Institut für Informatik
Programmier- und Modellierungssprachen
Prof. Dr. Hans Jürgen Ohlbach

Visualisierung einer Straßenkarte mit Hilfe von SVG

Projektarbeit
von
Frank Ipfelkofer

Aufgabenstellung:
Prof. Dr. Hans Jürgen Ohlbach

Betreuung:
Dipl.-Inf. Bernhard Lorenz

Zusammenfassung

Kapitel 1 beschäftigt sich mit der Aufgabenstellung dieser Arbeit:
Aus verschiedenen Datenquellen soll eine Karte in SVG erzeugt werden. Dazu müssen die gesamten Daten in ein einheitliches Format überführt werden. Da es eine große Menge an verschiedenen bestehenden Formaten gibt, gilt es einen Standard zu finden, in den alle anderen transformiert werden können. Aus diesem entwickelten Zwischenformat heraus ist es dann möglich, eine Karte zu erzeugen oder aber bestimmte Informationen zu sammeln. Auch ein Routingalgorithmus kann mit den Daten arbeiten. Kapitel 2 beschäftigt sich mit den Anforderungen, die das Zwischenformat erfüllen muss.
In Kapitel 3 wird das Format in den derzeitigen Standard GML integriert. Damit auch große Datenmengen verarbeitet werden können, muss eine Möglichkeit gefunden werden, die es zulässt, nur Teile der Daten auszuwählen bzw. anzuzeigen.
Kapitel 4 beschäftigt sich mit dem Konzept des dynamischen Nachladens, der Detailstufen und der Auswirkung auf das Datenformat. Als ein mögliches Ausgangsformat, das in das Zwischenformat überführt werden soll, wird in dieser Arbeit SQD gewählt, da Daten der Stadt München in diesem Format vorliegen.
Kapitel 5 beschäftigt sich mit eben diesem Format und den Problemen bei der Überführung in das Zwischenformat.
Um die Daten in SVG darstellen zu können, muss das Zwischenformat in ein SVG-Dokument transformiert werden. Die Techniken dazu werden in Kapitel 6 dargestellt.
Sämtliche in dieser Arbeit verwendeten Dateien werden in Kapitel 7 beschrieben. Im letzten Kapitel wird ein Ausblick auf zukünftige leistungsstärkere Versionen gegeben, sowie offene Fragen angesprochen.

Inhaltsverzeichnis

1 Aufgabenstellung **3**

2 Zwischenformat **4**
2.1 Anforderungen 4
2.2 Grundlegender Aufbau des Zwischenformats 5
2.3 Geometrischer Bestandteil des Zwischenformats 5
2.4 Abstrakter Bestandteil des Zwischenformats 6

3 GML **9**
3.1 Einführung GML 9
3.2 GML-Geometrie 10
3.3 GML-Features 11
3.4 Weitere GML-Definitionen 12
3.5 Verwendung von GML für das Zwischenformat 12

4 Dynamisches Nachladen und Detailstufen **17**
4.1 Rasterung 17
4.2 Detailstufen 19
4.3 Dynamisches Nachladen 19
4.4 Umsetzung aller Konzepte zur Generierung einer Karte 20

5 SQD - Sequential Data File **22**
5.1 Grunddatentypen 23
5.2 Satzdatentypen 25
5.3 Zusätzliche externe Daten 27

6 Realisierung der Karte mittels SVG **28**
6.1 Aufbau der Straßen 29

6.2 Aufbau der Straßennamen 29
6.3 Realisierung des dynamischen Nachladens 30
6.4 Realisierung der Level 32

7 Beschreibung der verwendeten Dateien 33
7.1 Zwischenformat 33
7.2 Graphische Oberfläche 33
7.3 Java-Sourcen zur Transformation der Daten 34

8 Ausblick 35
8.1 Mögliche Verbesserungen der Leistungsfähigkeit 35
8.2 Mögliche Erweiterungen 36

9 Literaturverzeichnis 37

A Auszug aus PmsMap.xsd 38

B ConcretMap.xsd 40

C Auszug aus Dynamic.html 42

D Dynamic.svg 43

E Auszug aus Dynamic.js 44

F Interface SvgWriter.java 47

G Interface XMLizable.java 48

H Utility SvgJsWriter.java 49

Kapitel 1

Aufgabenstellung

Die Aufgabe der Projektarbeit ist die Visualisierung von geographischen Daten mittels SVG (Scalable Vector Graphics). Als Ausgangsformat werden geographische Daten verwendet (hier SQD (Sequential Data File)), die in ein GML (Geographic Markup Language) - Zwischenformat transformiert werden. Aus diesem Zwischenformat heraus wird nun eine SVG - Karte generiert, die alle Straßen anzeigt. Des weiteren ist es möglich, Straßenabschnitte auszuwählen und ihre Daten an einen Server zu schicken. Dieser liefert dann ein Dokument, das mit Hilfe von Javascript-Befehlen die Routing - Ergebnisse anzeigt. Außerdem wird als Orientierungshilfe eine Möglichkeit geboten, nach Straßen auf der Karte zu suchen und diese zu fokussieren.

Kapitel 2

Zwischenformat

Nachdem bisher noch kein geographisches Standardformat entwickelt wurde und es eine große Menge an verschiedenen verbreiteten Formaten (SQD, ESRI, EDBS, ...) gibt, dürfen die Entwicklungen nicht auf ein einziges Format beschränkt werden. Deswegen muss ein eigenes Zwischenformat definiert werden, in das die benötigten Daten transformiert werden können und das als Grundlage für alle weiteren Operationen dient wie z.B. dem Aufbau der Karte, dem Routing, oder der Straßensuche. Eine Übersicht über die häufigsten in Deutschland verwendeten Formate findet sich in [GISMNGT].

2.1 Anforderungen

Alle geographischen Informationen lassen sich mit Hilfe von Punkten, Pfaden oder Flächen beschreiben. Weil das Ziel der Arbeit die graphische Darstellung und das Routing von Verkehrswegen (z.B. Straßen, Fußwege, Schienen) ist, muss das Zwischenformat hauptsächlich Pfade unterstützen. Dabei ist es nötig, die Pfade nicht direkt über Koordinaten, sondern über Knotenpunkte zu beschreiben, über die ein Routingalgorithmus dann sämtliche anliegende Pfade ermitteln kann. Diese Knotenpunkte besitzen als Eigenschaft die ihnen zugeordneten Koordinaten. Zur vollständigen Beschreibung der geographischen Situation benötigen die Pfade ebenfalls mehrere Eigenschaften (z.B. Distanz, Dauer bei Schienenverkehr, Fahrtrichtung, ...). Da immer wieder neue Eigenschaften hinzukommen können, muss das Format so aufgebaut werden, dass es problemlos zu erweitern ist. Es soll zudem auch für verschiedenste Zwecke genutzt werden. Deshalb muss es übersichtlich und

leicht lesbar sein. Darüber hinaus ist eine Gruppierungsart zu finden, die Objekte desselben Typs zusammenzufassen (z.B. alle Straßen in ein großes Straßennetz) und auf diese Weise Objekte eines bestimmten Typs ein- bzw. auszublenden.

2.2 Grundlegender Aufbau des Zwischenformats

Als grundlegende Datenstruktur bietet sich aufgrund der Anforderungen das XML-Format an. Nachdem alle geographischen Objekte mit Hilfe der drei Grundgeometrien, d.h. durch geometrische Formen (Punkte, Pfade, Flächen) und weiterer Eigenschaften beschrieben werden können, empfiehlt sich die Trennung der geometrischen Daten von den restlichen (abstrakten) Daten. Die abstrakten Daten sind Eigenschaften, die ein Objekt beschreiben, aber keinen geometrischen Bezug besitzen. Dazu zählen Informationen wie z.B. Benutzbarkeit (zeitlich, Fahrtrichtung, Verkehrsteilnehmerart, ...), Beschränkungen (Geschwindigkeit, Gewicht, ...) oder auch sonstige Informationen wie Straßenbreite.

2.3 Geometrischer Bestandteil des Zwischenformats

Das Augenmerk liegt auf Pfaden. Deshalb werden im geometrischen Teil des Zwischenformats hauptsächlich verschiedene Arten von Pfaden behandelt. Alle Knotenpunkte von Pfaden sowie auch geometrische Hilfspunkte werden als Referenzen zu Knotenpunkten („NodeRefs“) gespeichert. Diese Referenzen verweisen über eine „ID“ auf einen Knoten („Node“), der graphische Informationen wie z.B. Koordinaten enthält. Wegen des Ausgabemediums SVG werden alle Arten von Pfaden, die in SVG definiert werden können, unterstützt. Grundsätzlich hat jeder Pfad einen Start- und einen Endpunkt (repräsentiert durch die ersten beiden „NodeRefs“). Dann folgt die geometrische Figur. Im Falle einer quadratischen bzw. kubischen Kurve werden als nächstes die Zwischen-, und Interpolationspunkte angegeben. Im Falle eines Linienzuges ist es möglich, noch weitere Zwischenpunkte zu erfassen. Über die Angabe zusätzlicher Zwischenpunkte können auch verschiedene Arten von

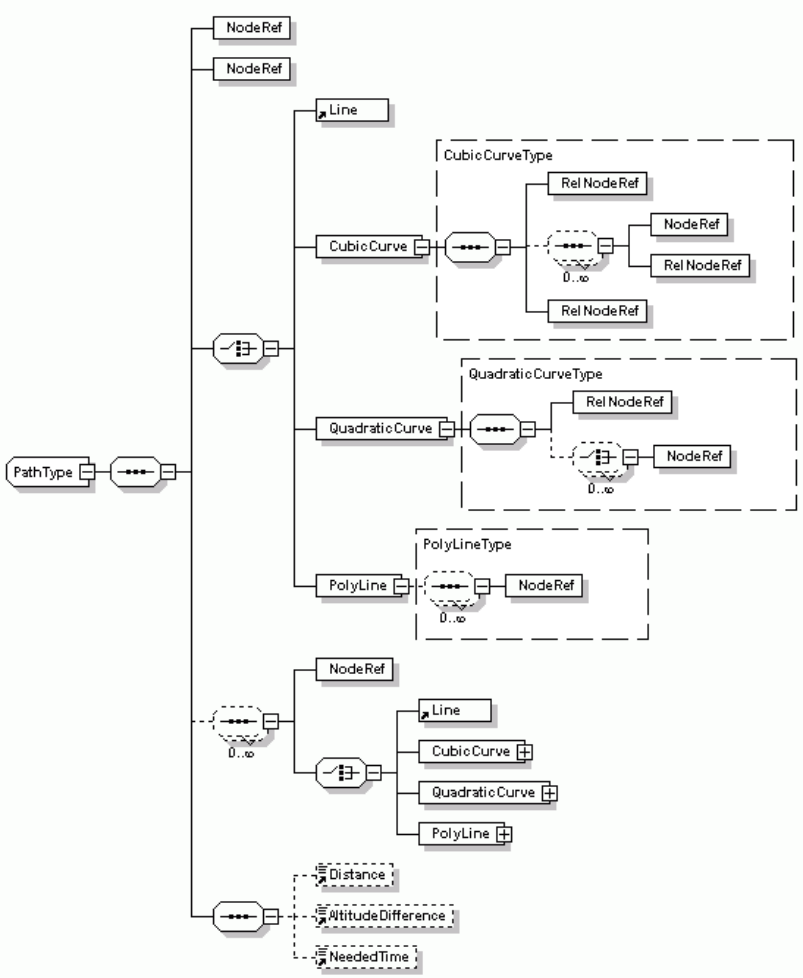

Abbildung 2.1: Ausschnitt aus dem geometrischen Datenformat

Pfaden zwischen dem Start-, und Endpunkt liegen. Als letztes folgen pfadspezifische Informationen wie z.B. die Länge des Pfades, die Höhendifferenz oder aber die Zeitspanne, die benötigt wird, diesen Pfad zu passieren (z.B. für U-Bahnen).

