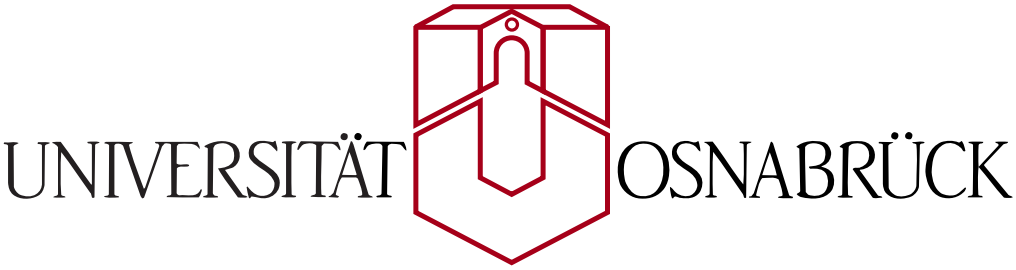


Entwicklung einer SVG Web Mapping Applikation zur Visualisierung von Geoinformationen

Diplomarbeit
von
Dorothee Langfeld

Gutachter:
Prof. Dr. Oliver Vornberger
Prof. Dr.-Ing. Manfred Ehlers

Institut für Informatik
Universität Osnabrück

17.02.2006

Vorwort

Diese Diplomarbeit entstand an der Universität Osnabrück im Institut für Informatik. Sie ist der schriftliche Teil der Diplomprüfungen für meinen Abschluss als Diplom-Mathematikerin.

Danksagung

An dieser Stelle möchte ich folgenden Personen für ihre Unterstützung bei der Erstellung dieser Diplomarbeit danken:

- Herrn Prof. Dr. Oliver Vornberger für die sehr gute Betreuung der Arbeit, seine Hinweise und Anregungen
- Herrn Prof. Dr. Manfred Ehlers für die Tätigkeit als Zweitgutachter
- Ralf Kunze für seine sehr gute Betreuung, die konstruktiven Vorschläge, das Beantworten meiner Fragen und das Korrekturlesen dieser Arbeit
- Patrick Fox für seine Bereitschaft, Fragen zu beantworten und das Korrekturlesen von Teilen dieser Arbeit
- Meiner Mutter und Tobias Schwegmann für das Korrekturlesen dieser Arbeit
- Friedhelm Hofmeyer für die Bereitstellung der benötigten Hard- und Software im Institut für Informatik

Ein ganz besonderer Dank gilt meinen Eltern, die mir das Mathematikstudium ermöglicht und mich während der Zeit unterstützt haben.

Warenzeichen

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Für viele Betriebe, Behörden oder auch Privatpersonen ist es von Interesse, Geodaten visualisieren zu können und die Grafiken im Internet zu veröffentlichen, da das Internet heutzutage immer mehr als Informationsquelle für die verschiedensten Bereiche dient. Immer häufiger werden Routenplaner online verwendet oder auf Webseiten Anfahrtsskizzen und Wegbeschreibungen veröffentlicht. Größere Geodatenportale erfreuen sich ebenfalls immer größerer Beliebtheit. Es existieren bereits unterschiedliche Applikationen im Internet, die geografische Daten visualisieren. Zu den bekanntesten zählen Google Maps[1] und Map24[2].

Die Benutzung der Karten ist jedoch zum Teil sehr eingeschränkt. So ist es bei den meisten Anwendungen nicht möglich, selbst weitere beliebige georeferenzierte Daten einzubinden beziehungsweise die Applikationen auf einem eigenen Server zu verwenden. Auch Interaktionsmöglichkeiten wie das Ein- oder Ausblenden bestimmter Datensätze existieren nur begrenzt.

Viele Visualisierungen von Geodaten basieren außerdem auf einem pixelbasierten Ansatz, wodurch die Karten nicht beliebig skalierbar sind. Zoomvorgänge erfordern das Austauschen der Datensätze und verursachen dadurch einen hohen Datentransfer sowie Wartezeiten auf Clientseite.

Aufgabenstellung

Ziel dieser Arbeit ist es, eine Web Mapping Applikation zur Visualisierung geografischer Daten mittels des Vektorgrafikformats SVG zu entwickeln. Dabei sollen die Vorteile dieses Formates bezüglich der Interaktionsmöglichkeiten des Betrachters mit dem Kartenmaterial und die freie Skalierbarkeit ausgenutzt werden. Weiterhin soll es die Möglichkeit geben, die Applikation mit beliebigen geografischen Daten zu erstellen. Jeder, der an der Visualisierung geografischer Daten interessiert ist, soll ein möglichst einfaches Werkzeug zur Verfügung gestellt bekommen, die gewünschten Daten in hoher Qualität darzustellen und der Öffentlichkeit zugänglich zu machen.

Aufbau der Arbeit

Diese Diplomarbeit ist in drei wesentliche Teile gegliedert. Der erste Abschnitt behandelt die Grundlagen, im zweiten wird die Realisierung der Aufgabenstellung vorgestellt. Im letzten Abschnitt folgen Ausblick und Zusammenfassung.

Im ersten Teil wird in Kapitel 2 die Extensible Markup Language (XML) vorgestellt und ihre Arbeitsweise und Einsatzgebiete erläutert. XML ist ein grundlegendes Datenformat für diese Arbeit. Im Rahmen dieser Arbeit sind vor allem das Verarbeiten von

[1] maps.google.com
[2] www.map24.com

XML-Dokumenten, die Manipulation zur Laufzeit sowie das Verwenden von XML als Datenaustauschformat von Bedeutung. In diesem Zusammenhang wird das Document Object Model (DOM) vorgestellt, außerdem Parser, die für die Dokumentverarbeitung zuständig sind. Kapitel 3 beschäftigt sich mit Scalable Vector Graphics (SVG), dem dieser Arbeit zu Grunde liegenden Grafikformat. Es werden Aufbau und Grundelemente von SVG-Dateien beschrieben, insbesondere wird auf Eigenschaften eingegangen, die für eine Web Mapping Applikation wichtig sind. Die beiden darauf folgenden Kapitel gehen auf die verwendeten Skriptsprachen ein. Kapitel 4 beschreibt die Möglichkeiten, den DOM eines XML-Dokumentes, insbesondere eines SVG-Dokumentes, mit ECMA-Script zu manipulieren und so Interaktionen mit dem Betrachter zu ermöglichen. Einen kurzen Einblick in das serverseitig ausgeführte PHP-Skript bietet Kapitel 5. Dabei wird auf die Verwendung eines DOM in dieser Sprache eingegangen. Kapitel 6 gibt einen kurzen Überblick über Shapefiles, das in der Arbeit verwendete Geodatenformat. Der erste Teil der Arbeit wird mit allgemeinen Informationen zum Konzept verschiedener Geographischer Informationssysteme (GIS) abgeschlossen (Kapitel 7).

Der zweite Abschnitt beginnt mit Kapitel 8, in dem die entstandene Web Mapping Applikation beschrieben wird. Dabei wird der Aufbau sowie der Datenaustausch zwischen Server und Client näher erläutert. Auf den Umgang mit dem Kartenmaterial wird ebenfalls eingegangen. Wie es möglich gemacht wird, beliebige Geodaten in der Applikation zu visualisieren, beschreibt Kapitel 9. Hier wird ein Überblick über das Java-Programm gegeben, mit dem sowohl das Kartenmaterial aus den Shapefiles generiert wird als auch Teile der Skripte erzeugt werden. Ein weiterer Schwerpunkt dieses Kapitels ist die Beschreibung der Konfigurationsdatei, die die Grundlage aller Web Mapping Applikationen bildet, die mit dem Programm erzeugt werden. Zwei konkrete Anwendungen werden im darauf folgenden Kapitel 10 vorgestellt, um die Leistungsfähigkeit des entwickelten Tools aufzuzeigen.

Im letzten Teil wird ein Ausblick gegeben, wie die entwickelte Software erweitert werden kann (Kapitel 11). Ergebnisse und Erfahrungen, die beim Erstellen der Arbeit gemacht wurden, sind in Kapitel 12 zusammengefasst.

Der Anhang enthält Teile der verwendeten Skripte sowie die Schemabeschreibung der Konfigurationsdatei.

I Grundlagen

2 eXtensible Markup Language

Die *eXtensible Markup Language (XML)* ist ein plattformunabhängiger Standard, der 1998 vom W3C verabschiedet wurde [W3C2005b]. Sie entstand als Nachfolger von SGML, der Standard Generalized Markup Language, die in den 70er Jahren von Charles F. Goldfarb, Ed Mosher und Ray Lorie bei IBM erfunden wurde [HaMe2005]. XML wurde von vielen Entwicklern weiter vervollständigt und 1986 als ISO-Standard 8879 angenommen. SGML ist eine semantische und strukturelle Markup-Sprache für Textdokumente, die unter anderem Anwendung als HTML fand. HTML selbst ist sehr speziell und nur für Webseiten nutzbar. SGML selbst lässt sich für allgemeine Anwendungen nutzen, ist jedoch kompliziert und umfasst viele Spezialfälle. Seit 1996 entwickelten Jon Bosak, Tim Bray, C. M. Sperberg-McQueen, James Clark und andere eine Sprache, die die Vorteile und Leistungsfähigkeit von SGML behalten, aber auf überflüssige und redundante Funktionen verzichten sollte. Das Ergebnis war 1998 XML 1.0. Es ist eine ebenfalls strukturelle Markup-Sprache, die aber weitaus weniger komplex ist. XML dient als Metasprache dazu, eigene Auszeichnungssprachen und Dokumenttypen zu beschreiben. Sie stellt Vorschriften bereit, um eine beliebige Anzahl konkreter Auszeichnungssprachen für die verschiedensten Arten von Dokumenten zu definieren, also eine Struktur, mit der man über so genannte *Tags* Elemente definieren kann. Dabei ist nicht vorher festgelegt, welche Elemente zur Verfügung stehen, daher stammt die Bezeichnung der Sprache als *extensible*. XML wurde und wird weiter entwickelt, so entstanden Technologien wie *XSL*, *XSLT*, *XLink*, *XPointer* und viele mehr. Zusätzlich wurde der Gebrauch von *Namespaces (Namensräume)* eingeführt [HaMe2005]. Namensräume werden dazu verwendet, Konflikte bei der Namensvergabe zu verhindern. Mit Hilfe von Namensräumen kann ein Autor große Programmpakete mit vielen definierten Namen schreiben, ohne sich Gedanken machen zu müssen, ob die neu eingeführten Namen in Konflikt zu anderen Namen stehen. Solch ein Konflikt tritt häufig bei der Verwendung von mehreren XML-Ausprägungen auf. Zur eindeutigen Zuordnung in XML gibt es ein Namensraum Präfix, das durch einen Doppelpunkt vom lokalen Namen getrennt wird [W3C2005b]. Weiterhin wurden Interfaces entwickelt, damit das Verarbeiten von XML-Dokumenten standardisiert werden konnte, so entstand die *Simple API for XML (SAX)*. SAX2 wurde 2000 veröffentlicht. Eine weitere Standardisierung stellt das *Document Object Model* dar, das in diesem Kapitel noch näher erläutert wird. Im Jahre 2004 wurde XML 1.1 veröffentlicht, es bietet im Vergleich zur Version 1.0 jedoch keine relevanten Neuheiten für Entwickler. In dieser Arbeit werden wesentliche Stärken von XML ausgenutzt: die einfache Syntax, die Möglichkeit, komplexe Datenstrukturen einfach zu beschreiben, die leichte Fehlersuche durch einfaches Auswerten und die Unabhängigkeit von Programmiersprachen und Betriebssystemen.

2.1 Das Document Object Model

Eine XML-Datei besteht aus einer Menge von Elementen mit ihren Attributen und ihrem Inhalt. Eine Strukturierung erfolgt dadurch, dass Elemente andere Elemente enthalten können. So kann man den Aufbau eines XML-Dokumentes als Baumstruktur auffassen (Abbildung 2.1). Um XML-Dokumente verarbeiten zu können ist es sinnvoll, eine

Schnittstelle zur XML-Datenstruktur zu definieren, welche die logische Abarbeitung des Dokumentes als Baum ermöglicht. Eine Vereinbarung über solch eine Schnittstelle ist das Document Object Model (DOM) vom W3C [W3C2005g].

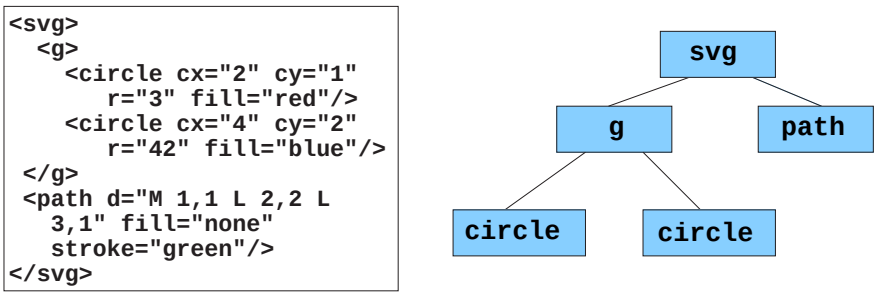


Abbildung 2.1: Baumstruktur eines SVG-Dokumentes

Das DOM ist ein Modell, das die Baumstruktur des Dokumentes in einer Objekthierarchie, dem so genannten *DOM-Tree*, abbildet. Dabei bieten die Objekte genau definierte Schnittstellen für unterschiedliche Objekttypen an, die das Auslesen und Verändern von Objektinhalten und -hierarchien erlauben. Die DOM-Spezifikation des W3C definiert diese Interfaces plattformunabhängig. Fast alle Interfaces für den jeweiligen Objekttyp stammen vom `Node`-Interface ab, das einen einzelnen Baumknoten repräsentiert und Methoden zur Navigation in der Objekthierarchie bietet. Der Aufbau des Objektmodells ist bestimmten Regeln unterworfen, die garantieren, dass nur *wohlgeformte*, also syntaktisch korrekte XML-Dokumente dargestellt werden können. So ist für jeden Objekttyp festgelegt, welche anderen Objekttypen die untergeordneten Bestandteile bilden dürfen. In der DOM-Spezifikation ist aber nicht nur die Struktur eines XML-Dokuments festgelegt, sie enthält außerdem umfangreiche Vereinbarungen, wie man einen DOM-Tree verarbeiten und manipulieren kann. Über diese Schnittstelle ist es möglich, sehr variabel auf einzelne Elemente des DOM-Trees zuzugreifen und ihn dynamisch zu verändern. Daher ist die Verwendung des DOM für interaktive Anwendungen sehr gut geeignet ist. Viele Programmier- und Skriptsprachen implementieren diese Schnittstelle. Dadurch ist es möglich, XML-Dokumente nicht nur vor, sondern auch dynamisch während der Verwendung zu verändern. Seit Java 5 ist eine Implementation der Interfaces in der Laufzeitumgebung von Java enthalten, es wird die DOM-Implementation des Xerces2 Java Parser der Apache Foundation [Apac2004c] verwendet. In der vorliegenden Arbeit wird das DOM-Interface sowohl für die Erzeugung von SVG (Kapitel 2.4) als auch für die interaktive Web Mapping Applikation verwendet. Hier findet ECMAScript zum Manipulieren des SVG-DOM Verwendung (Kapitel 4.1).

2.2 Documenttyp

XML-Anwendungen sind meist für ein bestimmtes Problem entwickelt. Es wünschenswert, dass in XML-Dokumenten, die in solch einer Anwendung verarbeitet werden, nur bestimmte Elemente enthalten sind, also die Struktur des Dokumentes klar definiert ist.

Dann kann sich das Programm darauf verlassen, dass das Dokument entsprechend einer vereinbarten Struktur aufgebaut ist, und nur dann ist eine reibungslose Verarbeitung möglich. Die zur Verarbeitung nötige Überprüfung anhand einer solchen Strukturbeschreibung wird als *Validierung* bezeichnet, ein so genannter Parser (Kapitel2.3) übernimmt diese Aufgabe. Für die Festlegung der Struktur eines XML-Dokumentes, auch *Dokumenttyp* genannt, existieren zwei Ansätze. Entweder erfolgt die Definition durch eine *Document Type Definition (DTD)* oder durch ein *XML-Schema* [W3C2005h].

Die Syntax und Semantik für eine DTD sind innerhalb des XML-Standards festgelegt und beschreibt grundlegende Strukturregeln für Dokumente. XML-Schema wurde als eigener Standard entwickelt und enthält eine formale Beschreibung dessen, was ein gültiges XML-Dokument ausmacht. Beide definieren neben den möglichen Element- und Attributnamen auch, in welcher Form die Elemente ineinander verschachtelt werden können, sie bieten jedoch unterschiedliche Vor- und Nachteile, die im Folgenden betrachtet werden.

2.2.1 DTD

DTDs ermöglichen die grundlegende Validierung von XML-Dokumenten in Hinblick auf die Verschachtelung von Elementen, bedingte Häufigkeitsbeschränkungen für Elemente, zulässige Attribute und Attributtypen sowie Default-Werte. Sie bieten keine Kontrolle über das Format der Werte und die Datentypen von Element- und Attributwerten. Der Inhalt von Elementen und Attributen ist durch verschiedene spezielle Attributtypen (`ID, ENTITY,` etc.) nur begrenzt einschränkbar, weitere Möglichkeiten gibt es nicht. Ist festgelegt, dass ein Element oder Attribut Zeichendaten enthält, kann keine weitere Festlegung der Länge, des Typs oder des Formates stattfinden. In einer DTD lässt sich die Anzahl der Kindelemente nur begrenzt einschränken, beispielsweise auf `0`, `1` oder `*` (viele). Explizite Zahlwerte lassen sich nicht angeben.

Soll ein Dokument gegen die DTD `beispiel.dtd` validiert werden, so muss man dies im Prolog (alles was vor dem Wurzelelement erscheint) des Dokumentes als *Dokumenttyp-Deklaration* hinter der XML-Deklaration angeben.

```
<?xml version="1.0" ?>
<!DOCTYPE wurzel SYSTEM "beispiel.dtd">
<wurzel/>
```

Die Deklaration gibt an, dass das Wurzelelement `wurzel` heißt und dass die DTD im gleichen Verzeichnis unter `beispiel.dtd` zu finden ist.

2.2.2 XML Schema

Für bestimmte Arten von Dokumenten wie Webseiten und Ähnliches sind die Kontrollmöglichkeiten, die eine DTD bietet, sicherlich ausreichend. Durch die zunehmende Verwendung von XML für eher datensatzähnliche Dokumente wird jedoch eine exakte Kontrolle über Textinhalte von Elementen und Attributen notwendig. Ein XML-Schema

bietet diese Möglichkeiten, ist jedoch auch wesentlich komplexer als eine DTD. Der XML-Schema-Standard vom W3C umfasst Funktionalitäten wie einfache und komplexe Datentypen, Ableitung und Vererbung von Typen, Häufigkeitsbeschränkungen für Elemente und namensraumsensitive Element- und Attributdeklarationen. Wichtige Unterschiede zu DTDs sind die Ergänzung von einfachen Datentypen für geparste Zeichendaten und Attributwerte sowie die Möglichkeit, die Anzahl der Kindelemente auf jeden beliebigen Zahlwert oder Zahlwertbereich festzulegen. So können mit einem Schema viel strengere Regeln für die Struktur eines XML-Dokumentes festgelegt werden. Die Schema-Sprache enthält spracheigene einfache Datentypen wie `string`, `integer`, `double`, `dateTime`, von denen man neue Typen ableiten kann. Generell ist es möglich Typen zu definieren und durch Einschränkungen oder Erweiterungen wieder zu verwenden. Ein großer Vorteil von XML-Schema ist außerdem, dass es im Gegensatz zu einer DTD in XML formuliert ist. Ob man eine DTD oder ein XML-Schema für eine bestimmte Anwendung verwenden sollte, ist von Fall zu Fall zu entscheiden. Je nach Art und Umfang der Anwendung kann es sinnvoll sein, trotz vieler Vorteile von Schema eine DTD zu verwenden. Zum Beispiel kann die Beschreibung einfacher Strukturen mit einem Schema sehr komplex werden.

Das W3C hat eine eigene Schema-Sprache entwickelt, das *W3C XML Schema*, es gibt aber eine Vielzahl anderer XML-Schema Sprachen. Wie genau man eine DTD oder ein Schema mit der Schemasprache des W3C schreibt und welche vielfältigen Möglichkeiten von Typen und Einschränkungen existieren, findet man in [HaMe2005]. Ein konkretes Schema-Beispiel wird in Kapitel 9.5 genauer erläutert.

Ein XML-Dokument, das durch ein Schema beschrieben wird, nennt man *Instanzdokument*. Im Wurzelelement gibt man an, dass es sich um eine Schema-Instanz handelt und gegen welches Schema die Validierung erfolgen soll. Das Schema ist hier in der Datei `beispiel.xsd` definiert.

```
<?xml version="1.0" ?>
<wurzel
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="beispiel.xsd"/>
```

2.3 XML-Verarbeitung

Es gibt für die Verarbeitung von XML mehrere gängige Modelle. Will man ein XML-Dokument einlesen und verarbeiten, sind die beiden grundlegenden Modelle die *ereignisorientierte* und die *baumbasierte* Verarbeitung. Das Lesen des Dokumentes übernimmt in beiden Fällen ein Parser. Das Einlesen von XML ist relativ unkompliziert, da es durch die Auftrennung von Struktur (Tags) und Inhalt eine einfache Syntax besitzt. Jeder Parser überprüft beim Lesen, ob das Dokument wohlgeformt ist. Wenn es gewünscht ist, kann ein Parser ein Dokument dahingehend überprüfen, ob es bezüglich einer DTD oder eines Schemas *valide*, also gültig ist.

Bei der ereignisorientierten Verarbeitung läuft der Parser das Dokument vom Anfang bis zum Ende ab und meldet direkt während der Verarbeitung alle auftretenden Ereignisse an

das aufrufende Programm. Solche Ereignisse sind zum Beispiel das Lesen von Start-Tags, Elementinhalten oder End-Tags. Die auftretenden Ereignisse müssen direkt verarbeitet werden, weil dem Parser nur jeweils das neu erzeugte Ereignis zur Verfügung steht. Da der Parser sequenziell durch das XML-Dokument läuft und jeweils nur einen kleinen Teil betrachtet, ist er sehr schnell und benötigt wenig Speicherplatz.

Im Gegensatz zur sequenziellen Abarbeitung des Dokumentes steht die baumbasierte Verarbeitung. Bei dieser Verarbeitung muss das gesamte Dokument vollständig eingelesen werden und im Arbeitsspeicher wird die Baumstruktur des Dokumentes nachgebaut. So kann man, anstatt auf einen Strom von Ereignissen zu reagieren, durch den Baum navigieren und die gewünschten Teile des Dokumentes finden. Grundlegender Vorteil dieses Konzepts im Vergleich zur ereignisorientierten Verarbeitung ist, dass zu jeder Zeit das komplette Dokument zur Verfügung steht, so dass man es leicht verändern oder Teile im Dokument verschieben kann. Die vollständige Kontextinformation zu einem beliebigen Ausschnitt des Dokuments ist jederzeit verfügbar. In Hinblick auf Geschwindigkeit und Effizienz ist die baumbasierte Methode jedoch unterlegen. Baummodelle können große Mengen Speicher belegen und der Zugriff auf das DOM ist recht langsam, da die Elemente und Attribute selbst speicherintern als Objekte in einer Baumstruktur vorgehalten werden. Außerdem muss das komplette Dokument eingelesen sein, bevor eine weitere Verarbeitung stattfinden kann.

2.3.1 Parser

In Java kann ein Parser über die *Simple API for XML (SAX)* [SAX2004] angesprochen werden, die im Rahmen der XML-DEV Mailingliste entwickelt wurde. SAX ist kein vom W3C definierter Standard, stellt jedoch einen de-facto Standard zur Verarbeitung von XML-Dokumenten in Java dar. Ein XML-Parser, der SAX implementiert, ist seit Version 1.4 in Java enthalten. Der Parser stellt Möglichkeiten der Fehlerbehandlung zur Verfügung, falls ein nicht wohlgeformtes oder nicht valides Dokument verarbeitet wird. Um auf Fehler reagieren zu können, muss man beim Parser einen `ErrorHandler` anmelden, der ein Interface mit folgenden Methoden implemetieren muss:

```
void warning(SAXParseException e);
void error(SAXParseException e);
void fatalError(SAXParseException e);
```

Sobald der Parser eine Zeichenfolge gefunden hat, die nicht dem gültigen syntaktischen Aufbau einer XML-Datei entspricht, liegt ein `FatalError` vor. Zu einer `Warning` oder einem `Error` kommt es, wenn der Aufbau des Dokumentes zwar syntaktisch korrekt ist, aber nicht der DTD oder dem Schema entspricht. Die Art des Fehlers bestimmt somit, welche Methode letztendlich aufgerufen wird. Einen Überblick über die unterschiedlichen Fehlertypen gibt die XML-Spezifikation [W3C2005b]. Einem Parser, der ein Baummodell erstellt, liegt ein ereignisorientierter Parser zu Grunde. Wie man in Java einen gegen ein Schema validierenden Parser verwendet, der ein Baummodell des Dokumentes erstellt, zeigt Quellcode 2.1. Hier wird der `DOMParser` verwendet, der ebenso

```
public class MyDOMParser extends DOMParser {
  public MyDOMParser() {
    try {
      setFeature("http://xml.org/sax/features/"
                 +"validation", true);
      setFeature("http://apache.org/xml/features/validation/"
                 +"schema", true);
    }
    catch (SAXNotRecognizedException e) {
      e.printStackTrace();
    }
    catch (SAXNotSupportedException e) {
      e.printStackTrace();
    }
    setErrorHandler(new ConcreteErrorHandler());
  }

  public static void main(String [] arg){
    DOMParser parser=new MyDOMParser();
    parser.parse("beispiel.xml");
    Document doc=parser.getDocument();
  }
}
```

Quellcode 2.1: Verwendung eines DOMParsers in Java

wie der `SAXParser` in der Laufzeitumgebung von Java enthalten ist. Ein Parser überprüft grundsätzlich die Wohlgeformtheit eines Dokumentes. Soll eine Validierung stattfinden, muss man diese erst anschalten, indem man das entsprechende Feature des Parsers setzt. Im Fall der Validierung gegen ein Schema, müssen zwei Features gesetzt werden. Zum einen muss man veranlassen, dass der Parser überhaupt validiert, zum anderen, dass er explizit gegen ein Schema validiert. Im Beispiel (Quellcode 2.1) wird außerdem noch ein eigener `ErrorHandler` angemeldet. In der `main-` Methode wird das Dokument „beispiel.xml“ geparst, wozu der *URI* [3] der Datei als Parameter der `parse`-Methode übergeben wird. Mit dem Aufruf von `getDocument()` am Parser erhält man einen Verweis auf das `Document`-Objekt aus dem Speicher. Mit diesem Objekt kann jetzt die weitere Verarbeitung stattfinden. Das erzeugte Baummodell genügt, wie schon der Name des Parsers sagt, der DOM-Spezifikation des W3C (Kapitel 2.1).

2.4 DOM-Erzeugung

XML-Dokumente lassen sich auf verschiedene Arten und Weisen erzeugen. Die einfachste Variante ist das Schreiben einer Datei in einem Texteditor. Will man XML aus einem

[3] Der URI (Uniform Resource Identifier) ist eine Zeichenfolge, die zur Identifizierung einer abstrakten oder physikalischen Ressource dient.

Programm heraus automatisch generieren lassen, so ließe sich das mit einer Aneinanderreihung von Strings verwirklichen. Dabei wäre aber eine Verschachtelung von Elementen sehr kompliziert und man müsste dazu selbst die Baumstruktur des Dokumentes im Speicher nachbilden. Eine fertige komfortable Lösung steht mit dem DOM-Interface (Kapitel 2.1) zur Verfügung. Es bietet nicht nur die Möglichkeit der einfachen Verarbeitung von XML-Dokumenten, sondern erleichtert auch das Erzeugen von XML. Da in der vorliegenden Arbeit diese Technik für SVG-Dokumente angewandt wird, beziehen sich die folgenden Ausführungen auf SVG-Dokumente, die generelle Technik erfolgt aber bei allen anderen XML-Formaten analog.

Viele Programmiersprachen implementieren das DOM-Interface, so auch Java, PHP und ECMAScript, die in dieser Arbeit verwendet werden. Die grundlegenden Methoden- bzw. Funktionsaufrufe sind die gleichen, daher wird die Technik hier anhand von Java erläutert. In Kapitel 5 werden Informationen zum DOM in PHP gegeben. Im Folgenden wird gezeigt, wie man ein SVG-Dokument als DOM in Java erzeugt, Elemente einfügt und das Dokument schreibt.

Um das Dokument zu erzeugen, ist eine `DOMImplementation` nötig, die zunächst den Dokument-Typ (Kapitel 2.2) als `DocumentType`-Objekt erzeugt. Mit diesem kann das `DOMImplementation`-Objekt anschließend das `Document` erzeugen, auf welches über `doc.getDocumentElement()` zugegriffen werden kann.

```
DOMImplementation impl = DOMImplementationImpl.
                  getDOMImplementation();
DocumentType docType = impl.createDocumentType("svg",
             "-//W3C//DTD SVG 1.1//EN","http://www.w3.org"
             +"/Graphics/SVG/1.1/DTD/svg11.dtd");
Document doc = impl.createDocument("http://www.w3c.org/"
         +"2000/svg","svg", docType);
Element svg = doc.getDocumentElement();
```

Mit der Methode `createElement(elementname)` kann man das nun vorhandene `Document`-Objekt dazu auffordern, SVG-Elemente zu erzeugen. An diesen Elementen werden wie auch bei der DOM-Manipulation (Kapitel 4.1) Attribute gesetzt. Es genügt nicht, das Element vom Dokument erzeugen zu lassen. Es muss an das Wurzelelement (oder an ein anderes untergeordnetes Element, wie zum Beispiel eine Gruppe) durch Aufruf der Methode `appendChild(child)` angehängt werden. Sonst existiert das Objekt zwar, ist aber noch nicht in den DOM-Tree eingefügt.

