Patrick Cernko
Mainstraße 2
66125 Saarbrücken - Dudweiler
10. Fachsemester Informationswissenschaft (Nebenfach)

Universität des Saarlandes, WS 2000/01
Forschungsseminar (P): Elektronisches Publizieren (am Beispiel von F. Nietzsche)
Leiter: Prof. Dr. H. Zimmermann

Proseminararbeit

Technologie XML

Patrick Cernko

7. Februar 2002

Inhaltsverzeichnis

1. Einführung

1.1. Entstehung von XML

XML steht für „eXtensible Markup Language", was man ungefähr mit „erweiterbare Auszeichnungssprache" übersetzen kann. Auszeichnungssprachen haben eine lange Tradition mit der Auszeichnungssprache SGML, die schon in den 60er Jahren des vergangenen Jahrhunderts entstand [11]. In ihr wurde erstmals die Idee der Trennung von „Layout und Content" (vgl. Abschnitt 4.1) realisiert. SGML ließ dabei große Freiheiten bei der Erstellung der Dokumentenstruktur. Damit einher ging jedoch auch eine erhöhte Schwierigkeit bei der maschinellen Verarbeitung. Nichtsdestotrotz muss man würdigen, dass SGML erstmals überhaupt ermöglichte, Dokumente standardisiert zu strukturieren und damit maschinell zu verarbeiten.

XML ist nun eine „Anwendung" von SGML, d.h. XML ist mit Hilfe von SGML definiert. Dabei wurden die Freiheiten, die SGML bietet, zugunsten einer einfacheren maschinellen Verarbeitung reduziert (Beispielsweise ist in XML im Gegensatz zu SGML die Groß-Klein-Schreibung relevant.). Idee dieser Einschränkung war eine leichte Verarbeitung von XML-Dokumenten mit Hilfe von WebBrowsern „on-the-fly", also zur Betrachtungszeit. Damit soll XML irgendwann HTML als Dokumentenbeschreibungssprache für Webseiten ablösen.

Entwickelt wurde XML vom „World Wide Web Consortium" (http://www.w3c.org). Das W3C besteht aus über 500 Vertretern von Firmen und Organisationen die sich mit dem World Wide Web beschäftigen. Das Consortion hat dabei die Aufgabe, unabhängige, offene Standards zur effektiven und effizienten Nutzung des World Wide Web zu erstellen. Seit seiner Gründung im Jahre 1994 hat es unter anderen 5 verschiedene Standards für die Dokumentenbeschreibungssprache HTML entwickelt. Da bei HTML aber keine exakte Trennung zwischen „Layout und Content" möglich war und ist, hat man sich 1998 entschlossen ein völlig neues System namens XML zu entwickeln, das ausschließlich die Struktur eines Dokuments beschreibt [5]. Dieses System sollte es ermöglichen ein Dokument für die verschiedensten Ausgabemedien, wie Bildschirm, Drucker oder sogar Lautsprecher, ausgebbar zu machen.

1.2. Grundlagen

Ein XML-Dokument zeichnet sich dadurch aus, dass jeder Textabschnitt anhand seiner Bedeutung eindeutig klassifiziert ist. Dies geschieht durch sogenannte *„Tags"* (engl.: „Marke"), die den entsprechenden Textabschnitt umklammern. Dazu ein Beispiel:

Beispiel 1 *Wir betrachten den Fall dass jemand innerhalb seines Dokuments ein Datum als solches spezifizieren will. Da das Dokument später noch übersetzt werden soll, sollte das Datum nicht in Textform, sondern als reine Zahlen gespeichert werden. Um*

eine spätere Sortierung zu ermöglichen, werden die Zahlen zusätzlich in der Reihenfolge „Jahr-Monat-Tag" gespeichert, wodurch ein alphanumerisches (und damit maschinell einfach verarbeitbares) Sortierverfahren verwendet werden kann. Das dem 1. September 1975 entsprechende Datum würde somit in der XML-Datei wie folgt auftauchen:

```
<datum>19730901</datum>
```

Wie man erkennen kann, wurde als Tagname hier `<datum>` *genommen. Der zugehörige schießende Tag heißt dann dem entsprechend* `</datum>`.

Tags werden also durch „<>"-Klammern gekennzeichnet, schließende Tags zusätzlich mit einem vorangestellten „/" gekennzeichnet.

Zu klassifizierende Textabschitte lassen sich wiederum zu Tags zusammenfassen, um sie in einen Zusammenhang zu bringen. Ein weiteres Beispiel soll das erläutern.

Beispiel 2 *Das in Beispiel 1 typisierte Datum soll dazu verwendet werden eine Person mit ihrem Geburtsdatum zu versehen:*

```
<person>
  <name>Patrick Cernko</name>
  <datum>19750901</datum>
</person>
```

Aus einem `<name>`*-Tag und dem oben erwähnten* `<datum>`*-Tag wurde hier ein neuer Tag mit Namen* `<person>` *zusammengesetzt, der die beiden erstgenannten Tags enthält. Es entsteht eine sogenannte Baumstruktur.*

Ein XML-Dokument besteht, wie Beispiel 2 zeigt, immer aus einer vollständigen Baumstruktur. Dabei ist die Wurzel — also der Tag auf oberster Ebene — eine strukturelle Beschreibung des gesamten Dokumente, z. B. `<brief>` für einen Brief. Die Blätter des Baumes, also alles was zwischen zwei Tags im Dokument steht, die keine weiteren Tags mehr enthalten, bilden den textuellen Inhalt des Dokuments.

Um nicht für alle Bedeutungen der einzelnen Textabschnitte einen neuen Tag zu definieren, kann man sogenannte Attribute verwenden, die einen Tag — und damit seinen textuellen Inhalt — näher spezifizieren. Ein Attribut besteht immer aus einem eindeutigen Namen aus Textzeichen und einem Wert. Der Wert kann in Sonderfällen auch leer sein, muss aber auf jeden Fall angegeben werden. Beispiel 3 erklärt den Sachverhalt genauer.