2.4 Abstrakter Bestandteil des Zwischenformats

Der abstrakte Teil beinhaltet das Objekt Karte („Map“). Dieses enthält einen Rahmen, repräsentiert durch ein Rechteck mit einer Referenz auf die linke

obere und rechte untere Koordinate. Des weiteren finden sich dort alle Definitionen der verwendeten Koordinaten, die als Attribut eine „ID“ besitzen, über welche sie referenziert werden können. Als letztes enthält es die verschiedenen Ebenen („Levels“), die auf der Karte definiert sind. Jedes Level hat da Attribut „Name“, das den eindeutigen Namen der Ebene speichert. Ansonsten liegen in der Ebene eine Menge von abstrakten Objekten. Jedes dieser Objekte besitzt einen Namen. Ansonsten sind sie unterteilt in „AbstractNodes“ (liegen an einem einzigen Punkt, z.B. Sehenswürdigkeiten), „AbstractPaths“ (besitzen einen Pfad, z.B. Straßen, Schienen) und in „AbstractArea“ (besitzen eine Fläche, z.B. Parks). Jedes von diesen abstrakten Objekten hat noch weitere spezifische Eigenschaften. Die Straße hat z.B. eine linke und eine rechte Straßenbreite.

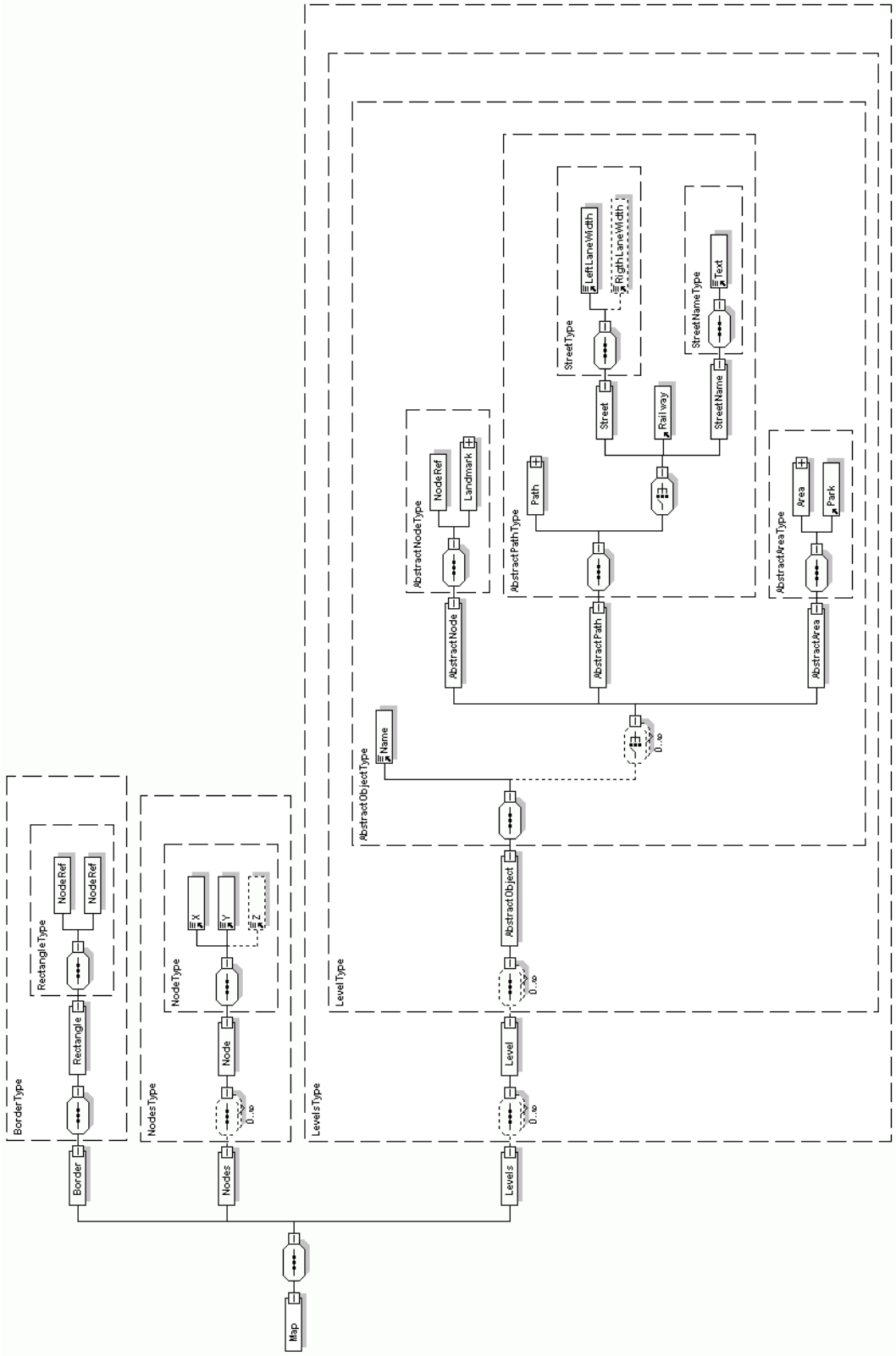

Abbildung 2.2: Ausschnitt aus dem abstrakten Datenformat

Kapitel 3

GML

Die Verwendung eines Standardformats erleichtert das Importieren bzw. Exportieren in andere Formate. Deshalb liegt es im Sinne der Wiederverwendbarkeit auf der Hand, das Zwischenformat in ein Standardformat zu bringen. Das Open GIS Consortium bietet dazu das Framework GML (Geography Markup Language [GML3]) an. Im Folgenden wird erst das Framework GML und im letzten Abschnitt die durchgeführte Integration des Zwischenformates in GML vorgestellt.

3.1 Einführung GML

GML ist eine XML (Extensible Markup Language) - basierte Sprache zur Kodierung von geographischen Daten. Dazu stellt sie eine Menge von XSD-Schemata (XML Schema definition language) zur Verfügung, die ein geographisches Dokument definieren können. GML bietet viele Vorteile für die Nutzung eines graphischen Informationssystems im Internet. Es ist eine Methode zum Transport und zur Speicherung geographischer Informationen im Kontext des Internets. Durch die effiziente und leicht verständliche Kodierung dieser Informationen und der klaren Trennung von Inhalt (Daten) und Darstellung wird eine einfache Integration von geographischen und nicht-geographischen Daten und deren leichte Erweiterbarkeit erreicht. GML ist jedoch nicht mehr als ein Framework zur Entwicklung untereinander kompatibler Anwendungen. Durch die Richtlinien die GML definiert, können einzelne Daten aber leichter ineinander überführt werden. Merkmale („Features“) werden durch geographische und nicht-geographische Eigenschaften beschrie-

ben. Eine geographische Eigenschaft kann als Geometrie abgebildet werden. GML muss definieren,

- welche geometrischen Grundobjekte benutzt werden sollen,
- wie Geometrie kodiert wird,
- wie Features modelliert werden.

3.2 GML-Geometrie

Die Geometrie ist die unterste Ebene der Architektur. Sie ist definiert in der Datei „geometry.xsd“. Im Gegensatz zu einer SVG - Architektur wird nur eine kleine Teilmenge von Pfaden definiert(Geometrie für „simple features“). Dies sind: Punkt („Point“), Linie („Line“), Linienzug(„LineString“), Polygon(„Polygon“) und eine Sammlung von gleichartigen Objekten („MultiPoint“, „MultiLine“, „MultiLineString“, „MultiPolygon“). Bei der Vermessung von Objekten können immer nur gerade Strecken (Linien) zwischen zwei Punkten vermessen werden. Deswegen werden alle nicht geraden Strecken (z.B. Kurven) über eine Menge von hintereinander liegenden Linien beschrieben. Aus diesem Grund gibt es in diesem Modell keine extra Beschreibung von Kurvenzügen, sondern es werden dafür Linienzüge verwendet. Die Geometrie ist in dem XML Schema geometry.xsd definiert. Alle geometrischen Daten erben vom Datentyp „AbstractGeometryType“. Wenn neue geometrische Datentypen erzeugt werden, müssen diese ebenfalls von diesem Typ erben. Die geometrischen Datentypen bestehen aus einer Menge von Koordinaten, über die sie definiert sind.
In Abbildung 3.1 ist ein Ausschnitt aus der Definition bereits vorgegebener Geometrietypen dargestellt, die alle den Datentyp „gml:_Gemoetry“ beerben, der seinerseits wiederum von „AbstractGeometryType“ erbt.

Die Typen, die für das Zwischenformat verwendet werden, sind „LineString“ und „Point“. Beide bestehen entweder aus einem einzigen oder einer Menge von Koordinaten, repräsentiert entweder durch die Datentypen „gml:coord“ Typen oder durch „gml:coordinates“. Während „gml:coord“ X,Y und evtl. die Z-Komponente getrennt verwaltet, werden in „gml:coordinates“ alle Informationen in einer Zeichenkette gespeichert.

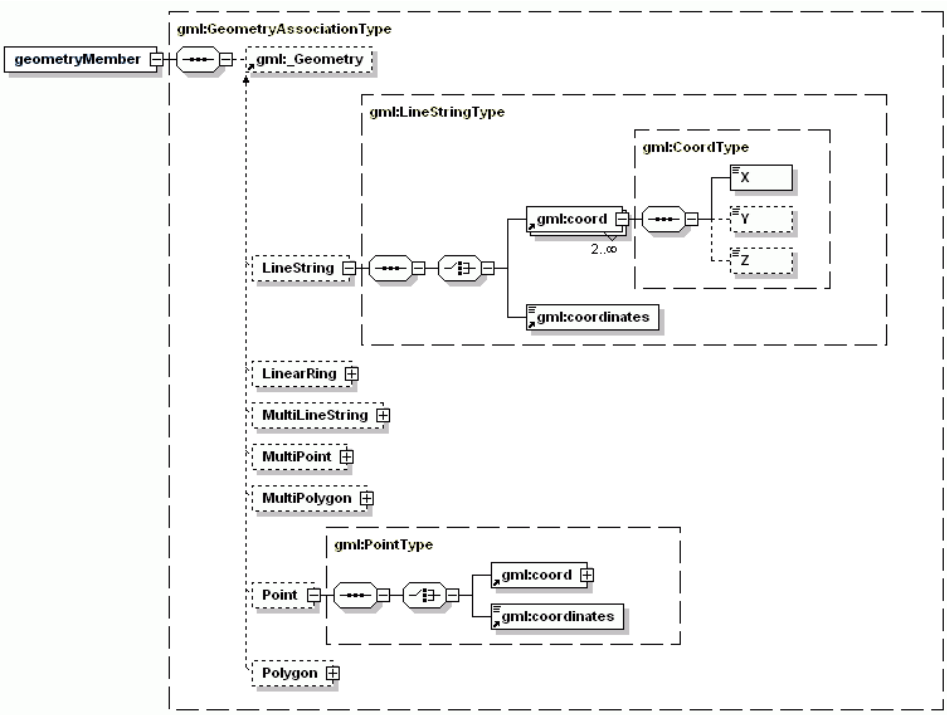

Abbildung 3.1: Ausschnitt aus geometry.xsd

3.3 GML-Features

Ein geographisches Merkmal („geographic feature“) wird als eine mit einem Namen versehene Liste geographischer und nicht-geographischer Eigenschaften modelliert und ist in der Datei „feature.xsd“ definiert. Die geographischen Eigenschaften werden dabei mittels der bereits definierten Geometrie-Typen kodiert. Ansonsten ist die konkrete Modellierung von Merkmalen anwendungsabhängig. Lediglich gewisse Eigenschaften, wie etwa Name, ID, Position oder Ausdehnung werden von den meisten Features benötigt. Um für solche gewöhnlichen Eigenschaften eine einheitliche Namensgebung zu erreichen, sind bereits eine Reihe von Namen standardmäßig definiert. Jedes Feature enthält die Attribute „gml:name“, „gml:description“ und „gml:boundedBy“. Alle abstrakten Objekte erben vom Datentyp „gml:AbstractFeatureType“
Abbildung 3.2 zeigt einen Ausschnitt aus der Definition der Merkmale. So beinhaltet jeder „gml:AbstractFeatureType“ die Eigenschaften Beschreibung („gml:description“), Name („gml:name“) und das sie umgebende Rechteck („gml:boundedBy“), das entweder „null“ ist oder durch eine linke obere und

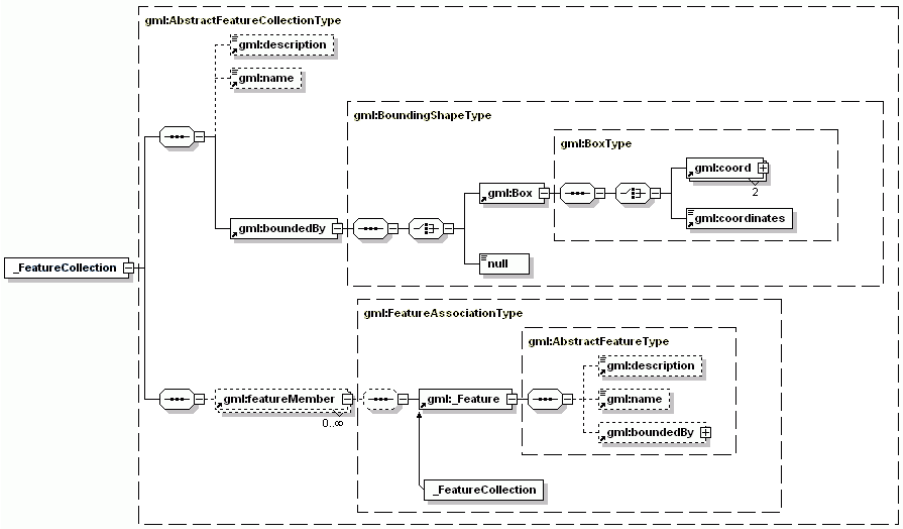

Abbildung 3.2: Ausschnitt aus feature.xsd

eine rechte untere Koordinate repräsentiert wird.