Um zum Beispiel einen roten Kreis im Urspung mit Radius `10` zu erzeugen, ist folgender Code notwendig:

```
Element e=doc.createElement("circle");
e.setAttribute("cx","0");
e.setAttribute("cy","0");
e.setAttribute("fill","red");
e.setAttribute("r","10");
e.setAttribute("stroke","none");
svg.appendChild(e);
```

Um das Dokument schreiben zu können, ist ein `OutputStream` oder ein `Writer` notwendig. Im folgenden Beispiel wird das Dokument in eine Datei geschrieben. Zum Schreiben verwendet man einen `XML-Serializer`, dem man ein `OutputFormat` mitgibt, welches festlegt, welches Encoding verwendet wird. Gibt man im dritten Parameter `true` an, so ist die Ausgabe in der Datei sinnvoll formatiert (Einrückungen, Zeilenumbrüche etc.). Der Serializer schreibt das Dokument mit Hilfe des `Writer`s in eine SVG-Datei.

```
String uri = "output.svg";
OutputStream writer= new FileOutputStream(uri);
OutputFormat of = new OutputFormat(doc, "utf-8", true);
XMLSerializer out = new XMLSerializer(writer, of);
out.serialize(doc);
writer.close();
```

Die so erstellte Datei `output.svg` hat folgenden Inhalt:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
 "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg>
  <circle cx="0" cy="0" fill="red" r="10" stroke="none"/>
</svg>
```

3 Scalable Vector Graphics

Scalable Vector Graphics (SVG) ist eine in XML formulierte Sprache zur Beschreibung von 2D-Grafiken mittels Vektordaten und wurde 2003 vom W3C [W3C2005a] in der Version 1.1 als Recommendation verabschiedet. Sie beschreibt beliebig skalierbare und animierbare Vektorbilder, ermöglicht das Einbetten von Sound und weist in vielen Punkten starke Ähnlichkeit mit dem propietären *Flashformat* von Macromedia [Mac2005] auf. SVG hat aber den entscheidenden Vorteil, dass es vollständig auf dem leicht lesbaren, gut zu verarbeitenden und weit verbreiteten XML-Format basiert (Kapitel 2) und so leichter erstellbar ist. Zudem ist es im Gegensatz zu Flash ein offener Standard.

Die SVG-Version 1.2 befindet sich momentan in der Entwicklung, es existieren aber bereits SVG-Versionen für mobile Geräte wie SVG Tiny für Handys und SVG Basic für PDAs. Da hier nur begrenzte Ressourcen zur Verfügung stehen, sind diese Versionen auf die grundlegenden Eigenschaften von SVG reduziert worden.

Als Grafikformat für das Internet bietet SVG in seinem vollen Umfang viele Vorteile. Auf Grund der Standardisierung des W3C ist es öffentlich frei nutzbar und es steht in Einklang mit anderen Standards wie DOM und ECMAScript. Hierdurch lassen sich SVG-Dokumente auch zur Laufzeit dynamisch verändern und bieten eine Vielzahl an Interaktionsmöglichkeiten (Kapitel 4.1).

Eine Vektorgrafik wird aus geometrischen Objekten zusammengesetzt, wodurch sie sich wesentlich von Rastergrafiken unterscheidet, deren Bildinformationen pixelweise gespeichert werden. Zur Beschreibung einer Vektorgrafik benötigt man lediglich eine mathematische Beschreibung der geometrischen Objekte mit Hilfe ihrer Koordinaten (Vektordaten). Ein Rechteck wird zum Beispiel über seinen linken oberen Eckpunkt, seine Höhe und Breite definiert. Für die Darstellung kommen dann nur noch Layout-Beschreibungen hinzu, wie Form und Farbe der Begrenzungslinie und die Füllfarbe. Ein Vorteil, der aus diesem Ansatz resultiert, ist eine geringe Dateigröße, solange man keine Grafiken mit sehr vielen verschiedenen Objekten vorliegen hat. So ließen sich Rastergrafiken nicht sinnvoll im Vektorgrafikformat darstellen, weil die Beschreibung dann pixelgenau erfolgen müsste. Für jedes Pixel müsste ein eigenes Objekt angelegt werden, was zu einem erheblichen Mehraufwand an Speicher führen würde.

Ein weiterer großer Vorteil von Vektorgrafiken ist, dass sie stufenlos skalierbar sind und so in jeder Zoomstufe eine gleich bleibende Auflösung bieten (Abbildung 3.1). Möglich wird dies durch die Neuberechnung der Grafik für jeden Zoomvorgang, die mit Hilfe der mathematischen Beschreibung der grafischen Objekte stattfindet. Treppeneffekte, die beim Skalieren von Rastergrafiken entstehen, lassen sich so vollständig vermeiden. Für eine serverbasierte Applikation bietet die Verwendung von Vektorgrafiken den Vorteil, dass die übertragenen Datenmengen relativ klein gehalten werden können. Ändert sich bei einer Anzeige das Bild, kommen beispielsweise Daten hinzu oder soll nur ein Ausschnitt betrachtet werden, muss bei Rastergrafiken die komplette Grafik durch eine andere ersetzt werden. Im Gegensatz dazu lassen sich bei Vektorgrafiken durch die freie Skalierbarkeit lediglich die neu hinzu kommenden Details dazu laden.

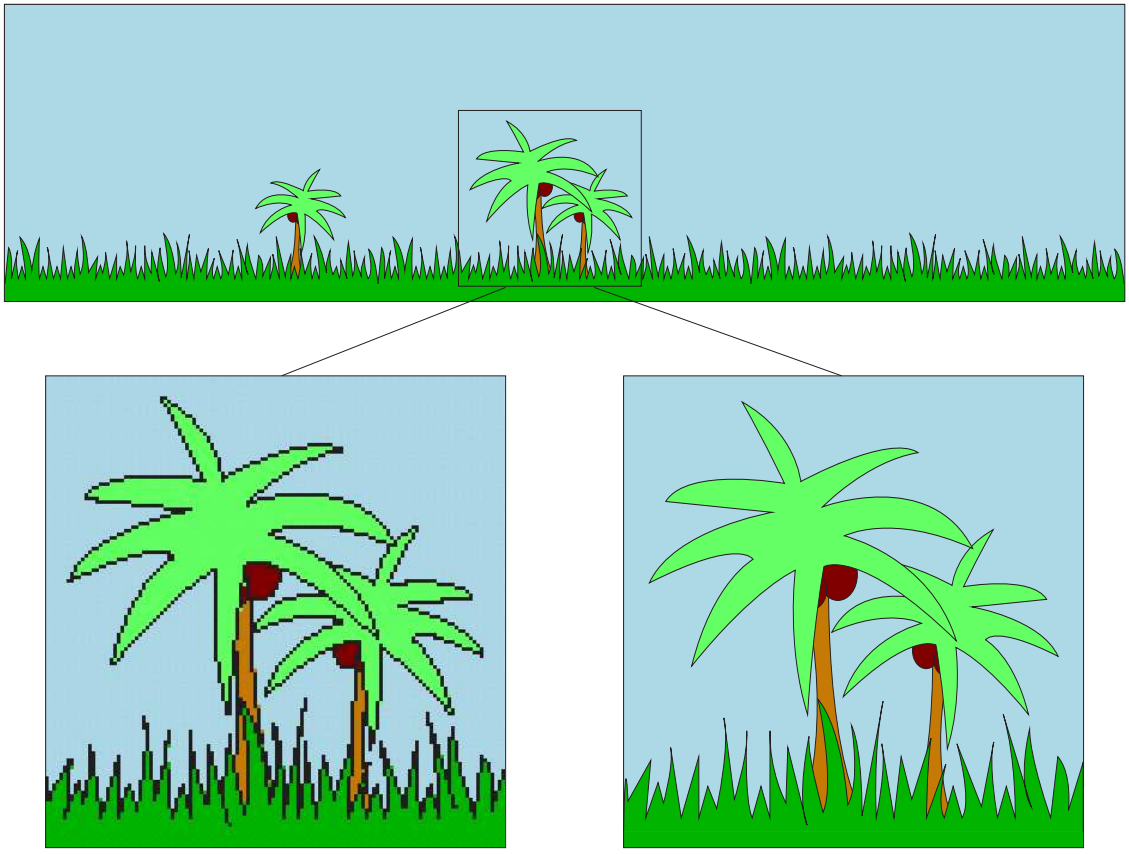


Abbildung 3.1: Pixel- versus Vektorgrafik

Um SVG betrachten zu können, benötigt man Viewer-Programme, deren Funktionsumfang und Entwicklung in Kapitel 3.2 erläutert werden. Zunächst wird im Folgenden anhand von einigen Beispielen ein sehr kurzer Einblick in den Funktionsumfang von SVG gegeben. Im Wesentlichen werden spezielle Grundformen und Eigenschaften vorgestellt, die in der Web Mapping Applikation verwendet wurden.

3.1 SVG-Gerüst und Elemente

Eine SVG-Datei enthält als XML-Dokument als erstes den XML-Prolog mit Verweis auf die DTD von SVG. Das `svg`-Tag ist das Wurzelelement, in dem mit den Attributen `width` und `height` die Größe der Grafik angegeben werden kann (Quellcode 3.1). Außerdem können Angaben über Namespaces angegeben werden, wie in späteren Beispielen zu sehen sein wird.

Es gibt die Möglichkeit, ein `defs`-Element innerhalb des Wurzelelementes einzufügen. Das `defs`-Element kann beispielsweise ECMAScript für die Interaktion mit dem Benutzer und *CSS*-Angaben zur Formatierung enthalten. ECMAScript und CSS sind selber kein XML und müssen daher in einem `CDATA`-Bereich eingeschlossen werden. Ein Bei-

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
 "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100%" height="100%">
</svg>
```

Quellcode 3.1: Grundgerüst eines SVG-Dokumentes

spiel dazu befindet sich in Kapitel 4.1. Außerdem können im `defs`-Element Prototypen von SVG-Objekten definiert werden (Kapitel 3.1.3).

Das Wurzelelement kann eine Vielzahl von Objekten, wie Kreise, Rechtecke, Ellipsen, Polygone, Polylinien, Text und Text entlang von Pfaden enthalten. Alle Objekte werden über ihre Vektorkoordinaten definiert. Das Koordinatensystem in SVG hat seinen Ursprung in der linken oberen Ecke, die x-Koordinaten werden nach rechts, die y-Koordinaten nach unten angegeben. Bei allen Formen gibt man im Tagnamen an, was für ein Objekt man erzeugen möchte, innerhalb des Tags werden die entsprechenden Attribute für das Layout angegeben. Alle Elemente können über das `g`-Tag gruppiert werden, was den Vorteil bietet, dass `style`-Eigenschaften oder Transformationen (Skalieren, Translatieren) auf mehrere Elemente gleichzeitig angewandt werden können.

Für eine Web Mapping Applikation sind nicht alle in SVG verfügbaren Grundformen wichtig. Entscheidende Formen sind:

1. **Polylinien** zur Darstellung von Liniendaten (Flüsse oder Strassen)
2. **Polygone** zur Visualisierung von Flächen (Grundstücke, Seen)
3. **Kreise** zur Anzeige von Punktdaten (Städte, Points of Interests)
4. **Text** für die Beschriftung, evtl. entlang von Pfaden definiert

3.1.1 Linien

In SVG gibt es zwei Möglichkeiten Polylinien zu definieren. Die einfachste Möglichkeit ist, das `polyline`-Tag zu verwenden. Eine weitere Möglichkeit der Definition von Polylinien ist die Verwendung des `path`-Elementes. In beiden Fällen gibt man über ein Attribut die Punkte an, die die Linie beschreiben. Ein Unterschied besteht darin, dass beim `path` angegeben werden kann, ob die Punkte verbunden werden sollen (`L` oder `l`) oder ob zum jeweiligen Punkt „gesprungen“ werden soll (`M` oder `m`). Die Groß- und Kleinschreibung gibt an, ob es sich um relative oder absolute Koordinatenangaben handelt (Abbildung 3.2).

Ein weiterer wichtiger Aspekt ist die Darstellung von Liniendaten mit besonderer Bedeutung, wie Eisenbahnlinien oder Autobahnen. Eine Linie in SVG hat nur eine Farbe. Die einzigen Möglichkeiten, die eine Linie selbst zum Verändern des Layouts bietet,

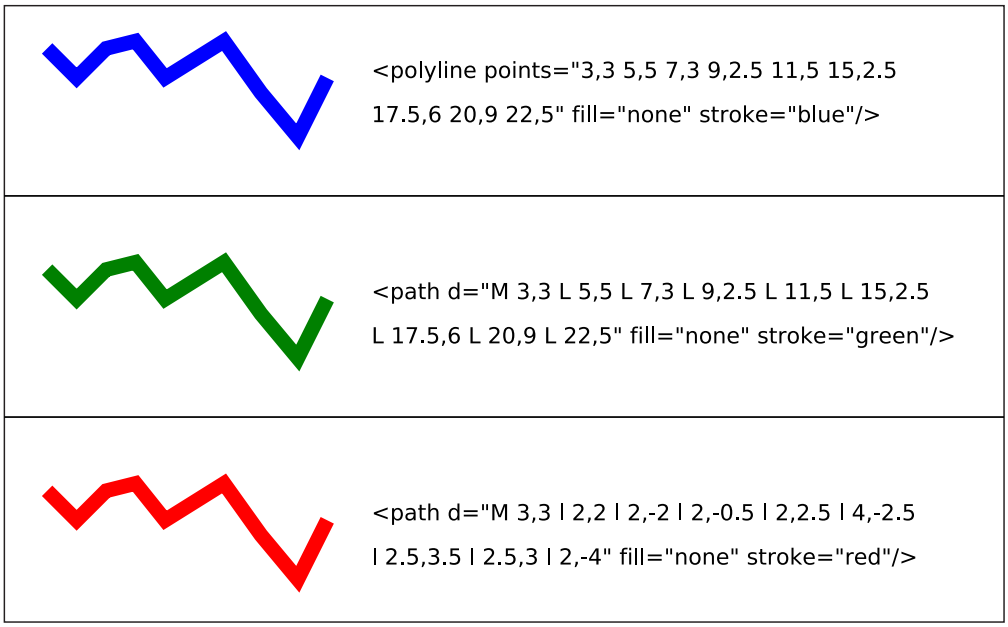


Abbildung 3.2: Unterschiedliche Erzeugung von Polylinien

sind unterschiedliche Strichelungen und verschiedene Linienenden, was für kartografische Darstellungen nicht ausreichend ist.

Eine Lösung dieses Problems kann das Verwenden von *Patterns* sein. Man kann Muster definieren und diese als Farbe oder Füllung für ein Objekt angeben. Solange man nur waagerechte oder senkrechte Linien zeichnet, ist das eine sehr komfortable Lösung. Sobald die Linien aber nicht mehr gradlinig in eine Richtung verlaufen, entsteht nicht das gewünschte Layout, da sich ein Pattern nicht entlang der Linienführung orientiert. Quellcode 3.2 zeigt solch eine Verwendung eines Patterns, mit dem eine gelbe Linie mit schwarzem Rand erzeugt werden soll. In Abbildung 3.3 findet man die dazugehörige Grafik. Man sieht, dass das Pattern selbst eine gerade Linie nicht wie gewünscht füllt, wenn diese nicht „passend“ zum Muster liegt. Die Linie, die die Richtung wechselt, erscheint gestreift. Für die Darstellung von geografischen Liniendaten, die in den seltensten Fällen gradlinig sind, eignet sich diese Methode also nicht.

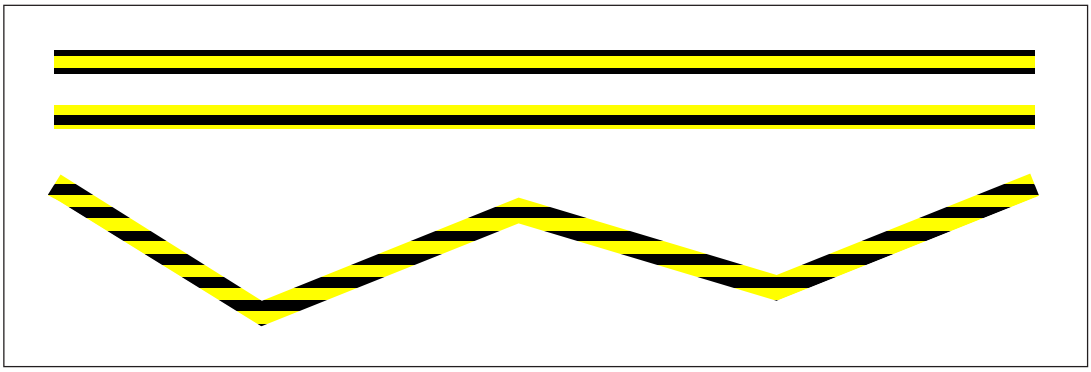


Abbildung 3.3: Verwendung von Patterns in SVG

```
<svg>
  <pattern patternUnits="userSpaceOnUse" id="pat1" x="0" y="0"
   width="10" height="9">
      <rect x="0" y="0" width="10" height="2" fill="black"/>
      <rect x="0" y="2" width="10" height="5" fill="yellow"/>
      <rect x="0" y="7" width="10" height="2" fill="black"/>
  </pattern>
  <path stroke="url(#pat1)" d="M 20,22.5
        L 400,22.5" stroke-width="9.0" fill="none" />
  <path stroke="url(#pat1)" d="M 20,44
        L 400,44" stroke-width="9.0" fill="none" />
  <path stroke="url(#pat1)"
        d="M 20,70 L 100,120 L 200,80 L 300,110 L 400,70"
        stroke-width="9.0" fill="none" />
</svg>
```

Quellcode 3.2: Verwendung von Patterns in SVG

Will man trotzdem Linien mit Rand zeichnen, so muss man zwei unterschiedlich dicke und farbige Linien übereinander legen. Durch die Kombination mehrerer Linien, die auch gestrichelt sein können, lassen sich geografische Objekte wie verschiedene Straßen und auch Eisenbahnlinien darstellen (Abbildung 3.4). Wichtig ist, dass man auf die Reihenfolge der definierten Linien achtet, damit sie in der Grafik in der richtigen Reihenfolge übereinander liegen.

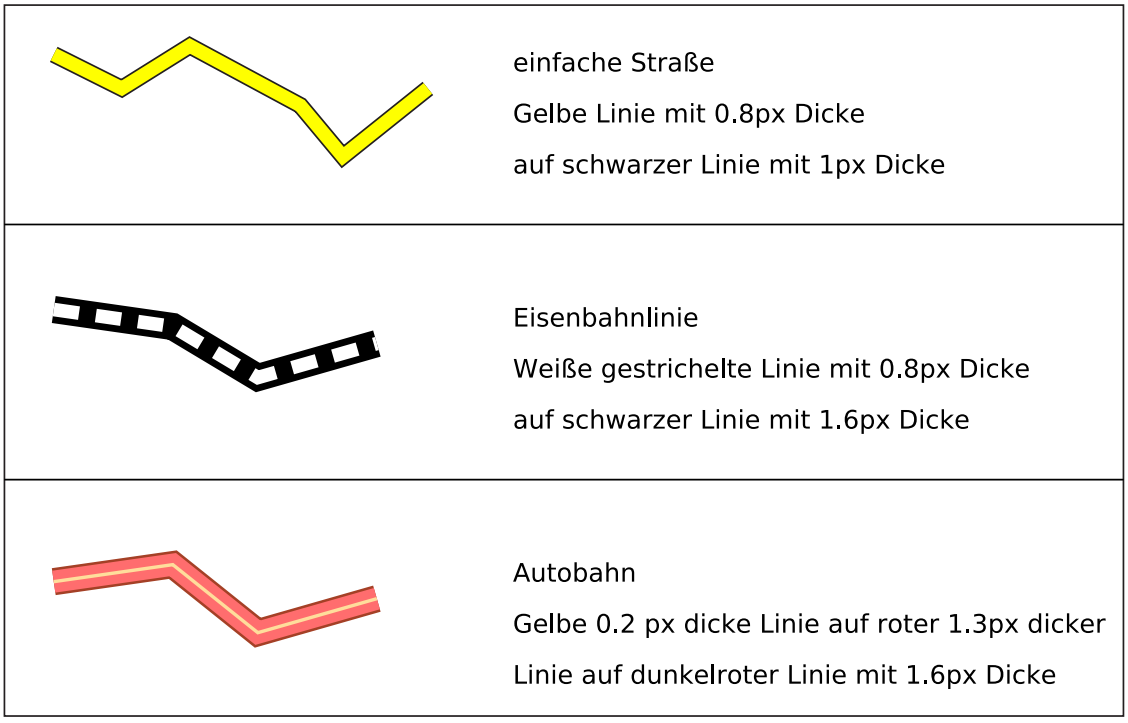


Abbildung 3.4: Unterschiedliche Linien-Layouts

Der Nachteil dieses Verfahrens ist die erhöhte Anzahl von grafischen Objekten. Statt einer einzelnen Linie verwendet man zur Darstellung eines geografischen Objektes gleich mehrere Linien, was bei einer großen Anzahl von Objekten zu Performance-Einbußen bei der Darstellung führen kann. Von der optischen Seite betrachtet ist diese Lösung aber absolut zufrieden stellend und in SVG die einzig mögliche.

3.1.2 Flächen

Für Polygone gibt es in SVG wie für Polylinien wiederum zwei Varianten, sie zu definieren. Eine Möglichkeit sind die `polygon`-Tags, die genauso verwendet werden, wie die `polyline`-Tags. Im Fall des Polygons wird automatisch eine Verbindungslinie vom ersten zum letzten Punkt gezogen. Ein Polygon kann auch mit dem `path`-Element erzeugt werden. Ob es sich bei dem erzeugten Objekt um ein geschlossenes oder offenes Polygon und damit um eine Polylinie handelt, ist davon abhängig, ob am Ende des angegebenen Pfades ein `z` angegeben wird oder nicht.

```
<svg>
 <polygon points="10,30 30,10 50,10 70,30 50,50 30,50"
           fill="yellow" stroke="black"/>
 <path d="M 10,30 L 30,10 L 50,10 L 70,30 L 50,50 L 30,50 z"
          fill="green" stroke="black"
          transform="translate(0,50)"/>
 <polygon points="10,30 30,10 50,10 70,30 50,50 30,50 10,30
                  25,30 35,20 45,20 55,30 45,40 35,40 25,30"
                  fill="yellow" stroke="black"
                  transform="translate(75,0)"/>
 <path d="M 10,30 L 30,10 L 50,10 L 70,30 L 50,50 L 30,50 z
          M 25,30 L 35,20 L 45,20 L 55,30 L 45,40 L 35,40 z"
          fill="green" stroke="black"
          transform="translate(75,50)"/>
 <polygon points="10,30 30,10 50,10 70,30 50,50 30,50 10,30
                  25,30 35,20 45,20 55,30 45,40 35,40 25,30"
                  fill="yellow" stroke="black"
                  fill-rule="evenodd"
                  transform="translate(150,0)"/>
 <path d="M 10,30 L 30,10 L 50,10 L 70,30 L 50,50 L 30,50 z
          M 25,30 L 35,20 L 45,20 L 55,30 L 45,40 L 35,40 z"
          fill="green" stroke="black"
          fill-rule="evenodd"
          transform="translate(150,50)"/>
</svg>
```

Quellcode 3.3: Unterschiedliche Erzeugung von Polygonen

Ein Code-Beispiel für Polygone ist in Quellcode 3.3 gegeben. In der Web Mapping Applikation ist es sinnvoll, das `path`-Element zu verwenden, denn so ist es möglich, ineinander liegende, zusammengehörige Polygone als ein Objekt zu zeichnen. Die grafischen

Darstellungen zum Quellcode-Beispiel befinden sich in Abbildung 3.5. Man sieht, dass bei Polygonobjekten eine Verbindungslinie zwischen dem ersten Punkt und dem letzten innen liegenden Punkt gezeichnet wird. Bei `path`-Elementen lässt sich das verhindern. Man muss dazu lediglich mit `m` oder `M` zu dem Punkt auf der innen liegenden Linie „springen“. Die schließende Linie wird bei `path`-Angaben immer zum letzten Punkt gezogen, zu dem gesprungen wurde. Mit dieser Technik ist es auch möglich „Donuts“, also Polygone, die innerhalb „Löcher“ haben, zu zeichnen. Hierfür muss man lediglich als Füllvorschift `evenodd` angeben. Durch diese Möglichkeit lassen sich zum Beispiel ineinander liegende Länder, Gebäude mit Innenhof und Ähnliches korrekt darstellen.

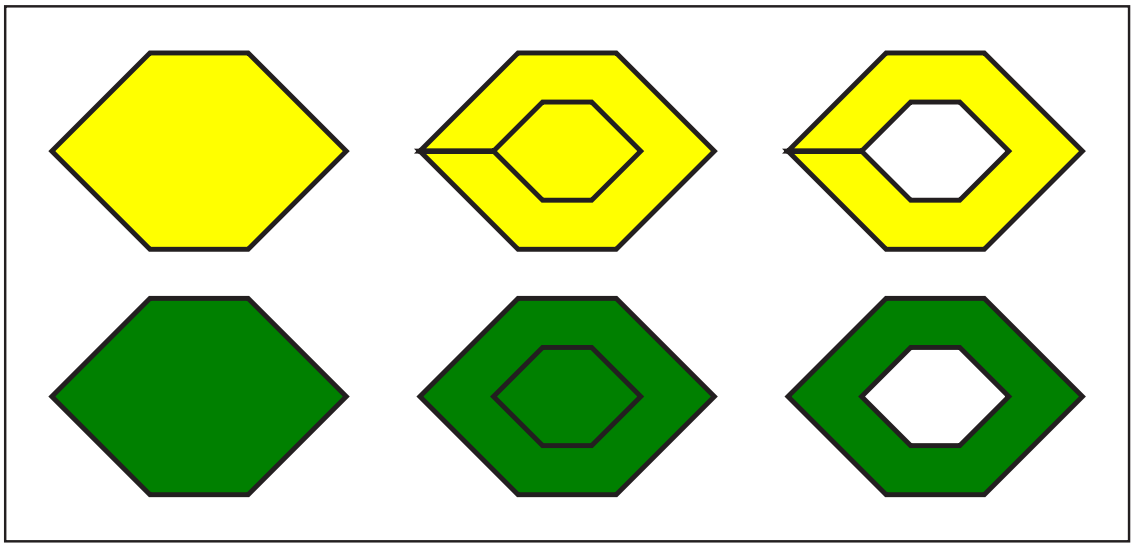


Abbildung 3.5: Unterschiedliche Erzeugung von Polygonen

3.1.3 Punktdaten

Punktdaten kann man in SVG sehr einfach mit einem `circle`-Tag darstellen. Ein Kreis hat als wichtigste Attribute die Koordinaten seines Mittelpunktes und seinen Radius sowie Füllfarbe, Linienfarbe und -dicke. Ein blauer Kreis im Ursprung mit Radius `5px` und einer roten `1px`-dicken Randlinie wird mit

```
<circle cx="0" cy="0" r="5px" fill="blue"
 stroke="red" stroke-width="1px"/>
```

beschrieben.

In der Web Mapping Applikation macht es Sinn, für Punktdaten nicht nur einfache Kreise verwenden zu können, sondern auch Symbole einfügen zu können, wie Kirchen, Krankenhäuser, Flugplätze oder Ähnliches. Diese Symbole können aus allen möglichen SVG-Elementen aufgebaut sein. Welche Elemente es außer den genannten noch gibt und wie sie zu verwenden sind, findet man unter Anderem in [WaLi2004] oder unter [Apti2004].

Wie bereits erwähnt, können im `defs`-Bereich des Dokumentes Prototypen von Elementen angelegt werden, die später mehrfach verwendet werden können. Quellcode 3.4 zeigt

solch eine Verwendung eines Kirchensymbols, Abbildung 3.6 die entsprechende Grafik. Es wird ein Element mit einer ID definiert, die man bei Verwendung des Prototyps als Referenz angibt. Im Beispiel wird auf den definierten Prototyp jeweils eine Translation angewandt und ein Farbwert gesetzt.

Durch die Verwendung von Prototypen hat man eine bequeme Handhabung der Symbole, weil man sie nur ein einziges Mal selbst definieren muss. Außerdem wird durch dieses Konzept Speicherplatz auf dem Server gespart, weil nicht an jeder Stelle, an der das Symbol auftritt, die komplette Definition des Objektes in der SVG-Datei steht. Zur Laufzeit sind ebenfalls weniger Objekte im Speicher, so wird der Client ebenfalls entlastet.

```
<svg xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <path id="kirche" d="M -8,6 L -8,-5 L -5,-9 L -2,-5
          L -2,-2 L 6,-2 L 8,1 L 8,6 L -8,6 M -5,-5
          L -5,-1 M -6, -4 L -4,-4" stroke="black"
          stroke-linecap="square" stroke-width="1"/>
  </defs>
  <use xlink:href="#kirche" fill="red"
   transform="translate(15,12)"/>
  <use xlink:href="#kirche" fill="blue"
   transform="translate(35,12)"/>
  <use xlink:href="#kirche" fill="yellow"
   transform="translate(25,30)"/>
</svg>
```

Quellcode 3.4: Verwendung von Prototypen in SVG

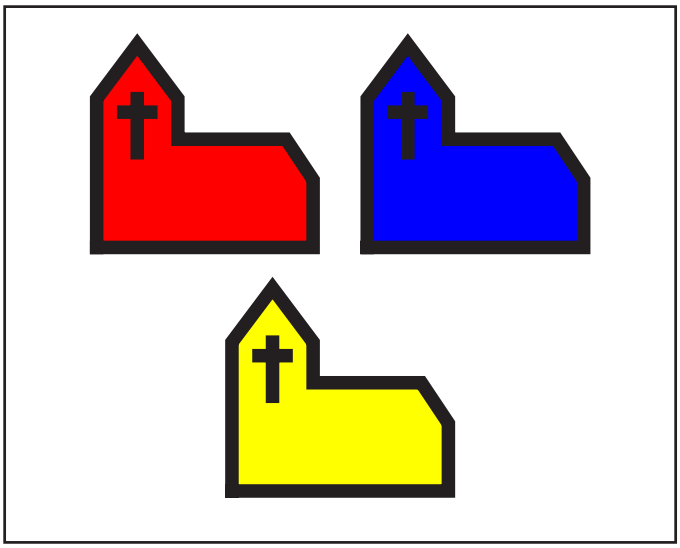


Abbildung 3.6: Verwendung von Prototypen in SVG

3.1.4 Text

Zum Einfügen von Text in ein SVG-Dokument verwendet man das `text`-Tag. über Attribute werden Position, Schriftart und -größe, Farbe und Ausrichtung angegeben. Außerdem kann man Text entlang von Pfaden definieren. Dies ist wichtig für die Web Mapping Applikation, da sich so Liniendaten wie zum Beispiel Straßen sehr einfach beschriften lassen. Als Beispiel dazu ist Quellcode 3.5 gegeben, Abbildung 3.7 zeigt die Grafik.

```
<svg xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink" >
  <defs>
    <path d="M 3,3 L 7,5 L 11,2.5 L 17.5,6 L 21,7
     L 28,5 L 33,3 L 37,5" fill="none" stroke="red" id="linie"/>
  </defs>
  <text x="3" y="5" style="font-size:2" >
    Ein wenig Text!!!
  </text>
  <use xlink:href="#linie" transform="translate(0,8)"/>
  <text style="font-size:2">
    <textPath xlink:href="#linie" transform="translate(0,8)">
      Text entlang eines sichtbaren Pfades
    </textPath>
  </text>
  <text style="font-size:2" transform="translate(0,16)">
    <textPath xlink:href="#linie">
      Text entlang eines Pfades
    </textPath>
  </text>
</svg>
```

Quellcode 3.5: Text in SVG

Abbildung 3.7: Text in SVG

Es wird zunächst ein einfacher Text mit `x`- und `y`-Koordinate erzeugt. Um Text entlang von Pfaden zu schreiben, muss man zunächst den Pfad definieren und ihm eine ID geben. Dieser Pfad wird dann referenziert, indem man in einem `text`-Tag ein `textPath`-Element definiert, dessen `xlink:href`-Attribut auf die ID des Pfades verweist. Innerhalb des `textPath`-Elementes fügt man den zu schreibenden Text ein. Soll der Pfad mit angezeigt werden, muss man diesen mit einem `use`-Tag explizit angeben.

3.1.5 Viewbox

Das `viewBox`-Attribut gehört zum Wurzelelement `svg` des SVG-DOM. Es gibt an, welchen Ausschnitt der Grafik der Benutzer gerade sieht. Die ersten beiden Werte geben die linke obere Ecke (x- und y-Koordinate) des Ausschnitts an, die nächsten beiden Breite und Höhe. Der Bereich der Grafik auf dem Bildschirm wird als *Viewport* bezeichnet. Er wird über die Höhe und Breite des `svg`-Tags festgelegt. Quellcode 3.6 und Abbildung 3.8 zeigen die Verwendung des `viewBox`-Attributes. In der Abbildung ganz links ist die Viewbox so gesetzt, dass alle vorhandenen Elemente vollständig sichtbar sind. Der Viewport nimmt 250 mal 250 Pixel des Bildschirms ein. Man kann nun einen beliebigen Bereich in gleicher Größe betrachten, wenn die Viewbox andere Koordinaten bekommt und Höhe und Breite beibehalten werden. Dies ist in der Mitte zu sehen. Über das Setzen der Viewbox kann auch ein Zoomeffekt erzeugt werden, indem man die Höhe und Breite der Viewbox verringert. Die Elemente werden dann entsprechend größer skaliert, so dass der gesamte Viewport ausgefüllt ist, wie rechts abgebildet. Auf Zoomvorgänge zur Laufzeit wird in Kapitel 4.1 und 8.2 näher eingegangen. Sollte das Höhen-Breiten-Verhältnis von Viewbox und Viewport nicht übereinstimmen, wird defaultmäßig ein der Größe des Viewports entsprechender Ausschnitt angezeigt. Dabei stimmt der Mittelpunkt der Viewbox mit dem des Viewports überein. Man kann über das Attribut `preserveAspectRatio` das Verhalten ändern. Gibt man hier `none` an, so wird der angezeigte Bereich jeweils in x- und y-Richtung skaliert. So kann es zum Verzerren des Bildes kommen. Weitere Möglichkeiten, das Verhalten der Anzeige bei Gebrauch der Viewbox zu beeinflussen, findet man in [Eise2002].

```
<svg width="250" height="250" viewBox="0 0 70 70">
  <rect x="10" y="10" width="20" height="20"
        fill="red" stroke="black"/>
  <rect x="40" y="40" width="20" height="20"
        fill="blue" stroke="black"/>
  <circle cx="50" cy="20" r="10" fill="yellow" stroke="black"/>
  <circle cx="20" cy="50" r="10" fill="green" stroke="black"/>
</svg>
```

Quellcode 3.6: Das `viewBox`-Attribut

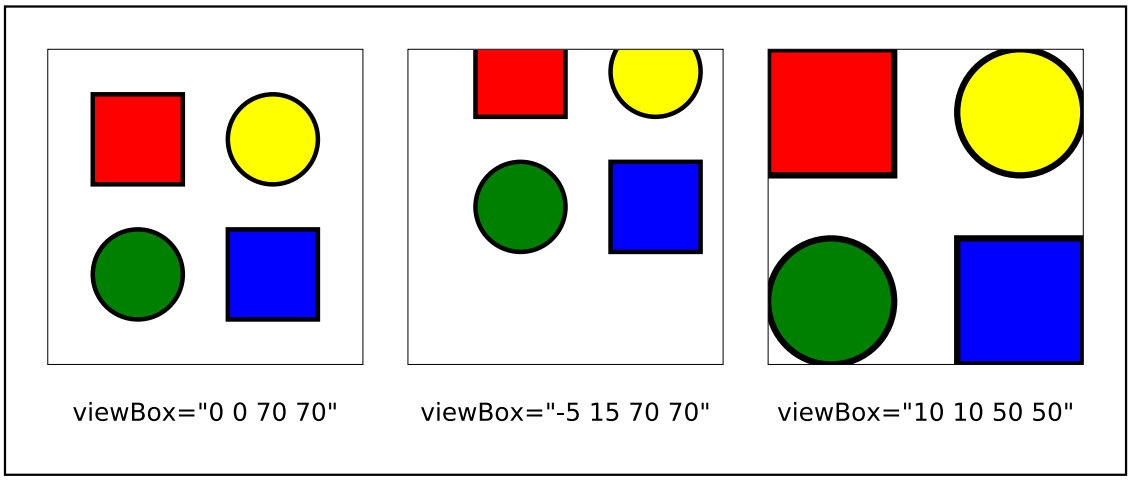


Abbildung 3.8: Das `viewBox`-Attribut

3.2 SVG-Viewer

Da SVG ein relativ junges Format ist, sind viele Entwicklungen, die sich mit der Anzeige von SVG-Dokumenten befassen, noch nicht ausgereift. Es existieren Plugins für Browser, aber auch eigenständige Viewer. Eine Reihe von Projekten beschäftigt sich mit der Anzeige von SVG-Dokumenten, so gibt es zur Zeit Plugins von Adobe und Corel. Viewer sind beispielsweise das nicht mehr weiterentwickelte CSIRO SVG Toolkit und ein im Batik SVG Toolkit von Apache [Apac2004c] enthaltener SVGViewer. Batik befindet sich noch in der Entwicklung, derzeit wird in der Version 1.5.1 weder Animation noch Interaktion unterstützt, statische SVG-Elemente werden aber annähernd vollständig unterstützt. Eine Übersicht zu existierenden SVG-Viewern, Plugins und ihrem Entwicklungsstand findet man bei [W3C2005e].

