INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

# Basisontologie und Anwendungs - Framework für Visualisierung und Geospatial Reasoning

Frank Ipfelkofer

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Abgabetermin: 30. November 2004

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

# Basisontologie und Anwendungs - Framework für Visualisierung und Geospatial Reasoning

Frank Ipfelkofer

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Abgabetermin: 30. November 2004

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 30. November 2004

.....
(Unterschrift des Kandidaten)

Ziel dieser Arbeit ist es, ein Verkehrsnetz zu analysieren und in eine Ontologie abzubilden. Ein Framework soll daher die Daten auf einem Stadtplan sichtbar machen und die Möglichkeit schaffen, Antworten auf geographische Fragen zu geben.

Der derzeitige ISO-Standard zur Beschreibung von Straßennetzen, das *Geographic Data File* (GDF), reicht für manche der gestellten Aufgaben nicht aus. Daher wird nach der Einführung in die in der vorliegenden Arbeit verwendeten Websprachen und in die Ontologiesprache *Web Ontology Language* (OWL) in Kapitel 2 anschließend in Kapitel 3 mit *Ontology based Transport Network* (OTN) eine Ontologie entwickelt, die GDF in OWL modelliert und erweitert.

Das Kapitel 4 beinhaltet die Entwicklung und Funktionsweise eines auf OTN basierenden Frameworks sowie die Verwendung seiner Schnittstellen. Dabei ist das Hauptaugenmerk auf die Visualisierung gerichtet, doch wird auch eine Schnittstelle zur Anfragebearbeitung beschrieben, mit der bestimmte Daten (z.B. für eine Straßensuche) ermittelt und ausgewertet werden können.

Abschließend folgt in Kapitel 5 die Zusammenfassung und ein Ausblick auf zukünftige Entwicklungen. Die Beschreibung der in dieser Arbeit verwendeten und erstellten Dateien befinden sich im Anhang A.

# Inhaltsverzeichnis


**1 Einleitung** **7**
- 1.1 Aufgabenstellung . . . 7
- 1.2 Entwicklung und Aufbau der Arbeit . . . 8
  - 1.2.1 Entwicklung des Datenmodells . . . 8
  - 1.2.2 Entwicklung des Frameworks . . . 10

**2 Verwendete Sprachen** **15**
- 2.1 Grundlagen zur Adressierung und Speicherung von Daten . . . 15
  - 2.1.1 URI . . . 15
  - 2.1.2 XML . . . 16
- 2.2 SVG . . . 18
  - 2.2.1 Koordinaten in SVG . . . 19
  - 2.2.2 Aufbau eines SVG-Dokuments . . . 19
  - 2.2.3 Standardfiguren . . . 20
  - 2.2.4 Der SVG-Pfad . . . 21
  - 2.2.5 Gestaltung der Figuren mit Hilfe von Stilen . . . 22
  - 2.2.6 Gruppieren und Wiederverwenden von Figuren . . . 23
  - 2.2.7 Text . . . 24
  - 2.2.8 Manipulation des SVG-DOM mit ECMAScript . . . 24
  - 2.2.9 Events in SVG . . . 26
- 2.3 Ontologiebeschreibungssprachen . . . 28
  - 2.3.1 RDF . . . 28
  - 2.3.2 RDF-Syntax . . . 29
  - 2.3.3 RDF-Schema . . . 31
  - 2.3.4 OWL . . . 33

**3 Modellierung eines Verkehrsnetzes** **41**
- 3.1 GDF . . . 41
  - 3.1.1 Aufbau des Datenmodells . . . 41
  - 3.1.2 Themen in GDF . . . 43
  - 3.1.3 Allgemeine Eigenschaften . . . 45
  - 3.1.4 Das Straßennetz (*„Road and Ferries“*) . . . 46
  - 3.1.5 Das öffentliche Verkehrsnetz (*„Public Transport“ und „Railways“*) 51
  - 3.1.6 Hilfsbauten (*„Structures“*) . . . 51
  - 3.1.7 Landbelegung und -nutzung (*„Land Cover and Use“*) . . . 52
  - 3.1.8 Dienste (*„Services“*) . . . 52
  - 3.1.9 Zeitangaben . . . 54

3.1.10 Speicherung der Daten . . . 55
3.2 Ontologiebasiertes Verkehrsnetz . . . 55
3.2.1 Grundlegender Aufbau von OTN . . . 56
3.2.2 Orts- und Verkehrsmittelwechsel an einem *„Transfer_Point“* . . . 57
3.2.3 Geometrien . . . 58
3.2.4 Allgemeine Eigenschaften von Merkmalen . . . 59
3.2.5 Das Straßennetz . . . 60
3.2.6 Das Schienennetz und der öffentliche Verkehr . . . 64
3.2.7 Landbelegung und -nutzung . . . 67
3.2.8 Dienste . . . 67
3.2.9 Meteorology . . . 69

**4 Framework für Visualisierung und Geospatial Reasoning 71**
4.1 Aufbau des Frameworks . . . 71
4.2 Visualisierung der Daten in SVG . . . 72
4.2.1 Aufbau des SVG-Dokuments . . . 72
4.2.2 Rasterung der Daten . . . 75
4.2.3 Detailstufen . . . 77
4.2.4 Erweiterung der Gruppen von SVG . . . 77
4.3 Dynamische Karten in SVG mit Hilfe von Javascript . . . 78
4.3.1 Dynamisches Nachladen und Ausblenden . . . 78
4.3.2 Erweiterung des Zooms . . . 79
4.4 Transformations-Ontologie . . . 80
4.4.1 Transformationsformeln . . . 82
4.4.2 Auswahl und Transformation von Merkmalen . . . 83
4.4.3 Transformation geometrischer Merkmale . . . 84
4.5 Transformationsalgorithmen in Java . . . 85
4.5.1 Einlesen der Daten . . . 85
4.5.2 Transformation nach SVG . . . 87
4.5.3 Hilfsmittel zur Datenanalyse . . . 88
4.6 Konfiguration des Frameworks . . . 89
4.6.1 Konfiguration der SVG-Transformation . . . 89
4.6.2 Konfiguration des Servers zur Verwaltung von Anfragen . . . 91
4.7 Plugins . . . 91
4.7.1 SvgTransformationInterface . . . 91
4.7.2 AbstractOwlRequest . . . 92

**5 Zusammenfassung und Ausblick 95**

**A Beschreibung der verwendeten Dateien 97**
A.1 Dateien von OTN und der Transformationsontologie . . . 97
A.2 Dateien des Frameworks . . . 98
A.2.1 Beschreibung der Javasourcen . . . 99
A.3 Dateien zur Anzeige in SVG . . . 102

**B Auszüge aus dem Sourcecode** **105**
B.1 OTN-SVG-Transformation.dtd . . . . . 105
B.2 defaultConfig.xml . . . . . 106
B.3 OTN-Request.dtd . . . . . 108
B.4 requestConfig.xml . . . . . 108
B.5 Demo.svgz . . . . . 109
B.6 SVGLoader.js . . . . . 110
B.7 FakeWeather.java . . . . . 120
B.8 OTNStreetNameFinder.java . . . . . 126

**C Literaturverzeichnis** **131**

# Abbildungsverzeichnis


1.1 Aufbau und Funktionsweise des Frameworks . . . . . 11

2.1 Beispiele von kubischen Béziérkurven . . . . . 22
2.2 Verwendung des TextPath-Elements . . . . . 24
2.3 Beispiel zur Verwendung von Ereignis-Attributen . . . . . 27
2.4 Klassenhierarchie im RDF-Schema (aus [18]) . . . . . 32
2.5 Ausschnitt aus der Klassenhierarchie im OWL-Schema . . . . . 36

3.1 Aufbau der Merkmale in GDF . . . . . 42
3.2 In GDF definierte Merkmal-Themen . . . . . 44
3.3 Beispiel für die Levelarchitektur (Abbildung 5.17 aus [29] ) . . . . . 47
3.4 Landbelegung und -nutzung in GDF . . . . . 53
3.5 Aufbau der Daten in OTN . . . . . 56
3.6 Verkehrsmittelwechsel an einem *„Transfer_Point“* . . . . . 58
3.7 Beziehungen im Straßen- und Fährennetz in OTN . . . . . 60
3.8 Beispiel eines öffentlichen Verkehrsnetzes . . . . . 65
3.9 Ermittlung der Richtung einer Strecke in OTN . . . . . 66
3.10 Gruppeneinteilung von Diensten in OTN . . . . . 68

4.1 Prinzipieller Aufbau eines Frameworks . . . . . 71
4.2 Visualisierung in SVG . . . . . 73
4.3 Lage der Punkte, ihre Bereiche und der dazugehörige R-Baum . . . . . 76
4.4 Transformationsontologie für die Umwandlung von OTN nach SVG . . . 81

# 1 Einleitung

## 1.1 Aufgabenstellung

Sowohl im Internet als auch im Alltag (z.B. bei Navigationssystemen) existieren eine große Anzahl von Routenplanern, die Wege von einem Punkt zu einem anderen berechnen. Dabei spielen Kriterien, wie Verfügbarkeit, Schnelligkeit, Kosten oder die Entfernung eine große Rolle. Jedoch sind diese Anwendungen meistens nur auf eine einzige Art von Verkehrsmittel begrenzt. So gibt es einen Routenplaner für das Auto, einen für das öffentliche Nahverkehrsnetz, einen für die Bahn, einen für den Flugverkehr und wiederum einen anderen für Fußgänger. Diese Routenplaner arbeiten in den meisten Fällen nicht zusammen, so dass man, wenn man mehrere Verkehrsmittel verwenden will, eine Verbindung nach der anderen berechnen und die Ergebnisse dann evtl. sogar von Hand abstimmen muss. Das ist meistens sehr kompliziert und zeitaufwändig.

Ein Geschäftsreisender, der z.B. von Ingolstadt zu einem Geschäftspartner nach New York reisen will, hat viel vorauszuplanen. Er muss erst zum Flughafen nach München kommen, um dort in das Flugzeug nach New York steigen zu können. In New York kann er mit dem Taxi oder mit öffentlichen Verkehrsmitteln zu seinem Ziel gelangen. Benützt er öffentliche Verkehrsmittel und nur für die Reststrecke ein Taxi, hat er den jeweiligen Zeitbedarf zu ermitteln. Dafür muss er zunächst die nächstgelegene Haltestelle suchen sowie einen Routenplaner für die Metro, der ihm die Route und den Zeitverbrauch berechnet, anschließend, wieder zurückberechnend, einen geeigneten Flug von München nach New York, der ihm genügend Zeit lässt, um dort auch rechtzeitig an sein Ziel zu kommen. Schließlich ist auch noch der Weg zum Flughafen nach München zu berechnen und die Zeit zum Einchecken zu berücksichtigen.

Wenn alle diese unterschiedlichen Verkehrsanbindungen in einem großen System gespeichert wären, könnte der Benutzer mit wenig Aufwand die geeignetste Route erfragen, um zur gewünschten Zeit am Zielort zu sein. Dieses System würde ihm dann automatisch den kompletten Weg berechnen und auf einer Karte könnte er, wie gewohnt, die Route nachverfolgen.

Ein solches System muss in der Realität auf die Daten aus verschiedenen Verkehrsnetzen zugreifen können. Dazu ist jedoch eine Datenstruktur erforderlich, die sowohl die Details und Zusammenhänge aus einzelnen Verkehrsnetzen als auch die Zusammenhänge untereinander beschreibt. Erst damit kann eine Anwendung erstellt werden, welche die gewünschte Route über mehrere Verkehrsnetze hinweg berechnet.

Des Weiteren müssen entsprechende Schnittstellen existieren, über die ein einfacher Zugriff auf die Daten möglich ist, mit denen diese Daten zudem auf einer Karte dargestellt werden können und die eine Reihe von Anfragen zulassen, von der Darstellung des Zielortes auf einer Karte bis hin zur komplizierten Routenplanung über den gesamten Globus.

Bisher gibt es kein frei zugängliches Datensystem, das all diese Aufgaben lösen kann. Deshalb ist es das Ziel dieser Arbeit, ein solches zu erstellen, also ein Verkehrsnetz zu analysieren und in einer allgemein zugänglichen Ontologie abzubilden. Unter dem Begriff „Ontologie“ versteht man dabei ein formales System, das Regeln vorgibt zur Beschreibung von Dingen, Konzepten und deren Beziehungen zueinander. Mit Hilfe einer solchen Ontologie soll sowohl für den Straßenverkehr als auch für die öffentlichen Verkehrsmittel, wie Bus, Bahn oder Flugzeug eine Struktur geschaffen werden, in die sie abgebildet werden können. Des Weiteren soll es auch möglich sein, bestimmte Dienstleistungsunternehmen, wie Apotheken, Kinos oder Tankstellen darin abzubilden und deren Verkehrsanbindungen darzustellen.

In der vorliegenden Diplomarbeit ist mit *OTN* eine Ontologie entwickelt worden, die es ermöglicht, alle Bestandteile des weltweiten Transportnetzwerkes zu erfassen und über die darin definierten Zusammenhänge durch logische Kombination dieser Kriterien Rückschlüsse auf geographische Anfragen sowohl zu ziehen als auch automatisch zu beantworten, wie z.B. die Anfragen „alle Autobahnen nördlich von München“ oder „alle offenen Kirchen im Stadtzentrum“.

Ein Framework soll die Daten auf einer Art Stadtplan sichtbar machen. Unter dem Begriff „Framework“ versteht man ein Anwendungsprogramm, das als Gerüst für eine Aufgabe dient. Die grobe Architektur ist zwar bereits vorgegeben; für Funktionen, die darin noch nicht enthalten sind, besitzt ein Framework jedoch Schnittstellen, über die entsprechende Komponenten eingebunden werden und die dadurch Unteraufgaben übernehmen können.

Mit diesem Framework soll es auch möglich sein, solche Komponenten einzubinden, die Anfragen zu den Daten zulassen. Auf Fragen, wie z.B. „Wo ist die nächste Apotheke?“, liefern sie dann dank der vorhandenen Struktur geographische Schlussfolgerungen (*„Geospatial Reasoning“*).

## 1.2 Entwicklung und Aufbau der Arbeit

### 1.2.1 Entwicklung des Datenmodells

Die dieser Arbeit vorausgehenden Projektarbeit [17] beschäftigte sich mit dem Thema, Straßen in einem Straßennetzwerk zu modellieren und in SVG grafisch darzustellen. Dazu wurde ein einfaches, auf dem Austauschformat *Geography Markup Language* (GML) [15] basierendes Datenmodell entwickelt, das die Möglichkeit bietet, geographische Objekte durch ihre geometrischen Gegebenheiten zu beschreiben. Mit Punkten, Linien, Linienzügen und Polygonen standen vier Arten geometrischer Figuren zur Verfügung. Darauf aufbauend wurde eine Straße als ein Linienzug mit den Eigenschaften *„LeftLaneWidth“* und *“RightLaneWidth“* für die Breite der rechten und linken Straßenseite definiert.

Zur Modellierung eines komplexen Verkehrsnetzes reichen jedoch Straßen und ihre Breiten nicht aus. Deshalb wurden zu Beginn dieser Arbeit Überlegungen darüber angestellt, welche Objekte notwendigerweise zu einem Verkehrsnetz gehören und welche weiteren Eigenschaften festzulegen sind:

Generell werden die Verkehrswege unterteilt in:

**Straßen:** Sie sind Wege, die z.B. mit einem Auto oder mit einem Fahrrad befahren oder aber zu Fuß gegangen werden können. Unterschieden wird dabei z.B. zwischen Autobahn, Landstraße, Durchfahrtsstraße, Fußgängerweg, Radweg oder eine Kombination aus diesen.

**Luftwege:** Entlang dieses Weges findet Fortbewegung in der Luft statt, z.B. mit Flugzeugen.

**Schienen:** Mit Schienenfahrzeugen kann man sich darauf fortbewegen.

**Wasserwege:** Auf Schiffen z.B. kann man Wasserflächen auf meist festgelegten Routen überqueren.

Mit Hilfe von Bauten, wie Brücken oder Tunnels, können sich verschiedene dieser Wege kreuzen sowie Hindernisse, wie Wasser oder Berge, überquert werden.

Allen Wegen sind spezifische Eigenschaften zugeordnet, wie z.B. Fahrpläne, zeitliche Verwendbarkeit, Geschwindigkeitsbegrenzungen, Höhenbegrenzungen, Gewichtsbegrenzungen, Beschränkung auf eine Verkehrsteilnehmerart, Einengungen (z.B. Baustellen) oder Einschränkung der Fahrtrichtungen.

Innerhalb eines Verkehrsnetzes besteht die Möglichkeit der aktiven oder passiven Fortbewegung:

- Bei der aktiven Fortbewegung bestimmt man selbst die Art der Fortbewegung über das eigenständige Lenken eines Kraftfahrzeugs (z.B. PKW, LKW oder Motorrad), eines Fahrrads oder über das Gehen zu Fuß. Gleichzeitig legt man den ganz speziellen Weg fest, den man benutzen will.

- Bei der passiven Fortbewegung wird man gefahren. Als wichtigste Verkehrsmittel bietet sich dazu das Taxi, der Bus, das Flugzeug, das Schiff oder eine Bahn (u.a. U-Bahn, S-Bahn oder Trambahn) an. Außer bei der Benutzung eines Taxis ist man hierbei jedoch an feste Routen gebunden.

An bestimmten Orten, wie Flughafen, Bahnhof, Parkplatz, Haltestelle oder Taxistand kann man die Fortbewegungsart ändern oder das Fortbewegungsmittel wechseln. Fast immer muss man zum neuen Verkehrsmittel jedoch erst eine bestimmte Strecke zu Fuß zurücklegen, es sei denn, es handelt sich um ein Verkehrsmittel derselben Art (z.B. S-Bahn).

Ist man mit dem Taxi oder Fahrrad unterwegs, kann man an fast jedem Punkt einer Strecke zu Fuß weitergehen, insofern das Benutzen des Weges sowohl für Fußgänger als auch für die oben genannten Fahrzeuge erlaubt ist.

Genauso ist es möglich, als Fußgänger über den Taxiruf von beinahe jeder beliebigen Position des Weges auf ein Taxi umzusteigen, nicht jedoch auf ein Auto oder Fahrrad.

Diese Fortbewegungsmittel müssen entweder zu einem früheren Zeitpunkt an einem entsprechenden Ort bereitgestellt worden sein oder aber man muss sich zum Ausgangspunkt zurückbegeben oder zu einer Vermietstation.

Diese Vorüberlegungen verdeutlichen, welche Fähigkeiten das zu erstellende Datenformat aufweisen muss, um all die komplexen Zusammenhänge erfassen und umsetzen zu können.

Das Format GDF entspricht schon zu großen Teilen den gestellten Anforderungen, doch sind einige wichtige Ergänzungen erforderlich. Die Einbindung dieser Erweiterungen führt schließlich zu OTN.

### 1.2.2 Entwicklung des Frameworks

Zur Darstellung der Straßen auf einer Karte gab es in der Projektarbeit ein Javaprogramm, das den genauen Aufbau des Datenmodells kannte und mit diesem exakten Wissen der Datenstruktur Straßen in Javascript-Befehle umwandelte. Diese Befehle wurden im Browser, der die SVG-Karte beinhaltete, in einem neuen Fenster ausgeführt. Sie fügten dann eigenständig Daten zum SVG-DOM hinzu, ehe sie das geöffnete Fenster wieder schlossen. Es konnte zu einem bestimmten Zeitpunkt jedoch immer nur ein einziges neues Dokument aufgerufen werden. Die Anwendung war demnach so lange blockiert, bis alle Daten geladen waren. Mit diesem Konzept waren außerdem Erweiterungen des Datenmodells relativ kompliziert, da dafür sowohl neue Javaroutinen als auch neue Javascript-Funktionen geschrieben werden mussten. Des Weiteren musste eine einfache Anfrage, wie die Suche nach einer bestimmten Straße oder alle Daten über die Straßen sowie ihre Namen zum Client übertragen werden, was für große Datenmengen wenig sinnvoll war.

Es galt daher, ein Framework zu schaffen, das es ohne großen Aufwand ermöglichte, für jede Art von Objekten im Datenmodell Regeln zur Visualisierung anzugeben und daraus die gesamte Karte zu generieren. Da sowohl das Ausgangsformat OTN als auch das Zielformat SVG auf XML basieren, bot sich an, die Transformationsregeln in Form von XSLT-Stylesheets zu beschreiben und die Auswahl der zu transformierenden Objekte mit XPath durchzuführen. Zur Umwandlung geometrischer Merkmale oder einiger besonderer Eigenschaften (z.B. Fahrpläne) sind in der Regel sehr umfangreiche Berechnungen nötig. Daher wird eine solche Transformation, falls sie überhaupt realisierbar ist, relativ schnell unübersichtlich. Die Daten sind auf mehrere SVG-Dokumente zu verteilen, damit der Client nicht immer die gesamten Daten laden muss. Dies ist bei der Verwendung von XSLT schwierig, da das Resultat stets nur ein einziges Dokument darstellt.

Diese Überlegungen führten zu folgendem Ansatz: Ein Javaprogramm stellt zwar bestimmte Arten von Transformationen bereits standardmäßig zur Verfügung, kann jedoch über Regeln gesteuert werden. Die Transformationsregeln geben an, welche Objekte transformiert werden und wie das Resultat aussehen soll. Durch die Verwendung einer Programmiersprache wie Java, können dabei selbst komplizierteste Transformationen relativ einfach durchgeführt werden.

Abbildung 1.1 zeigt den Aufbau des dabei entstandenen Frameworks, das in Client und Server unterteilt ist. Der Client erhält vom Server ein HTML-Dokument, das u.a. eine

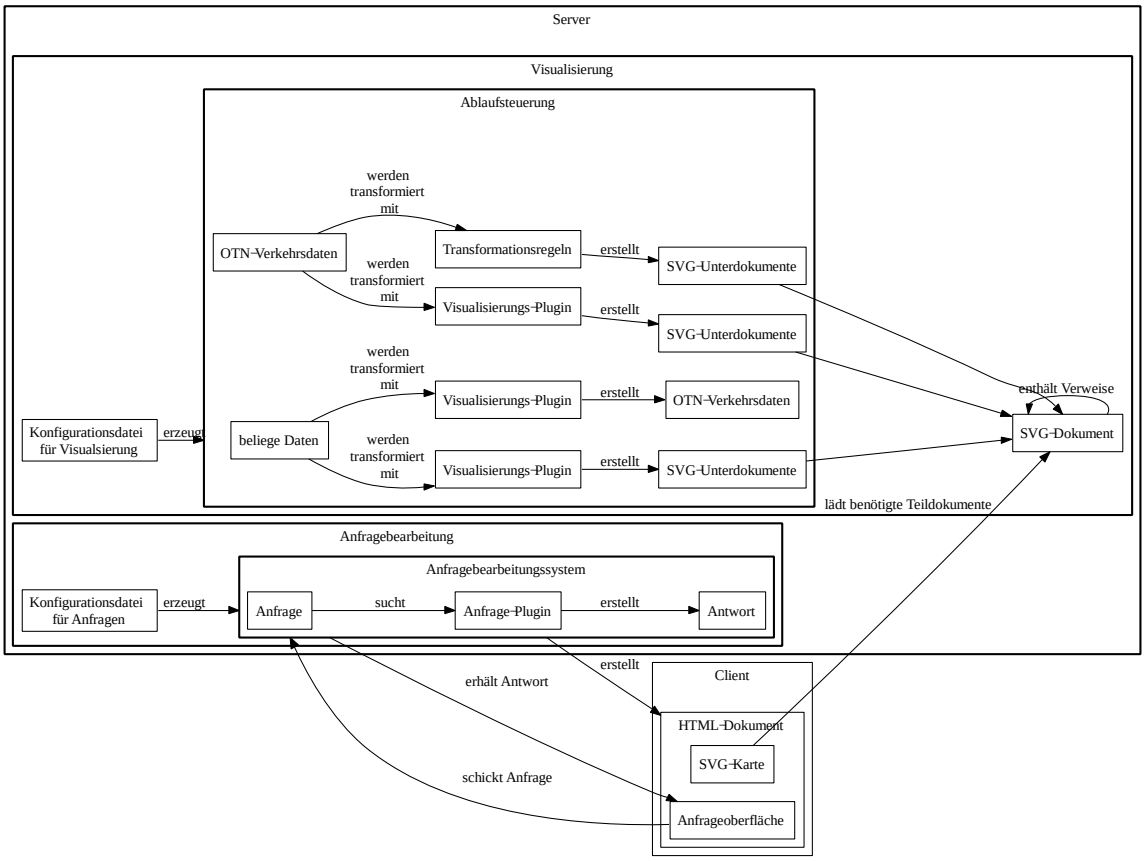



Abbildung 1.1: Aufbau und Funktionsweise des Frameworks

SVG-Karte und eine Anfrageoberfläche beinhaltet. Beide können mit dem Server kommunizieren. Der Server selbst besitzt zwei Hauptaufgaben, die man parallel ausführen kann:

**Anfragebearbeitung** leitet Anfragen an dafür geschriebene Plugins weiter, die diese dann ggf. beantworten.

**Visualisierung** wandelt beliebige Daten in SVG-Dokumente um, die auf der Karte angezeigt werden können.

### Anfragebearbeitung:

Um Anfragen beantworten zu können, muss der Server eine Art Webserver darstellen, der dynamische Antworten generieren kann. In dieser Arbeit ist dafür ein Apache Tomcat Webserver mit JSP-Unterstützung gewählt worden. Zur Beantwortung von Anfragen wird ein relativ einfacher Mechanismus verwendet: In einer Konfigurationsdatei für Anfragen werden all jene Plugins angegeben, an die Anfragen gestellt werden dürfen. Damit wird ein Anfragebearbeitungssystem aufgebaut, das diese verwaltet und an die entsprechenden Plugins weiterleitet.

Der Client fordert vom Server zu Beginn die anzuzeigende HTML-Seite an. Über die Schnittstelle zu den Plugins ist festgelegt, dass jedes Plugin Angaben zum gewünschten Aussehen der entsprechenden Anfrageoberfläche auf der HTML-Seite zu machen hat. Die Oberflächen werden nun zusammen mit der SVG-Karte auf der HTML-Seite zusammengesetzt und zum Client zurückgeschickt, der damit sowohl über eine Karte als auch über eine Reihe von Anfrageoberflächen verfügt. Bei Eingaben von Anfragen an diesen Oberflächen wird die Anfrage an das Anfragebearbeitungssystem auf dem Server geschickt, welches seinerseits das dafür zuständige Plugin heraussucht und die Anfrage weiterleitet. Nach der Beantwortung durch das Plugin wandelt das System die Antwort dann in eine für den Client verständliche Form um und schickt schließlich die Antwort zur Auswertung an den Client zurück.

**Visualisierung:**
Mit welchen Regeln und welche Daten nach SVG transformiert werden soll, wird in einer Konfigurationsdatei für Visualisierung festgelegt. Diese enthält u.a. Informationen über die zu transformierenden OTN-Dokumente wie auch über die Datei mit den Transformationsregeln, die auf diese Dokumente anzuwenden sind. Mit Hilfe dieser Daten wird ein Mechanismus (Ablaufsteuerung) geschaffen, der die Transformationen kontrolliert. Da bei den meisten Transformationen angegeben werden kann, wann sie erneut durchgeführt werden sollen, generiert dieser Mechanismus bei entsprechender Konfiguration im Hintergrund fortlaufend neue Dokumente. Nicht alle Transformationen sind über Regeln zu definieren. Darum besitzt das Framework eine Schnittstelle, mit der Plugins geschrieben werden können, die spezielle Umwandlungen durchführen. Weil diese nicht notwendigerweise an das Datenformat OTN gebunden sind, besteht damit sogar die Möglichkeit, neben der Generierung von SVG-Dokumenten aus OTN-Daten auch beliebige andere Datenformate umzuwandeln. Als Resultat kann dann entweder ein SVG- oder aber ein OTN-Dokument entstehen, das in einem weiteren Schritt (z.B. mit Hilfe von Transformationsregeln) nach SVG umwandelbar ist. Bei der Generierung entsteht eine Reihe von SVG-Dokumenten.

Durch die neu geschaffene Möglichkeit der Definition von Verweisen auf eingebundene Unterdokumente kann auf allen Dokumenten eine Baumstruktur erzeugt werden, dessen Wurzelelement rekursiv Verweise auf die Dokumente enthält. Da bei jedem Verweis auch angegeben werden kann, in welchem Bereich der Karte das Unterdokument liegt, bei welcher Auflösung es sichtbar sein soll und wie lange es ohne Aktualisierung gültig ist, kann die SVG-Karte, beim Wurzelelement beginnend, rekursiv alle Verweise durchlaufen und untersuchen, ob das entsprechende Dokument angezeigt werden soll. Ist dies der Fall und ist das Dokument bisher noch nicht angezeigt, wird es geladen. Ist es bereits in die Karte eingebunden, aber nicht mehr gültig, wird es entfernt und neu geladen. Eine im Adobe SVG-Viewer eingebettete Funktion ermöglicht es, mehrere Dokumente parallel zu laden. In einem neu geladenen Unterdokument werden ebenfalls rekursiv dann die darin enthaltenen Verweise solange untersucht, bis alle Dokumente im sichtbaren Bereich vorhanden sind und angezeigt werden können.

Als Baumstruktur wird der in Datenbanksystemen verwendete R-Baum eingesetzt.

Dieser sorgt dafür, dass nahe zusammenliegende Elemente in etwa gleich große Gruppen eingeteilt und jeweils in einem eigenen Dokument gespeichert werden. Durch die Einteilung aller Elemente in Bereiche werden zum größten Teil nur sichtbare Elemente geladen.

**Probleme und deren Lösung bei der Darstellung in SVG:**
Im Laufe der Arbeit entstanden hin und wieder Probleme dadurch, dass der verwendete SVG-Viewer bestimmten Einschränkungen unterliegt oder aber bestimmte gewünschte Funktionen nicht standardmäßig implementiert sind. Mit Javascript sollten diese Schwierigkeiten abgefangen oder gar umgangen werden.

Zwar besitzt der SVG-Viewer eine Funktion, mit der Dateien geladen werden können, doch gibt es keine Möglichkeit anzuzeigen, dass ein SVG-Dokument in ein anderes eingebettet ist und dieses somit bei Bedarf automatisch geladen wird. Dynamisches Nachladen ist also standardmäßig nicht möglich. Mit Javascript wird in regelmäßigen Abständen deshalb untersucht, ob Dateien geladen werden sollen. Sind neue geladen, werden diese im aktuellen Dokument an der Stelle eingefügt, an der ihr Verweis definiert ist.

Um herauszufinden, welche Dateien benötigt werden, muss man erst den sichtbaren Bereich ermitteln, d.h. den Bereich, der gerade im Fenster angezeigt wird. Jedoch besitzt SVG zur Abfrage des Bereichs lediglich eine Eigenschaft für die Verschiebung in x- und y- Richtung und eine für den Vergrößerungsfaktor. Mit diesen beiden Werten muss unter Berücksichtigung verschiedener Kriterien, wie u.a. Fenstergröße, Ausrichtung am Fenster und an der Größe des gesamten Kartenbereiches zuerst der sichtbare Bereich in Werten des zu Grunde liegenden Koordinatensystems bestimmt werden. Diese Berechnungen werden ebenfalls von einem Skript vorgenommen, so dass man den sichtbaren Bereich in Werten des Koordinatensystem abfragen und damit die richtigen Dateien laden kann.

Häufig sollen sich Objekte (z.B. die öffentlichen Verkehrsmittel wie Bus, Bahn und Tram) entlang eines vorgegebenen Weges bewegen. Dazu definiert SVG Animationen. Jedoch verbraucht die Berechnung der Animation auf der ohnehin schon rechenintensiven Karte so viel Ressourcen, dass weder die Animation flüssig dargestellt noch ohne lange Wartezeiten auf der Karte gezoomt oder gescrollt werden kann. Für die Darstellung von Bewegung ist also eine alternative Architektur erforderlich: Die Positionen der einzelnen Verkehrsmittel liegen in einem eigenen SVG-Dokument. Serverseitig ist die Generierung des Dokuments so angelegt, dass es im Abstand von wenigen Sekunden neu erstellt wird. Die Gültigkeitsdauer des Dokuments im Client beträgt ebenfalls nur wenige Sekunden (z.B. 5 Sekunden). Deshalb wird immer nach Ablauf der Zeit das aktuelle Dokument vom Server geladen. Da die Verkehrsmittel in diesem neuen Dokument eine andere Position besitzen, wird damit Bewegung simuliert. Meistens sind alle Verkehrsmittel einer Art in nur einem SVG-Dokument enthalten und jedes ist dabei etwa 2 KB groß. Bei fünf Verkehrsmittelarten werden demnach alle 5 Sekunden 10 KB übertragen, was jedoch kein Problem darstellt. Neben der daraus resultierenden Darstellung von Bewegung ohne gleichzeitige Blockierung der Karte ist die Möglichkeit, aktuelle Daten der öffentlichen Verkehrsmittel auszuwerten und damit die genaue Position sowohl zu bestimmen als auch anzuzeigen, ein weiterer Vorteil.

Der SVG-Viewer von Adobe unterstützt nur Vergrößerungen und Verkleinerungen um einen maximalen Faktor von 16. Wenn eine große Karte dargestellt werden soll, reicht dies jedoch nicht aus. Bei hohen Vergrößerungsstufen wird zudem die Karte, vermutlich aufgrund eines Überlaufs, ab einer bestimmten Stelle nicht mehr gezeichnet. Mit einer neu entwickelten Befehlskette ist es im Javascriptprogramm nun möglich, jedes Zoom- und Scroll-Ereignis abzufangen, die Karte durch Manipulation der Eigenschaft „viewBox“ auf den gewünschten Ausschnitt zu setzen und dann den internen Zoom zurücksetzen, so dass sich der SVG-Viewer immer in derselben Zoomstufe befindet und sich lediglich die anzuzeigende Karte in Position und Größe verändert.

Oft ist es praktisch, zu bestimmten Objekten einen Tooltip anzuzeigen. Auch dies ist ebenfalls keine Standardfunktion von SVG. Über Javascript kann in vorliegender Arbeit der Tooltip bei Bedarf generiert und an der aktuellen Mausposition platziert werden. Jedoch ist dabei darauf zu achten, dass die Schrift immer in gleicher Größe dargestellt wird. Ein eigens dafür entwickelter Mechanismus vergrößert oder verkleinert die Schrift je nach Zoomstufe, so dass die Größe am Bildschirm stets konstant ist. Mit dessen Hilfe wäre es zwar auch möglich, beliebige Objekte immer in derselben Größe darzustellen. Dies findet bisher jedoch noch keine Verwendung.

Die Unterstützung von Javascript ist ein wichtiges Werkzeug, neue Funktionen in SVG zu erschaffen und Fehler oder Probleme darin zu umgehen. Jedoch ist zu wünschen, dass in zukünftigen Versionen von SVG selbst wenigstens ein Teil dieser Mängel behoben wird. Denn das erleichtert das Erstellen von dynamischen Dokumenten doch ganz erheblich.

# 2 Verwendete Sprachen

Das *World Wide Web Consortium* (W3C) erarbeitete eine Reihe von Technologien, die zueinander kompatibel sind und dafür sorgen, dass die Softwareentwickler im Internet gewisse Richtlinien befolgen. Mittels der eigens dafür entwickelten Sprachen können Daten und Informationen über das Internet publiziert werden, da sie einem Standard entsprechen, für den es in den meisten Betriebssystemen die entsprechenden Programme gibt. Nachfolgend werden die für die vorliegende Diplomarbeit wichtigsten Standards vorgestellt und deren Verwendung erläutert.

## 2.1 Grundlagen zur Adressierung und Speicherung von Daten

### 2.1.1 URI

In einem großen Netzwerk existiert eine riesige Menge an Ressourcen. Unter einer *Ressource* versteht man alles, was wiederauffindbar ist, dem also eine Identität zugeordnet werden kann. Im Internet sind es u.a. Webseiten oder andere Daten. Aber auch Email-Empfängern, Personen oder Firmen kann eine Identität zugeordnet werden und somit sind auch diese Ressourcen. Ein *Uniform Resource Identifier* (URI) [1] ist eine Zeichenfolge zur eindeutigen Benennung und Adressierung von Ressourcen, damit diese ausgetauscht und überall verwendet werden können. Sie unterliegen dabei einem vordefinierten Aufbau:

```
<Schema>:<Schemata-spezifischer Teil>
```

Das Schema beschreibt die Art der Ressource, die verwendet wird. Dabei gibt es bereits eine Reihe vordefinierter Schemata. Die vollständige Liste findet sich unter [2]:

- http : Hypertext Transfer Protocol - Adressierung von Webseiten
- ftp: File Transfer Protocol - Adressierung von Daten und Dateien
- mailto: Email Adresse
- urn: Uniform Resource Names (URN)
- tel: Telefonnummer

Der „Schemata-spezifische Teil“ gibt die Adresse der Ressource bzgl. des Schemas an. Viele der Schemata, wie etwa http oder ftp, besitzen dabei folgenden Aufbau:

```
<Schema>://<Server>[:<Port>]/<Pfad,...>?<Anfrage>
```

An URI kann optional noch mit „#“ ein Fragmentbezeichner angehängt werden. Diese Kombination wird auch URI-Referenz genannt.
Es gibt zwei Unterarten von URI:

**Uniform Resource Locator (URL)** gibt den Ort einer Ressource im Netz an. Die wichtigsten Beispiele hierfür sind die weit verbreiteten http oder ftp Adressen. Ursprünglich waren URLs die einzige Form von URIs und deswegen werden oft die Begriffe URI und URL gleichbedeutend verwendet.

**Uniform Resource Name (URN)** beginnt immer mit dem Schema „urn“, gefolgt von einem bestehenden oder frei zu vergebenden Namen, z.B. urn:isbn, der die genauere Art der Ressource definiert. Damit kann jede Ressource identifiziert werden.

Durch die Verwendung von URLs zur Adressierung von Webseiten entsteht häufig der Eindruck, dass eine URI im Web angesprochen werden kann. Dies ist jedoch ein Irrtum, denn eine URI ist lediglich ein Identifikator für eine Ressource und daher nicht notwendigerweise eine Adresse, mit der auf eine Repräsentation der Ressource zuzugreifen ist.

### 2.1.2 XML

Die zur Darstellung, Konfiguration und Speicherung der Daten benutzten Sprachen basieren auf XML [4], einem im Internet weit verbreiteten Standard des W3C, der seit Februar 2004 in der dritten Ausgabe erschienen ist. XML steht für Extensible Markup Language und bedeutet frei übersetzt „erweiterbare Auszeichnungssprache“. Erweiterbar heißt hier, dass XML keine komplett vordefinierte Sprache ist, wie beispielsweise HTML, sondern eine Metasprache, die zur Beschreibung einer anderen Sprache benutzt wird. Mit ihrer Hilfe kann man eine für die Anwendung spezifische individuelle Sprache definieren, um damit Informationen zu speichern oder auszutauschen.

Vereinfacht ausgedrückt handelt es sich bei XML um ein System verbindlicher Regeln, mit denen Informationen gekennzeichnet oder „etikettiert“ werden können. Es ist leicht, XML-Dokumente auf ihren korrekten formalen Einsatz zu testen und damit Programme zu schreiben, welche die Daten automatisch verarbeiten. Um damit umgehen zu können, muss man nicht den genauen Aufbau des Dokuments kennen, sondern nur wissen, welche Strukturen es besitzt. Im Gegensatz zu vielen anderen Formaten sind in XML die Daten in einer Form angegeben, die auch vom Menschen beim Lesen verstanden werden kann.

Um zu zeigen, dass es sich um ein XML-Dokument handelt, beginnt dieses immer mit der Deklaration `<?xml version='1.0'?>`. Oft wird dabei noch der verwendete Zeichensatz angegeben, so dass das Dokument auf der ganzen Welt einheitlich verarbeitet werden kann. Häufig folgt die Angabe einer *Document Type Definition* (DTD) zur Definition der Regeln, denen das Dokument entsprechen soll, damit es sich um ein gültiges Dokument handelt. Anschließend folgt ein *„Element“*, das auch *„Wurzelelement“* genannt wird und von dem es nur genau eines geben darf. Jedoch kann dieses beliebig viele *„Unterelemente“* enthalten, die ihrerseits wiederum eigene Unterelemente enthalten können. Bei jedem Element werden Anfang und Ende durch einen sogenannten *„Tag“* markiert, d.h. einen

Elementnamen, der in spitzen Klammern steht. Der Slash „/“ vor dem Elementnamen zeigt an, dass es sich um einen *„Endtag“* handelt. Damit hat ein Element die Form:

```
<ElementName>Inhalt des Elements</ElementName>
```

Der Inhalt der spitzen Klammer kann sich aus einer Kombination von Text und anderen Elementen zusammensetzen, wobei ein Teil des Textes (z.B. Zeilenumbrüche und Leerzeichen) zum Teil ignoriert wird, damit man die Elemente lesbar einrücken kann. Enthält ein Element nur Text und keine weiteren Elemente, so wird er auch Textknoten genannt. Im Text dürfen fast alle Zeichen des definierten Zeichensatzes verwendet werden. Es gibt einige Ausnahmen, wie z.B. „<“ oder „>“, die bereits für das „Markieren“ des Textes verwendet werden. Damit diese Zeichen trotzdem einzusetzen sind, werden sie durch andere Zeichenketten umschrieben. So stehen beispielsweise das Zeichen „<“ für „<“ und „>“ für „>“. Wenn ein Element keinen Inhalt besitzt, kann auch eine abgekürzte Form verwendet werden, indem der Slash nach dem Elementnamen folgt und der Tag sowohl Start- als auch Endtag ist (`<ElementName/>`). Enthält ein Element Unterelemente, so werden diese als seine direkten Kinder bezeichnet. Wenn diese Unterelemente ihrerseits Unterelemente enthalten, sind diese seine indirekten Kinder. Das Element selber wird aus Sicht seiner direkten Kinder Elternknoten genannt. Elemente, die denselben Elternknoten besitzen, heißen Geschwister. Wegen seines Aufbaus spricht man bei einem XML-Dokument von einem Baum, dessen Wurzel alle Elemente enthält. Ein Element mit Kindern wird dabei Knoten und ein Element, das keine Kinder besitzt, Blatt genannt.