Eine „FeatureCollection“ ist eine Sammlung von Features und kann ebenfalls wieder als eigenes Feature aufgefasst werden.

3.4 Weitere GML-Definitionen

Zu den in dieser Arbeit verwendeten Standard-Schemata „Feature“ und „Geometry“ gibt es noch eine Menge anderer Objekte und Definitionen.

Abbildung 3.3 zeigt den Aufbau des GML 3 Schemas. Dort finden sich u.a Schemata zur Modellierung von Koordinatensystemen (z.B. Längen- und Breitengrade), Zeiten (Datum, Zeitdauer), Einheiten (Entfernungen) und Standardwerte. Eine genauere Beschreibung des gesamten Formates und der einzelnen Schemata enthält die Spezifikation [GML3].

3.5 Verwendung von GML für das Zwischenformat

Zur Integration des Zwischenformates in GML wird der Datentyp „MapMember“ als Grunddatentyp eingeführt. Dieser erbt von „gml:AbstractFeatureType“ und besitzt daher dessen Eingeschaften „gml:description“, „gml:name“ und

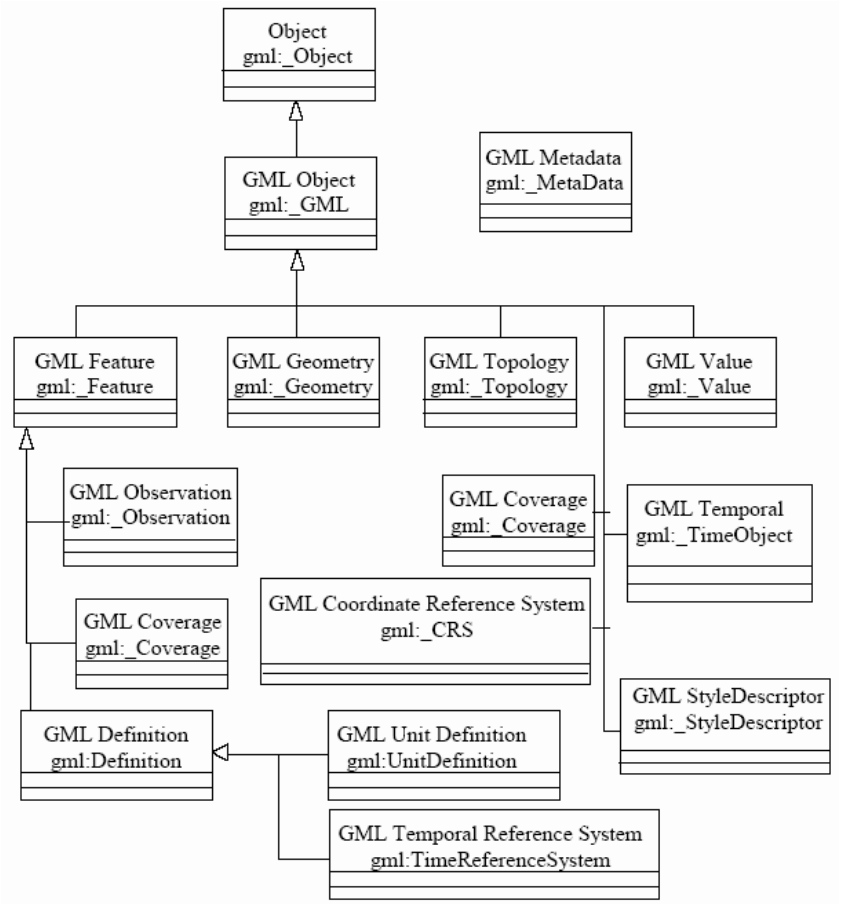

Abbildung 3.3: GML 3 Schema aus [GML3] (Kapitel 6.5)

„gml:boundedBy“. Alle Objekte, die auf der Karte enthalten sind, müssen von „MapMember“ erben. An abstrakten Datentypen gibt es bisher Bilder(„Image“), Straßen („Street“) und Straßennamen („Streetname“).
Abbildung 3.4 zeigt alle bisher definierten „MapMember“. Eine Straße besitzt einen Namen („gml:name“) und ist in mehrere Teilabschnitte(„StreetPart“) unterteilt. Jeder dieser Teilabschnitte repräsentiert den Teil der Straße zwischen zwei Kreuzungen. Nachdem dieser selten gerade ist, enthält er einen Linienzug, d.h. Zwischenpunkte, die über Linien verbunden sind.

Jeder dieser Abschnitte ist also ein Pfad zwischen zwei Knotenpunkten. Dieser wird in GML geometrisch durch die Eigenschaft „gml:centerLineOf“

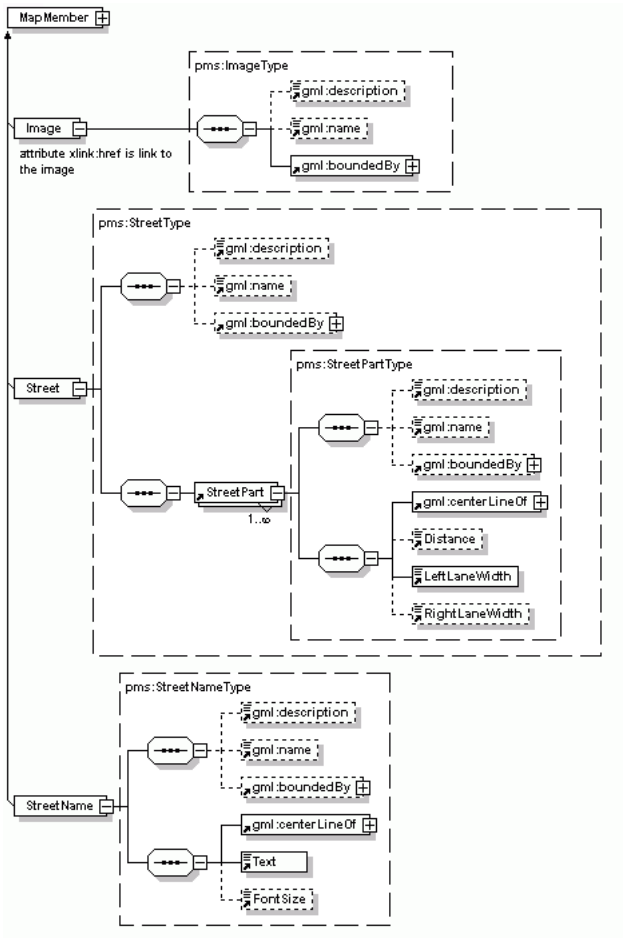

Abbildung 3.4: Überblick über MapMember

beschrieben, die den Datentyp „gml:LineString“ besitzt und daher alle Koordinaten des Linienzuges zwischen den beiden Kreuzungen enthalten kann.

Zusätzlich zu der geometrischen Information besitzt die Straße auch noch weitere Eigenschaften: Bisher sind „Distance“ für die Entfernung zwischen den beiden Knotenpunkten, „LeftLaneWidth“ bzw. „RightLaneWidth“ für die linke bzw. rechte Fahrbahnbreite definiert. Weitere benötigte Eigenschaften können jedoch beliebig hinzugefügt werden (z.B. Einbahnstraße, Geschwindigkeitsbeschränkung, keine Kraftfahrzeuge, Parkverbote, ...).

Nachdem nicht immer automatisch ermittelt werden kann, wo und wie ein Straßenname graphisch auf der Karte angezeigt werden soll, muss die

genaue Position angegeben werden. Daher wird der Datentyp „StreetName“ eingeführt. Der Name bzw. Text, der angezeigt werden soll, steht in der Eigenschaft „Text“. Die Positionierung des Namens erfolgt entlang eines Pfades, der in „gml:centerLineOf“ definiert ist. Der Text wird dabei möglichst oft in diesem Pfad wiederholt. Optional kann die Größe der Schrift mit der Eigenschaft „FontSize“ definiert werden.

Nachdem nicht auf der gesamten Karte Straßen sind, gibt es viele leere Flächen. Damit man sich besser orientieren kann, ist es sinnvoll auch andere Objekte anzuzeigen. Darüber existieren jedoch nur wenige oder keine Daten und darum werden diese Objekte in Form von Bildern (z.B. Satellitenbild) auf der Karte platziert. Für diesen Zweck wird der Datentyp „Image“ eingeführt. Die Adresse der Grafik wird mit der Eigenschaft „xlink:href“ definiert, die Größe und Position des Bildes durch den es umgebenden Rahmen („gml:boundedBy“).

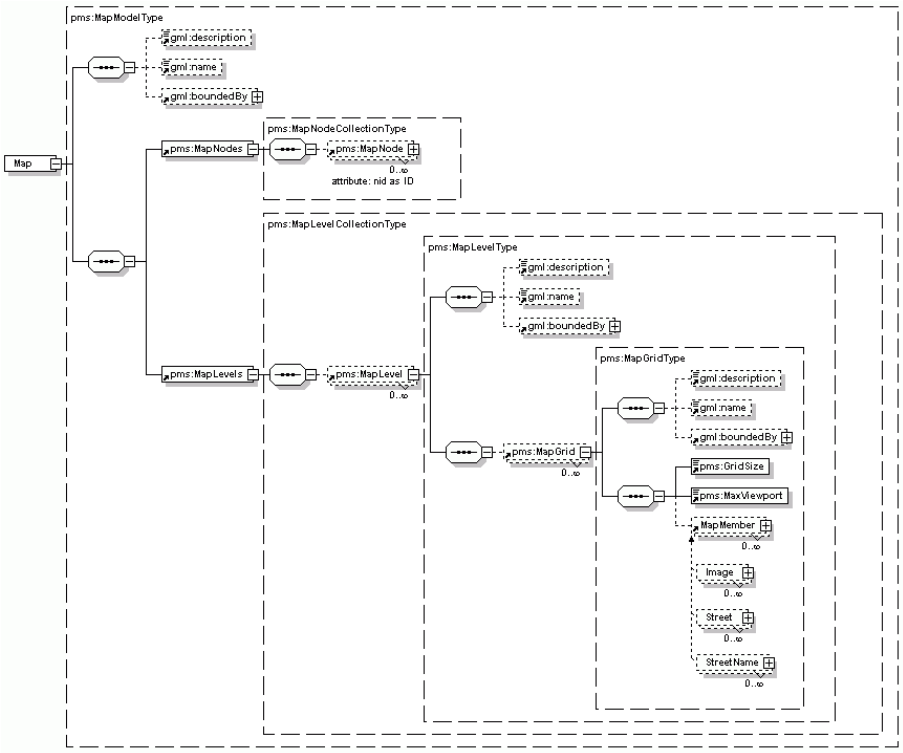

Abbildung 3.5: Auszug aus PMSMap

Abbildung 3.5 zeigt die komplette Definition einer Karte („Map“). Eine Karte beschreibt einen Bereich, der durch ein Rechteck eingegrenzt ist.