Viele Viewer und Plugins haben Schwierigkeiten, Animation und Interaktion über Skript-Sprachen zu verwirklichen, was jedoch unumgänglich ist, wenn man alle Features von SVG nutzen möchte. Gerade für eine Web Mapping Applikation ist Interaktion ein wesentliches Merkmal (Kapitel 7). Möchte man also das SVG-Dokument interaktiv verändern, empfiehlt sich derzeit die Verwendung des *SVG-Plugins von Adobe (ASV)*, erhältlich unter [Adob2005]. Im Vergleich zu anderen SVG-Viewern unterstützt das Plugin die meisten Eigenschaften von SVG. Für alle gängigen Plattformen existiert der ASV zurzeit in Version 3, für Windows liegt bereits eine beta-Version mit Versionsnummer 6 vor, die sich in Details von der Version 3 unterscheidet. Auf einige Unterschiede der Plugins und Versionen bezüglich Scripting wird in Kapitel 4 eingegangen. Unter den Web-Browsern können zurzeit nur Mozilla Firefox 1.5 [Moz2005], Opera [Oper2005] und der Konqueror [Konq2005] SVG nativ anzeigen. Firefox und Konqueror unterstützen allerdings keine Animationen. Ein für die Web Mapping Applikation wesentliches Feature des ASV, nämlich die Möglichkeit externe Daten zu laden, ist bei den drei genannten Browsern nicht umgesetzt.

4 ECMAScript

Netscape führte 1995 *JavaScript* als proprietäre Sprache ein und lizenzierte sie. Um den Lizenzvorgaben von Netscape zu entgehen, entwickelte Micosoft zur gleichen Zeit die eigene JavaScript-Variante *JScript*. Netscape ließ JavaScript 1997 von der European Computer Manufacturers Association (ECMA) unter dem Namen *ECMAScript* als Industriestandard definieren, wodurch es gegenüber Microsofts JScript, das viele plattformabhängige Erweiterungen besitzt, in den Status eines offiziellen Standards [Ecma1999] versetzt wurde. Heute kann man ECMAScript als eine Schnittmenge von JavaScript und JScript beschreiben, die die Schnittstelle zum DOM 2 vollständig umsetzt. Es ist somit möglich, mit ECMAScript nicht nur HTML-Dokumente, sondern beliebige XML-Dokumente, wie zum Beispiel SVG, über das DOM zu manipulieren.

Eine ebenfalls wichtige Eigenschaft von ECMAScript ist die Möglichkeit der Event-Verarbeitung. Es kann auf Ereignisse wie Mausklicks, Mausbewegungen etc. reagiert werden, das Ereignis wird in Form eines Event-Objektes weitergereicht. Von diesem kann man beispielsweise Quelle oder Ort des Ereignisses abfragen und entsprechend darauf reagieren.

Wie in Kapitel 3 beschrieben, gibt es unterschiedliche Plugins, um SVG im Browser betrachten zu können. Zum Teil verwenden diese Plugins eine eigene Implementation von ECMAScript, um nicht vom ECMAScript des Browsers abhängig zu sein. Da das SVG-Plugin von Adobe das zur Zeit am weitesten entwickelte Plugin bezüglich möglicher Interaktionen innerhalb eines SVG-Dokumentes ist (Kapitel 3.2) und dadurch einige wesentliche Vorteile bietet, wird es in der vorliegenden Arbeit genutzt. Im Folgenden werden einige einfache Beispiele für die Manipulation eines SVG-DOM und für die Event-Verarbeitung gegeben, die exemplarisch einige Techniken vorstellen, welche in der Web Mapping Applikation Verwendung finden.

4.1 SVG-DOM-Manipulation

Will man ECMAScript zum Manipulieren des SVG-DOM verwenden, gibt es zwei Möglichkeiten, das Script einzubinden. Die erste ist, ein `script`-Tag einzufügen und darin in einer `CDATA`-Section das Skript direkt innerhalb des SVG-Dokumentes zu schreiben, wie es auch in den folgenden Beispielen der Fall ist. Die andere Möglichkeit ist, über eine Referenz im `script`-Tag das gewünschte Script einzubinden. Will man beispielsweise die Datei `script.js` in ein Dokument einbinden, schreibt man folgende Zeile:

```
<script type="text/javascript" xlink:href="script.js"/>
```

Wie bereits erwähnt, gibt es in ECMAScript die Möglichkeit, Events zu verarbeiten. Die wichtigsten Events für eine Web Mapping Applikation sind sicherlich die verschiedenen Maus-Events wie `mouseclick`, `mousemove`, `mousedown` und `mouseup`. Sollen diese verarbeitet werden, erhalten die entsprechenden Elemente im Dokument einen Event-Handler als zusätzliches Attribut, welcher entsprechend des jeweiligen Events benannt ist, nämlich als Event-Name mit vorangestelltem `on`. Beispielsweise kann man

mit `onmousemove` auf eine Mausbewegung reagieren. Dem Attribut wird jeweils eine Funktion zugewiesen, die gegebenenfalls das entstandene Event als Parameter übergeben bekommt, welches den reservierten Namen `evt` trägt. Eine ebenfalls wichtige Möglichkeit ist, beim Laden des SVG-Dokumentes in den Browser bereits Script auszuführen, um zum Beispiel Elemente zu erzeugen. Hierfür erhält das `svg`-Tag das Attribut `onload` mit einer entsprechend zugewiesenen Funktion. Ein Beispiel ist in Quellcode 4.1 gegeben, es wird beim Laden des Dokumentes ein blauer Kreis mit schwarzem Rand im Ursprung mit Radius 5 erzeugt.

```
<svg onload="createCircle()">
   <script language="javascript">
     <![CDATA[
        function createCircle(){
          var c= document.createElement('circle');
          c.setAttribute('cx','0');
          c.setAttribute('cy','0');
          c.setAttribute('r','5');
          c.setAttribute('fill','blue');
          c.setAttribute('stroke','black');
          document.rootElement.appendChild(c);
        }
     ]]>
  </script>
</svg>
```

Quellcode 4.1: Script zum Erzeugen eines SVG-Kreises beim Laden

Das SVG-Dokument kann über die reservierte Variable `document` angesprochen werden. Es hat die Eigenschaft, Elemente erzeugen zu können, wozu man es mit der Funktion `createElement(elementname)` auffordern kann. Man sieht in diesem Beispiel außerdem, wie man Attribute eines Elementes setzt. Würde man dies nicht tun, würden die Default-Werte verwendet. Wichtig ist, dass man das neu erzeugte Element in den DOM-Tree einhängt, indem man es an das Wurzelelement als Kind anhängt. Bevor man diese Anweisung nicht gegeben hat, ist das Element nicht sichtbar.

Es besteht auch die Möglichkeit, vorhandene Attribute abzufragen und eventuell entsprechend neu zu setzen. Über den Aufruf von `getAttribute(attributname)` an einem Knoten kann man dessen Attribute abfragen, als Parameter übergibt man dann den Attributnamen (Quellcode 4.2). Hier sieht man außerdem, dass man Elemente direkt über ihre ID ansprechen kann bzw. sich ein bestimmtes Element vom Dokument geben lassen kann. Dies erfolgt durch den Aufruf der Funktion `getElementById(id)`. Es gibt noch weitere Funktionen, sich bestimmte Elemente vom Dokument geben zu lassen, z. B. `getElementsByTagName(tagname)`, die alle Elemente mit dem gegebenen Namen liefert. Ebenso kann man beliebige Knoten mit `getChildNodes()` nach all ihren Kindknoten fragen. Diese Funktionen werden jedoch zum Teil nur vom ASV unterstützt, bei einigen anderen Plugins muss man auf die Attribute des Knotens direkt zugreifen. So würde der Aufruf von `node.getChildNodes()` identisch sein mit

`node.childNodes`. Die Namen der Attribute und die Namen der get-Methoden stimmen überein, die Attribute beginnen lediglich mit einem Kleinbuchstaben. Um möglichst allgemein funktionierende Skripte zu schreiben, sollte man immer die zweite Variante wählen, die auch für den ASV funktioniert. Hier sind nur einige der möglichen Funktionen bzw. Attribute genannt, mit denen man Elemente des SVG-Dokumentes erhält, eine vollständige Übersicht findet man in [Flan2002].

```
<svg>
  <script language="javascript">
    <![CDATA[
       function changeVisibility(){
         e=document.getElementById('mycircle');
         if(e.getAttribute('visibility')!='hidden'){
            e.setAttribute('visibility','hidden');
         }
         else{
           e.setAttribute('visibility','visible');
         }
       }
    ]]>
  </script>
  <circle id="mycircle" cx="10" cy="7" r="5" stroke="black"
   fill="yellow" onclick="changeVisibility()"/>
</svg>
```

Quellcode 4.2: Attribute abfragen und neu setzen

Das Beispiel in Quellcode 4.3 zeigt, wie man das (einmalige) Auszoomen aus einer SVG-Grafik per Mausklick auf ein Element verwirklichen kann. Für alle Zoomvorgänge ist das Neusetzen des `viewBox`-Attributes der entscheidende Vorgang, weil man so jeden beliebigen Ausschnitt in beliebiger Größe dynamisch auswählen kann.

```
<svg height="800" width="400" viewBox="0 0 80 40">
 <script language="javascript">
  <![CDATA[
   function changeVB(){
    document.rootElement.setAttribute('viewBox','0 0 160 80');
   }
  ]]>
 </script>
 <circle cx="10" cy="7" r="5" stroke="black" fill="yellow"
  onclick="changeVB()"/>
</svg>
```

Quellcode 4.3: Zoomen durch Neusetzen der Viewbox

4.2 Laden von externen Daten

Mit dem ASV ist es im Gegensatz zu anderen Plugins möglich, Daten von außerhalb in das Dokument zu laden. Hierzu wird die Funktion `getURL(url,functionname)` verwendet, welche jedoch plugin-spezifisch ist. Man übergibt als ersten Parameter den URL [4] der zu ladenden Daten, als zweiten Parameter einen Funktionsnamen. Diese so genannte *Callback-Funktion* bekommt im Laufe der Ausführung von `getURL` die erhaltenen Daten übergeben und kann sie weiterverarbeiten, sofern die Datenlieferung erfolgreich war.

Quellcode 4.5 zeigt, wie ein Kreis, dessen SVG-Beschreibung in einer Textdatei (Quellcode 4.4) steht, in das vorhandene Dokument eingefügt wird.

```
<circle cx="0" cy="0" r="5" fill="blue" stroke="black"/>
```

Quellcode 4.4: Textdatei mit SVG-Kreis-Beschreibung

```
<svg onload="getURL('circle.xml',getData)">
 <script language="javascript">
  <![CDATA[
    function getData(data){
      if(data.success){
        st=data.content;
        node=parseXML(st,document);
        document.rootElement.appendChild(node);
      }
    }
  ]]>
 </script>
</svg>
```

Quellcode 4.5: Laden von externen Daten

Beim Laden des Dokumentes wird die Funktion `getURL` aufgerufen. Sie liest die Datei `circle.xml` aus und erzeugt ein `data`-Objekt. Dieses wird an die durch den Benutzer angegebene Funktion `getData(data)` weitergereicht. Das nun zur Verfügung stehende `data`-Objekt hat ein Datenfeld `success`, dass `true` ist, falls wirklich Daten vorhanden sind. Dieses kann man über das Datenfeld `content` abfragen. Der Inhalt der Textdatei ist nun als String vorhanden und wird mit der Funktion `parseXML(string)` zu einem XML-Knoten umgewandelt, also geparst (Kapitel 2.3). Hierfür muss der Funktion der String und das Dokument, in dessen DOM der Knoten eingefügt werden soll, übergeben werden. Der zurückgelieferte Knoten kann anschließend in den DOM-Tree eingefügt werden.

[4] Ein *Uniform Resource Locator (URL)* ist eine eine Unterart von Uniform Resource Identifiern. URLs identifizieren eine Ressource über ihren primären Zugriffsmechanismus (häufig http oder ftp) und den Ort der Ressource in Computernetzwerken.

Es ist nicht nur möglich, Inhalte von Textdateien in ein XML-Dokument zu laden. Die Funktion `getURL` kann mit beliebigen URLs aufgerufen werden. Es ist beispielsweise möglich, dass es sich bei dem angegebenen URL um ein Skript handelt, das bei Aufruf XML-Ausgaben erzeugt. Als Programmierer schreibt man jeweils eine geeignete Funktion, die mit den angelieferten Daten umgehen kann. In Kapitel 8.4 wird solch ein Vorgang anhand des Aufrufs eines PHP-Skriptes beschrieben.

Die Funktion `parseXML(string)` kann auch gezipte Daten erhalten, was von Vorteil ist, wenn bei einer Client-Server-Applikation größere Datenmengen ausgetauscht werden. Der Server kann die Daten gezipt schicken, `parseXML` nimmt diese entgegen und entpackt sie automatisch, bevor der XML-Knoten erzeugt wird.

Wenn ein Programmierer die Funktion `parseXML` verwendet, muss er beachten, dass ihr Verhalten in verschiedenen Plugin-Versionen unterschiedlich ist. Beispielsweise verarbeitet der ASV6 beta im Gegensatz zum ASV3 kein XML, das Versions- und/oder Encoding-Angaben enthält. Ein weiterer Unterschied ist, dass der ASV6 auch XML-Knoten verarbeitet, die keinen gemeinsamen Elternknoten haben. Der ASV3 hingegen erkennt nur den ersten geparsten Knoten, eventuell auf gleicher Ebene folgende Knoten werden ignoriert.

5 PHP

PHP Hypertext Preprocessor wurde 1995 von Rasmus Lerdorf als Open Source Skriptsprache entwickelt [Wiki2005]. Aktuell ist PHP in der Version 5 erhältlich, in der viele Veränderungen und Ergänzungen gegenüber PHP 4 vorgenommen wurden. Eine entscheidende Erweiterung der Sprache ist die Verfügbarkeit von vielen Funktionen zur objektorientierten Programmierung.

Für die Web Mapping Applikation ist die Implementation des DOM Interfaces (Kapitel 2.1) relevant. Im Unterschied zu anderen Sprachen wird der PHP-Code serverseitig ausgeführt, was den Vorteil bietet, dass auf Client-Seite keinerlei Voraussetzungen bzw. Fähigkeiten zum Ausführen vorhanden sein müssen. Das Skript wird bei Aufruf ausgeführt und die Ausgabe an den Client weitergeleitet.

PHP bietet einen sehr großen Funktionsumfang, der für professionelle Programmierer ausreichende Möglichkeiten bietet. Trotzdem ist PHP auch von Anfängern leicht zu erlernen, da Einsteiger einen schnellen Zugang zu den Funktionen und deren Handhabung finden [PHP2005a] [PHP2005b]. Die Syntax von PHP lehnt sich an C, Java und Perl an.

Im folgenden Kapitel wird ein Beispiel für die objektorientierte Programmierung mit PHP gegeben. Dabei werden die wichtigsten Funktionalitäten für die Web Mapping Applikation gezeigt: DOM-Erzeugung, XML parsen und „fremde“ XML-Knoten in ein Dokument importieren.

5.1 DOM-Erzeugen und XML-Laden

PHP 5 implementiert das DOM-Interface vom W3C [W3C2005g]. Daher sind die Funktionsaufrufe zum Erzeugen von Elementen und Setzen von Attributwerten identisch mit ECMAScript (Kapitel 4.1) oder Java (Kapitel 2.4):

```
$dom=new DomDocument();

$data=$dom->createElement('data');
$dom->appendChild($data);

$g=$dom->createElement('g');
$g->setAttribute('id','kind0');
$data->appendChild($g);
```

Mit dem Aufruf von `new DOMDocument()` erzeugt man ein leeres DOM. Es besteht auch die Möglichkeit, ein Dokument bestimmten Typs zu erzeugen. Dann würde man allerdings einen anderen Aufruf benötigen und ein `DOMImplementation`-Objekt verwenden [PHP2005a]:

```
DOMImplementation->createDocument(uri,name,doctype);
```

In diesem Beispiel wird gezeigt, wie man in PHP externe Daten aus einer Datei in einen DOM einfügt. Zunächst muss man sich mit dem Inhalt dieser Datei einen eigenen DOM erzeugen. Die Datei „beispiel.xml“ wird ausgelesen und ein neues `DOMDocument` erstellt:

```
$filename='beispiel.xml';
if(file_exists($filename)){
   $handle=fopen ($filename,"r");
   $buffer=fread ($handle, filesize($filename));
   fclose($handle);
   $dom2=new DomDocument();
   $dom2->loadXML($buffer);
   $neu=$dom->importNode($dom2,true);
   $neu->setAttribute('id','kind2';
   $data->appendChild($neu);
}
```

Das zunächst leere Dokument `$dom2` kann mit `parseXML(string)` die zuvor ausgelesenen XML-Daten parsen und in das eigene Baummodell laden. Das so entstandene DOM muss zunächst mit der Funktion `importNode(param)` in das Dokument `$dom` importiert werden. Der Parameter `true` gibt an, dass auch alle Kindknoten des zu importierenden Knotens mit importiert werden. Der Import ist notwendig, da ein Knoten nur in den DOM-Tree eines Dokumentes eingehängt werden kann, von dem er erzeugt wurde. Ein Knoten kann immer nur zu einem Dokument gehören (Kapitel 2). Anschließend kann der Knoten in den DOM-Tree eingefügt werden.

Um das Dokument auszugeben, kann man am Dokument die Funktion `saveXML()` aufrufen, die dann eine Stringrepräsentation des Dokumentes liefert. Im Beispiel wird dieser String vom Skript ausgegeben:

```
$ausgabe=$dom->saveXML();
echo $ausgabe;
```

6 Shapefiles

Die Firma ESRI [Esri2005a] entwickelte das *Shapefile*-Format, ein binäres Geodaten-Format, ursprünglich für das Geoinformationssystem (GIS) ArcView. Es hat sich mittlerweile zu einer Art Quasi-Standard im GIS-Umfeld verbreitet, da es ein recht einfaches Format darstellt. Unter [Esri1998] ist die technische Beschreibung von Shapefiles zu finden. Im Folgenden wird ein kurzer Überblick dazu gegeben und die für eine SVG Web Mapping Applikation wichtigen Eigenschaften erläutert.

6.1 Aufbau

Das Shape-Format besteht aus mindestens 3 Dateien pro „Shapefile“ mit folgenden Endungen:

1. **.shp** zur Speicherung der Geometriedaten (Hauptdatei)
2. **.shx** enthält Index der Geometrie zur Verknüpfung der Sachdaten (Attributdaten)
3. **.dbf** Sachdaten im dBase-Format

Weitere optionale Angaben können in Dateien mit folgenden Endungen gemacht werden:

1. **.atx** Attributindex
2. **.sbx** und **.sbn** Tabellenverbindungen
3. **.aih** und **.ain** Tabellenverknüpfungen
4. **.shp.xml** Metadaten zum Shapefile
5. **.prj** Projektion der Daten
6. **.avl** Legendendatei

In der Hauptdatei stehen in Datensätzen variabler Länge die Beschreibungen von Vektorgrafikobjekten. Pro Datei können jeweils nur Objekte eines Shape-Typs gespeichert werden, der im Header der .shp-Datei über Angabe eines Integer-Wertes angegeben wird. Insgesamt gibt es 14 unterschiedliche Typen, die in Tabelle 6.1 mit ihrem Wert aufgelistet sind.

Im Wesentlichen lassen sich diese Typen in drei Klassen zusammenfassen, nämlich Punkte, Polylinien und Polygone. `Point`, `PolyLine` und `Polygon` sind 2-dimensionale Objekte, ihre Koordinaten bestehen also aus x- und y-Wert. Es können nicht nur 2D-Objekte, sondern auch 3D-Geometrien erfasst werden. Die Typen mit der Endung `M` enthalten einen zusätzlichen Messwert, die Objekte mit der Endung `Z` außerdem eine z-Koordinate. Ein `MultiPoint` ist ein Satz von Punkten. Ein `MultiPatch` ist aus

Wert	Shape	Wert	Shape
0	Null Shape	15	PolygonZ
1	Point	18	MultiPointZ
3	PolyLine	21	PointM
5	Polygon	23	PolyLineM
8	MultiPont	25	PolygonM
11	PointZ	28	MultiPointM
13	PolyLineZ	31	MultiPatch

Tabelle 6.1: Mögliche Shape-Typen

unterschiedlichen Formen zusammengesetzt und somit komplizierter. Zusammengefasst enthalten Shapefiles also Punktdaten, Liniendaten oder Flächendaten.

Die Reihenfolge der Grafikobjekte in der `.shp`-Datei ist identisch mit der Reihenfolge der Attribute in der `.dbf`-Datei. Dieser Sachverhalt ist in Tabelle 6.2 skizziert.

ID	city.shp	city.dbf
1	type=1 (point) x=9.9748 y=53.5358	label=Hamburg category=1
2	type=1 (point) x=10.1312 y=54.2962	label=Kiel category=2
3	type=1 (point) x=8.0727 y=52.2697	label=Osnabrück category=4
4	...	...

Tabelle 6.2: Zusammenhang der Einträge in `.shp` und `.dbf`-Dateien

Es gibt Bibliotheken in verschiedenen Sprachen wie PHP, C [Warm2000] oder Java-Klassen [OpMa2005a], die das Auslesen von Shapefiles erleichtern. In dieser Arbeit wird mit einigen der zuletzt genannten Java-Klassen gearbeitet, sie wurden jedoch zum Teil in Java 1.5 umgeschrieben, verändert und ergänzt (Kapitel 9).

6.2 Shapefiles als Grundlage für SVG Web Mapping

Generell eignen sich Shapefiles sehr gut für beliebige Web Mapping Applikationen. Obwohl ihr Format kein verabschiedeter Standard ist, halten sich alle GIS, die mit Shapefiles arbeiten, an die technische Beschreibung von ESRI. So kann man sich beim Arbeiten mit Shapefiles darauf verlassen, dass sie nach einem bestimmten Konzept abgespeichert sind und bestehende Programme zum Auslesen verwenden.

Alle gängigen GIS unterstützen den Export von Geodaten im Shapeformat. Verfügt man über Geodaten in einem anderen Format, ist es also möglich, diese Daten in ein GIS zu importieren und diese Daten ins Shapefile-Format zu konvertieren.

Vorteile bietet die Speicherung der Daten als Vektordaten, insbesondere die Verwaltung der Objekte als Polygon, Polylinie und Punktobjekte besonders für die Verwirklichung einer SVG Web Mapping Applikation. Diese Grundobjekte sind problemlos als `polygon`-, `polyline`- oder `path`-Objekte sowie als `circle`-Objekte in einer SVG-Grafik darzustellen (Kapitel 3.1).

7 Web Mapping

Der Begriff *WebGIS* beschreibt im Allgemeinen ein *Geoinformationssystem (GIS)*, dessen Funktion teilweise auf Netzwerktechnologien wie Internet und Intranet basiert. WebGIS wird häufig synonym mit *GIS online*, *Internet-GIS*, *NetGIS*, *Distributed GIS*, *Internet Mapping* oder *Web Mapping* verwendet. Unter dem Begriff Web Mapping kann man ganz allgemein den Abruf von geografischem Kartenmaterial aus dem Internet verstehen. Solange mindestens zwei Rechner miteinander kommunizieren und Geodaten austauschen bzw. GIS-Funktionalität bereitstellen (Client-Server Prinzip) kann schon von einem WebGIS gesprochen werden [Wiki2005].

Da der Begriff Web Mapping recht allgemein ist, ist es sinnvoll, verschiedene WebGIS-Typen zu unterscheiden [Plew1997]. Im Folgenden werden verschiedene existierende Typen vorgestellt. Diese unterscheiden sich hauptsächlich in der bereitgestellten Funktionalität des Server- bzw. des Client-Rechners. So sind bei einem typischen Desktop-GIS sowohl die gesamte GIS-Funktionalität als auch die Daten auf einem Client-Rechner vorhanden. Im Gegensatz dazu ist bei modernen freien WebGIS-Architekturen wie dem UMN MapServer [MaSe2005] die Funktionalität des Clients auf die Visualisierung der Daten und triviale GIS-Funktionen (Bewegen in der Karte, Zooming, Wählen von Ausschnitten oder Ebenen) beschränkt. Diese Systeme arbeiten in Verbindung mit einem WebClient. Der Großteil der Arbeit liegt bei einem oder mehreren Servern, für den Client wird nur noch ein einfacher Webbrowser benötigt.

Im Fall eines *Geodaten-Servers* hat ein Desktop-GIS die Möglichkeit, Daten von entfernten Rechnern zu laden. Diese können lokal weiter verarbeitet werden. Als Geodaten-Server wird der Rechner bezeichnet, der die Geodaten zum Download bereitstellt. Die Aufgabe der Kartenerstellung liegt weiterhin ausschließlich beim Client, der entsprechend dem Umfang seiner Funktionalität als *thick client* bezeichnet werden kann.

Beim *Map-Server* liegt die Aufgabe der Kartenerstellung beim Server, wobei diese nicht zwangsläufig dynamisch erfolgen muss. In diesem Fall spricht man von einem *statischen* Map-Server. Die so genannten „static maps“ sind weit verbreitet, wie Wegbeschreibungen (Anfahrtskizzen), bei denen das Rasterbild einer Karte angeboten wird. Im Internet werden viele Karten auch über die HTML-Technik der „imagemaps“ bereitgestellt. Diese bieten dem Benutzer die Möglichkeit, bestimmte Kartenausschnitte näher zu betrachten. Die Karte ist dabei ebenfalls statisch. Bei Map-Servern benötigt der Client nur noch eine eingeschränkte Funktionalität (*thin client*), da er sich nicht um die Erstellung des Kartenmaterials kümmern muss. Dies ist ebenso bei einem *dynamischen* Map-Server der Fall, von dem die Karten auf Anfrage des Clients dynamisch erzeugt und zurückgegeben werden. Der Server wertet die Anfrage aus und die relevanten Angaben werden an einen Kartengenerator weitergeleitet. Hier werden Angaben über die unterschiedlichen Darstellungsformen wie Farbe, Symbole und Schriftgröße verarbeitet und es können, wie bei Online-Auskunftsystemen themenbezogene Auswahlmöglichkeiten (Layer) vorhanden sein. Ausgehend von der Client-Anfrage wird aus den vorhandenen Daten der angefragte Ausschnitt ausgewählt und eine entsprechende Karte produziert.

Bei dieser Einteilung sollte man darauf achten, die Begriffe Client/Server und Map-Client/Map-Server voneinander zu unterscheiden. Es wurde bisher von der technischen

Seite aus vom Internet-Client und vom Internet-Server gesprochen, wobei unterschieden wurde, ob der Client viel oder wenig Funktionalität bereit stellen muss (thick oder thin client). Bei der Verwendung der Begriffe Map-Server und Map-Client sollte man beachten, dass ein Map-Server nicht ohne einen geeigneten Map-Client auskommt. Im Regelfall ist dieser eine Webseite, die das Kartenbild darstellt, Navigations- und Interaktionsmöglichkeiten wie Zooming, Panning, Layerauswahl etc. bietet. Den Map-Client kann man also als Bindeglied zwischen dem Internet-Client (Benutzer) und dem Map-Server bezeichnen. Der Betreiber des Map-Servers kann solch einen Map-Client selbst anbieten oder es wird ein externer Map-Client verwendet, der mit dem Map-Server kommunizieren kann.

Noch umfangreicher als ein Web-Server ist ein *Online-GIS*. Auf Serverseite wird auf ein GIS zugegriffen, so dass die Verwendung komplexerer GIS-Funktionen möglich ist. Hierfür stellt der Client eine Anfrage über ein Internetprotokoll an einen HTTP-Server, der sie an eine GIS-Schnittstelle weiterleitet, wo die Anfrage zu GIS-spezifischen Kommandos ausgewertet werden kann. Diese werden an ein zu Grunde liegendes GIS weitergeleitet und dort interpretiert. Es folgt der Zugriff auf die geografischen Daten, die mit unterschiedlichen Skripten bearbeitet werden (z.B. Projektion, Clipping) und aus denen letztendlich eine Karte erstellt wird. Die so erzeugte Karte wird an die GIS-Schnittstelle zurückgegeben, von dort an den HTTP-Server weitergeleitet und die Antwort an den Client wird so auf Basis des Internetprotokolls fertig gestellt.

Das *Open Geospatial Consortium (OGC)* [OpGe2005] hat Schnittstellen für Map-Server vorgegeben, so dass die Möglichkeiten eines verteilten GIS noch stärker ausgenutzt werden können. Als Grundlage dient dabei die Standardisierung der Kommunikation zwischen den unterschiedlichen Systemen, die zur Datenaufbereitung dienen. Es gibt weiterhin einen Client, der aber nicht nur auf einen Server, sondern auf ein ganzes Serversystem zugreift. In diesem System kann ein Server die Rolle des Clients eines anderen Servers einnehmen. Die Geo-Daten müssen weder gesammelt auf einem einzigen Rechner liegen noch müssen sie auf Rechnern mit GIS-Funktionalität vorhanden sein. Sie existieren eigenständig im Netz. Auch die GIS-Funktionalitäten können nun auf viele Rechner, die unterschiedliche Dienste bereitstellen, verteilt sein. Entscheidend ist, dass das verwendete Dateiformat von allen Dienstleistern lesbar und schreibbar ist, damit Daten untereinander ausgetauscht werden können. Konkret sind die verschiedenen Dienste dafür zuständig, Daten, die auf vielen Rechnern in unterschiedlichen Formaten vorliegen, in ein einheitliches Datenformat zu bringen und gegebenenfalls zwischen unterschiedlichen Koordinatensystemen zu transformieren. Erst danach kann ein *Web Map Server* die Karte generieren und an den Benutzer weiterleiten. Die Entwicklung vom Geodaten-Server zu OGC-konformen Map-Servern zeigt deutlich den Unterschied vom Geografen als sein eigener Datenaufbereiter zum Geografen als Dienstleister, Bereitsteller und Organisator verschiedener Dienste. Am Beispiel des verteilten GIS erkennt man, dass durch das Zusammenspiel von der Auswahl der Daten, dem Datenzugriff, der Koordinatentransformation, den Zeichenvorschriften und der Visualisierung eine recht große Komplexität bei der Frage auftritt, in welcher Reihenfolge diese Aufgaben abgearbeitet werden müssen. In der vorliegenden Arbeit wird zwar kein OGC-konformer Map-Server entwickelt, aber die Aufgaben Datenauswahl, Datenauslesen, Transformationen, Wählen des Layouts und Darstellung müssen auch hier gelöst werden (Kapitel 8 und 9). So entsteht ein statischer

SVG Web Mapping Server, mit dem man beliebige Geo-Daten im Vektorgrafikformat visualisieren kann und mit dem möglichst viele Interaktionsmöglichkeiten des Benutzers mit dem Kartenmaterial verwirklicht werden.

II Umsetzung

8 Aufbau der Web Mapping Applikation

Die in dieser Arbeit entwickelte Applikation kann man als statischen Map-Server bezeichnen. Es handelt sich um eine Client-Server Applikation, bei der auf Anfrage des Clients die passenden, zuvor einmalig aus Shapefiles (Kapitel 6) berechneten Daten geliefert werden. Das Kartenmaterial besteht aus SVG-Grafiken (Kapitel 3). Die gesamte Interaktion auf Clientseite erfolgt mittels ECMAScript (Kapitel 4.1) und die Kommunikation von Client und Server mittels PHP (Kapitel 5).

Hauptziel war es, nicht nur eine Web Mapping Applikation zu entwickeln, die möglichst viele Interaktionsmöglichkeiten für den Benutzer bietet, sondern auch, dass beliebige im Shape-Format vorliegende Geodaten dargestellt werden können. Die „automatisierte“ Erstellung einer Web Mapping Applikation mit bestimmten Daten wird in Kapitel 9 erläutert. Zunächst wird in den folgenden Abschnitten der Aufbau der Web Mapping Applikation verdeutlicht.

8.1 SVG-Template

Auf Clientseite wird bei Aufruf der Applikation eine SVG-Datei geladen. Diese Datei wird als *Template* bezeichnet [KuMeVo2005], da sie im Grunde nur aus „Platzhaltern“ besteht. In Abbildung 8.1 sieht man, wie das leere Template aussieht und wofür die verschiedenen Platzhalter stehen.

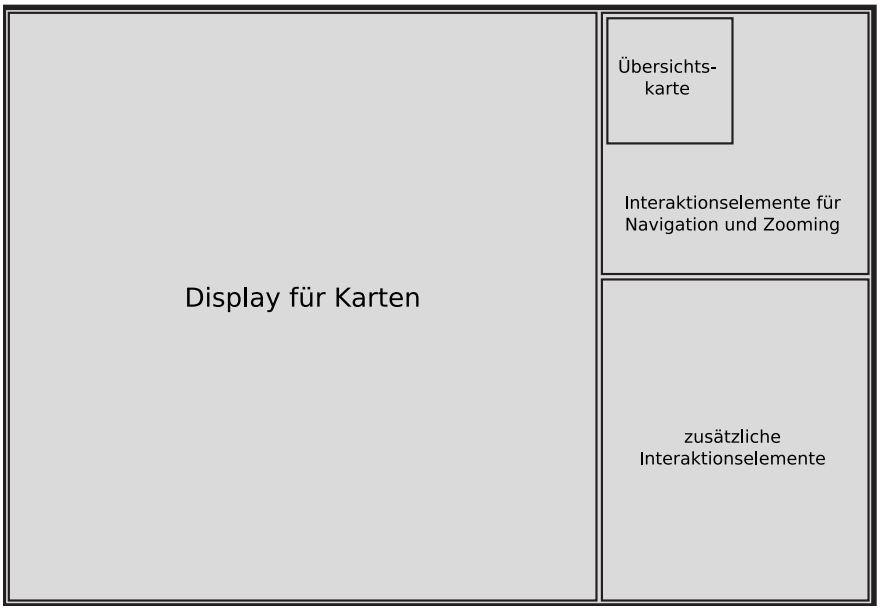


Abbildung 8.1: Das SVG-Template

Im Template selbst ist kein ECMAScript-Code eingefügt, es hat im Grunde lediglich die Aufgabe, die Daten an entsprechender Stelle aufzunehmen und anzuzeigen. Die Kontrolle über die Daten liegt beim Server, er hält alle Daten und entscheidet, welche Daten

in welchem Moment angezeigt werden sollen. Der Vorteil des Template-Konzeptes ist, dass das Template mit beliebigen Daten gefüllt werden kann. So ist lediglich festgelegt, dass im linken Bereich die Karten angezeigt werden. Es ist aber völlig frei, ob Landkarten, Stadtpläne oder andere Daten angezeigt werden, ebenso wie beim Bereich für die Übersichtkarte. Die Platzhalter für Navigations-Elemente und Buttons können Objekte aufnehmen, die durch Skripte erzeugt werden. Es ist wiederum nicht festgelegt, wie die Elemente auszusehen haben. Alle Platzhalter sind über eine ID ansprechbar. So kann zur Laufzeit das Template mit Hilfe von ECMAScript „gefüllt“ werden. Durch die Möglichkeit, die einzelnen Bereiche mit austauschbaren Skripten zu erzeugen, ist die Applikation grundsätzlich sehr flexibel und kann leicht mit speziellen (zusätzlichen) Funktionen versehen werden. Die einzelnen Bereiche sind dem Hauptdokument untergeordnete SVG-Dokumente.

