcStat;
    private TMCServer tmcserver;
    private String tmcOriginalFile;
    private String tmcForOwl;
    private String tmcSVG;
    int frequence = 20000;
    /**
     *Constructor
     *TMCEvents initialize
     *@param tmcStat
     */
    public TMCEvents() throws JDOMException, IOException,Exception{
        tmcOriginalFile = MyProperties.read("tmcOriginalFile");
        tmcForOwl = MyProperties.read("tmcForOwl");
        tmcSVG = MyProperties.read("tmcSVG");
        frequence = Integer.parseInt(MyProperties.read("frequence"));
        tmcStat = new TMCStat();
        tmcserver = new TMCServer();
        updateTMC();
    }

    /**
     *periodic recieve the message from the server and update the SVG map
     */
    public void updateTMC() throws JDOMException, IOException,Exception {
        while (true) {
            try {
                tmcserver.loadData();
                System.out.println("TMC Meldungen erfolgreich aktualisiert!");
                outputFile(tmcSVG, "590430 4738710 910430 751850");
                Thread.sleep(frequence);
            }
            catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

    /**
     *parse a xml-file
     *@param url    URL of th file
     *@return A JDOM <i>Document</i> object
     */
    public Document loadDocument(String url) throws JDOMException, IOException{
        SAXBuilder builder = new SAXBuilder();
```

```
    Document doc = null;
    try{
        doc = builder.build(url);
    }
    catch(FileNotFoundException f){
        System.err.println(f);
    }
    return doc;
}

/**
 *Evaluates an XPath expression and returns the list of selected items.
 *@param doc the Document to use as context for evaluating the XPath expression.
 *@param path the XPath expression to evaluate.
 *@return the list of selected items
 */
public Iterator evaluateXPath(Document doc, String path) throws Exception{
    XPath xpath = XPath.newInstance(path);
    List nodes = null;
    Iterator iter = null;
    try{
        nodes = xpath.selectNodes(doc);
        iter = nodes.iterator();
    }
    catch(NullPointerException f){
        System.err.println(f);
    }
    return iter;
}

/**
 *generate the xml files, with which the svg map will be generated
 *@param file the original xml file, where all traffic info are stored
 *@param parentEle parent node
 */
public void getTMCMessage(String file, Element parentEle) throws Exception{
    Document doc = loadDocument(file);

    Iterator iter =
    evaluateXPath(doc,
    "//TMCMessage/primary/location/descendant-or-self::location[last()]/..");
    int index = 0;
    Element root = new Element("Root");
    try{
            while(iter.hasNext()){
                index ++;
                System.out.println(index + "-te TMC Message");
                Iterator event =
                evaluateXPath(doc,
                "//TMCMessage[position()="+ index +"]/primary/event/eventName");
                String eventName ="";
                String info ="";
                while(event.hasNext()){
                    Element en = (Element)event.next();
                    eventName += "-" + en.getTextTrim();
                    if(info==""){
                        info = "- " + en.getTextTrim();
                    }
                }
                Element item = (Element)iter.next();
                Element fromCity = item.getChild("positiveEndName");
                Element toCity = item.getChild("negativeEndName");
                Element street = item.getChild("roadNumber");
                String from = "---";
                String to = "---";
                String streetName = "---";
```