Dieser steht in der Eigenschaft „gml:boundedBy“. Alle verwendeten Knoten liegen in „pms:MapNodes“. Jeder Knoten erbt von „gml:coordinate“, fügt jedoch ein neues Attribut „nid“ hinzu, über das der Knoten referenziert werden kann. Als Gegenstück wird der Datentyp „pms:NodeRef“ definiert. Dieser substituiert den Datentyp „gml:coordinate“, besitzt jedoch als einzige Eigenschaft das Attribut „nid“, welches angibt, um welchen Knoten es sich handelt. Alle Koordinaten in den geometrischen Figuren werden dann mit diesem Datentyp definiert. Als letztes enthält die Karte noch die Eigenschaft „pms:MapLevels“. Darin sind alle Objekte enthalten. In einem „pms:MapLevel“ werden Objekte desselben Typs gruppiert. Innerhalb dieser Ebene liegen verschiedene Gitter, welche die Karte in verschiedene Raster aufteilen. Neben der Größe eines Rasters („GridSize“) wird auch noch ein Detaillevel („MaxViewport“) definiert. Des weiteren enthält das Gitter alle Objekte („MapMember“) des gruppierten Typs, die durch dieses Gitter gerastert werden sollen. Alle Informationen über das Konzept der Rasterung stehen im folgenden Kapitel.

Kapitel 4

Dynamisches Nachladen und Detailstufen

Es werden alleine zum Darstellen der Straßen einer Stadt wie z.B. München sehr große Datenmengen benötigt (allein für den Münchner Straßenplan 3 MB). Wenn nun eine Stadt, ein Landkreis oder ein noch größeres Gebiet modelliert werden soll, ist es nicht möglich, die gesamten Daten auf einmal zu laden, sondern immer nur die Bereiche, die gebraucht werden und sichtbar sein sollen.

4.1 Rasterung

Ein Problem von SVG ist, dass es keine Möglichkeit besitzt, ausschließlich die wirklich angezeigten Objekte zu berechnen, was bei großen Datenmengen zu großen Performanceeinbußen führt. Daher ist es nötig, die nicht sichtbaren Elemente auszublenden (unsichtbar zu machen), so dass deutlich weniger Objekte angezeigt und damit berechnet werden müssen. Deswegen wurde im Laufe dieser Arbeit das Konzept der Rasterung entwickelt. Dabei wird die Karte in Gebiete aufgeteilt. Jedes Gebiet hat eine gewisse Fläche und enthält alle in ihr enthaltenen Objekte. Bei jeder Änderung des Sichtbarkeitsbereiches durch Vergrößern / Verkleinern (Zooming) oder Verschiebung (Panning) werden nur all jene Gebiete angezeigt, die den aktuellen sichtbaren Bereich schneiden. Die anderen werden ausgeblendet. Deswegen muss SVG zum Zeichnen auch nur noch alle relevanten Bereiche berechnen.

Zur Rasterung gibt es verschiedene Modelle:

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

Abbildung 4.1: Karte mit quadratischem Raster

2. Es wird versucht, ein Raster zu bilden, in dem möglichst keine Objekte in mehreren Rastern liegen, z.B. Ausrichtung entlang der Straßenzüge.

Während die Bestimmung der sichtbaren Rasterpunkte im Modell 1 relativ einfach ist, muss im Modell 2 für jeden Rasterpunkt eine detaillierte Positionsbeschreibung vorliegen. Auch die Ermittlung von sinnvollen Rasterungen im Modell 2 ist relativ aufwendig. 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 aufwändigeren Implementierbarkeit des Modells 2. Deswegen wird in dieser Arbeit das quadratische Raster gewählt. Obwohl nun einige Objekte mehrfach vorkommen, bringt die Rasterung einen deutlichen Geschwindigkeitsgewinn.

4.2 Detailstufen

Gewisse Objekte sind zu klein, um in einem großen Ausschnitt wahrgenommen zu werden, z.B. kleine Straßen, Straßennamen,... und so macht es keinen Sinn, diese immer anzuzeigen. Deshalb wird mit der Funktionalität der Detailstufen eine Möglichkeit erzeugt, zu kleine Objekte auszublenden. Im Datenmodell wird dieses Konzept durch das „MapGrid“ umgesetzt.

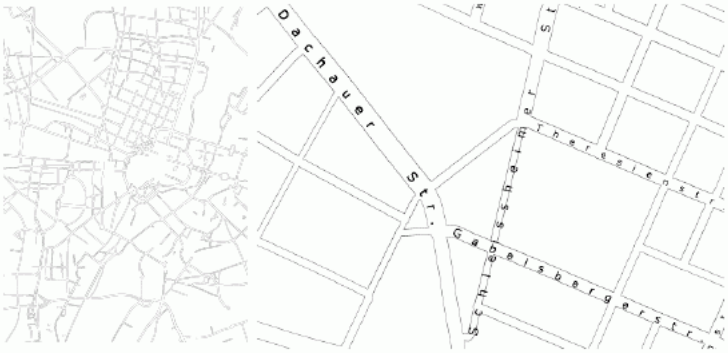

Abbildung 4.2: Detailstufen: Einige Details (links) - viele Details (rechts)

4.3 Dynamisches Nachladen

Dynamisches Nachladen bedeutet, Daten lediglich dann nachzuladen, wenn sie benötigt werden. Das Prinzip dabei ist, dass zu Beginn nur eine (kleine) Standard-Karte geladen wird. Bei Bewegung auf der Karte bzw. beim Zoomen werden neu benötigte Kartenbereiche nachgeladen und in die Karte eingebaut. Dies ist leider keine Standard SVG-Funktion und muss deswegen implementiert werden. Nachdem SVG kein Nachladen weiterer Dokumente unterstützt, muss die Karte dynamisch ergänzt werden. Dafür verwendet man die Javascript-Schnittstelle. Benötigt man einen neuen Bereich, wird in einem neuen Fenster eine HTML-Seite aufgerufen, die den Javascriptcode enthält. Dieser ergänzt die Karte dynamisch. Nach Beendigung der Einfügung wird diese Seite wieder geschlossen. Somit ist es innerhalb der Laufzeit möglich, die Karte immer weiter zu vervollständigen.

4.4 Umsetzung aller Konzepte zur Generierung einer Karte

Durch die Verbindung dieser drei Konzepte kann eine sehr leistungsfähige Karte implementiert werden.

Zuerst werden dazu alle Objekte in verschiedene Gruppen (Level) wie z.B. Straßen oder Bahnstrecken aufgeteilt. Alle Objekte in einem Level werden nun nach ihrer Größe (Breite der Straße, Verkehrsdichte, ...) sortiert und Objekte mit ähnlicher Größe wieder gruppiert. Jeder dieser Gruppen wird eine Detailstufe zugeordnet, ab der sie sichtbar ist. So besitzen kleinere Straßen oder Straßennamen eine geringe Detailstufe, während Hauptstraßen oder Autobahnen immer größere Detailstufen zugegeordnet werden und sie deswegen schon bei einem viel größeren Kartenausschnitt sichtbar sind.

In einem weiteren (automatischen) Schritt werden alle Objekte eines Levels mit derselben Detailstufe gerastert. Die Größe des Rasters ist dabei so groß gewählt, dass wenn der sichtbare Kartenausschnitt der Detailstufe entspricht, genau ein ganzer Rasterpunkt angezeigt werden kann. Nach Bestimmung der Rastergröße werden nun die Objekte in die Raster aufgeteilt. Wenn Objekte in mehreren (benachbarten) Rasterpunkten liegen, enthält jedes von diesen eine Kopie des Objektes und somit kommt es auf der Karte mehrfach vor. Da jedoch alle größeren Objekte in kleinere Abschnitte zerlegt sind (z.B. Straßen in Straßenabschnitte) und nur über diese Abschnitte gerastert wird, gibt es nur sehr wenige doppelte Informationen. Eine Optimierung zur Reduzierung dieser Mehrfachinformationen würde also fast keine Verbesserung erbringen.

Nun wird für jeden Rasterpunkt, der Objekte enthält, eine Javascript-Datei erzeugt, die die Befehle enthält, um die Objekte graphisch in das SVG Dokument einzubetten. Dabei wird folgende einheitliche Namensgebung verwendet: „[LEVEL] [RASTERGRÖSSE] - [X],[Y]“ , wobei X und Y angeben, dass es sich um den Rasterpunkt (X,Y) in dem Raster mit der Größe „RASTERGRÖSSE“ handelt. Da die enthaltenen Objekte von der Detailstufe und der Position im Raster abhängen, enthalten diese Dateien verschieden viele Daten. So gibt es Dateien mit nur einem Objekt (100 Bytes), aber auch mit über 7000 (650 KB).

Eine Information über alle existierenden Rasterpunkte und ihre zugehörigen Dateinamen wird in einer separaten Datei gespeichert, die zu Beginn geladen werden muss. Bei Änderungen des sichtbaren Bereiches auf der Karte

wird mit Hilfe dieser Informationen ermittelt, welche Raster angezeigt werden müssen. Dazu werden in allen sichtbaren Leveln für alle Detailstufen die sichtbaren Rasterkoordinaten berechnet und verglichen für welche von diesen Daten existieren. Wenn Daten von einem dieser Rasterpunkte noch nicht vorhanden sind, wird die dazugehörige Datei geöffnet (in einem seperaten Browserfenster geladen). Der darin enthaltenen Javascriptcode fügt die entsprechenden Objekte dynamisch in das SVG-Dokument ein. Dabei liegen alle Objekte eines Rasterpunkts in einer Gruppe. Nun werden alle Gruppen, die sichtbare Rasterpunkte enthalten sichtbar (`style='display:block'`) und alle anderen unsichtbar (`style='display:none'`) gemacht.

Damit wird erreicht, dass nur sichtbare Objekte berechnet bzw. geladen werden müssen und auch größere Datenmengen verarbeitet werden können.

Kapitel 5

SQD - Sequential Data File

Das Zwischenformat wird entwickelt, um verschiedene Datenformate zu verwenden und diese in das Zwischenformat zu konvertieren. Als Beispiel ist in der Projektarbeit SQD [SQDGISM] als Ausgangsformat gewählt, nachdem es das Standard - Geobasisformat des Baureferats München [BAUMUC] dargestellt. Innerhalb des Systems SICAD speichert die SICAD-GDB (Geographische Datenbank) alle geographischen und nichtgeographischen Informationen. SQD ist ein Schnittstellenformat, das einen beliebigen rechteckigen Ausschnitt der SICAD-GDB beschreibt. Es kann für Datenaustausch und Datenerhaltung verwendet werden. Beim Erzeugen dieser Datei werden alle in dem Ausschnitt enthaltenen Elemente in die Geometriedatei geschrieben. Zur Lesbarkeit wird jeweils zwischen zwei Elemente eine Kommentarzeile, beginnend mit „*“, eingefügt.

Innerhalb des SQD-Formates gibt es zwei Arten der Speicherung von geographischen Informationen, entweder als Grunddaten oder als Satzdaten. Während die Grunddaten hauptsächlich zur Speicherung von geometrischen Informationen konzipiert sind, bieten die Satzdaten die Möglichkeit, Objekte genauer zu beschreiben. Alle zusätzlichen Informationen, die weder in Grunddaten noch in Satzdaten gespeichert werden können, müssen in eine externe Datei ausgelagert werden. In den für diese Arbeit vom Baureferat München zur Verfügung gestellten Daten werden alle drei Arten der Informationsspeicherung verwendet. Bei den Zusatzdaten handelt es sich um eine separate Excel-Datei.