In der Templatedatei ist ein eigenes Kontext-Menü definiert [xml2005], so dass man zum Beipsiel die Möglichkeit hat, die Größe des Templates zur Laufzeit zu verändern. Das Menü wird automatisch durch drücken der rechten Maustaste aufgerufen.

8.2 Interaktionsfunktionalitäten

Eine Web Mapping Applikation sollte gewisse Grundfunktionalitäten bezüglich der Interaktion erfüllen. Navigation und Zooming sind dabei die minimal nötigen Funktionalitäten. Um die Applikation komfortabler bedienen zu können, sind in der vorliegenden Applikation weitere Funktionen implementiert. Insgesamt hat man folgende Möglichkeiten:

1. Navigation (nord, süd, ost, west)
2. Ein- und Auszoomen
3. Vollansicht
4. Neues Kartenzentrum setzen
5. Neues Kartenzentrum setzen mit Ein- oder Auszoomen
6. Kartenausschnitt wählen
7. Karte verschieben (Panning)

Alle diese Funktionalitäten werden über ECMAScript verwirklicht. Im rechten oberen Bereich des Templates werden passende Symbole erzeugt, die auf Events reagieren und die gewünschte Aktion ausführen.

Will der Benutzer eine der Zoomaktionen ausführen, muss er zunächst auf das entsprechende Symbol klicken. In der Applikation wird unter den Symbolen der jeweilige Status angezeigt (aktuell gewählte Zoomfunktion) sowie die notwendigen Aktionen, die der Benutzer ausführen muss. Die jeweils gewählte Funktion bleibt solange ausgewählt, bis ein

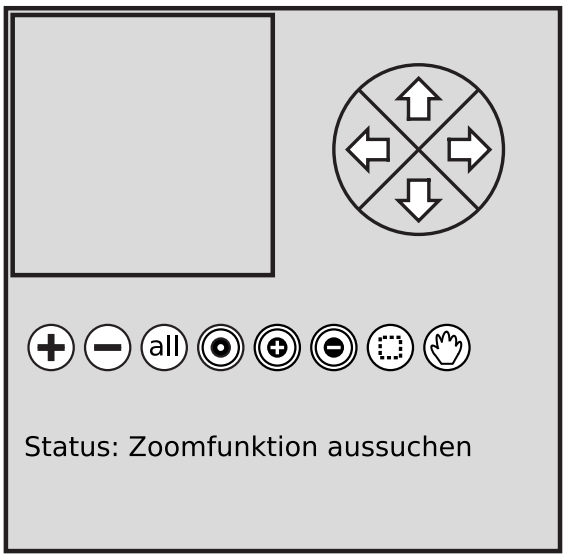


Abbildung 8.2: Navigations-Elemente

anderes Symbol angeklickt wird. Abbildung 8.2 zeigt die Navigations-Elemente. Neben den bereits genannten Interaktionsmöglichkeiten lässt sich außerdem der sichtbare Ausschnitt in der Übersichtskarte verschieben.
Die gezeigten Zoombuttons werden in der Funktion `appendZoomElements()` zur Laufzeit erzeugt. Jedes Element wird durch eine eigene Funktion erstellt, diese Teilfunktionen werden nacheinander aufgerufen. Durch diese Einteilung bleibt das Skript sehr modular, und es ist einfach, zusätzliche Symbole einzufügen. Auch die einzelnen Zoomfunktionen, die bei `onclick` auf ein Symbol ausgeführt werden, sind durch jeweils eigene Skript-Funktionen verwirklicht.

Alle genannten Navigationsmöglichkeiten beruhen auf dem Neusetzen der Viewbox eines SVG-Dokumentes (Kapitel 4.1), das in das Template eingebettet ist und die ID `display` besitzt. Die aktuellen Viewboxkoordinaten sind im Skript in globalen Variablen gespeichert. Wird nun gezoomt oder eine Bewegung ausgeführt, werden die neuen Koordinaten berechnet und gegebenenfalls die Breite mit einem Zoomfaktor multipliziert. Diese werden auf Gültigkeit bezüglich minimaler und maximaler Breite sowie Startkoordinaten der Viewbox geprüft, welche im ECMAScript ebenfalls in globalen Variablen gespeichert sind. Falls ungültige Werte entstehen, werden sie auf gültige Werte gesetzt. Vor der Neuberechnung werden die alten Koordinaten gemerkt, damit eine Meldung erscheinen kann, falls sich bei der ausgeführten Aktion nichts an der Ansicht ändert bzw. die Aktion nicht möglich ist. Sind die neuen Koordinaten bestimmt, wird das Attribut `viewBox` von `display` gesetzt und der Benutzer sieht den gewünschten Ausschnitt. Jedesmal, wenn die Viewbox neu gesetzt wird, kann es sein, dass neue Daten geladen werden müssen. Deshalb wird in diesem Fall immer eine Anfrage an den Server geschickt. Wie genau solch ein Anfrage-Vorgang abläuft, wird in Kapitel 8.4 beschrieben.

Für die Interaktionen selbst spielt die Event-Verarbeitung von ECMAScript die entscheidende Rolle. Durch die verschiedenen Event-Typen sind differenzierte Reaktionen auf Benutzeraktionen möglich (Kapitel 4.1). Die meisten Funktionen werden zwar auf einen

Mausklick (`onclick`) hin ausgeführt, jedoch sind für manche Aktionen andere Events unentbehrlich. Zum Beispiel ist für das Panning wichtig, dass nicht nur geklickt wird, sondern die Maustaste gedrückt gehalten wird. Man muss also insgesamt auf das erstmalige Drücken der Maustaste (`onmousepressed`) reagieren, um das Panning zu aktivieren, dann auf das Bewegen der Maus (`onmousemove`) und schließlich auf das Loslassen der Maustaste (`onmouseup`), um das Panning zu deaktivieren. Beim Aufziehen des Zoomrechteckes zum Wählen eines Ausschnittes ist es ähnlich.

Das `mouseover`-Event wird ebenfalls häufig abgerufen, beispielsweise um die Funktionalitäten der unterschiedlichen Zoombuttons anzuzeigen. Über das `mouseclick`-Event lassen sich auch die Koordinaten des Klicks bestimmen. Für die Applikation bietet sich dadurch die Möglichkeit, bestimmte Ausschnitte der Karte zu wählen oder die Karte per *drag and drop* zu verschieben. Um keine falschen Werte beim Abfragen der Koordinaten zu erhalten und die gesamte Kontrolle über das Zooming zu haben, wird die bereits vom Plugin zur Verfügung stehende Möglichkeit des Zoomens und Verschiebens innerhalb der SVG-Grafik abgeschaltet. Dazu wird das Attribut `zoomAndPan` des `svg`-Elementes im Template auf `disable` gesetzt. Die Möglichkeit des Ein- und Auszoomens ist im selbst definierten Kontext-Menü (Kapitel 8.1) ebenfalls nicht enthalten.

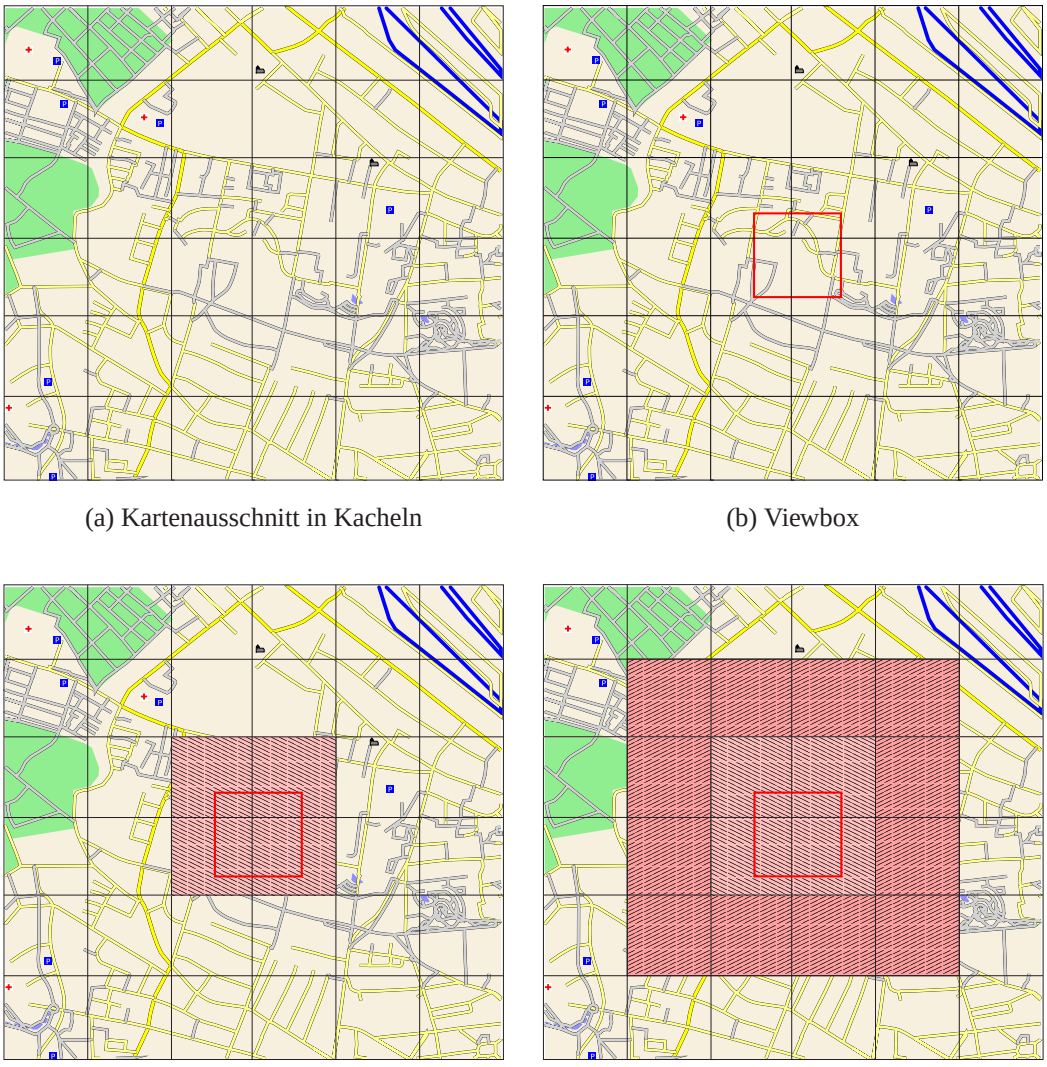
Im rechten unteren Teil des Templates befinden sich die zur Laufzeit von ECMAScript erzeugten Namen der vorhandenen *Layer* mit dazugehörigen Checkboxen, über die die Layer ein- und ausgeblendet werden können. Unter einem Layer versteht man eine der einzelnen Ebenen, in die eine geografische Karte eingeteilt sein kann. Beispielsweise könnten Seen und Flüsse auf einer Ebene liegen, Städte auf einer anderen. In der SVG-Applikation werden die Layer durch eine Gruppierung der Elemente in einem `g`-Tag mit entsprechender ID realisiert. Das Ein- und Ausblenden erfolgt über das Setzen des Attributes `visibility` der Gruppe. Die einzelnen Layer sind wiederum in zwei Ebenen eingeteilt. In die obere Ebene werden alle Texte oder Textpfade eingefügt, in die untere die geometrischen Elemente, weil Beschriftungen immer über ihren Objekten liegen müssen, um nicht verdeckt zu werden.

8.3 Umgang mit dem Kartenmaterial

Bevor ein Anfrage-Vorgang vom Client an den Server beschrieben werden kann, ist es notwendig zu wissen, wie mit dem anzuzeigenden Kartenmaterial umgegangen wird. Ganz allgemein werden die Karten zur Laufzeit als SVG in den linken Teil des Templates geladen. Sie werden vom Server geliefert, auf dem die vorberechneten SVG-Dateien liegen. Man könnte alle Daten in eine SVG-Datei schreiben und Ausschnitte lediglich über das Setzen der ViewBox anzeigen. Diese Vorgehensweise hätte jedoch entscheidende Nachteile. Plugins wie der ASV können nur eine begrenzte Anzahl von Objekten anzeigen und die Performanz leidet sehr, wenn viele Elemente im DOM sind. Die Anzeige von SVG-Dokumenten wird umso langsamer, je mehr Daten auf Clientseite verwaltet werden müssen. Um dieses Problem zu umgehen, werden die Daten während des Erzeugens der Karten einerseits nach Layern aufgeteilt in Dateien geschrieben und andererseits in Kacheln eingeteilt (Kapitel 9). So ist es möglich, in höheren Zoomstufen viel mehr Details anzuzeigen, ohne die Anzeigegeschwindigkeit wesentlich zu beeinträchtigen. Wenn

der Benutzer einen Ausschnitt der Karte betrachtet, werden nicht alle Daten geladen, sondern nur die relevanten. In höheren Zoomstufen können diese Teilkarten wesentlich kleiner sein als in niedrigen Zoomstufen.

Um zu verdeutlichen, welche Kacheln relevant sind, folgt ein Beispiel. Abbildung 8.3(a) zeigt schematisch einen gekachelten Stadtplan.



(a) Kartenausschnitt in Kacheln

(b) Viewbox

(c) Kacheln, die geladen werden müssen

(d) Kacheln, die geladen werden

Abbildung 8.3: Ermittlen der zu ladenden Kachlen

Das weitere Vorgehen zum Ermitteln der zu ladenden Kacheln ist wie folgt: Angenommen, der Betrachter will den rot markierten Ausschnitt sehen (Abbildung 8.3(b)), so genügt es zunächst die vier Kacheln zu laden, von denen jeweils ein Teil sichtbar ist wie in Abbildung 8.3(c) markiert. Bewegt sich der Betrachter auf der Karte nun zur Seite, so müssen spätestens in dem Moment, in dem er die nächste Reihe Kacheln „betritt“, diese nachgeladen werden. Um so auftretende Wartezeiten zu minimieren, werden nicht nur die vier bereits erwähnten Kacheln geladen, sondern um den sichtbaren Ausschnitt

herum jeweils eine Reihe Kacheln zusätzlich (Abbildung 8.3(d)). Bewegt sich der Betrachter nun auf der Karte, sind die Kacheln in dem Moment, in dem er sie erreicht, bereits vorhanden. Im Hintergrund wird dann bereits die nächste Reihe geladen und auf der entsprechend anderen Seite eine Reihe entfernt. Das erhöht zwar insgesamt den Traffic, hält aber die Datenmenge, die auf Clientseite verwaltet werden muss, gering. Durch die ebenfalls geringeren Datenmengen, die der Server auf einmal liefern muss, wird ebenfalls der Anfragevorgang selbst beschleunigt.

8.4 Datenaustausch

Wie in Kapitel 8.2 erwähnt, muss für jeden Navigations- oder Zoomvorgang bestimmt werden, ob neue Daten geladen oder Daten entfernt werden müssen. Die gesamte Kontrolle über die anzuzeigenden Daten liegt beim Server und wird über ein PHP-Skript 5 ausgeführt. Dazu teilt der Client dem Server seinen aktuellen Status mit und erhält als Antwort Auskunft über die zu löschenden und die zu ladenden Daten. Der Client selbst hat dabei keinen Einblick in die Datenstruktur und Skripte auf dem Server. So ließe sich leicht der gesamte Teil der Applikation auf Serverseite austauschen, falls ein anderes Konzept verfolgt werden soll (Kapitel 11).

8.4.1 Dateiverwaltung

Alle Kartengrafiken befinden sich in einem eigenen Verzeichnis. Die Dateien, die das Kartenmaterial enthalten, sind nach Layern und Zoomstufen bezeichnet. Ebenfalls ist im Namen codiert, welche Kachel in der Datei gespeichert ist. Die Dateinamen entsprechen den Namen der Layer in der Applikation. Daran angehängt und durch Unterstriche getrennt sind die Zoomstufe, zu der die Datei gehört, sowie die Kachelindizes. Die Kacheln sind nach Zeilen und Spalten indiziert. Heißt eine Datei beispielsweise `river_1_5_4.psvg`, bedeutet das, dass sie die Flussdaten der Kachel in Zeile 5 und Spalte 4 in Zoomstufe 1 enthält. In dieser Datei befinden sich alle Objekte innerhalb eines `g`-Tags, dass eine mit dem Dateinamen (ohne Dateiendung) identische ID hat.

Im gleichen Verzeichnis liegen weitere XML-Dateien, die angeben, ob in einer Zoomstufe Daten hinzugeladen oder komplett ausgetauscht werden. In diesen Dateien ist lediglich ein `g`-Tag enthalten, mit dem Attribut `action`. Dies kann die Werte `add` oder `replace` annehmen. Die Dateinamen sind nach Zoomstufe und Layer gewählt. Die Datei `river_1_action.xml` enthält beispielsweise die Angabe der Aktion, die für die Flussdaten ausgeführt werden muss, wenn man in Zoomstufe 1 gelangt. Wie genau die Angaben in den `action`-Dateien ausgewertet werden, ist in Kapitel 8.4.2 beschrieben.

8.4.2 Ermitteln der zu ladenden Daten

Um zu entscheiden, welche Daten für einen bestimmten Ausschnitt geliefert werden müssen, benötigt der Server die aktuellen Viewboxkoordinaten vom Client. Im PHP-Skript wird über die Angabe der Viewboxkoordinaten berechnet, in welcher Zoomstufe

und auf welchen Kacheln man sich befindet. Die Zoomstufe wird über die Viewboxbreite berechnet: In einem Array sind die maximalen Werte der Viewboxbreite für eine bestimmte Zoomstufe gespeichert, beginnend mit der maximalen Breite. Der aktuell übermittelte Wert wird mit den Einträgen verglichen. Ist er kleiner als der Eintrag im Array, aber größer als der nachfolgende, entspricht der Index des Eintrags der Zoomstufe. Über die Kachelbreite und die gegebenen Koordinaten wird bestimmt, auf welcher Kachel die linke obere Ecke der Viewbox liegt und auf welcher die rechte untere Ecke. Von den kleineren Indizes wird 1 subtrahiert, zu den größeren 1 addiert. So erhält man die minimalen und maximalen Kachelkoordinaten.

Aus den so erhaltenen Werten werden die zu ladenden Dateinamen generiert. Dazu sind alle Layernamen und alle vorhandenen Dateinamen jeweils in einem Array gespeichert. Die Namen der potenziell zu ladenden Dateien lassen sich wie folgt zusammensetzen:

```
layer_zoomstufe_zeilenindex_spaltenindex.psvg
```

Die Dateien enthalten SVG-Segmente, also Teilstücke eines vollständigen SVG-Dokuments (im Englischen *parts*). Daher wurde die neue Dateiendung `.psvg` als Abgrenzung zu vollständigen SVG-Dokumenten gewählt.

Bei der Generierung der Dateinamen werden alle Layernamen im Array durchlaufen. Es kann vorkommen, dass Namen von nicht existierenden Dateien generiert werden, daher wird im Dateinamen-Array nachgesehen, ob es diese Datei tatsächlich gibt. Ist dies der Fall, wird der Dateiname in einem weiteren Array gespeichert. Nach den vollständigen Durchlaufen des Layer-Arrays, liegen alle zu ladenden Dateien für die aktuelle Zoomstufe vor. Es kann jedoch sein, dass zusätzlich Daten aus vorherigen Zoomstufen mitgeladen werden müssen. Ob dies der Fall ist, wird anhand der Angabe in der `action`-Datei für den entsprechenden Layer und die aktuelle Zoomstufe überprüft. Sie wird geparst und an dem erhaltenen Knoten das `action`-Attribut abgefragt. Handelt es sich um den Wert `add`, wird die Berechnung der Kachelindizes und die Generierung der Dateinamen erneut für die vorherige Zoomstufe ausgeführt, die PHP-Funktion ruft sich also rekursiv auf. Ist als Attribut in der `action`-Datei `replace` angegeben wird die Aufrufkette abgebrochen. Es kommt solange zum rekursiven Aufruf, wie Daten aus vorherigen Zoomstufen hinzugefügt werden sollen oder bis Zoomstufe 0 erreicht ist. Vor dieser Zoomstufe existieren keine Daten, und somit stellt dies die Bedingung zum Rekursionsabbruch dar.

8.4.3 XML-Ausgabe erzeugen

Für die Entscheidung, ob der Client neue Daten geliefert bekommen muss oder ob Daten gelöscht werden müssen, werden sowohl die alten als auch die neuen Koordinaten benötigt. Eine Möglichkeit wäre, dass der Server für jede Anfrage abspeichert, welche Daten aktuell beim Client vorhanden sind. Dies wäre mit PHP zwar mit Hilfe von *Sessions* möglich, wirft aber Probleme auf. Es muss festgelegt werden, nach welcher Dauer eine Session beendet wird, was nicht immer sinnvoll zu entscheiden ist. Bei zu kurzer Dauer würde man den Benutzer eventuell mitten in seiner Tätigkeit unterbrechen, bei zu langer Dauer wäre unter Umständen relativ viel Speicher auf dem Server unnötig belegt.

Um solchen Problemen zu entgehen, werden dem Server bei dieser Applikation sowohl die neuen (`vbXt`, `vbYt`, `vWXt`) als auch die alten Viewboxkoordinaten (`oldvbXt`, `oldvbYt`, `oldvWXt`) mitgeteilt. Sie werden bei Aufruf des PHP-Skriptes auf Client-Seite als Parameter mitgegeben. Der notwendige ECMAScript-Code sieht wie folgt aus:

```
getURL(unescape('./script/getdata.php?X='+vbXt+'&Y='
       +vbYt+'&W='+vbWt+'&oldX='+oldXt+'&oldY='+oldYt
       +'&oldW='+oldWt+''),manage_loading);
```

Das PHP-Skript ermittelt für beide Fälle wie in Kapitel 8.4.2 beschrieben die zu ladenden Dateien. Dabei werden zwei Arrays erzeugt: Im Array `oldids` sind dann die Dateinamen für die alten Koordinaten enthalten, im Array `ids` die Namen der Dateien, die für die neuen Koordinaten geladen werden müssen. Die Dateien, deren Namen in beiden Arrays stehen, brauchen nicht mehr geladen zu werden, da sie bereits bei einem vorherigen Aufruf an den Client geliefert worden sind. Entfernt man diese aus beiden Arrays, erhält man in `oldids` die Dateien bzw. die IDs der bereits geladenen Daten, die aus der Applikation gelöscht werden müssen. In `ids` sind die Dateien enthalten, die nachgeladen werden müssen:

```
$common=array_intersect($oldids,$ids);
$oldids=array_values(array_diff($oldids,$common));
$ids=array_values(array_diff($ids,$common));
```

Mit diesen Informationen wird im PHP-Skript ein DOM aufgebaut, dessen XML-Repräsentation an den Client geschickt wird. Es hat als Wurzelelement ein `data`-Tag, als untergeordneten Knoten ein `delete`- und ein `append`-Tag. Diese Verschachtelung ist notwendig, weil sich das SVG-Plugin von Adobe in unterschiedlichen Versionen nicht gleich verhält (Kapitel 4.2).

```
$dom=new DomDocument();
$del=$dom->createElement('delete');
$app=$dom->createElement('append');
$data=$dom->createElement('data');
$dom->appendChild($data);
$data->appendChild($del);
$data->appendChild($app);
```

Ohne weitere Daten über die zu löschenden und zu ladenden Daten sieht die Stringrepräsentation dieses DOM wie folgt aus:

```
<data>
  <delete>
  <delete/>
  <append>
  <append/>
</data>
```

Der `delete`-Knoten

In den `delete`-Knoten werden die Informationen für die zu löschenden Daten eingefügt. Dazu wird lediglich für jede Kachel ein `g`-Element erzeugt, das die ID der zu löschenden Gruppe als Attribut erhält. Diese Elemente werden als Kindknoten eingefügt.

```
for($i=0;$i<sizeof($oldids);$i++){
  $g=$dom->createElement('g');
  $g->setAttribute('id',$oldids[$i]);
  $del->appendChild($g);
}
```

Der `append`-Knoten

In der Applikation sind zwei unterschiedliche Wege verwirklicht, wie dem Client die zu ladenden Daten mitgeteilt werden. Einerseits besteht die Möglichkeit, dass auf Serverseite die Kartendateien bereits ausgelesen werden und diese als Kindknoten an das `append`-Tag angehängt werden. Zuvor werden sie mit der ID ihres zukünftigen Elternknotens in der Applikation als `parent`-Attribut versehen, damit auf Clientseite bekannt ist, wo die Gruppe in den DOM-Tree eingehängt werden muss. Die ID wird aus dem Dateinamen ermittelt, die Informationen, zu welcher Kachel und Zoomstufe die Gruppe gehört, sind für den Elternknoten irrelevant. Im Skript sieht diese Vorgehensweise wie folgt aus:

```
for($j=0;$j<sizeof($ids);$j++){
   $id=$ids[$j];
   $filename ='../svgfiles/'.$id.'.psvg';
   $id=split('_',$id);
   $parent='';
   for($k=0;$k<(sizeof($id)-4);$k++){
   $parent=$parent.$id[$k].'_';
   }
   $parent=$parent.$id[(sizeof($id)-4)];
   if(file_exists($filename)){
      $handle = fopen ($filename, "r");
      $buffer=fread ($handle, filesize ($filename));
      fclose($handle);
      $dom2=new DomDocument();
      $dom2->loadXML($buffer);
      $list=$dom2->getElementsByTagName('g');
      $g=$list->item(0);
      $g->setAttribute('parent',$parent);
      $neu=$dom->importNode($g,true);
      $app->appendChild($neu);
   }
}
```

Mit den ausgelesenen Daten einer Kachel wird durch die Funktion `loadXML(string)`, die an einem `DOMDocument`-Objekt aufgerufen wird, der Inhalt der Kachel als DOM-Tree erzeugt. Dessen Wurzelknoten wird in das Dokument, das die Ausgabe für den Client enthält, importiert. Anschließend werden Informationen über den Elternknoten in der Applikation als Attribut eingefügt und der Knoten als Kindelement an den `delete`-Knoten angehängt.

Die zweite Variante, dem Client mitzuteilen, welche Daten geladen werden müssen, ist das Übermitteln der zu ladenden Dateinamen. Der Client bekommt dann die Namen der Dateien und die IDs der Elternknoten genannt und holt selbst die Daten vom Server.

```
for($j=0;$j<sizeof($ids);$j++){
   $id=$ids[$j];
   $filename ='../svgfiles/'.$id.'.psvg';
   if(file_exists($filename)){
      $filename='svgfiles/'.$id.'.psvg';
      $id=split('_',$id);
      $parent='';
      for($k=0;$k<(sizeof($id)-4);$k++){
          $parent=$parent.$id[$k].'_';
      }
      $parent=$parent.$id[(sizeof($id)-4)];
      $g=$dom->createElement('g');
      $g->setAttribute('file',$filename);
      $g->setAttribute('parent',$parent);
      $app->appendChild($g);
   }
}
```

Die Ausgabe

Das so erzeugte DOM wird mit der Funktion `saveXML()` zu einer Stringrepräsentation umgewandelt und überflüssige Angaben aus dem String entfernt. Die XML-Versionsangabe muss entfernt werden, damit die Funktion `parseXML()` auf Clientseite den übergebenen String verarbeiten kann. Die Angaben zum Namespace für das `xlink`-Attribut sind auf Clientseite bereits vorhanden, müssen also nicht noch mal angegeben werden. Mit dem Aufruf von `trim(string)` werden überflüssige Leerzeichen entfernt und mit `gzencode(string)` der String gezipt. Dieser String wird mit `echo` zurückgeliefert:

```
$ausg=$dom->saveXML();
$ausg=str_replace('<?xml version="1.0"?>','',$ausg);
$ausg=str_replace(
      'xmlns:xlink="http://www.w3.org/1999/xlink"',
      '',$ausg);
$ausg=trim($ausg);
$ausg=gzencode($ausg);
echo $ausg;
```

Weiter oben ist der Aufruf des PHP-Skriptes aus ECMAScript heraus angegeben. Die Funktion `getURL` hat als zweiten Parameter einen Funktionsnamen. Die Ausgabe des PHP-Skriptes wird an diese Funktion weitergereicht. Die Funktion `manage_loading()` bekommt also die erhaltenen Daten als Parameter übergeben und verarbeitet sie weiter (Kapitel 4.2). Je nachdem mit welcher Variante die zu ladenden Daten übermittelt werden, muss diese Funktion unterschiedlich aufgebaut sein. Wie das Einfügen der Daten in den SVG-DOM-Tree erfolgt, wird im folgenden Kapitel beschrieben.

8.4.4 Daten in DOM-Tree löschen und einfügen

Die ECMAScript-Funktion `manage_loading(data)` ist für das Löschen und das Einfügen des Kartenmaterials in der Applikation zuständig. Die Ausgabe des PHP-Skriptes wird auf Clientseite wieder in einen DOM-Tree umgewandelt und die so vorliegenden Informationen verarbeitet. Im Folgenden werden nur Auszüge aus der Funktion gegeben, um die Übersichtlichkeit zu gewährleisten. Ausführliche Angaben zum Laden von XML-Daten in das DOM sind bereits in Kapitel 4.2 gegeben, weshalb nicht jeder Funktionsaufruf hier genau erläutert wird.

Zunächst wird vom DOM, der aus der PHP-Ausgabe erzeugt wird, der `delete`- und der `append`-Knoten geholt:

```
st=data.content;
var node=parseXML(st,document);
var child=node.childNodes;
var dat=child.item(0);
var del=dat.firstChild;
var app=dat.lastChild;
```

Daten löschen

Die Daten, die gelöscht werden müssen sind einfach zu ermitteln. Jeder Kindknoten des `delete`-Elementes enthält die ID einer zu löschenden Gruppe. Die Liste der Kindknoten wird durchlaufen, über die angegebene ID das Element geholt und mit der Funktion `removeChild(child)` vom Elternknoten entfernt:

```
var list=del.childNodes;
for(j=0;j<list.length;j++){
   el=list.item(j);
   id=el.getAttributeNS(null,'id');
   child=document.getElementById(id);
   if(child!=null){
      child.parentNode.removeChild(child);
   }
}
```

Werden Navigations- oder Zoomvorgänge relativ schnell hintereinander ausgeführt, kann es aufgrund der Nebenläufigkeit von ECMAScript zu Problemen kommen. Die Funktion

`manage_loading()` wird gegebenenfalls mehrfach parallel ausgeführt. Es kann dann vorkommen, dass versucht wird, eine Gruppe zu löschen, bevor sie überhaupt eingefügt wurde. Da in der Funktion mit dem Test auf `null` abgefragt wird, ob das zu löschende Element existiert kommt es zwar zu keinem Fehler, das Element wird allerdings noch eingefügt. So passiert es, dass nach Abarbeitung aller Aufrufe von `manage_loading()` nicht die gewünschten Daten angezeigt werden, sondern noch Daten aus anderen Zoomstufen oder Ausschnittsansichten zurück bleiben können. Um dieses Problem zu lösen, wird bei Nichterfolg des Löschens die ID in einem globalen Array gespeichert. Nachdem die zu ladenden Daten abgearbeitet sind wird noch mal auf dieses Array zurückgegriffen und die Elemente mit den enthaltenen IDs gelöscht. So ist sicher gestellt, dass spätestens beim letzten Aufruf der Funktion alle Elemente, die gelöscht werden müssen auch tatsächlich entfernt werden. Bei Einfügen der IDs in das globale Array wird darauf geachtet, dass IDs nicht doppelt eingetragen werden, um mehrfache Löschversuche und somit überflüssige Funktionsaufrufe zu umgehen. Der Code der gesamten Funktion ist im Anhang B und C zu finden.

Daten einfügen

In Kapitel 8.4.2 wurden zwei Varianten vorgestellt, wie dem Client mitgeteilt wird, welche Daten geladen werden müssen. Je nachdem, welche Variante gewählt wird, muss der Client anders reagieren, die Funktion `manage_loading()` also anders implementiert sein.

In der Variante, in der alle Daten vollständig an den Client geliefert werden, muss der Client lediglich bei allen Kindern des `append`-Knotens das `parent`-Attribut entfernen und den Knoten an den angebenen Elternknoten anhängen:

```
list=app.childNodes;
while(list.length!=0){
   el=list.item(0);
   p=el.getAttributeNS(null,'parent');
   id=el.getAttributeNS(null,'id');
   el.removeAttributeNS(null,'parent');
   document.getElementById(p).appendChild(el);
}
```

Es ist wichtig zu beachten, dass mit dem Einfügen des Elementes in den DOM-Tree automatisch das Element aus der Kindknotenliste des `append`-Elementes entfernt wird. Dies ist der Fall, weil ein Knoten nur jeweils zu einen Dokument gehören kann (Kapitel 2). Um die Elemente aus der Kindknotenliste des `append`-Knotens in das Dokument der Web Mapping Applikation einzufügen, ist es daher sinnvoll, eine `while`-Schleife zu verwenden. Würde man innerhalb einer `for`-Schleife über eine Zählvariable auf die Liste zugreifen, liesse man einzelne Knoten aus.