Ein Element kann beliebig viele Attribute, d.h. Eigenschaften besitzen, sofern keines mehrfach vorkommt. Diese werden nach dem Elementnamen innerhalb der spitzen Klammern im Starttag definiert. Ein Attribut besitzt einen Namen, dem ein Wert zugewiesen ist:

```
<ElementName AttributName="AttributWert">...</ElementName>
```

Innerhalb eines Dokuments können beliebig viele Kommentare eingefügt werden, die dem menschlichen Betrachter ein leichteres Verstehen ermöglichen. Der Text wird innerhalb von „`<!--`“ und „`-->`“ eingeschlossen und von einem XML-Parser, d.h. von einem Programm, das XML-Dateien lesen und verarbeiten kann, ignoriert:

```
<!--Hier steht der Kommentar-->
```

Alle Element- oder Attributnamen sind einem Namensraum [5] zugeordnet. Da ein Dokument mehrere DTDs besitzen kann, helfen sie, die Namen eindeutig zu identifizieren, die dieselben Elementnamen aufweisen. Jede dieser DTDs besitzt einen oder evtl. mehrere solcher Namensräume, in denen sie die verfügbaren Namen definiert. Innerhalb eines Dokuments kann angegeben werden, zu welchem Namensraum ein Name gehört, indem vor dem Namen die Abkürzung eines Namensraumes, gefolgt von einem Doppelpunkt, steht:

```
<nr:ElementName nr2:Attribut="Wert">...</nr:ElementName>
```

Die verwendete Abkürzung muss vor der ersten Verwendung definiert worden sein. Meist werden alle Namensräume im Wurzelelement definiert, so dass sie im kompletten Dokument eingesetzt werden können. Die Definition des Namensraumes erfolgt durch das Hinzufügen eines Attributs zu dem Namen der verwendeten Abkürzung im vordefinierten Namensraum *XML-Namespace* (xmlns). Der Wert ist eine beliebige URI, die den Namensraum beschreibt:

```
<Wurzelelement xmlns:nr="http://Beispiel.de/Nr-Namensraum"
               xmlns="http://Beispiel.de/Standardnamensraum">
```

Ist bei einem Element explizit kein eigener Namensraum angegeben, wird ein Standardnamensraum verwendet. Dieser kann mit Hilfe des Attributs „xmlns“ bestimmt werden.

Die Wahl der Abkürzung ist fast beliebig, da sie durch die angegebene URI ersetzt wird. Es gibt jedoch ein paar vordefinierte Abkürzungen, die nicht als eigene Abkürzung eingesetzt werden dürfen, weil sie, wie z.B. xml oder das schon verwendete xmlns, bereits eine feste Funktion besitzen. Jede der im Folgenden beschriebenen Sprachen besitzt innerhalb ihres Namensraumes bestimmte Schlüsselwörter, denen bestimmte Funktionen zugewiesen sind.

## 2.2 SVG

*Scalable Vector Graphics* (SVG) [8] ist eine mit XML definierte Auszeichnungssprache zur Beschreibung zweidimensionaler Vektorgrafiken, d.h. die damit erzeugte Grafik wird aus einer Menge von Befehlen zusammengesetzt. Diese Befehle generieren aus einer Menge von Koordinaten und Attributen verschiedene Figuren, wie Linien, Rechtecke, Kreise oder Béziér-Kurven [9]. Das hat den Vorteil, dass die Grafik nicht wie in herkömmlichen Grafikformaten (JPEG, GIF) in einer bestimmten Auflösung vorliegt, sondern ohne Qualitätsverlust beliebig zoombar ist. Deswegen eignet sie sich sehr gut für den Einsatz in Straßenkarten. Ein weiterer Vorteil von SVG ist, dass sie weit mehr kann als nur einfache Grafiken anzeigen. Sie bietet die Möglichkeit, komplexe Grafiken mittels Effekten, wie Verläufe, Muster, Filter, Licht und Schatten, zu definieren und sogar Animationen zu erzeugen. Darüber hinaus unterstützt SVG eine Skriptsprache, mit der die Grafiken dynamisch erzeugt bzw. verändert werden können. Ein ebenfalls sehr wichtiges Merkmal von SVG ist, dass dem Benutzer automatisch eine Funktion zum „Zoomen“ und „Scrollen“ zur Verfügung steht, und er sich damit bestimmte Teile detailliert anzeigen lassen kann. Auf einer Straßenkarte eignet sich dies hervorragend, um zu bestimmten Gebieten oder Straßen zu navigieren und diese in voller Auflösung darzustellen.

Ähnlich wie *HTML* [3] bietet auch SVG die Möglichkeit, das Aussehen der Objekte durch die Definition von Stilen zu beeinflussen. Durch die Unterstützung von *Cascading Style Sheets* (CSS) [13] können in SVG die Stile getrennt von den Daten definiert werden, so dass das Aussehen spezieller Objekte an einer Stelle global definiert oder geändert werden kann.

Seit September 2001 ist SVG eine Empfehlung des W3C, und damit wird sie voraussichtlich in zukünftigen Browserversionen automatisch integriert sein. Bisher bietet

lediglich *„Mozilla“* eine rudimentäre SVG-Unterstützung an. Deshalb ist derzeit noch die Verwendung eines Plugins nötig. Da SVG auf XML basiert, liegt sie in einem lesbaren Format vor und kann daher mit einem beliebigen Texteditor erstellt und bearbeitet werden. Die Dateien haben die Endung „.svg“. Es gibt die Möglichkeit, die Daten mit gzip [44] auf durchschnittlich 20% der ursprünglichen Größe zu komprimieren. Solche Dateien besitzen die Endung „.svgz“ und werden im Browser automatisch dekomprimiert. Sowohl die komprimierten als auch die unkomprimierten Dateien können entweder direkt im Browser betrachtet oder mittels eines speziellen Elements (z.B. „embed“, „object“ oder „image“) in ein HTML-Dokument eingebettet werden. Die Einbettung mittels „image“ wird von fast keinem und „object“ nur von wenigen Browsern unterstützt. Bei der am meisten verbreiteten Form „embed“ handelt es sich jedoch um keinen W3C-Standard, so dass je nach Anwendung entschieden werden muss, wie die SVG-Grafiken eingebettet werden sollen. Ein Beispiel dafür ist:

```
<embed src="Map.svgz">
```

Innerhalb eines SVG-Dokuments werden die Befehle in der Regel in der Reihenfolge ihres Vorkommens abgearbeitet, d.h. aus jedem Befehl wird das resultierende grafische Objekt über die bereits bestehenden gezeichnet.

### 2.2.1 Koordinaten in SVG

Im Gegensatz zu vielen anderen Grafikprogrammen liegt hier der Koordinatenursprung links unten, so dass die x-Koordinate nach rechts und die y-Koordinate nach oben angetragen wird. Ist im Dokument das Attribut „viewBox“ nicht gesetzt (s.u.), entspricht der Ursprung dem Wert (x, y) = (0, 0). Als Einheiten gibt es verschiedene Angaben, wie z.B. px (Pixel), cm (Zentimeter), mm (Millimeter) oder Prozentangaben mittels „%“. Wenn eine Koordinate keine Einheit besitzt, wird sie relativ zum aktuellen Benutzerkoordinatensystem gemessen. Durch die Verwendung von bestimmten Transformationsbefehlen können diese Koordinaten jedoch verschoben bzw. verzerrt werden.

### 2.2.2 Aufbau eines SVG-Dokuments

Alle Befehle von SVG liegen im Namensraum „http://www.w3.org/2000/svg“. Das Wurzelelement von SVG hat immer den Elementnamen „svg“. Als mögliche Attribute besitzt es u.a. „viewBox“, „width“, „height“. Während „width“ und „height“ die Breite und Höhe der Grafik angeben, bestimmt „viewBox“ die Koordinaten des sichtbaren Bereiches. Dabei wird die Angabe von vier durch jeweils ein Leerzeichen getrennte Koordinaten erwartet, die der Reihe nach x-Koordinate, y-Koordinate, Breite und Höhe repräsentieren. Mit diesen Einstellungen wird das aktuelle Benutzerkoordinatensystem bestimmt. Standardmäßig füllt SVG den kompletten zur Verfügung stehenden Bereich mit der Grafik. Eine evtl. dabei entstehende Verzerrung des Inhalts kann durch das Attribut „preserveAspectRatio“ vermieden werden, das angibt, wie der Inhalt in diesem Fall dargestellt werden soll. Während der Wert „none“ (Standardwert) eine Verzerrung zulässt,

verhindern alle anderen erlaubten Wert diese, indem sie den sichtbaren Bereich einschränken. Der Wert wird mit folgenden Regeln gebildet: Erst bestimmt der Buchstabe „x“, dass die Ausrichtung der Grafik in x-Richtung anzugeben ist. Darauf folgt die Ausrichtung, wobei „MIN“ linksbündig, „MID“ zentriert und „MAX“ rechtsbündig bedeutet. Nun folgt ein „Y“, wiederum gefolgt von „MIN“(oben), „MID“(Mitte), „MAX“(unten). Obwohl die y-Koordinaten unten beginnen, bedeutet der Wert „MIN“, dass die Grafik oben ausgerichtet werden soll. Wenn nun das Verhältnis der Breite und der Höhe nicht mit dem Verhältnis der „viewBox“ übereinstimmt, wird der Inhalt entsprechend dieser Einstellung angeordnet. „xMIDYMAX“ gibt an, dass der Inhalt mittig am unteren Ausschnitt des SVG-Bereichs auszurichten ist:

```
<svg width="5cm" heigth="5cm" preserveAspectRatio="xMIDYMAX"
     viewBox="0 0 600 800">
```

Innerhalb dieses Elements stehen als Unterelemente die Definitionen der eigentlichen Zeichenbefehle.

### 2.2.3 Standardfiguren

Während komplexere geometrische Figuren aus einzelnen Teilen „zusammengebaut“ werden müssen, gibt es vordefinierte Befehle für einfache Figuren:

**Linie:** Das Element „Line“ mit den Attributen „x1“, „y1“, „x2“, „y2“ zeichnet eine Linie von (x1, y1) nach (x2, y2).

**Rechteck:** Mit „Rect“ wird ein Rechteck gezeichnet. Die Attribute „x“ und „y“ geben die Anfangs-Koordinate an, während „width“ und „height“ die Breite und Höhe bestimmen.

**Kreis:** „Circle“ zeichnet einen Kreis mit dem Ursprung in den Attributwerten „cx“ und „cy“ und dem Radius „r“.

**Ellipse:** „Ellipse“ zeichnet eine Ellipse mit dem Ursprung in den Attributwerten „cx“ und „cy“ und den Radien „rx“, „ry“ in x- bzw. y-Richtung.

**Linienzug:** „Polyline“ zeichnet einen Linienzug. Das Attribut „points“ enthält eine durch Komma getrennte Menge von x- und y-Werten, welche die Ecken darstellen.
`<polyline points="40 10, 50 80, 10 10" />`
zeichnet einen Linienzug, beginnend bei (40,10) über (50,80) nach (10,10).

**geschlossener Linienzug:** „Polygon“ zeichnet einen geschlossenen Linienzug, d.h. es entspricht dem Befehl “Polyline“, wobei der Startpunkt automatisch auch gleichzeitig der Endpunkt ist.

### 2.2.4 Der SVG-Pfad

Mit dem Befehl „path“ bietet SVG ein sehr vielseitiges Konstruktionselement an, mit dessen Hilfe beliebige komplexe Figuren erzeugt werden können. So lassen sich u.a. alle im vorherigen Kapitel beschriebenen Figuren über einen Pfad ausdrücken. Einen Pfad kann man als „Zeichnen mit einem Stift“ betrachten, der von bestimmten Positionen aus durch verschiedene Bewegungen unterschiedlichste Formen zeichnet. Als einziges Attribut wird dafür das „d“ verwendet, dessen Wert die einzelnen Zeichenschritte in der Figur bestimmt. Jeder Schritt beginnt mit einem durch einen Buchstaben abgekürzten Kommando, dem eine bestimmte Menge Koordinaten folgen. Eine Koordinate besteht aus einem x- und einem y-Wert. Die Trennung der Koordinaten bzw. Schritte erfolgt entweder durch ein Leerzeichen oder durch ein Komma. Lediglich die erste Koordinate nach einem Kommando darf unmittelbar an das Kommando selbst anschließen. Die Kommandos können entweder mit einem Großbuchstaben oder mit einem Kleinbuchstaben beginnen. Der Großbuchstabe bedeutet, dass die folgenden Koordinaten absolut zum Koordinatenursprung gemessen werden, während bei einem Kleinbuchstaben die Koordinaten relativ zur aktuellen Stiftposition zu interpretieren sind. Es folgt eine Auflistung aller unterstützten Kommandos. Die Klammer gibt an, welche Art von Koordinaten erwartet wird. Das „+“ bedeutet, dass mehrere dieser Koordinaten hintereinander geschrieben werden können:

**M,m (x y)+** platziert den Stift auf eine bestimmte Koordinate. Bei der Angabe von mehreren Koordinaten wird jeweils eine Linie zur nächstfolgenden gezogen.

**L,l (x y)+** zieht nacheinander eine Linie von der jeweilig aktuellen Position zu der nächsten Koordinate.

**H,h (x)+** zieht eine waagrechte Linie zu der angegebenen x-Koordinate.

**V,v (y)+** zieht eine senkrechte Linie zu der angegebenen y-Koordinate.

**C,c (x1 y1 x2 y2 x y)+** zeichnet eine kubische Béziérkurve nach (x, y) mit den Stützpunkten (x1, y1) und (x2, y2). (Abbildung 2.1)

**S,s (x2 y2 x y)+** zeichnet eine kubische Béziérkurve nach (x, y) mit dem zweiten Stützpunkt (x2, y2). Als erster Stützpunkt wird dabei entweder derjenige aus dem vorherigen Kommando genommen (falls es ein C oder S Kommando war) oder der Startpunkt.

**Q,q (x1 y1 x y)+** zeichnet eine quadratische Béziérkurve nach (x, y) mit dem Stützpunkt (x1, y1).

**T,t (x y)+** zeichnet eine quadratische Béziérkurve nach (x, y). Als Stützpunkt wird dabei entweder der aus dem vorherigen Kommando genommen (falls dies ein Q oder T Kommando war) oder der Startpunkt.

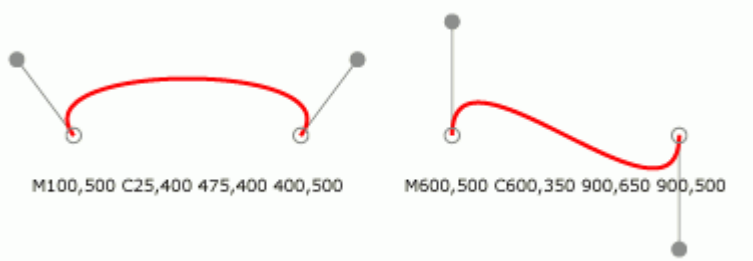



Abbildung 2.1: Beispiele von kubischen Béziérkurven

**A,a (rx ry X-Rotation GroßerBogen RechtsKrümmung x y)+** zeichnet einen elliptischen Bogen nach (x, y). Dabei hat er die Radien „rx“ und „ry“. „X-Rotation“ gibt die Drehung der Ellipse in Grad relativ zur x-Achse an. „GroßerBogen“ bestimmt, welcher Teil der Ellipse gezeichnet wird. Der Wert „1“ zeichnet den Teil der Ellipse, der mindestens 180 Grad hat und „0“ den, der höchstens 180 Grad hat. „RechtsKrümmung“ gibt an, ob die Ellipse nach rechts (1) oder links (0) gekrümmt ist.

**Z,z ()** steht am Ende des Wertes und zieht eine Linie von der aktuellen Position zur Startposition

### 2.2.5 Gestaltung der Figuren mit Hilfe von Stilen

Zusätzlich zur geometrischen Ausprägung kann auch das Aussehen der Figuren bestimmt werden. SVG bietet eine Menge von Eigenschaften an, die zu jedem Zeichenbefehl hinzugefügt werden können. Dabei ist es möglich, in dem Attribut „style“ mehrere dieser Stile zu definieren. Der Wert des Attributs ist eine durch Strichpunkte getrennte Liste einzelner Stile, die aus einem Stilnamen, gefolgt von einem Doppelpunkt und dem Stilwert, bestehen:

```
<circle cx="2" cy="2" r="8"
       style="fill:blue;stroke:#F00;stroke-width:2>
```

Farben können dabei auf verschiedene Weise ausgedrückt werden, entweder als eine vordefinierte Farbe (z.B. „blue“, „red“, ...) oder als ein hexadezimaler Wert, beginnend mit „#“. Eine Angabe im RGB Format ist ebenfalls mit dem Wert rgb(ROT, GRÜN, BLAU) möglich, wobei „ROT“, „GRÜN“ und „BLAU“ Werte zwischen 0 und 255 besitzen. Des Weiteren bietet SVG auch die Möglichkeit, Figuren mit vordefinierten Mustern oder Farbverläufen (Gradienten) zu füllen. Dazu wird die Füllung „fill:url(#[URI])“ eingesetzt. Der Wert URI ist die Referenz auf ein zuvor definiertes Objekt. Wenn keine Farbe verwendet werden soll, kann der Wert „none“ angegeben werden.
Nachfolgend die wichtigsten Stile:

**„fill“** gibt die Farbe an, mit der eine Figur gefüllt werden soll.

**“fill-opacity“** definiert die Deckkraft der Füllfarbe. Sie reicht von vollständig transparent (0) bis undurchsichtig (1.0).

**„stroke“** gibt analog zu „fill“ die Farbe der Linie(n) an.

**„stroke-opacity“** definiert die Deckkraft der Linie(n).

**„stroke-width“** gibt die Dicke der Linie(n) an.

**„display“** gibt an, ob ein Element und seine Unterelemente gezeichnet (z.B. „block“) oder nicht gezeichnet („none“) werden sollen. Damit ist es u.a. möglich, einzelne Objekte, die außerhalb des aktuellen Ausschnitts liegen, auszublenden, da SVG sie ansonsten zeichnen (wenn auch unsichtbar) und dabei viel Rechenleistung unnötig verbrauchen würde.

Während Stile lediglich das Aussehen von Figuren verändern, gibt es Transformationen, welche die geometrischen Beschaffenheiten ändern können. Diese werden im Attribut „transform“ definiert. Mehrere Transformationen können hintereinander ausgeführt werden, indem sie, getrennt durch Leerzeichen, im Wert des Attributs stehen. Dabei spielt die Reihenfolge ihres Vorkommens eine große Rolle. Folgende wichtige Transformationen gibt es:

**translate(X,Y)** verschiebt den Ursprung der Figur um „X“ nach rechts und „Y“ nach oben.

**scale(SX [,SY])** streckt die Figur um den Faktor „SX“ in der Breite und „SY“ in der Höhe. Falls „SY“ nicht angegeben ist, wird dafür derselbe Wert wie für „SX“ verwendet.

**rotate(ANGLE, [CX CY])** „ANGLE“ gibt den Winkel in Grad an, um den die Figur gedreht wird. Falls die Werte „CX“ und „CY“ definiert sind, hat der Drehpunkt die Koordinaten (CX, CY), ansonsten (0,0).

Darüber hinaus besitzt SVG noch eine Vielzahl weiterer Attribute, Transformationen und Stile, die das Aussehen verändern können.

### 2.2.6 Gruppieren und Wiederverwenden von Figuren

Es besteht die Möglichkeit, verschiedene Elemente zu einem großen Objekt zusammenzusetzen. Dazu muss ein (Gruppen-) Element mit dem Namen „g“ erzeugt und alle Teile darin als Unterelemente eingefügt werden. Zusätzlich können weitere Elemente mit dem Namen „svg“ oder „symbol“ zur Gruppierung von Objekten benutzt werden. Dabei kann bei den beiden letzteren durch das Attribut „viewBox“ (s.o.) für die eingebetteten Objekte das Benutzerkoordinatensystem verändert werden. Während die gruppierten Elemente in der Regel an der vordefinierten Stelle auch gezeichnet werden, bietet „symbol“ die Möglichkeit an, Objekte zwar zu definieren, aber nicht zu zeichnen. Jedem Objekt kann ein eindeutiger Identifikationsbezeichner (ID) hinzugefügt werden, der nur einmal

im gesamten Dokument vorkommen darf. Dieser hat den Attributnamen „id“. Mit Hilfe dieser ID kann nun ein einzelnes Objekt oder eine ganze Gruppe im darauffolgenden Dokument wieder verwendet werden. Dies geschieht durch das Element „use“. Der Wert des Attributs „xlink:href“ gibt die ID des zu verwendenden Objekts. Die Attribute „x“, „y“, „width“, „height“ bestimmen die Positionierung bzw. Größe des Objekts.

SVG bietet darüber hinaus eine weitere Möglichkeit, Objekte zu definieren, ohne dass sie gezeichnet werden, dann nämlich, wenn sie innerhalb eines Elements „defs“ liegen.

### 2.2.7 Text

Mit Hilfe des Elements „text“ kann Text in einem SVG-Dokument verwendet werden. Der anzuzeigende Text steht in den Textknoten zwischen Start- und Endtag. Die Attribute „x“ und „y“ geben die Position des Textes an. Der Stil „font-size“ bestimmt die Schriftgröße. Sollen innerhalb eines Textes einzelne Textteile verändert werden, können diese in ein Abschnittselement „tspan“ gelegt werden. Hierin wird zusätzlich zu den normalen Stilen noch die Möglichkeit geboten, den Text mit den Attributen „dx“ und „dy“ relativ zu ihrer ursprünglichen Position zu verschieben.

Herkömmliche Grafikelemente können mit dem Befehl „use“ wiederverwendet werden. Für den Text setzt man den Befehl „tref“ ein. In der Regel wird der Text horizontal von links nach rechts geschrieben. SVG bietet darüber hinaus die Möglichkeit, den Text entlang eines Pfades zu definieren. Dazu muss jedoch ein Pfad („path“) eingerichtet werden, der eine ID besitzt. Anschließend kann dann ein Element „textPath“ diesen über seine ID mit Hilfe des Attributs „xlink:href“ referenzieren, d.h. darauf verweisen. Innerhalb des Elements steht analog zum Element „text“ der anzuzeigende Text.
Abbildung 2.2 kann somit durch folgende Befehle erzeugt werden:

```
<defs>
  <path id="MyPath" d="..." />
</defs>
<text>
  <textPath xlink:href="#MyPath">
    Hier läuft ein Text am Pfad entlang !
  </textPath>
</text>
```

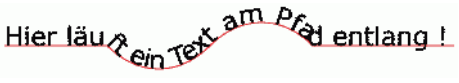



Abbildung 2.2: Verwendung des TextPath-Elements

### 2.2.8 Manipulation des SVG-DOM mit ECMAScript

SVG besitzt mit dem *Document Object Model* (DOM) [14] eine Programmierschnittstelle zur Manipulation der Daten. Innerhalb des DOMs wird die Baumstruktur des

Dokuments verarbeitet, wobei jedes Element einen bestimmten Typ besitzt, der die Art des Elements festlegt. So werden z.B. Attribute eines Elements als Kinder mit dem Typ „Attribut“ dargestellt. Als weitere Typen gibt es u.a. „Textknoten“, „Kommentare“, „Dokument“ (entspricht der Wurzel) sowie „Elemente“, die sonstige Elemente repräsentieren. Mit Hilfe spezieller Funktionen können innerhalb des Dokuments Kinderknoten hinzugefügt, geändert oder gelöscht und dadurch die Grafik verändert werden. So ergeben sich über das Anzeigen bzw. einfache Animieren von Objekten hinaus weitere zahlreiche Anwendungsmöglichkeiten. Zur Manipulation des DOMs wird das auch in HTML eingesetzte *ECMAScript* [14, Anhang E] verwendet. 1995 wurde von der Firma Netscape mit *Javascript* eine Sprache eingeführt, die zur Manipulation des DOMs in Webbrowsern diente. Wegen eines Streit über die Weiterentwicklung dieser Skriptsprache mit der Firma Microsoft, die ihre eigene Variante *JScript* besaß, wurde die European Computer Manufacturers Association (ECMA) beauftragt, diese zu standardisieren. In der Spezifikation „ECMA-262“ wurde dieser Standard unter dem Namen ECMAScript festgeschrieben, das seit Dezember 1999 in der dritten Ausgabe vorliegt. Die meisten Browser unterstützen diese spezifizierten Befehle, doch bieten viele darüber hinaus noch zusätzliche Funktionen an. Die von SVG verwendete Skriptsprache erweitert Javascript und damit auch ECMAScript um einige Funktionen. Da SVG derzeit nicht standardmäßig in den Internetbrowsern vorhanden ist, wird ein Plugin vorausgesetzt, das, ähnlich wie ein Internetbrowser, die Spezifikation unterschiedlich weit unterstützt und darüber hinaus noch eigene Funktionen definiert. In dieser Arbeit ist das am meisten verbreitete SVG-Plugin [10] von Adobe verwendet worden, das zur Zeit in der Version 3.01 vorliegt und bereits für fast alle Betriebssysteme (Windows, MAC, RedHat Linux und Solaris 8) Implementierungen anbietet.

In einem SVG-Dokument können Skriptbefehle innerhalb eines Elements mit dem Namen Skript definiert werden:

```
<script type="text/ecmascript">
    Hier stehen die ECMAScript-Befehle
</script>
```

Das gesamte Dokument wird durch ein Element „SVGDocument“ repräsentiert, das eine Erweiterung des in ECMA definierten „Document“ ist. Neben dem Attribut „root-Element“, welches die Wurzel des Dokuments liefert, besitzt es die ECMAScript Befehle zum Erzeugen neuer Elemente. Die wichtigsten beiden Befehle hierbei sind:
`createElement(tagName)` zum Erzeugen eines neuen Elements (wobei „tagName“ dessen Namen angibt) und `createTextNode(data)`, das einen Textknoten mit dem Text „data“ erstellt. Die Methode `getElementById(id)` liefert, falls vorhanden, das Element mit der eindeutigen Identifikationsnummer „id“ (Attribut mit dem Namen „id“).

Ein Element des Dokuments mit dem Typ „Element“ wird in SVG durch ein „SVG-SVGElement“ oder ein davon abgeleitetes spezifisches Element (z.B. SVGRectElement, SVGPathElement) dargestellt, das eine Erweiterung des ECMA „Element“ und damit eine Erweiterung von „Node“ ist. So bietet es eine Menge von Funktionen zum Einfügen und Entfernen von Kindern an. Die Methode `appendChild(node)` fügt das übergebene Element („node“) als letztes Kind ein, während `insertBefore(newChild, refChild)`

das neue Kind („newChild“) vor einem anderen („refChild“) einfügt. Ein Kind („child“) kann mit `removeChild(child)` aus dem Baum entfernt werden.

Der Befehl `cloneNode(deep)` erzeugt eine Kopie des Elements. Wenn „deep“ den Wert „true“ besitzt, wird ebenfalls eine Kopie all seiner Kinder angelegt, bei dem Wert „false“ wird lediglich das Element selber kopiert.

Auf die Kinder eines Elementes kann mit dem Attribut „childNode“ zugegriffen werden, das eine Liste aller Kinder enthält. Der Zugriff auf seinen Elternknoten kann mit dem Attribut „parentNode“ erfolgen.

Mit dem Befehl `setAttribut(name, value)` wird automatisch ein Attributknoten erzeugt und eingefügt, der dem Namen „name“ den Wert „value“ zuweist. Der Inhalt eines Attributs mit dem Namen „name“ kann mit `getAttribut(name)` zurückgegeben und mit `removeAttribut(name)` gelöscht werden. Auf die Stile eines Elements kann mit Hilfe der Methode `getStyle()` zugegriffen und diese mit `getAttribut, setAttribut, removeAttribut` bearbeitet werden. Innerhalb eines SVGSVGElements ist mit der Methode `getOwnerDocument()` der Zugriff auf das zugehörige SVGDocument möglich.

Mit `getElementsByTagName(name)` liefern sowohl das SVGDocument als auch das SVGSVGElement eine Liste alle Kinderknoten, welche den Namen „name“ besitzen.

### 2.2.9 Events in SVG

Das wichtigste Anwendungskriterium ist die Möglichkeit der Interaktion mit dem Benutzer. Dazu unterstützt SVG sogenannte Events, d.h. benutzergesteuerte Ereignisse. Diese können u.a. auftreten, wenn die Maus über ein Objekt bewegt wird, es verlässt, darauf klickt oder aber beim Wechseln der aktuellen Ansicht. Obwohl Tastatureingaben im SVG-Event Modell nicht standardisiert sind, bieten einige Plugins, wie auch der SVG-Viewer von Adobe, deren Unterstützung an. Neben der Möglichkeit, vordefinierte Animationen zu starten oder zu beenden, können ECMAScript-Methoden aufgerufen werden. Dabei gibt es zwei Arten, solche Ereignisse zu definieren. Jedes SVGSVGElement unterstützt eine Reihe von „Event-Attributen“, die bestimmen, welche Befehle aufgerufen werden sollen, wenn ein Ereignis bei ihnen auftritt. Der Name des Attributs bestimmt die Art des abzufangenden Ereignisses und der Wert die Folge beim Auslösen des Ereignisses.

Die zweite Möglichkeit, Ereignisse zu definieren, ist die Registrierung eines sogenannten „EventListener“ an einem Objekt mit Hilfe eines ECMAScript. Hier wird mit der Funktion `addEventListener(EVENT-NAME, METHODE, false)` für jedes Objekt des Dokuments eine Methode („EventListener“) registriert, die aufgerufen werden soll, wenn das gewünschte Ereignis auftritt. Der „EVENT-NAME“ ist der Name des zu verarbeitenden Events und METHODE ist der Name der Methode, die aufzurufen ist, wenn das Ereignis auftritt. Falls ein Ereignis nicht mehr verarbeitet werden soll, kann die Definition mit der Methode `removeEventListener(EVENT-NAME, METHODE, false)` wieder entfernt werden.

Im Folgenden werden die wichtigsten Ereignisse beschrieben. Die Spalte „EVENT-NAME“ enthält den Namen des Ereignisses, der beim Registrieren mittels der Methode `addEventListener` angegeben werden muss. Die Spalte „Event-Attribut“ gibt den Na-

men des Attributs an, welches das Ereignisse abfangen soll und „Beschreibung“ gibt an, wann ein Ereignis auftritt.

SVGSVGElement und SVGDocument unterstützen:

| **EVENT-NAME** | **Event-Attribut** | **Beschreibung** |
|---|---|---|
| click | onclick | Maus klickt auf ein Element |
| mouseover | onmouseover | Maus bewegt sich auf einem Element |
| mousemove | onmousemove | Maus bewegt sich über ein Element |
| mouseout | onmouseout | Maus verlässt ein Element |

SVGDocument unterstützt:

| **EVENT-NAME** | **Event-Attribut** | **Beschreibung** |
|---|---|---|
| SVGLoad | onload | Dokument wurde vollständig geladen |
| SVGZoom | onzoom | Zoom-Level wurde verändert |
| SVGResize | onresize | Größe des Dokuments wurde verändert |
| SVGScroll | onscroll | Innerhalb des Dokuments wurde gescrollt |

Ein Ereignis tritt immer nur auf dem obersten im Dokument liegenden Element auf, das sich gerade an der aktuellen Position befindet. Im Beispiel 2.3 wird über ein Rechteck ein Kreis und über den Kreis ein Text gezeichnet:

```
<rect   ... onClick="alert('Rechteck')" />
<circle ... onClick="alert('Kreis')"    />
<text   ... > TEXT </text>
```

Abbildung 2.3: Beispiel zur Verwendung von Ereignis-Attributen

Lediglich das Rechteck und der Kreis definieren mit obigen Befehlen das onclick-Event, d.h. sie reagieren auf einen Mausklick und schreiben den Text „Rechteck“ oder „Kreis“. Beim Klicken auf das Rechteck wird die Meldung „Rechteck“ und beim Kreis die Meldung „Kreis“ ausgegeben, da der Kreis über dem Rechteck definiert ist. Obwohl der Text sowohl im Kreis als auch im Rechteck liegt, wird bei einem Mausklick darauf keine Meldung ausgegeben, weil das Event zum Textelement weiter geschickt wird, dieses das Ereignis jedoch nicht weiter verarbeitet und es daher verfällt. Ist ein Ereignis im SVG-Document definiert, dann gilt das für alle im Dokument enthaltenen Objekte, die nicht ihrerseits das Event abfangen. So würde im obigen Beispiel beim Klick auf den Text genau der Befehl aufgerufen werden, der im SVGDocument definiert ist.

Während beim Verarbeiten eines Ereignisses mit EventListenern der Name einer Methode erwartet wird, muss bei der Verwendung von Ereignis-Attributen der Befehl direkt innerhalb des Attributwertes stehen. Dieser kann eine Anweisung der Form `METHODE(evt)` sein, die ihrerseits damit ebenfalls eine Methode aufruft. Beim Auftreten eines Ereignisses wird eine Instanz eines Ereignis-Objekts (SVGEvent) erzeugt, das Informationen über dieses besitzt. Bei „Event-Attributen“ enthält die Variable „evt“ eine Referenz auf das Objekt, während bei EventListenern als erster Parameter immer das Event selbst übergeben wird (z.B. `function clickCircle(evt) { ... }`). Jedes SVGEvent beinhaltet die Methode `getTarget()`, die das Objekt liefert, bei dem das Ereignis auftritt. Ereignisse, die durch die Maus ausgelöst werden, besitzen die DOM-Level-2 typischen Attribute und Methoden. So liefern z.B. die Attribute „clientX“ und „clientY“ jeweils die x- und die y-Koordinate der Mausposition relativ zum Bildschirmbereich. Um daraus die entsprechenden Koordinaten im SVG-Koordinatensystem zu bestimmen, sind umfangreiche Berechnungen nötig, da u.a. die aktuelle Zoomstufe, der aktuell sichtbare Bereich, verschachtelte SVG-Elemente, Transformationen und Verzerrungen mitberücksichtigt werden müssen. In [37] werden diese Berechnungen genauer behandelt und Umrechnungsroutinen zur Verfügung gestellt.

Die Kombination aus der Sprache SVG, dem zugehörigen, veränderbaren DOM und der Möglichkeit, interaktiv auf Benutzereingaben zu reagieren, bietet sich an, um Anwendungen zu schreiben, die zweidimensionale Daten anzeigen, wie z.B. Straßenkarten. Deswegen gibt es bereits eine Menge von Projekten, die sich mit dem Thema der Darstellung geographischer Gegebenheiten in SVG beschäftigen.

## 2.3 Ontologiebeschreibungssprachen

### 2.3.1 RDF

*Resource Description Framework (RDF)* [19] ist ein Standard des W3C, der 1999 verabschiedet wurde. Er bildet die Grundlage zur Verbreitung und zum Austausch von Metadaten, also semantischen Informationen zwischen verschiedenen Beschreibungssprachen. Eines der größten Hindernisse des Datenaustausches ist die große Menge an inkompatiblen Formaten und Definitionen für Metasprachen. Hier setzt RDF an, dessen Ziel eine einheitliche, vom System unabhängige Darstellung von Metadaten ist. Dabei soll es keine bestehenden Sprachen ersetzen, sondern zum Austausch dieser Daten dienen. Sein Hauptanwendungsgebiet ist das Semantic Web, d.h. Informationen sollen zusätzlich zu der für Menschen lesbaren auch in einer für Maschinen verarbeitbaren Form repräsentiert werden, damit Programme darauf operieren können.

Zwischen dem RDF-Modell und deren Serialisierung (RDF-Syntax) sowie der RDF Schema-Spezifikation, welche die Typisierung, Vererbung und Gültigkeit von Ressourcen definiert, ist zu unterscheiden.

### RDF-Modell

Grundlage von RDF stellt die Möglichkeit dar, alle Eigenschaften durch Aussagen der Form „S hat die Eigenschaft P mit dem Wert O“ auszudrücken. Dabei wird S analog zum natürlichen Sprachgebrauch als Subjekt, P als Prädikat und O als Objekt dargestellt. Daraus entstand das RDF-Modell, das drei Arten von Typen besitzt:

**Ressource (Subjekt):** Wie bereits in Abschnitt 2.1.1 erwähnt, ist eine Ressource etwas, das durch eine Identität und damit durch eine URI beschrieben werden kann.

**Eigenschaften (Prädikat):** Eine Eigenschaft ist ein Aspekt, eine Charakteristik, ein Attribut oder eine Beziehung zur spezifischen Beschreibung einer Ressource. Dabei hat jede Eigenschaft eine Bedeutung, die aber kein Bestandteil des RDF Datenmodells sein muss. Es ist jedoch möglich, die erlaubten Werte zu spezifizieren und die zu anderen Objekten erlaubten Beziehungen zu bestimmen. Eine Eigenschaft besitzt eine Identität und könnte daher auch als Ressource dargestellt werden.

**Aussage:** Eine Aussage besteht aus einer Ressource, einer Eigenschaft und einem Wert (Objekt), den die Eigenschaft annimmt. Der Wert kann entweder ein Literal, d.h. ein einfacher Datentyp, wie Text, Zahl oder Datum, oder wiederum eine Ressource sein. Auch Aussagen sind Ressourcen. Damit ist es möglich, Aussagen über Aussagen zu treffen, was als **Reification** bezeichnet wird.

So lassen sich Aussagen wie „eine Münze (S) fällt fast immer (P) auf die Seite mit Kopf oder Zahl(O)“ oder auch „eine Münze (S) besitzt jeweils (P) eine Seite mit Kopf und Zahl (O)“ definieren.

Zusätzlich enthält RDF vordefinierte Datentypen („Container“) für Listen und Mengen, um Gruppen von Objekten darzustellen. Dabei wird unterschieden zwischen einer ungeordneten Liste („Bag“), einer geordneten Liste („Sequence“) und einer Liste aus Alternativen („Alternative“) für den einzigen Wert einer Eigenschaft.