```
                String road = "- Unknown";
                if(fromCity != null && toCity != null && street != null){
                    from = fromCity.getTextTrim();
                    to = toCity.getTextTrim();
                    streetName = street.getTextTrim();
                    road =  "- Von " + from +" Nach " + to + ", auf " + streetName;
                }
                Element tmc = new Element("TMCMessage");
                Element von = new Element("Von");
                von.addContent(from);
                tmc.addContent(von);
                von = new Element("Nach");
                von.addContent(to);
                tmc.addContent(von);
                von = new Element("Ereignis");
                von.addContent(eventName);
                tmc.addContent(von);
                von = new Element("Strasse");
                von.addContent(streetName);
                tmc.addContent(von);
                Element track = new Element("Strecke");
                tmc.addContent(track);
                root.addContent(tmc);
                String path =  "//TMCMessage[position()="+ index +"]/primary/location/
                descendant-or-self::location[last()]/../location/locationCode";
                Iterator TMCMessages = evaluateXPath(doc, path);
                while(TMCMessages.hasNext()){
                    Element locationCode = (Element)TMCMessages.next();
                    String lc = locationCode.getTextTrim();
                    int[] pos = tmcStat.query(Integer.valueOf(lc));
                    if(pos[0] > 0 && pos[1] > 0){
                        Element lnEle =
                        locationCode.getParentElement().getChild("locationName");
                        String ln = "Unknown";
                        if (lnEle != null){
                            ln = lnEle.getTextTrim();
                        }
                        drawCircle(parentEle,Integer.toString(pos[0]),
                        Integer.toString(pos[1]),
                        "4000","red","stroke:green; stroke-width:4px;",
                        ln, road , info);
                        Element place = new Element("Ort");
                        place.addContent(ln);
                        track.addContent(place);
                    }
                }
            }
            outputXMLFile(root, tmcForOwl);
    }
    catch(NullPointerException f){
        System.err.println(f);
    }
}

/**
 *generate the XML file with the given root name and the file name
 *@param root root element name
 *@param outputFile the file name
 */
public void outputXMLFile(Element root, String outputFile)
 throws IOException,JDOMException{
    Document doc = new Document(root);
    XMLOutputter outp = new XMLOutputter();
    outp.setFormat(Format.getPrettyFormat());
    outp.output(doc, new FileOutputStream(outputFile));
  }
```

```
/**
 *generate the SVG map
 *@param outputFile    name of the SVG file
 *@param viewBox    viewBox of the SVG file
 */
public void outputFile(String outputFile, String viewBox)
throws IOException,JDOMException{
    Element svgroot = new Element("svg");
    Attribute svg = new Attribute("viewBox",viewBox);
    svgroot.setAttribute(svg);
    Namespace ns = Namespace.getNamespace("xlink", "http://www.w3.org/1999/xlink");
    svgroot.addNamespaceDeclaration(ns);
    addJavaScriptHead(svgroot,"SVG_DOCUMENT = document;RELATIVEROOT ='../../';");

    Element g = new Element("g");
    Comment s = new Comment(" Start of LOADNODE ");
    Comment e = new Comment(" End of LOADNODE ");
    g.addContent(s);
    getTMCMessage(tmcOriginalFile, g);
    g.addContent(e);
    svgroot.addContent(g);
    addJavaScript(svgroot,"../../scripts/ViewBox.js");
    addJavaScript(svgroot,"../../scripts/SVGLoader.js");
    addJavaScript(svgroot,"../../scripts/toolbar.js");

    //Comment f = new Comment(" file:maps/TMC/" + outputFile + " ");
    Comment f = new Comment(" file:maps/TMC/tmc.svg ");
    Document doc = new Document(svgroot);
    doc.setDocType(new DocType("svg","-//W3C//DTD SVG 1.1//EN",
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"));
    doc.addContent(f);
    XMLOutputter outp = new XMLOutputter();
    outp.setFormat(Format.getPrettyFormat());
    outp.output(doc, new FileOutputStream(outputFile));
    System.out.println("Neue TMC-Karte erfolgreich generiert!");
  }

/**
 *insert the javascript file reference in the SVG file
 *@param parent    parent node, where the codes are inserted
 *@param jsFile    path of the javascript file
 */
public void addJavaScript(Element parent, String jsFile){
    Element script = new Element("script");
    script.setAttribute("type","text/javascript");
    Namespace ns = Namespace.getNamespace("xlink", "http://www.w3.org/1999/xlink");
    Attribute href = new Attribute("href",jsFile,ns);
    script.setAttribute(href);
    parent.addContent(script);
}