In der Variante, bei der der Client selbst die Daten vom Server holt, wird ist ein etwas anderer Code notwendig:

```
  list=app.childNodes;
  for(j=0;j<list.getLength();j++){
     el=list.item(j);
     p=el.getAttributeNS(null,'parent');
     file=el.getAttributeNS(null,'file');
     appendFile(file,p);
  }
```

Die Knoten in der Kindliste besitzen ein `file`-Attribut, das den Pfad der zu ladenden Datei auf dem Server enthält. Die Funktion `appendFile(file,parent)` ist eine selbst implementierte Funktion, in der letztendlich wiederum die Funktion `getURL` aufgerufen wird. Es ergibt sich hier allerdings das Problem, dass die gelieferten Daten an ein bestimmtes Element im DOM angehängt werden sollen. Die Callback-Funktion, die an `getURL` übergeben wird, darf aber nur einen Parameter besitzen. So muss für jeden unterschiedlichen Elternknoten eine eigene Callback-Funktion vorliegen. In `appendFile` wird dann je nach übergebener ID die passende Funktion aufgerufen. Quellcode 8.1 zeigt beispielhaft die nötigen Funktionen, wenn in der Applikation nur die zwei Layer `river` und `country` vorhanden wären.

```
function appendFile(fileName,parent){
   if(parent=='river')getURL(fileName,getData_river);
   if(parent=='country')getURL(fileName,getData_country);
}
function getData_river(data){
   appendData(data,'river');
}
function getData_country(data){
   appendData(data,'country');
}
function appendData(data,parent){
   if(data.success){
      string=data.content;
      node=parseXML(data.content,svgDocument);
      document.getElementById(parent).appendChild(node);

   }
}
```

Quellcode 8.1: Eigene Callback-Funktionen für jeden Layer

8.4.5 Vor- und Nachteile der Varianten der Datenlieferung

In der vorliegenden Arbeit besteht die Möglichkeit zwischen den beiden Versionen der Datenlieferung zu wählen (Kapitel 9.5), weil nicht klar zu entscheiden ist, welche Variante besser ist. Im ersten Fall ist die auf einen Anfragevorgang hin übermittelte Datenmen-

ge erheblich größer, wodurch die Anfrage selbst länger dauert. Alle zu ladenden Daten müssen ausgelesen werden und komplett an den Client gesandt werden, so dass dieser nicht schon eher mit dem Laden der ersten Daten beginnen kann. Im zweiten Fall muss der Client sich alle Daten einzeln selbst holen. Es werden also häufig kleinere Datenmengen übertragen, was zwar insgesamt länger dauert, als einmal eine große Menge zu übertragen, dafür kann aber während des Datenholens bereits der auf Clientseite vorhandene Teil der Daten angezeigt werden. Bei schmaler Bandbreite ist diese Variante daher vorzuziehen.

9 Aufbau des Java-Programms

Die entstehende Web Mapping Applikation soll für beliebige Geodaten nutzbar sein. Dazu muss festgelegt werden, welche Daten angezeigt werden sollen, welche Layer es geben soll, wie die Daten aussehen sollen (Farben, Liniendicken, Schriftgröße) und wieviele Zoomstufen existieren sollen. Als Basis für die Geodaten dient das Shapefileformat (Kapitel 6), aus dem die Karten als SVG-Grafiken erstellt werden müssen. Die Karten sollen nicht nur die gesamten Daten enthalten, sondern wie in Kapitel 8.3 beschrieben nach Layern gegliedert sein und für höhere Zoomstufen als Kacheln vorliegen. Um jemandem, der Karten im Internet veröffentlichen will, dieses Vorhaben möglichst einfach zu gestalten, müssen alle nötigen Angaben nach einem bestimmten Schema (Anhang A) in eine Konfigurationsdatei geschrieben werden. Aus den dort gemachten Angaben werden mit einem Java-Programm alle Kartengrafiken sowie angepasste Skripte erzeugt und alles in einer bestimmten Ordnerstruktur abgelegt. Alle SVG-Grafiken und Skripte, die vom Programm erzeugt werden, müssen dann nur noch auf einen Webserver mit PHP5-Unterstützung gelegt werden.

Um eine bessere Vorstellung der Arbeitsweise des Programms zu bekommen, ist in Abbildung 9.1 der Datenfluss dargestellt. Die einzelnen Schritte im Datenfluss werden innerhalb dieses Kapitels näher erläutert.

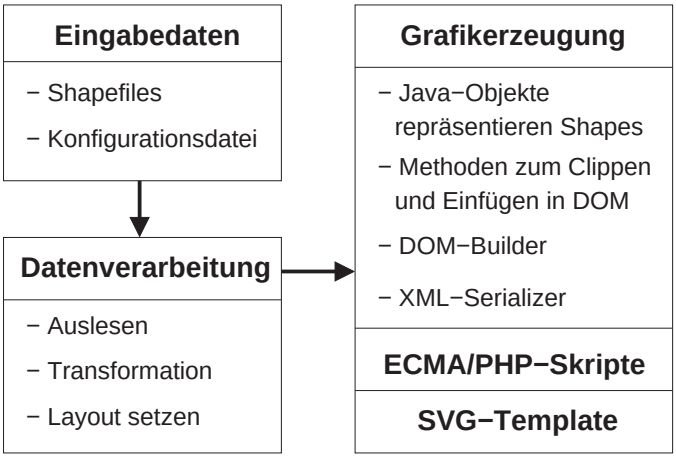


Abbildung 9.1: Datenfluss des Java-Programms

Um das Kartenmaterial erstellen zu können, müssen die geografischen Objekte zunächst als Java-Objekte mit bestimmten Eigenschaften erzeugt werden, was in Kapitel 9.1 und 9.3 ausführlich erläutert wird. Der Benutzer des Programms muss im Besitz von Geodaten im Shapefileformat sein und eine XML-Konfigurationsdatei schreiben können. In Kapitel 9.5 wird der Aufbau dieser Konfigurationsdatei erläutert. Die eigentliche Verarbeitung der Konfigurationsdatei und damit die Hauptaufgabe des Programms beschreibt Kapitel 9.6.

9.1 Grafikobjekte in Java

Beim Erzeugen des Kartenmaterials werden zum Teil aufwändige Transformationen und Algorithmen aus der Computergrafik durchgeführt. Diese lassen sich nur auf Grafikobjekte anwenden, die auch als solche im Speicher vorliegen. Dazu werden Java-Klassen benötigt, die eine Schnittstelle zwischen Java und Shapefiles schaffen. Diese Klassen sind in der Lage, bestimmte Shapes (Polygone, Polylinien, Punkte) zu repräsentieren, es sind also Grafikobjekte mit ihren Koordinaten und Attributen. Weiterhin ermöglichen festgelegte Methoden, die SVG-Struktur eines Grafikobjekts in einen DOM-Tree einzufügen. Um aus den in Binärformat als Shapefiles vorliegenden Daten SVG-Grafiken zu erzeugen, benötigt man Klassen, die zweidimensionale Shapetypen repräsentieren. Dreidimensionale Objekte müssen nicht berücksichtigt werden, weil SVG ein zweidimensionales Grafikformat ist (Kapitel 3).

9.1.1 Java-Klassenhierarchie

Alle verwendeten Grafikobjekte müssen einige gemeinsame Eigenschaften haben. Im Wesentlichen müssen sie sich als SVG-Struktur in einen DOM-Tree einfügen können, dabei gegebenenfalls vorher Clippen und eine Beschriftung liefern können. Im Speicher müssen beim Erzeugen des Kartenmaterials relativ viele Grafikobjekte verwaltet werden, an denen immer die gleichen Methodenaufrufe erfolgen. Daher ist es sinnvoll, Grafiklisten zu verwenden, die die entsprechenden Methoden enthalten und bei Aufruf an die enthaltenen Objekte weiterreichen. Dieses Konzept bezeichnet man als *Composite Pattern* [GHJV1995] [Tarr2004]. Abbildung 9.2 verdeutlicht das grundlegende Vorgehen. Man definiert eine `Composite`-Klasse, die bestimmte Methoden hat, hier die Methode `operation`. Ein Composite kann wiederum andere Composites enthalten, diese wiederum andere und so weiter. Diese Objekte leiten den Methodenaufruf an die enthaltenen Composites weiter. Es existieren außerdem die `Leafs`, die selber Composites sind, aber keine weiteren Composites enthalten können. Sie bilden also das Endglied der Aufrufkette von `operation`. Dieses Konzept hat den Vorteil, dass man alle Grafikobjekte gleich behandeln kann. Man braucht sich nicht darum zu kümmern, ob man eine Liste oder ein einzelnes Objekt vorliegen hat und muss sich keine Gedanken um den speziellen Typ machen. Grafikobjekte, die unterschiedliche Shapes repräsentieren, können gemeinsam verwaltet werden, die entsprechenden Methoden werden an den „Containern“ aufgerufen. Abbildung 9.3 zeigt ein stark vereinfachtes UML-Diagramm der in der Arbeit verwendeten Klassen zum Verwalten und Repräsentieren von Grafikobjekten.

Eine abstrakte Klasse `Graphic` definiert diejenigen Eigenschaften, die alle Grafikobjekte haben müssen. Hierbei handelt es sich um die Methoden `appendTo(element)`, `appendTo(element,point,point)`, `appendLabelTo(element,int)` und `appendLabelTo(element,point,point,int)`. Mit der Methode `appendTo` wird ein Objekt dazu aufgefordert, sich als SVG-Struktur an den übergebenen Knoten anzuhängen. Gibt man zwei Punkte mit, werden diese als linke obere und rechte untere Ecke eines Clipping-Rechtecks verwendet. Die Java-Objekte können sich jeweils anhand dieses Rechteckes „zuschneiden“ und fügen sich erst danach an den übergebenen Knoten an. Mit `appendLabel` können Objekte ihre Beschriftung erzeugen, je nach Aufruf

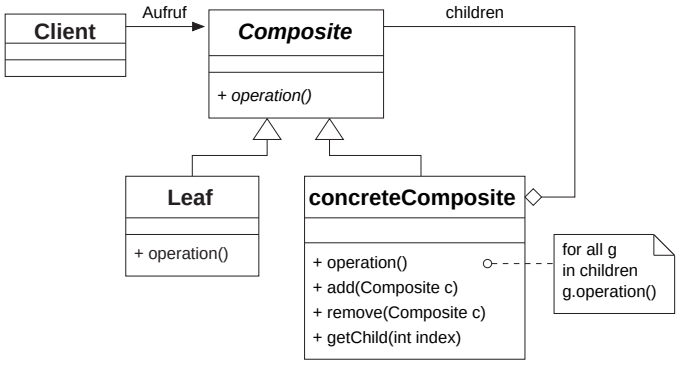


Abbildung 9.2: Das Composite Pattern

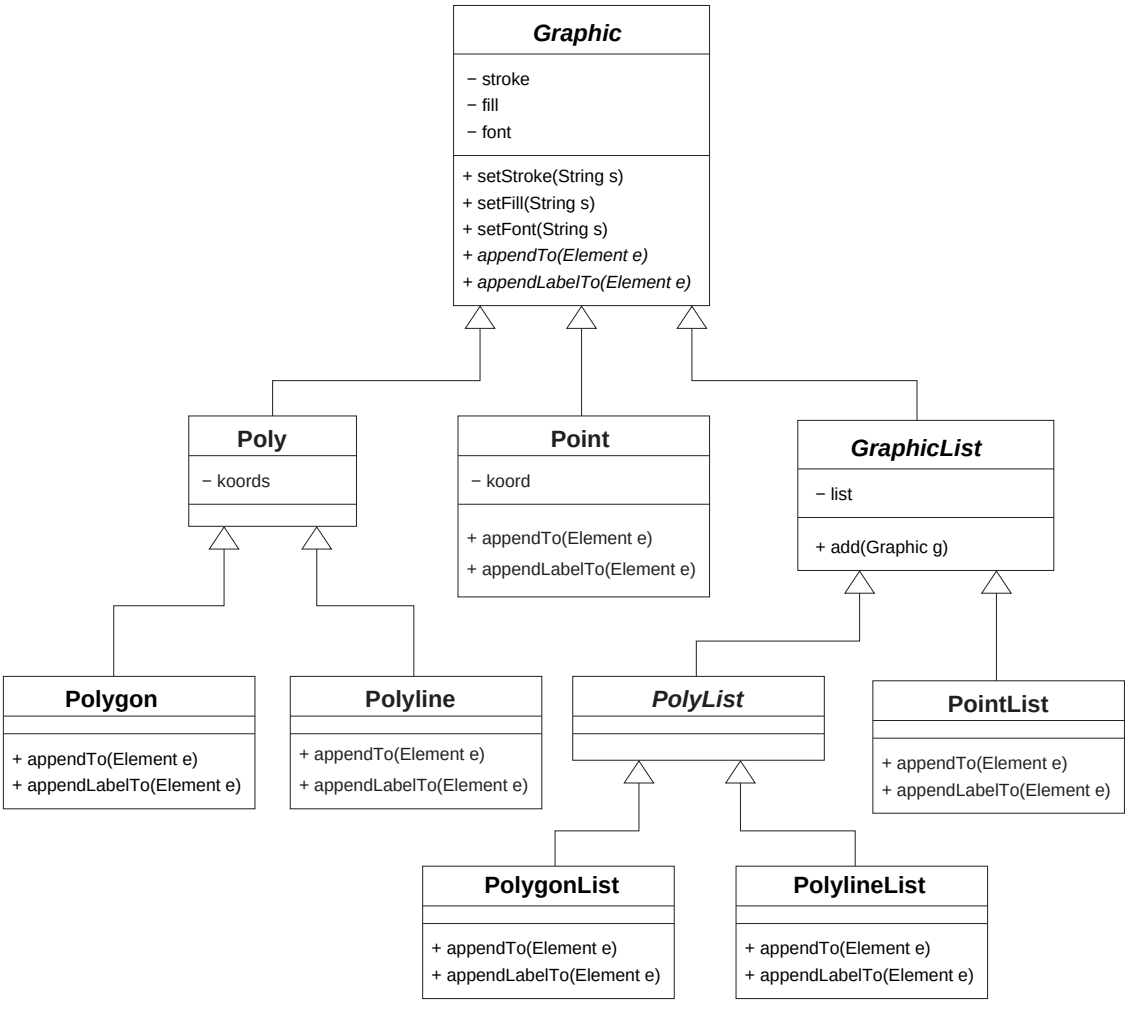


Abbildung 9.3: Vereinfachtes UML-Diagramm der Java-Klassen

auch geclippt. Eine weitere abstrakte Klasse ist die Klasse `GraphicList`, die eine Schnittstelle zu den Klassen für das Verwalten der Grafikobjekte darstellt. Alle Grafiklisten leiten Methodenaufrufe an die enthaltenen Objekte weiter. Für die Polygone und Polylinien (und die entsprechenden Listen) existiert eine gemeinsame Oberklasse, die unter anderem dem einfacheren Verwalten der Objekte dient. Im folgenden Kapitel 9.2 wird auf die einzelnen Grafikklassen näher eingegangen und die jeweiligen Vorgehensweisen beim Clipping beschrieben.

9.2 Einzelne Klassen und ihre Eigenschaften

Für die Klassen `Point`, `Polyline` und `Polygon` wird die Hilfsklasse `MyPoint` verwendet. Diese repräsentiert einen zweidimensionalen Punkt. Sie wird verwendet, um die Vektordaten der Grafikobjekte zu speichern und enthält Informationen, zu welchen `Graphic`-Objekten der Punkt gehört. Ein `Point`-Objekt benötigt nur ein `MyPoint`-Objekt, die `Poly`-Klassen enthalten jeweils eine Liste von `MyPoint`-Objekten, um die Punkte in passender Reihenfolge zu speichern. Alle Klassen haben ein Datenfeld, in dem die Informationen aus der `dbf`-Datei gespeichert werden und für die Beschriftung verwendet werden können.

Die Beschriftungen der Objekte in den Karten überlappen zum Teil, weil alle Beschriftungen auf relativ einfache, intuitive Weise erfolgen. Beispielsweise liegen in dicht besiedelten Gegenden wie dem Ruhrgebiet dadurch Städtenamen sehr nah beieinander und eine Platzierung der Städtenamen ohne Überlappung in lesbarer Schriftgröße ist nicht möglich. Als Geograf stellt man sicherlich den Anspruch einer besseren Beschriftung. Das Problem, Karten automatisiert zu beschriften, ist jedoch `NP`-vollständig [Sch1995]. Es existieren zwar Heuristiken [Wolf2002], die gute Lösungen liefern, jedoch wird dieser Bereich in der vorliegenden Arbeit nicht behandelt, da er mit der eigentlichen Aufgabenstellung nur indirekt zusammenhängt.

Alle Grafikobjekte haben verschiedene Felder, in denen Layoutinformationen abgespeichert werden können. Wenn ein Objekt sich in einen DOM-Tree einfügen soll, so verwendet es für Farbangaben, Schriftgröße und andere Layoutangaben diese Felder.

Um den Rahmen der Arbeit nicht zu sprengen, handelt es sich bei den folgenden Ausführungen nicht um eine vollständige Beschreibung der Klassen,. Die komplette Dokumentation ist auf der beiliegenden CD zu finden. Die in diesem Kapitel betrachteten Methoden sind diejenigen, bei denen entweder Probleme, die durch das Verwenden von SVG als Grafikformat entstehen, gelöst werden mussten oder die Methoden liefern wichtige Eigenschaften der Grafikobjekte, auf die bisher noch nicht eingegangen wurde.

Die Klasse `Point`

Die Klasse `Point` repräsentiert Punktdaten. Sie besitzt als Speicherplatz für die Vektordaten lediglich ein `MyPoint`-Objekt. Für das Clipping wird anhand der Koordinaten überprüft, ob der Punkt innerhalb oder außerhalb des Clippingrechteckes liegt. Soll der Punkt beschriftet werden, wird die Schrift anhand der Punktkoordinaten platziert. Gegebenenfalls wird sie daran links, rechts oder zentriert ausgerichtet.

Die Klasse `Poly`

Die Klasse `Poly` enthält alles, was Polygone und Polylinien gemeinsam haben. Abgesehen von einer Liste von `MyPoint`-Objekten besteht die gemeinsame Eigenschaft, die eigenen Datenpunkte generalisieren zu können, also die Anzahl der Punkte zu reduzieren. Ein solches „Vereinfachen“ der Daten erfolgt über den Aufruf der Methode `simplify()`. Dabei muss beim Entfernen von Punkten entschieden werden, ob ein Punkt anhand eines bestimmten Kriteriums gelöscht werden darf. Zusätzlich muss überprüft werden, ob er in anderen Polygonen oder- linien vorhanden ist. Ist dies der Fall, könnte es passieren, dass durch das Löschen ursprünglich aneinander liegende Linien nicht mehr zusammen passen. Das hätte bei geografischen Karten beispielsweise den Effekt, dass Grenzen nicht mehr übereinstimmen, was natürlich vermieden werden soll.

In der Methode `simplify()` werden alle Punkte eines `Poly`-Objekts durchlaufen und mit der Methode `is_deletable(point,point,point,double)` geprüft, ob der Punkt gelöscht werden darf. Von den drei übergebenen Punkten ist der mittlere der Punkt, der geprüft wird. Die beiden anderen Punkte sind Vorgänger und Nachfolger innerhalb der Punktliste. In `is_deletable` wird die Determinante der Matrix berechnet, die die beiden Vektoren, die vom mittleren Punkt jeweils zu einem der beiden anderen verlaufen, als Spalten enthält. Der Absolutbetrag der Determinante von reellen Vektoren ist gleich dem Volumen des Parallelepipeds (auch Spat genannt), das durch diese Vektoren aufgespannt wird [Wiki2005]. So ergibt sich ein Maß der Fläche des von den Vektoren eingeschlossenen Parallelogramms. Ist die eingeschlossenene Fläche klein genug, kann man den Punkt, von dem die Vektoren ausgehen, ohne großen Genauigkeitsverlust entfernen (Abbildung 9.4).

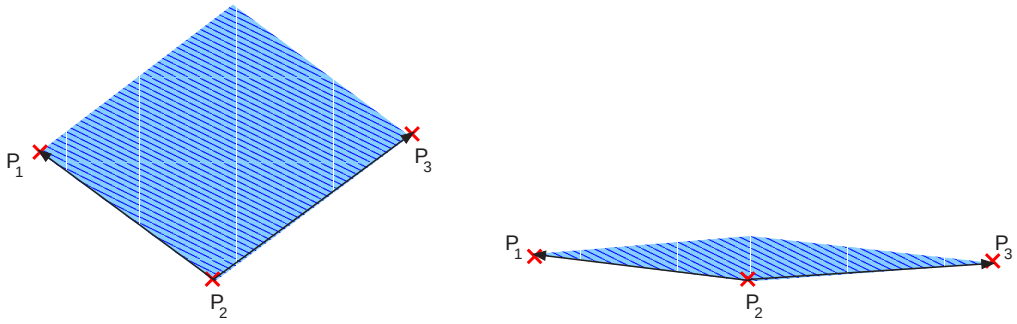


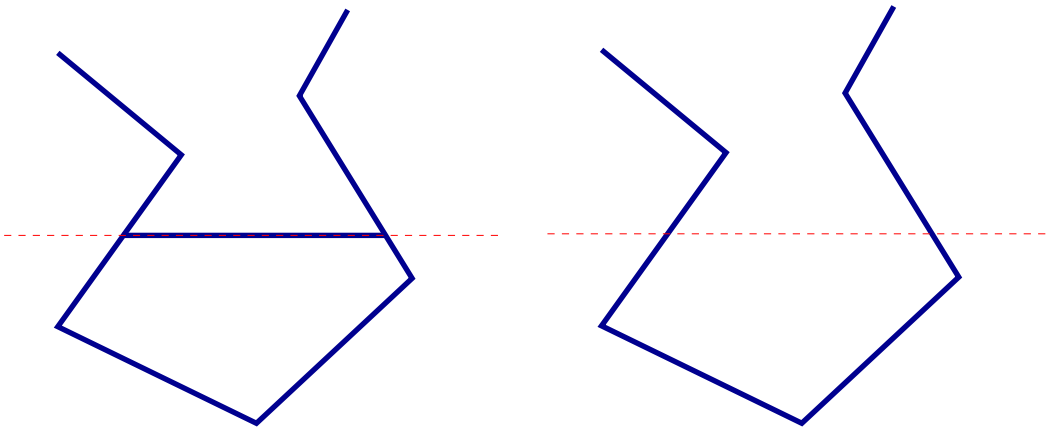
Abbildung 9.4: Generalisierung

Der Schwellwert, ab dem der Punkt gelöscht wird, ist durch den übergebenen `double`-Wert festgelegt. Falls sich an dieser Stelle herausstellt, dass der Punkt entfernt werden sollte, wird er daraufhin überprüft, ob er zu anderen `Graphic`-Objekten gehört. Gehört der Punkt zu einem `Point`-Objekt, darf er nicht gelöscht werden, damit die betroffenen Punktdaten korrekt auf der Polylinie liegen bleiben. Falls er zu anderen `Poly`-Objekten gehört, wird überprüft, ob in diesen Objekten Vorgänger und Nachfolger ebenfalls identisch sind. Diese Überprüfung erfolgt mit Hilfe von Inklusionsabfragen, um nicht auf die

Reihenfolge der Punkte achten zu müssen. Wenn diese Punkte identisch sind, darf der Punkt aus allen Objekten, zu denen er gehört, entfernt werden. So ist sichergestellt, dass aus allen aneinandergrenzenden Linien immer dieselben Punkte entfernt werden und sie somit danach immer noch aneinander liegen.

Die Klasse `PolyLine`

Bei Polylinien ist das Clipping weitaus komplizierter als bei Punktdaten. Es existieren Algorithmen zum Clippen von Linien und Polygonen. Einen sehr bekannten und effizienten Algorithmus haben *Sutherland und Hodgeman* entwickelt [Vorn2004]. Clippt man jedoch die Polylinien mit diesem Algorithmus, tritt beim Zeichnen ein für die Kartengrafiken ungewollter Effekt auf. Der Algorithmus fügt an den Rändern des Clippingrechtecks Punkte ein, damit er ein einziges zusammenhängendes Polygon erzeugt. Bei den geografischen Objekten auf den Kacheln der Karte müssen die Linien beim Zeichnen in der SVG-Grafik jedoch getrennt werden. Abbildung 9.5 verdeutlicht diesen Sachverhalt, die rote Linie kennzeichnet die Clippingkante.



(a) unerwünschte Linien (b) gewünschtes Ergebnis

Abbildung 9.5: Sutherland-Hodgeman für Liniendaten

Um das Objekt so zu zeichnen, wie in der Abbildung 9.5(b) zu sehen ist, muss die Liste der Punkte durchlaufen werden und festgestellt werden, ob man sich innerhalb oder außerhalb des Clippingrechtecks befindet. Bei jedem Wechsel von außen nach innen oder umgekehrt muss ein neuer Teil begonnen oder abgeschlossen werden.

Dabei müssen jedoch Sonderfälle abgehandelt werden. Beispielsweise können zwei Punkte der Linie außerhalb des Clippingrechteckes liegen, ein Teil der Verbindungslinie liegt jedoch innerhalb des Clippingrechteckes (Abbildung 9.6). In den Fällen, dass zwei aufeinander folgende Punkte außerhalb des Clippingrechtecks liegen, wird darauf geprüft, ob ihre Verbindungslinie mindestens eine Kante des Clippingrechteckes schneidet. Es wird dann der Schnittpunkt, beziehungsweise die Schnittpunkte berechnet und als Teilstück zum Zeichnen verwendet.

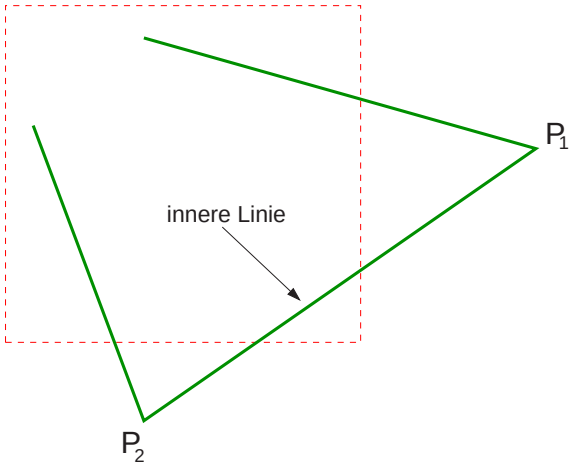
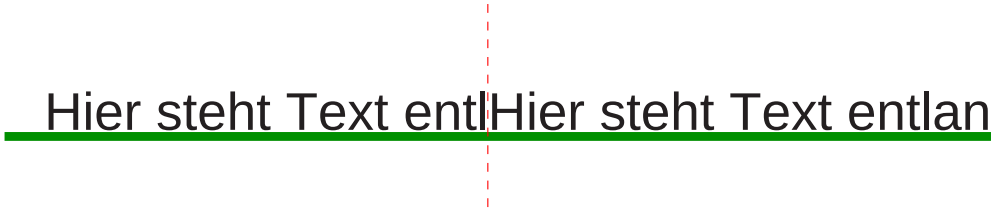
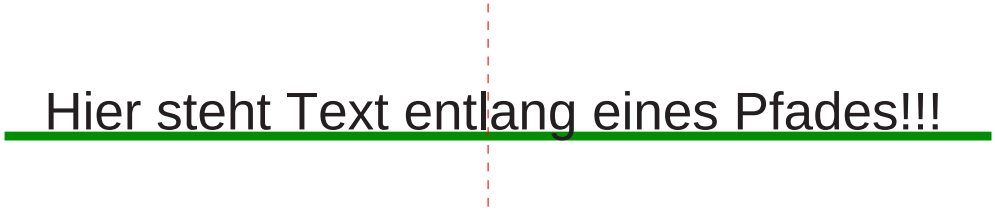


Abbildung 9.6: Sonderfall beim Linien-Clipping

Die Beschriftung von Polylinien erfolgt entlang des beschriebenen Pfades, was in SVG sehr einfach zu realisieren ist (Kapitel 3.1.4). An aneinander stoßenden Kachelgrenzen kann dies zu unschönen, nicht gewollten Beschriftungen führen, da die Beschriftung bei jedem neuem Pfad neu beginnt (Abbildung 9.7(a)). Das Programm löst dieses Problem, indem es für Beschriftungen von Polylinien für die einzelnen Abschnitte den `offset` für den Beginn des Textes berechnet. Es bestimmt die Länge des vollständigen, ungeclippten Pfades bis zum Punkt, an dem gerade die Beschriftung beginnt. Dieser Wert wird mit negativem Vorzeichen als `offset` angegeben, so dass die Beschriftung viel eher anfangen würde. Sie ist aber immer nur sichtbar, wenn sie auf dem Pfad verläuft, so dass sie erst zu Beginn des Pfades an der passenden Textstelle beginnt (Abbildung 9.7(b)).



(a) falsche Beschriftung



(b) korrekte Beschriftung mit `offset`-Angabe

Abbildung 9.7: Pfad-Beschriftung an Kachelgrenzen

Die Klasse `Polygon`

Beim Clippen von Polygonen stößt man auf die gleichen Probleme wie bei Polylinien. Das führt dazu, dass man die Füllung des Polygons und die Randlinien getrennt zeichnen muss. Das Zeichnen der Randlinien erfolgt nahezu analog zu dem der Polylinien, die Linien werden lediglich einen Punkt weiter als der Kachelrand gezeichnet, damit sie nicht von der Polygonfüllung einer Nachbarkachel verdeckt werden (Abbildung 9.8).

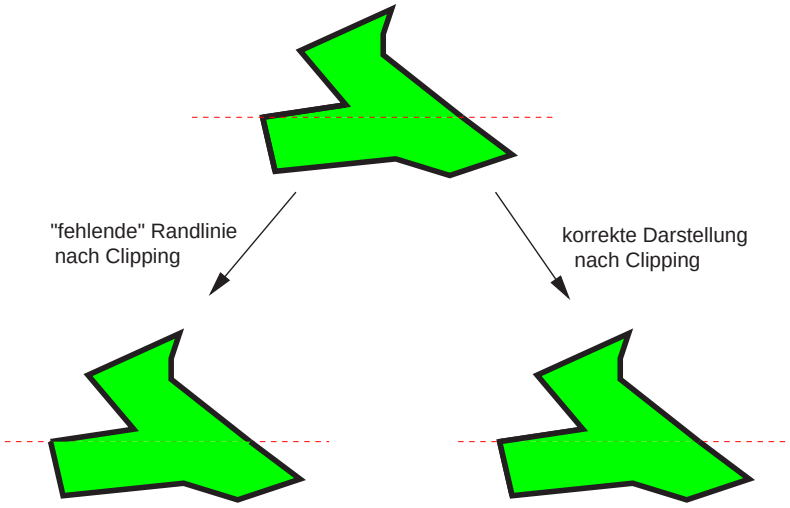


Abbildung 9.8: Durch Nachbarkachel verdeckte Randlinie

Für das Clippen der Füllfläche benötigt man die Randpunkte, könnte also dafür mit dem Algorithmus von Sutherland und Hodgeman clippen. In der Praxis führt das bei SVG-Grafiken jedoch zu Darstellungsfehlern vom ASV-Plugin. Ragt ein Polygon an zwei Stellen in eine Kachel hinein, so entsteht beim Clippen ein Polygon mit zum Teil aufeinander liegenden Randlinien. Die Randlinie des Polygons, das die Füllung darstellt ist auf `0px` gesetzt. Der ASV zeichnet nun an den Stellen, wo die Randlinien aufeinander liegen und somit keine Füllung nötig ist, eine sehr dünne Linie wie in Abbildung 9.9.

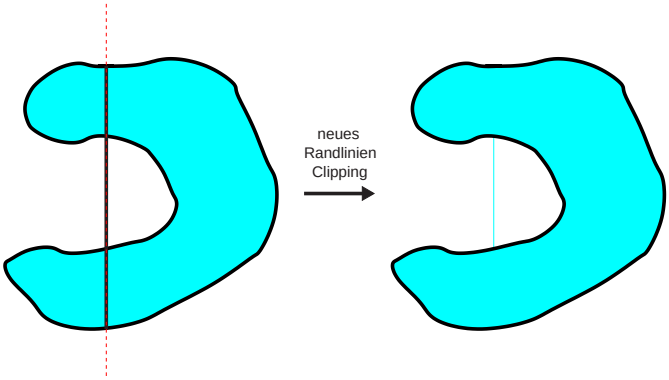


Abbildung 9.9: Darstellungsfehler bei geclipptem Polygon

Der Fall, in dem Polygone in der beschriebenen Art verlaufen, tritt bei geografischen Karten durchaus auf, so dass ein anderer Lösungsweg, also ein anderer Clippingalgorithmus, verwendet wird.

Die Idee dieses Algorithmus wird kurz vorgestellt, da nicht nur bereits existierende Algorithmen verwendet wurden, sondern eigene Überlegungen mit eingeflossen sind. Die Grundidee ist, dass ein Polygon in mehrere kleinere zerlegt wird und nicht mehr zusammenhängend ist. Dafür werden alle innerhalb der Clippingrechteckes liegenden Teilstücke der Randlinie des zu clippenden Polygons benötigt. Betrachtet man das Clippingrechteck ebenfalls als Polygon, werden auch die Teilstücke der Randlinie benötigt, die innerhalb des zu clippenden Polygons liegen [GrHo1998]. In Abbildung 9.10 sind die entsprechenden Linien in blau und grün gekennzeichnet.