### 2.3.2 RDF-Syntax

Das RDF-Modell ist unabhängig von einer speziellen Darstellungsform. Die dabei am häufigsten verwendete ist die Repräsentation im XML-Format. Hier gibt es zwei mögliche Formen der Syntax, die „serialization Syntax“ und die „abbreviated Syntax“, wobei letztere eine Teilmenge der ersten ist, jedoch eine kompaktere Notation der Aussage erlaubt. Diese Formen der Syntax werden anhand eines Beispieles erklärt.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:wiki="http://de.wikipedia.org/wiki/">
  <rdf:Description rdf:about="http://de.selfhtml.org/">
    <wiki:Autor>Stefan Münz</wiki:Autor>
  </rdf:Description>
</rdf:RDF>
```

Das Beispiel bedeutet, das Stefan Münz der Autor von „http://de.selfhtml.org/“ ist. Das Attribut „about“ enthält immer die URI, die das Objekt definiert. Anstelle des Attributs „about“ kann auch das Attribut „ID“ verwendet werden, mit der Einschränkung, dass der Wert dieses Attributs kein weiteres Mal als „ID“-Attribut definiert werden darf. Mit „ID“ ist also eine neue Ressource innerhalb des Dokuments bestimmt. Die Eigenschaft wird durch ein XML-Element ausgedrückt, das sich in einem von RDF verschiedenen Namensraum befindet. In diesem Beispiel handelt es sich um die Eigenschaft „Autor“. Durch die Definition mehrerer Eigenschafts-Elemente innerhalb eines Objekts kann es mehrere Eigenschaften besitzen:

```
<rdf:Description rdf:about="http://de.selfhtml.org/">
   <wiki:Autor>Stefan Münz</wiki:Autor>
   <wiki:Datum>27.10.2001</wiki:Datum>
   <wiki:Version>8.0</wiki:Version>
 </rdf:Description>
```

Durch das Schreiben der Eigenschafts-Elemente als Attribute erreicht man die erste abgekürzte Form:

```
<rdf:Description rdf:about="http://de.selfhtml.org/"
   wiki:Autor="Stefan Münz"
   wiki:Datum="27.10.2001"
   wiki:Version="8.0"/>
```

Aber es gibt noch weitere: Eine dieser Möglichkeiten ist die Einbettung von Description-Elementen innerhalb eines anderen Description-Elements. Aus obigem Beispiel wird dazu die Aussage folgendermaßen abgeändert: Der Autor von „http://de.selfhtml.org/“ ist der Autor mit der ID 3111. Er hat den Namen „Stefan Münz“ und ist am 10.12.1960 geboren. Diese Aussage wird zuerst in ausführlicher Notation dargestellt:

```
<rdf:Description rdf:about="http://de.selfhtml.org/">
   <wiki:Autor
      rdf:resource="http://de.selfhtml.org/AutorenID/3111" />
</rdf>
<rdf:Description
    rdf:about="http://de.selfhtml.org/AutorenID/3111">
   <wiki:Name>Stefan Münz</wiki:Name>
   <wiki:Geburtsdatum>10.12.1960</wiki:Geburtsdatum>
</rdf>
```

Hier werden zwei verschiedene Ressourcen („`http://de.selfhtml.org/`“ und „`http://de.selfhtml.org/AutorenID/3111`“) definiert. Das Attribut „rdf:resource“ erlaubt es, eine Ressource für den Wert einer Eigenschaft anzugeben, die eigentlich an einer anderen Stelle definiert wird. Durch eine Verschachtelung kann der Zusammenhang zwischen diesen beiden Ressourcen verdeutlicht und der Ausdruck abgekürzt werden:

```
<rdf:Description rdf:about="http://de.selfhtml.org/">
   <wiki:Autor>
     <rdf:Description
        rdf:about="http://de.selfhtml.org/AutorenID/3111">
       <wiki:Name>Stefan Münz</wiki:Name>
       <wiki:Geburtsdatum>10.12.1960</wiki:Geburtsdatum>
     </rdf>
  </wiki:Autor>
</rdf>
```

Da es in RDF möglich ist, mit der Eigenschaft „rdf:type“ den Typ einer Resource anzugeben, kann folgender ungekürzter Ausdruck entstehen:

```
<rdf:Description
    rdf:about="http://de.selfhtml.org/AutorenID/3111">
   <rdf:type resource="http://de.wikipedia.org/wiki/Autor" />
   <wiki:Name>Stefan Münz</wiki:Name>
   <wiki:Geburtsdatum>10.12.1960</wiki:Geburtsdatum>
</rdf>
```

Die URI der Ressource „Autor“ entspricht bis auf die Endung der Deklaration des Namensraumes „wiki“. Sie lässt sich auch schreiben als „wiki:Autor“. Mit dem Ersetzen der „rdf:Description“ durch den Elementtypen erhält man folgende verkürzte Form:

```
<wiki:Autor
    rdf:about="http://de.selfhtml.org/AutorenID/3111">
   <wiki:Name>Stefan Münz</wiki:Autor>
   <wiki:Geburtsdatum>10.12.1960</wiki:Geburtsdatum>
</wiki:Autor>
```

### 2.3.3 RDF-Schema

Ein Nachteil des RDF-Modells ist, dass dessen Regeln den Nutzer kaum einschränken und somit selbst einfache Aussagen ohne Kenntnis von der Bedeutung der verwendeten Eigenschaften und Werte nicht möglich sind. Deswegen wurde das folgende RDF-Schema für die Verwendung von typischen Zusammenhängen standardisierter Begriffe entwickelt, dessen Elemente im Namensraum „http://www.w3.org/2000/01/rdf-schema“ liegen, der in der Regel mit „rdfs“ abgekürzt wird:

**Typisierung:** Durch die Regeln der Typisierung kann man Typen für Ressourcen definieren. So ist es u.a. möglich, Eigenschaften von anderen Ressourcen zu unterscheiden. Dazu werden im RDF-Schema Klassen und Unterklassen bestimmt, wobei Ressourcen dann Instanzen, d.h. Individuen einer solchen Klasse sein können.

**Einschränkungen:** Durch Regeln ist es möglich, die erlaubten Werte einer Eigenschaft einzuschränken. Außerdem kann definiert werden, welche Eigenschaften von welchen Objekten verwendet werden dürfen.

### Typisierung

Ähnlich wie bei einer objektorientierten Sprache gibt es fünf Arten von Klassifizierungen:

**rdfs:Class** ist wie in einer objektorientierten Sprache der Oberbegriff einer Gruppe zusammengehöriger Ressourcen eines Typs (z.B. Autor, Buch).

**rdfs:Resource** gibt Elemente an, die durch RDF-Ausdrücke beschrieben werden können (z.B. die Person Stefan Münz).

**rdfs:Property** repräsentiert eine Ressource, die eine Eigenschaft definiert (z.B. „ist Autor von“).

**rdfs:Statement** stellt eine Aussage dar und wird benötigt, wenn Aussagen über Aussagen getroffen werden sollen (z.B. um auszudrücken, dass die Feststellung „Im Sommer schneit es in Deutschland nicht“ an über 99% der Sommertage zutrifft).

**rdfs:Literal** wird verwendet, wenn eine Klasse keine Ressource als Eigenschaft besitzt, sondern nur eine Menge von Zeichen wie Texte oder Zahlen (z.B. den Namen einer Person).

Da rdfs:Class, rdfs:Property und rdfs:Statement als RDF-Ausdrücke dargestellt werden, sind sie eine Unterklasse von rdfs:Resource. Lediglich rdfs:Literal ist kein RDF-Ausdruck und daher eine selbständige Klasse.

Zusätzlich zu der bereits bekannten Eigenschaft „rdf:type“, die anzeigt, dass eine Ressource eine Instanz einer Klasse ist, gibt es noch die Eigenschaften „rdfs:subClassOf“ und „rdfs:subPropertyOf“, die festlegen, dass eine Klasse (Eigenschaft) Unterklasse (Untereigenschaft) einer anderen Klasse (Eigenschaft) ist. Dabei ist im Gegensatz zu vielen objektorientierten Sprachen erlaubt, dass eine Klasse Unterklasse von mehreren Klassen ist. Sie darf jedoch keine Unterklasse von sich selbst oder einer ihrer eigenen Unterklassen sein, um keine Zyklen zu erhalten (entsprechendes gilt bei den Eigenschaften).
Abbildung 2.4 zeigt den Zusammenhang zwischen den einzelnen Klassifizierungen.

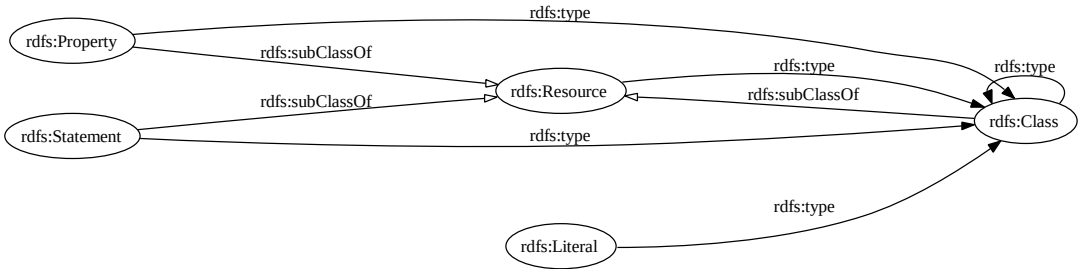



Abbildung 2.4: Klassenhierarchie im RDF-Schema (aus [18])

Die Eigenschaften „Unterklasse“ und „Untereigenschaft“ sind transitiv, d.h. beispielsweise, wenn „PKW“ Unterklasse von „Fahrzeug“ und „Porsche“ Unterklasse von „PKW“ ist, dann ist automatisch auch „Porsche“ Unterklasse von „Fahrzeug“.

#### Einschränkungen

Wie bereits erwähnt, sind Einschränkungen darüber möglich, von welchem Objekt eine Eigenschaft genutzt werden und welche Werte diese Eigenschaft annehmen darf. Dazu gibt es zwei vordefinierte Eigenschaften:

- rdfs:domain definiert eine Klasse von Ressourcen, für welche die jeweilige Eigenschaft verwendet werden darf.
- rdfs:range gibt die Klasse von Werten an, welche die Eigenschaft annehmen kann.

Allerdings dürfen „rdfs:domain“ und „rdfs:range“ nur jeweils einmal innerhalb einer Eigenschaft definiert sein. Es kann also jeweils nur eine einzige Klasse festgelegt werden, für die die Eigenschaft zu verwenden ist bzw. deren Werte die Eigenschaft annehmen darf. Deswegen gibt es Überlegungen, mehrere Einschränkungen desselben Typs zuzulassen. Jedoch erhebt sich dann die Frage, der Interpretation. Es gibt allerdings folgende Alternativen:

- Es ist nur maximal eine Aussage desselben Typs erlaubt.
- Verschiedene Aussagen werden als Alternativen (Disjunktion) betrachtet.
- Verschiedene Aussagen werden, wie üblich, als Schnittmengen (Konjunktion) betrachtet.

Während der erste Ansatz unrealistisch ist, widerspricht die Disjunktion der normalen Verknüpfung von Aussagen mittels Konjunktion. Dennoch kann ausschließlich mit ihr definiert werden, dass eine Eigenschaft Werte aus zwei verschiedenen Klassen besitzt (z.B. rdfs:Literal und rdfs:Resource).

Die Möglichkeit der Kombination verschiedener Ansätze bietet die Sprache OWL, die im folgenden Kapitel behandelt wird. Ausführlichere Information zu RDF befinden sich in [18].

### 2.3.4 OWL

Es ist eine Sprache zur Modellierung von komplexen Sachverhalten nötig, die sowohl vom Menschen als auch vom Computer „verstanden“ werden kann. Dazu sind formale Systeme (sogenannte Ontologien) entwickelt worden, die Regeln zur Beschreibung von Dingen, Konzepten und deren Beziehungen zueinander vorgeben.

Ursprünglich stammt der Begriff Ontologie aus der Philosophie und definiert darin eine Disziplin, die sich mit dem „Sein“, den fundamentalen Typen von Entität (Individuen) und deren Zusammenhängen beschäftigt. Unter der modernen analytischen Ontologie versteht man hingegen eine Lehre von Begriffen, wie Dingen, Eigenschaften oder Ereignissen und von Unterbegriffen, wie Teil, Ganzes oder (un)abhängig, die alle Eigenschaften dieser Entitäten sind. Daraus entwickelte sich die Verwendung des Begriffs „Ontologie“ auch in der Informatik. Eine der bekanntesten Definitionen in der Informatik ist die von Gruber [20] aus dem Jahr 1993:

„Eine Ontologie ist eine explizite Spezifikation einer gemeinsamen Konzeptualisierung (eine abstrakte, vereinfachte Sicht auf die Welt)“.

RDF ist eine gute Basis zur Beschreibung einfacher Ontologien. Jedoch sind damit keine komplizierteren Sachverhalte darzustellen:

- Es ist dort nicht möglich, die Werte einer Eigenschaft in Bezug auf ein bestimmtes Subjekt zu spezifizieren. So kann z.B. die Eigenschaft „ist Kind“ in RDF nicht unterscheiden zwischen Mensch „ist Kind“ von Mensch und Hund „ist Kind“ von Hund.

- Es kann nicht ausgedrückt werden, dass ein Mensch genau zwei Elternteile besitzt.

- Man kann nicht ausdrücken, dass zwei Ressourcen dieselben sind oder sich unterscheiden.

- $x \xrightarrow{E} y,\ y \xrightarrow{E} z \Rightarrow x \xrightarrow{E} z$ (Transitivität)

- $x \xrightarrow{E} y, z \xrightarrow{E} y \Rightarrow x = z$ (Invertierbarkeit)

- $x \xrightarrow{E} y \Rightarrow y \xrightarrow{E} x$ (Symmetrie)

Zur Erschließung des Semantic Web wurde 1999 vom Amerikanischen Verteidigungsministerium die Sprache *DARPA Agent Markup Language Ontology* (DAML-Ont) entwickelt, die auf RDF aufbaute und zur Beschreibung weiterer Zusammenhänge und Regeln Sprachkonstrukte hinzufügte. Fast zeitgleich wurde in Europa mit *Ontology Interference Layer* (OIL) eine andere auf RDF basierende Ontologie-Austauschsprache entwickelt.

Sowohl DAML-Ont als auch OIL hatten dasselbe Ziel und basierten auf denselben Grundlagen. Deswegen beauftrage im Jahr 2000 das W3C die *Web-Ontology Working Group* (WebOnt) mit der Zusammenlegung dieser beiden Sprachen zu DAML+OIL, welche die Stärken aus beiden enthalten, aber deren Schwächen minimieren sollte. Dabei wurde ein neues Vokabular zur Beschreibung von Ressourcen eingeführt:

- In verschiedenen Ontologieteilen können Ressourcen dasselbe ausdrücken. Deswegen ist es nötig, mit Hilfe einer Eigenschaft anzugeben, dass zwei Ressourcen gleich sind.

- Eigenschaften können transitiv, invers oder symmetrisch sein.

- Literale können analog zum XML-Schema [6] Typen, wie z.B. Zahl, Datum oder Text besitzen.

- Einschränkungen auf Subjekte und Werte von Eigenschaften können fast beliebig spezifiziert werden und auch die Kardinalität (Anzahl der Werte) kann festgelegt oder eingeschränkt werden.

Diese neue Sprache DAML+OIL fand häufige Verwendung und viele Programme wurden dafür entwickelt. Im November 2001 begann die WebOnt Group, basierend auf den Erfahrungen mit DAML+OIL, eine neue, von Firmeninteressen unabhängige Sprache zu entwickeln. Diese neue *Web Ontology Language* (OWL) beruht ebenfalls auf dem RDF-Schema, beseitigt jedoch etliche schwere Mängel von DAML+OIL. Je nach Komplexität der Anwendung werden hier drei verschiedene Varianten von OWL definiert:

**OWL Lite** ist eine einfache Variante. Sie beinhaltet eine Klassifizierungshierarchie und einfache Mittel zur Einschränkung.

**OWL DL (Description Logic)** beinhaltet den kompletten OWL-Wortschatz, der allerdings mit ein paar Beschränkungen interpretiert wird. Der Funktionsumfang entspricht in etwa dem von DAML+OIL.

**OWL Full** enthält den gleichen Wortschatz wie OWL DL, doch entfallen hier alle Beschränkungen.

Da in einer Ontologie wichtige Zusammenhänge durch feste Regeln und Vokabeln definiert sind, ohne das zu Grunde liegende Modell genau zu kennen, ist es möglich, Programme zu schreiben, die aus beliebigen Daten effektive logische Schlussfolgerungen ziehen können. Solche Programme heißen *Reasoner* und sind für verschiedene Aufgaben verwendbar:

**Konsistenzprüfung** überprüft, ob die Ontologie in sich konsistent ist oder ob Widersprüche vorliegen.

**Instanzprüfung** kontrolliert, ob ein Sachverhalt oder eine Ressource mit der Ontologie vereinbar ist.

**Abfrage** fragt alle Instanzen ab, die gewisse Eigenschaften erfüllen.

Durch den Einsatz einer Ontologie mit einem zugehörigen Reasoner sind Daten in dieser Form leicht erschließbar und zum Austausch gut geeignet. Für OWL stehen bereits eine Reihe von Reasonern zur Verfügung, wobei Racer [24] der bekannteste ist und die Sprache OWL DL unterstützt.

Der folgende Abschnitt vermittelt einen Überblick über die wichtigsten Sprachelemente von OWL. Die komplette Übersicht über den gesamten Wortschatz befindet sich in [21]. Dabei steht der Namensraum „owl“ für „http://www.w3.org/2002/07/owl“.

### Grundlagen der Modellierung

Während es beim RDF-Schema keine gemeinsame Oberklasse aller Objekte gibt, ist in OWL der Datentyp **„owl:Thing“** eingeführt, der Oberklasse aller Klassen darstellt. Als Gegenstück dazu ist **„owl:Nothing“** definiert als eine Unterklasse aller Klassen. Er besitzt jedoch keine Instanzen, d.h. konkrete Individuen.

Diese Eigenschaft wird von „rdfs:Class“ nicht erfüllt. Deshalb müssen in OWL Klassen mit dem Datentyp **„owl:Class“** definiert werden, der seinerseits den Typ „rdfs:Class“

besitzt und Unterklasse von „owl:Thing“ ist. Bei den Eigenschaften gibt es in OWL zwei verschiedene Grunddatentypen, die beide Untereigenschaften von „rdfs:Property“ sind:

**owl:ObjectProperty** definiert die Beziehung zwischen zwei Ressourcen.

**owl:DatatypeProperty** definiert eine Eigenschaft einer Ressource, dessen Wert entweder ein RDF-Literal oder ein XML-Schema-Datentyp [6] sein kann (z.B. Zahl, Datum, Text).

Da die OWL-Datentypen Unterklassen der RDF-Datentypen sind, können alle RDF-Eigenschaften zur Entwicklung von Ontologien in OWL verwendet werden. Zusätzlich zu diesen bietet OWL noch eine Menge neuer Eigenschaften zur Modellierung weiterer Zusammenhänge an.

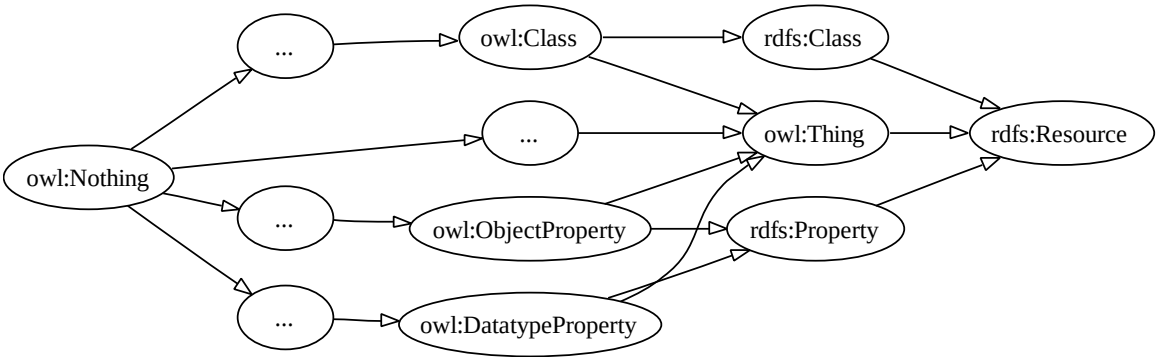



Abbildung 2.5: Ausschnitt aus der Klassenhierarchie im OWL-Schema

### Modellierung identischer und verschiedener Ressourcen

OWL stellt eine Reihe von Eigenschaften zur Verfügung, die ausdrücken, dass zwei oder mehr Ressourcen dasselbe oder aber ein unterschiedliches Subjekt beschreiben. Zu unterscheiden ist dabei jedoch, ob es sich bei der Ressource um eine Klasse, eine Eigenschaft oder eine Instanz handelt.

**owl:equivalentClass** beschreibt die Gleichheit zweier Klassen. Während in OWL Lite und OWL DL das Subjekt und das Objekt einfache Klassen sein müssen, ist es möglich, in OWL Full die Gleichheit komplexer Klassen, wie Aufzählungsklassen oder Klassen von „logischen Verknüpfungen“, zu definieren. Damit kann beispielsweise ausgedrückt werden, dass die Klasse „Auto“ identisch zur Klasse „Automobil“ ist.

**owl:equivalentProperty** gibt an, dass zwei Eigenschaften identisch sind (z.B. „ist Fahrer“ ≡ „fährt“). Dadurch ist automatisch jede der beiden Eigenschaften Untereigenschaft (rdfs:subProperty) der anderen.

**owl:sameAs** wird verwendet, um anzuzeigen, dass zwei Instanzen dasselbe bezeichnen (z.B. „G. Huber“ ≡ „Gerd Huber“).

**owl:differentFrom** bestimmt verschiedene (disjunkte) Instanzen. Nachdem Instanzen auch identisch sein können, muss explizit angegeben werden, dass bestimmte Instanzen für verschiedene Objekte stehen (z.B. für Sommer oder Winter).

**owl:AllDifferent:** Während die Äquivalenz transitiv ist, kann bei der Disjunktion nicht automatisch geschlossen werden, dass gilt: a disjunkt zu b, b disjunkt zu c ⇒ a disjunkt zu c. Deswegen gibt es mit „owl:AllDifferent“ die Möglichkeit, eine Sammlung (Collection) von Elementen zu definieren, die paarweise disjunkt sind:

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Farbe rdf:about="#Rot" />
    <Farbe rdf:about="#Grün" />
    <Farbe rdf:about="#Blau" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

### Typen von Eigenschaften

Während im RDF-Schema mit „rdfs:Property“ nur ein einziger Typ für Eigenschaften vordefiniert ist, bietet OWL eine Reihe von Möglichkeiten zur Definition von Eigenschaften:

**owl:TransitiveProperty** gibt an, dass die Eigenschaft transitiv ist, d.h. es gilt:

$$x \xrightarrow{E} y, y \xrightarrow{E} z \Rightarrow x \xrightarrow{E} z$$

**owl:SymmetricProperty** definiert eine Eigenschaft, die symmetrisch ist:

$$x \xrightarrow{E} y \Rightarrow y \xrightarrow{E} x$$

**owl:FunctionalProperty** gibt an, dass bei jeder Instanz die Eigenschaft maximal einen Wert besitzt:

$$x \xrightarrow{E} y, x \xrightarrow{E} z \Rightarrow y \equiv z$$

**owl:InverseFunctionalProperty** zeigt eine Eigenschaft an, die injektiv ist, d.h.

$$x \xrightarrow{E} z, y \xrightarrow{E} z \Rightarrow x \equiv y.$$

Die Eigenschaft „hat Personalausweisnummer“ ist eine injektive Eigenschaft, denn zwei Personen mit derselben Nummer müssen dieselbe Person sein.

**owl:inverseOf** ist eine Eigenschaft von Eigenschaften, die angibt, dass eine Eigenschaft die Umkehrung der anderen ist, d.h.

$$\text{wenn } E \xrightarrow{invers} F \text{ dann gilt: } x \xrightarrow{E} y \Rightarrow y \xrightarrow{F} x.$$

So sind z.B. die Eigenschaften „fährt“ und „wird gefahren von“ inverse Eigenschaften.

### Einschränkungen von Eigenschaften

Eine weitere wichtige Neuerung von OWL ist die Möglichkeit, die Verwendung von Eigenschaften weitaus effektiver als im RDF-Schema einzugrenzen. Dort kann nämlich lediglich jeweils eine Oberklasse der Subjekte („rdfs:domain“) und Objekte („rdfs:range“) angegeben werden. Jede Eigenschaft darf dabei maximal einmal verwendet werden. OWL hingegen bietet die Möglichkeit, jede Eigenschaft spezifisch einzuschränken und erlaubt sogar, dass eine Instanz mehrmals dieselbe Eigenschaft besitzt.

Dazu hat jede Klasse die Eigenschaft **„owl:Restriction“**. Innerhalb dieses Elements bestimmt der Wert der Eigenschaft **„owl:onProperty“**, welche spezielle Eigenschaft eingeschränkt werden soll. Des Weiteren enthält das Element auch die Art der Einschränkung. Hierbei gibt es folgende Möglichkeiten:

**owl:allValuesFrom** gibt mit dem Wert (vom Typ Klasse) an, dass alle Werte Instanzen dieser Klasse sein müssen. Dabei kann natürlich die Klasse der Instanz auch eine Unterklasse der angegebenen Klasse sein (z.B. hat ein Mensch immer einen Menschen als Elternteil).

**owl:someValuesFrom** bedeutet, dass mindestens ein Wert der Eigenschaft eine Instanz dieser Klasse sein muss. So hat z.B. ein Prominentenkind mindestens einen Elternteil, der prominent ist.

**owl:hasValue** gibt explizit eine Instanz an, die Wert der Eigenschaft sein muss. (Ein Mann z.B. hat das Geschlecht „männlich“.)

**owl:cardinality** gibt an, wie oft die Eigenschaft genau vorkommen muss. In OWL Lite sind hier nur Werte zwischen 0 und 1 möglich, während in OWL DL und OWL Full jeder beliebige positive Wert erlaubt ist. (Ein Mensch hat z.B. genau zwei Elternteile.)

**owl:minCardinality, owl:maxCardinality:** ist ähnlich wie „owl:cardinality“. Hierbei wird jedoch angegeben, wie oft die Eigenschaft mindestens bzw. höchstens vorkommen darf. (Eltern haben mindestens ein Kind; jedes Kind hingegen jeweils maximal zwei lebende Elternteile.)

Zur Kombination von verschiedenen Elementen mit Einschränkungen ist es nötig, eine Schnittmenge zu bilden, da nur die gemeinsamen (eingeschränkten) Eigenschaften verwendet werden dürfen. Für diesen Zweck erzeugt **„owl:intersectionOf“** eine Schnittmenge aus einer Menge von Objekten, Eigenschaften und Einschränkungen.

Oft ist es praktisch, dass man bei einer Einschränkung nicht an eine Oberklasse gebunden ist, sondern dass eine Reihe von verschiedenen Klassen zur Verfügung steht, aus denen eine auszuwählen ist. So kann z.B. der „Freund“ eines Menschen ein Mensch oder ein Haustier sein. Zur Darstellung solcher Sachverhalte wurde mit **„owl:unionOf“** ein Element definiert, das die Vereinigungsmenge aus einer Menge von Elementen bildet. Dies ist jedoch erst ab OWL DL verfügbar.