5.1 Grunddatentypen

Für jedes Element werden in der ersten Zeile die Grunddaten angegeben. Das sind: Ebenennummer (EB), Elementnummer (ENUM), Hierarchiestufennummer (STU), Elementtyp (ETYP), Strichmodus (SM) und Strichstärke (ST). Die beiden wichtigsten Typen sind dabei der Elementtyp und die Stufe.

An Elementtypen gibt es hauptsächlich folgende:

- PG: Punkt
- LI: Linie (Verbindung zwischen zwei Punkten)
- LY: Polygon (Verbindung zwischen zwei Punkten über Zwischenpunkte)
- BO: Bogen (Bogen von einem Punkt zu einem anderen)

Das Konzept der Stufen dient dazu, komplexere Objekte in einfachere Teilobjekte zu zerlegen. Dabei liegen Elemente einer höheren Stufe in einem Element niedrigerer Stufe und sind damit ein Teil davon. Linien, Polygone und Bögen mit der Stufe 1 besitzen einen untergeordneten Start- und Endpunkt mit Stufe 2, z.B.:

```
****************************************************
ETYP=LY STU=1 ENUM=9c4000000001bf49 EB=2 ST=16 SM=1
*****************************************************
ETYP=PG STU=2 ENUM=9c4000000001bf47 EB=11 ST=65 SM=1
*****************************************************
ETYP=PG STU=2 ENUM=9c4000000001bf48 EB=11 ST=65 SM=1
****************************************************
```

Es gibt zwei Formate von Koordinaten. Das ist das X-Format und das Z-Format, gekennzeichnet mit einem X bzw. Z vor der Koordinate. Leider gibt es zu diesen Formaten keine frei verfügbare Beschreibung im Netz. Aber die Koordinaten können beim Export auch zusätzlich als Floatingpointzahl angegeben werden, die für alle Operationen verwendet werden kann:
`X415104BDC7BE76C9 4.461303121e+006`

Ein Punkt (ETYP=PG) besitzt zwei Attribute „X“, „Y“ für seine Koordinaten und eine eindeutige Nummer (PNR), z.B. `PNR 37030092`.

```
ETYP=PG STU=2 ENUM=9c4000000001bf47 EB=11 ST=65 SM=1
X X415104BDC7BE76C9   4.461303121e+006
Y X415452206867381D   5.326977631e+006
...
PNR 37030092
```

Zwischenpunkte eines Polygons (ETYP=LY) werden mit dem Attribut „FLD“ definiert. Dabei gibt jede Zeile abwechselnd X- und Y- Koordinaten der Zwischenpunkte an:

```
ETYP=LY STU=1 ENUM=9c40000000000561 EB=2 ST=16 SM=1 NAM 'YYYY'
FLD X4151054B20780000   4.461868507e+006
    X415451CC05000000   5.326640078e+006
    X4151054E83840000   4.461882055e+006
    X415451D5C7000000   5.326679109e+006
```

Somit definiert z.B. folgender Ausschnitt ein vollständiges Polygon mit Start- und Endpunkt und zwei Zwischenpunkten:

```
**************************************************
ETYP=LY STU=1 ENUM=9c4000000001bf49 EB=2 ST=16 SM=1 NAM '    '
FLD X4151054B20780000   4.461868507e+006
    X415451CC05000000   5.326640078e+006
    X4151054E83840000   4.461882055e+006
    X415451D5C7000000   5.326679109e+006
UTOBJD 'A22\_\_030d5W2206BM00000' POINTER kob\_object OBJID
'A22\_\_030d5W2206BM00000' POINTER strassenabschnitte OBJID
'A22\_\_030d5W2206BM00000'
**************************************************
ETYP=PG STU=2 ENUM=9c4000000001bf47 EB=11 ST=65 SM=1 X
X415104BDC7BE76C9   4.461303121e+006 Y X415452206867381D
5.326977631e+006 PKZ 'T1  ' PNR 37030092 UTOBJD
'012\_\_Op0d5W2206BK00000' POINTER kob\_object OBJID
'012\_\_Op0d5W2206BK00000' POINTER strassennetzknoten OBJID
'012\_\_Op0d5W2206BK00000'
**************************************************
ETYP=PG STU=2 ENUM=9c4000000001bf48 EB=11 ST=65 SM=1 X
X41510490F420C49C   4.461123815e+006 Y X4154523DA7A92A30
```

```
5.327094620e+006 PKZ 'T1  ' PNR 36040017 UTOBJD
'i12\_\_KA0d5W2206BL00000' POINTER kob\_object OBJID
'i12\_\_KA0d5W2206BL00000' POINTER strassennetzknoten OBJID
'i12\_\_KA0d5W2206BL00000'
************************************************
```

5.2 Satzdatentypen

Die Grunddaten enthalten lediglich geometrische Informationen, aber keine Angaben, um welchen Verbindungstyp es sich handelt. Daher wird ein Datentyp benötigt, der flexibler ist als die Grunddaten. Das bietet SQD. Es stellt dazu Satzdaten zur Verfügung, die mit dem Befehl SATZTYP beginnen. Anschließend folgt der Name des Typs.

„SATZTYP strassennetzknoten“ repräsentiert einen Punkt mit den wichtigen Attributen: XKOORD, YKOORD, NR.

„SATZTYP strassenabschnitte“ repräsentiert eine Linie mit den wichtigen Attributen: „KNOT_NR_VON“, „KNOT_NR_NACH“, „STR_NR“, „NR“ (Abschnittsnummer). Darüber hinaus können die Satztypen weitere Informationen enthalten, wie Länge, Erfassungsdatum, Abschnittsnummer, ...

Beispiel der Satzdaten:

```
*
SATZTYP     strassennetzknoten
NR          37030014
OBJID       '0000000005__sknoten___'
XKOORD      4461860.213
YKOORD      5326619.068
LOCATION    '9c40000000000560'
BBEZ_NR     'S'
MEIST_NR    '61'
*
SATZTYP     strassenabschnitte
KNOT_NR_VON 37030014
KNOT_NR_NACH 37030013
KENNZ       '0'
STR_NR      '9099'
NR          1500
BBEZ_NR     'S'
MEIST_NR    '61'
LAENGE      557.5955
RICHTUNG    'N'
BAULAST     'M'
DATUM       04.07.2002
STATUS      '3'
OBJID       '0000000004__sabschn___'
STRKLASSE   'L'
LOCATION    '9c40000000000561'
DSP_ABSCHNITT '1500'
*
```

5.3 Zusätzliche externe Daten

Die Satzdaten bieten zwar im Gegensatz zu den Grunddaten schon mehr Möglichkeiten, weitere Informationen zu speichern. Jedoch können auch hier nicht alle benötigten Informationen abgelegt werden. Deshalb werden diese in einer separaten Datei gespeichert. Für die Daten, die in der Projektarbeit verwendet werden, gibt es diese Informationen in einer Excel-Datei. Diese enthält folgende Felder:

Feld	Beschreibung
STR_N	Name der Straße
STR_BEZ	Bezeichnung der Straße
ABS_NR	Erste Nummer des Straßenabschnitts
VON_STR	Bezeichnung der kreuzenden Straße, ab welcher der Abschnitt beginnt
BIS_STR	Bezeichnung der kreuzenden Straße, bei welcher der Abschnitt endet
RI	Richtung der Hausnummern
VON	Abschnittsbeginn in Metern
BIS	Abschnittsende in Metern
GW_LI	Breite des Gehwegs auf der linken Straßenseite
RW_LI	Breite des Radwegs auf der linken Straßenseite
FB_LI	Breite der linken Fahrbahnseite
FB_RE, RW_RE, GW_RE	Analog zu links
MITTELSTR	Breite des Mittelstreifens
ABS_KNOT_NR_VON	Knotennummer, bei welcher der Abschnitt beginnt
ABS_KNOT_NR_NACH	Knotennummer, bei welcher der Abschnitt endet

Die Zuordnung zwischen SQD-Datei und Excel-Zusatzdatei erfolgt nun entweder über Knotennummern bzw. über Straßennummern und Abschnittsnummern. Da in den vorliegenden Dateien keine einheitliche Zuordnung möglich ist (nicht immer sind Knotennummern bzw. Satzdaten vorhanden), müssen beide Optionen verwendet werden.

Damit ist eine Zuordnung von über 99,7% der Straßen möglich. Zum Teil fehlen in der externen Datei auch Angaben über Straßenbreiten. Dann definiert die Breite des vorhergehenden Abschnitts der Straße (falls vorhanden) hilfsweise die Breite des folgenden Abschnitts.

Kapitel 6

Realisierung der Karte mittels SVG

Ziel der Projektarbeit ist die Darstellung der Daten in einer Art Stadtplan mit Hilfe von SVG. Straßen sollen durch weiße Linien dargestellt und mit einem dunklen Rand gegen den Hintergrund abgegrenzt werden. Die Straßennamen sollen auf den Straßen angezeigt werden können.

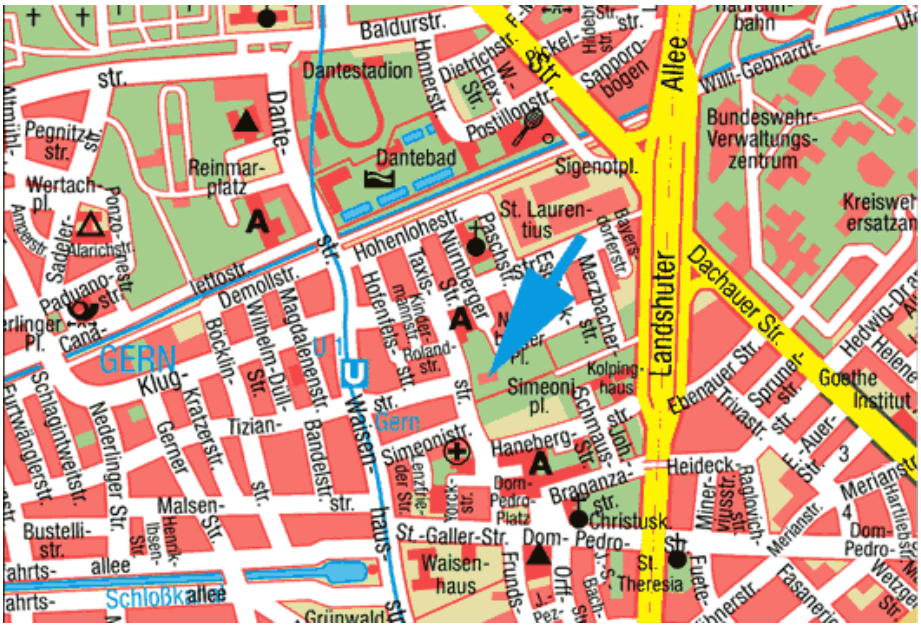

Abbildung 6.1: Herkömmlicher Stadtplan

6.1 Aufbau der Straßen

Straßen werden durch Linienzüge repräsentiert, die mittels des SVG-Elements `<path>` erzeugt werden. Die Farbe wird mit dem Style „stroke“ bestimmt. Die Breite gibt der Style „stroke-width“ an. Um das Aussehen der Straßen wie in einem Stadtplan zu erhalten, wird folgende Vorgehensweise entwickelt: Jede Straße wird doppelt gezeichnet. Dabei wird zunächst jeder Straßenhintergrund in grauer Farbe gezeichnet (Abbildung 6.2 links). Danach werden die eigentlichen Straßen mit weisser Farbe ein wenig dünner gezeichnet (Abbildung 6.2 Mitte).