Beispiel 3 *Wir wollen unsere* `<person>`*-Struktur um ihr Geschlecht erweitern. Das Geschlecht gehört logisch zum Namen, ist aber eigentlich nicht Teil des textuellen Inhalts des Namens.*

```
<person>
  <name geschlecht="männlich">Patrick Cernko</name>
  <datum>19750901</datum>
</person>
```

geschlecht *ist ein Attribut das sinnvollerweise nur die beiden Werte "männlich" und "weiblich" annehmen können sollte. Dem Attribut mit dem Namen* geschlecht *wird hier also der Wert "männlich" zugewiesen, da „Patrick" ein männlicher Vorname ist. Wie man solch einen eingeschränkten Wertebereich definiert, kann man unter Abschnitt 3.2 nachlesen.*

Es gibt auch Fälle, in denen man etwas ausdrücken will, ohne das ein textueller Inhalt vorhanden ist. Es soll also einen Sinn ergeben, ohne dass irgendein Text spezifiziert wird. Für solche Fälle gibt es sogenannte „leere Tags" (eng. *empty tags*). Im Gegensatz zu anderen Tags bestehen diese nicht aus einem Paar aus öffnendem und schließendem Tag, sondern haben beides in einem Tag vereint, wie in Beispiel 4 zu sehen.

Beispiel 4 *Wir wollen zusätzlich in unserer Struktur einen Verweis auf ein Bild der Person mit speichern.*

```
<person>
  <name geschlecht="männlich">Patrick Cernko</name>
  <bild adresse="http://www.ps.uni-sb.de/~errror/ich.jpg"/>
  <datum>19750901</datum>
</person>
```

Der Tag <**bild**> *enthält keinerlei textuelle Informationen sondern steht nur als Platzhalter für das Bild von Patrick Cernko, das unter der URL http://www.ps.uni-sb.de/~errror/ich.jpg hoffentlich zu finden ist*[1]*. Um den Tag als leeren Tag zu kennzeichnen, wird er von einem „/" abgeschlossen.*

1.3. Warum XML

Im Rahmen des Forschungsseminars „Elektronisches Publizieren am Beispiel von F. Nietzsche" standen wir anfangs vor dem Problem, welche Möglichkeiten wir hatten, die große Datenmenge in ein gut maschinell verarbeitbares Format überführen zu können.

Ursprünglich war geplant, statische HTML-Seiten zu erstellen und eine relationale Datenbank mit einzubinden, jedoch nur für dynamische Inhalte, wie etwa ein Forum. Nach angeregter Diskussion kamen alle Projektteilnehmer zu dem Schluss, dass bei Verwendung statischer HTML-Seiten die Dokumente wie Briefe und Porträts eigentlich

[1] Zum Zeitpunkt der Erstellung dieses Textes hatte der Autor unter der genannten Adresse seine Homepage mit einem aktuellen Photo von sich selbst.

nicht für die Nachwelt aufbereitet würden, sondern nur in einem neuen Layout entständen. Eine maschinelle Weiterverarbeitung, wie beispielsweise eine Klassifizierung oder Indexierung, wäre weiterhin nicht möglich.

So kamen wir schnell von der Idee wieder ab, statische Seiten zu erstellen. Statt dessen diskutierten wir zwei dynamische Ansätze:

- relationale Datenbanken
- Einzeldokumente mit XML

Für beide Ansätze gab es einige gute Gründe. Da im Teilnehmerkreis des Seminars wenig praktische Erfahrung mit XML vorlag, sahen wir XML als eine Herausforderung an und entschieden uns dafür. Allerdings stellten wir von vorneherein klar, dass auch Teile des Projekts eine relationale Datenbank verwenden sollten.

1.4. Stylesheets

Da in einer XML-Datei lediglich die Struktur eines Dokuments enthalten ist, benötigt man zur Darstellung — ob in einem WebBrowser oder zum Drucken — eine Vorlage, anhand derer die Struktur in ein konkretes Layout für das entsprechende Ausgabemedium umgeformt wird. Eine solche Vorlage bezeichnet man als XSL-Stylesheet oder einfach Stylesheet. XSL steht dabei für eXtensible Stylesheet Language und bezeichnet eine spezielle Beschreibungssprache die vom Word Wide Web Consortium speziell für den oben genannten Zweck, der Umformung einer XML-Datei in ein anderes Format, entwickelt wurde und wird[2] [1]. Umformung bedeutet dabei, dass das Ausgabeformat wiederum XML sein kann, was bedeutet, dass man XSL auch zum umkonvertieren veralteter XML-Formate in eine neue Version benutzen kann, etwa weil man die Struktur einer XML-Datei erweitern will.

XSL ist dabei wiederum in XML definiert, eine XSL-Datei ist also auch gleichzeitig eine XML-Datei. Allerdings ist das Format dieser Datei nicht benutzerdefinierbar, sondern schon fest vorgegeben.

Man muss sich klar machen, dass ein XSL-Stylesheet kein Programm ist, das eine gegebene XML-Datei in ein bestimmtes Zielformat umformt, sondern lediglich Anweisungen enthält, wie ein generisches XML-Konvertierungsprogramm eine gegebene XML-Datei umwandeln soll.

[2] Derzeit aktuell ist XSL Version 1.0, jedoch wird zur Zeit auch intensiv an der Weiterentwicklung gearbeitet.

2. Details von XML

In diesem Kapitel will ich auf das Format einer XML-Datei in allen notwendigen Details eingehen. Dazu wende ich die Projektdatei „Wie zu benützen" aus der FAQ, zu finden unter http://nietzsche.ps.uni-sb.de/faq/xsl/faq_00001.xml. In Anhang A finden Sie den kompletten Text des Dokuments. Bei meinen Erläuterungen will ich lediglich das XML-Format erklären. Jegliche Intentionen, warum irgendeine Strukturierung vorgenommen wurde, obliegen der Verantwortlichen des Teilprojekts FAQ, Frau Brigitte Jörg [13].