```
<owl:Class rdf:ID="Weinregal">
  <rdfs:intersectionOf>
    <owl:Class rdf:about="#Regal" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#enthaelt" />
      <owl:allValuesFrom>
        <owl:Class rdf:ID="Wein">
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Rotwein" />
            <owl:Class rdf:about="#Rosewein" />
            <owl:Class rdf:about="#Weisswein" />
          </owl:unionOf>
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:intersectionOf>
</owl:Class>
```

Nachfolgend weitere Elemente von OWL-DL:

**owl:complementOf** drückt aus, dass eine Klasse das Gegenteil einer anderen ist (z.B. Mann und Frau).

**owl:oneOf** ermöglicht analog zu „owl:hasValue“ die Auswahl eines Elements aus einer Menge von Individuen: Eine Jahreszeit ist z.B. entweder Frühling, Sommer, Herbst oder Winter.

**owl:disjointWith** erklärt eine Klasse disjunkt zu anderen, d.h. eine Instanz dieser Klasse kann nie Instanz einer der Klassen sein, die mit dieser Eigenschaft als disjunkt erklärt wurde: Ein Monat ist z.B. keine Jahreszeit.

Des Weiteren können viele Eigenschaften (z.B. „owl:equivalentClass“), die in OWL Lite nur für eine einzelne Klasse definiert sind, in OWL DL auch für Verknüpfungen und Kombinationen aus Klassen verwendet werden. Es ist sogar möglich, solche Eigenschaften für andere Arten von Objekten wie Instanzen oder Eigenschaften zu verwenden. Eine noch größere Auswahl an solchen Kombinationen sind in OWL Full erlaubt.

OWL stellt eine sehr umfassende Sprache zur Beschreibung von Sachverhalten dar. Durch die Menge an Sprachelementen ist es jedoch ziemlich kompliziert, größere Projekte zu definieren und den Überblick über bestehende Definitionen zu behalten. Deswegen wurden besondere Tools entwickelt, mit deren Hilfe Ontologien in einer grafischen Oberfläche dargestellt und bearbeitet werden können. In dieser Arbeit wurde zur Erstellung der Ontologie das Programm Protégé [23] der Stanford Universität verwendet, das über ein OWL-Plugin verfügt und mit dem problemlos auch komplizierte Ontologien erzeugt werden können. Bei der Modellierung des Verkehrsnetzes stellte sich allerdings heraus, dass selbst OWL Lite nicht ausreicht, da Vereinigungen und höhere Kardinalitäten benötigt werden. Deswegen wurde als Grundlage der Ontologie OWL-DL gewählt.

# 3 Modellierung eines Verkehrsnetzes

## 3.1 GDF

Seit Ende der 80er Jahre verlangten aufgrund der enormen Menge an verschiedenen geographischen Datenformaten immer mehr Benutzer und Entwickler nach einem globalen Austauschformat. Das Fehlen eines gemeinsamen Standards war ein großer Nachteil für Firmen, die solche Daten benötigten. Lediglich auf regionaler Ebene gab es bis dahin verschiedene Entwicklungen eines gemeinsamen Formats. Deswegen wandte sich die Industrie an die *International Organisation for Standardization* [25] (ISO) als der führenden Organisation für die Entwicklung von Standards. Die Gründung einer technischen Kommission (ISO TC204) zur Schaffung eines *Intelligenten Transport Systems* [26] (ITS) im Jahre 1993 war dabei der erste Schritt. Anfang des Jahres 1994 begann mit der *Working Group 3* (WG3) eine Gruppierung dieser Kommission mit der Durchsicht aller bestehenden regionalen Standards, wobei erhebliche Unterschiede festgestellt wurden. Während das Japanische Datenformat *Japan Digital Road Map Association* [27] (JDRMA) hauptsächlich für Navigationssysteme entworfen wurde und daher auf Geschwindigkeit und Minimierung der Dateigröße ausgelegt war, diente das Amerikanische Format *Spatial Data Transfer Standard* [28] (SDTS) zur genauen Beschreibung von Datensätzen, jedoch nicht zur Standardisierung des Inhalts. Die Arbeiten in Europa hingegen hatten mit *Geographic Data File* (GDF) 3.0 [29] zu einem anwendungsunabhängigen Datenmodell geführt, das bereits über die Modellierung von Straßenverkehr hinausging und leicht erweiterbar war.

Bei der Durchsicht aller verfügbaren Standards wurde beschlossen, dass das europäische Format das vernünftigste sei und daher wurde dieses in Zusammenarbeit mit sieben Ländern[1] in den Jahren von 1994 bis 2000 überarbeitet und erweitert. Im Mai 2001 wurde es zur Veröffentlichung freigegeben, und seit dem 21.März 2002 ist GDF 4.0 offizieller ISO Standard (ISO 14825) zum Austausch geographischer Daten von Straßennetzwerken.

### 3.1.1 Aufbau des Datenmodells

Alle Objekte in GDF sind in der ISO-Norm lediglich in englischer Sprache beschrieben. Um einen Zusammenhang zum offiziellen Dokument herzustellen, sind im Folgenden alle in der ISO-Norm verwendeten Benennungen in Klammern angegeben (*„ISO-BEZEICHNUNG“*).

In GDF werden alle Instanzen durch Merkmale (*„Features“*) beschrieben und repräsentieren geographische Objekte auf der Welt (z.B. Eiffelturm). Abbildung 3.1 zeigt den Aufbau der Merkmale in GDF. Jedes Merkmal ist genau einer Klasse (*„Feature*

[1] Australien, Deutschland, Japan, Kanada, Korea, Niederlande und USA

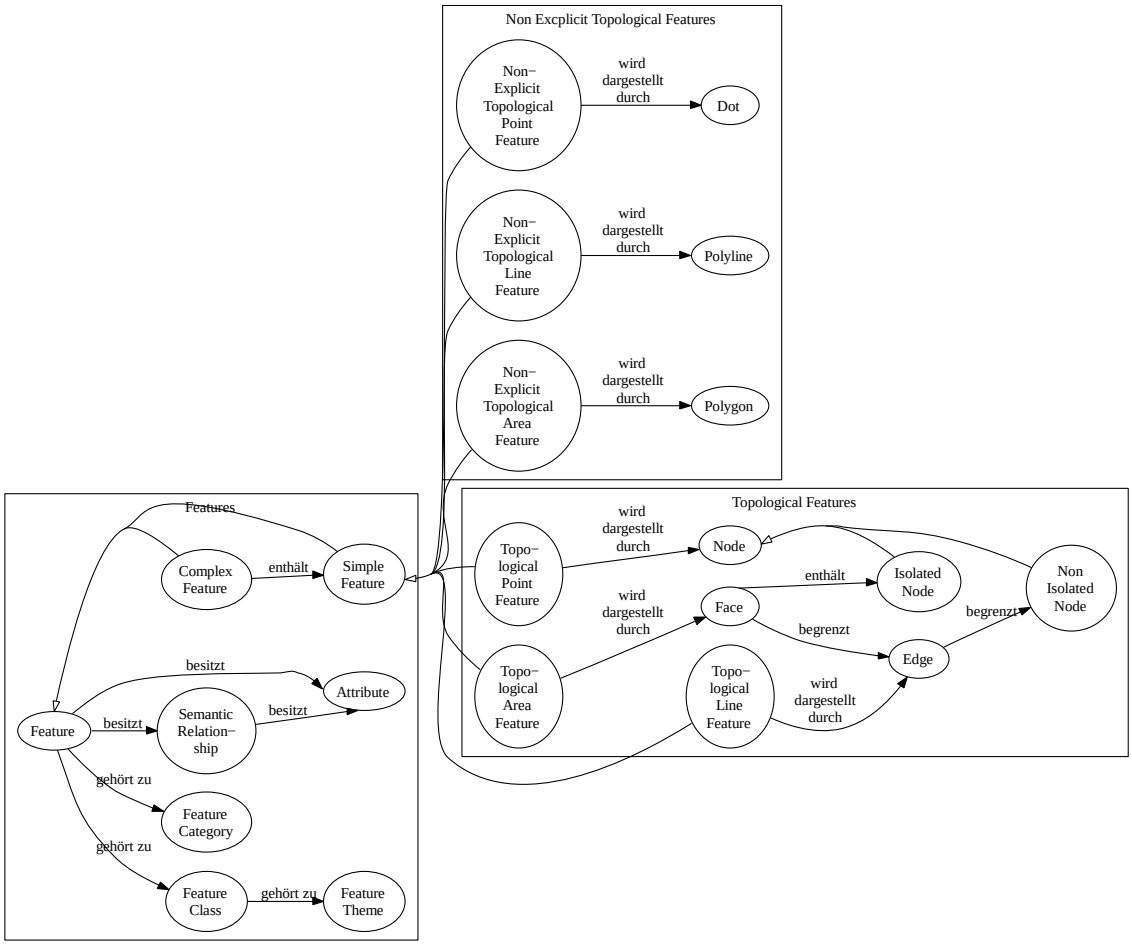



Abbildung 3.1: Aufbau der Merkmale in GDF

*Class*“) zugeordnet (z.B. Gebäude), die ihrerseits wiederum zu einem Thema („*Feature Theme*“) gehört (z.B. Flächenbeschaffenheit). Es kann Attribute („*Attribute*“) sowie Beziehungen („*Semantic Relationship*“) beinhalten. Des Weiteren besitzt es einen Typ („*Feature Category*“), der die geometrische Darstellung des Merkmals bestimmt. Dabei kann es sich entweder um ein einfaches (Punkt, Linienzug oder Fläche) oder um ein zusammengesetztes (komplexes) Merkmal („*Complex Feature*“) handeln.

Die einfachen Typen („*Simple Feature*“) werden noch weiter unterschieden in strukturierte („*Topological Feature*“) und unstrukturierte („*Non-Explicit-Topological Feature*“) Typen. Während unstrukturierte Typen immer geometrisch eigenständig sind, können strukturierte Typen mit anderen fest verbunden sein, d.h. eines der beiden Merkmale ist Teil des anderen. So ist eine Kreuzung, die geometrisch einen Punkt darstellt, fest mit einem in ihr mündenden Straßenabschnitt (Linienzug) verbunden und muss deswegen strukturiert dargestellt werden. Die Umrisse von Gebäuden jedoch haben in der Regel keinen Bezug zu anderen Merkmalen und deswegen werden Gebäude als unstrukturierte Flächen dargestellt.

Jedes einfache Merkmal besitzt eine geometrische Repräsentation, welche die Lage des Merkmals bestimmt. Für unstrukturierte Merkmale gibt es einen Punkt (*„Dot“*), einen Linienzug (*„Polyline“*) und ein Vieleck (*„Polygon“*). Die strukturierten Gegenstücke sind der Knoten (*„Node“*), die Kante (*„Edge“*) und die Fläche (*„Face“*). Bei Knoten gibt es noch zwei weitere Unterscheidungen, nämlich einen Knoten, der an einem Ende einer Kante liegt (*„Non Isolated Node“*) und einen, der in einer Fläche enthalten ist (*„Isolated Node“*).

Ein komplexes Merkmal enthält eine Gruppe von logisch zusammengehörenden strukturierten Merkmalen, die es beschreiben. So wird eine Straße durch eine Menge von Straßenabschnitten definiert.

GDF besitzt drei verschiedene Auflösungslevel, in denen die Merkmale definiert sind. Level-0 enthält alle unstrukturierten, Level-1 alle einfach strukturierten und Level-2 alle komplexen Merkmale.

Mit Hilfe von sogenannten Schichten (*„Layer“*) können zusammengehörende Merkmale auch zusammengefasst werden. Dabei dürfen jedoch innerhalb einer Schicht entweder nur strukturierte oder nur unstrukturierte Merkmale definiert sein. Damit ist es z.B. möglich, alle Straßen zu einer Gruppe zusammenzufassen (z.B. alle Autobahnen).

Ein Merkmal kann Eigenschaften und Beziehungen besitzen, wobei eine Eigenschaft entweder einfach (*„Simple Attribute“*), zusammengesetzt (*„Composite Attribute“*) oder einschränkend (*„Restrictive Sub-Attribute“*) sein kann. Eine zusammengesetzte Eigenschaft besteht aus einer Menge von Untereigenschaften, welche die Eigenschaft näher beschreiben. Solche Untereigenschaften können wiederum einfach, zusammengesetzt oder einschränkend sein. So besitzt z.B. die Eigenschaft „Maut“ die zwei Untereigenschaften „Betrag“ und „Währung“.

Einschränkende Eigenschaften werden nur im Zusammenhang mit Beziehungen oder anderen Eigenschaften verwendet und bestehen ihrerseits aus einer Menge von Untereigenschaften, die jedoch nicht wiederum einschränkend sein dürfen. Mit ihnen kann die Gültigkeit der zu Grunde liegenden zusammengesetzten Eigenschaften bzw. Beziehungen eingegrenzt werden. Die wichtigste einschränkende Eigenschaft ist die Angabe eines Zeitbereichs (*„Validity Period“*), in der diese Eigenschaft gültig ist: Eine Geschwindigkeitsbeschränkung gilt z.B. nur von 22.00 Uhr bis 6.00 Uhr.

Eine Beziehung modelliert eine Eigenschaft die mindestens zwei Merkmalen miteinander verbindet. Sie kann ebenfalls Untereigenschaften besitzen und damit auch eingeschränkt werden. Eine in GDF modellierte Beziehung ist z.B. die Lage eines Gebäudes entlang eines Straßenabschnitts, die angibt, dass ein Gebäude über einen bestimmten Straßenabschnitt erreichbar ist.

### 3.1.2 Themen in GDF

In GDF gibt es 12 Themen, in denen die Merkmale kategorisiert sind (Abbildung 3.2):

**Road and Ferries:** Dies ist das wichtigste Thema des Datenmodells. Es enthält Merkmale zur Modellierung von Straßen und Fährverbindungen.

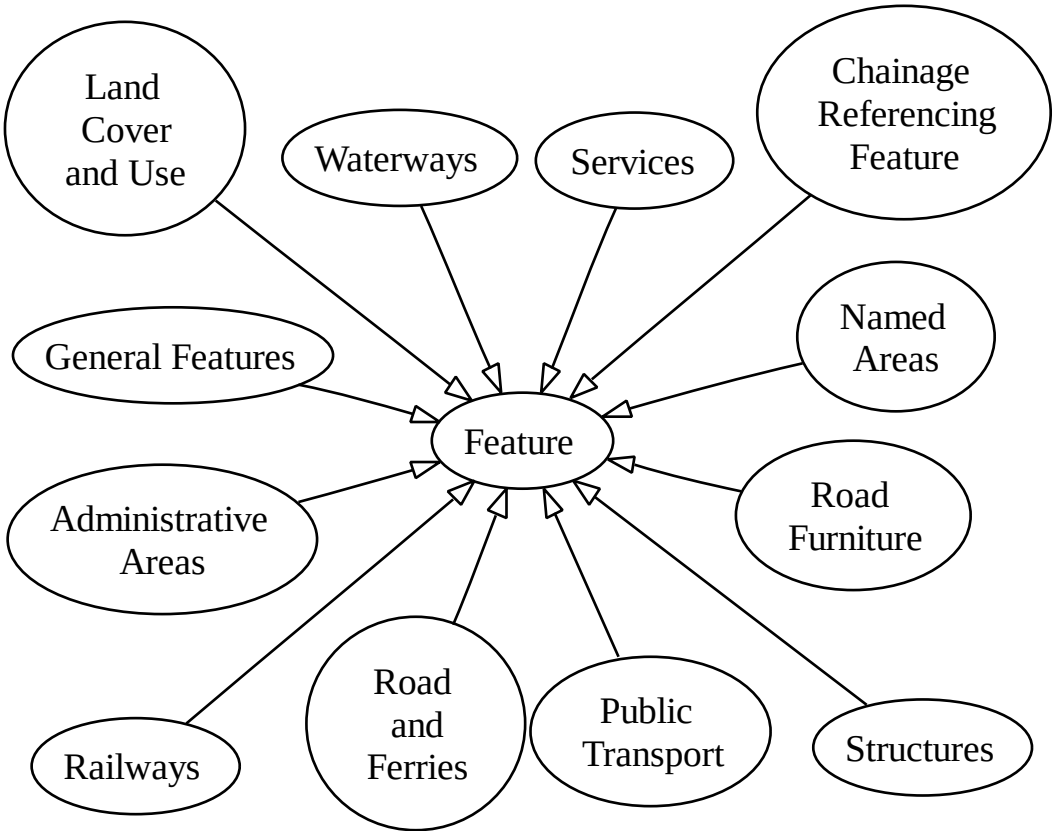



Abbildung 3.2: In GDF definierte Merkmal-Themen

**Structures:** Mit diesem Thema werden Hilfsbauten, wie Brücken oder Tunnel, modelliert.

**Administrative Areas:** Damit erfolgt die Aufteilung eines Landes in Regionen (Bundesländer, Regierungsbezirke, Landkreise, ...).

**Named Areas:** Gebiete, die logisch zusammengehören, können mit diesen Merkmalen modelliert werden, z.B. Plätze oder Schulbezirke.

**Land Cover And Use:** Hiermit wird die Beschaffenheit einer Fläche beschrieben. Es umfasst Merkmale, die von Inseln über Wälder bis hin zu Industriegebieten und Gebäuden reichen.

**Waterway:** Es beschreibt Wasserflächen von Kanälen bis hin zu Meeren.

**Road Furniture:** Sie stellen Merkmale zur Definition von Straßenschildern und Ampeln dar.

**Services:** Hiermit sind Dienstleistungen definiert, z.B Flughafen-Terminal, Apotheke, Schule oder Tankstelle.

**Public Transport:** Dieses Thema dient zur Modellierung des öffentlichen Verkehrsnetzes.

**Railways:** Es beschreibt das Schienennetz.

**Chainage Referencing Features:** Ist ein Thema zur Modellierung allgemeiner Netzwerke. Damit können Graphen mit Knoten und Kanten modelliert werden.

**General Features:** Sie sind übergeordnete Merkmale, die im Zusammenhang mit den meisten anderen Themen verwendet werden können.

### 3.1.3 Allgemeine Eigenschaften

Eine Reihe von Eigenschaften werden durch mehrere verschiedene Merkmale unterstützt. Das nachfolgende Kapitel vermittelt eine Übersicht über den Aufbau und die Funktion der wichtigsten gemeinsamen Eigenschaften:

Viele Merkmale besitzen einen oder mehrere Namen. Dabei wird unterschieden zwischen offiziell gültigen (*„Official Name“*) und ausgewählten Namen (*„Alternate Name“*), die zwar nicht offiziell gültig sind, aber in der Bevölkerung verwendet werden, wie Abkürzungen oder die Bezeichnung des Merkmals in einer anderen Sprache. In München hätte der „Karlsplatz/Stachus“ den offiziellen Namen „Karlsplatz“ und den umgangssprachlichen Namen „Stachus“. Ein Name ist eine komplexe Eigenschaft und besteht meistens aus einer Vorsilbe (*„Prefix“*) und einem Namensrumpf (*„Name Body“*). Die Vorsilbe steht vor dem eigentlichen Namen, der in der Regel den Typ des Merkmals angibt. (In „Hotel Vierjahreszeiten“ ist „Hotel“ die Vorsilbe.) Der Namensrumpf besteht immer aus einer Untereigenschaft, die den Text (*„Alternate Name Text“* oder *„Official Name Text“*) enthält und besitzt optional eine zusätzliche weitere zusammengesetzte Eigenschaft (*„Name Component“*), die den eigentlichen Namen innerhalb des Textes beschreibt. Diese enthält Eigenschaften für Startposition (*„Name Offset“*) und Länge (*„Name Length“*) zur Identifizierung des Namens. Mit *„Name Component Type“* wird die Art des Namens bestimmt (z.B. Text oder Zahl für Autobahnnummern).

Straßennamen werden durch die Eigenschaften *„Official Street Name“* und *„Alternate Street Name“* beschrieben. Diese sind eine Erweiterung der Namen um straßenspezifische Aspekte und bieten Möglichkeiten zur Angabe von Straßenart (z.B. „A6“) und richtungsspezifischen Informationen als Vorsilbe oder Nachsilbe. Darüber hinaus gibt es noch Eigenschaften, welche die Aussprache (*„Pronounciation“*) beschreiben und eine weitere zur Angabe einer Postleitzahl.

Manche Merkmale besitzen eine Referenz in anderen Daten. Dazu speichert man zu einem Merkmal eine ID, über die auf die Daten zuzugreifen ist. Diese wird mit der Eigenschaft *„External Identifier“* angegeben und kann eine beliebige Kombination aus Zahlen und Buchstaben sein. Wenn zu einem Merkmal eine Ressource außerhalb des GDF-Formats existiert, ist es möglich, diese über eine zusammengesetzte Eigenschaft *„Multimedia File Attachment“* zu referenzieren. Damit können z.B. Bilder zu einem Merkmal angegeben werden.

Die Gewichtung der geographischen Darstellung eines Merkmals wird durch *„Display Class“* bestimmt, das eine Unterteilung in zehn Klassen anbietet, wobei die erste Klasse („First Class“) die wichtigsten und die zehnte („Tenth Class“) die unwichtigsten Merkmale enthält.

Öffnungszeiten (*„Opening Period“*) und Zeiten, in denen eine Eigenschaft gültig ist (*„Validation Period“*), werden als zusammengesetzte Eigenschaften beschrieben. Sie beinhalten ein oder mehrere Startzeiten und eine entsprechend große Anzahl an Zeitdauern. Dabei gilt eine Eigenschaft ab der Startzeit für eine gewisse Dauer. Wenn ein Attribut optional eingeschränkt werden kann, wird die Abkürzungsform der Einschränkung in Klammern, beginnend mit einem „+“, angegeben. So kann, wenn neben einer Eigenschaft der Zusatz *„ (+VP)“* steht, ihre Gültigkeit eingeschränkt werden, da die Eigenschaft *„Validation Period“* die Abkürzung *„VP“* besitzt.

Die Eigenschaft *„Positional Accuracy“* legt fest, wie genau die Position eines Merkmals bestimmt ist. Ihr Wert in Metern ist durch eine ganze Zahl angegeben, wobei die Zahl „0“ bedeutet, dass diese Eigenschaft unbekannt ist.

Viele Merkmale werden relativ zu einem Straßenabschnitt angegeben. Dabei bestimmt die Eigenschaft *„Street Side“ (SI)*, auf welcher Straßenseite ein Merkmal liegt. Es wird immer in Straßenrichtung gemessen, d.h. in Richtung der Startkreuzung zur Endkreuzung, weswegen Startkreuzung und Endkreuzung eines Straßenabschnitts nie identisch sein dürfen. Bei einer Ringstraße muss deshalb eine imaginäre Zwischenkreuzung eingefügt und der Straßenabschnitt in zwei Teile geteilt werden. Die möglichen Werte dieser Eigenschaft sind „Left Side“ für die linke Seite und „Right Side“ für die rechte Seite des Straßenabschnitts. Meist wird sie einschränkend verwendet, um anzuzeigen, dass eine Eigenschaft nur für eine bestimmte Straßenseite gilt. Analog zur Einschränkung der Straßenseite existiert noch die Eigenschaft *„Validity Direction (VD)“*, welche die für die Eigenschaft geltende Fahrtrichtung angibt. Hier gibt es die Werte „Valid in positive direction“ und „Valid in negative direction“ für eine Fahrtrichtung in bzw. gegen die Straßenrichtung.

Manche Merkmale kosten für ihre Benutzung eine Mautgebühr (*„Toll“*). Diese ist eine zusammengesetzte Eigenschaft, die sich aus einem Betrag (*„Toll Charge“*) und evtl. aus einer Währung (*„Currency“*) zusammensetzt, aber bisher z.B. kein Verkehrsmittel und keine Personenanzahl berücksichtigt.

### 3.1.4 Das Straßennetz („Road and Ferries“)

Da in GDF das Straßennetz hauptsächlich als Transportweg für Autos angesehen wird, gehören Straßen und Fährverbindungen zu einem gemeinsamen Thema. Es kann auf zwei Detaillierungsstufen betrachtet werden: So beinhaltet die Level-1 Sicht die exakten Verläufe der einfachen Merkmale. U.a. werden darin werden Fahrbahnen, in denen beide Richtungen durch einen Grünstreifen physikalisch voneinander getrennt sind, auch als zwei getrennte Straßenkomponenten betrachtet, die jeweils nur in einer Richtung befahren werden dürfen. Die Level-2 Sicht bietet eine aggregierte Sicht der Level-1 Elemente. Hier werden zusammengehörige Elemente, wie die oben erwähnten getrennten Fahrbahnen, als eine Straße definiert.

Abbildung 3.3 zeigt ein Beispiel der Levelarchitektur in GDF. Die „PARK AVENUE“ besteht aus zwei voneinander getrennten Spuren für jeweils eine Fahrtrichtung. Alle diese Spuren werden in Level-1 als eigene Straßenkomponente dargestellt (L452, L854) und beginnen und enden an jeweils verschiedenen Kreuzungen (S234, S235, S103, S104). In der aggregierten Sicht werden sowohl die Straßenabschnitte zu einer Straße (CL583) als auch die Kreuzungen zu zwei aggregierten Kreuzungen (CS723, CS721) zusammengefasst.

Die im Datenmodell enthaltenen Level-1 Elemente sind *„Address Area“*, *„Address Area Boundary Element“*, *„Enclosed Traffic Area“*, *„Ferry Connection“*, *„Junction“* und *„Road Element“*. Die komplexen Level-2 Elemente sind *„Aggregated Way“*, *„Ferry“*, *„Interchange“*, *„Intersection“*, *„Road“* und *„Roundabout“*.

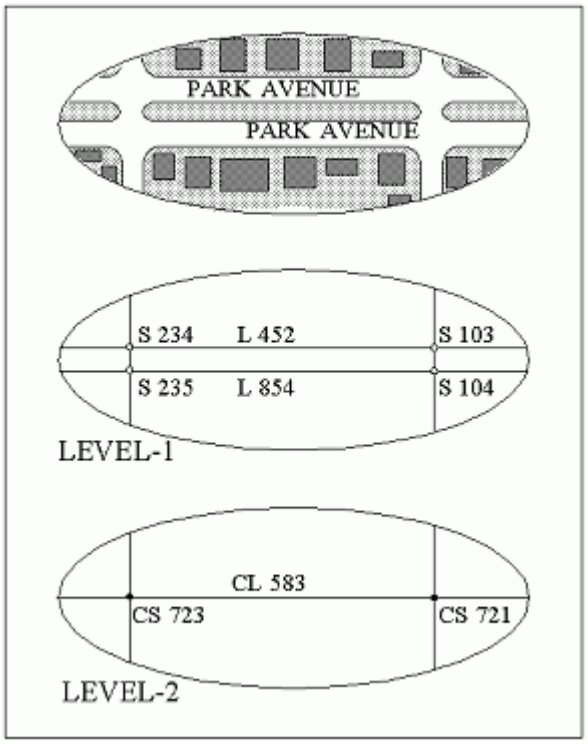



Abbildung 3.3: Beispiel für die Levelarchitektur (Abbildung 5.17 aus [29] )

Eine Straßenkomponente (*„Road Element“*) ist eine Kante und beschreibt zusammenhängende Fahrbahnen zwischen zwei benachbarten Kreuzungen. Sie beginnt und endet an einer Kreuzung (*„Junction“*), wobei Startpunkt und Endpunkt zur Bestimmung der Straßenrichtung verschieden sein müssen. Die aggregierte Sicht der Straßenkompenten ist eine Straße (*„Road“*). Sie enthält eine Menge Straßenkomponenten (in seltenen Fällen auch keine) und beginnt bei einer aggregierten Kreuzung (*„Intersection“*), die ihrerseits die einfachen Kreuzungen enthält, an denen die Straßenkomponenten beginnen oder enden.

Die Darstellung von Straßen- und Fährverbindungen sind im Datenmodell gleich aufgebaut. So entspricht bei Fähren die Fährverbindung (*„Ferry Connection“*) im Wasser bei Straßenfahrzeugen einer Straßenkomponente an Land. Auch sie beginnt und endet

an einer Kreuzung, die in der Regel Anschluss zu einer Straßenkomponente besitzt. Eine aggregierte Sicht auf zusammengehörende Fährverbindungen anolog zur Straße bietet das Merkmal Fähre (*„Ferry“*), das eine Fährstrecke mit Hin- und Rückfahrt beschreibt.

Flächen, auf denen sich Fahrzeuge nahezu beliebig bewegen können (z.B. Parkplätze oder Campingplätze), werden durch Verkehrsflächen (*„Enclosed Traffic Area“*) beschrieben, die über Kreuzungen mit dem Straßennetzwerk verbunden sind.

Meistens wird eine Adresse durch eine Straße und eine Hausnummer bestimmt. Hin und wieder gibt es Plätze oder Bereiche, die einen eigenständigen Namen besitzen und deren Adresse nicht über eine oder mehrere anliegende Straßen identifiziert werden kann. Dazu stellt GDF das Merkmal Adressbereich (*„Address Area“*) zur Verfügung. Diese Fläche wird durch Adresskanten (*„Address Area Boundary Element“*) festgelegt, die jeweils an einer Kreuzung beginnen und enden und somit die Verbindung zum Straßennetzwerk bilden. Ein Adressbereich muss über mindestens eine Straßenkomponente erreichbar sein.

Zusätzlich zur Zusammenfassung von Straßenkomponenten zu Straßen kann eine Menge Komponenten zu einem Kreisverkehr (*„Roundabout“*) oder zu einer großen Kreuzung (*„Intersection“*) zusammengefasst werden. Damit werden u.a. Autobahnkreuze oder Auffahrten modelliert. Andere Straßenkomponenten, die funktional zusammengehören, können zu einem *„Aggregated Way“* zusammengelegt werden.

#### Einschränkende Eigenschaften des Straßennetzes

Merkmale des Straßennetzes besitzen neben den allgemeinen noch eine Reihe weiterer Eigenschaften. Die einschränkende Eigenschaft *„Vehicle Type“* (VT) gibt die Art des Verkehrsteilnehmers an, für welchen die eingeschränkte Eigenschaft gilt. Sie kann eine Reihe von Werten besitzen, z.B. Fußgänger („Pedestrian“), Fahrradfahrer („Bicycle“), PKW („Passenger Car“) oder Taxi („Taxi“). Diese Werte können auch negiert werden, so dass die Eigenschaft für alles außer eben diesem einen Verkehrsteilnehmer gilt (z.B. kein Fußgänger oder kein Radfahrer).

Neben der Einschränkung auf eine Fahrtrichtung mittels *„Validity Direction“* kann auch eine Eigenschaft für eine Fahrspur angegeben werden (z.B. zur Darstellung der Meldung „Die rechte Fahrspur ist wegen eines Unfalls gesperrt“). Dies geschieht durch die Einschränkung *„Lane Dependant Validity“ (LD)*, deren Wert mit einem „R“ oder „L“ beginnt und damit festlegt, ob die folgenden Werte in Straßenrichtung rechts oder links angegeben werden. Anschließend folgt eine Reihe Ziffern, die bestimmt, ob die Eigenschaft für die Fahrspur gilt („1“) oder nicht gilt („0“). Wenn mehr Fahrspuren als angegeben existieren, gilt die Eigenschaft für diese nicht. Auf einer Autobahn mit jeweils 3 Fahrspuren auf jeder Seite gibt der Wert „R0101“ an, dass die Eigenschaft nur für die mittlere Fahrspur auf der rechten Seite und für die Überholspur auf der linken Seite gilt.

#### Straßen und ihre grundlegenden Eigenschaften

Bei Straßenabschnitten und Adressbereichen ist die Angabe einer Menge von Hausnummern möglich, um zu definieren, welche Hausnummern entlang des Merkmals liegen. GDF stellt hierfür die zusammengesetzte Eigenschaft *„House Number Range“* zur

Verfügung. Dabei besitzt jedes Merkmal diese Eigenschaft doppelt, da der Bereich für jede Straßenseite getrennt angegeben wird. Die Straßenseite, für die der Bereich definiert ist, steht in der Eigenschaft *„StreetSide“*. Die Eigenschaften *„First House Number“* und *„Last House Number“* geben die erste und letzte Hausnummer, gezählt in Straßenrichtung an, die innerhalb des Bereichs liegen. Die Art der Nummerierung wird durch *„House Number Structure“* definiert. Der Wert „No house numbers“ gibt an, dass in diesem Bereich keine Hausnummern existieren. Mit „Regular with even numbers“ („Regular with odd numbers“) werden bei der Nummerierung alle (un)geraden Zahlen zwischen Start- und Endwert durchlaufen. Falls die Nummerierung ungeordnet ist, wird „Irregular“ verwendet. Durch die weitere Eigenschaft *„Intermediate House Numbers“*, die mehrere Instanzen besitzen kann, werden explizit die in dem Bereich liegenden Hausnummern angegeben.

*„Length of a Road Element“* legt die theoretische zweidimensionale Länge und *„Measured Length“* die gemessene dreidimensionale Länge des Straßenabschnitts in Metern fest. *„Width“* bestimmt die Breite des Straßenabschnitts in Zentimetern.

Die Art einer Straße gibt die zusammengesetzte Eigenschaft *„Composite Form of Way“* (+VP) mit ihrer Untereigenschaft *„Form of Way“* an. Diese kann u.a. die Werte Autobahn („Part of a Motorway“), Schnellstraße („Part of a Multiple Carriageway which is not a motorway“), Straße mit nur einer Fahrbahn („Part of a Single Carriageway“), Kreisverkehr („Part of a Roundabout Circle“) oder Fußgängerzone („Part of a Pedestrian Zone“) besitzen.

Die auf einem bestimmten Straßenabschnitt erwartete Durchschnittsgeschwindigkeit eines Fahrzeuges in km/h kann mit *„Average Vehicle Speed“* (+VP) (+VT) (+LD) angegeben werden.

Während als Straßenrichtung immer die Richtung von der Start- zur Endkreuzung zu verstehen ist, gibt die Eigenschaft *„Direction of Traffic Flow“* die tatsächliche Richtung an, auf der sich die Fahrzeuge bewegen dürfen. Dabei sind beide Richtungen („Traffic is allowed in both directions“), nur in Straßenrichtung („Traffic is closed in positive direction, and open in negative direction“), gegen die Straßenrichtung („Traffic is closed in negative direction, and open in positive direction“) oder in keine Richtung („Traffic is closed in both directions“) möglich.

#### Einschränkungen der Nutzbarkeit

Mit *„Maximum Height Allowed“* (+VP), *„Maximum Width Allowed“* (+VP) und *„Maximum Length Allowed“* (+VP) wird die maximal zulässige Höhe, Breite und Länge eines Fahrzeugs in Zentimetern angegeben. *„Maximum Total Weight Allowed“* und *„Maximum Total Weight per Axle Allowed“* bestimmen das maximal zulässige Gewicht pro Achse in Zehnteltonnen.

Für die sich im Bau befindliche Straßenabschnitte gibt es die Eigenschaft *„Construction Status“* (+VP) (+LD), welche definiert, ob die Strecke befahrbar („Under Construction - Open“), unbefahrbar („Under Construction - Closed“) oder geplant („Planned“) ist.

Sowohl an Kreuzungen als auch auf Straßenabschnitten können Unfälle passieren. Zur

Modellierung wird die zusammengesetzte Eigenschaft „Accident“ verwendet, die neben dem Zeitpunkt des Unfalls (*„Accident Date“*) mit „Accident Identifier“ eine ID besitzt, welche die Art des Unfalls angibt. Diese ID ist in GDF nicht genauer festgelegt und hat daher eher eine anwendungsspezifische Bedeutung.

#### Erweiterte Eigenschaften von Straßen

*„Ownership“* bestimmt, ob eine Straße öffentlich („Public owned“) oder privat („Privately owned“) ist.

Blockierte Straßen werden durch die zusammengesetzte Eigenschaft *„Blocked Passage“* (+VP) (+LD) beschrieben. Der Ort der Blockade *„Blocked Passage Location“* ist entweder die Startkreuzung („Physically blocked at Start Junction“), die Endkreuzung („Physically blocked at End Junction“) oder aber er liegt innerhalb des Abschnitts („Physically blocked between Start and End Junction“). Ob eine Blockade durchfahren werden kann, gibt *„Blocked Passage Type“* mit „Removable“ für durchfahrbar oder „Permanently fixed“ an. Wenn sie zu durchfahren ist, bestimmt die Eigenschaft *„Removable Blockage“*, wer sie benutzen kann. Dies sind entweder Rettungsfahrzeuge („Accessible for Emergency Vehicles only“), Besitzer einer Art Schlüssel („Accessible via Keyed Access“) oder aber es bestimmt ein Wächter („Guarded“), wer durchfahren darf.

*„Number of Lanes“* (+VP) bzw. *„Minimum number of Lanes“* (+VP) und *„Maximum number of Lanes“* (+VP) geben die (minimale bzw. maximale) Anzahl der Spuren in einer Richtung an.

Der Wert von *„Emergency Vehicle Lane“* (+VP) (+LD) legt fest, ob eine Spur für Rettungswagen vorhanden ist („Present“) oder nicht („Not present“).

Mit *„Passing Restrictions“* (+VP) wird festgelegt, ob ein Überholverbot existiert („Present“) oder nicht („Not Present“).

Die Art der Trennung der einzelnen Fahrtrichtungen wird durch die zusammengesetzte Eigenschaft *„Divider“* (+VP) bestimmt, wobei *„Divided Road Element“* angibt, ob die Fahrtrichtungen getrennt („Divided“) oder nicht getrennt („Not Divided“) sind. Falls sie getrennt sind, dann wird mit *„Divider Type“* die Art und *„Divider Width“* die Breite der Trennung in Metern ausgedrückt. Entweder gibt es eine imaginäre Trennung („Legal Divider“), wie eine Fahrbahnmarkierung, eine überquerbare Trennfläche („Physical divider; Crossable“), wie eine Trambahnspur ohne Bordstein, oder aber, wie auf der Autobahn, eine unüberquerbare Fläche („Physical divider; Not Crossable“).

Im Zusammenhang mit digitalen Radio-Datendiensten, wie TMC oder RDS, ist es erforderlich, eine Position durch eine möglichst kurze ID beschreiben zu können, da dort nur geringe Mengen an Daten übertragen werden. Solche IDs sind bei den Datendiensten bereits fest definiert und können mit der zusammengesetzten Eigenschaft *„Location Reference“* einer Kreuzung oder einem Straßenabschnitt zugeordnet werden. Die Untereigenschaften *„Location Reference Code“* und *„Location Reference Type“* dienen zur Speicherung der ID und der Art des Dienstes. Es gibt dabei u.a. auch die Möglichkeiten „RDS/TMC“ für *Radio Data System* [31] bzw. *Traffic Message Channel* [32] oder „VICS“ für *Vehicle Information and Communication System* [33] zu verwenden.

### 3.1.5 Das öffentliche Verkehrsnetz („Public Transport“ und „Railways“)

In der Modellierung des öffentlichen Verkehrsnetzes besteht eine klare Trennung zwischen den geographischen, fest installierten Gegebenheiten, wie Straßen und Schienen, und den darauf verkehrenden Fahrzeugen. Während die Modellierung von Straßen relativ komplex ist, werden Schienen nur durch zwei Level-1 Merkmale modelliert. Es gibt die Schienenabschnitte (*„Railway Element“*), die analog zu den Straßenkomponenten durch zwei Schienen-Kreuzungen (*„Railway Junction“*) begrenzt sind. Dabei werden jedoch im Gegensatz zu Straßenkreuzungen nur die wirklichen physikalischen Kreuzungen modelliert und keine imaginären, wie einzelne Bahnhöfe, die entlang einer Schienenstrecke liegen.

Das öffentliche Verkehrsnetz wird komplett getrennt von den physikalischen Gegebenheiten modelliert, so dass im Modell kein Zusammenhang zu Straßen oder Schienen existiert. Auch hier sind Abschnitte (*„Route Link“*) durch zwei Kreuzungen des öffentlichen Verkehrs (*„Public Transport Junction“*) begrenzt. Ein Abschnitt kann eine Menge besonderer Punkte (*„Public Transport Point“*) enthalten, mit denen z.B. Messstationen definiert werden. Es gibt jedoch keine Angabe darüber, auf welcher Höhe des Abschnitts dieser Punkt dabei liegt, da diese Punkte lediglich Positionsangaben besitzen. Verschiedene Abschnitte werden zu Strecken (*„Route“*) zusammengesetzt, welche ihrerseits wiederum Bestandteil einer durch einen Namen oder eine Nummer beschrieben Linie (*„Line“*) sind. So besteht z.B. die U-Bahn Linie U5 aus zwei Strecken, die den Verkehr in Hin- und Rückrichtung definieren. Es ist jedoch möglich, dass eine Linie auch mehr als zwei Strecken besitzt, wenn z.B. eine Buslinie zu einem gewissen Zeitpunkt auf einer anderen Route fährt. Eine Strecke enthält eine Menge Haltestellen (*„Stop Point“*), deren Positionen lediglich durch ihre Koordinaten angegeben sind. Damit ist es nicht möglich, ohne Berechnungen eine Haltestelle zu einem Streckenabschnitt oder sogar zu einer genauen Position innerhalb eines Abschnitts zuzuordnen. Haltestellen, die an derselben physikalischen Position liegen, können zu einem Haltestellenbereich (*„Stop Area“*) zusammengefasst werden. Ein Beispiel hierfür sind Busbahnhöfe, bei denen verschiedene Buslinien dieselbe Haltestelle anfahren.

Weitere Informationen und Eigenschaften, wie Umsteigemöglichkeiten oder Fahrpläne, können in GDF nicht modelliert werden. Es ist lediglich möglich, Dienste an denselben Positionen wie den Haltestellen zu definieren, um mit ihnen die Anschlussmöglichkeiten abzubilden. Jedoch existiert zwischen diesen Diensten und dem eigentlichen öffentlichen Verkehrsnetz kein Zusammenhang.

### 3.1.6 Hilfsbauten („Structures“)

Hilfsbauten bezeichnen wichtige Bauten, die einen Teil des Verkehrsnetzes ausmachen. Sie können dabei entweder zu einem Verkehrselement (Lärmschutzwand) gehören oder eine kreuzungsfreie Überführung zwischen zwei Verkehrselementen bilden (Brücke, Unterführung). Je nach Bauwerk werden sie entweder als Punkt, als Linie oder als Fläche dargestellt. In den meisten Fällen ist jedoch die Darstellung als Punkt gewählt. Bei Brücken oder Unterführungen z.B. ergibt sich der Punkt, an denen sich zwei Verkehrs-

elemente geometrisch schneiden. Neben den üblichen Eigenschaften, wie Namen, enthält ein Hilfsbauwerk Angaben zu der Art des Verkehrsflusses über eben dieses Bauwerk (*„Structure Categorie“*). Dabei stehen für Brücken die Werte „over“, für Tunnels „under“ und ansonsten „other“. Die Art des Bauwerks gibt die Eigenschaft *„Structure Type“* an, welche die Werte „Bridge“, „Viaduct“, „Aqueduct“, „Tunnel“, „Cutting“, „Gallery“, „Retaining Wall“ und „Embarkment“ annehmen kann. Mit *„Structure Identifier“* kann eine ID gespeichert werden, die das Bauwerk in einer externen Datenquelle enthält.

### 3.1.7 Landbelegung und -nutzung („Land Cover and Use“)

In diesem Themengebiet liefern Merkmale Informationen über die Beschaffenheit und Verwendung der Erdoberfläche. Dazu ist es, wie Abbildung 3.4 zeigt, in Unterthemen unterteilt, die ihrerseits wiederum weiter unterteilt sein können. In GDF besitzen sie jedoch außer ihrer Unterteilung und Standardeigenschaften keine weiteren Funktionen. Es wird unterschieden zwischen Gebäuden (*„Building“*), künstlichen Geländen (*„Artificial Surface“*), Agrarflächen (*„Agricultural Area“*), Wäldern und halb-natürlichen Flächen (*„Forest and Semi-Natural Area“*), Feuchtgebieten (*„Wetland“*) und Inseln (*„Island“*).

Nur wenige Themenklassen bieten außer den Standardeigenschaften noch weitere beschreibende Eigenschaften an. Bei Grünflächen (*Green Urban Area*) kann mit *Park Type* angegeben werden, wie bedeutend dieser Park ist („City Park“, „Regional Park“, „Country Park“, „State/Provincial Park“ oder „National Park“). Bei Flächen, die Sand enthalten (*„Beach Dune and Sand Plain“*), gibt *„Sand Area Type“* die genaue Art der Fläche mit Strand/Düne („Beach/Dune“), Wüste („Desert“) oder „Other“ an.

Die Art eines Gebäudes (z.B. Turm, Schloss oder Hochhaus) kann mit *„Building Class Name“* angegeben werden. Ein Gebäude kann Funktionen oder Dienstleistungen wie Apotheke, Friseur, Universität, Laden oder Krankenhaus aufweisen. Da jedoch darin auch gleichzeitig mehrere Dienstleistungen angeboten werden können, ist es nicht immer möglich, dem Gebäude eindeutig nur eine zuzuordnen. Deshalb macht die weitere Unterteilung des Themas Gebäude nach Funktionalität keinen Sinn. Zur Verbindung von Merkmalen mit Funktionen wurde in GDF der Begriff „Dienste“ eingeführt, so dass über diesen auch verschiedene Dienstleistungen und Funktionen in Gebäuden modelliert werden können.

### 3.1.8 Dienste („Services“)

Ein Dienst ist ein allgemeiner Ausdruck für eine Dienstleistung, die an einem bestimmten Ort angeboten wird. Dabei ist es wichtig zu beachten, dass ein Dienst die Aktion selbst beschreibt und nicht das Gebäude, in dem sie stattfindet. Parkplätze, Rastplätze oder Werkstätten usw. zählen ebenfalls zu dem Begriff „Dienste“. Diese sind besonders eng verbunden mit dem Straßennetz, da diese entweder über eine Kreuzung oder einen Straßenabschnitt erreicht werden können. Andere Dienste wiederum, wie z.B. Bahnhof oder Kirche, sind unmittelbar mit einem Gebäude verbunden, so dass diese über das zugehörige Gebäude erreichbar sind.

Die Definition von Diensten und deren Klassifizierung ist eher eine Interpretation der

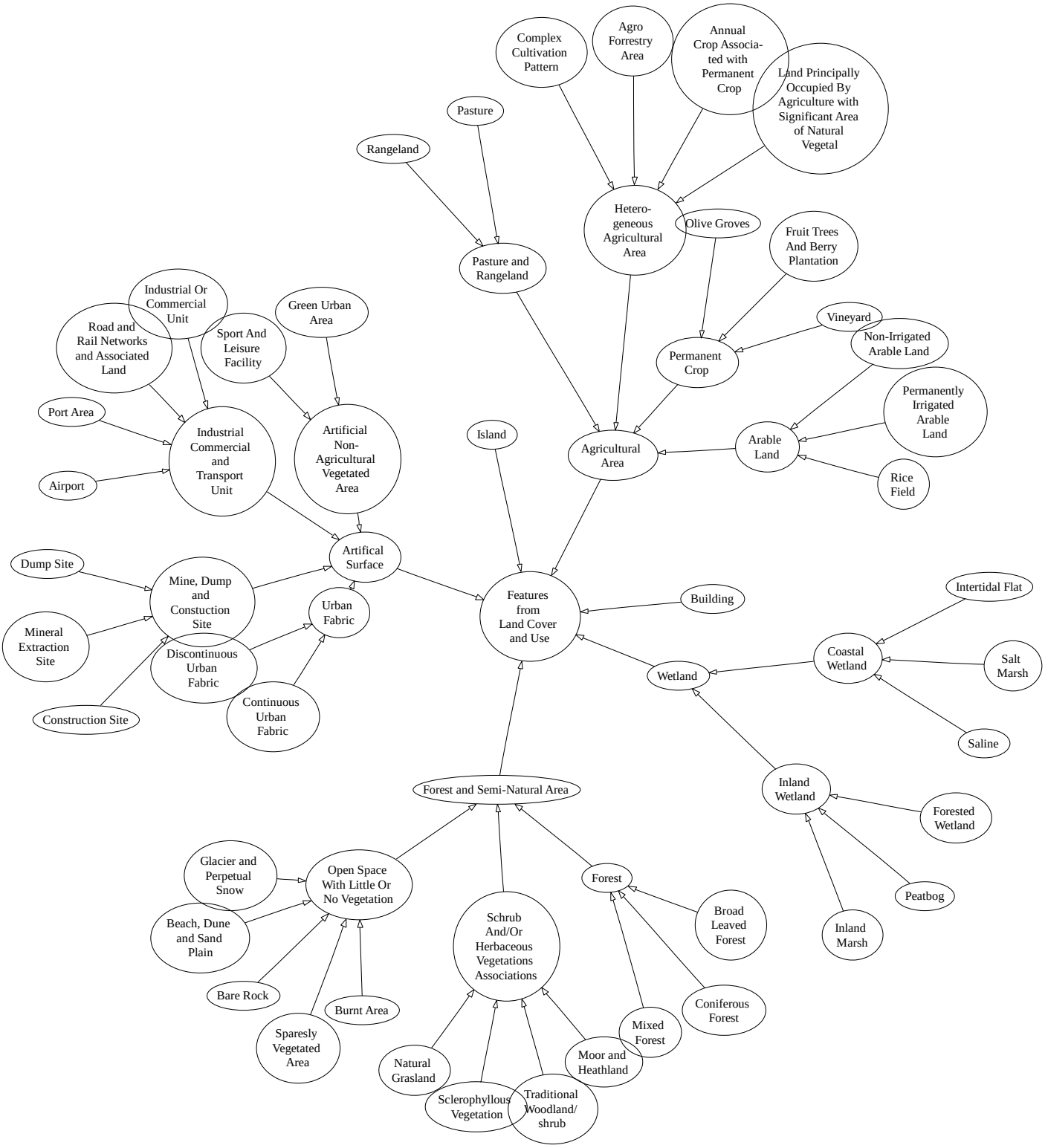



Abbildung 3.4: Landbelegung und -nutzung in GDF

Realität als eine klare abstrakte Darstellung davon und deswegen ist sie kein maßgeblicher Teil in der Spezifikation von GDF. Die einzige Ausnahme ist der Dienst *„Entry Point“*, der den Eingang oder die Verbindung eines oder mehrerer Dienste mit dem Datenmodell darstellt. Ansonsten liefert GDF nur eine allgemeine Menge möglicher Dienste und deren Beschreibung wie Flughafen, Hotel, Restaurant, Bahnhof, Bank, Apotheke, Bowlingzentrum, Campingplatz, Autohändler, Kasino, Kino, Stadthalle, Stadtzentum, Gerichtsgebäude, Kulturzentrum, Einkaufszentrum, ... .

Da das Thema Dienste in der Spezifikation nur angedacht wurde, gibt es kaum zusätzliche Attribute zur Beschreibung der Funktionalität. Auf diese wird daher nicht weiter eingegangen.

### 3.1.9 Zeitangaben

Zur Angabe von Zeiten verfügt GDF über einen großen Sprachwortschatz, der es u.a. ermöglicht, Intervalle oder auch sogenannte „Fuzzy Times“, d.h. allgemeine, nicht durch einfache Regeln bestimmbare Zeiträume zu definieren. Es ist damit möglich, Angaben wie „jeden ersten Werktag im Monat“, „bei Schnee“ oder „von Sonnenaufgang bis Sonnenuntergang“ zu beschreiben.

Generell setzt sich eine Zeitangabe immer aus einem Zeitpunkt *„Starting Date“* und einer Dauer *„Duration“* zusammen, in der eine Eigenschaft gilt. Jede dieser Angaben besteht aus einer in verschiedene Abschnitte unterteilten Zeichenkette, die jeweils einen Teil der Zeitangabe definieren. Jeder Abschnitt beginnt mit einem Symbol, gefolgt von einer Zahl, wobei „y“ für Jahr, „M“ für Monat, „w“ für Woche, „d“ für Tag, „h“ für Stunde, „m“ für Minute und „s“ für Sekunde stehen.

Der Wert „y2004M10d30h12“ bedeutet als Zeitpunkt verwendet, 30.10.2004 um 12.00 Uhr. Der Wert „d1h8m30“ ist der Zeitpunkt des ersten Tages jeden Monats um 8.30 Uhr. Damit können sich zu einem bestimmten Zeitpunkt wiederholende Zeitangaben definiert werden.

Die Angabe „y1s12“ als Zeitdauer bedeutet eine Dauer von einem Jahr und 12 Sekunden.

Zusätzlich zu den Standardsymbolen kann ein Zeitpunkt noch mit einer Reihe anderer Symbole definiert werden. „t“ gibt den Tag innerhalb einer Woche an, wobei „1“ Sonntag und „7“ Samstag bedeutet. Mit „f“ bzw. „l“ kann diese Angabe auch noch mit der Angabe einer Woche im Monat kombiniert werden. Dabei folgt dem Symbol die Angabe der Woche und dann die Angabe des Wochentages. So bedeutet „f21“ der zweite (2) Sonntag (1) im Monat und „l21“ der vorletzte Sonntag im Monat.

Mit „z“ können „Fuzzy-Times“ angegeben werden, wobei z.B. der Wert „z4“ Ferienbeginn und „z6“ Winterbeginn bedeutet.

Eine Angabe von „Fuzzy-Times“ ist auch bei Zeitdauern möglich. Dort besitzen diese jedoch eine andere Bedeutung, z.B. ist „z54“ die Dauer der Ferien und „z55“ die Dauer eines Winters.

Eine Zeitangabe setzt sich aus einem Startpunkt in runden und einer Dauer in geschweiften Klammer zusammen. Dabei können Zeitangaben in GDF auch noch durch die Operationen „*“, „+“ und „-“ miteinander verknüpft werden, wobei „*“ eine logische

„UND“, „+“ eine „ODER“ und „-“ eine „UND NICHT“ - Verknüpfung darstellt.

Dass ein Geschäft während der Woche von 9.00 Uhr bis 17.30 Uhr und samstags von 10.00 Uhr bis 14.00 Uhr geöffnet hat, ist durch folgende Zeichenfolge darzustellen: (h9){h8m30} * (t2){d5} + (t7h10){h4}.

### 3.1.10 Speicherung der Daten

Bei der Speicherung der Daten in GDF wird auf eine möglichst kompakte Form Wert gelegt. Deswegen sind die Daten ähnlich wie in einer relationalen Datenbank aufgebaut, jedoch geschieht die Speicherung in einer einzigen Textdatei in Datensätzen (*„Records“).* Jeder Datensatz hat eine feste Anzahl von Zeichen, die je nach Art des Zeilenumbruchs bei 81 oder 82 liegt. Dabei ist es möglich, verschiedene Datensätze logisch zu gruppieren und einen Datensatz in einer neuen Zeile fortzusetzen, falls die Anzahl der zur Verfügung stehenden Zeichen nicht ausreicht.

Die ersten zwei Zeichen einer Zeile geben die Art des Datensatzes an oder sind Leerzeichen, wenn es sich um die Fortsetzung eines anderen Datensatzes handelt, z.B. „05“ oder „07“ zur Definition von Attributen oder Merkmalen. Dabei wird jedem Attribut bzw. Merkmal eine Nummer zugewiesen, über die es später referenziert werden kann. So erhält das *„Road Element“* immer die „4110“ und die *„Junction“* immer die „4120“. Durch die explizite Definition von Merkmalen und Attributen am Anfang einer Datei können beliebige Merkmale neu definiert bzw. den Bedürfnissen angepasst werden. Eine sehr wichtige Datensatzart ist die „18“, die den Wert eines Attributs angibt. Die expliziten geometrische Strukturen werden ebenfalls mit speziellen Datensätzen dargestellt.

Je nach Art des Datensatzes haben die restlichen Zeichen in einer Zeile unterschiedliche Bedeutung. Bei der Definition von Attributen folgen z.B. zwei Zeichen für die Attributart, zehn für den Attributwert, drei für die Sprache und beliebig viele für eine Beschreibung als Text.

## 3.2 Ontologiebasiertes Verkehrsnetz

GDF ist ein sehr gut durchdachter Standard zur Modellierung von Straßennetzen. Für zusätzliche Aufgaben, z.B. zur Darstellung eines öffentlichen Netzes oder von Diensten sind allerdings einige Erweiterungen nötig. Die Art der Datenspeicherung macht es jedoch fast unmöglich, auf Daten zuzugreifen, ohne eine genaue Kenntnis der Spezifikation zu besitzen, und diese müssen, wie viele andere ISO-Spezifikationen, käuflich erworben werden. Daher wird auf dieses Format verzichtet zugunsten eines anderen, dessen Beschreibung kostenlos zur Verfügung steht. Nur so kann eine nicht kommerzielle Plattform für verschiedene Anwendungsgebiete geschaffen werden.

Aus diesem Grund wurde in der vorliegenden Arbeit die Ontologie von GDF zur Beschreibung eines Verkehrssystemes in das Format OWL überführt, das zudem noch die Vorteile einer leichten Verständlichkeit und Erweiterbarkeit sowohl für den Menschen als auch für ein Computerprogramm bietet. Es entstand das *Ontology based Traffic Network* (OTN).

Zur Entwicklung dieser Ontologie wurde die von der Stanford University entwickelte grafische Oberfläche Protégé [23] verwendet.

### 3.2.1 Grundlegender Aufbau von OTN

GDF stellt in vielen Bereichen eine sehr gute Sprache zur Modellierung des Verkehrssystemes dar. Deshalb baut die Entwicklung der Sprache OTN so weit wie möglich darauf auf. Allerdings erlaubt OWL bei der Benennung der Objekte keine Leerzeichen in Objektnamen. So werden in OTN für die Benennung alle Leerzeichen durch Unterstriche („_“) ersetzt. Wie bei der Objektorientierung üblich, beginnen alle Klassennamen mit einem Großbuchstaben und die Namen aller Eigenschaften mit einem kleinen Anfangsbuchstaben, so dass z.B. *„validity_Period“* den Namen einer Eigenschaft und *„Validity_Period“* den dahinter steckenden zusammengesetzten Datentyp beschreibt. In seltenen Fällen wird zur Adressierung einer zusammengesetzten Eigenschaft ein anderer Name gewählt, wenn dieser eher dem Sinn der Eigenschaft entspricht. So hat z.B. die Eigenschaft *„opening_Period* ebenfalls einen Wert mit dem Datentyp *„Validity_Period“*. In Anlehnung an GDF gibt es fünf verschiedene Arten von Datentypen (Abbildung 3.5):

**Feature** enthält alle Merkmale von OTN.

**Geometric** definiert die geometrischen Formen von Merkmalen.

**Composite_Attributes** sind Klassen, die zusammengesetzte Eigenschaften definieren.

**Relationship** beschreibt die (nicht geometrischen) Zusammenhänge zwischen Merkmalen.

**Transfer_Point** ist ein Punkt, an dem man ein Merkmal über ein anderes ansteuern kann. Dieser ist vor allem für die Erreichbarkeit von Haltestellen und Diensten wichtig.

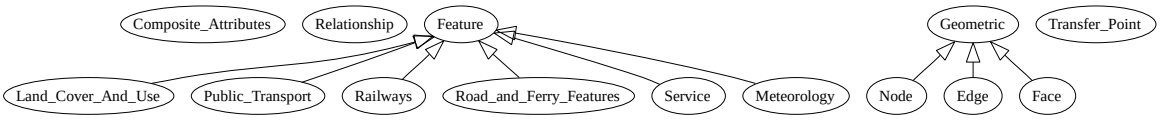



Abbildung 3.5: Aufbau der Daten in OTN

Ein Merkmal in OTN entspricht in etwa einem Merkmal aus GDF. So gehört es zu einer Klasse und zu einem Thema, besitzt sowohl Eigenschaften als auch Beziehungen und hat meistens eine geometrische Repräsentation.

Durch die Möglichkeit der Vererbung wird die Zugehörigkeit zu einer Klasse bzw. zu einem Thema nicht wie in GDF über Verweise gelöst, sondern dadurch, dass eine Klasse Unterklasse einer anderen bzw. des Themas ist. Dabei ist *„Feature“* die Oberklasse aller Merkmale und unterteilt sich in verschiedene Themengebiete. Diese wiederum sind Oberklassen der einzelnen Merkmalklassen, deren Instanzen die einzelnen Merkmale

darstellen. Hierbei wird zwischen einfachen und zusammengesetzten Eigenschaften unterschieden. Während bei einfachen Eigenschaften der Wert der Eigenschaft ein Literal ist und durch den OWL-Datentypen *„owl:DatatypeProperty“* dargestellt wird, verweist eine zusammengesetzte Eigenschaft auf eine weitere Ressource und basiert daher auf dem *„owl:ObjectProperty“* Datentyp. Alle diese zusammengesetzten Eigenschaften werden zur Übersichtlichkeit mit der Oberklasse „Composite_Attributes“ zusammengefasst.

Während in GDF alle Beziehungen durch bestimmte „Beziehungs-Records“ abgebildet werden, sind die meisten Beziehungen hier durch Eigenschaften bestimmt. Es gibt jedoch einige wenige Ausnahmen, für die eine solche Darstellung unpraktisch ist. Deswegen existiert die Oberklasse *„Relationship“*, die solche Beziehungen enthält.

In GDF ist die geometrische Beschaffenheit von Merkmalen durch eine Eigenschaft festgelegt, die auf die zugehörige geometrische Form verweist. Mit der Möglichkeit der Mehrfachvererbung ist dies hier jedoch nicht mehr notwendig. So sind alle Merkmale, die eine geometrische oder geographische Position besitzen, zusätzlich auch noch Unterklassen von *„Geometric“*. In OTN wird jedoch nicht mehr unterschieden zwischen strukturierten und unstrukturierten Geometrien, sondern es gibt nur noch einen strukturierten Datentyp für jede Art von Geometrie.

Auch hier existieren Knoten (*„Node“*), Kanten (*„Edge“*) und Flächen (*„Face“*). Allerdings muss nicht jeder Knoten notwendigerweise Bestandteil einer Kante und eine Kante nicht die Begrenzung einer Fläche sein, sondern die Geometrien können sowohl mit als auch ohne Zusammenhänge definiert werden.

### 3.2.2 Orts- und Verkehrsmittelwechsel an einem „Transfer_Point“

Eine Reihe von Merkmalen in OTN besitzen die Eigenschaft *„is_Accessible_At“*, die angibt, über welches andere Merkmal sie **zu Fuß** erreicht werden können. Dabei ist es u.a. möglich, ein öffentliches Verkehrsmittel, ein Gebäude, einen Dienst oder ein Gelände zu betreten. Diese Zugangsmöglichkeiten sind unter der Oberklasse *„Transfer_Point“* zusammengefasst. Allerdings ist nicht jeder in OTN definierte Zutritt automatisch möglich: Als Autofahrer kann man deshalb nicht direkt ein Gebäude betreten (obwohl dies eigentlich über den Straßenabschnitt erreichbar wäre), weil das Auto erst irgendwo abgestellt werden muss. Wenn man jedoch mit einem Taxi oder zu Fuß unterwegs ist, kann man das Gebäude ohne Parkplatzproblem direkt über den Straßenabschnitt betreten.

In OTN wird nur modelliert, dass ein Merkmal zugänglich ist. Eine Plausibilitätsprüfung muss jedoch an anderer Stelle durchgeführt werden. Damit diese Prüfung nicht zu komplex wird, sind hier nur wenige wichtige Zutrittsmöglichkeiten erlaubt, die in Abbildung 3.6 dargestellt werden.

Ein Zutritt direkt über das Straßennetz ist über Kreuzungen (*„Junction“*, *„Intersection“*) oder Straßen (*„Road“*, *„Road_Element“*, *„Enclosed_Traffic_Area“*) möglich. Auch ein Zutritt über ein Gebäude (*„Building“*) oder eine Haltestelle (*„Stop_Point“*, *„Stop_Area“*) ist erlaubt.

Wenn es sich um einen Wechsel des Verkehrsmittels handelt, wird dies über einen sogenannten „Umstiegs-Dienst“ (*„Transfer_Service“*) modelliert, der in verschiedene Arten unterteilt ist. Zum Abstellen des Autos gibt es die Unterklasse *„Parking“*, die sich

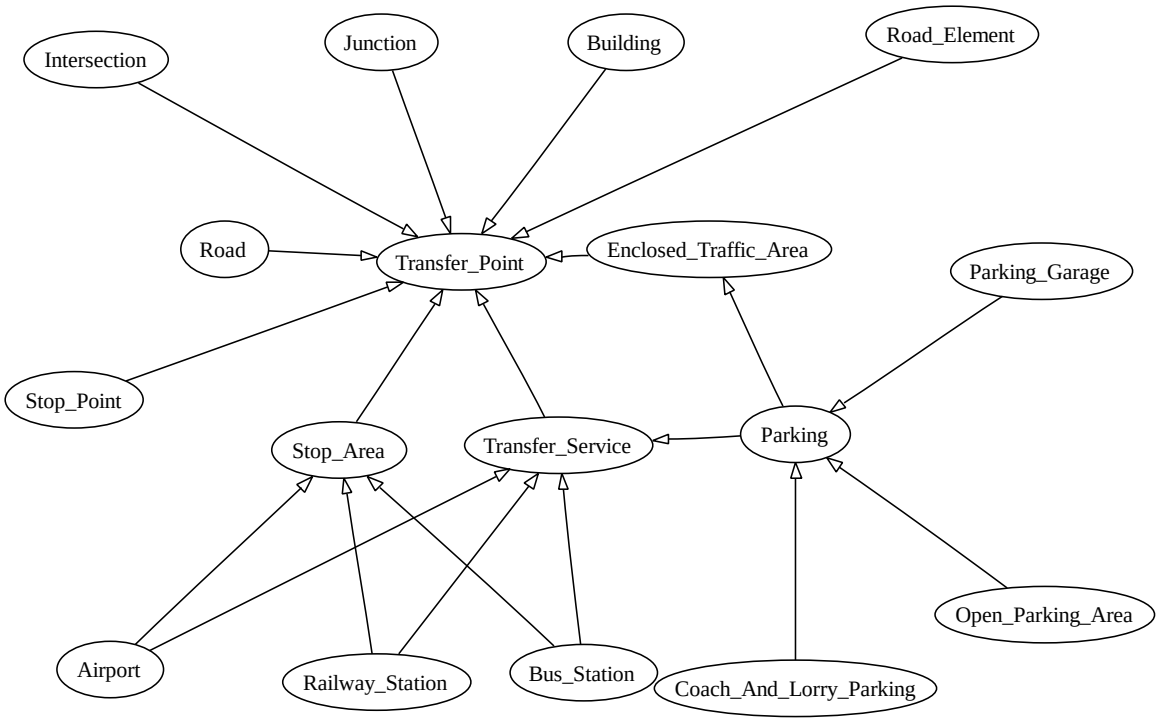



Abbildung 3.6: Verkehrsmittelwechsel an einem *„Transfer_Point“*

in *„Open_Parking_Area“*, *„Parking_Garage“* und *„Coach_And_Lorry_Parking“* unterteilt. Nachdem diese Dienste auch mit dem Auto befahrbare Flächen darstellen, sind sie ebenfalls Unterklassen von *„Enclosed_Traffic_Area“*.

Ein Verkehrsmittelwechsel an einem Bahnhof, Busbahnhof oder Flughafen kann über die Dienste *„Railway_Station“*, *„Airport“* und *„Bus_Station“* erfolgen, die eine Sammlung von Haltestellen darstellen und daher neben *„Transfer_Service“* auch noch *„Stop_Area“* als Oberklasse besitzen.

### 3.2.3 Geometrien

Jeder Knoten besitzt als Eigenschaft eine *„x“*, eine *„y“* und ggf. eine *„z“* Koordinate, welche die genaue Position bestimmen. Eine Kante beginnt (*„starts_at“*) und endet (*„ends_at“*) an einem Knoten. Sie muss jedoch nicht unbedingt geradlinig vom Start- bis zum Endpunkt verlaufen.

Bei einem Straßenabschnitt bietet es sich an, das Stück zwischen zwei Kreuzungen durch eine Kante zu modellieren, die bei der einen Kreuzung beginnt und bei der anderen endet. Jedoch ist ein Straßenabschnitt selten eine gerade Linie, sondern meistens ein Linienzug. Soll der kürzeste Weg zwischen zwei Orten gesucht werden, ist es wichtig zu wissen, von welcher Kreuzung bis zu welcher Kreuzung die Straße verläuft und wie lang sie ist, d.h. graphentheoretisch gesehen, ist ein Straßenabschnitt, der zwischen zwei Knoten liegt, eine Kante mit einer bestimmten Länge. Der exakte Verlauf des Abschnitts wird lediglich zur geographischen Visualisierung benötigt und kann mit der Eigenschaft

*„intermediate_coordinates“* in Form von Koordinaten angegeben werden. Diese Eigenschaft definiert die Koordinaten der Punkte, an denen die Kante eigentlich entlang läuft. Sie ist ähnlich wie der Pfad-Befehl (*„Path“*) von SVG aufgebaut. Jeder Punkt besteht aus einem „x“ und einem „y“ Teil, der, genauso wie die Koordinaten ihrerseits, jeweils durch ein Leerzeichen von anderen getrennt ist. Die x- und y- Werte werden relativ zu dem Vorgängerwert und der erste Wert innerhalb der *„intermediate_coordinates“* relativ zum zugehörigen Wert des Startknotens gemessen.

**Beispiel:**
Die Kante von Punkt A(10,5) nach B(20,30) führt über die Punkte (12,0) und (18,20) und wird daher folgendermaßen dargestellt:

```
<Edge rdf:ID="A_nach_B">
 <starts_at>
  <Node rdf:ID="A">
   <x>10</x>
   <y>5</y>
  </Node>
 </starts_at>
 <ends_at>
  <Node rdf:ID="B">
   <x>20</x>
   <y>30</y>
  </Node>
 </ends_at>
 <intermediate_coordinates>2 -5 6 20 </intermediate_coordinates>
</Edge>
```

### 3.2.4 Allgemeine Eigenschaften von Merkmalen

Alle Merkmale besitzen die Eigenschaft *„alternate_Name“*, um ihnen eine beliebige Menge von Namen zuzuweisen. Einfachheitshalber werden, im Gegensatz zu GDF, die Namen bis auf wenige Ausnahmen mit dieser Eigenschaft definiert. So gibt es außer bei Straßen keine Unterscheidung zwischen offiziellen und inoffiziellen (wechselnden) Namen. Auch die Unterteilung in Vorsilbe und Namensrumpf entfällt.

Wenn ein Merkmal Bestandteil einer anderen Datenquelle ist, können ihm durch die Eigenschaft *„external_Identifier“* eine oder mehrere Identifikationsnummern zugeordnet werden, mit denen es dort referenziert werden kann.

Oft wird ein Merkmal auch im Internet genauer beschrieben. Dazu gibt es die Eigenschaft *„external_Link“*, über die eine Ressource im Web referenziert werden kann.

Die Eigenschaft *„location_Reference“* ist eine zusammengesetzte Eigenschaft zur Identifikation des Merkmals in einem lokationsbezogenen Datendienst. Über die Untereigenschaft *„location_Reference_Type“* wird die Art des Dienstes angegeben. Hierbei sind die Werte „RDS“, „TMC“, „VICS“ und „OTHER“ möglich. Die Identifikationsnummer

ist ein beliebiger Text, der durch die Eigenschaft *„location_Reference_Code“* angegeben wird.

Während in GDF hauptsächlich das Straßennetz genauer definiert werden kann, spielt in OTN die Möglichkeit der Fortbewegung und Kombination verschiedener Arten von Fortbewegungsmitteln eine große Rolle. Somit gibt es neben der bereits schon sehr ausführlichen Betrachtung der Fortbewegung auf einer Straße auch noch Erweiterungen für öffentliche Verkehrsmittel, wie Busse, Bahnen und Flugzeuge.

### 3.2.5 Das Straßennetz

Wie in GDF beinhalten die Merkmale aus dem Thema Straßen- und Fährennetz (*„Road_and_Ferry_Features“*) Objekte zur Beschreibung dieses Verkehrsnetzes. Es unterteilt sich in die aus GDF bekannten Elemente *„Address_Area_Boundary_Element“*, *„Address_Area“*, *„Aggregated_Way“*, *„Enclosed_Traffic_Area“*, *„Interchange“*, *„Intersection“*, *„Junction“*, *„Road“*, *„Road_Element“*, *„Roundabout“*, *„Ferry“* und *„Ferry_Connection“*. Neu ist das Element *„Enclosed_Traffic_Area_Boundary“*, das eine Verkehrsfläche eingrenzt. Abbildung 3.7 beschreibt die Beziehungen zwischen diesen Elementen.

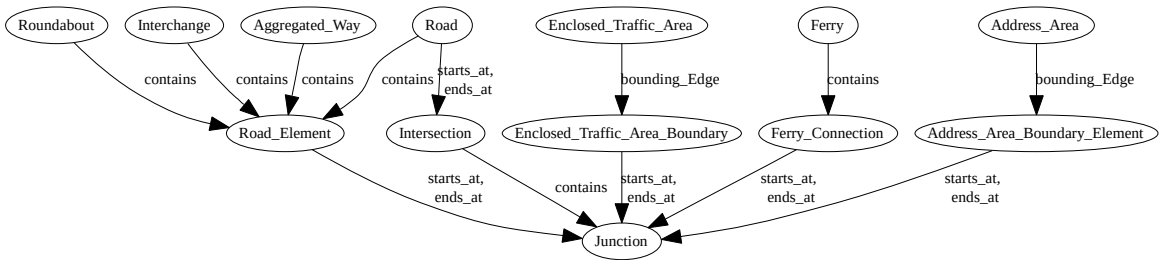



Abbildung 3.7: Beziehungen im Straßen- und Fährennetz in OTN

#### Unreguläre Adressen

Sowohl einem Gebiet *„Address Area“* als auch den Rändern (*„Address_Area_Boundary_Element“*) kann mit der Eigenschaft *„postal_Code“* eine Postleitzahl zugewiesen werden.

Außerdem können in den Rändern Hausnummern mit der zusammengesetzten Eigenschaft *„house_number_Range“* für beide Straßenseiten angegeben werden. Die Straßenseite (*„street_Side“*) gibt mit den Werten („Left“ oder „Right“) an, für welche der Seiten die Festlegung gilt. Mit *„first_House_Number“* und *„last_House_Number“* wird die in Straßenrichtung liegende erste und letzte Hausnummer bestimmt. Die Struktur der Hausnummern (*„house_Number_Structure“*) kann die Werte „No_House_Numbers“, „Regular_with_odd_numbers“, „Regular_with_even_numbers“, „Regular_with_odd_and_even_numbers“ und „Irregular“ annehmen. Falls die Hausnummern unregelmäßig sind, kann der exakte Verlauf der Hausnummern mit der Eigenschaft *„intermediate_House_Numbers“* festgelegt werden. Dabei beginnt die Aufzählung beim ersten Wert nach dem Startwert und endet beim letzten vor dem Endwert. Da es in OWL zwar geordnete Listen gibt, diese aber von Protégé (bisher noch) nicht ausreichend unterstützt sind, werden die Werte, durch ein Komma getrennt, in dieser Eigenschaft gespeichert.

Die Art einer Verkehrsfläche kann durch die Eigenschaft *„enclosed_Traffic_Area_Type“* bestimmt werden. Mögliche Werte sind dabei „Parking_Place“, „Parking_Building“, „Unstructured_Traffic_Square“ und „Another_Type“.

### Fährverbindungen

Die Art einer Fährverbindung (*„Ferry_Connection“*) wird durch *„ferry_Type“* bestimmt. Dabei kann es sich entweder um eine Schiffs- (*„Ship“*), eine *„Hovercraft“*-Verbindung oder um einen Autotransportzug (*„Train“*) handeln. Die Länge der Strecke in Metern wird mit *„length“* angegeben.

Das maximale Gewicht, Breite, Höhe und Länge werden durch die zusammengesetzten Eigenschaften *„maximum_Total_Weight_Allowed“*, *„maximum_Width_Allowed“*, *„maximum_Length_Allowed“* und *„maximum_Height_Allowed“* bestimmt. Diese besitzen neben den Angaben in Zehnteltonnen bzw. Zentimetern noch die Begrenzung der Gültigkeitsdauer mit der einschränkenden zusammengesetzten Eigenschaft *„validity_Period“* (VP). Im Gegensatz zu GDF werden in OTN (vorerst) nur einfache Zeitangaben unterstützt und keine „Fuzzy-Times“. So beginnt eine Gültigkeitsperiode bei einem Zeitpunkt (*„starting_Date“*) und gilt für eine gewisse Zeit (*„time_Duration“*). In OTN wird dabei dieselbe Syntax zur Angabe von Zeitpunkt und Zeitdauer verwendet wie in GDF. Unterschiedlich ist jedoch, dass diese beiden Angaben getrennt gespeichert werden und somit nur noch eine eingeschränkte Verknüpfung von Zeitangaben möglich ist. Lediglich die häufigste Verknüpfung mit „ODER“ wird unterstützt, weil mehrere definierte Gültigkeitsperioden damit verknüpft werden.

### Fahrpläne

Die wichtigste Neuerung von OTN ist die Möglichkeit, Fahrpläne zu definieren, nach denen Verkehrsmittel, wie Fähren, entlang einer Kante verkehren. Während in GDF nur angegeben wurde, von wann und bis wann Verkehrsmittel verkehren, können mit diesen Definitionen die exakten Fahrzeiten und Fahrtdauern bestimmt werden. Das ist für eine genaue Routenplanung wichtig. So besitzt eine Fährverbindung die Eigenschaft *„timetable“*, die eine Reihe von Fahrplänen enthalten kann. Bei jedem Fahrplan muss mit *„validity_Period“* der Zeitbereich angegeben werden, in dem das Verkehrsmittel in regelmäßigen Abständen verkehrt. Dieser Abstand wird durch die Zeitdauer *„loop_Time“* definiert. Da ein Fahrplan vom Startpunkt abhängt, der auf beiden Seiten einer Kante liegen könnte, enthält die Eigenschaft *„starts_at“* eine Referenz auf diesen und muss entweder Start- oder Endpunkt der Kante sein. Die Fahrtdauern (*„travel_Time“*) für die Strecke können variieren; also muss diese beim Fahrplan mit festgelegt werden. Optional kann noch die Aufenthaltsdauer an dem Startpunkt vor dem Start mit der Eigenschaft *„waiting_Time“* angegeben werden. Man kann eine Fähre, die von 6.30 Uhr bis einschließlich 18.30 Uhr stündlich von Punkt A nach Punkt B fährt, dabei 30 Minuten Fahrzeit hat und 20 Minuten vor Abfahrt beladen werden kann, durch folgenden Fahrplan ausdrücken:

```
<Timetable rdf:ID="Timetable_A-B">
  <starts_at rdf:resource="#A"/>
```

```
  <waiting_Time>m20</waiting_Time>
  <loop_Time>h1</loop_Time>
  <travel_Time>m30</travel_Time>
  <validity_Period>
    <Validity_Period rdf:ID='validity_Timetable_A-B'>
      <time_duration>h12</time_duration>
      <starting_Date>h6m30</starting_Date>
    </Validity_Period>
  </validity_Period>
</Timetable>
```

### Kreuzungen

*„Junction“* legt eine normale Straßenkreuzung oder eine besondere Art von Übergang fest, der durch *„junction_Type“* mit den Begriffen „Mini-Roundabout“ (Verkehrsinsel), „Railway_Crossing“ (Bahnübergang) oder „Border_Crossing“ (Grenzübergang) näher bestimmt wird.

Wenn an einer Kreuzung ein Unfall stattgefunden hat, wird dieser durch die zusammengesetzte Eigenschaft *„accident“* beschrieben. Er besteht aus einer eindeutigen Identifikationsnummer (*„accident_Id“*) und einem Unfalldatum (*„accident_Date“*).

### Straßen und ihre grundlegenden Eigenschaften

Eine Straße (*„Road“*) kann neben den Standardeigenschaften eines Merkmals als einzige weitere Eigenschaft einen offiziellen Namen („official_Name“) besitzen.

Dieser kann auch in einem Straßenabschnitt (*„Road_Element“*) angegeben werden. Zusätzlich ist dort ebenfalls die Angabe einer Straßennummer (*„route_Number“*), einer Postleitzahl (*„postal_Code“*) und eines Hausnummernbereiches (*„house_Number_Range“*) möglich.

Die Länge des Abschnitts in Metern und die maximale Breite in Zentimetern wird mit *„length“* bzw. *„width“* angegeben.

Die Art der Straße wird durch *„form_of_Way“* bestimmt und kann folgende Werte annehmen: „Motorway“, „Multiple_Carriageway_which_is_not_a_motorway“, „Single_Carriageway“, „Roundabout_Circle“, „Traffic_Square“, „Enclodes_Traffic_Area“, „Slip_Road“, „Service_Road“, „Entrance_or_exit_to_a_car_park“, „Entrance_or_exit_to_Service“, „Pedestrian_Zone“ oder „Walkway_not_passable_for_Vehicles“. Da ein Straßenabschnitt evtl. zu verschiedenen Zeiten verschiedenartig benutzt werden darf, ist die Einschränkung des Gültigkeitsbereichs mit *„validity_Period“* möglich. So kann z.B. ausgesagt werden, dass ein Straßenabschnitt von 8.00 Uhr bis 20.00 Uhr als Fußgängerzone gilt und in der übrigen Zeit als normale Straße.

Die erwartete Durchschnittsgeschwindigkeit (*„average_Vehicle_Speed“*) für ein Fahrzeug entlang des Abschnitts setzt sich zusammen aus der Geschwindigkeit (*„speed“*) in Kilometern pro Stunde, der Angabe der Spur (LD : *„lane_Dependant_Validity“*), einem Gültigkeitsbereich (VP) und einer Fahrzeugart (VT : *„vehicle_Type“*). Dabei sind, wie in GDF, sowohl Inklusionen wie auch Exklusionen möglich, z.B. „All_Vehicles“, „Pas-

senger_Cars“, „not_Passenger_Cars“, „Pedestrian“, „not_Pedestrian“, ... .

Die erlaubte Fahrtrichtung für ein Fahrzeug entlang des Abschnitts wird durch *„direction_of_Traffic_Flow“* bestimmt und besteht aus der Fahrzeugart (VT), einer Zeitangabe (VP) und der Richtung (*„traffic_flow_Direction“*), welche die Werte „Both_Directions“, „Positive_Direction“, „Negative_Direction“ und „No_Direction“ annehmen kann.

### Einschränkungen der Nutzbarkeit

Analog zu einer Fährverbindung besitzt ein Straßenabschnitt auch Angaben zur maximal zulässigen Breite, Höhe, Länge und Gewicht eines Fahrzeugs durch die Eigenschaften *„maximum_Width_Allowed“*, *„maximum_Height_Allowed“*, *„maximum_Length_Allowed“* und *„maximum_Total_Weight_Allowed“*.

Den Bauzustand einer Straße bestimmt die zusammengesetzte Eigenschaft *„construction_Status“*. Mit *„construction_Stage“* wird der Fortschritt der Arbeiten angezeigt. Dieser reicht von geplant („Planned“), über im Bau („Under_Construction-Closed“) bis hin zu Ausbesserungsarbeiten („Under_Construction-Open“), wobei bei letzterem die Straße befahrbar ist. Mit *„lane_Dependant_Validity“* können, wie in GDF, die Spuren angegeben werden, die sich in Bau befinden und mit *„validity_Period“* der Zeitraum, in dem die Baumaßnahmen stattfinden.

Falls sich ein Unfall auf dem Straßenabschnitt ereignet hat, kann dieser analog zu einer Kreuzung mit der Eigenschaft *„accident“* belegt werden.

### Erweiterte Eigenschaften von Straßen

Mit *„special_Restriction“* kann definiert werden, ob ein Straßenabschnitt während einer bestimmten Zeit (VP) öffentlich zugänglich ist (*„public_Accessible“*).

Eine Blockade auf einem Straßenabschnitt wird beschrieben mit den Angaben der Fahrspur (LD), des Zeitbereichs (VP) und der Art der Einschränkung (*„blocked_Passage_Type“*), die entweder dauerhaft („permanently_fixed“) oder entfernbar(„removable“) ist. Die Position der Blockade (*„blocked_Passage_Location“*) kann sich entweder am Startpunkt („at_Start_Junction“), am Endpunkt („at_End_Junction“) oder innerhalb des Straßenabschnitts („between_Start_and_End_Junction“) befinden.

Handelt es sich um eine Mautstraße, bestimmt *„toll“* die Höhe der Maut, die sich aus einer Währung (*„currency“*), einem Betrag (*„toll_Charge“*) und evtl. einer Gültigkeit (VP) zusammensetzt. Neu in OTN ist die Möglichkeit, die Maut für bestimmte Fahrzeugarten zu definieren (VT).

Die Anzahl der verfügbaren Spuren wird bestimmt durch die Eigenschaften *„minimum_Number_of_Lanes“*, *„maximum_Number_of_Lanes“* und *„number_of_Lanes“*. Diese setzen sich zusammen aus einem Gültigkeitsbereich (VP), der Anzahl der Straßen (*„lanes“*) und einer Straßenseite (*„street_Side“*), die sich entweder in Straßenrichtung links(„Left“) oder rechts („Right“) befindet.

Die maximale Steigung einer Straße in Grad wird mit (*„road_Gradient“*) angegeben und, falls es sich um eine Passstraße handelt, definiert *„height_Of_Pass“* die maximale Höhe in Metern.

Das Vorhandensein einer Rettungswagenspur wird mit *„emergency_Vehicle_Lane“* angezeigt, das aus einem Gültigkeitsbereich (VP), der Angabe, ob eine Rettungsspur existiert (*„has_Emergency_Vehicle_Lane“*) und wenn ja, deren genauen Lage (LD) besteht.

Ein eventueller Stau wird mit *„traffic_Jam_Sensitivity“* festgelegt und besteht aus der Wahrscheinlichkeit (*„traffic_Jam_Probability“*) sowie einer Zeitangabe (VP). Die Wahrscheinlichkeit kann die Werte „No“, „Low“, „Medium“, „High“ und „Very High“ annehmen.

Die Beschaffenheit einer Straße gibt *„road_Surface“* an. Sie setzt sich zusammen aus den Informationen, ob die Straße geteert ist (*„paved“*), aus dem Zustand (*„road_Surface_Condition“*), der entweder gut („Good“) oder schlecht(„Pure“) sein kann und der Art des Belages (*„road_Surface_Type“*), welcher die Werte „Rigid“ (Beton), „Flexible“ (Asphalt), „Blocks“ (Kopfsteinpflaster), „Gravel“ oder „Dirt“ besitzen kann.

Auf welche Weise eine Straßenspur von der entgegenkommenden Spur getrennt ist, gibt *„divider“* an. Die Untereigenschaft „divided“ zeigt dabei auf, ob die Spur getrennt ist. Die Art der Trennung (*„divider_Type“*) kann „Physical_divider;Not_Crossable“, „Physical_divider;Crossable“ und „Legal_Divider“ annehmen und ist *„divider_Width“* Meter breit. Diese Angaben können zeitlich eingeschränkt werden (VP).

### 3.2.6 Das Schienennetz und der öffentliche Verkehr

Wie auch in GDF, besitzt das Schienennetz in OTN nur Informationen über den Ort, an dem die Schienen verlegt sind. So besteht das Thema *„Railways“* einzig aus Schienen (*„Railway_Element“*) und Schienenkreuzungen (*„Railway_Element_Junction“*). Beide Klassen beinhalten keine weiteren Eigenschaften.

Nachdem öffentliche Verkehrsmittel ein wesentlicher Bestandteil eines globalen Verkehrsnetzes sind, wurde das Thema *„Public_Transport“*, im Gegensatz zu GDF, um wichtige Klassen und Eigenschaften erweitert. So ist es möglich, den genauen Weg eines Verkehrsmittels zwischen zwei Haltestellen zu beschreiben.

Auf einer Linie (*„Line“*) fährt in der Regel eine bestimmte Art von Verkehrsmitteln, die mit *„public_Transport_Mode“* beschrieben wird und entweder Bus („Bus“), Straßenbahn („Light_Rail“), U-Bahn („Underground“), S-Bahn („Urban_Rail“), Bahn („Rail“) oder Flugzeug („Airplane“) sein kann. In seltenen Fällen sind auch mehrere Arten angegeben (z.B. für Ersatzbusse). Des Weiteren enthält (*„contains“*) eine Linie eine Menge von Strecken (*„Route“*), die ihrerseits wiederum eine Menge von Fahrstrecken (*„Route_Section“*) enthalten (*„contains“*). Diese bilden im Unterschied zu GDF ein Zwischenstück zwischen Strecken und den in ihnen enthaltenen (*„contains“*) Streckenabschnitten (*„Route_Link“*). Dabei beginnen sie (*„starts_at“*) bei einer Haltestelle (*„Stop_Point“*) und enden (*„ends_at“*) an einer zweiten. Die Richtung ist hier nicht zwingend vorgeschrieben. Haltestellen verschiedener Linien, die geographisch sehr nah beieinander liegen (z.B. in einem Busbahnhof), können, wie in GDF, zu einer *„Stop_Area“* zusammengefasst werden, welche die einzelnen Haltestellen enthält (*„contains“*).

Eine Haltestelle ist eine Unterklasse einer Kreuzung (*„Public_Transport_Junction“*) und kann ebenfalls einen Streckenabschnitt begrenzen. Damit bilden die in einer Fahrstrecke enthaltenen Streckenabschnitte einen Weg von der ersten Haltestelle zu einer

zweiten, den das Verkehrsmittel anfährt. So ist, im Gegensatz zu GDF, die exakte Position einer jeden Haltestelle innerhalb des Verkehrsnetzes definiert, ebenso wie der genaue Weg zu der nächstgelegenen. Durch die weitere Unterteilung von Fahrabschnitten in Streckenabschnitte können gemeinsame Teilwege verschiedener Linien abgebildet werden.

Abbildung 3.8: Beispiel eines öffentlichen Verkehrsnetzes

Da eine Fahrstrecke eine Menge von Streckenabschnitten besitzt, die an der ersten Haltestelle beginnen und an der anderen enden, müsste eine Fahrstrecke keine Informationen über Start- und Endhaltestelle enthalten. Jedoch sind meistens die Unterteilungen in Streckenabschnitte und Kreuzungen uninteressant (z.B. bei der Routenplanung). Vor-

teilhafter ist es, direkt zu wissen, wo eine Fahrstrecke beginnt und wo sie endet, ohne diese Fakten erst aus den einzelnen Abschnitten zusammensetzen zu müssen. Deswegen besitzt die Fahrstrecke eben oben aufgeführte Attribute. Lediglich, wenn eine Linie grafisch angezeigt werden soll, spielen auch die Streckenabschnitte eine Rolle.

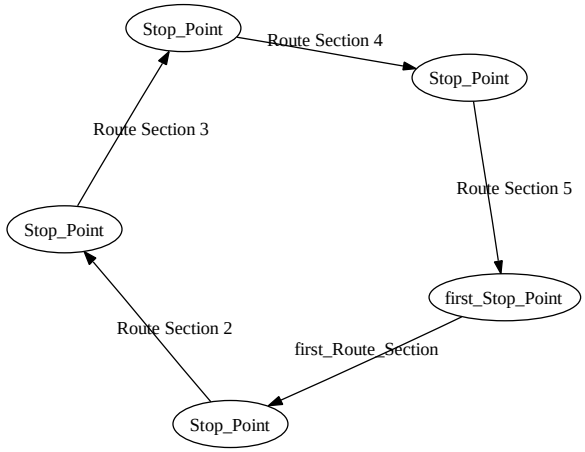
### Beispiel

In Abbildung 3.8 ist der Ausschnitt aus einer Linie aufgezeigt. Er stellt eine Strecke dar, die ihrerseits wiederum drei Fahrstrecken enthält: von „Stop_Point 1“ zu „Stop_Point 2“, von dort zu „Stop_Point 3“ und von dort zu „Stop_Point 4“. Zwischen „Stop_Point 2“ und „Stop_Point 3“ trifft der Weg auf eine Kreuzung („Public_Transport_Junction 1“), an der sie auf eine andere Linie stößt. Deswegen ist die Fahrstrecke in zwei Streckenabschnitte unterteilt. Nach der Haltestelle „Stop_Point 3“ gibt es wieder eine Kreuzung, die entweder zu „Stop_Point 4“ oder zu „Stop_Point 5“ weiterführt. Die vorliegende Linie folgt dem Weg zu „Stop_Point 4“.

### Fahrtrichtung entlang einer Strecke

Während in GDF die Elemente geordnet sind, d.h. automatisch eine Reihenfolge besitzen, ist dies in einem OWL-Dokument nur eingeschränkt möglich. Da eine Strecke jedoch in einer bestimmten Richtung entlang einer Menge von Fahrstrecken verläuft, muss diese Richtung modelliert werden können.

(a) Lineare Linie




(b) Linie in Form eines Ringes

Abbildung 3.9: Ermittlung der Richtung einer Strecke in OTN

Eine Starthaltestelle (*„first_Stop_Point“*) gibt den Beginn der Strecke an. Durch diesen Knoten kann sowohl die zugehörige erste Fahrstrecke und dann rekursiv daraus sowohl die Reihenfolge der weiteren Fahrstrecken als auch der Aufbau der Streckenabschnitte ermittelt werden (siehe Abb. 3.9 a). Wenn eine Strecke ein zusammenhängender Ring

ist (siehe Abb. 3.9 b), besitzt sie keinen Startpunkt, denn an jeden Knoten schließen immer zwei Abschnitte an. Für diesen Fall wird ein beliebiger Knoten als Startknoten gewählt und die erste Fahrstrecke mit *„first_Route_Section“* angegeben. Ansonsten hat diese Eigenschaft keine Bedeutung. Durch die Fahrtrichtung ist für jede Haltestelle auch die nachfolgende genau festgelegt. Mit Hilfe von Fahrplänen können realistische Abbildungen von öffentlichen Verkehrsmitteln erstellt werden. Da Fahrpläne von der Fahrtrichtung und damit vom Verlauf einer Strecke abhängen, werden sie analog zu den Fähren mit *„timetable“* definiert. Während bei den Fähren der Knoten, für den der Fahrplan (*„starts_at“*) gilt, nur Start oder Endknoten einer Fährverbindung sein kann, verweist er bei einer Strecke auf eine beliebige darin enthaltene Haltestelle. Die Strecke zur nächsten Haltestelle ergibt sich dann aus der Ausgangshaltestelle und der Fahrtrichtung. Alle anderen Untereigenschaften des Fahrplans haben im öffentlichen Verkehr dieselbe Bedeutung wie bei der Fähre. Lediglich die Angabe eines Fahrplans für eine Endhaltestelle macht keinen Sinn, falls es sich dabei nicht gleichzeitig in einem Ring um eine Starthaltestelle handelt.

### 3.2.7 Landbelegung und -nutzung

Weil Informationen über die Landnutzung keinen großen Einfluss auf ein Verkehrsnetz haben, sondern lediglich zur Visualisierung nützen und in GDF bereits sehr detailliert untergliedert sind, wurden sie weitgehend unverändert in OTN eingebaut. Dabei sind die Eigenschaften *„building_Class_Name“*, *„sand_Area_Type“* exakt übernommen worden. In die Eigenschaft *„park_Type“* wurde jedoch zusätzlich der Wert „Cemetery“ aufgenommen, da eine Grünfläche auch ein Friedhof sein kann. Neu ist allerdings die Eigenschaft *„is_Accessible_at“*, über die der Zugang zu dem jeweiligen Merkmal beschrieben ist.

### 3.2.8 Dienste

Bei der Entwicklung von OTN wurde darauf geachtet, dass auch Anfragen aus dem Alltag in dem Datenmodell abgebildet sind und damit bearbeitet werden können, wie z.B.:

- Finde den Weg zur nächsten Apotheke.
- Gib mir alle Restaurants in meiner Nähe an.

Zur Beschreibung solcher Sachverhalte ist in GDF der Begriff der Dienste eingeführt worden. Es existieren zwar Vorschläge für eine Reihe möglicher Dienste, doch sind dies eben nur Vorschläge. Der einzige fest definierte Dienst ist *„Entry_Point“*, der den echten Zugang zu einem Dienst darstellt.

In OTN wurden die Vorschläge für Dienste aus GDF mit eigenen Ideen kombiniert. Hier enthält das Thema *„Services“* einen Großteil der Vorschläge aus GDF. Es entfällt aber der Dienst *„Entry_Point“*, da er, wie in Kapitel 3.1.8 erwähnt, keinen wirklichen Dienst darstellt, sondern nur zur Beschreibung der Lage und damit der Erreichbarkeit eines Dienstes mit der Eigenschaft *„is_Accessible_at“* nützt. Aber das ist automatisch

jedem Dienst impliziert, gleichgültig, ob er in einem Gebäude liegt oder z.B. erst durch das Überqueren einer Straße erreichbar ist.

Die wichtigste Art von Dienst ist der *„Transfer_Service“*, der in Kapitel 3.2.2 bereits erläutert wurde. Die darin enthaltenen Parkplätze *„Parking“* besitzen die Eigenschaften *„parking_Facilities_Available“* zur Angabe der verfügbaren Parkplätze sowie *„free_Parking“* und *„park_And_Ride“*, um zu bestimmen, ob ein Parkplatz kostenlos ist bzw. zu einem Bahnhof dazugehört.

Ansonsten gibt es viele aus GDF übernommene Dienste, die ebenfalls nur angedacht und keinesfalls vollständig sind und deswegen hier auch nicht weiter erwähnt werden. Abbildung 3.10 gibt eine Übersicht über die verfügbaren Dienste und deren mögliche Gruppierung. Bei Bedarf muss diese Liste noch ergänzt und mit weiteren Eigenschaften ausgestattet werden.

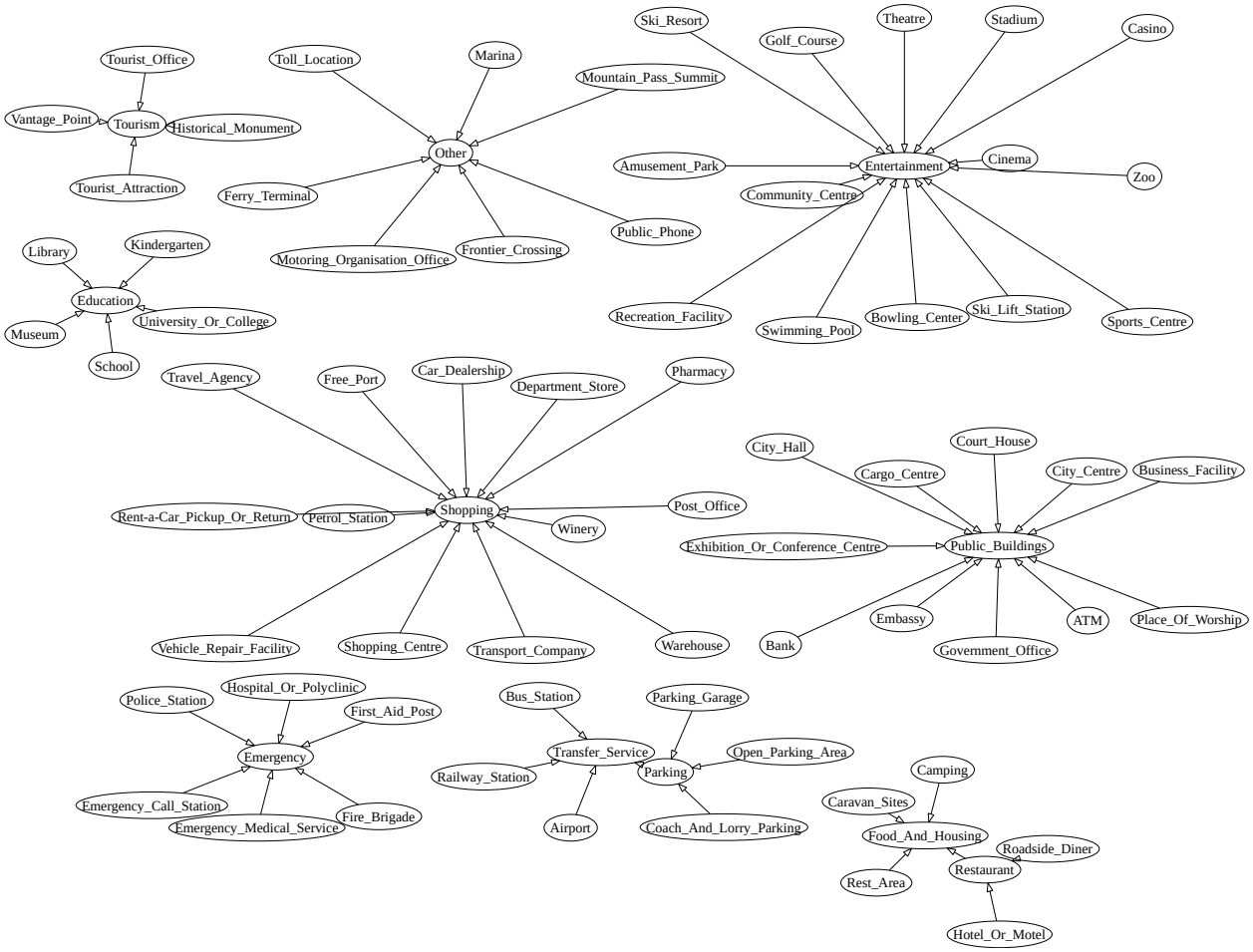



Abbildung 3.10: Gruppeneinteilung von Diensten in OTN

### 3.2.9 Meteorology

Neu in OTN ist die Möglichkeit, Informationen über das Wetter im Datenmodell zu speichern. Dazu gibt es das Thema *„Meteorology“*, das sich in die Klassen *„Temperature“* und *„Weather“* aufteilt. Diese sind Unterklassen von *„Face“* und definieren für eine Fläche die aktuelle Temperatur in Grad mit der Eigenschaft *„degree“* bzw. die Art des Wetters mit *„weatherType“*. Dabei sind die Werte Schnee („snow“), Schneeregen („sleet“), Hagel („hail“), Tau(„dew“), Regen („rain“), Schauer („shiver“) und Gewitter („storm“) möglich.

# 4 Framework für Visualisierung und Geospatial Reasoning

Mit OTN steht eine umfangreiche Beschreibungssprache für Verkehrsnetzwerke zur Verfügung. Für diese Ontologie sollte ein *„Framework“* geschaffen werden, mit dem man Informationen extrahieren, analysieren und auf einer Karte anzeigen kann.

Dabei war es wichtig, eine frei konfigurierbare Zusammensetzung der einzelnen Daten zu ermöglichen, so dass sowohl verschiedene Datenquellen zusammen als auch einzelne Teilbereiche daraus verarbeitet werden können, ohne den zu Grunde liegenden Rahmen ändern zu müssen. Dabei sollte es möglich sein, dass die Daten von einem anderen Server kommen oder sogar in einem anderen Format vorliegen, welches allerdings erst nach OTN konvertiert werden muss. Das folgende Kapitel beschäftigt sich mit der Entwicklung dieses *„Frameworks“*.

## 4.1 Aufbau des Frameworks

Der Begriff Framework heißt wörtlich übersetzt (Programm-) Gerüst oder Skelett. Dies bedeutet, dass ein Hauptprogramm existiert, welches die grobe Steuerung übernimmt und in das einzelne Komponenten „eingebettet“ werden, die dann spezifische Aufgaben innerhalb dieser Anwendung ausführen. Dazu müssen diese Komponenten im Framework registriert werden, d.h. dem Framework muss mitgeteilt werden, welche Komponenten zur Verfügung stehen und wie diese verwendet werden sollen. Für diesen Zweck stellt ein Framework Schnittstellen zur Verfügung, die bestimmen, wie die Komponenten angesprochen und welche Art von Antworten zu erwarten sind. Mit Hilfe solcher Schnittstellen können individuelle Komponenten geschrieben werden, mit denen das Framework kommunizieren kann, insofern sie die Schnittstelle implementieren, d.h. ihr entsprechen.

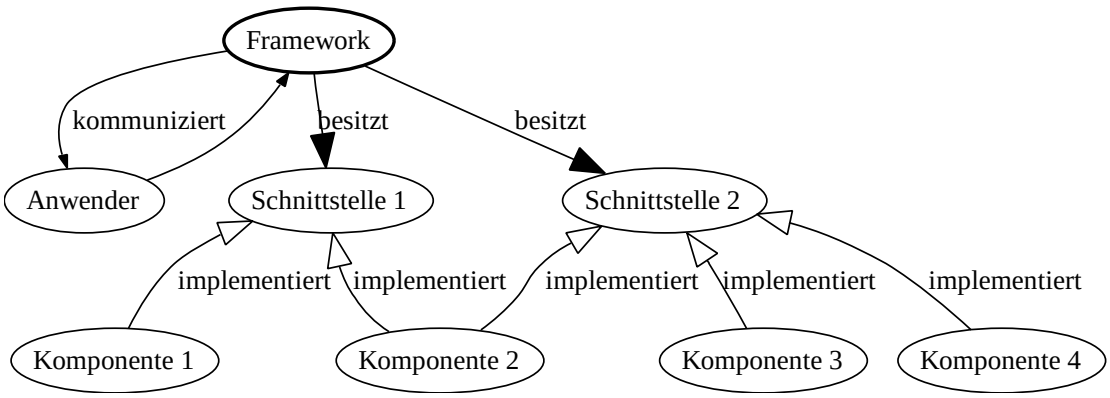



Abbildung 4.1: Prinzipieller Aufbau eines Frameworks

Abbildung 4.1 zeigt den schematischen Aufbau eines beliebigen Frameworks mit zwei Schnittstellen, für deren Implementierung eine Reihe von Komponenten geschrieben wurde. Ein Anwender kommuniziert über bestimmte Befehle mit dem Framework. Falls erforderlich, leitet das Framework die Anfrage an eine oder mehrere Komponenten weiter, damit diese bestimmte Aufgaben erfüllen. Anschließend wertet es das Resultat aus und liefert die Antwort für den Anwender.

Das geschaffene Framework ist in Java [41] geschrieben und hat zwei grundlegende Aufgaben:

- Visualisierung der Daten in SVG
- Vorverarbeitung der Daten für ortsbezogene Anfragen und automatische Folgerungen (*Geospatial Reasoning*)

Die Visualisierung der Daten in SVG ist der Hauptbestandteil der vorliegenden Arbeit. Dabei baut das Framework auf den Erfahrungen aus der dieser Arbeit vorausgehenden Projektarbeit [17] auf, in der eine Transformation von Daten aus dem Format *Geography Markup Language* (GML) [15] nach SVG durchgeführt wurde. Die Architektur ist hier jedoch grundlegend überarbeitet und durch viele neue Konzepte entscheidend umgestaltet worden.

Zusätzlich zu einer Schnittstelle zur Durchführung von Anfragen an einen Server bietet das Framework eine Reihe von Funktionen an, mit denen die Daten eingelesen und vorverarbeitet werden können. Jedoch gibt es bislang lediglich eine einzige Komponente zum Suchen von Straßen. Es herrscht also immer noch ein großer Bedarf an zusätzlichen Komponenten für Anfragen, wie z.B. für die Suche nach dem kürzesten Weg zu einem bestimmen Ort.

## 4.2 Visualisierung der Daten in SVG

Die Visualisierung der Daten erfolgt mit Hilfe einer SVG-Komponente, die in ein HTML-Dokument eingebettet ist. Dieses unterteilt sich, wie Abbildung 4.2 zeigt, in eine Karte (links) und eine Baumstruktur (rechts) und dient dazu, bestimmte Teile der Karte ein- oder auszublenden. Der Baum unterteilt sich in die Bereiche „Module“ und „Ontologien“. Zusammengehörende Elemente, wie „Straßen in München“ oder „MVV“, können in der Konfiguration in sogenannten Modulen zusammengefasst werden (s. Kapitel 4.6). Neben der eigentlichen Ontologie ist es auch noch möglich, eine „Darstellungsontologie“ zu definieren, in welcher die dargestellten Objekte eingeordnet werden. Knoten in diesem Baum können mit Hilfe einer Checkbox ein- bzw. ausgeschaltet und damit alle dazugehörigen Informationen auf der Karte sichtbar bzw. unsichtbar gemacht werden.

### 4.2.1 Aufbau des SVG-Dokuments

Eine Karte in SVG soll über das Internet verfügbar sein. Deswegen ist es wichtig, so wenig Daten wie möglich zum Client zu übertragen und nur bei Bedarf neue Elemente

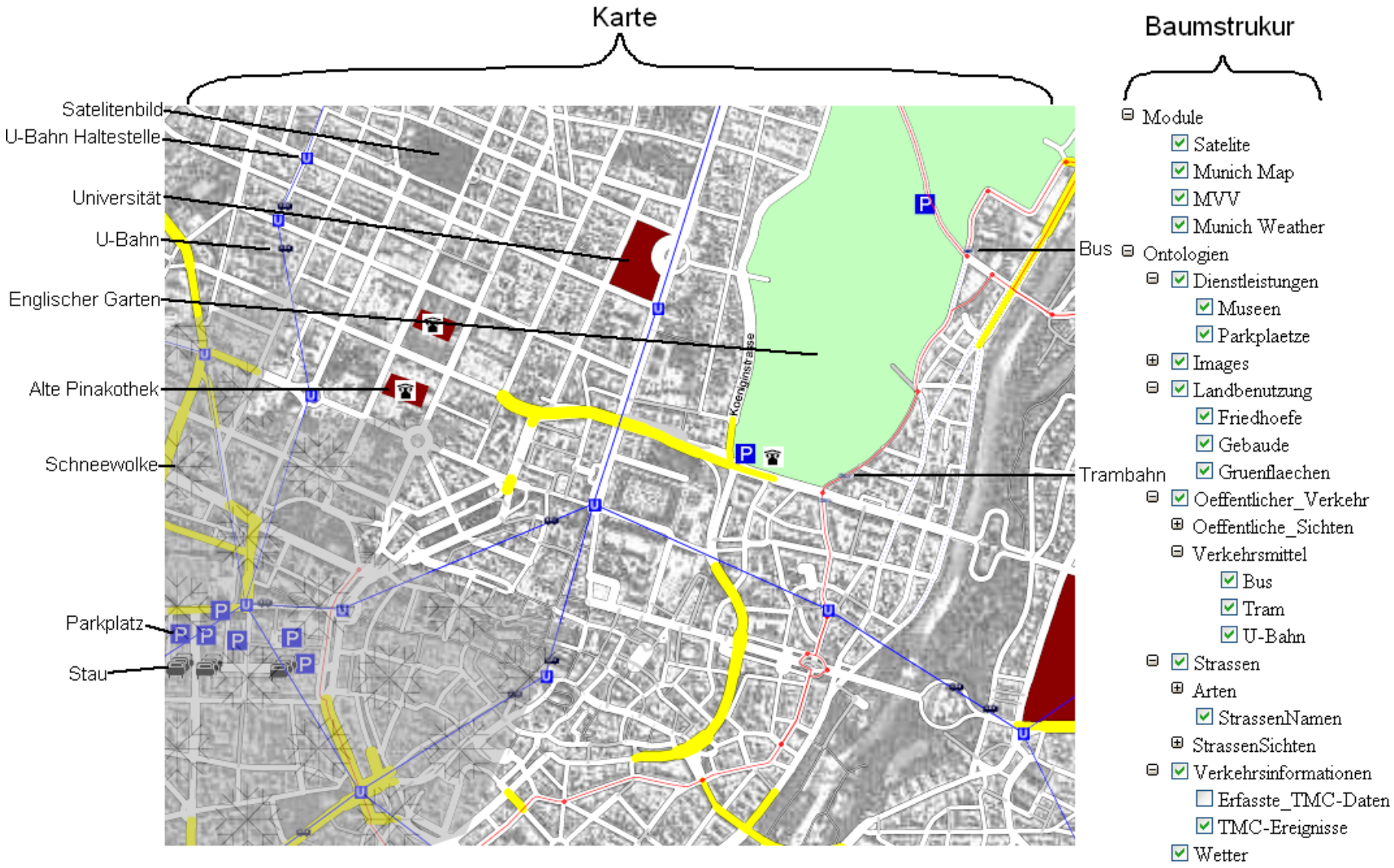



Abbildung 4.2: Visualisierung in SVG

nachzuladen. Zwar kann durch die Verwendung von komprimierten SVG-Dokumenten (svgz) die zu übertragende Dateigröße um 80% reduziert werden, jedoch besitzt SVG keine Funktion, um nachträglich andere Dokumente in ein SVG-Dokument einzufügen. Deswegen wurde mit Hilfe der SVG-DOM Schnittstelle ein Mechanismus geschaffen, der Daten dynamisch nachladen kann. Doch ist beim Aufbau eines SVG-Dokuments eine Reihe von Konventionen zu befolgen:

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- file:Dateiname.svgz -->                                    (1)
<svg>
  <g >
    <!-- Start of LOADNODE -->                                  (2)
    ...
    <!-- End of LOADNODE -->                                    (3)
  </g>
</svg>
```