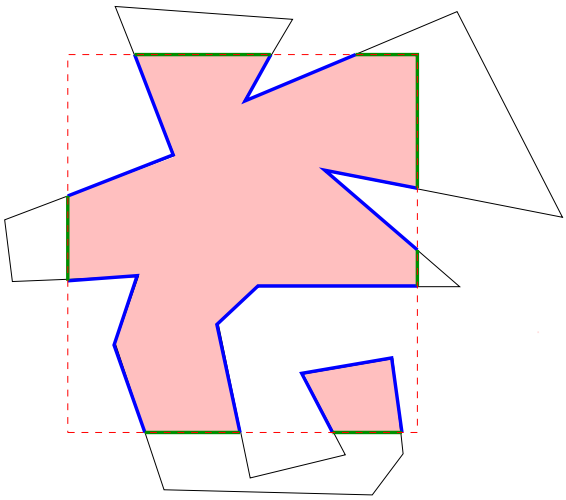


Abbildung 9.10: Beim Clippen zerfallendes Polygon

Die blau markierten Linien findet man analog zum bereits bei Polylinien beschriebenen Algorithmus. Dabei vermerkt man außerdem die vorhanden Randpunkte, also die Punkte, die auf den Kanten des Clippingrechteckes liegen. Um nun die grün markierten Linien zu erhalten, sortiert man die Randpunkte nach Kanten und anschließend je Kante nach x- oder y-Koordinaten. Die Kanten werden nun nacheinander nach folgendem Konzept abgearbeitet:

1. Man betrachtet zwei aufeinander folgende Randpunkte, berechnet den Mittelpunkt auf der Kante zwischen den beiden und prüft, ob er innerhalb des zu clippenden Polygons liegt. Dazu existieren bereits Algorithmen [Vorn2004].

2. Liegt der Punkt innen, so liegt die ganze Linie innen und wird zwischengespeichert. Liegt der Punkt außerhalb, so auch die Linie und sie kann verworfen werden. Bei diesen Betrachtungen muss man beachten, dass man die Eckpunkte des Clippingrechteckes ebenfalls zu den Randpunkten hinzunimmt.

3. Hat man alle innen liegenden Linien gefunden, muss man sie anhand ihrer Anfangs- und Endpunkte zu zusammenhängenden Linien und somit zu kleineren, einzelnen Polygonen zusammenfassen.

Die Beschriftung eines Polygons erfolgt, indem der Mittelwert von x- und y-Koordinate aller Punkte, also der Schwerpunkt des Polygons, berechnet wird und anhand dessen der Text platziert wird.

Die Klasse `PointList`

Die Klasse `PointList` hat keine besonderen Eigenschaften, sie dient hauptsächlich der einfacheren Verwaltung der Grafikobjekte.

Die Klasse `PolyList`

Eine `PolyList` hat ein zusätzliches Attribut `treatasone` und eine Methode zum Setzen dieses Attributes. In den Shapefiles kann es sein, dass einzelne Polygone oder Polylinien gemeinsam ein geografisches Objekt beschreiben. In diesen Fällen werden sie in eine Liste eingefügt, die dann nach außen hin wie ein einzelnes `Poly`-Objekt reagiert. Sie leitet Aufrufe nicht weiter, sondern holt sich beispielsweise beim Erzeugen des SVG-Fragmentes alle Teilpfade der enthaltenen Objekte und fügt diese zu einem einzigen Pfad zusammen.

Die Klasse `PolylineList`

Die `PolylineList` kann aufgefordert werden, die enthaltenen Polylinien anhand eines bestimmten Wertes in der `recordinfo`-Liste, also anhand eines Attributes aus dem Database-File, zusammenzufassen. Zum Beispiel sind Strassen zum Teil in vielen kleinen Teilstücken ungeordnet in einem Shapefile enthalten. Vor allem für die Beschriftung ist es erwünscht, dass die Teilstücke nicht einzeln sondern als zusammenhängender Pfad behandelt werden können. Die Methode `aggregate(int)` sortiert zunächst alle Objekte anhand des durch den Index bestimmten Attributes in Gruppen. In einer Gruppe sind dann also die Objekte mit gleichem Attributwert. Diese werden anschließend gruppenweise auf gleiche Anfangs- und Endpunkte überprüft und soweit wie möglich zusammengefasst. Vorteilhaft daran ist, dass weniger Objekte zum Zeichnen entstehen und vor Allem die Beschriftungen weitaus besser und nicht gestückelt aussehen (Kapitel 10).

Die Klasse `PolygonList`

Die `PolygonList` dient hauptsächlich zum Verwalten der Grafikobjekte, sie verhält sich allerdings nach außen wie ein Polygon-Objekt, falls sie durch Aufruf der Methode `treatasone()` entsprechend gekennzeichnet worden ist. Sowohl beim Erzeugen der SVG-Struktur des Objektes als auch der Beschriftung wird der Aufruf nicht an die enthaltenen Objekte weitergeleitet. Stattdessen werden die Pfade der enthaltenen Polygone mit einem `M` (moveto) aneinandergehängt. Beim Erzeugen der Beschriftung werden die Mittelwerte der Koordinaten aller Punkte der enthaltenen Polygone berechnet und anhand dieser Werte der Text platziert.

9.3 Daten auslesen

Bevor das Kartenmaterial als SVG-Grafiken erstellt werden kann, müssen die Java-Grafikobjekte erzeugt werden. Dazu müssen die Geodaten aus den Shapefiles und die Attributdaten aus den Database-Files ausgelesen werden. Hierzu werden bereits vorhandene

Klassen von [OpMa2005a] verwendet. Die Klassen sind Open Source und unterliegen dem OpenMap Software License Agreement [OpMa2005c]. Sie sind kostenlos zu erhalten und dürfen verwendet und verändert werden. Für diese Arbeit wurden die Klassen in Java 1.5 umgeschrieben, um Typsicherheit beim Verwalten von Listen zu garantieren. Außerdem wurden die Klassen zum Auslesen der Daten erweitert und auf die Bedürfnisse dieser Arbeit angepasst. Die Klassen zur Repräsentierung von geografischen Objekten wurden vollständig ausgetauscht, stattdessen werden die Objekte aus dem vorangegangenen Kapitel 9.1 verwendet.

Die Klasse `ShpInputStream` liest Shapefiles aus und erzeugt mit den Daten entsprechende Grafikobjekte. Aus einem Shapefile wird somit eine `GraphicList` erzeugt. Je nach Typ des gelesenen Shapefiles unterscheidet der `ShpInputSteam`, ob Punktgeometrien oder Polygeometrien gelesen werden. Die notwendigen Informationen über den Shapetyp befinden sich im Header des Shapefiles, ebenso wie Angaben über minimale und maximale Koordinaten. Beim Lesen eines Records wird dann jeweils entweder ein `Point`-, `Polyline`- oder `Polygon`-Objekt erzeugt. Dazu werden zunächst `MyPoint`-Objekte erzeugt, bei denen vermerkt wird, dass sie zum gerade entstehenden `Graphic`-Objekt gehören. Dies ist für die eventuell später folgende Generalisierung (Kapitel 9.2, 9.6) notwendig. Auf den Punkt kann zusätzlich eine geografische Projektion angewandt werden, falls ein `Converter`-Objekt im Konstruktor des `ShpInputStream` übergeben wurde (Kapitel 9.3.1). Der Punkt wird anschließend entweder in eine Liste eingefügt, mit der bei fertig gelesenem Record ein `Poly`-Objekt oder eine Instanz der Klasse `Point` erzeugt wird. Falls mehrere `Poly`-Objekte zusammengehören, wird eine `PolyList` erstellt und an ihr `treatasone()` aufgerufen.

Ist ein Database-File vorhanden, werden zu jedem Record die vorhandenen Attributdaten ausgelesen und innerhalb des entsprechenden Objekts im Feld `recordinfo`, einer Referenz auf eine Objekt-Liste, gespeichert. Das Auslesen des Database-Files übernimmt ein `DbfInputStream`, der die Informationen zu jedem Record eines Shapefiles als Liste von `String`- oder `Double`-Objekten liefern kann.

9.3.1 Daten transformieren

Auf geografische Daten werden häufig Projektionen angewandt, entweder bereits vor dem Speichern oder erst vor der Darstellung. Je nachdem, ob die Daten zum Beispiel in Längen- und Breitengraden oder Gauß-Krüger-Koordinaten angegeben oder rotiert sind, müssen sie gegebenenfalls transformiert werden, um eine visuell ansprechende Grafik zu erhalten. Wie bereits beschrieben werden alle gewünschten Transformationen mit Hilfe eines `Converter`-Objektes durchgeführt. Es existiert das Interface `Converter`, das lediglich eine Methode enthält, die ein `MyPoint`-Objekt transformieren kann:

```
public interface Converter {
    public void convert(MyPoint p);
}
```

In der vorliegenden Arbeit sind bereits verschiedene Klassen, die das `Converter`-Interface implementieren, vorhanden. Als Beispiel einer gängigen Projektion dient die

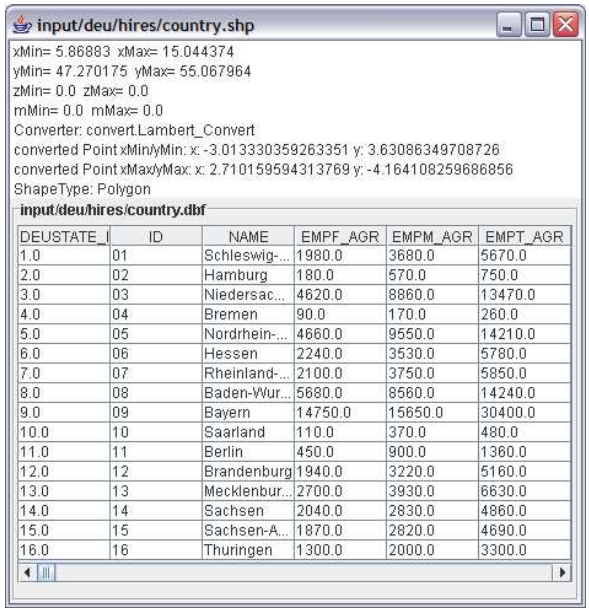
Lambertsche Azimuthal Projektion [BuSn1995]. Möchte der Benutzer des Programms eine eigene Projektion verwenden, muss er eine Klasse schreiben, die das `Converter`-Interface implementiert. In den meisten Fällen ist eine Umkehrung des Vorzeichens der y-Koordinate nötig, da das SVG-Koordinaten-System mit der y-Achse nach unten verläuft. Geodaten sind aber häufig in geografischen Koordinaten abgespeichert, wobei die y-Achse nach oben zeigt.

Bei Ausführung des Programms ist bekannt, welche Klasse zum Transformieren der Daten verwendet werden soll (Kapitel 9.6). Mit dem gegebenen Klassennamen wird über *Reflection* ein Objekt dieser Klasse erzeugt und dem lesenden `ShpInputStream` mitgegeben.

9.4 Programme zum Datensichten

Bevor man eine Konfigurationsdatei für das Programm schreiben kann, muss man wissen, was für Daten überhaupt vorliegen. Eine notwendige Angabe ist zum Beispiel der Shapetyp. Genauso wichtig ist es zu wissen, welche Attributdaten vorliegen und in welcher Spalte des Database-Files welches Attribut steht, wenn man Beschriftungen wünscht. Das Java-Programm enthält im Package `util` die Klasse `ShowInfo`, die einerseits die Header-Informationen eines Shapefiles ausliest und anzeigt, andererseits den Inhalt des Database-Files als Tabelle darstellt. Abbildung 9.11 zeigt dies beispielhaft mit einer Datei, die Informationen über Deutschland enthält.

Beim Aufruf des Programms muss als Kommandozeilenparameter der Pfad zur Datei mitgegeben werden. Wird als zweiter Parameter ein `Converter`-Klassenname angegeben, werden auch die transformierten Minimum- und Maximum-Werte mit angezeigt.



DEUSTATE_I	ID	NAME	EMPF_AGR	EMPM_AGR	EMPT_AGR
1.0	01	Schleswig-...	1980.0	3680.0	5670.0
2.0	02	Hamburg	180.0	570.0	750.0
3.0	03	Niedersac...	4620.0	8860.0	13470.0
4.0	04	Bremen	90.0	170.0	260.0
5.0	05	Nordrhein-...	4660.0	9550.0	14210.0
6.0	06	Hessen	2240.0	3530.0	5780.0
7.0	07	Rheinland-...	2100.0	3750.0	5850.0
8.0	08	Baden-Wur...	5680.0	8560.0	14240.0
9.0	09	Bayern	14750.0	15650.0	30400.0
10.0	10	Saarland	110.0	370.0	480.0
11.0	11	Berlin	450.0	900.0	1360.0
12.0	12	Brandenburg	1940.0	3220.0	5160.0
13.0	13	Mecklenbur...	2700.0	3930.0	6630.0
14.0	14	Sachsen	2040.0	2830.0	4860.0
15.0	15	Sachsen-A...	1870.0	2820.0	4690.0
16.0	16	Thuringen	1300.0	2000.0	3300.0

Abbildung 9.11: `ShowInfo`-Datendisplay

Mit Hilfe der Tabelle lässt sich herausfinden, in welcher Spalte bestimmte Attributdaten stehen. Dieses Programm dient somit als Unterstützung für den Benutzer, der eine Konfigurationsdatei schreiben möchte.

Eine weitere Möglichkeit, sich die Daten der Shapefiles mit einigen Informationen anzeigen zu lassen, bietet das Programm ArcExplorer der Firms ESRI, das man kostenlos unter [Esri2005c] herunterladen kann. Mit Hilfe dieses Programms kann man sich die Daten bereits visualisieren lassen und gewinnt bereits im Vorfeld einen optischen Eindruck.

9.5 Die Konfigurationsdatei

Die Konfigurationsdatei ist eine XML-Datei, da XML relativ einfach zu erstellen und vor allem zu Verarbeiten ist (Kapitel 2). Die Beschreibung, welche Elemente und welche Werte für Attribute erlaubt sind, ist in einer Schemadatei (Kapitel 2.2.2 und Anhang A) festgelegt. Da einige Werte besonders eingeschränkt werden müssen und eine starke Kontrolle über die Benutzerangaben notwendig ist, um eine einwandfreie Verarbeitung zu gewährleisten, wurde keine DTD sondern ein Schema erstellt. Für einige Benutzerangaben sind dort auch Default-Werte hinterlegt, so dass dem Benutzer das Erstellen einer Konfigurationsdatei möglichst leicht gemacht wird.

Das Wurzelelement einer Konfigurationsdatei ist das `map`-Tag. Dieses muss jeweils genau ein `configuration`-Tag und ein `zoom`-Tag enthalten. Außerdem müssen ein bis viele `layer`-Tags darin enthalten sein. Diese Tags werden nun im Einzelnen näher erläutert.

Im `configuration`-Tag können einige grundlegende Angaben zur Applikation gemacht werden. Zwingend erforderlich ist die Angabe eines Ordners, in den die vom Java-Programm erzeugten Dateien geschrieben werden sollen. Alle anderen Attribute sind optional. Es lassen sich verschiedene Farben für die Applikation auswählen, Hintergrundfarbe der Appliaktion und des Kartendisplays, Linienfarbe, Symbolfarbe und Textfarbe. Außerdem kann über das Attribut `converterclass` ein Klassenname angegeben werden, der festlegt, welche Klasse für Projektionen oder Skalierungen der Geodaten verwendet werden soll (Kapitel 9.3.1). Gibt man keinen Klassennamen an, wird ein Default-Converter verwendet. Die optionalen Attribute `simplify` und `text-simplify` können angegeben werden, wenn die Geodaten generalisiert werden sollen. In diesem Fall werden nach bestimmten Regeln Punkte aus Polygonen und Polylinie entfernt (Kapitel 9.2). Ebenfalls optional ist die Angabe über x- und y-Werte der Viewbox sowie deren Breite. Diese Angaben müssen gemacht werden, wenn nur ein Ausschnitt der vorhandenen Daten angezeigt werden soll. Innerhalb dieses Tags kann über das `script`-Tag ebenfalls ausgewählt werden, welche Art des Datenladens (Kapitel 8.4.2) verwendet werden soll. Zur Veranschaulichung hier ein Beispiel für ein `configuration`-Tag:

```
<configuration displaybgcolor="grey" vbX="3427000.29"
 vbY="-5802737.21" vbWidth="17003.70" outputfolder="lkall"
 script="v2" converterclass="util.Convert_OS"
 simplify="100" text-simplify="100"/>
```

Im `zoom`-Tag wird festgelegt, wie viele Zoomstufen es gibt und mit welchen Faktoren das Ein- und Auszoomen und die Navigation stattfinden. Innerhalb des `zoom`-Tags werden ein bis viele `zoomstep`-Tags angegeben. Ihre Anzahl entspricht der Anzahl der Hauptzoomstufen in der Applikation. Das Attribut `steps` gibt an, nach wievielen „Zoom-in"-Schritten die Zoomstufe gewechselt wird, `stepfactor` gibt an, um welchen Anteil der Viewboxbreite sich beim Navigieren nach oben, unten, links oder rechts bewegt wird. Das Attribut `zoomfactor` ist der Faktor, mit dem die Breite der Viewbox bei einem Einzoomvorgang multipliziert wird.

Ein `layer`-Tag repräsentiert einen Layer in der Applikation, also eine Informationsebene innerhalb der Karten, auf der bestimmte Daten liegen und die ein- und ausgeblendet werden kann. Für einen Layer kann man angeben, ob er ein- und ausgeblendbar sein soll oder dauerhaft angezeigt wird, indem man das Attribut `changeable` auf `true` oder `false` setzt. Ebenso kann man durch entsprechendes Setzen des Attributes `visiblility` bestimmen, ob der Layer beim ersten Laden sichtbar sein soll oder nicht. Dem Layer muss eine ID gegeben werden, die innerhalb der Applikation verwendet wird. Diese ID wird ebenfalls als Beschriftung für den Button zum Ein- und Ausblenden des Layers verwendet, sofern nicht das Attribut `label` explizit angegeben wird. Innerhalb eines `layer`-Tags können verschiedene Geodaten angegeben werden. Dazu besitzt der Layer untergeordnete `zoomstep`-Tags, deren Anzahl pro Layer mit der Anzahl der `zoomstep`-Tags im `zoom`-Element übereinstimmen muss. Pro Zoomstufe in einem Layer kann man festlegen, ob die angegebenen Daten dazu geladen werden sollen oder mit den zuvor geladenen Daten ausgetauscht werden sollen. Dazu gibt man im Attribut `action` entweder `add` oder `replace` an (Kapitel 8.4.2). Wird nichts angegeben, werden die Daten hinzugefügt. Innerhalb eines `zoomstep`- Elementes werden die Daten in Form von `selection`-Tags angegeben. Eine solche Auswahl hat eine Quelldatei (Shapefile), deren Pfad ohne Dateiendung im Attribut `sourcefile` angegeben wird. Außerdem ist es zwingend notwendig, ein `layout`-Tag anzugeben. Dieses Tag ist je nach Art der ausgewählten Daten über das Attribut `xsi:type` zu spezifizieren. Es gibt hier die Werte `Point`, `Polygon`, `Polyline`, `Text` und `Textpfad`. Mit dem Layout werden Füllfarben, Linienfarben, Liniendicken und Schriftgrößen festgelegt. Für Text kann die Position der Textverankerung sowie der Style festgelegt werden. Ein zwingendes Attribut für `Text` und `TextPath`-Layouts ist `index`. Der Index gibt die Spalte der zu den Geodaten gehörigen `dbf`-Datei an, in der die Werte zur Beschriftung enthalten sind. In Kapitel 9.4 ist beschrieben, wie man die Indizes herausfinden kann, falls sie nicht bekannt sind. Ist das Layout vom Typ `Point`, kann auch der Name einer Datei aus dem Verzeichnis `symbols` angegeben werden. In dieser Datei muss das Symbol definiert sein, dass dann in der Applikation verwendet wird.

Je nach Typ gibt es weitere Attribute wie beispielsweise `aggregate` bei Polylinien. Diesem Attribut wird wie beim Beschriften ein Indexwert einer Spalte des Database-Files mitgegeben. Anhand dieses Wertes werden beim Verarbeiten die Polylinien zusammengefasst. Anschließend werden sie sortiert und passend aneinander gereiht, um schönere Beschriftungen und eine geringere Anzahl an Objekten zu erreichen (Kapitel 9.2). Um nicht zu unübersichtlich zu werden, sei zu weiteren Layout-Attributen auf die Schema-Datei im Anhang A, auf das noch folgendene Beispiel und die ausführlichen Beispiele in Kapitel 10 verwiesen.

Im `selection`-Tag kann man nicht nur das Layout der angegebenen Shapes bestimmen. Es besteht zusätzlich die Möglichkeit der Selektion einzelner geografischer Objekte anhand von Attributen aus der `dbf`-Datei. Hierzu muss man über ein oder mehrere `param`-Tags die Auswahl eingrenzen. Ein `param`-Tag besitzt das Attribut `include`, welches die beiden Werte `include` und `exclude` annehmen kann. In beiden Fällen muss man den Index der Spalte im Database-File angeben, in der das Attribut steht, nach dem selektiert werden soll. Im Attribut `valuelist` folgt eine Liste mit den gewünschten Werten. Die Werte werden durch Semikoli getrennt aneinander gereiht, alternativ kann explizit ein anderes Trenn-Zeichen im `delimiter`-Attribut angegeben werden. Je nachdem, ob `include` oder `exclude` angegeben wurde, werden die Objekte mit den definierten Werten in der Liste der Grafikobjekte behalten oder gelöscht. Die Einschränkungen über die `param`-Tags lassen sich kombinieren. Um den Sachverhalt zu verdeutlichen sei folgendes Beispiel gegeben:

```
<selection sourcefile="input/os/strassen-joined">
   <param index="2" include ="include"
    valuelist="1.0;2.0;3;0"/>
   <param index="5" include ="exclude"
    valuelist="kein Name vorhanden;nicht attributiert"/>
   <layout xsi:type="TextPath" font="Arial"
    font-fill="#000000" offset="30" repeat="20"
    spacing="100" font-size="11.5" index="5" />
</selection>
```

In dieser Auswahl sollen nur bestimmte Straßen beschriftet werden. Die Straßen sind je nach Größe in Kategorien eingeteilt, diese Angabe ist in Spalte 2 in der `dbf`-Datei vermerkt. Das erste `param`-Tag besagt, dass nur Straßen der Kategorie 1 bis 3 beschriftet werden sollen. Die Beschriftung erfolgt mit den Angaben aus Spalte 5, den Straßennamen. Das zweite `param`-Tag bewirkt, dass Strassen nicht beschriftet werden, die nicht attributiert sind oder bei denen kein Name vorhanden ist. Diese Angaben befinden sich in Spalte 5 der `dbf`-Datei. In diesem Beispiel sieht man für das Layout weitere Angaben für die Beschriftung. Das Attribut `offset` gibt an, nach wie vielen Einheiten mit der Beschriftung begonnen wird, `repeat` besagt, wie oft die Beschriftung entlang des Pfades wiederholt wird und `spacing` gibt die Größe des Freiraumes zwischen den einzelnen Beschriftungen an.

In der Konfigurationsdatei muss ebenfalls angegeben werden, welche Daten auf der Übersichtskarte erscheinen sollen. Diese Angaben erfolgen innerhalb eines `overview`-Tags mit Hilfe der bereits beschriebenen `selection`-Tags. Für die Übersichtskarte kann eine eigene Generalisierung erfolgen, indem das Attribut `simplify` angegeben wird.

9.6 Verarbeitung der Konfigurationsdatei

Zu Beginn von Kapitel 9 wurde ein kurzer Überblick über den Datenfluss des Java-Programms gegeben. Nachdem in den vorangegangenen Kapiteln behandelt wurde, was

zur Grafikerzeugung notwendig ist und wie eine Konfigurationsdatei aufgebaut ist, kann jetzt auf die Verarbeitung der Konfigurationsdatei eingegangen werden.

Da die Konfigurationsdatei in XML geschrieben ist und einem Schema genügt, wird zum Verarbeiten ein gegen ein Schema validierender Parser verwendet. In Kapitel 2.3 ist beschrieben, wie solch ein Parser in Java erzeugt und verwendet wird. Die Klasse `ConfigDOM` verwendet einen solchen Parser zum Einlesen und Validieren. Wenn Fehler, entweder syntaktische oder semantische, in der Konfigurationsdatei enthalten sind, werden diese direkt vom Parser an den angemeldeten `ErrorHandler` gemeldet. Von diesem wird eine entsprechende Fehlermeldung erzeugt. In dieser Klasse werden nach erfolgreichem Einlesen noch einige zusätzliche Abfragen am erzeugten DOM-Tree vorgenommen, um falsche Angaben des Benutzers, die nicht durch Validierung der Konfigurationsdatei überprüft werden können, zu erkennen. In diesem Fall wird eine `ConfigException` geworfen. Ist auch diese Überprüfung erfolgreich verlaufen, befindet sich ein vollständiges Document Object Model der Konfigurationsdatei im Speicher. Sämtliche Abfragen am DOM-Tree der Konfiguratonsdatei werden über die Klasse `ConfigDOM` vorgenommen. Sie implementiert einige Methoden zum direkten Zugriff auf die `layer`-Tags oder die `selection`-Tags und andere hilfreiche Zugriffe.

Ein Objekt der Klasse `ConfigDOM` wird im Hauptprogramm, der Klasse `Creator` erzeugt. Dazu wird der Pfad zur Konfigurationsdatei als Kommandozeilenparameter übergeben. Außerdem wird eine Instanz der Klasse `MySVGDocument` erzeugt, diese repräsentiert das in Kapitel 8 beschriebene SVG-Template. Im Template werden die vom Benutzer angegebenen Farben für Hintergrund, Text und Zoom-Symbole gesetzt. Zusätzlich wird der Name der Template-Datei aus der `output`-Angabe des Benutzers in der Konfigurationsdatei ermittelt, so dass das `MySVGDocument` seinen Inhalt in eine Datei mit entsprechendem Namen schreiben kann.

Im Laufe der weiteren Verarbeitung werden alle `layer`-Tags nach ihren `id`- und `label`-Attributen befragt und ECMAScript erzeugt, das später beim Laden der Applikation im Browser in das Template die passenden Layer-Ebenen und Buttons zum Ein- und Ausblenden einfügt. Zusätzlich wird in ein PHP-Skript ein Array mit allen Layernamen geschrieben, dass zum Ermitteln der zu ladenden und zu löschenden Daten dient (8.4.2).

Die weitere Verarbeitung verläuft abhängig von der Angabe, ob eine Generalisierung (9.2) erfolgen soll oder nicht. Zunächst wird der Fall beschrieben, dass eine Generalisierung stattfinden soll. Bevor die SVG-Grafiken und somit das Kartenmaterial erzeugt werden kann, müssen alle verwendeten geografischen Objekte einmal gemeinsam im Speicher vorliegen. Um die Objektanzahl zu reduzieren, wird eine `HashMap` zum Speichern der auftretenden Punkte verwendet, die den `ShpInputStream`s mitgegeben wird. Treten nun Punkte mehrfach auf, wird das schon vorhandene Objekt aus der `HashMap` verwendet. Diese `MyPoint`-Objekte erhalten entsprechende Informationen, zu welchen geografischen Objekten sie gehören. Anschließend werden alle vorhandenen `Poly`-Objekte durchlaufen und an ihnen die Methode `simplyfy(double)` mit dem vom Benutzer angegebenen Wert aufgerufen und so generalisiert. Hat der Benutzer keine Angaben über minimale und maximale Viewbox-Koordinaten gemacht, werden diese Werte durch das Suchen der minimalen und maximalem x- und y-Koordinaten der Punkte in der

`HashMap` ermittelt. Sie werden für das Berechnen der Kacheln und das Setzen der Viewbox in der Applikation benötigt. Aus den im Speicher vorliegenden `GraphicList`-Objekten werden anschließend alle SVG-Grafiken (Kacheln) erzeugt.

Falls keine Generalisierung erfolgen soll, müssen alle Shapefiles einmal eingelesen werden, falls der Benutzer keine Angaben über die Viewbox gemacht hat, um die minimalen und maximalen Koordinatenwerte zu ermitteln. Es ist in diesem Fall aber nicht notwendig, alle eingelesenen Daten im Speicher zu behalten. Das hat den Vorteil, dass auch große Datenmengen verarbeitet werden können (9.6.1). Das Erzeugen der jeweiligen `GraphicList`-Objekte zum Erstellen der Karten erfolgt dann direkt vor dem Erzeugen der Kacheln.

Das grundlegende Vorgehen der Kachelerzeugung aus den `GraphicList`-Objekten ist mit oder ohne Generalisierung gleich. Für jeden Layer und jeden `zoomstep` werden die Grafikobjekte für eine `selection` hergenommen und dazu aufgefordert, sich in die DOM-Trees für die jeweiligen Kacheln einzufügen. Die Werte zum dafür notwendigen Clippen werden im Programm aus der Viewbox-Breite (aus minimalen und maximalen Koordinaten) und den Angaben über Zoomfaktor und Anzahl der Zoomschritte aus dem `zoom`-Tag und dessen `zoomstep`-Tags errechnet. Zum Schreiben der Datei einer Kachel werden Instanzen der Klasse `DOMFileWriter` verwendet. Bevor diese die Datei schreibt, wird geprüft, ob überhaupt Objekte im DOM der Kachel enthalten sind. Nur wenn das der Fall ist, wird die Datei geschrieben, um Speicherplatz zu sparen. Die ID der Kachel wird außerdem innerhalb des Hauptprogramms in einer Liste gespeichert, damit in das später erzeugte PHP-Skript die existierenden IDs eingetragen werden können. Während des Erzeugens der Kacheln werden auch die `action`-Dateien (Kapitel 8.4.1) für die jeweiligen Layer und Zoomstufen mit Hilfe der Klasse `ActionFileWriter` erzeugt und geschrieben.

Sind alle Grafiken erzeugt, wird das ECMAScript mit den Funktionen zum Setzen der Viewbox, Erzeugen der Layer und der Buttons geschrieben (Kapitel 4.1, 8.2). Auch Angaben zur minimalen Viewboxbreite und zu den Zoomstufen werden im Skript vermerkt. Falls in der Applikation Punktdaten mit Symbolen dargestellt werden sollen, werden diese ebenfalls durch dieses Skript in das `defs`-Element des Templates eingefügt. Beim Abarbeiten der Layer wurden im Hauptprogramm alle auftretenden Symbol-IDs gespeichert. Die generierte ECMA-Skript-Funktion verwendet zum Schreiben jedes Symbols eine ECMAScript-Funktion, die als Parameter die ID und den String, der das Symbol definiert erhält. Dieser String wird im Hauptprogramm durch Auslesen der durch die ID definierten Datei ermittelt. Das vom Programm generierte ECMAScript wird in das Ausgabe-Verzeichnis geschrieben. Dorthin wird auch das gesamte Skript zum Erzeugen der Zoomelemente und für die Interaktion kopiert.

Ein Teil vom nötigen PHP-Skript wird ebenfalls vom Java-Programm geschrieben. Es handelt sich um Angaben zu den existierenden Kachel-IDs und Layern, sowie Werte der Viewboxbreite beim Wechsel der Zoomstufen. Je nach Benutzerangabe (8.4.2) wird die passende Version des Skriptes zum Datenübermitteln in das Ausgabe-Verzeichnis kopiert.

Die Klasse `Creator` hat nun alle Benutzerangaben verarbeitet und die notwendigen SVG-Grafiken und Skripte erzeugt. Das entstandene Datei-Material muss dann auf einen

PHP5-fähigen Server gelegt werden. Die SVG-Datei, die sich unmittelbar im erzeugten Verzeichnis neben den Verzeichnissen `script` und `svgfiles` befindet, ist die Template-Datei, die für das Betrachten der Applikation im Browser geladen werden muss.

9.6.1 Encoding und Speicherplatz

Sowohl in der Konfigurationsdatei als auch in den Database-Files können Umlaute vorkommen. Diese Tatsache muss sowohl beim Erstellen der Web Mapping Applikation durch das Java-Programm, beim Compilieren sowie beim Ausführen des Programms beachtet werden.

Treten in der Konfigurationsdatei in den Beschriftungen für die Layer-Buttons Umlaute auf, so muss dies für das generierte ECMAScript beachtet werden. Sobald in ECMAScript Umlaute auftreten, kommt es beim Ausführen zu einem Laufzeitfehler. Will man trotzdem zulassen, dass die Label Umlaute enthalten, muss man in der erzeugenden ECMAScript-Funktion die Umlaute kodieren und bei Aufruf mit Hilfe der Funktion `unescape(string)` wieder dekodieren [Flan2002].

Da innerhalb des Java-Programms die Umlaute aus den Label-Strings durch die entsprechende ECMA-Script-Codierung ersetzt werden müssen, stehen auch im Quellcode Umlaute. Je nach Editor ist das Encoding unterschiedlich eingestellt. Daher sollte man beim Compilieren des Java-Programms sicherstellen, dass ein entsprechendes Encoding verwendet wird, welches die Umlaute korrekt interpretiert. Beispielsweise würde das Hauptprogramm der vorliegenden Arbeit durch den Aufruf

```
javac -encoding ISO-8859-1 export.Creator.java
```

mit dem Encoding für westeuropäische Sprachen kompiliert werden. Eine Übersicht über verschiedene Encodings findet man in [HaMe2005].