2.1. Deklarierung

Am Anfang jeder XML-Datei muss die Zeile

<?**xml version**=*"1.0"* **encoding**=*"iso−8859−1"*?>

stehen. Dies ist eine Sonderform von Tags, gekennzeichnet durch die „ ? " am Anfang und Ende des Tags. Dieser Tag spezifiziert, dass es sich bei der Datei um XML-Daten handelt.

Das Attribut version ist dabei zwingend erforderlich, da sonst keine einwandfreie maschinelle Verarbeitung gewährleistet werden könnte (Spätere XML-Versionen könnten ganz anders aufgebaut sein, was sich nicht ohne weiteres erkennen ließe.). Da es zur Zeit noch keine neuere Version als 1.0 der XML-Spezifikation gibt, muss hier immer „ 1.0 " als version -Wert stehen.

Das zweite Attribut encoding ist hingegen optional. Es spezifiziert den im Dokument verwendeten Zeichensatz. Falls es nicht angegeben ist, gilt, dass das Dokument den Zeichensatz „ASCII" verwendet. Der hier angegebene Zeichensatz *"iso−8859−1"* ist der westeuropäische Standardzeichensatz, der neben den reinen ASCII-Zeichen zusätzlich die deutschen Umlaute, die französischen Zeichen mit „Accents" und ähnliches enthält. Die Deklaration des Attributs encoding ermöglicht es uns, von nun an die deutschen Umlaute im Dokumenttext direkt verwenden zu können. In den Tagnamen dürfen jedoch weiterhin nur die Zeichen des normalen ASCII-Zeichensatz (was dem standardmäßigen amerikanischen Zeichenvorrat entspricht) verwendet werden.

Die zweite Zeile des Dokuments spezifiziert einen Verweis auf die DTD, die Document Type Definition, des Dokuments:

<**!DOCTYPE faq SYSTEM** *"faq.dtd"*>

Das Dokument wird hier so spezifiziert, dass der Wurzel-Tag — der Tag, der den kompletten Dokumenttext einrahmt — der Tag mit Namen <**faq**> ist. Ferner wird auf die DTD-Datei „faq.dtd" verwiesen, in der spezifiziert werden kann, nach welchen Regeln die im Dokument erlaubten Tags ineinander geschachtelt werden können und welche Attribute die einzelnen Tags haben können und müssen.

Anstelle des Verweises auf eine DTD-Datei ist es auch möglich die Struktur der XML-Datei direkt in der `<!DOCTYPE>`-Anweisung zu spezifizieren. Dies ist jedoch nur bei sehr keinen Strukturbeschreibungen sinnvoll. Grundsätzlich ist es überhaupt nicht notwendig, aber extrem sinnvoll, eine Strukturbeschreibung anzugeben, da nur so gewährleistet werden kann, dass das Dokument eine eindeutige Struktur hat, was eine maschinelle Weiterverarbeitung mit Hilfe von Stylesheets erst möglich macht. Mehr zu Document Type Definitions erfahren Sie in Abschnitt 3.

In der dritten Zeile der XML-Datei steht nun ein Verweis auf das Stylesheet, dass standardmäßig zum Anzeigen der Datei verwendet werden soll:

```
<?xml-stylesheet href="faq-html.xsl" type="text/xsl"?>
```

Die Stylesheetdatei „faq-html.xsl" wird hier als XSL-Stylesheet spezifiziert.

In der letzten Zeile der Deklaration steht noch eine Anweisung an die von uns verwendete Software *Cocoon*, was sie mit der Datei bei Abruf anfangen soll. Sie besagt lediglich, dass das oben spezifizierte Stylesheet auf die Datei angewendet werden soll. Näheres zu den verwendeten Programmen des Projektes findet Sie unter Abschnitt 5.

2.2. Der Inhalt

Nachdem die XML-Datei ausreichen deklariert ist, kann nun der eigentlich Dokumentinhalt folgen. Er muss mit dem in der `<!DOCTYPE>`-Anweisung spezifizierten Wurzel-Tag beginnen, wie im Beispieldokument mit

```
<faq id="faq_00001">
```

Der Tag eröffnet ein neues FAQ-Dokument und vergibt im die ID „faq_00001". Eine ID ist eine eindeutige Nummer, anhand derer man ein Objekt, wie hier unser Beispieldokument, eindeutig spezifizieren kann, sozusagen der interne, eindeutige Name des Dokuments.

Laut der in der Deklaration spezifizierten DTD enthält der Tag `<faq>` lediglich die beiden Tags `<head>` und `<body>`. Dem entsprechen folgt in unserem Beispiel nun der `<head>`-Tag.

Im `<head>`-Tag werden die Metadaten des Dokuments spezifiziert. Beim Tag `<autor>` taucht hier wiederum das Attribut `id="per_prossliner_johann_00"` auf. Im Gegensatz zum `<faq>`-Tag hat es hier aber die Bedeutung, dass auf ein anderes Dokument mit der angegebenen ID verwiesen wird. Dieses Dokument ist im Bereich „Personen" des Projekts zu finden und beschreibt eine Person. Anhand des Attributwertes kann man schon erkennen, dass es sich wahrscheinlich um Herrn „Johann Prosliner" handelt. Da das Attribut im Tag `<autor>` vorkommt, ist Herr Prosliner wohl hier als Autor des Dokuments ausgewiesen. Durch diesen Verweis ist es später beim Layout möglich weitere Daten über den Autor in das generierte Dokument zu integrieren, ohne die Daten explizit in diesem FAQ-Dokument aufzuführen.