/**
 *declare the global javascript variable
 *@param parent    parent node, where the codes are inserted
 *@param jsFile    declaration of the the global javascript variable
 */
public void addJavaScriptHead(Element parent, String text){
    Element script = new Element("script");
    script.setAttribute("type","text/javascript");
    script.addContent(text);
    parent.addContent(script);
}


/**
```

```
     *draw the point of the occurrence place
     *@param parentEle parent node
     *@param cx x coordinate circle
     *@param cy y coordinate circle
     *@param r radius of the circle
     *@param fill fill color of the circle
     *@param style css style of the element
     *@param sn locationName of the occurrence
     *@param info the reason of the traffic jam
     */
    public void drawCircle(Element parentEle, String cx, String cy, String r,
    String fill, String style,String sn, String road, String info)
    throws JDOMException, IOException{
        Element p = new Element("circle");
        p.setAttribute("cx",cx);
        p.setAttribute("cy",cy);
        p.setAttribute("r",r);
        p.setAttribute("fill",fill);
        p.setAttribute("style",style);
        p.setAttribute("onmouseover","TOOLBAR.Show(evt)");
        Element title = new Element("title");
        title.addContent(sn);
        title.addContent(new Element("br"));
        title.addContent(road);
        title.addContent(new Element("br"));
        title.addContent(info);
        p.addContent(title);
        parentEle.addContent(p);
    }

    /**
     *main method
     */
    public static void main(String[] args) throws JDOMException, IOException,Exception{
        TMCEvents tmc = new TMCEvents();
    }
}
```

5.2 Paket tmc.shp

Klasse ShapeFileInfo

```
package tmc.shp;

/**
 *This reads the data from the shapefiles
 *@author bo
 */
public class ShapeFileInfo{

    public ShapeFileInfo() throws Exception{    }

    /**
     *get the germany boundary daten from the shapefile
     *@param shapefile the shapefile, which represents the boundary
     *@param outputFile plain text file, that contains all shapefile daten of boundary
     *@return a list of important attributes of the boundary
     */
    public ArrayList getBoundaries(String shapeFile, String outputFile)
    throws Exception{
        System.out.println("Boundaries BEGIN");
        FileWriter fw = new FileWriter(outputFile,false);
```

```
    URL shapeURL = null;
    try{
        shapeURL = (new URI(shapeFile)).toURL();
    }
    catch(URISyntaxException e){
        e.printStackTrace();
    }

    // get feature results
    ShapefileDataStore store = new ShapefileDataStore(shapeURL);
    String name = store.getTypeNames()[0];
    FeatureSource source = store.getFeatureSource(name);
    FeatureCollection fsShape = source.getFeatures();

    // print out a feature type header and wait for user input
    FeatureType ft = source.getSchema();
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getType().getName()+" --- \n");
        }
    }
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getName()+" +++ \n");
        }
    }

    //now print out the feature contents (every non geometric attribute)
    FeatureIterator reader = fsShape.features();
    ArrayList al = new ArrayList();
    while (reader.hasNext()) {
        Feature feature = reader.next();//IllegalAttributeException
        fw.write(feature.getID()+" *** \n");
        for (int i = 0; i < feature.getNumberOfAttributes(); i++) {
            Object attribute = feature.getAttribute( i );
            if (!(attribute instanceof Geometry)){
                if(attribute.toString().equalsIgnoreCase("Germany")){
                    if(!al.contains(feature))
                        al.add(feature);
                }
                fw.write(attribute.toString()+" ### \n");
            }
        }
    }

    // and finally print out every geometry in wkt format
    reader = fsShape.features();
    while (reader.hasNext()) {
        Feature feature = reader.next();
        fw.write("\n\n wkt ============\n");
        fw.write(feature.getID() + " $$$ \t" + feature.getDefaultGeometry()+" @@@ \n");
    }
    reader.close();
    fw.close();
    System.out.println("Boundaries END");
    return al;
}

/**
 *get the germany city daten from the shapefile
 *@param shapefile the shapefile, which represents the city
 *@param outputFile plain text file, that contains all shapefile daten of city
 *@return a list of important attributes of the city
 */
```