Die System-Property `file.encoding` spielt beim Ausführen des Programms eine wichtige Rolle, da zur Laufzeit des Java-Programms das Auslesen der Database-Files erfolgt und die dort enthaltenen Umlaute dann richtig interpretiert werden müssen. Andernfalls werden die Umlaute in den Beschriftungen der Karten nicht korrekt dargestellt. Um beim Ausführen das korrekte Encoding zum Lesen der Dateien zu verwenden, erfolgt der Aufruf des Programms mit explizitem Setzen der entsprechenden Property:

```
java -Dfile.encoding=8859_1 export.Creator konfig.xml
```

Je nachdem, wie viele Daten in die Karten aufgenommen werden sollen, wird unterschiedlich viel Arbeitsspeicher vom Java-Programm benötigt. Der ausführende Rechner muss dann entsprechend viel Arbeitsspeicher und Prozessorleistung besitzen. Falls beim Ausführen des Programms die Fehlermeldung

```
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

erscheint, muss der zur Verfügung stehende Speicherplatz über Kommandozeilenparameter erweitert werden. Im folgenden Aufruf wird der Initialwert auf 512 Megabyte gesetzt, der maximale Wert auf 1024 Megabyte:

```
java -Dfile.encoding=8859_1 -Xms512m -Xmx1024m
      export.Creator konfig.xml
```

Der maximale aktuelle Speicherplatzbedarf des Programms ist größer, wenn eine Generalisierung erfolgen soll. Wie in Kapitel 9.6 erwähnt, müssen dann alle Grafikobjekte gemeinsam im Speicher vorliegen. Ohne Generalisierung werden die Grafikobjekte nach und nach abgearbeitet, so ist zum aktuellen Zeitpunkt weniger Speicher belegt.

10 Anwendung

Bevor in diesem Kapitel auf die Anwendung des Java-Programms eingegangen, also gezeigt wird, wie man eine Web Mapping Applikation mit dem vorliegenden Programm erstellt, wird kurz darauf eingegangen, woher man Geodaten im Shapefileformat erhalten kann. Die angegebenen Beispiele erheben keinen Anspruch auf Vollständigkeit. Sie sind nicht zu umfangreich gewählt, um auf wesentliche Punkte eingehen zu können. Einen umfangreichen Überblick über die Konfigurationsmöglichkeiten findet man in Kapitel 9.5 und in der Schemadatei (Anhang A). Die vollständigen Konfigurationsdateien zu zwei größeren Web Mapping Applikationen sind auf der beiliegenden CD-Rom im Verzeichnis `programm/schema` zu finden, die Applikationen selbst sind erreichbar unter:

`http://www.inf.uos.de/prakt/pers/dipl/dlangfel`

Alle SVG-Dateien und Skripte zu beiden Applikationen sind außerdem auf der beigefügten CD-Rom im Verzeichnis `app` zu finden. Um die Applikationen betrachten zu können, müssen sie jedoch auf einen Server mit PHP5-Unterstützung gelegt werden.

10.1 Datenbeschaffung

Die Beschaffung von Geodaten ist gerade in Deutschland nicht immer einfach. Daten sind zum Teil sehr teuer oder unterliegen dem Datenschutzgesetz. Die deutsche Gesellschaft für Kartografie e. V. [DeGK2005] und Cartogis [Cart2005] bieten digitale Landkarten in unterschiedlichsten Formaten zum Kauf an. Auf der Website ESRI World Basemap Data [Esri2005b] kann man sich von vielen Ländern der Welt Daten in unterschiedlicher Detailliertheit im Shapefileformat kostenlos herunterladen. In den Beispielen dieser Arbeit werden Daten von Deutschland verwendet, die vom Institut für Umweltsystemforschung der Universität Osnabrück zur Verfügung gestellt wurden. Weiterhin werden die von der Intevation GmBH im Projekt Frida [Frid2005] erstellten Shapefiles mit Daten der Stadt Osnabrück verwendet. Frida ist eines der ersten Open Source Projekte in der Kartendigitalisierung.

10.2 Deutschland

Als erstes sei ein sehr einfaches Beispiel anhand einer Deutschlandkarte mit Städten und Flüssen gegeben. Zunächst wird wegen der Übersichtlichkeit mit nur einer Zoomstufe gearbeitet. Quellcode 10.1 zeigt die Konfigurationsdatei, mit der in der Applikation drei Layer erzeugt werden. Auf einem befinden sich die Landesgrenzen, auf den anderen beiden ein- und ausschaltbar die Flüsse und Städte. Es wird zunächst der Default-Converter verwendet, also keine Projektion angewandt. Eine Generalisierung findet in diesem Beispiel ebenfalls nicht statt. Die entstandene Applikation zeigt Abbildung 10.1.

Will man eine geografische Projektion anwenden, so gibt man im `configuration`-Tag das Attribut `converterclass` mit entsprechendem Klassennamen an. Die angegebene Klasse muss das Interface `Converter` implementieren (Kapitel 9.3.1) und kann

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<map xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="SVGWebMap.xsd">
  <configuration outputfolder="deu"/>
  <overview >
    <selection sourcefile="input/deu/lowres/country">
      <layout xsi:type="Polygon" stroke="black"
       stroke-width="0.01" fill="lightgreen"/>
    </selection>
  </overview>
  <layer changeable="false" visibility="visible"
   id="country" label="Land">
    <zoomstep action="add">
      <selection sourcefile="input/deu/hires/country">
        <layout xsi:type="Polygon" stroke="black"
         stroke-width="0.01" fill="lightgreen"/>
      </selection>
    </zoomstep>
  </layer>
  <layer changeable="true" visibility="visible"
   id="river" label="Fluesse">
    <zoomstep >
      <selection sourcefile="input/deu/hires/river" >
        <layout xsi:type="Polyline" stroke="#0000FF"
          stroke-width="0.02" aggregate="2"/>
      </selection>
    </zoomstep>
  </layer>
  <layer changeable="true" visibility="visible"
   id="cityhi" label="Staedte">
    <zoomstep >
      <selection sourcefile="input/deu/lowres/city" >
        <layout xsi:type="Point" stroke="none"
         fill="red" radius="0.05"/>
        </selection>
      <selection sourcefile="input/deu/lowres/city">
        <layout xsi:type="Text" font="Arial"
         font-fill="black" font-size="0.15" index="1"/>
      </selection>
    </zoomstep>
  </layer>
  <zoom>
    <zoomstep stepfactor="0.1" steps="2" zoomfactor="0.5"/>
  </zoom>
</map>
```

Quellcode 10.1: Einfache Konfigurationsdatei für Deutschlandkarte

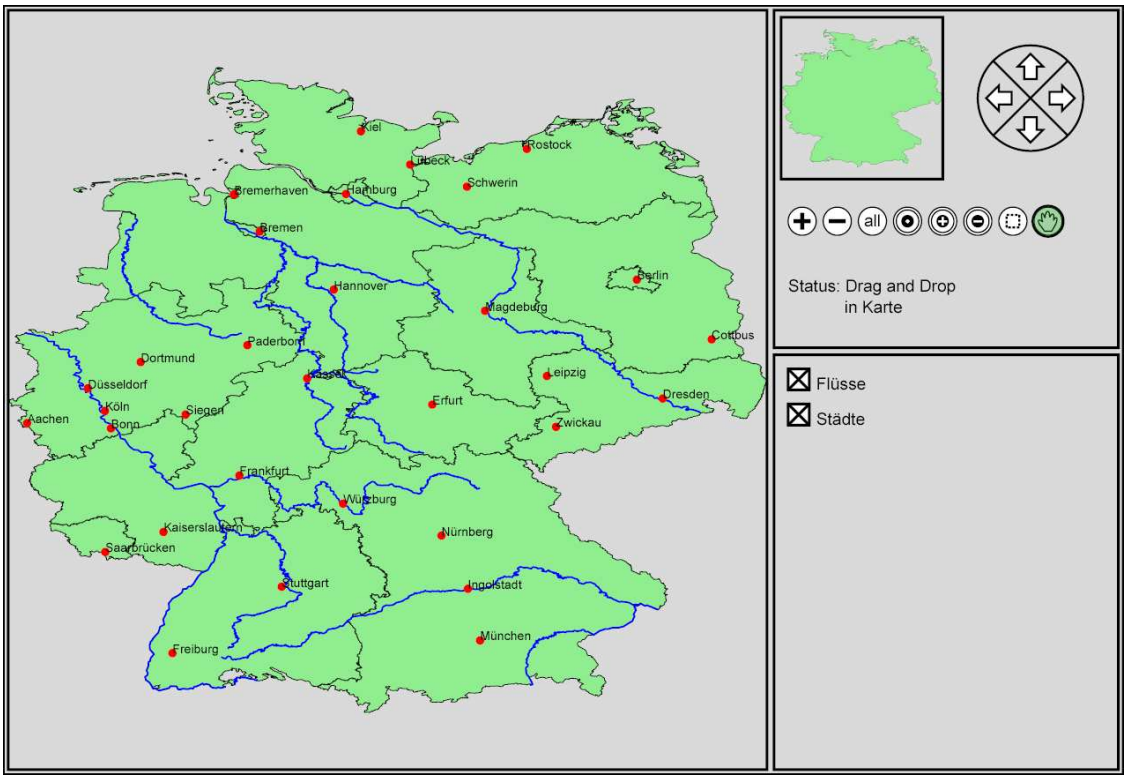


Abbildung 10.1: Einfache Deutschlandkarte

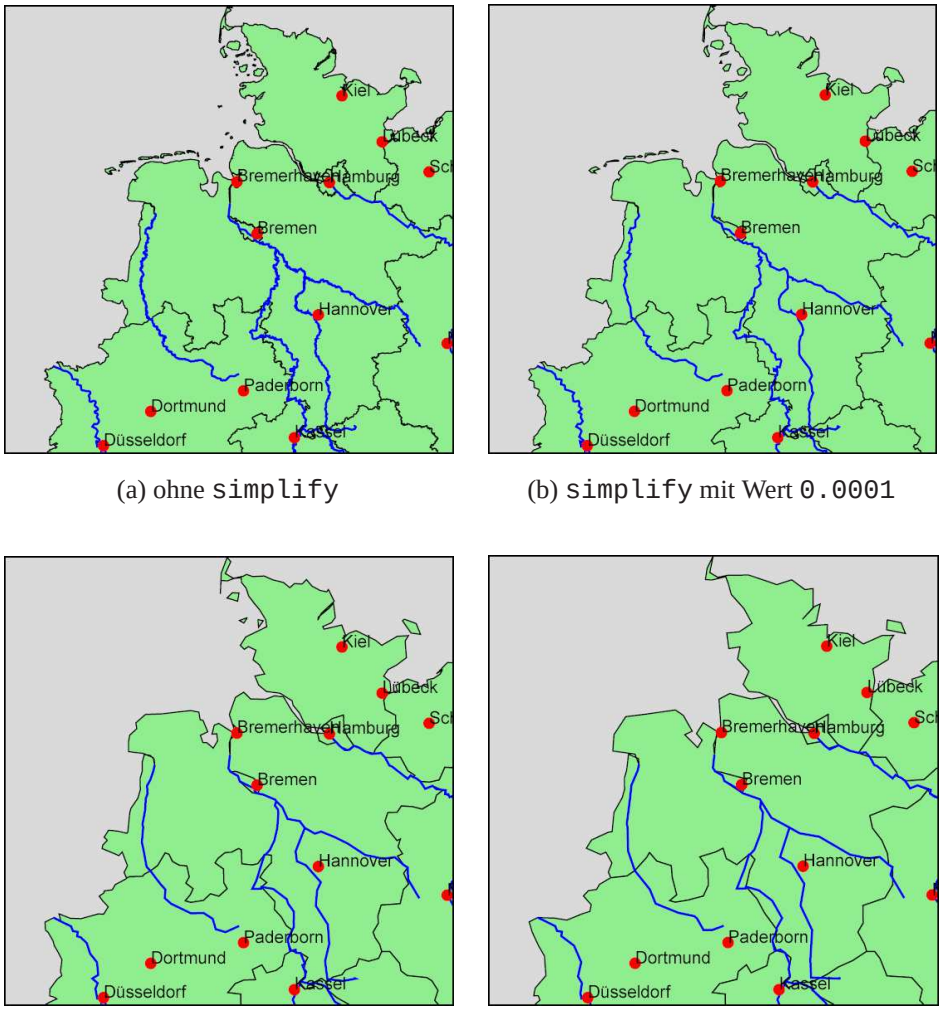
vom Benutzer selbst geschrieben werden. Als Beispiel wird hier die Lambertsche Azimuthal Projektion verwendet, die entsprechende Java-Klasse `Lambert_Convert` liegt im Verzeichnis `convert`. In der Konfigurationsdatei gibt man dementsprechend folgendes an:

```
<configuration outputfolder="deu"
 converterclass="convert.Lambert_Convert"/>
```

Der angebene `Converter` wird dann zur Laufzeit erzeugt und beim Einlesen der Daten im `ShpInputStream` verwendet. Wird keine Projektion angeben, wird die Default-Klasse `convert.Convert` verwendet, die lediglich das Vorzeichen der y-Koordinaten umkehrt (Kapitel 9.3.1). In Abbildung 10.2(a) ist die so erzeugte Deutschlandkarte zu sehen.

Die Daten können nicht nur projiziert sondern auch generalisiert werden. Dazu wird im `configuration`-Tag das Attribut `simplify` angegeben. Die Abbildungen 10.2(b)-(d) zeigen Karten mit verschiedenen Generalisierungsfaktoren.

Solch ein Generalisierungsfaktor über das Attribut `text-simplify` für Textpfade gesondert angegeben werden, falls es nicht erwünscht ist, dass sich der Text direkt am Pfad des geografischen Objektes orientiert. Solche Fälle treten beispielsweise bei Flüssen auf, die sich stark schlängeln. Abbildung 10.3 zeigt das Ergebnis bei unterschiedlicher Generalisierung von geografischen Objekten und Textpfaden.



(a) ohne `simplify` (b) `simplify` mit Wert `0.0001`

(c) `simplify` mit Wert `0.0005` (d) `simplify` mit Wert `0.001`

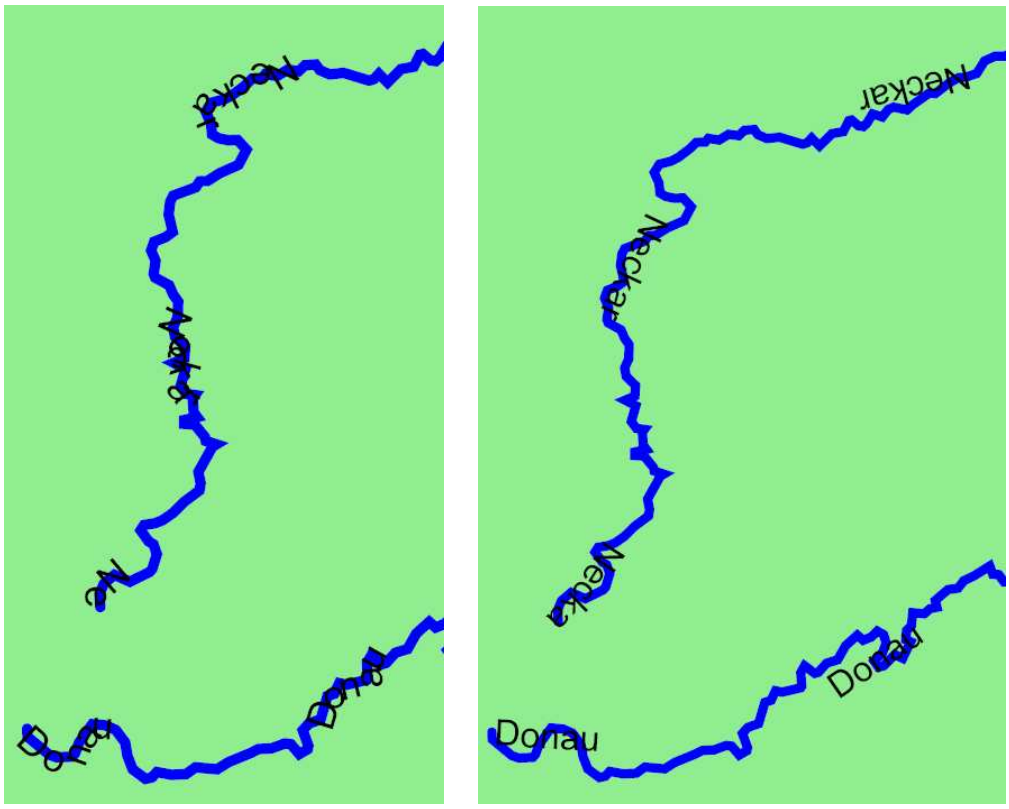
Abbildung 10.2: Verschiedene Generalisierung

10.3 Osnabrück

Aus Gründen der Übersicht wird keine vollständige Konfigurationsdatei zur Applikation der Osnabrücker Stadtplans abgedruckt. Anhand einiger zum Teil vereinfachter Ausschnitte soll auf wichtige Vorgehensweisen für das bestimmen des Layouts und weitere Konfigurationsmöglichkeiten eingegangen werden.

Wie in Kapitel 3.1.3 und Kapitel 9.5 erwähnt, lassen sich für Punktdaten nicht nur Kreise sondern auch Symbole verwenden. Um für die Kirchen im Osnabrücker Stadtplan ein Symbol zu verwenden, ist folgende Angabe in der Konfigurationsdatei notwendig:

```
<selection sourcefile="input/os/poi-joined">
  <param index="1" valuelist="6.0" include="include"/>
  <layout xsi:type="Point" symbolid="kirche" scale="2"/>
</selection>
```



(a) Textpfade mit gleichen Faktor wie geografische Objekte generalisiert

(b) Textpfade mit größerem Faktor als geografische Objekte generalisiert

Abbildung 10.3: Textpfad-Generalisierung

Im Verzeichnis `symbols`, das sich im Programmordner befindet, liegt unter dem Namen `kirche.psvg` eine Datei, welche folgende Symboldefinition einer Kirche enthält:

```
<path d="M -8,6 L -8, -5 L -5, -9 L -2, -5 L -2,-2 L 6,-2
  L 8,1 L8,6 L -8, 6 M -5, -5 L -5,-1 M -6, -4 L -4,-4"
  stroke="black" fill="grey" stroke-linecap="square"
  stroke-width="1"/>
```

In der vollständigen Beispiel-Applikation des Osnabrücker Stadtplans werden drei verschiedene Symbole für Kirchen, Krankenhäuser und Parkplätze verwendet (Abbildung 10.4). Will der Benutzer eigene Symbole verwenden, muss er lediglich eine entsprechende Datei mit der Endung `.psvg` schreiben und in das Verzeichnis `symbols` legen. Der Dateiname wird als `symbolid`-Attribut des `layout`-Tags mitgegeben. Die Symbole sollten immer mit ihrem Mittelpunkt im Ursprung des Koordinatensystems definiert sein, damit sie über die Angabe des `scale`-Attributes skaliert werden können. Diese Skalierung erfolgt relativ zum Ursprung, daher würde bei anders definierten Symbolen eine unerwünschte Positionierung stattfinden.

In einem Stadtplan soll für die Darstellung von Straßen häufig nicht nur eine einfarbige Linie, sondern zum Beispiel eine Linie mit Rand verwendet werden. Wie in Kapitel 3.1.1

Abbildung 10.4: Osnabrücker Stadtplan

beschrieben, ist dies in SVG nur durch das Übereinanderlegen von Linien unterschiedlicher Farbe und Breite möglich. Beispielsweise muss das Layout für die Straßen der Kategorie 2 und 3 im Osnabrücker Stadtplan (Abbildung 10.4) in der Konfigurationsdatei wie folgt angegeben werden:

```
<selection sourcefile="input/os/strassen-joined">
  <param index="2" include ="include"
   valuelist="2.0;3.0"/>
  <layout xsi:type="Polyline" stroke="black"
   stroke-width="15"/>
</selection >
<selection sourcefile="input/os/strassen-joined">
  <param index="2" include ="include"
   valuelist="2.0;3.0"/>
  <layout xsi:type="Polyline" stroke="yellow"
   stroke-width="13"/>
</selection>
```

Bei dieser Vorgehensweise werden zunächst die etwas breiteren schwarzen Linien, dann die gelben Linien gezeichnet. Bei der Beschreibung des Layouts ist die Reihenfolge der Angaben also entscheidend, um das gewünschte Layout zu erhalten.

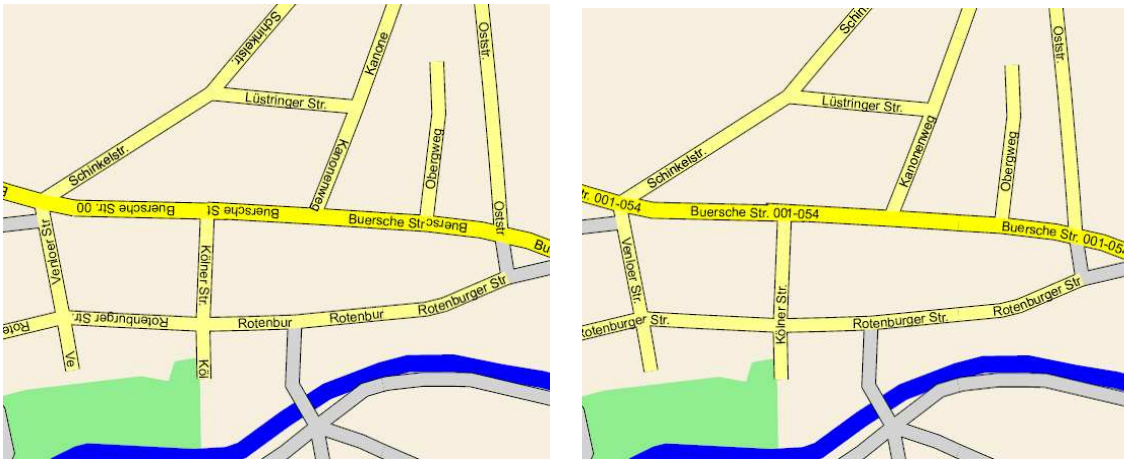
Der Layout-Typ `polyline` hat ein weiteres Attribut `aggregate`. Über die Angabe dieses Attributs kann man die Anzahl der benötigten `path`-Elemente reduzieren. In den Shapefiles sind die Straßen ungeordnet in vielen kleinen Abschnitten gespeichert. Wie in Kapitel 9.2 beschrieben, lassen sich Polylinien mit gleicher Eigenschaft zusammenfassen. Im Attribut `aggregate` muss der Index der Spalte im Database-File angegeben

werden, in der die entsprechende Eigenschaft steht. Im Beispiel des Osnabrücker Stadtplans stehen in Spalte 5 der `dbf`-Datei die Straßennamen. Man sollte also im Layout folgendes angeben:

```
<selection sourcefile="input/os/strassen-joined">
  <param index="2" include ="include"
   valuelist="2.0;3.0"/>
  <layout xsi:type="Polyline" stroke="black"
   stroke-width="15" aggregate="5"/>
</selection >
<selection sourcefile="input/os/strassen-joined">
  <param index="2" include ="include"
   valuelist="2.0;3.0"/>
  <layout xsi:type="Polyline" stroke="yellow"
   stroke-width="13" aggregate="5"/>
</selection>
```

Die Angabe hat auf den visuellen Eindruck der Karte kaum sichtbaren Einfluss, ist aber durchaus sinnvoll, da im Speicher eine geringere Zahl von Objekten verwaltet werden muss und so die Performanz der Applikation verbessert wird.

Bei der Beschriftung von Polylinien findet automatisch (ohne explizite Benutzerangabe) ein Zusammenfassen der Objekte anhand des Wertes der Beschriftung statt. Im Fall der Strassen ist dies wiederum der Straßenname. Für die Beschriftung entsteht dadurch nicht nur der Vorteil einer geringeren Anzahl von Objekten. Zusätzlich wird die Qualität der Beschriftung durch das Zusammenfassen und Sortieren der Teilstücke stark erhöht. Abbildung 10.5 zeigt das Ergebnis der Beschriftung ohne und mit Zusammenfassen der Polylinien.



(a) Text entlang nicht zusammengefasster Polylinien

(b) Text entlang sortierte und zussammengefasster Polylinien

Abbildung 10.5: `aggregate` bei Textpfaden

Je nach Zoomstufe können in der Applikation weitere Details nachgeladen werden. Dafür ist jeder Layer in der Konfigurationsdatei in `zoomsteps` eingeteilt (Kapitel 9.5). Man kann für jede Zoomstufe mit dem Attribut `action` angeben, ob Daten hinzu geladen oder ausgetauscht werden sollen. Angenommen, ein Layer soll die im Shapefile enthaltenen Kirchen beinhalten. Sie sollen in der ersten Zoomstufe gar nicht angezeigt, in der zweiten als schwarze Punkte und in der dritten Zoomstufe mit einem Symbol dargestellt werden. In der vierten Zoomstufe soll der jeweilige Name zusätzlich erscheinen. Die Konfigurationsangabe für den Kirchen-Layer sähe dann wie folgt aus:

```
<layer id="Kirchen">
 <zoomstep/>
 <zoomstep action="replace">
  <selection sourcefile="input/os/poi-joined">
   <param index="1" valuelist="6.0" include="include"/>
   <layout xsi:type="Point" fill="black" radius="10" />
  </selection>
 </zoomstep>
 <zoomstep action="replace">
  <selection sourcefile="input/os/poi-joined">
   <param index="1" valuelist="6.0" include="include"/>
   <layout xsi:type="Point" symbolid="kirche" scale="2"/>
  </selection>
 </zoomstep>
 <zoomstep action="add">
  <selection sourcefile="input/os/poi-joined">
   <param index="1" valuelist="6.0" include="include"/>
   <layout xsi:type="Text" font="Arial"
    font-fill="black" font-size="12" index="2"/>
  </selection>
 </zoomstep>
</layer>
```

Ein Austausch der Daten kann ebenfalls notwendig sein, um Liniendicken und Schriftgrößen anzupassen oder um die Objekte, die auf einem Layer liegen entsprechend ihrer Ebenen anzuordnen. Letzteres kann nötig sein, wenn die zuerst angezeigten Objekte über den neu hinzugefügten Objekten liegen sollen. Abbildungen 10.6-10.9 zeigt einige Ausschnitte vom Osnabrücker Stadtplan in unterschiedlichen Zoomstufen. Es werden bei unterschiedlichen Zoomstufen weitere Details wie Points of Interest und Straßen nachgeladen oder wenn nötig ausgetauscht. Ein Austausch ist hier zum Teil nötig, um Schriftgrößen anzupassen. Um einen besseren Eindruck vom Nachladen oder entfernen der Daten beim Auszoomen zu erhalten, sollten die Applikationen auf der genannten Webseite oder von der CD-Rom betrachtet werden.

Beim Verwenden von mehreren Zoomstufen ist zu beachten, dass alle definierten Layer die gleiche Anzahl `zoomstep`-Tags enthalten müssen wie im `zoom`-Element angegeben. In diesem Element werden alle Werte angegeben, die die Schrittweite der Navigation und die Zoomstufenwechsel bestimmen (Kapitel 9.5).

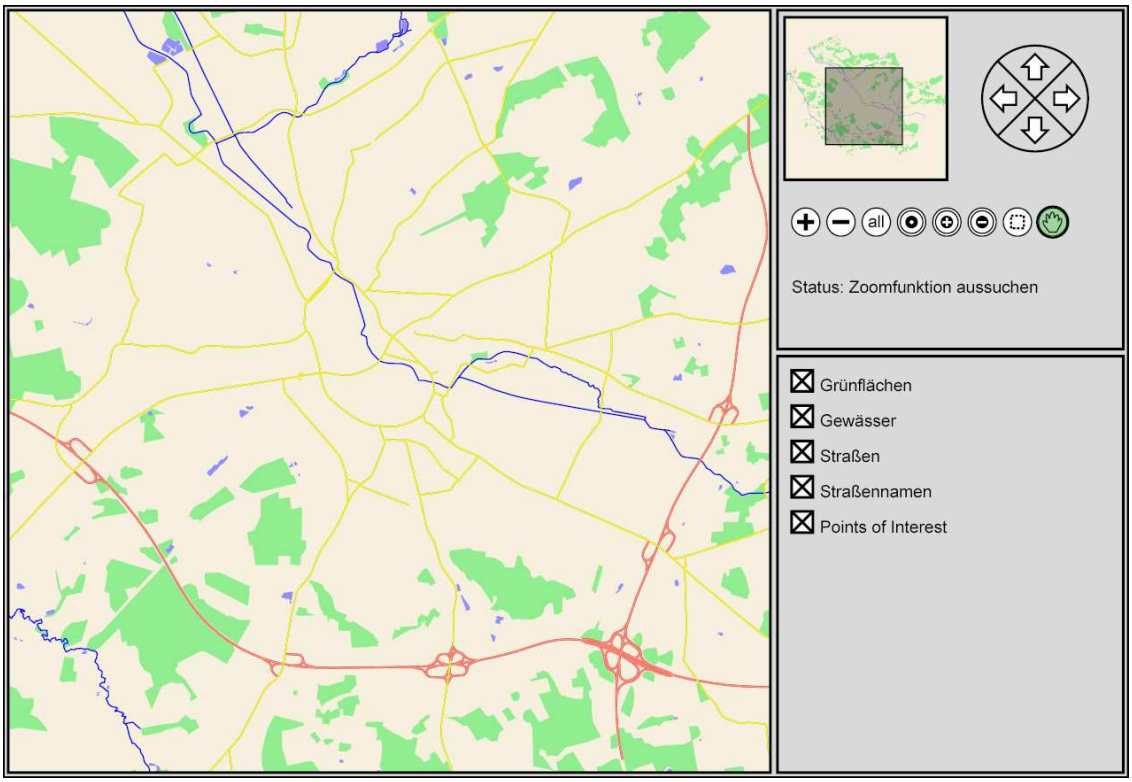


Abbildung 10.6: Zoomsstufe 1: Straßen der Kategorie 1, 2 und 3

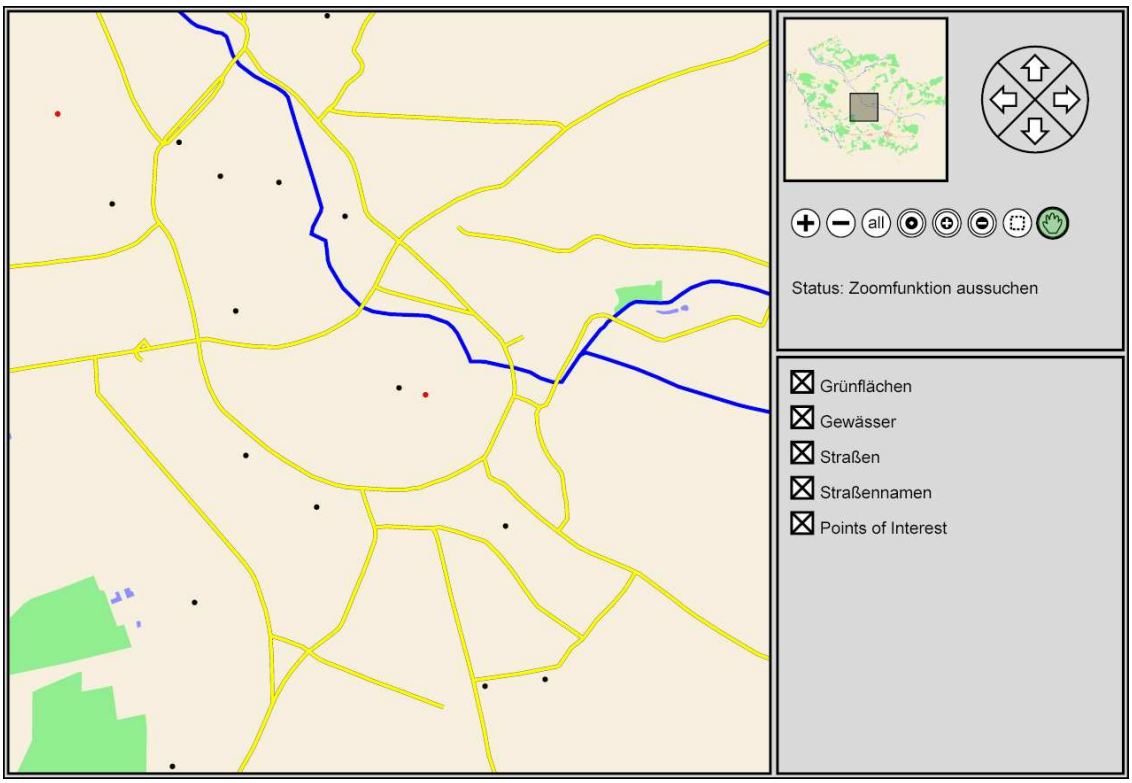


Abbildung 10.7: Zoomsstufe 2: Points of Interest kommen hinzu

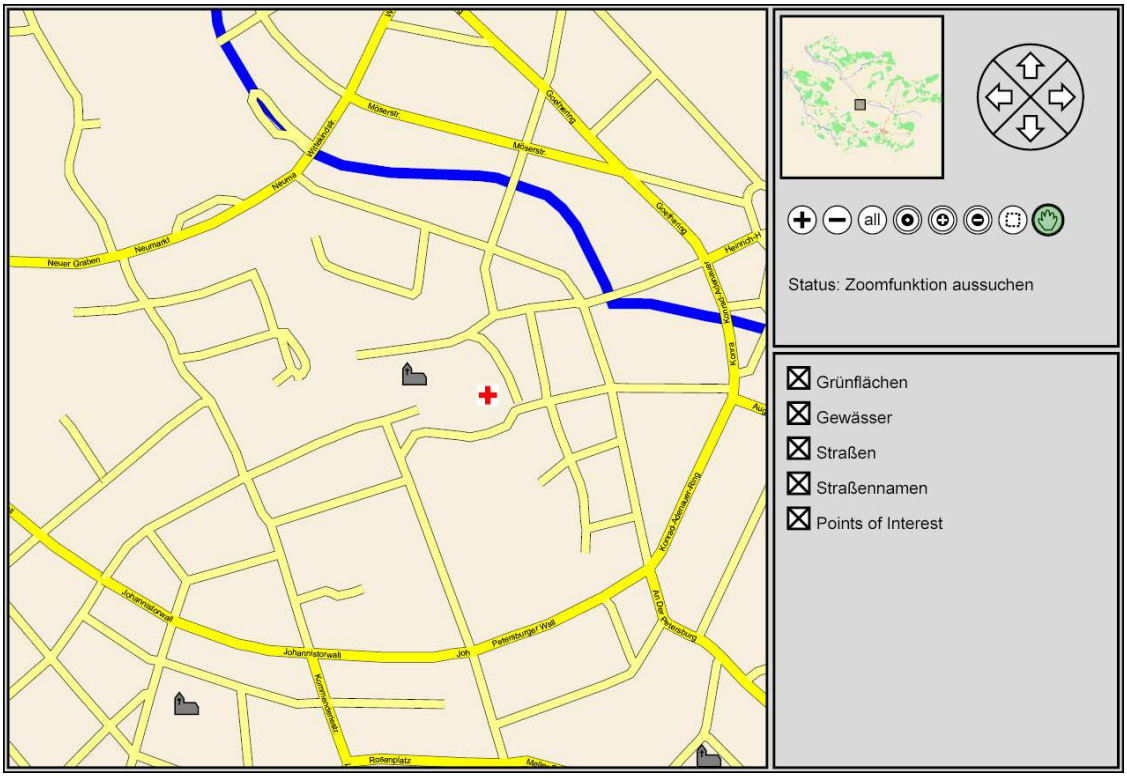


Abbildung 10.8: Zoomsstufe 3: Straßen der Kategorie 4 kommen hinzu, Symbole für Points of Interest

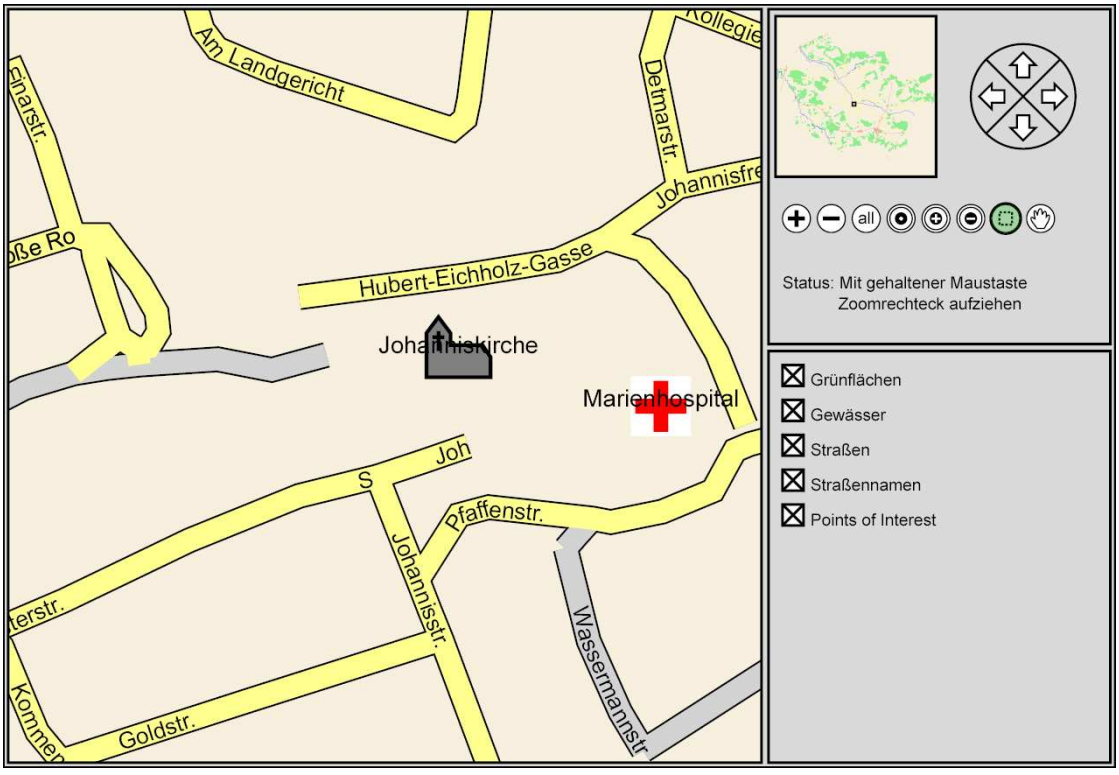


Abbildung 10.9: Zoomsstufe 4: alle Straßen, Beschriftung der Points of Interest

III Ausblick und Fazit

11 Ausblick

Die in der Applikation verwirklichten Interaktionen bieten dem Betrachter einen komfortablen Umgang mit dem Kartenmaterial, sie lassen sich aber noch erweitern. Ein wünschenswertes Feature wäre die Möglichkeit, weitere Informationen zu bestimmten geografischen Objekten anzuzeigen. Man könnte beispielsweise einen Mouseover-Effekt einfügen, so dass sich beim Überfahren eines Objektes mit der Maus ein Popup-Fenster mit Zusatzinformationen öffnet. Für so eine Funktionalität ließe sich ebenfalls ein eigener Button zum Ein- und Ausschalten einbauen.