Generell wurde das Modell so angelegt, dass man auch jedes beliebige Teildokument für sich betrachten bzw. in ein anderes Dokument einbinden kann. Deswegen ist jedes Dokument, das nachgeladen werden kann, ein komplettes SVG-Dokument, d.h. es beginnt mit `<?xml ...?>`, gefolgt von dem Element „svg“, das die Zeichenbefehle enthält. Innerhalb des Dokuments ist ein bestimmter Teil durch „Start of LOADNODE“ (2) und „End of LOADNODE“ (3) gekennzeichnet, der den eigentlichen, in das andere Dokument zu integrierenden Inhalt aufweist.

Das SVG-Plugin von Adobe enthält ein paar nützliche Funktionen, die bisher in keinem anderen Plugin verfügbar sind. So stellt es mit

```
getURL(URL, METHODE);
```

eine Funktion zur Verfügung, die eine Datei über eine Adresse („URL“) im Hintergrund lädt und dann die angegebene Methode („METHODE“) aufruft. Dieser Methode wird als Argument ein Objekt mit der Eigenschaft „success“ übergeben, welche anzeigt, ob der Ladevorgang erfolgreich war („true“) oder nicht („false“). Falls er erfolgreich war, steht in der Eigenschaft „content“ der Inhalt der Datei. Jedoch gibt es keine Information über den Namen der geladenen Datei. Da z.T. mehrere Dateien parallel geladen werden, ist die genaue Bestimmung der Datei oder der exakten Platzierung im Dokument nicht möglich. Deswegen muss dort eine Kommentarzeile stehen mit dem Inhalt „file:...“ (1), die den Namen der Datei enthält. Durch diese Angabe kann die zugehörige Stelle im Orginaldokument ermittelt und der angegebene Inhalt dort eingefügt werden.

Hier bietet das SVG-Plugin noch eine weitere nützliche Funktion an, die eine Zeichenkette („TEXT“) verarbeitet, welche im XML-Format vorliegen muss:

```
parseXML(TEXT, DOCUMENT);
```

Die oben genannte Funktion erzeugt daraus die zugehörige Baumstruktur für das angegebene SVG-Dokument („DOKUMENT“) und liefert sie als Ergebnis zurück. Mit Hilfe der DOM-spezifischen Funktion `appendChild(node)` kann dieser Teil dann an der gewünschten Stelle im Dokument eingefügt werden.

### 4.2.2 Rasterung der Daten

Ein Problem von SVG besteht in der mangelnden Fähigkeit, ausschließlich die wirklich angezeigten Objekte zu berechnen. Dies führt bei großen Datenmengen zu gewaltigen Performanceeinbußen. Sinnvollerweise müssen alle Elemente außerhalb des aktuell sichtbaren Bereichs ausgeblendet (unsichtbar gemacht) werden, so dass deutlich weniger Objekte anzuzeigen und damit zu berechnen sind. Außerdem sollte ermittelt werden, welche Objekte im aktuell sichtbaren Bereich liegen, damit diese nachgeladen werden können.

Also wurde das Konzept der Rasterung entwickelt. Dabei wird die Karte in Gebiete aufgeteilt. Jedes Gebiet hat eine gewisse Fläche und enthält alle in ihr liegenden Objekte. Bei jeder Änderung des Sichtbarkeitsbereiches durch Vergrößern / Verkleinern (Zooming) oder Verschiebung (Panning) werden nur all jene Gebiete geladen bzw. angezeigt, die den aktuell sichtbaren Bereich schneiden. Die anderen werden ausgeblendet. Somit müssen nur noch alle relevanten Objekte berechnet werden.

Zur Rasterung der Daten wurden in der vorausgehenden Projektarbeit zwei Modelle entwickelt:

**Quadratisches Raster:** Die Karte wird mit einem quadratischen Raster überzogen. Alle Objekte (z.B. Straßenzüge), die in mehreren Rastern liegen, kommen mehrfach vor und werden daher auch mehrfach gezeichnet.

**Raster entlang von Merkmalen:** Es wird versucht, ein so feines Raster zu bilden, dass möglichst keine Objekte in mehreren Rastern zu liegen kommen, z.B. bei der Ausrichtung entlang der Straßenzüge.

Die Bestimmung der sichtbaren Rasterpunkte im Modell 1 ist relativ einfach. Im Modell 2 muss jedoch für jeden Rasterpunkt eine detaillierte Positionsbeschreibung vorliegen, da dort die Raster keine Rechtecke darstellen. Auch die Ermittlung von sinnvollen Rasterungen im Modell 2 ist relativ aufwändig. In der Praxis entspricht der Teil von Objekten, die bei quadratischer Rasterung in mehreren Rastern liegen würden, weniger als 1% der Gesamtobjekte. Dies ist jedoch nur ein kleiner Nachteil im Vergleich zu der deutlich schwierigeren Implementierung des Modells 2, so dass in der Projektarbeit das erste Modell bevorzugt wurde.

In beiden Fällen muss eine Information darüber vorliegen, auf welchen Rasterpunkten sich Daten befinden, damit explizit nur diese geladen werden und keine unnötige Last entsteht. Dazu wurde eine externe Datei erzeugt, die Angaben über alle vorhandenen Rasterpunkte enthält. Bevor nun ein Rasterpunkt geladen wurde, erfolgte ein Abgleich über die vorhandenen Daten und die entsprechenden Dateien. Es stellte sich jedoch heraus, dass bereits bei mittleren Datenmengen (10 MB) diese Informationsdatei relativ groß (500 KB) wurde. Außerdem musste sie zu Beginn der Anwendung immer komplett

geladen werden. Dieses Konzept eignete sich daher nicht für größere Datenmengen, zumal die Anzahl der Daten in den einzelnen Rasterpunkten bei einem quadratischen Raster von einem einzigen bis zu mehreren tausend Objekten gehen kann.

**R-Baum:**
Für Datenbanksysteme wurde mit dem R-Baum [38] eine Struktur zum Speichern von zweidimensionalen Daten geschaffen: Jeder Knoten enthält neben seinen Kindern auch eine Information über sein „minimal umgebendes Rechteck“, d.h. über das kleinstmögliche Rechteck, das alle Kinder enthält. Dabei sind alle Objekte mitsamt ihren „minimal umgebenden Rechtecken“ nur in den Blättern des R-Baumes enthalten, und verschiedene „in der Nähe“ liegende Rechtecke werden in sogenannten Indexknoten zusammengefasst. Solche Indexknoten können ihrerseits wiederum solange in weiteren Indexknoten zusammengefasst werden, bis schließlich ein großes Rechteck alle Bereiche enthält. Dabei ist es erlaubt, dass sich einzelne Recktecke überlappen. Es ist in der Regel nicht möglich, einen „optimalen“ R-Baum zu erzeugen, da dieser Vorgang bei N Elementen eine Komplexität von $2^N$ besitzt. Deswegen gibt es keine eindeutige Darstellung für einen R-Baum, sondern nur eine Reihe von Algorithmen zur Optimierung seines Aufbaus. Dabei gilt jedoch das Kriterium, dass die Rechtecke möglichst kleine Bereiche abdecken sollen.

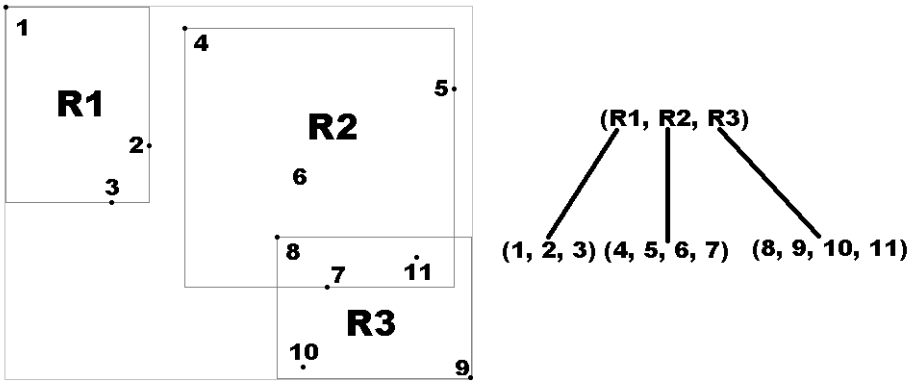



Abbildung 4.3: Lage der Punkte, ihre Bereiche und der dazugehörige R-Baum

Abbildung 4.3 zeigt ein Beispiel für einen möglichen Aufbau eines R-Baums: Bereich R1 enthält die Punkte 1-3, R2 die Punkte 4-7 und R3 die Punkte 8-11. Die Wurzel ist ein Indexknoten, der die Bereiche R1, R2 und R3 enthält, wobei sich die Bereiche R2 und R3 überlappen, jedoch jeder Punkt in der Überlappung nur in einem der beiden Bereiche enthalten ist.

Ein R-Baum als Rasterstruktur verbindet die Vorteile der quadratischen Rasterung mit denen der Rasterung entlang von Objekten und behebt deren Nachteile. „Rasterpunkte“ sind dabei die (sich überlappenden) minimalen Rechtecke der Blätter, wobei jedes Objekt in genau einem Rasterpunkt liegt und jeder Rasterpunkt annähernd gleich viele Objekte aufweist. Jeder Indexknoten enthält lediglich die Informationen über die Dateien seiner direkten Kinder und die Größen der sie umgebenden Rechtecke. Damit

können die Daten rekursiv an der Wurzel durchsucht werden und, falls ein Teil in den aktuell sichtbaren Bereich hineinreicht, dieser nachgeladen werden. Dadurch entfällt die Notwendigkeit einer Informationsdatei und das Modell ist leicht zu implementieren. Außerdem sind die Daten in den Rastern gleichmäßig verteilt und kommen nicht mehrfach vor.

### 4.2.3 Detailstufen

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 entspräche dann dem Wert 300.

### 4.2.4 Erweiterung der Gruppen von SVG

Zusätzlich zu den üblichen Attributen aus SVG werden durch ein Javascript-Programm noch weitere Attribute in Gruppen („<g>“) unterstützt. Dazu werden rekursiv alle sichtbaren Element daraufhin durchsucht, ob sie eben diese Attribute besitzen und dementsprechend notwendige Operationen ausgeführt, z.B. Laden oder Ausblenden. An neuen Attributen werden unterstützt:

**bBox** bestimmt das Rechteck, in dem die Elemente der Gruppe liegen.

**minDetail** gibt die minimale Detailstufe der Gruppe an.

**maxDetail** gibt die maximale Detailstufe der Gruppe an.

**loadUrl** ist die Adresse, aus welcher der Inhalt der Gruppe bei Bedarf geladen werden soll.

**updatePeriod** gibt die Zeit in Sekunden an, nach welcher der Inhalt der Gruppe aktualisiert werden soll.

**ontology** stellt eine durch Komma getrennte Liste der Klassennamen in der Darstellungsontologie dar, in die das Objekt eingeordnet werden kann.

**module** gibt den Namen des Moduls an, der im Baum dargestellt werden soll.

## 4.3 Dynamische Karten in SVG mit Hilfe von Javascript

### 4.3.1 Dynamisches Nachladen und Ausblenden

SVG beinhaltet standardmäßig Funktionen zum Ändern des sichtbaren Bereiches durch Zoomen oder Scrollen. Jedoch werden eine Reihe von weiteren notwendigen Funktionen, wie dynamisches Nachladen oder Ausblenden, dabei nicht unterstützt. Die Änderung des sichtbaren Bereiches kann mit `onzoom` und `onscroll` abgefangen und bearbeitet werden. Dazu wird nach jeder Änderung die Methode `checkLoading` aufgerufen, die weitere Aktionen überprüft.

Zuerst wird dabei der aktuell sichtbare Bereich ermittelt. Dies erfordert z.T. recht umfangreiche Berechnungen, da in SVG lediglich die aktuelle Zoomstufe und die Verschiebung relativ zur Zoomstufe in x- und y- Richtung verfügbar sind. Daneben spielt die bBox der übergeordneten „<svg>“ - Elemente und die Größe des Fensters eine Rolle. Des Weiteren können Elemente verschoben oder gedreht oder durch das Attribut „preserveAspectRatio“ Verzerrungen entfernt worden sein.

Es folgt ein Beispiel zur Berechnung des Bereiches, wenn alle Elemente lediglich in einem SVG-Element liegen und weder verschoben noch gedreht sind, jedoch evtl. eine Verzerrung vorliegt. Kompliziertere Situationen wurden in [37] behandelt, woher auch das Skript stammt, mit dem der sichtbare Bereich für die Karte ermittelt wird.

Die SVG-Grafik ist „getInnerWidth()“ Pixel breit, „getInnerHeight()“ Pixel hoch und stellt den Bereich (x, y) - (x + width, y + width) dar, der im Attribut bbox mit „x y width height“ angegeben ist. Für die Zoomstufe besitzt ein SVG-Dokument („svg“) die Methode „svg.getCurrentScale()“, wobei der Wert „4“ vierfache Vergrößerung bedeutet. Die Verschiebung in x- bzw. y- Richtung relativ zur Zoomstufe und Fenstergröße wird mit „svg.getCurrentTranslate().x“ und „svg.getCurrentTranslate().y“ ermittelt. Damit berechnet sich der sichtbare Bereich aus:

```
var viewboxes = svg.getAttribute("viewBox").split(' ');
var x = parseInt(viewboxes[0]);
var y = parseInt(viewboxes[1]);
var width = parseInt(viewboxes[2]);
var height = parseInt(viewboxes[3]);

var scale = svg.getCurrentScale();

var transX = -width * svg.getCurrentTranslate().x / getInnerWidth();
var transY = -height * svg.getCurrentTranslate().y / getInnerHeight();
CURRENT_VIEWBOX = new ViewBox(x + transX / scale,
                              y + transY / scale,
                              width / scale,
                              height / scale);
```

Mit Hilfe dieses sichtbaren Bereiches werden die Gruppen des Dokuments nach bestimmten Attributen durchsucht. Wenn eines der Werte von „bBox“, „minDetail“, „max-

Detail“ nicht mit dem aktuell sichtbaren Bereich vereinbar ist, wird die zugehörige Gruppe ausgeblendet, d.h. der Wert „style“ wird mit

```
group.getStyle().setProperty("display", style);
```

auf unsichtbar („none“) gesetzt, ansonsten auf sichtbar („inline“). Ist ein Element sichtbar, wird kontrolliert, ob es geladen werden muss. Dazu werden die Attribute „loadUrl“ und „updatePeriod“ gesucht. Falls sie existieren und ein Element noch nicht vorhanden oder aber nicht mehr gültig ist, wird es über diese „loadUrl“ geladen bzw. ersetzt.

Diese relativ umfangreichen Berechnungen können nicht bei jeder Bewegung gleichzeitig durchgeführt werden, da beim Scrollen evtl. viele solche Bewegungen hintereinander ablaufen und daher mit ständigen Unterbrechungen zu rechnen wäre. Deswegen wird die Berechnung erst gestartet, nachdem eine gewisse Zeit (200 ms) keine Bewegung mehr stattgefunden hat.

### 4.3.2 Erweiterung des Zooms

Das SVG-Plugin von Adobe weist eine Beschränkung des Zoomlevels auf einen Bereich zwischen 1/16 und 16 auf, und es führt zu Fehlern in der Anwendung, wenn die Verschiebung („getCurrentTranslate()“) Werte annimmt, die größer als 16000 sind. Da eine Karte jedoch die Möglichkeit besitzen muss, beliebig weit hinein oder heraus zu zoomen, wird folgender Trick angewandt: Generell befindet sich die Karte immer im Zoomlevel „4“ und besitzt eine Standard-Verschiebung. Wenn nun der sichtbare Bereich geändert wird, weichen die Werte von diesen Standardwerten ab. Daraufhin wird der aktuell sichtbare Bereich berechnet, die ViewBox des äußersten SVG-Elements an die neuen Werte angepasst und die Karte wieder in den Standardzustand zurückgesetzt. Da jede Änderung eines dieser drei Werte wiederum eine Änderung des sichtbaren Bereiches darstellt und damit die Funktion erneut aufruft, wird immer nur eine dieser Änderungen auf einmal durchgeführt, so dass die Funktion maximal viermal aufgerufen wird. Das Zoomlevel „4“ wird deswegen verwendet, weil beim Herauszoomen immer nur um den Faktor „2“ gezoomt werden kann und somit der Wert „1“ nicht erreicht wird. Falls nun der Zoomlevel den Wert „1“ besitzt, kann dies nur durch den Aufruf des Befehls „Original View“ im Standard SVG-Popupmenu erfolgt sein, und damit wird die Karte automatisch wieder auf die Anfangswerte zurückgesetzt.

Die tatsächliche Zoomstufe wird dabei in dem Skript, das den manuellen Zoom durchführt, in der Variablen „this.DYNAMIC_ZOOM_FACTOR“ und die aktuelle Verschiebung in „this.translateX“ bzw. „this.translateY“ gespeichert. Die Standardwerte sind „DYNAMIC_LEVEL“ und „this.newX“ und „this.newY“. Die ursprüngliche ViewBox liegt in dem Array „this.coords“.

```
if (currentScale != DYNAMIC_LEVEL) {
  // Zoomlevel hat sich geändert
  svg.setCurrentScale(DYNAMIC_LEVEL);
  if (currentScale == 1) {
    // Zurücksetzen in Originalzustand
```

```
    this.DYNAMIC_ZOOM_FACTOR = 1 / DYNAMIC_LEVEL;
    this.translateX = 0;
    this.translateY = 0;
  } else {
    this.DYNAMIC_ZOOM_FACTOR *= currentScale / DYNAMIC_LEVEL;
  }
} else {
  // Zoomlevel hat sich nicht geändert
  var dx = svg.currentTranslate.x - this.newX;
  var dy = svg.currentTranslate.y - this.newY;
  var newTranslate = svg.getCurrentTranslate();
  if (dx != 0) {
    // Es gab eine Verschiebung in x-Richtung
    var translateX = -this.width * dx / getInnerWidth() /
                     (this.DYNAMIC_ZOOM_FACTOR * svg.currentScale);
    this.translateX += translateX;
    svg.currentTranslate.x = this.newX;
  } else if (dy != 0) {
    // Es gab eine Verschiebung in y-Richtung
    var translateY = -this.height * dy / getInnerHeight() /
                     (this.DYNAMIC_ZOOM_FACTOR * svg.currentScale);
    this.translateY += translateY;
    svg.currentTranslate.y = this.newY;
  } else {
    // Aktualisiere die ViewBox des SVG-Elements
    var newCoords = new Array();
    for (var i = 0; i < 4; i++) {
      newCoords[i] =  this.coords[i] / this.DYNAMIC_ZOOM_FACTOR;
    }
    newCoords[0] += this.translateX;
    newCoords[1] += this.translateY;
    var newBBox = newCoords.join(" ");
    if (svg.getAttribute("viewBox") != newBBox) {
      svg.setAttribute("viewBox", newBBox);
    }
  }
}
```

## 4.4 Transformations-Ontologie

Zur Visualisierung von Daten aus OTN mit SVG wird ein Algorithmus benötigt, der diesen Vorgang automatisiert, jedoch Möglichkeiten bietet, sowohl das Aussehen jedes Merkmals zu bestimmen als auch zukünftige Merkmale zu transformieren. Auch soll die

Bedienung und Konfiguration einfach und intuitiv sein.

Zur Transformation gibt es zwei Ansätze: Der erste ist die Transformation mit Hilfe eines XSLT-Stylesheets [7]. Dies ist eine weit verbreitete Methode, ein XML-Dokument in ein anderes zu überführen. Jedoch sind die Transformationen, insbesondere die Umwandlung der geometrischen Strukturen nach SVG sehr kompliziert, und deswegen wird eine solche Transformation relativ schnell unübersichtlich.

Aus diesem Grund ist eine Anwendung in Java geschrieben worden, die Routinen für die wichtigsten Grundtransformationen besitzt, aber über Konfigurationsdateien beliebige weitere Transformationen durchführen kann. Die eigens dafür entwickelte sogenannte Transformationsontologie basiert ebenfalls auf OWL und beschreibt Regeln zur Überführung von Merkmalen nach SVG.

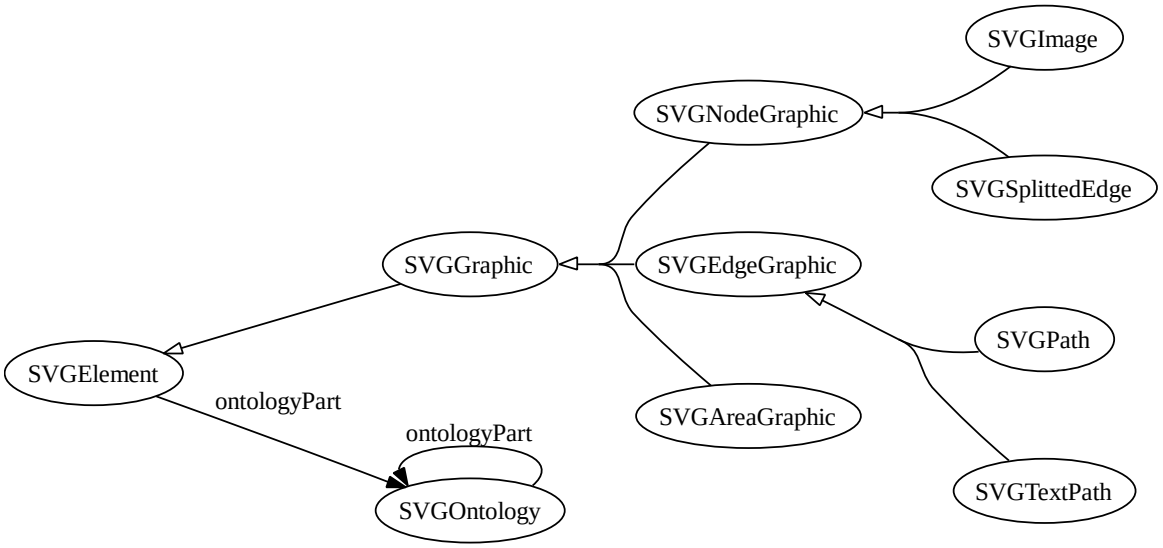



Abbildung 4.4: Transformationsontologie für die Umwandlung von OTN nach SVG

Abbildung 4.4 zeigt den Aufbau dieser Ontologie. Mit *„SVGOntology“* wird die Struktur der Elemente im SVG-Dokument definiert, die rechts im Baum im Unterpunkt „Ontologie“ angezeigt werden sollen. Dabei besitzt jedes dieser Strukturelemente die Eigenschaft *„name“*, die den dargestellten Namen im Baum angibt. Mit *„ontologyPart“* kann der Vaterknoten angegeben werden, in dem das Strukturelement liegt. Dadurch wird eine Baumstruktur definiert, denn alle Elemente, die einen Vaterknoten haben, liegen auch in diesem, und falls ein Element keinen Vater besitzt, wird es selbst als Kind des „Ontologie“-Knotens dargestellt. Mit *„hideable“* kann man angeben, ob ein Strukturelement über eine Checkbox ausgeblendet werden kann („true“) oder nicht („false“). Die Eigenschaft *„hidden“* bestimmt den Zustand des Strukturelementes beim Laden des Dokuments. Es ist eingeblendet (die Checkbox ist markiert und die zugehörigen Objekte sichtbar), falls der Wert „false“ angenommen wird. Bei „true“ ist es ausgeblendet.

*„SVGElement“* bildet eine Oberklasse für die Transformationen und beinhaltet dabei die gemeinsamen Eigenschaften einer Gruppe von Grafiken, welche durch dieselbe

Transformation erzeugt wurden. Diese Eigenschaften sind unabhängig von den Daten, die damit übertragen werden. Mit *„ontologyPart“* können der Gruppe beliebig viele ihr angehörende Strukturelemente zugewiesen werden. Zur Rasterung werden die Daten innerhalb einer Gruppe in einem R-Baum gespeichert. Die maximale Anzahl von Grafiken in einem Knoten kann mit der Eigenschaft *„leafCapacity“* bestimmt werden, wobei „50“ der Standardwert ist. Die Detailstufe muss mit *„minDetail“* und *„maxDetail“* angegeben werden. Wie lange ein Element gültig ist, bestimmt der Wert *„updatePeriod“*, der die Dauer in Sekunden festlegt. Bestimmte Gruppen sollen über andere gezeichnet werden (z.B. erst die Straßen und darüber dann die Straßennamen). Also ist es notwendig, eine Reihenfolge bei der Darstellung der Gruppen zu bestimmen. Dazu kann mit *„paintingOrder“* eine Ordnung definiert werden, nach der die Gruppe mit dem kleinsten Wert zuerst und die mit dem größten zuletzt gezeichnet wird. Benötigt eine Gruppe darüberhinaus noch weitere Attribute, sind diese mit der Eigenschaft *„groupAttributes“* festzulegen.

### 4.4.1 Transformationsformeln

Zur Darstellung der einzelnen Merkmale muss innerhalb der Transformationsontologie auf ihre Eigenschaften zugegriffen und mit ihnen Berechnungen durchgeführt werden. Für diesen Zweck können alle Eigenschaften, die einzelne Merkmale betreffen, als Formeln dargestellt werden. Eine Formel beginnt immer mit einem „=“, gefolgt von der eigentlichen Formel, die aus einer Reihe von vordefinierten Operationen besteht. Wenn eine Eigenschaft ohne „=“ beginnt, handelt es sich um einen statischen Text, der für alle Merkmale der Gruppe gleich ist. Falls eine Operation zu keinem Ergebnis führt, wird immer der Wert „0“ angenommen. Dies kann passieren, wenn ein Fehler in der Syntax vorliegt oder eine Berechnung nicht möglich ist. Da Texte häufig runde Klammern enthalten, werden zur Klammerung in den Formeln geschweifte Klammern „{}“ verwendet.

Innerhalb von Formeln können durch „+“, „-“, „*“, „/“ und „%“ Grundrechenarten auf Zahlen angewendet werden. Ist eines der Argumente der Operation keine Zahl, wird „0“ zurückgeliefert. Eine Ausnahme bildet dabei die Operation „+“, die beide Argumente zu einem Text zusammenfügt. Bei „*“ wird, falls das erste Argument ein Text ist und das zweite eine positive ganze Zahl, der Text der Zahl entsprechend oft hintereinander geschrieben. So liefert „Hallo * 2“ das Ergebnis „HalloHallo“.

An Vergleichsoperationen gibt es „<“, „>“, „<=“, „>=“, „<>“ und „=“. Sie liefern den Wert „1“, falls der Vergleich erfolgreich war und sonst „0“.

Durch die Verwendung eines Backslashes „\“ vor einem beliebigen Zeichen wird angezeigt, dass es sich hier um einen Text und nicht um eine Operation handelt. So können Operatoren als Text kodiert werden (z.B \<svg\>).

Umrechnungen in andere Formate werden durch das Symbol „@“ gekennzeichnet. Das Format wird dabei direkt vor dem „@“ angegeben, und der Wert, der umgerechnet werden soll, steht danach. Dabei gibt es bisher folgende vordefinierte Möglichkeiten:

- „INT@“ wandelt eine beliebige Zahl in eine ganze Zahl um.
- „TIME@“ extrahiert aus einer Zeitangabe die Uhrzeit.

- „DATE@“ extrahiert aus einer Zeitangabe das Datum.
- „DATETIME@“ extrahiert aus einer Zeitangabe das Datum und die Uhrzeit.

Durch den Ausdruck „IF ... THEN ... ELSE ...“ kann eine Bedingung definiert werden. Zwischen „IF“ und „ELSE“ steht die Bedingung. Falls diese erfüllt ist, wird der Wert zwischen „THEN“ und „ELSE“ angenommen, ansonsten der Wert hinter „ELSE“. Der Teil ab „ELSE“ ist jedoch optional.

Auf die Eigenschaften von Merkmalen kann mit „[EIGENSCHAFTS_NAME]“ zugegriffen werden, wobei innerhalb der Klammern der Name der gewünschten Eigenschaft steht. Die ID eines Merkmals, die im Prinzip keine Eigenschaft darstellt, kann mit „[ID]“ ermittelt werden. Solange eine Eigenschaft ein Merkmal ist, kann auf dessen Eigenschaften durch das Anschließen von „•[EIGENSCHAFTS_NAME]“ zugegriffen werden. Eine Kante beginnt z.B. an einem Punkt (*„starts_at“*), der eine *„x“*-Koordinate besitzt. Dessen Wert kann durch `[starts_at].[x]` ermittelt werden.

Falls es sich bei dem Merkmal um eine Linie (*„Line“*) handelt, wird für jedes Fahrzeug, das sich im Moment entlang dieser Linie bewegt, eine extra Grafik erzeugt. Dazu können zusätzlich zu den Standardeigenschaften weitere Eigenschaften abgerufen werden. Mit *„[x]“* bzw. *„[y]“* wird die aktuelle Position des Fahrzeugs ermittelt. Die Fahrstrecke, die gerade durchlaufen wird sowie die vorausgegangene bzw. nächste Haltestelle können mit *„[route_Section]“*, *„[starts_at]“* und *„[ends_at]“* referenziert werden. Die Zeitpunkte, an denen das Fahrzeug die Haltestelle verlassen bzw. erreichen wird, stehen in *„[startTime]“* bzw. *„[endTime]“*. Die Verweildauer an der letzten Haltestelle bzw. die Fahrtdauer stehen dabei mit *„[waitingTime]“* und *„[drivingTime]“* zur Verfügung.

### 4.4.2 Auswahl und Transformation von Merkmalen

Zur Transformation von einzelnen Merkmalen gibt es in der Transformationsontologie die Klasse *„SVGGraphic“*, die sich je nach geometrischer Beschaffenheit eines Merkmals in die Unterklassen *„SVGNodeGraphic“*, *„SVGEdgeGraphic“* und *„SVGAreaGraphic“* aufteilt. Im Gegensatz zu ihrer Oberklasse *„SVGElement“* dienen ihre Eigenschaften nur zur Auswahl und Transformation einzelner Merkmale innerhalb einer Gruppe.

Mit *„useOnClass“* können die Klassen aus OTN angegeben werden, aus denen die Merkmale dieser Gruppe stammen sollen. Wenn genau bekannt ist, welche Merkmale transformiert werden sollen, kann deren ID mit *„useOnInstance“* bestimmt werden. Mit *„condition“* können über Formeln Bedingungen definiert werden, die für die ausgewählten Merkmale gelten sollen. Dabei ist die Bedingung erfüllt, wenn das Ergebnis der Kondition kein leeres Element und nicht „0“ ist.

So können z.B. alle Straßenabschnitte, die Autobahnen und gleichzeitig breiter als 15 Meter sind, durch folgende Kriterien ausgewählt werden:

```
<useOnClass>Road_Element</useOnClass>
<condition>= [form_of_Way].[way_Type] = Motorway</condition>
<condition>= [width] > 1500</condition>
```

Häufig besitzen Merkmale Eigenschaften, denen mehrere Werte zugeordnet sind. Frägt man in einer Formel den Wert der Eigenschaft ab, wird nach dem Zufallsprinzip ein beliebiger dieser Werte aufgerufen. Oft ist es jedoch erwünscht, für jeden einzelnen ein eigenes Objekt zu zeichnen oder einen dieser Werte explizit durch eine *„condition“* zu bestimmen. Mit *„Line“* z.B. wird jedes Fahrzeug, das sich auf der Strecke befindet, als eigenes Objekt betrachtet. Bei allen anderen Merkmalen kann mit *„allocation“* eine Eigenschaft angegeben werden, die für jeden Wert ein eigenes Objekt erzeugt. Anders als bei sonstigen Eigenschaften darf zur Kennzeichnung einer Formel kein „=“ verwendet werden. Wenn z.B. innerhalb einer Straße alle Startpunkte der enthaltenen Straßenabschnitte benötigt werden, kann dies durch folgenden Befehl geschehen:

```
<useOnClass>Road</useOnClass>
<allocation>[contains].[starts_at]</allocation>
```

Zur Erzeugung einer Grafik in SVG muss der Name des Elements mit der Eigenschaft *„elementType“* angegeben werden. Bei den Unterklassen *„SVGPath“*, *„SVGTextPath“* und *„SVGImage“* ist er standardmäßig bereits mit den Werten „path“, „textPath“ und „image“ belegt.

Die Attribute des Elements können durch eine Menge von Eigenschaften beeinflusst werden. Mit *„addId“* wird angegeben, ob die ID des Merkmals als ID des Elements hinzugefügt werden soll. Die Stilklasse kann durch die Eigenschaft *„class“* gesetzt werden. Mit *„onClick“* ist die Javascript-Aktion beschrieben, die ausgeführt werden soll, wenn auf das Element geklickt wird. Stile können direkt über *„style“* und weitere zusätzliche Attribute mit *„additionalAttributes“* definiert werden. Soll innerhalb eines Elements ein Inhalt stehen, kann dieser mit *„innerTags“* angegeben werden. Der Inhalt eines Hilfetextes wird mit *„tooltip“* bestimmt. Bis auf *„addId“* ist es möglich, bei allen diesen Werten Formeln zu verwenden.

### 4.4.3 Transformation geometrischer Merkmale

Stellt ein Merkmal eine Fläche (*„Face“*) dar, so kann zur Transformation die Klasse *„SVGAreaGraphic“* eingesetzt werden. Diese erzeugt aus der geometrischen Fläche automatisch das SVG-Attribut *„d“*, das die Werte der Fläche enthält.

Analog wird bei einer Kante (*„Edge“*) durch die Verwendung von *„SVGEdgeGraphic“* die graphische Repräsentation der Kante erzeugt, wobei der Wert von *„rotatePath“* angibt, ob die Richtung der Kante gedreht werden soll („true“) oder nicht (“false“). Wird sie gedreht, verläuft der Pfad vom Endpunkt (*„ends_at“*) zum Startpunkt (*„starts_at“*) und sonst umgekehrt. Zwei besondere Formen der *„SVGEdgeGraphic“* sind *„SVGPath“* und *„SVGTextPath“*. Während bei *„SVGPath“* zusätzlich noch die Breite des Pfades mit *„width“* angegeben werden muss, enthält *„SVGTextPath“* die zusätzliche Eigenschaft *„text“*, die es erlaubt, einen beliebigen Text entlang des Pfades zu schreiben (siehe Abbildung 2.2). Damit lassen sich z.B. Straßennamen über die Straßen schreiben.

Jeder Punkt *„Node“* kann mit *„SVGNodeGraphic“* zu einem Symbol transformiert werden. Oft ist es erwünscht, dieses ein wenig von der ursprünglichen Stelle zu verschieben. Deswegen muss die Lage und die Größe des Symbols explizit mit *„xCoord“*,

*„yCoord“*, *„width“* und *„height“* bestimmt werden, wobei auf die Position des Punkts über die Formel „=[x]“ bzw. „=[y]“ zugegriffen werden kann. Ein spezielle Art von Symbol ist ein Bild, das über *„SVGImage“* erzeugt wird und bei dem die URL des Bildes mit *„url“* angegeben werden muss.

Die Klasse *„SVGSplittedEdge“* stellt ein besonderes Transformationsmittel dar. Damit kann eine Kante *„Edge“* nämlich in eine Menge von Punkten zerlegt werden. Neben den Eigenschaften, die *„SVGNodeGraphic“* zum Darstellen der einzelnen Punkte besitzt, stehen hier noch zwei weitere Eigenschaften zur Verfügung, mit denen die Lage der Punkte entlang der Kante bestimmt wird. Die minimale Entfernung, die der erste bzw. letzte Punkt zum Anfang oder zum Ende der Kante aufweisen muss, wird mit *„minDistanceToBorderNode“* angegeben. Wenn die Kante kürzer ist als das Zweifache dieses Wertes, wird kein Punkt angezeigt. Die Entfernung zweier benachbarter Punkte entlang der Kante bestimmt der Wert von *„distanceBetweenInstances“*. Mit Hilfe dieser Transformation ist es möglich, in regelmäßigen Abständen Grafiken, wie z.B. Bilder, zur Repräsentation von Staus auf der Kante anzuzeigen.