Der eigentliche Dokumenttext steht, wie man sieht, im `<body>`-Tag. Er ist strukturiert in eine Frage (Tag `<frage>`) und die dazugehörige Antwort (Tag `<antwort>`) wobei die Antwort wiederum in mehrere Absätze untergliedert ist.

2.3. Baumstruktur

Die einzelnen Tags in einem XML-Dokument bestehen entweder immer aus einem öffnenden und einem dazugehörigen schießenden Tag — wie in diesem Beispieldokument — oder beides in einem Tag vereint — wie in Beispiel 4. Ein schließender Tag darf niemals erst nach einem anderen schließenden Tag stehen, der zu einem früher geöffneten Tag gehört. Dadurch wird die Erzeugung einer „Baumstruktur" für die XML-Datei gewährleistet. Beispiel 5 zeigt, wie ein Dokument, das sich nicht an die Gewährleistung der Baumstruktur hält, aussehen würde.

Beispiel 5 *Betrachten wir eine Datumsspezifikation wie in dem FAQ-Dokument „Wie zu benützen", die leider fehlerhaft ist, da sich der* `<datum>`*- und der* `<jahr>`*-Tag gegenseitig überlappen:*

```
<datum>
  <jahr>1975
  <monat>09</monat>
  <tag>01</tag>
  </datum>
</jahr>
...
```

Jeglicher Versuch, den Daten eine Bedeutung zu geben wird scheitern:

- *Soll das Datenfragment das Datum „im Jahre ‚1. September 1975'" bedeuten?*
- *Der Tag* `<jahr>` *darf aber doch laut DTD nur Textdaten und keinen Tag* `<monat>` *oder* `<tag>` *enthalten!*
- *Alles was unter . . . folgt steht auch noch im* `<datum>`*-Tag?*
- . . .

Die in Beispiel 5 gezeigte fehlerhafte Struktur bezeichnet man im englischen auch mit *nested Tags*.

2.4. Entities

In manchen Fällen ist es nicht möglich oder nicht erwünscht ein oder mehrere Zeichen immer wieder an der Verwendungsstelle einzugeben. Deshalb bietet XML die Möglichkeit der Definition sogenannter Entities im Deklarationsteil der XML-Datei. Ein Entity

ist nicht anderes als eine Abkürzung, auch Macro genannt, deren Auftreten bei der Verarbeitung der XML-Datei durch ihren Wert ersetzt wird.

Beispiel 6 *Bei der Erstellung von Briefdokumenten kommt die Floskel „mit freundlichen Grüßen" oft zur Verwendung. Um sich nun Tiparbeit zu sparen oder sich die Option offen zu halten, später die Floskel mit einem anderen Text zu verwenden, kann man sich nun ein Entity definieren:*

<**!ENTITY** mfg *"mit␣freundlichen␣Grüßen"*>

Nun kann man immer, wenn man die Floskel „mit freundlichen Grüßen" verwenden will, statt dessen das Entity &mfg; *verwenden.*

Ein Entity wird also mit dem Tag <**!ENTITY**> definiert. Bei der Verwendung leitet man den Entity mit den „Entity-Zeichen" „&" ein, gefolgt vom Namen des Entity und schließt immer mit einem Semikolon ab.

Es besteht auch die Möglichkeit den Inhalt eines Entity mit den Inhalt einer ganzen Datei zu belegen. Darauf will ich aber mangels der Verwendung in unserem Projekt nicht eingehen. In [10, Kapitel 9] findet der interessierte Leser einen umfassenden Überblick über Entities.

2.5. Leerzeichen

Im folgenden will ich auf die besondere Behandlung von Leerzeichen in einem XML-Dokument eingehen. Ich werde in diesem Abschnitt unter dem Begriff „Leerzeichen" eine Klasse von Zeichen zusammenfassen, die alle der Worttrennung dienen. Im englischen bezeichnet man diese Klasse, die ich meine als „wide spaces". Gemeint sind damit:

- das normale Leerzeichen, im folgenden oft mit „␣" explizit gekennzeichnet
- das Tabulatorzeichen, im folgenden oft mit „→" explizit gekennzeichnet
- der Zeilenwechsel

Grundsätzlich ist es in einem XML-Dokument egal, ob man zwischen zwei Worten ein oder mehrere Leerzeichen zur Worttrennung verwendet. Bei der Verarbeitung wird die Zahl standardmäßig auf ein Leerzeichen reduziert, sodass ein sauberes Layout ohne breite Wortabstände entsteht.

Beispiel 7 *Aufgrund der standardmäßigen Leerzeichenbehandlung werden die folgenden drei XML-Fragmente das gleiche Layout produzieren.*

1. *ganz ohne Leerzeichen*

   <**name**>Patrick␣Cernko</**name**>

2. *mit Zeilenwechsel*

```
<name>Patrick
Cernko</name>
```

3. *mit Tabulator und Zeilenwechsel*

```
<name>Patrick
    →Cernko</name>
```

In einigen Fällen, z. B. bei einem Gedichtvers [9] oder bei Computerprogrammen, will man jedoch, dass der Text mit allen Leerzeichen (also auch Zeilenwechseln) ins Layout gelangt. Dazu gibt es auch Techniken, auf die ich in Abschnitt 4.7.5 näher eingehen werde.

Da XML den Zeilenwechsel auch als Leerzeichen betrachtet, ist es ein verbreiteter Fehler bei Daten-Tags nach dem öffnenden Tag zur besseren Übersichtlichkeit einen Umbruch zu machen. Dies führt im Layout später zu seltsamen Effekten wie einer ungewollten Einrückung von Absatzanfängen oder gar Vergleichsfehlern. Beispiel 8 erläutert den Sachverhalt nochmals.