Um weitere Interaktionsfunktionalitäten wie diese in die Applikation einzufügen, muss die Konfigurationsdatei angepasst und das verarbeitende Programm erweitert werden. Gegebenenfalls muss das erzeugte ECMAScript ergänzt werden. Solch eine Erweiterung wäre ebenfalls notwendig, wenn eine Legende zur Karte automatisch erstellt werden soll.

Beabsichtigt man in der Applikation die Visualisierung anderer georeferenzierter Daten, wäre es wünschenswert, innerhalb eines Layers Informationen vollständig austauschen zu können. Will man beispielsweise Klimadaten über viele Jahre hin betrachten, wäre es nicht sinnvoll (vom Platz für die Buttons her sogar unmöglich), für jedes Jahr einen eigenen Layer zu verwenden. Um geeignete Buttons zur Vefügung zu haben und den Austausch der Daten zu ermöglichen, müsste sowohl das ECMAScript als auch das PHP-Skript ergänzt werden.

Wie in Kapitel 9 erwähnt ist das automatisierte Beschriften von Karten ein umfangreiches Problem. In der vorliegenden Arbeit werden sehr einfache intuitive Verfahren benutzt, um die Beschriftungen zu platzieren. Eine sinnvolle Erweiterung des Programms wäre es, Beschriftungsheuristiken einzubauen. Dazu müsste man beispielsweise alle zu beschriftenden Daten speichern, auf sie einen Algorithmus anwenden und die Koordinaten anpassen. Gegebenenfalls muss dann die Schriftgröße automatisch angepasst werden und kann dann nicht mehr vom Benutzer, der die Karten erstellen will, beeinflusst werden.
In den Web Mapping Applikationen muss die Anpassung der Schriftgröße beim Zoomstufenwechsel in der Konfigurationsdatei vom Benutzer bestimmt werden. Das Schreiben der Konfigurationsdatei und das Betrachten der Karten in der Web Mapping Applikation wäre noch komfortabler, wenn die Schriftgröße bei jedem Zoomvorgang automatisch zur Laufzeit angepasst würde. Solch ein Anpassen ließe sich über ECMAScript verwirklichen, eventuell muss dann allerdings mit Stylesheets [W3C2005f] gearbeitet werden. Welchen Einfluss solch ein Anpassen auf die Performanz der Applikation hat, bleibt zu testen. Da jedes Textobjekt angesprochen und an ihm mindestens ein Attribut verändert werden muss, könnte das Anpassen der Schriftgröße im Verhältnis zu anderen Interaktionen relativ viel Zeit in Anspruch nehmen.

In dieser Arbeit wurde versucht, die Konfigurationsdatei möglichst einfach aufzubauen (Kapitel 9.5). Um das Erstellen einer Applikation mit eigenen Geodaten noch weiter zu erleichtern, könnte man eine grafische Oberfläche entwickeln, die einen Benutzer beim Erstellen der Konfigurationsdatei unterstützt und das Java-Programm „auf Knopfdruck“ startet. So wäre keinerlei Kenntnis von XML oder Java nötig, um das entstandene Programm nutzen zu können und eine noch breitere Anwenderschicht wäre ansprechbar.

Ein weiteres Feature wäre, die Beeinflussung des Layouts der Web Mapping Applikation in weit höherem Maße als bisher möglich. Das Setzen von Farbwerten reicht nicht immer aus, um eine Applikation nach den eigenen Wünschen zu erstellen. Für manche Geodaten wäre es sicherlich sinnvoll, wenn die Anzeige der Karten nicht immer quadratisch wäre (zum Beispiel bei einer Weltkarte) und die Anordnung der einzelnen Platzhalter im Template (Kapitel 8.1) variiert werden könnte. Dies ließe sich über eine weitere Konfigurationsdatei oder eine Erweiterung des bereits bestehenden Konfigurationsschemas verwirklichen. Weiterhin müsste das Skript für die Interaktion, also für alle Navigations- und Zoomvorgänge angepasst werden, weil in der aktuellen Version nur mit einer quadratischen Anzeige gearbeitet wird. Ebenso müssten im Java-Programm die Berechnungen der Kacheln und die Berechnungen für minimale und maximale Viewbox-Werte angepasst werden.

In dieser Arbeit dienen als Quelle für die Geodaten ausschließlich Shapefiles (Kapitel 6 und 9.3). Das Programm ließe sich dahingehend erweitern, dass auch andere Datenformate als Quelle genutzt werden könnten. Hierfür würden Bibliotheken oder andere Programme benötigt, mit denen man die entsprechenden Daten auslesen und aus ihnen die hier benutzten geometrischen Objekte, also Instanzen der Klassen `Point`, `Polygon` und `Polyline` erzeugen kann.

Durch den modularen Aufbau der Applikation sind auch weitreichende Erweiterungen beziehungsweise Änderungen relativ leicht umzusetzen. Hätte man eine Möglichkeit, die SVG-Karten wesentlich schneller zu erzeugen, könnte man die gesamte Applikation zu einem dynamischen Webmapserver (Kapitel 7) machen. Dazu wäre im PHP-Skript statt des Auslesens der vorberechneten Grafiken ein Aufruf eines Programms nötig, das die Karten dynamisch erstellt. Die Verarbeitung der Konfigurationsdatei sowie das Erstellen der Kartengrafiken (Kapitel 9) müssten dann zur Laufzeit erfolgen.

12 Fazit

Mit der vorliegenden Arbeit ist ein Tool entstanden, das auf relativ einfache Weise geografische Daten visualisiert. Ist man im Besitz von Shapefiles, beschränkt sich der Aufwand, eine interaktive Web Mapping Applikation im Internet zu veröffentlichen, auf folgende Schritte:

1. Konfigurationsdatei schreiben (Kapitel 9.5)
2. Java-Programm ausführen, eventuell mit Encoding-Angabe und Speicherplatzerweiterung (Kapitel 9.6.1)
3. Alle erzeugten Dateien auf einen Server mit PHP5-Unterstützung legen

Durch die einfach aufgebaute Konfigurationsdatei ist die entstehende Web Mapping Applikation und ihr Layout sehr leicht konfigurierbar. Das Java-Programm übernimmt alle anderen nötigen Aufgaben wie das Auslesen der Shapefiles, geografische Projektionen und das Erzeugen der Karten als SVG-Dateien (Kapitel 9). Die Darstellung der Karten übernimmt der Webbrowser mit Plugin Kapitel 3.2. Durch die Möglichkeit, den DOM-Tree eines XML-Dokumentes mit Hilfe von ECMAScript (Kapitel 4) zu manipulieren, sind viele Interaktionen mit dem Kartenmaterial für den Betrachter in den erzeugten Web Mapping Applikationen verwirklicht (Kapitel 8.2). Man kann in der Karte navigieren, es lassen sich Ausschnitte wählen und verschiedene Daten über das Verwenden von Layern beliebig kombinieren. So lässt sich eine erzeugte Applikation als Grundlage für andere georeferenzierte Daten wie Wahlergebnisse, Gewässerverschmutzung, Klima oder Wetter verwenden. Ein Beispiel für das zusätzliche Anzeigen von Temperatur Isoflächen zeigt Abbildung 12.1.

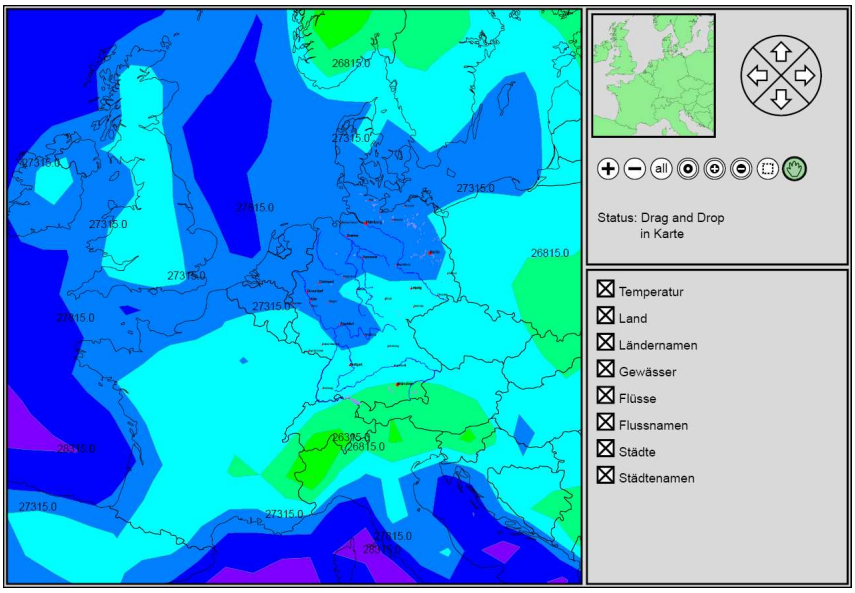


Abbildung 12.1: Zusätzlich eingefügte Klimadaten

Die Qualität der angezeigten Grafiken ist aufgrund des Vektorgrafik-Formates im Vergleich zu manchen pixelbasierten Web Mapping Applikationen sehr hoch (Kapitel 3). Außerdem erreicht man durch die Möglichkeit, zusätzliche Daten in die Karte zu laden, ohne alle bereits vorhandenen Daten austauschen zu müssen, geringeren Datenverkehr und kurze Wartezeiten auf Clientseite (Kapitel 8.4).

Aufgrund des in Kapitel 8.1 beschriebenen Template-Konzeptes sind die bereits bestehenden Interaktionsmöglichkeiten leicht zu erweitern (Kapitel 8.2) und erfordern keinen Eingriff in die Struktur der Applikation oder des Java-Programms. Weitere Erweiterungen wie in Kapitel 11 vorgeschlagen sind durch die klare Strukturierung der Web Mapping Applikation (Kapitel 8) und der Java-Klassen (Kapitel 9) verhältnismäßig einfach zu verwirklichen.

IV Anhang

Einige der abgedruckten Dateien enthalten Zeilenumbrüche, die in der Originaldatei nicht enthalten sein dürfen, aufgrund der begrenzten Zeilenlänge hier jedoch eingefügt wurden.

A SVGWebMap.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!--Root-Element of the Konfigurationsdatei-->
  <xs:element name="map">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="configuration" type="configType"
         minOccurs="1" maxOccurs="1"/>
        <xs:element name="overview" type="overviewType"
         minOccurs="1" maxOccurs="1"/>
        <xs:element name="layer" type="layerType"
         minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="zoom" type="zoomType"
         minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!--Some Basic-Values for the application-->
  <xs:complexType name="configType" mixed="false">
    <xs:attribute name="converterclass" type="xs:anyURI"
     default="convert.Convert"/>
    <xs:attribute name="title" type="xs:string"
     default="SVG Web Mapping"/>
    <xs:attribute name="zoomAndPan" type="ableType"
     default="disable"/>
    <xs:attribute name="outputfolder" type="xs:anyURI"
     use="required"/>
    <xs:attribute name="bgcolor" type="colorType"
     default="#dadada"/>
    <xs:attribute name="displaybgcolor" type="colorType"
     default="#dadada"/>
    <xs:attribute name="symbolhighlightcolor"
     type="colorType" default="green"/>
    <xs:attribute name="textcolor" type="colorType"
     default="black"/>
    <xs:attribute name="strokecolor" type="colorType"
     default="black"/>
    <xs:attribute name="symbolfillcolor" type="colorType"
     default="white"/>
```

```
    <xs:attribute name="simplify" type="linewidthType"
     default="0"/>
    <xs:attribute name="text-simplify"
     type="linewidthType" default="0"/>
    <xs:attribute name="scale" type="linewidthType"
     default="1"/>
    <xs:attribute name="vbX" type="xs:double"/>
    <xs:attribute name="vbY" type="xs:double"/>
    <xs:attribute name="vbWidth" type="xs:double"/>
    <xs:attribute name="script" type="scriptType"
     default="v1"/>
  </xs:complexType>
  <!--Basetype for different layouts-->
  <xs:complexType name="layoutType" abstract="true"
   mixed="false">
    <xs:attribute name="stroke" type="colorType"
     default="black"/>
    <xs:attribute name="stroke-width" type="linewidthType"
     default="1.0"/>
  </xs:complexType>
  <!--The types which inherit from the base-layout-type-->
  <xs:complexType name="Polygon">
    <xs:complexContent>
      <xs:extension base="layoutType">
        <xs:attribute name="fill" type="colorType"
         default="none"/>
        <xs:attribute name="stroke-linecap"
         type="linecapType" default="butt"/>
        <xs:attribute name="stroke-linejoin"
         type="linejoinType" default="miter"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Polyline">
    <xs:complexContent>
      <xs:extension base="Polygon">
        <xs:attribute name="aggregate" type="indexType"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Point">
    <xs:complexContent>
      <xs:extension base="layoutType">
        <xs:attribute name="fill" type="colorType"
         default="black"/>
        <xs:attribute name="radius" type="linewidthType"
```

```
        default="1"/>
       <xs:attribute name="scale" type="linewidthType"
        default="1"/>
       <xs:attribute name="symbolid" type="xs:string"/>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="Text">
   <xs:complexContent>
     <xs:extension base="layoutType">
       <xs:attribute name="index" type="indexType"
        use="required"/>
       <xs:attribute name="font" type="xs:string"
        default="Times"/>
       <xs:attribute name="font-fill" type="colorType"
        default="black"/>
       <xs:attribute name="font-size"
        type="linewidthType" default="1"/>
       <xs:attribute name="text-anchor" type="anchorType"
        default="start"/>
       <xs:attribute name="font-style" type="styleType"
        default="normal"/>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="TextPath">
   <xs:complexContent >
     <xs:extension base="Text">
       <xs:attribute name="repeat" type="countType"
        default="1" />
       <xs:attribute name="spacing" type="indexType"
        default="1" />
       <xs:attribute name="offset" type="offsetType"
        default="0" />
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
 <!--Param to restrict the chosen graphics-->
 <xs:complexType name="paramType" mixed="false">
   <xs:attribute name="index" type="indexType"
    use="required" />
   <xs:attribute name="include" type="includeType"
    use="required" />
   <xs:attribute name="valuelist" type="xs:string"
    use="required" />
   <xs:attribute name="delimiter" type="xs:string"
```

```
    default=";" />
  </xs:complexType>
  <!--Generell zoomstep-values-->
  <xs:complexType name="zoomstepType" mixed="false">
    <xs:attribute name="zoomfactor" type="faktorType"
     default="0.5"/>
    <xs:attribute name="stepfactor" type="faktorType"
     default="0.1"/>
    <xs:attribute name="steps" type="countType"
     default="1"/>
  </xs:complexType>
  <!--Type to select the graphics, which shall be used-->
  <xs:complexType name="selectType" mixed="false">
    <xs:sequence>
      <xs:element name="param" type="paramType"
       minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="layout" type="layoutType"
       minOccurs="1" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="sourcefile" type="xs:anyURI"
     use="required"/>
    <xs:attribute name="converterclass" type="xs:anyURI"/>
  </xs:complexType>
  <!--Decribes source and layout and action which is made
      when data will be loaded-->
  <xs:complexType name="layerzoomType" mixed="false">
    <xs:sequence>
      <xs:element name="selection" type="selectType"
       minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="action" type="actionType"
     default="add"/>
  </xs:complexType>
  <!--A Layer can be switched on and off,
      can contain several sources-->
  <xs:complexType name="layerType" mixed="false" >
    <xs:sequence>
      <xs:element name="zoomstep" type="layerzoomType"
       minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="label" type="xs:string" />
    <xs:attribute name="changeable" type="xs:boolean"
     default="true" />
    <xs:attribute name="visibility" type="visibleType"
     default="visible" />
```

```
</xs:complexType>
<!--Contains all values for the zoomsteps-->
<xs:complexType name="zoomType" mixed="false" >
  <xs:sequence>
    <xs:element name="zoomstep" type="zoomstepType"
     minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<!--Describes the values for the overview,
    similar to a layer-->
<xs:complexType name="overviewType" mixed="false" >
  <xs:sequence>
    <xs:element name="selection" type="selectType"
     minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="simplify" type="linewidthType"
   default="0" />
</xs:complexType>
<!-- some usefull simpleTypes-->
<xs:simpleType name="linewidthType">
  <xs:restriction base="xs:double">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="faktorType">
  <xs:restriction base="xs:double">
    <xs:minExclusive value="0"/>
    <xs:maxExclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="indexType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="countType">
  <xs:restriction base="xs:integer">
     <xs:minInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="offsetType">
  <xs:restriction base="xs:double">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="linecapType">
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="round"/>
      <xs:enumeration value="butt"/>
      <xs:enumeration value="square"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="linejoinType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="round"/>
      <xs:enumeration value="bevil"/>
      <xs:enumeration value="miter"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="colorType">
    <xs:restriction base="xs:string">
      <xs:pattern
       value="#[0-9a-fA-F]{6}|lightblue|lightgreen|brown|
       grey|lightgrey|pink|magenta|cyan|none|black|maroon|
       olive|white|silver|red|yellow|green|teal|navy|
       purple|lime|aqua|blue|fuchsia"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="anchorType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="start"/>
      <xs:enumeration value="middle"/>
      <xs:enumeration value="end"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="styleType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="normal"/>
        <xs:enumeration value="italic"/>
        <xs:enumeration value="oblique"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="visibleType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="visible"/>
      <xs:enumeration value="hidden"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="actionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="add"/>
      <xs:enumeration value="replace"/>
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ableType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="disable"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="includeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="include"/>
      <xs:enumeration value="exclude"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="scriptType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="v1"/>
        <xs:enumeration value="v2"/>
      </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B load_v1.js

```
function manage_loading(data){
 if(data.success){
  st=data.content;
  var node=parseXML(st,document);
  var nch1=node.childNodes;
  var dat=nch1.item(0);
  var del=dat.firstChild;
  var app=dat.lastChild;
  var list=del.childNodes;
  for(j=0;j<list.length;j++){
     el=list.item(j);
     id=el.getAttributeNS(null,'id');
     child= document.getElementById(id);
     if(child!=null){
        child.parentNode.removeChild(child);
     }
     else{
        var id_old;
        var found=false;
        var i=0;
        while(!found&&i<idstodelete.length){
          id_old=idstodelete.item(k)
                 .getAttributeNS(null,'id');
          found=(id==id_old);
          i++;
        }
        if(!found){
          var g=document.createElementNS(null,'g');
          g.setAttributeNS(null,'id',id);
          idstodelete.appendChild(g);
        }
     }
  }
  list=app.childNodes;
  while(list.length!=0){
     el=list.item(0);
     var p=el.getAttributeNS(null,'parent');
     var id=el.getAttributeNS(null,'id');
     el.removeAttributeNS(null,'parent');
     document.getElementById(p).appendChild(el);
  }
  var nodelist=idstodelete.childNodes;
  var index=0;
  for(m=0;m<nodelist.length;m++){
```

```
      el=nodelist.item(m);
      id=el.getAttributeNS(null,'id');
      child= document.getElementById(id);
      if(child!=null){
         child.parentNode.removeChild(child);
         idstodelete.removeChild(el);
         m=index;
         nodelist=idstodelete.childNodes;
      }
      else{
         index++;
      }
   }
  }
}
```

C load_v2.js

```
function manage_loading(data){
 if(data.success){
  st=data.content;
  var node=parseXML(st,document);
  var nch1=node.childNodes;
  var dat=nch1.item(0);
  var del=dat.firstChild;
  var app=dat.lastChild;
  var list=del.childNodes;
  for(j=0;j<list.length;j++){
     el=list.item(j);
     id=el.getAttributeNS(null,'id');
     child= document.getElementById(id);
     if(child!=null){
        child.parentNode.removeChild(child);
     }
     else{
       var id_old;
       var found=false;
       var i=0;
       while(!found&&i<idstodelete.length){
         id_old=idstodelete.item(k)
                .getAttributeNS(null,'id');
         found=(id==id_old);
         i++;
       }
       if(!found){
         var g=document.createElementNS(null,'g');
         g.setAttributeNS(null,'id',id);
         idstodelete.appendChild(g);
       }
     }
  }
  list=app.childNodes;
  for(j=0;j<list.getLength();j++){
     el=list.item(j);
     var p=el.getAttributeNS(null,'parent');
     var file=el.getAttributeNS(null,'file');
     appendFile(file,p);
  }
  var nodelist=idstodelete.childNodes;
  var index=0;
  for(m=0;m<nodelist.length;m++){
     el=nodelist.item(m);
```

```
      id=el.getAttributeNS(null,'id');
      child= document.getElementById(id);
      if(child!=null){
         child.parentNode.removeChild(child);
         idstodelete.removeChild(el);
         m=index;
         nodelist=idstodelete.childNodes;
      }
      else{
         index++;
      }
   }
  }
}
```

D Inhalt der CD-Rom

arbeit	**Ausarbeitung der Diplomarbeit**
arbeit/latex	Latex-Quellen, Grafiken, Beispielcode
arbeit/pdf	PDF-Version
programm	**Java-Klassen**
programm/convert	Converter-Klassen
programm/export	Klassen für das Hauptprogramm `Creator`
programm/io	Klassen zum Lesen von Shapefiles
programm/input	Shapefiles zu Deutschland und Osnabrück
programm/schema	SVGWebMap.xsd, Konfigurationsdateien
programm/script	ECMAScript, PHP-Skript
programm/shapes	Grafik-Objekte
programm/symbols	Dateien mit Symboldefinitionen
programm/util	Hilfs-Klassen
docs	**Dokumentation**
docs/programm	Javadoc des Java-Programms
docs/java	J2SE(TM) Development Kit Documentation 5.0
docs/xerces	Xerces2 API
app	**Beispiel Applikationen**
app/deu	Deutschland
app/os	Osnabrück
software	**benötigte Software**
software/adobe	Adobe SVG Viewer 3.03 und 6 beta (Windows) Adobe SVG Viewer 3.01 (Linux)
software/java	J2SE(TM) Development Kit 5.0 Update 6 (Windows und Linux)
software/xampp	xampp-Installation [Apac2005] (Windows und Linux)

E Literaturverzeichnis

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

Literatur

[Adob2005] **Adobe Systems**
Adobe SVG Zone
http://www.adobe.com/svg

[Apac2004a] **Apache Foundation**
Apache Xerces2 Java Parser
http://xml.apache.org/xerces2-j

[Apac2004b] **Apache Foundation**
API JavaDoc
http://xml.apache.org/xerces2-j/api.html

[Apac2004c] **Apache Foundation**
Batik SVG Toolkit
http://xml.apache.org/batik

[Apac2001] **Apache**
Apache XML Project
http://xml.apache.org

[Apac2005] **Apache friends**
xampp
http://www.apachefriends.org/de/xampp.html

[Apti2004] **Aptico GmbH**
SVG Tutorial
http://svg.tutorial.aptico.de

[BeMi2000] **Behme, Henning ; Mintert, Stefan**
XML in der Praxis, 2. erweiterte Auflage
Addison Wesley Lomgman Verlag, Pearson Education Deutschland GmbH 2000

[BuSn1995] **Bugayevskiy, Lev M.; Snyder, John P.**
Map Projections. A Reference Manual
Taylor & Francis, London 1995

[Cart2005] **Cartogis**
Digitale Landkarten
http://www.cartogis.de/parallel/karten/digkart.htm

[DeGK2005] **Deutsche Gesellschaft für Kartographie e.V.**
Deutsche Gesellschaft für Kartographie e.V.
http://www.dgfk.net

[Ecma1999] **ECMA International**
Standard ECMA-262: ECMAScript Language Specification
http://www.ecma-international.org/publications/
standards/Ecma-262.html

[Eise2002] **Eisenberg, J. David**
SVG Essentials
O'Reilly, Köln 2002

[Esri2005a] **ESRI**
ESRI - The GIS Software Leader
http://www.esri.com

[Esri2005b] **ESRI**
ESRI World Basemap Data
http://www.esri.com/data/download/basemap/index.html

[Esri2005c] **ESRI**
ArcExplorer GIS Data Viewer
http://www.esri.com/software/arcexplorer/index.html

[Esri1998] **ESRI**
ESRI Shapefile Technical Description
http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[Frid2005] **Intevation GmBH**
Frida: Freie Vektor-Geodaten Osnabrck
http://frida.intevation.org/

[Gali2005] **Galileo Computing**
JavaScript - Galileo Computing - OpenBook
http://www.galileocomputing.de/openbook/javascript

[GHJV1995] **Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John**
Design Patterns. Elements of Reusable Object-Oriented Software
Addison Wesley 1995

[HaMe2005] **Harold, Elliotte Rusty; Means, W. Scot**
XML in a nutshell, 3.Auflage
O'Reilly 2005

[Flan2002] **Flanagan, David**
JavaScript. Das umfassende Referenzwerk.
O'Reilly 2002

[GrHo1998] **Greiner G.; Hormann K.**
Efficient clipping of arbitrary polygons
ACM Transactions on Graphics, 1998

[Konq2005] **Konqueror**
Konqueror Homepage http://www.konqueror.org

[KuMeVo2005] **Kunze, Ralf; Mertens, Robert; Vornberger, Oliver**
Dynamic and interactive visualization of weather data with SVG
http://www.svgopen.org/2005/2005/papers/
DynamicInteractiveWeatherData/index.html

[Mac2005] **Macromedia**
Macromedia
http://www.macromedia.com

[MaSe2005] **MapServer**
MapServer
http://mapserver.gis.umn.edu

[Mei2005] **Meinike, Thomas**
SVG - Learning by Coding
http://www.datenverdrahten.de/svglbc

[Moz2005] **Mozilla**
Firefox 1.5
http://www.mozilla.com/firefox/

[Moz2004] **Mozilla Organisation**
Mozilla SVG Project
http://www.mozilla.org/projects/svg

[Oper2005] **Opera** Opera 8.5
http://www.opera.com/

[OpGe2005] **OGC**
OEG Open Geospatial Consortium
http://www.opengeospatial.org

[OpMa2005a] **BBN Technologies**
OpenMap
http://openmap.bbn.com/http://openmap.bbn.com

[OpMa2005b] **BBN Technologies**
OpenMap API
http://openmap.bbn.com/doc/api

[OpMa2005c] **BBN Technologies**
OpenMap Software License Agreement
http://openmap.bbn.com/license.html

[PHP2005a] **PHP**
PHP: Hypertext Preprocessor
http://www.php.net

[PHP2005b] **PHP**
PHP: PHP Handbuch - Manual
http://www.php.net/manual/de

[Pila2005] **Pilat Informative Educative**
Javascript, SVG and DOM
http://pilat.free.fr/english/routines/js_dom.htm

[Plew1997] **Plewe, Brandon**
GIS Online - Information retrieval, mapping and the Internet
OnWord Press 1997

[SAX2004] **Mailingliste `xml-dev`**
Simple API for XML
http://sax.sourceforge.net

[Sch1995] **Schmidt, Vasco Alexander**
Reine Forschung, praktische Resultate Beschriftung von Karten ist ein hartes Problem. Abstrakte Theorien führen zu seiner Lösung
http://i11www.ira.uka.de/map-labeling/points/one-square/info/zeit.html

[Sun2005] **Sun Developer Network**
J2SE(TM) Development Kit 5.0 Update 6
http://java.sun.com/downloads

[Sun2005] **Sun Developer Network**
API Specifications
http://java.sun.com/reference/api/index.html

[Tarr2004] **Tarr, Bob**
The Composite Pattern
http://www.research.umbc.edu/ tarr/dp/lectures/Composite.pdf

[Vorn2004] **Vornberger, Oliver**
Vorlesung Computergrafik
http://www-lehre.inf.uos.de/ cg

[W3C2005a] **World Wide Web Consortium**
World Wide Web Consortium
http://www.w3c.org

[W3C2005b] **W3C Recommendation**
Extensible Markup Language (XML)
http://www.w3.org/TR/REC-xml

[W3C2005c] **W3C Recommendation**
Scalable Vector Graphics (SVG) 1.1 Specification
http://www.w3c.org/TR/SVG11

[W3C2005d] **W3C Recommendation**
Mobile SVG Profiles: SVG Tiny and SVG Basic
http://www.w3c.org/TR/SVGMobile

[W3C2005e] **W3C**
SVG Implementations
http://www.w3.org/Graphics/SVG/SVG-Implementations.htm8#viewer

[W3C2005f] **W3C Recommendation**
Cascading Style Sheets (CSS), level 2
http://www.w3c.org/TR/REC-CSS2

[W3C2005g] **W3C Recommendation**
Document Object Model (DOM), level 3
http://www.w3.org/TR/DOM-Level-3-Core

[W3C2005h] **W3C**
XML Schema
http://www.w3.org/XML/schema

[WaLi2004] **Watt, Andrew H.; Lilley, Chris; et al.**
SVG Unleashed

[Warm2000] **Warmerdam, Frank**
Shapefile C Library
http://members.home.com/warmerda

[Watt2005] **Watt, Jonathan**
SVG Authoring Guidelines
http://jwatt.org/svg/authoring

[Wiki2005] **Wikipedia**
Wikipedia
http://de.wikipedia.org/wiki/Hauptseite

[Wolf2002] **Wolff, Alexander**
Map Labeling
http://i11www.ira.uka.de/map-labeling

[xml2005] **O'Reilly xml.com**
SVG Tips and Tricks: Adobe's SVG Viewer
http://www.xml.com/pub/a/2002/07/03/adobesvg.html

Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig angefertigt und keine Hilfsmittel außer den in der Arbeit angegebenen benutzt habe.

Osnabrück, den 17.02.2006 .....

(Unterschrift)