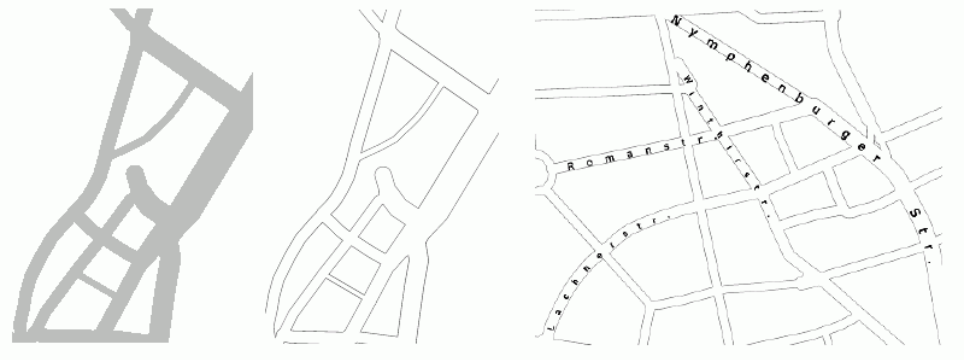

Abbildung 6.2: Schritte zum Aufbau der Straßen

Ein Straßenabschnitt wird also mit folgenden Befehlen in SVG gezeichnet:

```
<path d="M19982 16351 l80 1 l5 -4 l181 -2 l45 0 l163 2"
      style="stroke-width:10;stroke: gray" />
<path d="M19982 16351 l80 1 l5 -4 l181 -2 l45 0 l163 2"
      style="stroke-width:8;stroke: white" />
```

6.2 Aufbau der Straßennamen

Anschließend werden die Straßennamen gezeichnet. Entlang des definierten Pfades für den Namen wird der Straßenname im Dokument so oft wie möglich geschrieben. Nachdem die Berechnung der Pfade relativ rechenintensiv ist, wird aus Performancegründen jeder Teilabschnitt als eigenes Element gezeichnet. Dazu wird das SVG-Element `<textPath>` verwendet. Straßen können verschieden breit sein und deswegen auch die Straßennamen verschieden groß. Hier gibt es die Möglichkeit, die Schriftgröße zu bestimmen. Dazu wird der style „font-size“ gesetzt. Das folgende Beispiel zeigt die Befehle in SVG, um den Namen zu erzeugen. Erst wird der Pfad definiert. Anschließend wird mit `<textPath>` der Name entlang des Pfades geschrieben. (Abbildung 6.2 rechts)

```
<path d="M19982 16351 l80 1 l5 -4 l181 -2 l45 0 l163 2" id=" Text1203"
            style="stroke-width:10;stroke: gray" />
<textPath style="font-size:10" xlink:href="#Text1203"
            name="Oettingenstr."> Oettingenstr.</textPath>
```

6.3 Realisierung des dynamischen Nachladens

Wie in Kapitel 4.3 beschrieben, gibt es beim dynamischen Nachladen kein SVG-Dokument mehr, sondern nur noch Javascript-Befehle, die dann die Elemente im SVG-Dokument aufbauen z.B.:

```
function addStreet(strokeWidth, path, ID, name)
addStreet(10.0," M19982 16351 l80 1 l5 -4 l181 -2 l45 0 l163 2",
                 "STREET8191", "Oettingenstr.");
```

oder :

```
function addPath(strokeWidth, path, ID)
function addTextPath(PathID, Text, fontSize)

addPath(0.0, "M6513 9079 l12 -15 ... l48 47", "Text135");
addTextPath("Text135", "Verdistr.", 23);
```

Dabei werden zunächst neue Elemente erzeugt:
`var element = document.createElement("[TAGNAME]")`

Diesen werden benötigte Attribute wie „id“ gegeben:
`element.setAttribute("id", ID)`.

Anschließend wird dann dieser Knoten dem DOM-Baum an einer bestimmten Stelle hinzugefügt (richtiger Gitterpunkt in der Gruppierung, in die das Element gehört). Dies erfolgt mit `parentNode.appendChild(element)`.
Dabei gibt es folgende Regeln:

- Straße: Eine Straße besteht aus zwei Pfaden (`<path>`). Einer für den Hintergrund und einer für den Vordergrund. Beide besitzen das Attibut „name“, das den Namen der Straße beschreibt. Das Aussehen wird mit Hilfe einer Styleklasse beschrieben (z.B. „streetBackground“). Die Breite wird mit dem Attribut „strokeWidth“ bestimmt. Da auf den Vordergrund zugegriffen werden soll, z.B durch Markierung oder Auswahl, besitzt dieser das Attribut „id“, mit der eindeutigen Straßen-ID.

Abbildung 6.3: Problem des Zeichnen von Straßenkreuzungen

Abbildung 6.3 (links) zeigt die Graphik, wenn ein Straße vollständig über eine andere gezeichnet wird. An den Kreuzungen verläuft der graue Rand über die Querstraße hinweg, so dass optisch keine Kreuzung mehr sichtbar ist. Das kann verhindert werden, wenn man zuerst jeweils den Hintergrund zeichnet und danach erst den Vordergrund 6.3 (rechts). So liegen Hintergrund und Vordergrund in zwei verschiedenen Untergruppen der Gruppe Straße. Da die Untergruppe des Hintergrunds im Dokument vor dem Vordergrund steht, werden in SVG immer erst alle Hintergrundelemente gezeichnet und über diese dann die Vordergrundelemente.

- Straßennamen: Die Beschriftung verläuft entlang eines Pfades. Deswegen muss als erstes ein Pfad definiert werden. Nachdem dieser nicht sichtbar ist, wird er im Definitionsbereich des SVG-Dokuments (`<DEFS>`) erzeugt. Dieser Pfad enthält als Attribute nur den Verlauf (Attribut „d“) und eine „id“. Anschließend wird er in das zugehörige Raster der Straßennamen hinzugefügt (`<textPath>`). Dieser enthält neben der Referenz (Attribut „xlink:href“) auf die „id“ die Schriftgröße („font-Size“) und den anzuzeigenden Text. Ein Fehler in aktuellen SVG-Implementation ist, dass das dynamische Erzeugen von Referenzen nicht funktioniert, wenn man ein `<textPath>` Element erzeugt und diesem das Attribut „xlink:href“ zuweist. Dann wird trotz korrekter Syntax kein Text angezeigt. Deswegen wird folgender Trick angewandt: Im zu Beginn geladenen SVG-Dokument wird im Definitionsteil (`<DEFS>`) als Vorlage ein `<textPath>` Element erzeugt, das bereits eine Referenz auf einen Pfad enthält. Dieser wird dann mit `element.cloneNode(true)` kopiert und an der gewünschten Stelle eingefügt. Damit wird das Problem umgangen.

6.4 Realisierung der Level

Alle Elemente eines Levels werden in eine Gruppe geschrieben, die je nach Bedarf sichtbar oder unsichtbar gemacht werden können. Ebenso liegen alle Objekte desselben Gitters eines Gitternetzes in einer Gruppe. Die Level können mit Checkboxen aus- und eingeblendet werden. Die anzuzeigenden Gitterpunkte werden aus dem aktuellen sichtbaren Bereich berechnet und automatisch eingebettet.

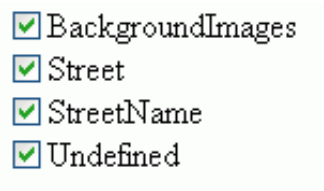

Abbildung 6.4: Auswahl der Level mittels Checkboxen

Kapitel 7

Beschreibung der verwendeten Dateien

Im folgenden Kapitel werden alle verwendeten Dateien und ihr Zusammenhang beschrieben.

7.1 Zwischenformat

Die Schemata „feature.xsd“ und „geometry.xsd“ sind lokale Kopien der offiziellen GML -Schemata, die unter [GMLSC] zu finden sind. „Xlink.xsd“ ist eine Kopie des allgemeinen Link - Schemas für XML-Dateien, das sich unter [XLINK] finden lässt. „PmsMap.xsd“ definiert den Aufbau der Karte, während „ConcretMap.xsd“ konkrete Objekte (Straßen, Straßennamen und Bilder) definiert.

7.2 Graphische Oberfläche

Die Dateien, welche die graphische Oberfläche mit HTML und SVG erzeugen, befinden sich in dem Ordner „dynamic“. Sie bestehen aus folgenden Dateien:

- Dynamic.html : Die Hauptdatei, die den gesamten Aufbau definiert.
- Loader.html: Wird aufgerufen, wenn neue Abschnitte nachgeladen werden sollen.
- RoutingWindow.html: Zeigt Routingergebnisse in einem Fenster und auf der Karte an.

- Dynamic.svg: Ein fast leeres SVG-Dokument, in das die Daten dynamisch eingebettet werden.
- MapCreator.css: Stylesheet der Karte.
- Dynamic.js: Das Herz des dynamischen Aufbaus. Es beinhaltet alle wichtigen Methoden zum Aufbauen und Verändern von Elementen im SVG-Dokument.
- DynamicChooser.js: Wird dynamisch erzeugt und beinhaltet eine Auflistung aller verfügbaren Gitterpunkte.
- GridManager.js: Reagiert auf Änderungen der Sicht und blendet entsprechend Gitterpunkte ein und aus.
- MarkingManager.js: Bietet Methoden, um Objekte verschieden zu markieren und Markierungen wieder zu entfernen.
- SelectionManger.js: Schnittstelle zum Abfangen von Mausklicks auf der Karte. Speichert die ausgewählten Abschnitte, um sie an das Routing zu übergeben.
- StreetSearch.js: Dynamisch erzeugte Auflistung aller vorhandenen Straßen mit Abschnitten.
- TmpRouting.js: Demonstriert die Verwendung der Markierungsschnittstelle, um den aktuellen Status des Routings anzuzeigen.
- Ordner maps: Alle dynamisch erzeugten Rasterpunkte der Karte.

7.3 Java-Sourcen zur Transformation der Daten

Die Java - Sourcen, die die Transformationen durchführen, sind in zwei Pakete unterteilt:

- svgMap: Beinhaltet eine Java-Repräsentation des Zwischenformates. Über die Verwendung der Schnittstelle svgMap.interfaces.XMLizable wird das Einlesen und Ausgeben in das Zwischenformat erreicht. Die Schnittstelle svgMap.interfaces.SvgWriter ermöglicht das Erzeugen eines SVG-Dokuments
- sqdParser: Liest die SQD - Daten aus der SQD - und der Excel - Datei und baut aus diesen Daten eine Karte (svgMap.object.Map) auf. Dabei werden die Straßen und Straßennamen in verschiedene Ebenen geschrieben.

Für weitere Informationen steht eine Javadoc-Dokumentation zur Verfügung.

Kapitel 8

Ausblick

Die Projektarbeit ist lediglich ein Prototyp eines Systems, das noch eine Menge an Erweiterungsmöglichkeiten bietet. Neben der Implementierung eines Routing-Algorithmusses könnten z.B. noch weitere Datenformate unterstützt werden. Das folgende Kapitel beschäftigt sich mit einer Reihe von möglichen Erweiterungen.

8.1 Mögliche Verbesserungen der Leistungsfähigkeit

Die Leistung des für diese Arbeit entwickelten Programmes ist für kleinere Datenmengen schon ausreichend. Doch gibt es noch ein paar Hindernisse, die bei größeren Datenmengen auftreten. Folgende Verbesserungen wären hilfreich:

- Straßensuche: Im Moment wurde noch kein Server integriert. Deswegen werden für eine Straßensuche alle Straßeninformationen in eine extra Javascriptseite gespeichert. Bei einem Suchaufruf werden alle Namen durchsucht und die passenden angezeigt. Bei Auswahl eines Elements wird überprüft, ob der zugehörige Gitternetzpunkt bereits geladen wurde, ggf. wird er nachgeladen. Dann wird der Straßenabschnitt fokussiert und markiert. Für größere Projekte muss aufgrund der Datenmenge die Suche auf einen Server ausgelagert werden.

- Datenkomprimierung: Die Menge an Daten sollte so klein wie möglich sein. Deswegen sollten die Javascriptdateien in ein SVG-Dokument integriert werden, das diese Daten in einem Javascript Bereich enthält. SVG unterstüzt die Komprimierung mittels „gzip“. So kann dieses SVG-Dokument und damit das Ladevolumen auf 10% reduziert werden.

- Aufteilung der Rasterinformationen: Die Informationen über die bestehenden Rasterpunkte in allen Ebenen steht in einer einzigen Datei. Diese muss zu Beginn des Ladevorgangs geladen werden. Mit der Anzahl an Ebenen und Daten wird sie enorm groß. Entweder sollte diese Datenmenge daher in mehreren Dateien statisch gespeichert werden oder sie sollte nach Bedarf von einem Server zur Verfügung gestellt werden.