Beispiel 8 *Im XML-Fragment*

```
<name>
Patrick␣Cernko
</name>
```

ist der Inhalt des `<name>`*-Tags nicht wie erwartet „Patrick Cernko", sondern „␣Patrick Cernko␣".*

Im Gegensatz zum obigen Sachverhalt werden allerdings Leerzeichen innerhalb von Tags, die nur andere Tags enthalten, und solche zwischen den Attributen eines Tags *nicht* mit ins Layout übernommen. Es ist deshalb erlaubt — und auch zu empfehlen — die Baumstruktur der XML-Datei durch Umbrechen und Einrücken´, wie in den bisher gezeigten Beispielen, auch wiederzugeben.

2.6. Kommentare

Unter Umständen will man manchmal auch Text in einem XML-Dokument haben, der bei der Verarbeitung ignoriert wird. Hierfür gibt es den speziellen Tag

```
<!-- Kommentar -->
```

der den Text „Kommentar" als nicht zu verarbeitenden Kommentar anzeigt. Ein Kommentar beginnt immer mit `<!--` und endet mit `-->`.

Bei der Verwendung von Kommentaren sind die folgenden Regeln zu beachten [10, Seiten 135–137]:

Zeichen	Entity
&	&amp;
<	&lt;
>	&gt;
’	&apos;
”	&quot;

Tabelle 1: Die fünf Standard-Entities

1. Kommentare dürfen erst nach der Deklaration der Datei als XML-Dokument stehen.
2. Kommentare dürfen nicht innerhalb des Tagnamens und seiner Attribute stehen.
3. Kommentare dürfen vollständige Tags (aus öffnendem und schließendem Tage bzw. leere Tags) umschließen und klammern diese dann aus.
4. Der Doppelbindestrich darf nur als Teil des Taganfangs bzw. -endes auftreten.

Punkt 3 zeigt, dass man mit Hilfe von Kommentaren Teile eines Dokuments von der Verarbeitung ausnehmen kann. Dabei sind jedoch wegen Punkt 4 die Einschränkungen zu beachten, dass innerhalb des entsprechenden Teils keine weiteren Kommentare liegen.

2.7. Sonderzeichen

Zum Schluss muss ich noch auf das wohl problematischste Thema bei der Erstellung von XML-Dateien eingehen, nämlich, dass bestimmte Zeichen wegen verschiedenen Gründen nicht direkt im Text vorkommen können. Zum einen ist es nicht möglich die fünf Zeichen „&”, „<”, „>”, „’” und „”” direkt einzugeben. die Begründung dafür ist wohl recht offensichtlich: Diese Zeichen werden zum Kennzeichnen von Tags, Attributen und Entities verwendet. Um diese fünf Zeichen im Text darzustellen, muss man spezielle Entities benutzen. Tabelle 1 zeigt die fünf oben genannten Zeichen und ihre zugehörigen Entities.

Die zweite Gruppe von nicht direkt verwendbaren Zeichen, sind die Zeichen, die nicht im normalen ASCII-Zeichensatz enthalten sind, z. B. die deutschen Umlaute. Hat man jedoch bei der Deklarierung des XML-Formates (vgl. Abschnitt 2.1) einen passenden Zeichensatz für die Datei spezifiziert, so kann man alle Zeichen dieses Zeichensatzes verwenden. Es ist aber auch möglich, statt der eigentlichen Zeichen Entities zu benützen, die auf die entsprechenden Zeichen per Zeichencode verweisen.

3. Document Type Definitions

XML läßt dem Benutzer die Freiheit, die Struktur seiner Dateien selbst zu beschreiben. Zum formalen Beschreiben der Struktur gibt es sogenannte „Document Type Definitions" (engl. für „Dokumententypdefinitionen"). Sie gestatten es dem Benutzer die Strukturierung formal eindeutig zu definieren. Im folgenden will ich das Format und die Möglichkeiten solcher DTDs erläutern. Dazu verwende ich wiederum eine Beispieldatei, nämlich die DTD für FAQs des Projekts „Elektronisches Publizieren". Sie finden diese Datei unter http://nietzsche.ps.uni-sb.de/faq/xsl/faq.dtd und in Anhang B.

3.1. Elemente

Unter dem Begriff „Elemente" versteht man in einer DTD die Tags des beschriebenen XML-Formats.

Beispiel 9 *In der fünften Zeile der Beispiel-DTD wird der Tag* <**faq**> *deklariert:*

<**!ELEMENT faq** (**head**,**body**,**anmerkung**∗)>

Elemente werden also durch den Tag <**!ELEMENT**> deklariert. Er muss den Namen des zu definierenden Elements ,hier der Tag <**faq**>, enthalten sowie eine Spezifikation, welches Element innerhalb vorkommen darf. Neben den sonstigen in der DTD definierten Elementen können dabei die beiden Spezialelemente **EMPTY** für kein Element (also einen „leeren Tag") und **#PCDATA** für textuellen Inhalt angegeben werden.

Mit Hilfe von Klammern kann man dabei mehrere Elemente zu einem Element zusammenfassen. Dabei können die Elemente als Sequenz oder als Konjunktion angegeben werden.

Bei einer Sequenz werden die Elemente durch Kommata getrennt. Alle in der Sequenz angegebenen Elemente müssen dabei in der angegebenen Reihenfolge in einer dieser Spezifikation entsprechenden XML-Datei auftauchen.

Im Gegensatz zur Sequenz darf bei der Konjunktion immer nur eines der angegebenen Elemente in der dieser Spezifikation entsprechenden XML-Datei auftauchen. Die Elemente einer Konjunktion werden dabei mit dem Zeichen „|" voneinander getrennt.

Es besteht auch die Möglichkeit, die Auftretungshäufigkeit eines Elements zu quantifizieren. Dies geschieht durch das Anhängen eines der drei Zeichen „?", „*" und „+", wobei die Zeichen folgende Auswirkung auf das vorangehende Element haben:

- Nach „?" darf das Element ein- oder keinmal auftreten. Es ist also optional.
- Nach „*" darf das Element beliebig oft auftreten. Es ist also optional mit beliebiger Anzahl.