Mittels der Transformationsontologie können bereits sehr viele Transformationen durchgeführt werden. Falls jedoch irgendeine Transformation damit nicht realisierbar ist, besteht die Möglichkeit, für diese spezielle Transformation ein Plugin zu erzeugen, das ohne weiteres in den Transformationsvorgang eingebunden werden kann (siehe Kapitel 4.7.1).

## 4.5 Transformationsalgorithmen in Java

### 4.5.1 Einlesen der Daten

Die Transformation der Daten aus OTN erfolgt in einem Javaprogramm, das zum Parsen der Ontologie die auch in Protégé verwendete OWLAPI-Bibliothek [43] benutzt. Jedoch erfordert diese durch die Verwendung von *„HashMap“* einen übermäßig großen Speicherplatzverbrauch zur Speicherung der Daten. Ca. 80% des Speicherplatzes bleiben dabei unbenutzt. Weil damit selbst bei den heutigen Arbeitsspeichergrößen große Ontologien schnell an ihre Grenzen stoßen, wird in vorliegender Arbeit die OWL-Logik der Bibliothek mit einer effizienteren Speicherung gemischt. Als sehr nützlich erweist sich dabei die Tatsache, dass sich OTN-Ontologien in zwei Bereiche aufteilen:

**Datenmodell:** Die Zusammenhänge und Klassen der Ontologie werden in einer separaten Datei gespeichert.

**Daten:** Die Daten, die mit Hilfe von Ontologien definiert werden, importieren mit *„owl:imports“* das zugehörige Datenmodell und bestehen im übrigen nur aus Instanzen.

Durch diese Trennung befinden sich die gesamten OWL-spezifischen Definitionen im Datenmodell. Jedoch besitzen die einzelnen Daten außer dem Verweis auf die ihnen zu Grunde liegenden Datentypen und den daraus resultierenden Zusammenhängen keine

weiteren Zusammenhangsinformationen. Deswegen wird das Datenmodell mit der Bibliothek importiert. Für die eigentlichen Daten ist jedoch im Rahmen dieser Arbeit ein eigener Parser geschrieben worden, der zwar die Datenstruktur möglichst speichersparend aufbaut, jedoch bei Operationen, wie bei der Bestimmung von Oberklassen, auf das zugehörige Datenmodell zugreifen kann.
Dieser Parser (*„OwlReader“*) befindet sich in den Javasourcen im Paket „owl.rdfreader“ und lädt eine Datei durch den Aufruf von

```
public OwlReader(URL url,
                 boolean loadFromCache)
```

wobei „url“ angibt, welche Datei gelesen werden soll und „loadFromCache“, ob die Datei, falls sie bereits schon einmal gelesen wurde, wieder verwendet werden kann oder neu zu laden ist. Mit

```
public Iterator      getIndividuals    ()
public RdfIndividual getObject         (String id)
public Iterator      getAllIndividuals (String className,
                                        boolean checkSuperClasses)
```

können bestimmte Individuen ermittelt werden. Dabei ist es möglich, eines, alle oder alle diejenigen, die zu einer bestimmen Klasse gehören, zurückzuliefern. Zusätzlich kann mit

```
public boolean isInstanceOf (RdfIndividual owlIndiv,
                             String        className,
                             boolean       checkSuperClasses)
```

überprüft werden, ob ein Individuum eine Instanz einer bestimmten Klasse ist. Mit „checkSuperClasses“ wird angegeben, ob es sich um eine direkte Instanz oder um eine Instanz einer Unterklasse handeln darf.

Ein Individuum wird mit *„RdfIndividual“* dargestellt. Dieses bietet Methoden an, auf seine Eigenschaften zuzugreifen. Dabei wird unterschieden, ob der Wert einer Eigenschaft ein Literal oder ein weiteres Individuum ist und ob ein (zufälliger) Wert oder alle zurückgegeben werden sollen:

```
public String        getId()
public String        getLiteral   (String    literalName)
public Iterator      getLiterals  (String    literalName)
public RdfIndividual getResource  (OwlReader ontology,
                                   String    resourceName)
public Iterator      getResources (OwlReader ontology,
                                   String    resourceName)
```

### 4.5.2 Transformation nach SVG

Mit Hilfe dieser Funktionen lassen sich alle für die Transformationen nach SVG nötigen Informationen aus den Daten ermitteln. Zu jeder Klasse aus der Transformationsontologie gibt es eine entsprechende Implementierung in Java, die dieselben Oberklassen sowie denselben Klassennamen besitzt und sich im Paket „svg.representation“ befindet. Da ein *„SVGElement“* keine Transformation beschreibt, sondern lediglich eine logische Unterteilung nach Gruppeneigenschaften darstellt, ist die zugehörige Klasse abstrakt.

Eine Transformation wird in mehreren Schritten durchgeführt: Zuerst wird die Transformationsontologie eingelesen, aus der für jedes Individuum eine Instanz der entsprechenden Klasse gebildet wird und danach die zu transformierenden OTN-Daten. Mit ihnen werden die SVG-Dateien erstellt. Dazu extrahiert man aus den OTN-Daten für jede Transformationsklasse diejenigen Individuen, die den definierten Auswahlkriterien genügen. Mit Hilfe der Bibliothek aus der *RTree Visualization Demo* [39] wird aus diesen Individuen eine R-Baum-Struktur erstellt. Anschließend wird im Dateisystem ein Verzeichnis angelegt, dessen Name die ID der Transformationsklasse ist. Innerhalb dieses Verzeichnisses wird für jeden Knoten in diesem R-Baum nach einer Namenskonvention eine eigene Datei erstellt. Der Wurzelknoten trägt immer den Namen der ID, gefolgt von der Endung „.svgz“. Alle anderen Knoten erhalten den Namen „[ID][X],[Y]-[BREITE],[HOEHE].svgz“, wobei „X“, „Y“, „BREITE“ und „HOEHE“ die Werte des „minimal umgebenden Rechtecks“ des Knotens darstellen.

Bei der Transformationsklasse „Autobahn“, heißt das Verzeichnis, das die Daten enthält, „Autobahn“, der Wurzelknoten „Autobahn.svgz“, und alle anderen Knoten haben beispielsweise die Form „Autobahn15482,11138-568,1093.svgz“.

Jede Datei ist wie in Kapitel 4.2.1 beschrieben aufgebaut, wobei der Inhalt von der Art des Knotens abhängt. Wenn ein Knoten kein Blatt ist, enthält er lediglich Elemente, die Verweise zu ihren Unterknoten darstellen und dazu dienen, diese Dateien zu laden bzw. den Bereich ein- oder auszublenden:

```
<g bBox="..." loadUrl="..." />
```

Ist ein Knoten ein Blatt, so werden aus den darin enthaltenen Individuen die zugehörigen Grafikbefehle erstellt und in der zugehörigen Datei gespeichert.

Anschließend wird eine Datei eingerichtet, mit Verweisen auf jene Dateien, die durch die Wurzelknoten der Transformationsklassen erzeugt worden sind. In der Reihenfolge, die durch die Eigenschaft *„paintingOrder“* bestimmt ist, sind diese Verweise in der Datei enthalten.

Bei der Verwendung von Modulen (siehe Kapitel 4.6.1) definieren alle diese Dateien ein Modul und liegen in einem Verzeichnis, das den Namen des Moduls besitzt. Alle Module werden wiederum in einer, die gesamte Karte enthaltenden Datei zusammengefasst. Bei der Verwendung von Daten verschiedener Server kann diese Datei wiederum Bestandteil einer größeren Karte sein. Mit dieser Struktur ist es möglich, ein globales Netzwerk aus vielen (unabhängigen) Einzelkomponenten aufzubauen.

Wenn eine Transformationsklasse als dynamisch gekennzeichnet ist, also eine Gültigkeitsdauer besitzt (siehe 4.6.1), dann werden die zugehörigen Dateien nach Ablauf dieser

Zeit neu generiert.

### 4.5.3 Hilfsmittel zur Datenanalyse

Für einige Eigenschaften sind Berechnung nötig, wie z.B. die Berechnung von Formeln und Zeitangaben. Dafür gibt es zwei Hilfsmittel, die im Paket „owl.util“ liegen.

Die Klasse *TimeEvaluator* dient zur Berechnung von Zeiten, die dem OTN-Format entsprechen. Dabei existieren folgende Funktionen:

```
public static boolean isValid       (RdfIndividual indiv,
                                     OwlReader onto);
public static boolean isValid       (RdfIndividual indiv,
                                     OwlReader onto,
                                     Date date)
public static Date evaluateStartDate(String startingDate,
                                     Date date)
public static Date evaluateEndDate  (String duration,
                                     Date startDate)
public static long evaluateDuration (String duration)
```

Mit *„isValid“* wird eine Ressource auf ihre Gültigkeit zum aktuellen oder zu einem anderen speziellen Zeitpunkt (*„date“*) hin überprüft. Dazu wird die Eigenschaft *„validaty_Period“* des angegebenen Individuums *„indiv“* in einer Ontologie *„onto“* ausgelesen und kontrolliert, ob einer der Zeitpunkte innerhalb einer dieser Werte liegt. Mit *„evaluateStartDate“* wird der erste Zeitpunkt, welcher der Zeitangabe *„startingDate“* entspricht, ab einem bestimmten Datum (*„date“*) zurückgegeben. Die Anzahl von Millisekunden, die eine Dauer (*„duration“*) beträgt, wird mit *„evaluateDuration“* bestimmt. Die Methode *„evaluateEndDate“* gibt an, an welchem Datum eine Dauer (*„duration“*) ab einem bestimmten Zeitpunkt (*„startDate“*) endet.

Formeln können mit *„OwlFormularEvaluator“* berechnet werden. Die wichtigsten Methoden sind dabei:

```
public static String evaluateExpression     (RdfIndividual indiv,
                                             OwlReader onto,
                                             String expression)
public static String evaluateExpression     (RdfIndividual indiv,
                                             OwlReader onto,
                                             String expression,
                                             HashMap allocation)
public static Collection evaluateAllocations(RdfIndividual indiv,
                                             OwlReader onto,
                                             String allocationExpression)
```

Mit *„evaluateExpression“* wird eine Formel *„expression“* für ein Individuum *„indiv“* in einer Ontologie *„onto“* berechnet. Durch die Angabe einer Belegung (*„allocation“*) kann

bestimmt werden, welchen Wert eine Eigenschaft dabei besitzen soll. Solche Belegungen sind mit Hilfe der Funktion *„evaluateAllocations“* zu ermitteln. Sie berechnet für ein Individuum *„indiv“* in einer Ontologie *„onto“* alle möglichen Belegungen für eine Eigenschaft *„allocationExpression“* und gibt diese in einer *„Collection“* zurück, welche die einzelnen Belegungen als *„HashMap“*-Objekte besitzt. Über diese Funktion wird z.B. die Eigenschaft *„allocation“* einer Transformationsklasse in Belegungen umgewandelt und dadurch aus einem Merkmal in OTN verschiedene Instanzen erzeugt.

## 4.6 Konfiguration des Frameworks

Zur Bestimmung der Daten, die nach SVG zu transformieren sind bzw. mit denen Anfragen (siehe Kapitel 4.7.2) beantwortet werden sollen, wurde eine Konfigurationsdatei geschaffen, die sowohl eine Reihe von Informationen enthält als auch die Reihenfolge der auszuführenden Aktionen.

Das Format XML ist Standard für solche Konfigurationsdateien. Deshalb wurde es auch für diese Aufgabe ausgewählt. Des Weiteren galt es, ein Schema zur Angabe der vorzunehmenden Einstellungen zu definieren. Da die verschiedenen Anwendungsgebiete des Frameworks zwar auf denselben OTN-Daten beruhen, aber verschiedene Einstellungen besitzen, stehen für die Konfigurationsdateien zwei Schemata zur Verfügung:

**OTN-SVG-Transformation.dtd** bezeichnet ein Schema zur Definition aller Einstellungen der Transformation nach SVG.

**OTN-Request.dtd** gibt das Schema zur Definition aller Einstellungen zu Anfragen an.

### 4.6.1 Konfiguration der SVG-Transformation

Die Erstellung eines SVG-Dokuments erfolgt in mehreren Schritten: Zuerst wird die Konfigurationsdatei gelesen und daraus eine grobe Dokumentenstruktur generiert. Anschließend werden der Reihe nach die Elemente dieses Dokuments abgearbeitet und die dazugehörigen Befehle ausgeführt, wobei die Ergebnisse der Befehle dieselbe Reihenfolge besitzen wie die Elemente in der Konfigurationsdatei.

Das Wurzelelement in „OTN-SVG-Transformation.dtd“ ist SVGDefinition. Es ist unterteilt in:

**DocumentName** gibt den Namen des zu generierenden SVG-Dokuments an.

**DstFolderURL** bildet den Pfad zu dem Ordner, in dem sowohl das SVG-Dokument als auch seine Unterdokumente abgelegt werden.

**OntologyDefinitionURL** enthält die Darstellungsontologie, welche rechts neben der Karte angezeigt wird.

**Module** stellt eine Sammlung von Elementen dar, die zusammengefasst werden.

Während alle Elemente bis auf *„Module“* lediglich eine Zeichenkette als Unterelement besitzen, hat ein Modul eine ganze Reihe optionaler Attribute und möglicher Unterelemente.

Folgende Attribute sind möglich:

**name** gibt den Namen an, der auf der grafischen Oberfläche angezeigt wird.

**minDetail** gibt den minimalen Wert der Detailstufe an, in der das Modul sichtbar ist.

**maxDetail** gibt den maximalen Wert der Detailstufe an, in der das Modul sichtbar ist.

**bBox** gibt das „minimal umgebende Rechteck“ des Modules im *„bBox“*-Format aus SVG an.

An Unterelementen sind erlaubt:

**Module** definiert weitere Module innerhalb des Moduls.

**Svg** schreibt den Inhalt direkt in das SVG-Dokument.

**SvgURL** enthält einen Verweis auf bereits bestehende SVG-Dokumente, die eingebunden werden sollen.

**Create** erzeugt mit Hilfe einer Transformationsontologie ein Dokument.

**Include** bindet Plugins ein.

Mit *„Svg“* können innerhalb der Konfigurationsdatei SVG-Elemente geschrieben werden, die an dieser Stelle angezeigt werden sollen.

Der Inhalt der *„SvgURL“* ist eine URL, die auf ein bestehendes (externes) SVG-Dokument verweist. Dabei sind dieselben Eigenschaften wie bei einem Modul einsetzbar. Doch besteht zusätzlich mit „updatePeriod“ noch die Möglichkeit, eine Zeitdauer in Sekunden anzugeben, nach der dieses Dokument aktualisiert werden soll.

Das wichtigste Element ist *„Create“* zur Transformation von OTN nach SVG. Dieses besteht aus den Unterelementen *„DocumentName“*, *„TransformationRuleURL“* und einer beliebigen Anzahl an *„ResourceURL“*. *„DocumentName“* gibt den Namen des Dokuments an, das erzeugt werden soll und *„TransformationRuleURL“* sowie *„ResourceURL“* die URL der Transformationsontologie und der zu transformierenden OTN-Dateien. An Eigenschaften weist *„Create“* alle diejenigen auf, die auch *„SvgURL“* besitzt und mit *„validityDuration“* zusätzlich noch eine Zeitdauer im OTN-Format, die angibt, nach welcher Zeit das Dokument neu erzeugt werden soll. Durch die Angabe von „y9999“ wird z.B. erreicht, dass das Dokument alle 9999 Jahre aktualisiert wird, und es, insofern es zum gegenwärtigen Zeitpunkt noch nicht existiert, einmal erzeugt und nie mehr geändert wird, was bei statischen Daten durchaus nützlich ist.

Für die Fälle, in denen die Transformationsontologie nicht ausreicht, wird eine Schnittstelle *„svg.transformation.SvgTransformationInterface“* zur Verfügung gestellt, mit der weitere Transformationen ausführbar sind. Eine solche Transformation kann dann durch

ein Plugin durchgeführt werden, das mit dem Element *„Include“* eingebunden wird. Dabei ist lediglich die Eigenschaft *„class“* vorgeschrieben, die den Namen der einzubindenden Klasse enthält (siehe Kapitel 4.7.1). Alle anderen Eigenschaften bzw. Unterelemente hängen von dem aufgerufenen Plugin ab.

### 4.6.2 Konfiguration des Servers zur Verwaltung von Anfragen

Das Wurzelelement in „OTN-Request.dtd“ ist „ServerDefinition“. Es enthält lediglich eine Reihe von Plugins, die mit *„Include“* und *„class“* analog zur Konfiguration der Visualisierung eingebunden werden. Dabei müssen diese Plugins eine Implementierung der abstrakten Klasse *„owl.request.AbstractOwlRequest“* darstellen.

## 4.7 Plugins

Wichtiges Kriterium eines Frameworks ist die Möglichkeit, Komponenten zu erzeugen, die in das Framework eingebunden werden können. Da sich das Framework in die Bereiche Transformation und Abfrage aufteilt, gibt es dafür zwei Arten von Komponenten: Mit *„SvgTransformationInterface“* besteht eine Schnittstelle zur Transformation und mit *„AbstractOwlRequest“* können Komponenten erzeugt werden, mit denen Anfragen auf die geographischen Daten gestellt werden, und die über das Datenmodell Ergebnisse folgern können („Geospatial Reasoning“).

### 4.7.1 SvgTransformationInterface

Jede Klasse, die eine Transformation durchführt, ist eine Implementierung der Schnittstelle *„SvgTransformationInterface“*, über die auch weitere Transformationsmechanismen definiert werden können (siehe Kapitel 4.7.1). Diese Schnittstelle ist folgendermaßen aufgebaut:

```
public interface SvgTransformationInterface
{
  public String    createSVGGraphic(boolean updateDynamicsOnly,
                                    String indent)
  public boolean   isDynamic();
  public Rectangle getBoundingBox();
}
```

Die wichtigste Methode ist „createSVGGraphic“, die aufgerufen wird, wenn das SVG-Dokument erzeugt werden soll. Die Eigenschaft *„updateDynamicsOnly“* gibt an, ob alle Dateien generiert oder nur, falls nötig, Aktualisierungen durchzuführen sind. Zu Beginn der Transformation besitzt die Eigenschaft den Wert „true“, während der sich anschließenden Aktualisierungsphasen jedoch den Wert „false“. Als Ergebnis erwartet die Funktion SVG-Elemente, die in das zu generierende Dokument eingebunden werden sollen. Dies dürfen zwar sowohl Verweise als auch Grafikbefehle sein. In der Regel sollten jedoch Verweise verwendet werden. Um eine lesbare Formatierung des Dokuments zu erhalten,

sind die Leerzeichen, die jeder Zeile vorausgestellt werden sollten, mit der Eigenschaft „indent“ mitzugeben.

Zusätzlich zur direkten Generierung von SVG-Elementen kann diese Methode auch noch dazu genutzt werden, Daten aus einem anderen Datenformat nach OTN zu konvertieren. Da die Konfigurationsdatei immer in einer festen Reihenfolge durchlaufen wird, ist es möglich, ein Plugin zu definieren, das OTN-Daten erzeugt und keine SVG-Befehle zurückliefert. Anschließend kann dann in der Konfigurationsdatei eine Transformation mit *„Create“* definiert werden, die diese Datei nach SVG überträgt.

Ein Beispiel dafür ist das Modul „Munich Weather“, das ein zufälliges Wetter mit Hilfe des „FakeWeather“-Plugins erzeugt und in der OTN-Ontologie „FakeWeather.owl“ speichert. Diese wird anschließend mit der Transformationsontologie „WeatherRules.owl“ in ein SVG-Dokument umgewandelt:

```
<Module hidden="true" name='Munich Weather'>
  <Include fakeNr="1"
           class="plugins.transformations.FakeWeather"
           bBox="0 0 25848 20322"
           dstFile="../Ontologie/data/FakeWeather.owl" />
  <Create validityDuration="s1" bBox="0 0 25848 20322">
    <DocumentName>MunichWeather</DocumentName>
    <TransformationRuleURL>
      ../Ontologie/transformations/WeatherRules.owl
    </TransformationRuleURL>
    <ResourceURL>../Ontologie/data/FakeWeather.owl</ResourceURL>
  </Create>
</Module>
```

### 4.7.2 AbstractOwlRequest

Es ist möglich, an einen Server Anfragen zu stellen. Dazu müssen in der zugehörigen Konfigurationsdatei Plugins definiert sein, die von *„AbstractOwlRequest“* erben:

```
public abstract class AbstractOwlRequest
{
  public abstract String[]    getHandledRequests();
  public abstract OwlResponse handleAction (String requestName,
                                            Map parameters);
  public abstract String      getMapHTML   ();
  public          String      getRequestURL(String requestName,
                                            Map parameters);
}
```

Jedes Plugin unterstützt eine Menge von Anfragen, deren Namen mit *„getHandledRequests“* aufgerufen werden können. Erhält nun der Server eine Anfrage mit einem der im Plugin definierten Namen, dann wird die Funktion *„handleAction“* aufgerufen

und es werden ihr der Name der Anfrage (*„requestName“*) sowie weitere Parameter (*„parameters“*) übergeben. Als Ergebnis wird ein *„OwlResponse“* Objekt erwartet, das Informationen über den Aufbau der Antwortwebseite enthält.

Wenn eine Karte über einen Webserver angezeigt wird, ist es möglich, unterhalb der Karte eine grafische Oberfläche zu integrieren, über die Abfragen an den Server geschickt werden können. Das Aussehen dieser Oberfläche kann mit *„getMapHTML“* bestimmt werden. Dabei bietet sich die bereits definierte Methode *„getRequestURL“* an, um die URL zu erzeugen, die ihrerseits eine Anfrage mit dem Namen *„requestName“* und den Parametern *„parameters“* an den Server schickt.

Das Antwortobjekt *„OwlResponse“* enthält bisher drei Variablen:

```
public class OwlResponse
{
  public String HTML_CONTENT;
  public String JAVASCRIPT_CONTENT;

  public boolean SHOW_CONTENT_ONLY = false;
}
```

Über *„HTML_CONTENT“* kann ein HTML-Inhalt definiert werden, der angezeigt werden soll. Mit *„SHOW_CONTENT_ONLY“* wird dabei bestimmt, ob automatisch auch ein HTML-Gerüst hinzugefügt werden soll oder nicht. Bei Verwendung der Anfrage über die mit *„getMapHTML“* generierten Oberfläche wird in der Regel die Antwort ebenfalls auf dieser Oberfläche ausgegeben und deswegen in ein bestehendes Dokument integriert. Daher benötigt sie kein HTML-Gerüst. Wenn jedoch eine Anfrage über eine eigene Webseite gestellt wird, soll eine komplette HTML-Seite angezeigt werden.

Zusätzlich zum HTML-Inhalt kann sich eine Antwort auch aus Javascript-Befehlen zusammensetzen. Falls diese innerhalb des *„HTML_CONTENT“* stehen, werden sie auf der HTML-Seite ausgeführt. Oft ist es jedoch erwünscht, direkt die SVG-Grafik mit Javascript zu manipulieren. Dafür können mit *„JAVASCRIPT_CONTENT“* Befehle angegeben werden, die direkt im SVG-Dokument ausgeführt werden sollen.

Über diese Schnittstellen können also sowohl beliebige Transformationen durchgeführt als auch jede Art von Anfragen definiert und beantwortet werden, so dass mit diesem Framework ein Großteil der Anforderungen an einen Routenplaner erfüllt werden kann, wenn nur entsprechende Plugins geschrieben werden.

# 5 Zusammenfassung und Ausblick

GDF ist eine sehr vielseitige Sprache zur Beschreibung von Straßennetzen und zur Visualisierung geographischer Merkmale. Lediglich bei der Art der Datenspeicherung und der Modellierung des öffentlichen Verkehrs weist sie einige Mängel auf. Diese Schwachpunkte werden durch die Integration in OTN und deren entsprechenden Erweiterung beseitigt. Aus der Erweiterung des Datenmodells von GDF und der Verwendung von OWL als Speichermedium ergibt sich eine Sprache, die eine detaillierte Modellierung des Straßen- und öffentlichen Verkehrsnetzes bietet. Sie ist zudem gut verständlich und kann die Zusammenhänge zwischen einzelnen Merkmalen präzise beschreiben. Darüber hinaus ist OTN auf Grund seiner Struktur leicht erweiterbar. Das bedeutet, dass bei fehlenden Merkmalen oder Zusammenhängen jederzeit problemlos Ergänzungen möglich sind.

Bei der Visualisierung mit SVG wurde im Hinblick auf eine einfache Darstellung komplexer Grafiken, ein Standard gewählt, der sich zwar bisher noch nicht überall etabliert hat, aber immer mehr Verbreitung findet. Es hat sich gezeigt, dass hier mit ein paar Tricks selbst große Datenmengen zügig verarbeitet werden können. Bisher wurde jedoch noch nicht erprobt, wann auch hier die Grenzen der Leistungsfähigkeit erreicht sind oder ob andere Probleme den Gebrauch als Visualisierungsmedium erschweren werden. Die große Schwierigkeit bei der Verwendung von SVG ist, dass die Implementierung zum Teil stark vom verwendeten Plugin abhängt und daher eine Verbreitung komplexerer SVG-Anwendungen bislang nicht auf allen Betriebssystemen möglich ist. Jedoch kann durch die Verwendung des am meisten verbreiteten Plugins von Adobe, das auf immer mehr Plattformen implementiert wird, in Zukunft ein einheitliches System zur Darstellung von SVG genutzt werden.

Das Framework zur Transformation von OTN nach SVG ist ein umfassendes Werkzeug zum Erzeugen von Karten. Durch die damit gewonnene Fähigkeiten, über Plugins kompliziertere Transformationen durchführen zu können, sind alle nur denkbaren Visualisierungen realisierbar. Jedoch stammen bislang alle Daten nur von einem einzigen Server. Es ist zwar bereits möglich, Daten aus anderen Quellen einzubinden, doch führt dies evtl. zu Problemen u.a. bei der Benennung der Dateien oder der Erstellung einer allgemeinen „Darstellungsontologie“, da in der Regel die Daten ohne das Wissen um die Existenz anderer Daten erzeugt werden. Hier muss noch ein gangbarer Weg gefunden werden, Informationen über weitere vorhandene Datenquellen zu sammeln und zu verwalten.

Das Framework zur Anfragebearbeitung ist in der Anfangsphase der Entstehung, da bisher keine komplexen Anwendungsfälle analysiert und auf ihre Machbarkeit überprüft worden sind. Zudem gibt es nur begrenzte Möglichkeiten, die Daten nach Informationen zu durchsuchen und daraus Schlüsse zu ziehen. Dazu müssen sowohl die Verwendung von bestehenden „Reasonern“ wie Racer als auch die Erstellung eines auf OTN zugeschnittenen Anfragebearbeitungssystems untersucht werden. Jedoch ist das bestehende

System schon jetzt für leichtere Aufgaben einsetzbar und bietet ein gute Grundlage für weitere Entwicklungen auf diesem Gebiet.

# A Beschreibung der verwendeten Dateien

In den vorhergehenden Kapiteln wurden der Aufbau und die Funktionen von einzelnen in der Diplomarbeit verwendeten Dateien behandelt. Dieses Kapitel bietet nun einen Überblick über all diese Dateien und deren gegenseitige Verknüpfungen und Zusammenhänge. Generell sind sie in drei verschiedene Gebiete eingeteilt:

**OTN-Daten:** Diese beinhalten sowohl die Datenmodelle von OTN und der Transformationsontologie als auch konkrete Daten aus dem Münchner Verkehrsnetz sowie explizite Transformationsregeln.

**Framework:** Das Framework wurde in Java geschrieben und enthält Klassen und Schnittstellen, die Transformationen und Abfragen durchführen. Dazu gehört neben den dazu benötigten Konfigurationsdateien auch die exemplarische Unterstützung eines Webservers.

**Anzeige in SVG:** Neben den generierten SVG-Dokumenten gibt es eine Reihe von Javascript-Programmen, welche die dynamischen Funktionen durchführen.

## A.1 Dateien von OTN und der Transformationsontologie

Alle Dateien, die OTN oder die Transformationsontologie betreffen, befinden sich im Ordner **Ontologie**. Dieser ist unterteilt in vier Unterordner:

- Das Verzeichnis **model** enthält die Datenmodelle von OTN (OTN.owl) und von der Transformationsontologie (OTN2SVG.owl).
- In **data** befinden sich Beispieldaten der Stadt München in OTN. Die ursprünglichen Daten wurden freundlicherweise durch das Baureferat München [35] für die vorliegende Arbeit zur Verfügung gestellt. Jedoch sind diese rechtlich geschützt und somit lediglich für Forschungszwecke nach Genehmigung zugänglich. Sie unterteilen sich in:

  **MunichNetwork.owl,** das die Daten aller Straßen in München beinhaltet.

  **MunichExtras.owl,** welches das MunichNetwork.owl um von Hand ergänzte Daten wie z.B. Gebäude, Parks und Dienste erweitert.

  **MunichTMC.owl,** das MunichNetwork.owl um TMC-Codes einiger Straßen in München erweitert.

  **FakeWeather.owl,** das durch das Plugin FakeWeather erzeugt wird und Wetterdaten enthält.

Die folgenden Dateien modellieren den öffentlichen Verkehr in München (MVV [36]). Dabei ist jede Linie in einer eigenen Datei gespeichert. Das „XX“ stellt die Nummer der entsprechenden Linie dar. Andere öffentliche Verkehrsmittel können analog abgebildet werden:

**UXX.owl** enthält mit U1.owl, U2.owl, U3.owl, U4.owl, U5.owl und U6.owl alle wichtigen U-Bahnlinien des MVV.

**SXX.owl** bildet alle wichtigen S-Bahnlinien des MVV ab.

**BusXX.owl** modelliert bisher lediglich exemplarisch die Buslinie Bus54.owl.

**TramXX.owl** bildet bisher als Beispiel nur die Trambahnlinie 17 ab.

- Alle definierten Transformationsregeln befinden sich im Ordner **transformation**:

  **SatelliteRules.owl** platziert ein statisches Satellitenbild auf der Karte, das mit dem BayernViewer der Bayerischen Vermessungsverwaltung [34] erstellt wurde.

  **BackgroundRules.owl** erzeugt die Grafiken, die im Hintergrund platziert werden sollen, wie z.B. Parks.

  **StreetRules.owl** gibt die Regeln zur Transformation von Straßen und des öffentlichen Verkehrs von OTN nach SVG an.

  **ForegroundRules.owl** erzeugt mit seinen Anweisungen Objekte, die über die Straßen gezeichnet werden sollen, z.B. die Symbole für Dienste, wie Parkplätze oder Museen.

  **TMCEvents.owl** ist eine Beispieldatei zur Anzeige von TMC-Events, wie Staus oder Straßensperrungen. Sie soll im Rahmen des Praktikums „Streamed Traffic and Travel Information“ [46] dynamisch aus TMC Daten erstellt werden.

  **WeatherRules.owl** zeigt Wetterdaten auf der Karte an.

  **Ontologies.owl** beinhaltet die Darstellungsontologie (siehe Kapitel 4.2).

- Der Ordner **config** enthält die Konfigurationsdateien und wird daher im folgenden Kapitel behandelt.

## A.2 Dateien des Frameworks

Das Framework ist ein wichtiger Bestandteil der Arbeit. Die dafür benötigten Dateien sind dreigeteilt:

- Die Dateien, die zur Konfiguration des Frameworks benötigt werden, befinden sich im Ordner **Ontologie/config** und unterteilen sich wiederum folgendermaßen:

  **OTN-SVG-Transformation.dtd** stellt ein Schema zur Konfiguration der Visualisierung dar.

**defaultConfig.xml** ist die Standard Konfigurationsdatei der Visualisierung.

**OTN-Request.dtd** beschreibt das Schema zur Definition von Anfragen.

**requestConfig.xml** stellt ein Beispiel zur Verwendung des Anfrageschemas dar, das die Straßensuche als bisher einzige durchführbare Anfrage definiert.

- Eine Reihe von Programmen hängt vom Betriebssystem ab. Da die Visualisierung bisher lediglich auf dem Internet Explorer unter Windows verfügbar ist, wurde als Beispielimplementation eines Webservers ebenfalls Windows gewählt. Alle dafür benötigten Dateien befinden sich im Ordner **windows**. Dabei enthält der Unterordner **windows/tomcat** einen Apache Tomcat Webserver [45]. Das eigentliche Tomcat-Projekt, welches das Framework definiert, ist plattformunabhängig und befindet sich im Ordner **Owl2Svg/webapps/OTNServer**. Neben den Java-Klassen und -Bibliotheken (in **WEB-INF**) enthält es alle Daten, die zur Visualisierung in SVG nötig sind und ist im Kapitel A.3 genauer behandelt.

  Da der Tomcat Webserver auch für eine Reihe anderer Betriebssysteme verfügbar ist, kann das Framework ebenfalls auf anderen Betriebssystemen verwendet werden.

  Neben dem Ordner, der den Webserver enthält, existiert mit **windows/bin** eine Reihe von nützlichen Windows-Skripten:

  **FirstRun.bat** muss zur Installation aufgerufen werden. Dieses Skript kompiliert Javasourcen. Da OWL bisher keine relativen Pfade unterstützt, sind alle Pfade von importierten (*owl:import*) Dateien in den Daten absolut angegeben. Die Korrektur dieser absoluten Pfade wird vom Skript ebenfalls automatisch durchgeführt.

  **StartServer.bat** startet den Webserver.

  **StopServer.bat** beendet den Webserver.

  **CreateJavadoc.bat** erzeugt aus den Javasourcen Javadoc-Dateien [42].

- Im Ordner **Owl2Svg/src** befinden sich alle verwendeten Javasourcen. Im folgenden Kapitel werden sie genauer beschrieben.

### A.2.1 Beschreibung der Javasourcen

Die Javasourcen sind in vier verschiedene Bereiche unterteilt:

**owl** enthält alle Dateien, die sowohl zum Einlesen von OWL-Dateien als auch für Anfragen auf den OWL-Dateien benötigt werden.

**svg** beinhaltet alle Dateien, welche die Transformation nach SVG durchführen.

**plugins** umfassen einige Plugins, welche die Schnittstellen verwenden.

**util** enthält neben der Datei DirectoryFix.java, die absolute Dateinamen bereinigt, den Ordner spatialindex, der die Klassen von [40] zur Erzeugung eines R-Baums ergänzt.

**owl** unterteilt sich in folgende Ordner:

- Der Ordner **owl/rdfreader** enthält Klassen, die sowohl RDF- als auch OWL-Dateien parsen können. Dazu wird die OWL-Bibliothek [43] verwendet (siehe Kapitel 4.5.1):

  **RdfReader.java** parst eine RDF-Datei.

  **OwlReader.java** parst eine OWL-Datei und bietet einige Operationen zur Analyse der Daten mit Hilfe zu Grunde liegender Ontologien an.

  **OwlUtils.java** stellt Hilfsfunktionen dar, die vom OwlReader verwendet werden.

  **RdfIndividual.java** repräsentiert eine Ressource innerhalb der Daten.

- Das Verzeichnis **owl/request** enthält die Schnittstelle für Anfragen sowie die dafür benötigten Dateien:

  **AbstractOwlRequest.java** bildet die Schnittstelle für Anfragen (siehe Kapitel 4.7.2).

  **OwlRequestHandler.java** ist eine Klasse zum Einlesen der Konfiguration und zur Initialisierung der entsprechenden Abfrage-Plugins.

  **OwlResponse.java** dient als Objekt zum Speichern der Antwort auf eine Anfrage.

- Für die Analyse und Erzeugung von OWL-Dokumenten werden in **owl/util** eine Reihe von Hilfsmitteln zur Verfügung gestellt:

  **OwlDocument.java** ist eine Klasse zur Erleichterung der Erzeugung eines OWL-Dokuments.

  **OwlFormularEvaluator.java** wertet Formeln aus (siehe Kapitel 4.5.3).

  **TimeEvaluator.java** wertet Zeitangaben aus (siehe Kapitel 4.5.3).

- Hilfsmittel zur Analyse von OTN-Daten befinden sich in **owl/otn**:

  **OTNEdgeUtils.java** bilden Hilfsfunktionen, die für die Auswertung von Kanten in OTN benötigt werden.

  **PositionInformation.java** erzeugt die aktuellen Positionen öffentlicher Verkehrsmittel mit Hilfe eines Fahrplans.

  **RoadFunctions.java** enthält Hilfsmittel zum Arbeiten mit Straßen.

**svg** wird wie folgt unterteilt:

- Die Schnittstelle für die Visualisierung und für die Transformation benötigter Dateien befinden sich in **svg/transformation**:

  **SvgTransformationInterface.java** ist die Schnittstelle für Visualisierungen (siehe Kapitel 4.7.1).

**ExtendedSvgCreator.java** liest die Konfigurationsdatei zur Visualisierung ein und erzeugt daraus die Dokumente.

- In **svg/layer** befinden sich Komponenten zur Bearbeitung von vordefinierten Elementen in der Konfigurationsdatei (siehe Kapitel 4.6):

  **SvgContent.java** zeigt mit <Svg> vordefinierten Inhalt an.

  **SvgModule.java** verarbeitet die Elemente <Module> und <Include>.

  **SvgReference.java** erzeugt ein neues SVG-Dokument mit <Create> oder referenziert eine Datei mit <SvgURL>.

- Das Verzeichnis **svg/representation** enthält die Implementierungen der Klassen der Transformationsontologie, wobei der Name der Java-Klasse mit dem Namen der Klasse in der Transformationsontologie übereinstimmt (z.B. SvgElement.java und SvgSplittedEdge.java).

- Hilfsmittel, die bei der Transformation verwendet werden, liegen in **svg/util**:

  **IndividuumInformation.java** dient als Objekt zur Speicherung von Informationen über die zu transformierende Resource.

  **SVGDocument.java** erzeugt das Gerüst für ein SVG-Dokument.

  **Utilities.java** stellen allgemeine für die Transformation verwendete Hilfsmittel.

  **XMLUtils.java** sind Hilfsmittel zur Analyse der XML-Konfigurationsdatei.

**plugins** ist nach der Art der verwendeten Schnittstelle unterteilt:

- Zur Visualisierung gibt es im Ordner **plugins/transformation** bisher folgende Plugins:

  **FakeWeather.java** erzeugt ein zufälliges Wetter aus einer Reihe von Parametern.

  **PublicNetworkCreator.java** liest eine Textdatei, die den öffentlichen Verkehr stark vereinfacht darstellt und erzeugt daraus eine OTN-Datei. Einige dieser Textdateien zur Erzeugung des MVV-Verkehrsnetzes befinden sich im Ordner **Owl2Svg/webapps/OTNServer/tram**.

  **TMCLocationConverter.java** liest eine Excel-Datei, die für TMC-Codes Orte beschreibt und versucht, die entsprechenden Stellen in den OTN-Daten zu finden und um den zugehörigen TMC-Code zu erweitern. Jedoch können damit bisher nur wenige Orte zugeordnet werden.

- Bisher gibt es in **plugins/request** lediglich eine Anfrage:

  **OTNStreetNameFinder.java** ist ein Plugin zum Suchen von Straßen.

## A.3 Dateien zur Anzeige in SVG

Alle Dokumente, die zum Anzeigen in einem SVG-fähigen Browser benötigt werden, befinden sich im Verzeichnis **Owl2Svg/webapps/OTNServer**. Neben den bereits erwähnten Verzeichnissen **WEB-INF** und **tram** gibt es noch folgende Verzeichnisse und Dateien:

**maps** beinhaltet alle SVG-Dokumente, die durch das Framework generiert werden.

**images** enthält auf der Karte verwendete Bilder.

**styles** sind CSS-Stylesheets zum Formatieren der Karte

**scripts** beinhalten Javascript-Dateien zum dynamischen Verändern der Karte.

**Demo.html** ist ein HTML-Dokument zum Anzeigen der Karte ohne Serverunterstützung.

**Demo.jsp** ist ein Standarddokument zum Anzeigen der Karte mit Serverunterstützung.

**OTN-Request.jsp** bildet eine Kommunikationsschnittstelle für Anfragen an den Server.

**MapEditor.html** stellt den Prototyp eines Karteneditors dar, der bisher jedoch nicht weiter dokumentiert ist.

**MapEditor.xml** ist die Definition der unterstützten Formate im Karteneditor.

**Pattern.svg** enthält die auf der Karte verwendeten Muster (z.B. Symbole für Schnee, einen Parkplatz oder ein Museum).

Die Javascript-Dateien untergliedern sich in:

**ViewBox.js** berechnet den aktuellen Sichtbarkeitsbereich nach [37] (siehe Kapitel 4.3.1).

**SVGLoader.js** übernimmt die meisten Aufgaben der dynamischen Visualisierung und hat dabei folgende Funktionen:

- Sie erweitert das <g> Element von SVG um eine Reihe dynamischer Eigenschaften (siehe Kapitel 4.2.4). Dafür wird der SVG-DOM analysiert und es werden bei Bedarf neue oder aktualisierte Dateien automatisch nachgeladen.
- Sie berechnet immer den jeweils aktuell sichtbaren Bereich und speichert ihn zur weiteren Verwendung in der Variablen CURRENT_VIEWBOX.
- Sie erweitert den Zoom auf beliebige Vergrößerung und Verkleinerung (siehe Kapitel 4.3.2).
- Sie bietet mit `zoomToRect(x, y, width, height)` eine Methode an, um zu einem bestimmten Bereich zu zoomen.