8.2 Mögliche Erweiterungen

Neben der Steigerung der Leistung gibt es noch eine Reihe weiterer möglicher angedachter Änderungen. Sie werden im Folgenden beschrieben:

- ESRI-Format: Das ESRI-Format ist ein weit verbreitetes Format zur Speicherung von Daten. Außerdem gibt es viele öffentlich zugängliche Karten in diesem Format. Es sollte ein Transformationsalgorithmus zur Verfügung gestellt werden, der die Daten von ESRI in das Zwischenformat konvertiert.

- Routing-Algorithmus: Bisher wurde noch kein Routing-Algorithmus implementiert. Das Ziel der Arbeit war die Schaffung einer Grundlage zur Anzeige eines Routingvorgangs. Dieser sollte mit Hilfe des Zwischenformates implementiert werden.

Kapitel 9

Literaturverzeichnis

[BAUMUC]: Baureferat München
http://www.muenchen-baut.de

[GISMNGT]: Beschreibung verschiedener Kartenformate
http://www.gismngt.de/format

[GML3]: Geography Markup Language Version 3
http://www.opengis.org/docs/02-023r4.pdf

[GMLSC]: Schemata in GML
http://www.opengis.net/gml

[SQDGISM]: Kurzbeschreibung des Austauschformats SQD für SICAD-Daten
http://www.gismngt.de/format/sqd.htm

[XLINK]: W3C - Link - Schema
http://www.w3.org/1999/xlink

Anhang A

Auszug aus PmsMap.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema targetNamespace="http://www.pms.informatik.uni-muenchen.de"
     xmlns="http://www.pms.informatik.uni-muenchen.de"
     xmlns:pms="http://www.pms.informatik.uni-muenchen.de"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:gml="http://www.opengis.net/gml"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:import namespace="http://www.opengis.net/gml"
              schemaLocation="feature.xsd"/>
  <xsd:element name="Map" type="pms:MapModelType"/>
  <xsd:element name="MapNode" type="pms:MapNodeType"/>
  <xsd:element name="MapNodeRef" type="pms:MapNodeRefType"
    substitutionGroup="gml:coord"/>
  <xsd:element name="MapRelNodeRef" type="pms:MapNodeRefType"
    substitutionGroup="gml:coord"/>
  <xsd:element name="MapNodes" type="pms:MapNodeCollectionType"/>
  <xsd:element name="MapLevels" type="pms:MapLevelCollectionType"/>
  <xsd:element name="MapLevel" type="pms:MapLevelType"/>
  <xsd:element name="MapMember" type="pms:MapMemberType"
    substitutionGroup="gml:featureMember"/>
  <xsd:element name="Image" type="pms:ImageType"
               substitutionGroup="pms:MapMember"/>
  <xsd:element name="MapGrid" type="pms:MapGridType"/>
  <xsd:element name="GridSize" type="xsd:integer" default="0"/>
  <xsd:element name="MaxViewport" type="xsd:integer" default="0"/>
  <xsd:complexType name="MapModelType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element ref="pms:MapNodes"/>
          <xsd:element ref="pms:MapLevels"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="MapNodeType">
```

```
    <xsd:complexContent>
      <xsd:extension base="gml:CoordType">
        <xsd:attribute name="nid" type="xsd:ID" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="MapNodeRefType">
    <xsd:attributeGroup ref="gml:AssociationAttributeGroup"/>
  </xsd:complexType>
  <xsd:complexType name="MapNodeCollectionType">
    <xsd:sequence>
      <xsd:element ref="pms:MapNode" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="MapLevelCollectionType">
    <xsd:sequence>
      <xsd:element ref="pms:MapLevel" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="MapLevelType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element ref="pms:MapGrid" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="MapGridType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element ref="pms:GridSize"/>
          <xsd:element ref="pms:MaxViewport"/>
          <xsd:element ref="MapMember" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="MapMemberType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureCollectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="ImageType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureCollectionBaseType">
        <xsd:attributeGroup ref="gml:AssociationAttributeGroup"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Anhang B

ConcretMap.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema targetNamespace="http://www.pms.informatik.uni-muenchen.de"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:pms="http://www.pms.informatik.uni-muenchen.de"
    xmlns="http://www.pms.informatik.uni-muenchen.de">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Contains a map for a real environment
        </xsd:documentation>
    </xsd:annotation>
    <xsd:include schemaLocation="PmsMap.xsd"/>
    <xsd:element name="Street" type="pms:StreetType"
        substitutionGroup="pms:MapMember"/>
    <xsd:element name="StreetPart" type="pms:StreetPartType"/>
    <xsd:element name="StreetName" type="pms:StreetNameType"
         substitutionGroup="pms:MapMember"/>
    <xsd:element name="Distance" type="xsd:decimal"/>
    <xsd:element name="LeftLaneWidth" type="xsd:decimal"/>
    <xsd:element name="RightLaneWidth" type="xsd:decimal"/>
    <xsd:element name="Text" type="xsd:string"/>
    <xsd:element name="FontSize" type="xsd:integer"/>
    <xsd:complexType name="StreetType">
        <xsd:complexContent>
            <xsd:extension base="gml:AbstractFeatureType">
                <xsd:sequence>
                    <xsd:element ref="StreetPart" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="StreetPartType">
        <xsd:complexContent>
            <xsd:extension base="gml:AbstractFeatureType">
                <xsd:sequence>
                    <xsd:element ref="gml:centerLineOf"/>
```

```
                    <xsd:element ref="Distance" minOccurs="0"/>
                    <xsd:element ref="LeftLaneWidth"/>
                    <xsd:element ref="RightLaneWidth" minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="StreetNameType">
        <xsd:complexContent>
            <xsd:extension base="gml:AbstractFeatureType">
                <xsd:sequence>
                    <xsd:element ref="gml:centerLineOf"/>
                    <xsd:element ref="Text"/>
                    <xsd:element ref="FontSize" minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Anhang C

Auszug aus Dynamic.html

```
<script language="javascript" src="MarkingManager.js"></script>
<script language="javascript" src="Dynamic.js"></script>
<script language="javascript" src="GridManager.js"></script>
<table>
<tr height="300">
  <td rowspan="3">
   <embed src="Dynamic.svg" id="svgEmbed" name="svgDoc"
          width="1000" height="800"/>
  </td>
  <td valign="top">
    <form name="ChooserForm" action='#'>
        <div id="DynamicChooser"> </div>
    </form>
  </td> </tr>
<tr>
 <td>
  <div id="StreetName"></div>
  <script language="javascript" src="DynamicChooser.js"></script>
  <script language="javascript" src="SelectionManager.js"></script>
 </td> </tr>
<tr>
 <td>
  <form name="SearchForm" action="javascript:searchStreet()">
   Search for: <input type=text name="StreetSearch"/>
   <input type=button name="SearchButton" value="Search" onClick="searchStreet()">
   <div id="StreetSearchResults"></div>
   <script language="javascript" src="StreetSearcher.js"></script>
   <script language="javascript" src="StreetSearch.js"></script>
  </form>
 </td>
</tr>
</table>
```

Anhang D

Dynamic.svg

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type='text/css' href='MapCreator.css'?>
<svg preserveAspectRatio="none meet" onzoom="perspectiveChange(evt)"
    onscroll="perspectiveChange(evt)" viewBox="0.0 0.0 800.0 800.0">
    <defs id="DEFS">
        <script type="text/javascript">
            SVG_DOCUMENT = document;
            createAllLevels();
        </script>
        <textPath startOffset="20.0" id="textPathTemplate" xlink:href="#Text1">
            <tspan dy="5">Template</tspan>
        </textPath>
        <image xlink:href="" id="imageTemplate" x="0" y="0" width="0" height="0" />
    </defs>
</svg>
```

Anhang E

Auszug aus Dynamic.js

```
function createLevel(ID)
{
    var level = SVG_DOCUMENT.getElementById(ID)
    if (level == null) {
      level = SVG_DOCUMENT.createElement("g");
      level.getStyle().setProperty("display", "none");
      level.setAttribute("id", ID);
      level.setAttribute('onmouseover','showElementName(evt)');
      level.setAttribute('onclick','handleClick(evt)');
      SVG_DOCUMENT.getDocumentElement().appendChild(level);
      if (HELP_LEVELS[ID] != null) createLevel(HELP_LEVELS[ID]);
    }
    return level;
}
function addTextPath(PathID, Text, fontSize)
{
  if (LAST_TEXT == null) return;
  if (TEXT_TEMPLATE_NODE == null) {
    TEXT_TEMPLATE_NODE = SVG_DOCUMENT.getElementById("textPathTemplate");
  }
  var textNode = TEXT_TEMPLATE_NODE.cloneNode(true);
  var textSpan = textNode.firstChild();
  textNode.setAttribute("xlink:href", "#"+PathID);
  textNode.setAttribute("id", "Text"+PathID);
  textNode.setAttribute("startOffset", TEXT_START_OFFSET);
  textSpan.setAttribute("dy", fontSize/2-1);
  if (fontSize > 0) textNode.getStyle().setProperty("font-size", fontSize);
  textNode.firstChild().firstChild().setData(Text);
  LAST_TEXT.appendChild(textNode);
}
function addGroup(name)
{
   if (ACTUAL_LEVEL_NODE == null) return;
   LAST_GROUP = SVG_DOCUMENT.createElement("g");
   LAST_GROUP.setAttribute("name", name);
   ACTUAL_LEVEL_NODE.appendChild(LAST_GROUP);
}
function appendPath(strokeWidth, path, ID)
```

```
{
  if (LAST_GROUP == null) return;
  var pathNode = createPathNode(strokeWidth, path, ID);
  LAST_GROUP.appendChild(pathNode);
}
function addStreet(strokeWidth, path, ID, name)
{
   if (ACTUAL_LEVEL_NODE == null) return;
   var pathNode = createPathNode(strokeWidth, path, null);
   pathNode.setAttribute("class", "streetBackground");
   ACTUAL_LEVEL_NODE.appendChild(pathNode);
   strokeWidth -= 2;
   if (strokeWidth <=0) strokeWidth =1;
   if (STREET_HELP_NODE == null) return;
   var pathNode = createPathNode(strokeWidth , path, ID);
   if (name) pathNode.setAttribute("name", name);
   pathNode.setAttribute("class", "streetForeground");
   STREET_HELP_NODE.appendChild(pathNode);
   if (MarkingManager != null) MarkingManager.styleElement(ID, pathNode);
}
function addPath(strokeWidth, path, ID, name)
{
   if (ACTUAL_LEVEL_NODE == null) return;
   var pathNode = createPathNode(strokeWidth, path, ID);
   if (name) pathNode.setAttribute("name", name);
   ACTUAL_LEVEL_NODE.appendChild(pathNode);
}
function addImage(name, x, y, width, height, ID)
{
  if (ACTUAL_LEVEL_NODE == null) return;
  if (IMAGE_TEMPLATE_NODE == null) {
    IMAGE_TEMPLATE_NODE = SVG_DOCUMENT.getElementById("imageTemplate");
  }
  var imageNode = IMAGE_TEMPLATE_NODE.cloneNode(true);
  if (ID) {
    imageNode.setAttribute("id", ID);
  }
  imageNode.setAttribute("x", x);
  imageNode.setAttribute("y", y);
  imageNode.setAttribute("width", width);
  imageNode.setAttribute("height", height);
  imageNode.setAttribute("xlink:href", name);
  ACTUAL_LEVEL_NODE.appendChild(imageNode);
}
function createPathNode(strokeWidth, path, ID)
{
  var pathNode = SVG_DOCUMENT.createElement("path");
  if (strokeWidth) {
    pathNode.setAttribute("style","stroke-width:"+strokeWidth+"px;");
  }
  if (ID) {
    pathNode.setAttribute("id", ID);
  }
  pathNode.setAttribute("d", path);
  return pathNode;
}
```