Type	Bedeutung
CDATA	Textdaten
Aufzählung	Eine umklammerte Liste von Werten, getrennt durch das Zeichen „|"
ID	Ein eindeutige ID
IDREF	Eine ID eines anderen Tag-Attributs

Tabelle 2: Attributtypen

- Nach „+" muss das Element mindestens einmal auftreten. Es ist also verpflichtend, aber mit beliebiger Anzahl

Im obigen Beispiel wird also spezifiziert, dass innerhalb des **<faq>**-Tags zunächst der **<head>**-Tag und anschließend der Tag mit Namen **<body>** kommen muss. Abschießend können noch beliebig viele **<anmerkung>** stehen.

3.2. Attribute

Attribute werden durch den Tag **<!ATTLIST>** spezifiziert. Ein gutes Beispiel dazu sehen Sie in Zeile 7–9 der Beispieldatei:

```
<!ATTLIST design id IDREF                                    #REQUIRED
                  typ  (erstellung|modifikation|verschlagwortung) #REQUIRED>
```

Hier werden dem Tag **<design>** zwei Attribute zugewiesen. Dazu wird zunächst der Tag spezifiziert, dem die Attribute zugewiesen werden. Anschließend kann man eine Liste von Attributnamen angeben, wobei jedem Attributnamen der Typ und der Standardwert folgen müssen.

Tabelle 2 zeigt einige Attributtypen. Mit Hilfe des **ID**-Typs ist es möglich, Tags eindeutig zu benennen um dann mit Hilfe des **IDREF**-Typs auf diese zu verweisen. Neben den aufgeführten Attributtypen gibt es noch andere, die jedoch in unserem Projekt keine Verwendung fanden. Eine vollständige Liste der Attributtypen und deren Bedeutung findet der interessierte Leser in [10, Kapitel 10].

Unter einem Standardwert eines Attributs versteht den Wert, den das Attribut eines Tags hat, wenn es nicht im Tag aufgeführt wird, d.h. der Tag das Attribut nicht mit einem Wert belegt. Der Standardwert wird mit Doppelanführungszeichen angegeben. Dabei kann ein vorangestelltes „ **#FIXED** " den Wert des Attributs als unveränderbar kennzeichnen. Eine Belegung des Attributs in einer XML-Datei mit einem anderen, als dem vorgegebenen Wert führt dann zu einem fehlerhaften XML-Format der Datei.

Läßt sich für ein Attribut kein Standardwert spezifizieren, so kann man eines der beiden Schlüsselwörter **#REQUIRED** oder **#IMPLIED** stattdessen einsetzen. **#REQUIRED** hat dabei die Bedeutung, dass es zwar keinen Standardwert gibt, aber

der Dokumentenersteller zwingend einen Wert für dieses Attribut angeben muss. Beim Schlüsselwort **#IMPLIED** gilt ebenfalls, dass es keinen Standardwert gibt, jedoch ist die Belegung des Attributs mit einem Wert nicht zwingend.

3.3. Entities

Wie auch in XML (vgl. Abschnitt 2.4) gibt es auch innerhalb einer DTD die Möglichkeit, wiederkehrende Textteile durch Platzhalter, die man einmalig definiert, zu ersetzen. Dazu dient, wie in XML, der Tag **<!ENTITY>**, jedoch mit einer leicht veränderten Syntax.

In der Beispiel-DTD findet sich in Zeile drei eine Entity-Definition:

<**!ENTITY** % block *"p|erweitert|sprache|rede|extrakt_ref|beleg_ref|komm_ref| brief_ref| ewerk_ref| bib_ref | link_ref | quelle_ref | per_ref | ort_ref | faq_ref | extern_ref"*>

Das DTD-Entity %block; wird hier mit einer Art langen Konjunktion belegt und im folgenden in der Beispiel-DTD häufig verwendet. Entities für Document Type Definitions müssen also durch ein vorangestelltes „% “ als solche gekennzeichnet werden.

Wie auch in XML, können auch bei DTD-Entities statt eines Textes eine Datei angegeben werden, wobei das Entity dann mit dem Inhalt der Datei belegt wird. Unter [6] habe ich diese Tatsache beispielhaft für den Versuch genutzt, DTD-Teile, die Strukturen beschreiben, die nicht Teil einer speziellen Dokumentenstruktur sind, in einzelne Dateien auszugliedern.

4. Die eXtensible Stylesheet Language

4.1. Trennung von Layout und Content

In einer XML-Datei wird lediglich die Struktur eines Dokuments beschrieben. Um es adäquat darzustellen, benötigt man also eine Beschreibung, wie dann die gegebene Struktur dem Benutzer dargestellt werden soll. Dazu verwendet man bei XML Stylesheets.

Stylesheets sind Angaben, wie eine XML-Datei formatiert werden soll. Wer sich mit aktuellem WebDesign beschäftigt, kennt bereits „Cascading Stylesheets" (CSS), eine Sprache zur Formatierung von HTML-Tags [3]. CSS ermöglichen es dem WebDesigner in gewissen Grenzen, den eigentlich zur Textformatierung und allgemeinen Strukturierung gedachten HTML-Tags eine genauere Bedeutung und damit einhergehend ein eigenes Layout zu geben. Jedoch ist der Autor dabei recht eingeschränkt, besonders durch die Tatsache, dass er die Reihenfolge seiner Strukturelemente kaum beeinflussen kann.