**tree.js** erzeugt eine dynamische Baumstruktur, die z.B. als Darstellungsontologie-Baum verwendet werden kann.

**toolbar.js** sind Methoden zum Anzeigen eines Hilfetextes (Tooltip) auf der Karte.

**MarkingManager.js** stellen Hilfsmittel zum Markieren von Objekten dar.

**ONTOLOGY.js** sind dynamisch generierte Javascriptbefehle zum Anzeigen der Darstellungsontologie.

**moduleManager.js** erzeugt den Baum auf der rechten Seite aus den Modulen und der Darstellungsontologie.

**mapEditor.js** enthält Javascriptbefehle, die im Karteneditor verwendet werden.

**measurement.js** misst Koordinaten und Objekte im Karteneditor.

# B Auszüge aus dem Sourcecode

## B.1 OTN-SVG-Transformation.dtd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT SVGDefinition (DocumentName, DstFolderURL,
                         OntologyDefinitionURL, Module) >
<!ELEMENT DocumentName (#PCDATA) >
<!ELEMENT DstFolderURL (#PCDATA) >
<!ELEMENT OntologyDefinitionURL (#PCDATA) >

<!ELEMENT Module ((Module | SvgURL | Create | Svg | Include)*) >
<!ATTLIST Module name CDATA #IMPLIED
                 minDetail CDATA #IMPLIED
                 maxDetail CDATA #IMPLIED
                 bBox CDATA #IMPLIED>

<!ELEMENT Include ANY>
<!ATTLIST Include class CDATA #REQUIRED>

<!ELEMENT Svg (#PCDATA)>

<!ELEMENT SvgURL (#PCDATA) >
<!ATTLIST SvgURL name CDATA #IMPLIED
                 minDetail CDATA #IMPLIED
                 maxDetail CDATA #IMPLIED
                 bBox CDATA #IMPLIED
                 updatePeriod CDATA #IMPLIED
>

<!ELEMENT Create (DocumentName, TransformationRuleURL, (ResourceURL)+) >
<!ATTLIST Create  name CDATA #IMPLIED
                  bBox CDATA #IMPLIED
                  minDetail CDATA #IMPLIED
                  maxDetail CDATA #IMPLIED
                  validityDuration CDATA #IMPLIED
                  updatePeriod CDATA #IMPLIED>
<!ELEMENT TransformationRuleURL (#PCDATA) >
<!ELEMENT ResourceURL (#PCDATA) >
```

## B.2 defaultConfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE OTN-SVG-Transformation
          SYSTEM "OTN-SVG-Transformation.dtd">
<SVGDefinition>
  <DocumentName>Demo</DocumentName>
  <DstFolderURL>../html/maps</DstFolderURL>
  <OntologyDefinitionURL>
   ../Ontologie/transformations/Ontologies.owl
  </OntologyDefinitionURL>
  <Module>
    <SvgURL>Pattern.svg</SvgURL>
    <Module hidden="true" name='Satelite'>
      <Create validityDuration="y9999" bBox="0 0 25848 20322">
        <DocumentName>Satellite</DocumentName>
        <TransformationRuleURL>
          ../Ontologie/transformations/SatelliteRules.owl
        </TransformationRuleURL>
      </Create>
    </Module>
    <Module name='Munich Map'>
      <Create validityDuration="y9999">
        <DocumentName>MunichBackground</DocumentName>
        <TransformationRuleURL>
          ../Ontologie/transformations/BackgroundRules.owl
        </TransformationRuleURL>
        <ResourceURL>
          ../Ontologie/data/MunichExtras.owl
        </ResourceURL>
      </Create>
      <Create validityDuration="y9999">
        <DocumentName>MunichStreets</DocumentName>
        <TransformationRuleURL>
          ../Ontologie/transformations/StreetRules.owl
        </TransformationRuleURL>
        <ResourceURL>
          ../Ontologie/data/MunichExtras.owl
        </ResourceURL>
      </Create>
      <Create validityDuration="y9999" >
        <DocumentName>MunichForeground</DocumentName>
        <TransformationRuleURL>
          ../Ontologie/transformations/ForegroundRules.owl
```

```
    </TransformationRuleURL>
    <ResourceURL>
      ../Ontologie/data/MunichTMC.owl
    </ResourceURL>
  </Create>
  <Create validityDuration="y9999" >
    <DocumentName>TMCEvents</DocumentName>
    <TransformationRuleURL>
      ../Ontologie/transformations/TMCEvents.owl
    </TransformationRuleURL>
    <ResourceURL>
      ../Ontologie/data/MunichTMC.owl
    </ResourceURL>
  </Create>
</Module>
<Module name='MVV'>
  <Include class="plugins.transformation.PublicNetworkCreator"
           validityDuration="y9999"
           srcFile="webapps/OTNServer/tram/U1src.txt"
           dstFile="../Ontologie/data/U1.owl"
           ontology="../Ontologie/data/MunichExtras.owl"
           type="Underground" />
  ...
  <Include class="plugins.transformation.PublicNetworkCreator"
           validityDuration="y9999"
           srcFile="webapps/OTNServer/tram/Tram17src.txt"
           dstFile="../Ontologie/data/Tram17.owl"
           ontology="../Ontologie/data/MunichExtras.owl"
           type="Light_Rail" />
  <Include class="plugins.transformation.PublicNetworkCreator"
           validityDuration="y9999"
           srcFile="webapps/OTNServer/tram/Bus54src.txt"
           dstFile="../Ontologie/data/Bus54.owl"
           ontology="../Ontologie/data/MunichExtras.owl"
           type="Bus" />
  <Create>
    <DocumentName>MVV</DocumentName>
    <TransformationRuleURL>
      ../Ontologie/transformations/StreetRules.owl
    </TransformationRuleURL>
    <ResourceURL>../Ontologie/data/U1.owl</ResourceURL>
    ...
    <ResourceURL>../Ontologie/data/Bus54.owl</ResourceURL>
    <ResourceURL>../Ontologie/data/Tram17.owl</ResourceURL>
```

```
      </Create>
    </Module>
    <Module hidden="true" name='Munich Weather'>
      <Include fakeNr="1"
               class="plugins.transformation.FakeWeather"
               bBox="0 0 25848 20322"
               dstFile="../Ontologie/data/FakeWeather.owl" />
      <Create validityDuration="s30" bBox="0 0 25848 20322">
        <DocumentName>MunichWeather</DocumentName>
        <TransformationRuleURL>
          ../Ontologie/transformations/WeatherRules.owl
        </TransformationRuleURL>
        <ResourceURL>
          ../Ontologie/data/FakeWeather.owl
        </ResourceURL>
      </Create>
    </Module>
  </Module>
</SVGDefinition>
```

## B.3 OTN-Request.dtd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT ServerDefinition (Include*) >

<!ELEMENT Include ANY>
<!ATTLIST Include class CDATA #REQUIRED>
```

## B.4 requestConfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE ServerDefinition SYSTEM "OTN-Request.dtd">
<ServerDefinition>
  <Include class="plugins.request.OTNStreetNameFinder"
           ontology="../Ontologie/data/MunichExtras.owl" />
</ServerDefinition>
```

## B.5 Demo.svgz

```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet href="styles/network.css" type="text/css"?>
<?xml-stylesheet href="styles/strokeWidths.css" type="text/css"?>
<!-- file:Demo.svgz -->
<svg preserveAspectRatio='xMinYMin meet' viewBox='-15 -15 25863 20337'>
<script type="text/javascript">
SVG_DOCUMENT = document;
 RELATIVEROOT =''
</script>
<g >
<!-- Start of LOADNODE -->
  <g  loadUrl="Pattern.svg" />
  <g  bBox="0 0 25848 20322"  module="Satellite"  hidden="true" >
    <g  bBox="0 0 25848 20322"  loadUrl="maps/Satelite/Satelite.svgz" />
  </g>
  <g  bBox="0 0 25848 20322"  module="Munich Map" >
    <g  bBox="14848 8831 3144 5782"
        loadUrl="maps/MunichBackground/MunichBackground.svgz" />
    <g  bBox="0 0 25848 20322"
        loadUrl="maps/MunichStreets/MunichStreets.svgz" />
    <g  bBox="6354 4147 12736 11066"
        loadUrl="maps/MunichForeground/MunichForeground.svgz" />
    <g  bBox="13526 4147 2757 8194"
        loadUrl="maps/TMCEvents/TMCEvents.svgz" />
  </g>
  <g  bBox="-15 -15 24932 17730"  module="MVV" >
    <g  bBox="-15 -15 24932 17730"
        loadUrl="maps/MVV/MVV.svgz" />
  </g>
  <g  bBox="0 0 25848 20322"  module="Munich Weather"  hidden="true" >
    <g  bBox="0 0 25848 20322"
        loadUrl="maps/MunichWeather/MunichWeather.svgz" />
  </g>
<!-- End of LOADNODE -->
</g>
<script type="text/javascript" xlink:href="scripts/ViewBox.js" />
<script type="text/javascript" xlink:href="scripts/SVGLoader.js" />
<script type="text/javascript" xlink:href="scripts/toolbar.js" />
</svg>
```

## B.6 SVGLoader.js

```
/**
this class manages the dynamic load/hide of parts of svg documents
the root document is recursively searched
if a child is a group element (g) following parameters are checked:
bBox: the boundingBox (x  y  width height) of this rectangle
      to hide content if bBox is not in the visible viewport

if bBox exist those attributes are checked too
minDetail: the minimum value of the Viewport the group is visible
maxDetail: the maximum value of the Viewport the group is visible
isLoaded: whether the children are already loaded
loadUrl: the URL from which this node should be loaded
  - the result must contain the [FILENAME] as
    <!-- file:[FILENAME] -->

  - eighter the complete document is the node,
    or the node must follow this convention:
      There must be a tag <!-- Start of LOADNODE -->
        before the nodes to insert
      There must be a tag <!-- End of LOADNODE -->
        after the last node to insert
updatePeriod: the time in seconds this node should be updated
*/

var CURRENT_VIEWBOX;

var SVGLOADER_STARTNODE_TAG = "<!-- Start of LOADNODE -->";
var SVGLOADER_ENDNODE_TAG   = "<!-- End of LOADNODE -->";
var SVGLOADER_FILE_TAG   = "<!-- file:";

var LOADING_COUNT = 0;
var START_LOADING;

var DYNAMIC_LEVEL = 4;

var LOADING_FILES = new Array();

var RELATIVEROOT;

var STATIC_NODES = new Array();
var ZOOM_NODES   = new Array();
var TOTAL_LOADED = 1;
```

```
var DEBUG_MODE = false;

var UPDATE_NODES = new Array();

var CHANGED_PARENTS = new Array();
var TOOLBAR;

var CURRENT_BOUNDING_BOX;

function firstLoad()
{
  window.top.parseXML = parseXML;
  window.top.checkLoading = checkLoading;
  window.top.checkNode = checkNode;
  window.top.ALLOW_ONCLICK = true;
  window.top.getURL = getURL;
  window.top.zoomToRect = zoomToRect;
  window.top.SVG_DOCUMENT = SVG_DOCUMENT;


  window.top.STREET_NAME =
         window.top.document.getElementById("StreetName");

  var svg = SVG_DOCUMENT.getDocumentElement();
  CURRENT_BOUNDING_BOX = new BoundingBox();

  svg.addEventListener("SVGZoom", perspectiveChange, false);
  svg.addEventListener("SVGScroll", perspectiveChange, false);

  checkLoading();

  notifyModuleChanged(svg);

}

function checkLoading()
{
  if (DYNAMIC_LOADER != null) window.clearTimeout(DYNAMIC_LOADER);
  DYNAMIC_LOADER = null;
  UPDATE_NODES = new Array();
  if (CURRENT_VIEWBOX == null || FORCE_RELOAD) createViewBox();
  var svg = SVG_DOCUMENT.getDocumentElement();
  for (var i= 0; i < svg.childNodes.length; i++) {
```

```
   checkNode(svg.childNodes.item(i), false);
  }
  if (DYNAMIC_LOADER == null) {
    DYNAMIC_LOADER = window.setTimeout("checkUpdate()", 3000);
  }
}

function checkUpdate()
{
  if (DYNAMIC_LOADER != null) window.clearTimeout(DYNAMIC_LOADER);
  DYNAMIC_LOADER = null;
  if (LOADING_COUNT == 0) {
    for (var loadUrl in UPDATE_NODES) {
     checkNode(UPDATE_NODES[loadUrl], true);
    }
    if (TOOLBAR && TOOLBAR.element &&
        TOOLBAR.element.getBBox().width == 0) TOOLBAR.Hide();
  }
  if (DYNAMIC_LOADER == null) {
    DYNAMIC_LOADER = window.setTimeout("checkUpdate()", 1500);
  }
}

function checkNode(node, checkOnlyUpdate)
{
  if (node.nodeName != "g") return;
  var isVisible = "true" != node.getAttribute("hidden");

  if (isVisible && !checkOnlyUpdate) {
    var bbox = node.getAttribute("bBox");
    if (bbox.length > 0) {
        bbox = bbox.split(' ');
      var x  = parseInt(bbox[0]);
      var y  = parseInt(bbox[1]);
      var x2 = x + parseInt(bbox[2]);
      var y2 = y + parseInt(bbox[3]);
      isVisible = x <= CURRENT_VIEWBOX.x2 &&
                  x2 >= CURRENT_VIEWBOX.x &&
                  y <= CURRENT_VIEWBOX.y2 &&
                  y2 >= CURRENT_VIEWBOX.y;
    }

    if (isVisible) {
      var minDetail = node.getAttribute("minDetail");
```

```
    var maxDetail = node.getAttribute("maxDetail");
    if (minDetail.length > 0 && maxDetail.length > 0) {
      minDetail = parseInt(minDetail);
      maxDetail = parseInt(maxDetail);
      isVisible = minDetail < CURRENT_VIEWBOX.detailLevel
               && maxDetail > CURRENT_VIEWBOX.detailLevel;
    }
  }
}
if (isVisible && window.top.ONTOLOGY_MANAGER) {
  var ontology = node.getAttribute("ontology");
  if (!window.top.ONTOLOGY_MANAGER.isVisible(ontology)) {
    isVisible = false;
  }
}
if (isVisible) {
  var isLoaded = node.getAttribute("isLoaded");
  var loadUrl  = node.getAttribute("loadUrl");
  var loadChildren = isLoaded != "true";
  if (!loadChildren) {
    var updatePeriod = node.getAttribute("updatePeriod");
    if (updatePeriod.length >0) {
      if (!checkOnlyUpdate) UPDATE_NODES[loadUrl] = node;
      updatePeriod = parseInt(updatePeriod);
      var lastUpdated = parseInt(node.getAttribute("lastUpdated"));
      var seconds = Math.round(new Date().getTime() / 1000);
      if (lastUpdated + updatePeriod < seconds) {
        loadChildren = true;
      }
    }
  }

  if (loadChildren) {
    var loadUrl  = node.getAttribute("loadUrl");
    if (loadUrl.length > 0) {
    loadNode(loadUrl, node);
    } else {
      loadChildren = false;
    }
  }
  if (!loadChildren) {
    for (var i= 0; i < node.childNodes.length; i++) {
     checkNode(node.childNodes.item(i), false);
    }
```

```
    }
  }
  var newStyle = isVisible ? "inline" : "none";
  if (node.getStyle().getPropertyValue("display") != newStyle) {
    node.getStyle().setProperty("display", newStyle);
  }
}

function loadNode(fileName, where) {
  LOADING_FILES[fileName] = where;
  if (LOADING_COUNT == 0) START_LOADING = (new Date()).getTime()
  LOADING_COUNT++;
  getURL(RELATIVEROOT+fileName, fileLoaded);
}

function fileLoaded(data) {
  var string = '';
  var fileName = '';
  if(data.success) {
    string = "<g>" + getLoadNode(data.content) + "</g>";
    fileName = getFileName(data.content);
    TOTAL_LOADED ++;
  } else {
    reduceLoading();
    return;
  }
  var node = parseXML(string, document);
  var parentNode = LOADING_FILES[fileName];
  if (parentNode != null) {
    parentNode.setAttribute("isLoaded", "true");
    parentNode.setAttribute("lastUpdated",
               Math.round(new Date().getTime() / 1000));
    parentNode.getStyle().setProperty("display", "none");
    while (parentNode.firstChild != null) {
      parentNode.removeChild(parentNode.firstChild);
    }
    parentNode.appendChild(node);
    var module = parentNode.getAttribute("module");
    if (module != "") CHANGED_PARENTS[module] = parentNode;
    parentNode.getStyle().setProperty("display", "inline");
    checkNode(parentNode, false);
  }
  reduceLoading();
}
```

```
function notifyModuleChanged(parentNode) {
  if (window.top.ModuleInsertionObservers) {
    for (var i= 0; i < window.top.ModuleInsertionObservers.length; i++) {
     window.top.ModuleInsertionObservers[i].notifyModuleChanged(parentNode);
    }
  }
}

function reduceLoading() {
  LOADING_COUNT--;
  if (LOADING_COUNT == 0) {
    LOADING_FILES = new Array();
    for (var changed in CHANGED_PARENTS) {
      notifyModuleChanged(CHANGED_PARENTS[changed]);
    }
    CHANGED_PARENTS = new Array();
  }
}


function getLoadNode(documentData) {
  var index = documentData.indexOf(SVGLOADER_STARTNODE_TAG);
  if (index == -1) return documentData;
  var index2 = documentData.indexOf(SVGLOADER_ENDNODE_TAG, index);
  if (index2 == -1) return documentData;
  return documentData.substring(index + SVGLOADER_STARTNODE_TAG.length,
                                index2);
}

function getFileName(documentData) {
  var index = documentData.indexOf(SVGLOADER_FILE_TAG);
  var index2 = documentData.indexOf(" -->", index);
  if (index2 == -1) return documentData;
  return documentData.substring(index + SVGLOADER_FILE_TAG.length,
                                index2);
}

var DYNAMIC_LOADER;
var FORCE_RELOAD = true;
var DIRECT_LOAD = false;

function zoomToRect(x, y, width, height)
{
```

```
  var svg = SVG_DOCUMENT.getDocumentElement();

  var ulXCorner  = CURRENT_BOUNDING_BOX.coords[0];
  var ulYCorner  = CURRENT_BOUNDING_BOX.coords[1];
  var bWidth     = CURRENT_BOUNDING_BOX.coords[2];
  var bHeight    = CURRENT_BOUNDING_BOX.coords[3];
  var pixWidth   = CURRENT_VIEWBOX.innerWidth;
  var pixHeight  = CURRENT_VIEWBOX.innerHeight;

  var DETAIL_LEVEL = 8

  var scale = Math.min(bWidth/width/DETAIL_LEVEL,
                 bHeight/height/DETAIL_LEVEL)/DYNAMIC_LEVEL;
  var translateX =(x-ulXCorner-bWidth/scale/DYNAMIC_LEVEL/2);
  var translateY =(y-ulYCorner-bHeight/scale/DYNAMIC_LEVEL/2);
  CURRENT_BOUNDING_BOX.set(scale, translateX, translateY);
}


function perspectiveChange(evt)
{
  FORCE_RELOAD = true;
  if (DYNAMIC_LOADER != null) window.clearTimeout(DYNAMIC_LOADER);
  DYNAMIC_LOADER = window.setTimeout("CURRENT_BOUNDING_BOX.update()",
                   DIRECT_LOAD?50:500);
}

BoundingBox.prototype.set = function(scale, x, y) {
  this.DYNAMIC_ZOOM_FACTOR = scale;
  this.translateX = x;
  this.translateY = y;
  this.updateView();
  FORCE_RELOAD = true;
  var scale = this.DYNAMIC_ZOOM_FACTOR * DYNAMIC_LEVEL;
  for (var i = 0; i < ZOOM_NODES.length; i++) {
    ZOOM_NODES[i].setAttribute("transform","scale("+(1/scale)+")");
  }

  checkLoading();
}


BoundingBox.prototype.update = function() {
  if (DYNAMIC_LOADER != null) {
```

```
    window.clearTimeout(DYNAMIC_LOADER);
    DYNAMIC_LOADER = null;
  }
  DIRECT_LOAD = true;
  var svg = SVG_DOCUMENT.getDocumentElement();
  var currentScale = svg.getCurrentScale();
  if (currentScale != DYNAMIC_LEVEL) {
    svg.setCurrentScale(DYNAMIC_LEVEL);
    if (currentScale == 1) {
      this.DYNAMIC_ZOOM_FACTOR = 1 / DYNAMIC_LEVEL;
      this.translateX = 0;
      this.translateY = 0;
    } else {
      this.DYNAMIC_ZOOM_FACTOR *= currentScale / DYNAMIC_LEVEL;
    }
    var scale = this.DYNAMIC_ZOOM_FACTOR * DYNAMIC_LEVEL;
    for (var i = 0; i < ZOOM_NODES.length; i++) {
      ZOOM_NODES[i].setAttribute("transform","scale("+(1/scale)+")");
    }
    this.updateView();
    return;
  }

  if (this.updateTranslation()) return;
  DIRECT_LOAD = false;
  if (DYNAMIC_LOADER == null) {
    window.setTimeout("checkLoading()", 50);
  }
}

BoundingBox.prototype.updateTranslation = function() {
// return false;
  var svg = SVG_DOCUMENT.getDocumentElement();
  var dx = svg.currentTranslate.x - this.newX;
  var dy = svg.currentTranslate.y - this.newY;
  var newTranslate = svg.getCurrentTranslate();
  if (dx != 0) {
    var translateX = -this.width * dx / getInnerWidth()
                  / (this.DYNAMIC_ZOOM_FACTOR * svg.currentScale);
    this.translateX += translateX;
    svg.currentTranslate.x = this.newX;
  } else if (dy != 0) {
    var translateY = -this.height * dy / getInnerHeight()
                 / (this.DYNAMIC_ZOOM_FACTOR * svg.currentScale);
```

```
    this.translateY += translateY;
    svg.currentTranslate.y = this.newY;
  } else {
    return false;
  }
  this.updateView();
  return true;
}

BoundingBox.prototype.updateView = function() {
  var svg = SVG_DOCUMENT.getDocumentElement();

  var newCoords = new Array();
  for (var i = 0; i < 4; i++) {
    newCoords[i] =  this.coords[i] / this.DYNAMIC_ZOOM_FACTOR;
  }
  newCoords[0] += this.translateX;
  newCoords[1] += this.translateY;
  var newBBox = newCoords.join(" ");
  if (svg.getAttribute("viewBox") != newBBox) {
    svg.setAttribute("viewBox", newBBox);
  }
}

function BoundingBox() {
  this.lastScale = 0;
  this.lastTranslateX = 0;
  this.lastTranslateY = 0;
  var svg = SVG_DOCUMENT.getDocumentElement();

  var viewBox = createViewBox();

  var viewboxes = svg.getAttribute("viewBox").split(' ');
  this.x = viewBox.x;
  this.y = viewBox.y;
  this.width = viewBox.width;
  this.height = viewBox.height;
  this.coords = new Array(this.x, this.y,
                          this.width, this.height);
  this.translateX = 0;
  this.translateY = 0;
  this.DYNAMIC_ZOOM_FACTOR = 1;

  this.newX = Math.floor(getInnerWidth() * this.x
```

```
                        / this.width * (DYNAMIC_LEVEL-1));
  this.newY = Math.floor(getInnerHeight() * this.y
                        / this.height * (DYNAMIC_LEVEL-1));

  this.DYNAMIC_ZOOM_FACTOR = 1/DYNAMIC_LEVEL;
  svg.setCurrentScale(DYNAMIC_LEVEL);
  var newCoords = new Array();
  for (var i = 0; i < 4; i++) {
    newCoords[i] =  this.coords[i] / this.DYNAMIC_ZOOM_FACTOR;
  }
  svg.setAttribute("viewBox", newCoords.join(" "));
  svg.currentTranslate.x = this.newX;
  svg.currentTranslate.y = this.newY;
}
firstLoad();
```

## B.7 FakeWeather.java

```
package plugins.transformation;

import java.awt.Point;
import java.awt.Rectangle;
import java.io.File;
import java.util.Random;
import org.w3c.dom.Node;
import owl.util.OwlDocument;
import svg.transformation.SvgTransformationInterface;
import svg.util.XMLUtils;

/**
 * FakeWeather.java
 */
public class FakeWeather implements SvgTransformationInterface
{
  private final static int WEATHER_STEPS = 10;
  private final static String[] WEATHER_TYPES =
               new String[] {"snow", "rain"};

  private String[]  weatherTypes_;
  private Point[] weatherCenters_;
  private Point[]   movingSpeed_;
  private Point[][] relativeWeatherNodes_;

  private int steps_;

  private Rectangle bBox_;
  private File dstFile_;

  private Random random_;

  public FakeWeather(Node node) throws Exception{
    steps_ = WEATHER_STEPS * 4;
    int fakeNr = XMLUtils.getIntAttribute(node, "fakeNr", 0);
    random_ = new Random(fakeNr == 0 ?
                 System.currentTimeMillis() : fakeNr);

    String dst = XMLUtils.getAttribute(node, "dstFile");
```

```
  bBox_ = XMLUtils.getBBoxAttribute(node, "bBox");
  int amount = XMLUtils.getIntAttribute(node,
              "amount", 3 + random_.nextInt(7));
  if (dst == null || bBox_ == null) {
    throw new Exception("missing attribute in "
        + this.getClass().getName()
        +" dstFile and bBox excepted");
  }
  dstFile_ = svg.util.Utilities.getRelativeFile(dst);
  if (!dstFile_.canWrite() && (!dstFile_.exists()
       &&  !dstFile_.createNewFile())) {
    throw new Exception("can't write to file "+dst);
  }
  weatherTypes_ = new String[amount];
  weatherCenters_ = new Point[amount];
  movingSpeed_ = new Point[amount];
  relativeWeatherNodes_ = new Point[amount][steps_];
  for (int i = 0; i < amount; i++) {
    createWeather(i, false);
  }
}


/**
 * Method createWeather
 *
 * @param    i                   an int
 * @param    restart             a  boolean
 *
 */
private void createWeather(int i, boolean restart) {
  weatherTypes_[i] =
     WEATHER_TYPES[random_.nextInt(WEATHER_TYPES.length)];
  int radius = bBox_.width / 20
             + random_.nextInt(Math.max(bBox_.width / 20, 100));
  int x = (1- 2*random_.nextInt(2)) * (10+random_.nextInt(70));
  int y = (1- 2*random_.nextInt(2)) * (10+random_.nextInt(70));
  if (restart) {
    if (x * movingSpeed_[i].x < 0) x = -x;
    if (y * movingSpeed_[i].y < 0) y = -y;
  }
  movingSpeed_[i] = new Point(x, y);
  for (int j = 0; j < steps_; j++) {
    int radius2 = radius + random_.nextInt(radius);
```

```
      if (restart && j % WEATHER_STEPS == 0) {
        radius2 = 2* radius;
      }
      double radians = Math.PI  *j / steps_ *2;
      x = (int) (radius2 * Math.cos(radians));
      y = (int) (radius2 * Math.sin(radians));
      relativeWeatherNodes_[i][j] = new Point(x, y);
    }
    if (restart) {
      if (weatherCenters_[i].x
          + relativeWeatherNodes_[i][0].x
          < bBox_.x) {
        weatherCenters_[i].x =  bBox_.x + bBox_.width - 1
            - relativeWeatherNodes_[i][WEATHER_STEPS *2].x;
      } else if (weatherCenters_[i].x
            +relativeWeatherNodes_[i][WEATHER_STEPS *2].x
            > bBox_.x + bBox_.width) {
        weatherCenters_[i].x =
            bBox_.x - relativeWeatherNodes_[i][0].x + 1;
      }
      if (weatherCenters_[i].y
          + relativeWeatherNodes_[i][WEATHER_STEPS].y
          < bBox_.y) {
        weatherCenters_[i].y =  bBox_.y + bBox_.height - 1
                - relativeWeatherNodes_[i][WEATHER_STEPS*3].y ;
      } else if (weatherCenters_[i].y
            + relativeWeatherNodes_[i][WEATHER_STEPS*3].y
            > bBox_.y + bBox_.height) {
        weatherCenters_[i].y =  bBox_.y
            -relativeWeatherNodes_[i][WEATHER_STEPS].y + 1;
      }
    } else {
      x = bBox_.x + random_.nextInt(bBox_.width);
      y = bBox_.y + random_.nextInt(bBox_.height);
      weatherCenters_[i] = new Point(x, y);
    }
  }

  /**
   * Method createWeatherOwl
   *
   * @param    i                   an int
   *
   * @return   a String
```

```
 *
 */
private String createWeatherOwl(int i) {
  StringBuffer content = new StringBuffer();
  content.append("<Weather rdf:ID=\"FakeWeather");
  content.append(i).append("\">\n");
  content.append(" <bounding_Edge>\n");
  content.append("  <Edge rdf:ID=\"FakeWeatherEdge");
  content..append(i).append("\">\n");
  int lastStepX = Integer.MAX_VALUE;
  int lastStepY = Integer.MAX_VALUE;
  int corners = 0;
  boolean contains = false;
  for (int j = 0; j < steps_; j++) {
    int x = weatherCenters_[i].x+relativeWeatherNodes_[i][j].x;
    int y = weatherCenters_[i].y+relativeWeatherNodes_[i][j].y;
    if (bBox_.contains(x,y))  contains = true;
    if (x < bBox_.x) x = bBox_.x;
    if (x > bBox_.x + bBox_.width) {
      x = bBox_.x + bBox_.width;
    }
    if (y < bBox_.y) y = bBox_.y;
    if (y > bBox_.y + bBox_.height) {
      y = bBox_.y + bBox_.height;
    }
    if (j == 0) {
      content.append("   <starts_at>\n");
      content.append("    <Node rdf:ID=\"StartNode");
      content.append(i).append("\">\n");
      content.append("     <x>").append(x).append("</x>\n");
      content.append("     <y>").append(y).append("</y>\n");
      content.append("    </Node>\n");
      content.append("   </starts_at>\n");
      content.append("   <intermediate_coordinates>");
    } else if (lastStepX != x || lastStepY != y) {
      if (j > 1) content.append(" ");
      content.append(x - lastStepX);
      content.append(" ").append(y - lastStepY);
    }
    lastStepX = x;
    lastStepY = y;
  }
  if (!contains) {
    createWeather(i, true);
```

```
      return createWeatherOwl(i);
    }
    content.append(" </intermediate_coordinates>\n");
    content.append("   <ends_at rdf:resource=\"#StartNode");
    content.append(i).append("\"/>\n");
    content.append("  </Edge>\n");
    content.append(" </bounding_Edge>\n");
    content.append(" <weatherType>")
    content.append(weatherTypes_[i]).append("</weatherType>\n");
    content.append("</Weather>\n");
    return content.toString();
  }

  /**
   * Method createWeather
   *
   * @return   a String
   *
   */
  private String createWeather() {
    StringBuffer weather = new StringBuffer();
    for (int i = 0; i < weatherCenters_.length; i++) {
      if (random_.nextInt(4) > 0) {
        weatherCenters_[i].x += movingSpeed_[i].x;
      }
      if (random_.nextInt(4) > 0) {
        weatherCenters_[i].y += movingSpeed_[i].y;
      }
      weather.append(createWeatherOwl(i));
    }
    return weather.toString();
  }

  /**
   *  returns whether the content is static
      or dynamic and must be updated without reload
   *  @return   whether the content is dynamic
   */
  public boolean isDynamic() {
    return true;
  }

  /**
   * Method createSVGGraphic is called to create the
```

```
   * SVGCommand to display the objects.
   * It must create all needed files to display its data.
   * If the result of the plugin is no svg data,
   * it can return null
   *
   * @param updateDynamicsOnly  a  boolean whether only the
   *                        dynamic objects should be updated
   *                        or all data must be created
   * @param indent the indent of the elements to format the
   *        resulting svg document
   *
   * @return a String containing the svg command to include
   *           the data or null, if no svgData is created
   *
   * @throws any Exception if an error has occured
   */
  public String createSVGGraphic(boolean updateDynamicsOnly,
                                 String indent) throws Exception {
    StringBuffer content = new StringBuffer();
    OwlDocument writer =
        new OwlDocument(dstFile_.getAbsolutePath(), true);
    content.append(createWeather());
    writer.write(content.toString());
    writer.endDocument();
    return null;
  }

  /**
   *  returns the boundingBox of the content or null,
   *  if the plugin has no boundingBox
   *  @return  the boundingBox of the content
   */
  public Rectangle getBoundingBox() {
    return bBox_;
  }
}
```

## B.8 OTNStreetNameFinder.java

```
package plugins.request;

import java.util.*;

import java.awt.Rectangle;
import org.w3c.dom.Node;
import owl.rdfreader.OwlReader;
import owl.rdfreader.RdfIndividual;
import owl.request.AbstractOwlRequest;
import owl.request.OwlResponse;
import svg.representation.SvgEdgeGraphic;
import svg.util.XMLUtils;

/**
 * this class is a @see owl.request.AbstractOwlRequest
 + to find streets on the map.
 */

public class OTNStreetNameFinder
             extends AbstractOwlRequest
             implements Comparator
{
  private final static String REQUEST_NAME = "FindStreet";

  private String ontology_;
  private OwlReader reader_;

  public OTNStreetNameFinder(Node node) throws Exception {
    super(node);
    ontology_ = XMLUtils.getAttribute(node, "ontology");
    if (ontology_ == null) {
      throw new Exception("missing attribute in "+
                this.getClass().getName() +" ontology excepted");
    }
  }

  /**
   * Method getHandledRequests returns the names of the requests,
```

```
 * which are defined in this module.
 *
 * @return   a String[] of all handled request names.
 * @throws   Exception
 */
public String[] getHandledRequests(){
  return new String[] {REQUEST_NAME};
}

/**
 * Method handleAction handles a request and
 * returns the result as a string,
 * containing parts of an javascript document.
 *
 * @param    requestName   the name of the request
 * @param    parameters    all additional parameters
                           which are sent with the request
 * @return   a String containing the result of the request
 */
public OwlResponse handleAction(String requestName,
                                Map parameters) {
  if (ontology_ == null) return null;
  OwlResponse response = new OwlResponse();
  if (!parameters.containsKey("name")) {
    response.HTML_CONTENT = "specify a street 'name' first!";
    return response;
  }
  String name = (String) parameters.get("name");
  if (name == null || name.length() < 3) {
    response.HTML_CONTENT = "Please specify a name first!";
    return response;
  }
  name = name.toLowerCase();
  try {
    reader_ = new OwlReader(OwlReader.createURL(ontology_));
    StringBuffer resultBuffer = new StringBuffer();
    Iterator iter = reader_.getAllIndividuals("Road_Element",
                                              false);
    SortedMap founds = new TreeMap();
    TreeSet v;
    while (iter.hasNext()) {
      RdfIndividual element = (RdfIndividual) iter.next();
      if  (element.getLiteral("official_Name")
           .toLowerCase().indexOf(name) > -1) {
```

```
        v = (TreeSet) founds.get(
              element.getLiteral("official_Name"));
        if (v == null) {
          v = new TreeSet(this);
          founds.put(element.getLiteral("official_Name"),v);
        }
        v.add(element);
      }
    }
    iter = founds.keySet().iterator();
    while (iter.hasNext()) {
      String newName = (String) iter.next();
      v = (TreeSet) founds.get(newName);
      resultBuffer.append("<br/>");
      Iterator iter2 = v.iterator();
      if (v.size() == 1)  {
        resultBuffer.append(getStreetInformation(null,
                  (RdfIndividual)iter2.next(), reader_));
      } else {
        resultBuffer.append(newName).append(":");
        for (int i = 0; iter2.hasNext(); i++) {
          resultBuffer.append(
            getStreetInformation(String.valueOf(i+1),
                                 (RdfIndividual)iter2.next(),
                                 reader_));
        }
      }
    }

    if (resultBuffer.length() == 0) {
      response.HTML_CONTENT =
          "no streets found for name " + name +"!";
    } else {
      response.HTML_CONTENT = resultBuffer.toString();
    }
  } catch (Exception e) {
  }
  response.SHOW_CONTENT_ONLY = true;
  return response;
}

/**
 * Method getStreetInformation creates a html link with a
 * given label to show a roadElement on the map
```

```
 *
 * @param    label        the label of the link
 * @param    roadElement  the roadElement which should be
 *                        shown on the map, if clicking the link
 * @param    reader       the OwlReader to which
 *                        the roadElement belongs to.
 * @return   a html link to mark and zoom to the roadElement
 * @throws   Exception if an error has occured
 */
private String getStreetInformation(String label,
                        RdfIndividual roadElement,
                        OwlReader reader) throws Exception {
  Rectangle bbox = SvgEdgeGraphic.getBoundingBox(roadElement,
                                                 reader);
  StringBuffer result = new StringBuffer();
  result.append("<a href='javascript:zoomToRect(");
  result.append(bbox.x).append(",");
  result.append(bbox.y).append(",");
  result.append(bbox.width).append(",");
  result.append(bbox.height).append(");");
  result.append("markZoomNode(\"");
  result.append(roadElement.getId()).append("\");'");
  result.append(">").append(label != null ?
      label : roadElement.getLiteral("official_Name"));
  result.append("</a>\n");
  return result.toString();
}

/**
 * Method getMapHTML creates the html code,
 * which is added to the map
 *
 * @return   a String containing html to request this module
 */
public String getMapHTML() {
  String name = getClass().getName();
  if (name.indexOf(".") > -1) {
    name = name.substring(name.lastIndexOf(".")+1);
  }
  StringBuffer html = new StringBuffer();
  html.append("\nSearch Street: <input type='text' ");
  html.append(" onKeyPress='if (window.event.keyCode==13) {");
  html.append("window.top.getURL(\"");
  html.append(getRequestURL(REQUEST_NAME));
```

```
    html.append("&name=\"+this.value, ");
    html.append(name).append("_Request)}' size='20' />\n");
    html.append("<div id='").append(name);
    html.append("_DIV'> </div>\n");
    html.append("<SCRIPT>\n function ").append(name);
    html.append("_Request(data) { \n");
    html.append("document.getElementById('").append(name);
    html.append("_DIV').innerHTML = data.content;\n}\n");
    html.append("</SCRIPT>\n");
    return html.toString();
  }


  /**
   * Method compare compares the roadElement,
   * to sort all parts of a road
   *
   * @param    o1      first roadElement
   * @param    o2      second roadElement
   *
   * @return   an int
   */
  public int compare(Object o1, Object o2) {
    if (o1 instanceof RdfIndividual
        && o2 instanceof RdfIndividual) {
      try {
        Rectangle r1 = SvgEdgeGraphic.getBoundingBox(
                        (RdfIndividual) o1, reader_);
        Rectangle r2 = SvgEdgeGraphic.getBoundingBox(
                        (RdfIndividual) o2, reader_);
        if (r1.height > r1.width * 2
           || r2.height > r2.width * 2) {
          return r1.y - r2.y;
        }else {
          return r1.x - r2.x;
        }
      } catch (Exception e) {}
    }
    return 0;
  }
}
```

# C Literaturverzeichnis


[1] URIs, URLs, and URNs: Clarifications and Recommendations 1.0, September 2001, http://www.w3.org/TR/uri-clarification/

[2] Uniform Resource Identifier (URI) SCHEMES, Januar 2004, http://www.iana.org/assignments/uri-schemes

[3] HyperText Markup Language (HTML), W3C Recommendation, April 1998, http://www.w3.org/TR/1998/REC-html40-19980424

[4] Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, Februar 2004, http://www.w3.org/TR/REC-xml/

[5] Namespaces in XML, W3C Recommendation, Januar 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114

[6] XML-Schema, W3C Recommendation, Mai 2001, http://w3.org/XML/Schema

[7] XSL Transformations (XSLT), W3C Recommendation, November 1999, http://www.w3.org/TR/xslt

[8] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, Januar 2003, http://www.w3.org/TR/SVG11/

[9] Erklärung von Béziérkurve in Wikipedia, http://de.wikipedia.org/wiki/B%C3%A9zierkurve

[10] Adobe SVG Viewer, Version 3.01, September 2003, http://www.adobe.com/svg/

[11] SVG-Tutorial von Tjorven Lauchardt, 2003, http://www.fh-wedel.de/~si/praktika/MultimediaProjekte/SVG/SVG_Tutorial_mi3794/index.htm

[12] Einführung in SVG - Hans Jürgen Ohlbach, 2003/2004, http://www.pms.ifi.lmu.de/lehre/neuemedien/slides/SVG/folien.html

[13] Cascading Style Sheets, level 2, W3C Recommendation, Mai 1998, http://www.w3.org/TR/REC-CSS2/

[14] Document Object Model (DOM) Level 2 Core Specification, November 2000, http://www.w3.org/TR/DOM-Level-2-Core/

[15] Geography Markup Language Version 3,
http://www.opengis.org/docs/02-023r4.pdf

[16] Visualisierung von GML mit XSLT und SVG, Diplomarbeit von Andreas Kupfer, September 2003,
http://www-public.tu-bs.de:8080/~y0010824/svg/index.html

[17] Visualisierung einer Straßenkarte mit Hilfe von SVG, Projektarbeit von Frank Ipfelkofer, April 2004,
http://www.pms.ifi.lmu.de/~ipfelkof/projektarbeit

[18] RDF: Resource Description Framework - Seminararbeit von Tim Furche, November 2001, http://www.furche.net/projects/2001/rdf-tutorial/

[19] Resource Description Framework (RDF), W3C Recommendation, Februar 2004, http://www.w3.org/RDF/

[20] A translation approach to portable ontologies von T.G. Gruber (1993)

[21] OWL Web Ontology Language, W3C Recommendation, Februar 2004,
http://www.w3.org/TR/owl-features/

[22] Web Ontology Language (OWL) - Übersicht über OWL von Martin Kost, Januar 2003, http://www.dbis.informatik.hu-berlin.de/lehre/WS0203/SemWeb/artikel/4/Kost_SemWeb-OWL-Ausarbeitung.pdf

[23] Protégé 2.1.2, August 2004, http://http://protege.stanford.edu

[24] Racer - ein OWL Reasoner
http://www.sts.tu-harburg.de/~r.f.moeller/racer/

[25] International Organisation for Standardization http://www.iso.org

[26] Intelligent transport systems, ISO/TC 204, http://www.iso.org/iso/en/stdsdevelopment/tc/tclist/TechnicalCommitteeDetailPage.TechnicalCommitteeDetail?COMMID=4559

[27] Japan Digital Road Map Association http://www.drm.jp/drm/e_index.htm

[28] Spatial Data Transfer Standard http://data.geocomm.com/sdts/

[29] Geographic Data Files 3.0 (GDF) Documentation, 1995,
http://www.ertico.com/links/gdf/gdfdoc/gdfdocon.htm

[30] Intelligent transport systems - Geographic Data Files 4.0 (GDF) - Overall data specification, ISO/DIS 14825, August 2002,
http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=30763&scopelist=PROGRAMME

[31] Radio Data System Forum http://www.rds.org.uk/rds98/rds98.htm

[32] Traffic Message Channel Forum http://www.tmcforum.com/

[33] Vehicle Information and Communication System
http://www.vics.or.jp/english/index.html

[34] BayernViewer der Bayerischen Vermessungsverwaltung,
http://www.bayernviewer.de/bv_beschreibung.html

[35] Baureferat München, http://www.muenchen-baut.de

[36] Münchner Verkehrs- und Tarifverbund, http://www.mvv-muenchen.de

[37] Skript zur Bestimmung der aktuellen ViewBox von Kevin Lindsey, 2002,
http://www.kevlindev.com/gui/utilities/viewbox/index.htm

[38] R-Tree Portal, Juni 2003, http://www.rtreeportal.org/

[39] RTree Visualization Demo der National Technical University of Athens, November 1999, http://www.dbnet.ece.ntua.gr/~mario/rtree/

[40] Spatial Index Library 0.44b der University of California, http://www.cs.ucr.edu/~marioh/spatialindex/

[41] Sun Java Technology, http://java.sun.com/

[42] Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/

[43] OWL-Parser in Java, http://sourceforge.net/projects/owlapi

[44] GNU zip, http://www.gzip.org/

[45] Apache Jakarta Tomcat, http://jakarta.apache.org/tomcat/index.html

[46] Practical Course „Streamed Traffic and Travel Information“,
http://www.pms.ifi.lmu.de/lehre/praktikum/traveltraffic/04ws05/