```
function loadDynamic(fileName)
{
    if (LOADED_FILES[fileName]) return;
    LOADED_FILES[fileName] = true;
    var oldViewBox = SVG_DOCUMENT.lastChild().getAttribute("viewBox");
    if (oldViewBox != VIEWBOX) {
      SVG_DOCUMENT.lastChild().setAttribute("viewBox", VIEWBOX);
    }
    showModalDialog("Loader.html",new Array(SVG_DOCUMENT, fileName,
        MarkingManager),'dialogHeight: 240 px; dialogWidth: 240 px; status: no;');
}
function handleClick(evt)
{
    var element = evt.target;
    var nameAtt = element.getAttribute("name");
    if (nameAtt.length == 0) nameAtt = element.parentNode.getAttribute("name");
    var idAtt = element.getAttribute("id");
    if (CLICK_HANDLER) CLICK_HANDLER.handleClick(idAtt, nameAtt, element);
}
function zoomToElement(id)
{
  var element = SVG_DOCUMENT.getElementById(id);
  if (!element || element.getNodeName()!="path") return;
  var type = MarkingManager.getType("Zoom","red");
  type.clear();
  type.add(id);
  type.flush();
  var svg = SVG_DOCUMENT.getDocumentElement();
  var viewboxes = VIEWBOX.split(' ');
  var ulXCorner = parseInt(viewboxes[0]);
  var ulYCorner = parseInt(viewboxes[1]);
  var width = parseInt(viewboxes[2]) - ulXCorner ;
  var height = parseInt(viewboxes[3]) - ulYCorner ;
  var pixWidth = DOCUMENT.getElementById("svgEmbed").width;
  var pixHeight = DOCUMENT.getElementById("svgEmbed").height;
  var SCALE_LEVEL = 10;
  var coords = new String(element.getAttribute("d")).split(' ');
  if (coords.length < 1) {
    alert("ilegal coords :"+element.getAttribute("d"));
    return;
  }
  var x = parseInt(coords[0].substr(1));
  var y = parseInt(coords[1]);
  var scale =  svg.getCurrentScale();
  scale = Math.max(SCALE_LEVEL, scale);
  svg.setCurrentScale(scale);
  var translate  = svg.getCurrentTranslate();
  translate.x = (ulXCorner - x) * scale * pixWidth / width + pixWidth /2;
  translate.y = (ulYCorner - y)  * scale * pixHeight / height + pixHeight /2;
}
```

Anhang F

Interface SvgWriter.java

```
package svgMap.interfaces;
import svgMap.utils.*;
import svgMap.utils.MapPath;
/**
 * Dieses Interface dient zum Erstellen des Svg-Dokuments mit Hilfe eines
 * SvgJsWriters,
 * der dynamischen Svg-Code generiert
 */
public interface SvgWriter
{
  /**
   * Erzeugt Javascript-Code mit Hilfe des JavascriptWriters
   * @param    jsWriter der SvgJsWriter, der die Javascript aufrufe generiert
   */
  public void createSvgJs(SvgJsWriter jsWriter);
}
```

Anhang G

Interface XMLizable.java

```
package svgMap.interfaces;
import svgMap.utils.*;
/**
 * Dieses Interface dient zum Lesen und Erzeugen von Xml-Dateien
 */
public interface XMLizable
{
  public String NAMESPACE = ""; //"pms:";
  public String GML_NAMESPACE = "gml:";
  /**
   * Methode readFromNode liest alle Daten aus dem angegebenen Pfad
   *
   * @param    xmlPath  der aktuelle Pfad, der die Daten enthaelt
   *
   */
  public void readFromNode(MapPath xmlPath);
  /**
   * Methode storeToNode speichert die Daten in den angegebenen Pfad
   *
   * @param    xmlPath  der aktuelle Pfad, in den die Daten gespeichert werden
   *
   * @return   liefert den Pfad, der neu angelegt wurde oder den alten Pfad
   *           (wichtig fuer Vererbung)
   *
   */
  public MapPath storeToNode(MapPath xmlPath);
}
```

Anhang H

Utility SvgJsWriter.java

```
package svgMap.utils;
import svgMap.objects.Map;
import java.io.*;
import java.util.HashMap;
import java.util.TreeSet;
import java.util.Iterator;
/**
 *  Diese Klasse nutzt die definierten Javascript-Funktionen, um ein SVG-Dokument
 *  aufzubauen
 *  Durch die vordefinierten Funktionen erleichtert es die Verwendung und bietet
 *  die Moeglichkeit des Aufteilens der Daten in verschiedene Dateien
 */
public class SvgJsWriter
{
  public final static String DEFS = "DEFS";
  private StringBuffer writer_ = new StringBuffer();
  private String name_;
  private String lastGroupName_;
  private Map map_;
  private boolean isText_;
  private String lastLevel_;
  private String viewBox_;
  private HashMap levels_ = new HashMap();
  private TreeSet levelNames_ = new TreeSet();
  private TreeSet existingFiles_ = new TreeSet();
  public SvgJsWriter(Map map)
  {
    map_ = map;
    map_.createSvgJs(this);
  }
  /**
   * Methode setLevel gibt den Level an, in den die folgenden Daten geschrieben
   * werden sollen
   *
   * @param    id        die ID des Levels
   *
   */
  public void setLevel(String id) {
    setLevel(id, Integer.MAX_VALUE, "0,0", Integer.MAX_VALUE);
```

```
}
/**
 * Methode setLevel gibt den Level an, in den die folgenden Daten geschrieben
 * werden sollen
 *
 * @param    id                  die ID des Levels
 * @param    gridSize            die Gittergroesse
 * @param    coord               die aktuelle Koordinate im Gitter
 * @param    maxViewport         der Detaillevel
 *
 */
public void setLevel(String id, int gridSize,  String coord, int maxViewport) {
  String desc = getDescription(id, gridSize, coord);
  if (!desc.equals(lastLevel_)) {
    if (!DEFS.equals(id)) prepareLevel(id, gridSize, coord, maxViewport, desc);
    writer_.append("setLevel('"+id+"',"+gridSize+","+coord+");\n");
    if (!DEFS.equals(id)) isText_ = false;
    lastLevel_ = desc;
  }
}
/**
 * Methode getDescription liefert eine eindeutige Bezeichnung
 *
 * @param    id                  die ID des Levels
 * @param    gridSize            die Gittergroesse
 * @param    coord               die aktuelle Koordinate im Gitter
 *
 * @return   Bezeichnung als String
 *
 */
public static String getDescription(String id, int gridSize,  String coord) {
  return id+" "+gridSize+" - "+coord;
}
/**
 * Methode prepareLevel bereitet ein Level zur Benutzung vor
 *
 * @param    id                  die ID des Levels
 * @param    gridSize            die Gittergroesse
 * @param    coord               die aktuelle Koordinate im Gitter
 * @param    maxViewport         der Detaillevel
 * @param    desc                Bezeichnung des Levels
 *
 */
private void prepareLevel(String id, int gridSize, String coord,
                          int maxViewport, String desc)
{
  writer_ = (StringBuffer) levels_.get(desc);
  if (writer_ == null) {
    writer_ = new StringBuffer();
    levelNames_.add(id);
    existingFiles_.add(
      "'"+id+"',"+gridSize+","+coord+","+maxViewport+",'"+desc+"'");
    levels_.put(desc, writer_);
  }
}
/**
 * Methode prepareText bereitet die Benutzung von folgendem Text vor.
 * Der Text wird an der aktuellen Stelle eingefuegt, auch wenn vorher andere
 * Objekte definiert wurden
 */
```

```
public void prepareText() {
  if (!isText_) {
    writer_.append("prepareText();\n");
    isText_ = true;
  }
}
/**
 * Methode addTextPath fuegt einen Text entlang eines Pfades ein
 *
 * @param    PathID              ID des Pfaged
 * @param    text                der Text
 * @param    fontSize            Schriftgroesse
 *
 */
public void addTextPath(String PathID, String text, int fontSize)
{
  if (!isText_) prepareText();
  writer_.append(
    "addTextPath('"+PathID+"','"+prepareScript(text)+"',"+fontSize+");\n");
}
/**
 * Methode addImage fuegt ein Bild ein
 *
 * @param    name      Name des Bildes
 * @param    coords    Koordinaten, an denen das Bild plaziert werden soll
 * @param    width     Breite des Bildes
 * @param    height    Hoehe des Bilders
 * @param    ID        ID des Bildes
 *
 */
public void addImage(String name, String coords, String width,
                     String height, String ID)
{
  writer_.append(
    "addImage('"+name+"',"+coords+","+width+","+height+",'"+ID+"');\n");
}
/**
 * Methode setViewBox setzt den aktuellen sichtbaren Bereich
 *
 * @param    viewBox             eine SVG-Viewbox als String
 *
 */
public void setViewBox(String viewBox)
{
  viewBox_ = viewBox;
}
/**
 * Methode addGroup oeffnet eine Gruppe mit dem angegebenen Namen.
 * Solange keine neue Gruppe und kein neues Level definiert wird werden alle
 * folgenden Elemente in diese Gruppe geschrieben
 *
 * @param    name                Name der Gruppe
 *
 */
public void addGroup(String name)
{
  if (name != null && !name.equals(lastGroupName_)) {
    lastGroupName_ = name;
    writer_.append("addGroup('"+name+"');\n");
    isText_ = false;
```

```
  }
}
/**
 * Methode setName definiert den defaultNamen der Gruppe
 *
 * @param    name                DefaultName der Gruppe
 *
 */
public void setName(String name)
{
  name_  = name;
}
/**
 * Methode prepareScript bereitet einen Text auf Javascriptfaehigkeit vor
 *
 * @param    script              Text der vorbereitet werden soll
 *
 * @return   Text, der Javascriptfaehig gemacht wurde
 *
 */
public static String prepareScript(String script)
{
  return script.replaceAll("'","\\'");
}
/**
 * Methode appendStreet fuegt eine Strasse hinzu
 *
 * @param    path                SVG-Pfad Angabe
 * @param    id                  ID des Pfades
 * @param    strokeWidth         Strichliniendicke
 *
 */
public void appendStreet(String path, String id, double strokeWidth)
{
  appendPath(path, id, strokeWidth, "Street");
}
/**
 * Methode appendPath fuegt einen Pfad hinzu
 *
 * @param    path                SVG-Pfad Angabe
 * @param    id                  ID des Pfades
 * @param    strokeWidth         Strichliniendicke
 *
 */
public void appendPath(String path, String id, double strokeWidth)
{
  appendPath(path, id, strokeWidth, "Path");
}
/**
 * Methode appendPath fuegt einen Pfad hinzu
 *
 * @param    path                SVG-Pfad Angabe
 * @param    id                  ID des Pfades
 * @param    strokeWidth         Strichliniendicke
 * @param    addTxt              Name des Pfades (Javascriptfunktion)
 *
 */
private void appendPath(String path, String id, double strokeWidth,
        String addTxt)
{
  if (name_ == null || name_.equals(lastGroupName_)) {
```

```
      writer_.append(
        "append"+addTxt+"("+strokeWidth+",'"+path+"','"+id+"');\n");
    } else {
      writer_.append(
        "add"+addTxt+"("+strokeWidth+",'"+path+"','"+id+"','"+name_+"');\n");
    }
  }
  /**
   * Methode writeToFile schreibt die Daten in ein Verzeichnis
   *
   * @param    directoryName       Name des Verzeichnisses
   *
   */
  public void writeToFile(String directoryName)
  {
    try {
      BufferedWriter chooserWriter = new BufferedWriter(
                        new FileWriter(directoryName+"\\DynamicChooser.js"));
      chooserWriter.write("setViewBox('"+viewBox_+"');\n");
      Iterator iter = existingFiles_.iterator();
      while (iter.hasNext()) {
        String content = (String) iter.next();
        chooserWriter.write("gridFile("+content+");\n");
      }
      chooserWriter.close();
      iter = levels_.keySet().iterator();
      while (iter.hasNext()) {
        String name = (String) iter.next();
        String content = levels_.get(name).toString();
        BufferedWriter writer = new BufferedWriter(
                      new FileWriter(directoryName+"/maps/"+name+".js"));
        writer.write(content);
        writer.write("finishLoading();");
        writer.close();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  /**
   * Methode getMap liefert die aktuelle Karte
   *
   * @return   Karte die gerade abgebildet wird
   *
   */
  public Map getMap()
  {
    return map_;
  }
}
```