```
public ArrayList getCities(String shapeFile, String outputFile)
throws Exception{
    System.out.println("City BEGIN");
    FileWriter fw = new FileWriter(outputFile,false);
    URL shapeURL = null;
    try{
        shapeURL = (new URI(shapeFile)).toURL();
    }
    catch(URISyntaxException e){
        e.printStackTrace();
    }

    // get feature results
    ShapefileDataStore store = new ShapefileDataStore(shapeURL);
    String name = store.getTypeNames()[0];
    FeatureSource source = store.getFeatureSource(name);
    FeatureCollection fsShape = source.getFeatures();

    // print out a feature type header and wait for user input
    FeatureType ft = source.getSchema();
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getType().getName()+" --- \n");
        }
    }
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getName()+" +++ \n");
        }
    }

    //now print out the feature contents (every non geometric attribute)
    FeatureIterator reader = fsShape.features();
    ArrayList al = new ArrayList();
    while (reader.hasNext()) {
        Feature feature = reader.next();//IllegalAttributeException
        fw.write(feature.getID()+" *** \n");
        for (int i = 0; i < feature.getNumberOfAttributes(); i++) {
            Object attribute = feature.getAttribute( i );
            if (!(attribute instanceof Geometry)){
                if(attribute.toString().equalsIgnoreCase("Germany")){
                    al.add(feature);
                }
                fw.write(attribute.toString()+" ### \n");
            }
        }
    }
    ArrayList pop = new ArrayList();
    for(int k = 0; k < al.size();k++){
        Feature f = (Feature)al.get(k);
        String popclass = f.getAttribute(12).toString();
        if(!pop.contains(popclass))
            pop.add(popclass);
    }

    //and finally print out every geometry in wkt format
    reader = fsShape.features();
    while (reader.hasNext()) {
        Feature feature = reader.next();
        fw.write("\n\n wkt ============\n");
        fw.write(feature.getID() + " $$$ \t" + feature.getDefaultGeometry()+" @@@ \n");
    }
    reader.close();
    fw.close();
```

```
    System.out.println("City END");
    return al;
}

/**
 *get the germany road daten from the shapefile
 *@param shapefile the shapefile, which represents the road
 *@param outputFile plain text file, that contains all shapefile daten of road
 *@return a list of important attributes of the road
 */
public ArrayList getRoads(String shapeFile, String outputFile)
 throws Exception{
    System.out.println("Roead BEGIN");
    FileWriter fw = new FileWriter(outputFile,false);
    URL shapeURL = null;
    try{
        shapeURL = (new URI(shapeFile)).toURL();
    }
    catch(URISyntaxException e){
        e.printStackTrace();
    }

    // get feature results
    ShapefileDataStore store = new ShapefileDataStore(shapeURL);
    String name = store.getTypeNames()[0];
    FeatureSource source = store.getFeatureSource(name);
    FeatureCollection fsShape = source.getFeatures();

    // print out a feature type header and wait for user input
    FeatureType ft = source.getSchema();
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getType().getName()+" --- \n");
        }
    }
    for (int i = 0; i < ft.getAttributeCount(); i++) {
        AttributeType at = ft.getAttributeType( i );
        if (!Geometry.class.isAssignableFrom(at.getType())){
            fw.write(at.getName()+" +++ \n");
        }
    }

    //now print out the feature contents (every non geometric attribute)
    FeatureIterator reader = fsShape.features();
    ArrayList al = new ArrayList();
    int m=0;
    while (reader.hasNext()) {
        m++;
        Feature feature = reader.next();
        fw.write(feature.getID()+" *** \n");
        for (int i = 0; i < feature.getNumberOfAttributes(); i++) {
            Object attribute = feature.getAttribute( i );
            if (!(attribute instanceof Geometry)){
                if(!al.contains(feature))
                        al.add(feature);
            }
            fw.write(attribute.getClass().toString()+
            "///"+attribute.toString()+" ### \n");
        }
    }

    //and finally print out every geometry in wkt format
    reader = fsShape.features();
    while (reader.hasNext()) {
        Feature feature = reader.next();
```