Aufgrund der Mängel in CSS haben sich die Entwickler von XML dazu entschlossen, zusätzlich zu den auch in XML verwendbaren Cascading Stylesheets, die eXtensible Stylesheet Language zu entwickeln. Diese erlaubt nun die vollständige Kontrolle über den aus den Daten einer XML-Datei generierten Inhalt und das Layout. Neben der kompletten Umstrukturierbarkeit und der Möglichkeit, jeden beliebigen Teil des Ausgangsdokuments wegzulassen, ist es auch dem Benutzer offengestellt, Daten aus andern Datenquellen mit in die produzierte Ausgabe zu übernehmen. Außer anderen XML-Dateien können hier auch Datenbanken, Grafikdateien oder Verzeichnisdienste Verwendung finden.

In einer XML-Umgebung besteht also eine strickte Trennung zwischen den Daten (in den XML-Dateien) und der Layoutbeschreibungen (in den XSL-Stylesheets). Dadurch wird eine wesentlich bessere Datenintegrität gewährleistet, da der schöpferische Prozess der Erstellung der XML-Dokumente vom gestalterischen Prozess zur Erzeugung eines Layouts für die Daten getrennt ist. Dieses Prinzip wird in der Fachliteratur immer wieder als die „Trennung von Layout und Content" besprochen. Layout bezeichnet dabei die Gestaltung, während Content die Daten selbst meint.

Es ist auch möglich, dass man für ein XML-Format mehrere Stylesheets für verschiedene Ausgabemedien bereithält. Beispielsweise ist es in der derzeitigen WebBrowser-Landschaft nahezu unmöglich mit dem gleichen HTML-Layout einigermaßen gleiche Darstellungen auf den verschiedenen Systemen zu erzielen, wenn das Layout nur hinreichend komplex ist oder eingebettete Scriptsprachen verwendet werden sollen [16]. Eine andere Anwendung für verschiedene Stylesheets zu einem XML-Format ist die gleichzeitige Bereitstellung der Daten im HTML-Format (zum Browsen) und als PDF-Datei (zum Ausdruck).

Bei der Erklärung der Möglichkeiten von XSL habe ich auch zunächst überlegt, eine Beispieldatei als Referenz anzugeben. Es gab aber letztlich verschiedene Gründe dagegen. Zum einen sind die in unserem Projekt „Elektronisches Publizieren" entwickelten

XSL-Stylesheets schon relativ komplex, zum anderen werden die Möglichkeiten in keiner Datei umfassend ausgenutzt.

4.2. Der Anfang

Grundlage bei der Umformung einer XML-Datei mit einem Stylesheet ist die maschinell aufbereitete Struktur der zugrundeliegenden Datei. Sie liegt in Form der in Abschnitt 2.3 beschriebenen Baumstruktur vor. Die Verarbeitung erfolgt dann weitestgehend entlang dieser Struktur, angefangen beim Wurzel-Tag bis hin zu den „Blättern", den Tags mit dem textuellen Inhalt. Dadurch ergibt sich eine Gestaltung „vom Groben ins Feinere".

Beispiel 10 verdeutlicht die beschriebene Verarbeitungstrategie.

Beispiel 10 *Wir betrachten uns eine imaginäre XML-Datei, die innerhalb eines Wurzel-Tags* `<adressen>` *eine Liste von Personen, klassifiziert mit dem Tag* `<person>`*, enthält. Jede Person soll einen Namen (Tag* `<name>`*), eine Straße (Tag* `<strasse>`*) und einen Wohnort (Tag* `<ort>`*) vorweisen*[3]*:*

```
<adressen>
  <person>
    <name>Patrick Cernko</name>
    <strasse>Mainstraße 2</strasse>
    <ort>66125 Saarbrücken − Dudweiler</ort>
  </person>
  <person>
    <name>Dagobert Duck</name>
    <strasse>Zum Geldspeicher 1</strasse>
    <ort>1234 Entenhausen</ort>
  </person>
  <person>
    <name>Klaus Mustermann</name>
    <strasse>Musterstraße 7</strasse>
    <ort>47110 Musterstadt</ort>
  </person>
  ...
</adressen>
```

Die gegebenen Daten sollen nun als Tabelle auf einer Webseite erscheinen. Bei der Verarbeitung muss das Transformationsprogramm angewiesen werden, beim Auftreten eines `<adressen>`*-Tags ein Grundgerüst eine Tabelle zu generieren, etwa:*

[3] Der Einfachheit wegen, verzichte ich in dem Beispiel darauf, Hausnummer und Postleitzahl explizit zu typen. Dem aufmerksamen Leser wird klar sein, dass eine solche Struktur nur einen beschränkten Wert bei der Ordnung der Daten nach Postleitzahlen haben kann.

```
<table summary="Meine Adressen">
<tr>
  <td>Patrick Cernko</td>
  <td>Mainstraße 2</td>
  <td>66125 Saarbrücken − Dudweiler</td>
</tr>
<tr>
  <td>Dagobert Duck</td>
  <td>Talerweg 1</td>
  <td>1234 Entenhausen</td>
</tr>
<tr>
  <td>Klaus Mustermann</td>
  <td>Musterstraße 7</td>
  <td>47110 Musterstadt</td>
</tr>
...
</table>
```

Listing 1: Eine XML-Adressdatenbank gestaltet als HTML-Tabelle

```
<table summary="Meine Adressen">
  Verarbeitung der Unter-Tags
</table>
```

wobei das Programm an der Stelle „Verarbeitung der Unter-Tags" angewiesen werden muss, die Unter-Tags von `<adressen>` *zu verarbeiten.*

Als Unter-Tags kommt hier nur `<person>` *vor. Dementsprechend muss dem Programm für jedes Auftreten dieses Tags eine neue Tabellenzeile generieren, in der es für die Unter-Tags* `<name>`, `<strasse>` *und* `<ort>` *je eine Zelle anlegt und dort jeweils einen der Unter-Tag-Inhalte einbringt:*

```
<tr>
  <td>Inhalt von <name></td>
  <td>Inhalt von <strasse></td>
  <td>Inhalt von <ort></td>
</tr>
```

Listing 1 zeigt das Ergebnis der Anwendung der Umformungen für unser Beispiel.