```
            fw.write("\n\n wkt \n");
            fw.write(feature.getID() + " $$$ \t" +
            feature.getDefaultGeometry()+" @@@ \n");
        }
        reader.close();
        fw.close();
        System.out.println("Road END");
        return al;
    }
}
```

Klasse ShapeFileToOWL

```
package tmc.shp;

/**
 *This store all information of shapefiles in xml format
 *@author bo
 */
public class ShapeFileToOWL {
    ShapeFileInfo sfi;
    int diff = 100;

    public ShapeFileToOWL() throws Exception{
        sfi = new ShapeFileInfo();
    }

    /**
     *store the important daten in xml format
     *@param file the name of the xml file
     *@param shpInput the original shapefile
     *@param shpSummary plain text file, that contains all shapefile daten of boundary
     *@return a list of important attributes of the boundary
     */
    public ArrayList createBoundaries(String file, String shpInput, String shpSummary)
    throws IOException, JDOMException, IllegalAttributeException, MalformedURLException{
        ArrayList result = new ArrayList();
        Document doc = SVGUtil.createFile("Shapefile");
        Element root = doc.getRootElement();
        ArrayList al = sfi.getBoundaries(shpInput, shpSummary);
        Feature f = (Feature)al.get(0);
        MultiPolygon mp = (MultiPolygon)f.getAttribute(0);

        Coordinate[] coor = mp.getCoordinates();

        for(int k = 0; k < mp.getNumGeometries();k++){
            Geometry g = mp.getGeometryN(k);
            Element boundary = new Element("Boundary");

            String points = "";
            Coordinate[] pp = g.getCoordinates();

            for(int l = 0; l < pp.length;l++){
                Element point = new Element("Point");
                point.addContent(pp[l].x+","+( diff - pp[l].y));

                points += pp[l].x+" "+( diff - pp[l].y);
                if(l != pp.length -1)
                    points += ", ";

                boundary.addContent(point);
            }
            result.add(points);
```

```
            root.addContent(boundary);
        }

        SVGUtil.save(doc, file);
        return result;
    }

    /**
     *store the important daten in xml format
     *@param file the name of the xml file
     *@param shpInput the original shapefile
     *@param shpSummary plain text file, that contains all shapefile daten of city
     *@return a list of important attributes of the city
     */
    public void createCities(String file, String shpInput, String shpSummary)
     throws IOException, JDOMException, IllegalAttributeException, MalformedURLException{
        Document doc = SVGUtil.createFile("Shapefile");
        Element root = doc.getRootElement();

        ArrayList al = sfi.getCities(shpInput, shpSummary);
        ArrayList pop = new ArrayList();
        for(int k = 0; k < al.size();k++){
            Feature f = (Feature)al.get(k);
            String popclass = f.getAttribute(12).toString();
            if(!pop.contains(popclass)){
                pop.add(popclass);
            }

            Element city = new Element("City");
            Point p = (Point)f.getAttribute(0);
            city.setAttribute("x", Double.toString(p.getX()));
            city.setAttribute("y", Double.toString(diff-p.getY()));
            city.setAttribute("name", f.getAttribute(5).toString());
            city.setAttribute("length", "0.06");
            root.addContent(city);
        }
        SVGUtil.save(doc, file);
    }

    /**
     *store the important daten in xml format
     *@param file the name of the xml file
     *@param shpInput the original shapefile
     *@param shpSummary plain text file, that contains all shapefile daten of road
     *@return a list of important attributes of the road
     */
    public ArrayList createRoads(String file, String shpInput, String shpSummary)
    throws IOException, JDOMException, IllegalAttributeException, MalformedURLException{
        ArrayList result = new ArrayList();
        Document doc = SVGUtil.createFile("Shapefile");
        Element root = doc.getRootElement();
        ArrayList al = sfi.getRoads(shpInput, shpSummary);

        for(int k = 0; k < al.size();k++){
            Element road = new Element("Road");
            Feature f = (Feature)al.get(k);
            MultiLineString    line = (MultiLineString)f.getAttribute(0);
            String points = "";
            Geometry g = line.getGeometryN(0);
            Coordinate[] pp = g.getCoordinates();
            for(int l = 0; l < pp.length;l ++){
                Element point = new Element("Point");
                point.addContent(pp[l].x+","+( diff - pp[l].y));

                points += pp[l].x+" "+( diff - pp[l].y);
                if(l != line.getNumPoints() -1)
```

```
                    points += ", ";
                road.addContent(point);
            }
            result.add(points);
            root.addContent(road);
        }
        SVGUtil.save(doc, file);
        return result;
    }
}
```

5.3 Paket tmc.util

Klasse MyProperties

```
package tmc.util;

/**
 *In order to avoid hard coding, e.g. input filenames and output filenames.
 *Alle these codes are written in the file <i>conf.proerties</i>.
 *@author bo
 */
public class MyProperties{

    /**
     *get the value of its corresponding key
     *@param key    the key of a proerties entry
     *@return         the value of its corresponding key
     */
    public static String read(String key){
        Properties properties = new Properties();
        try{
            properties.load(new FileInputStream("conf.properties"));
        }
        catch(FileNotFoundException nffe){
            nffe.printStackTrace();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
        return properties.getProperty(key);
    }
}
```

Literaturverzeichnis

[1] ARCGRID, 2007. http://docs.codehaus.org/display/GEOTOOLS/Articles.

[2] DAS JDOM-PROJEKT, 2007. http://www.fh-wedel.de/ si/seminare/ss01/Ausarbeitung/4.domjdom/dom3.htm.

[3] DBASE, 2007. http://de.wikipedia.org/wiki/DBase.

[4] ESRI SHAPEFILE TECHNICAL DESCRIPTION, 2007. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

[5] THE GEOAPI PROJECT, 2007. http://geoapi.sourceforge.net/.

[6] GEOGRAPHY MARKUP LANGUAGE (GML), 2007. http://www.opengeospatial.org/standards/gml.

[7] GEOMEDIA, 2007. http://de.wikipedia.org/wiki/Geomedia.

[8] GEOTIFF, 2007. http://de.wikipedia.org/wiki/GeoTIFF.

[9] GEOTOOLS, 2007. http://geotools.codehaus.org/.

[10] GEOWIDGETS, 2007. http://sourceforge.net/projects/geowidgets.

[11] JDOM, 2007. http://www.jdom.org/.

[12] JDOM MODELL, 2007. http://www2.informatik.hu-berlin.de/ simmchen/programmfluss.gif.

[13] MYSQL, 2007. http://www.mysql.com/.

[14] ORACLE SPATIAL, 2007. http://www.oracle.com/technology/products/spatial.

[15] POSTGIS, 2007. http://postgis.refractions.net/.

[16] SCALABLE VECTOR GRAPHICS (SVG), 2007. http://www.w3.org/Graphics/SVG/.

[17] WHAT IS TRAFFIC MESSAGE CHANNEL (TMC), 2007. http://www.tmcforum.com/.

[18] Christian Hänsel: PROJEKTARBEIT TMC-GOOGLEEARTH. Projektarbeit/project thesis, 2006. http://www.pms.ifi.lmu.de/rewerse-wga1/ldsms/publications/Projektdokumentation.pdf.

[19] Frank Ipfelkofer: BASISONTOLOGIE UND ANWENDUNGS-FRAMEWORK FÜR VISUALISIERUNG UND GEOSPATIAL REASONING. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2004. http://www.pms.ifi.lmu.de/publikationen/#DA_Frank.Ipfelkofer.

[20] Michael Buschmann Markus Krieser: PROTOKOLLIERUNG UND STATISTISCHE AUSWERTUNG VON RDS/TMC DATENSTRÖMEN, INSTITUTE OF COMPUTER SCIENCE, LMU, MUNICH, 2006. http://www.pms.ifi.lmu.de/publikationen/#PA_Markus.Krieser_Michael.Buschmann.