Die Verarbeitung in Beispiel 10 ist nur eine Möglichkeit zur Darstellung der Daten. Denkbar wäre auch eine zweistufige Aufzählung, wie Listing 2 sie exemplarisch zeigt.

```
<ol>
  <li>Patrick Cernko
    <ul>
      <li><b>Straße:</b> Mainstraße 2</li>
      <li><b>Ort:</b> 66125 Saarbrücken − Dudweiler</li>
    </ul>
  </li>
  <li>Dagobert Duck
    <ul>
      <li><b>Straße:</b> Talerbr&uuml;cke 1</li>
      <li><b>Ort:</b> 1234 Entenhausen</li>
    </ul>
  </li>
  <li>Klaus Mustermann
    <ul>
      <li><b>Straße:</b> Musterstraße 7</li>
      <li><b>Ort:</b> 47110 Musterstadt</li>
    </ul>
  </li>
...
</ol>
```

Listing 2: Eine XML-Adressdatenbank gestaltet als HTML-Aufzählung

Die zur Darstellung als solche Aufzählung nötigen Verarbeitungsanweisungen seien hier nicht erwähnt, der interessierte Leser kann sie zur Übung selbst entwickeln.

4.3. Vorlagen

Vorlagen (im englischen „templates") sind das wichtigste Werkzeug, bei der Verwendung von Stylesheets. Eine Vorlage kann bestimmen wie der Inhalt eines oder auch verschiedener Tags zur Ausgabe verarbeitet werden soll. Dabei unterscheidet man zunächst zwischen Mustervorlagen und Namensvorlagen.

Mustervorlagen werden eingeleitet durch den umrahmenden Tag

```
<xsl:template match="Muster">
```

Sie werden nur auf Tags der Eingabe angewendet, die der Spezifikation „*Muster*" entsprechen. Durch die Verwendung von Mustern lässt sich eine solche Vorlage für viele verschiedene Tags anwenden. Die Muster werden dabei mit Hilfe der XML Linking Language *XLink* angegeben [7]. Auf die Details der Möglichkeiten von XLink will ich in dieser Arbeit nicht ausführlich eingehen, da es sicherlich den Rahmen sprengen würde.

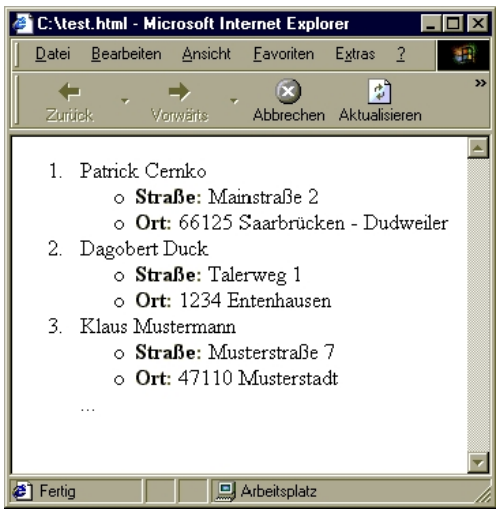

Abbildung 1: Eine als HTML-Aufzählung gestaltete Addressdatenbank im Internet Explorer™

In Abschnitt 4.6 finden Sie ein paar für die Verwendung im Projekt „Elektronisches Publizieren" notwendige Grundlagen.

Im Gegensatz dazu werden bei Namensvorlagen keine Muster angegeben, die die Art der Tags, auf die sie angewendet werden können, einschränken. Statt dessen werden die Vorlagen mit einem Namen versehen, anhand dessen man sie explizit aufrufen muss. Namensvorlagen werden umspannt von dem Tag

```
<xsl:template name="Name der Vorlage">
```

Mustervorlagen eignen sich also gut, um bestimmte Tags der Eingabe in eine Ausgabe umzuwandeln; mit Namensvorlagen kann man größere Layout-Anweisungsblöcke kapseln, die sich nicht einem bestimmten Eingabe-Tag zuweisen lassen. Beispiel 11 erläutert den Sachverhalt näher.

Beispiel 11 *Wir betrachten uns nochmal die Problematik aus Beispiel 10: Wir wollen unsere Adressdaten nach HTML konvertieren. Betrachten wir zunächst, wie wir einen einzelnen Eintrag zu konvertieren haben. Wir benötigen eine Vorlage für den Tag* `<person>` *:*

```
<xsl:template match="person">
  <tr>
    <td><xsl:apply-templates select="name"/></td>
    <td><xsl:apply-templates select="strasse"/></td>
```

```
    <td><xsl:apply-templates select="ort"/></td>
  </tr>
</xsl:template>
```

Dem aufmerksamen Leser wird die Ähnlichkeit der Vorlage mit dem entsprechenden Abschnitt in Beispiel 10 auffallen. Anstatt der Pseudo-Anweisungen „Inhalt von <...>" stehen nun die XSL-Tags `<xsl:apply-templates select="name"/>`*, usw. Diese veranlassen das Konvertierungsprogramm, die Ausgabe der Vorlagen für* `<name>`*, usw. an den entspechenden Stellen einzufügen. Da für diese Tags keine speziellen Vorlagen angegeben wurden, wird eine Art „Standardvorlage" genommen: Es wird einfach der Inhalt des Tags ohne seine eventuell vorhandenen Attribute ausgegeben.*

Um nun eine vollständige Tabelle zu erhalten wollen wir auch eine Vorlage für den `<adressen>`*-Tag erstellen. Diese soll neben der umspannenden Tabellendefinition zusätzlich einen Seitenkopf generieren:*

```
<xsl:template match="adressen">
  <xsl:call-template name="seitenkopf"/>

  <table summary="Meine␣Adressen">
    <xsl:apply-templates>
  </table>
</xsl:template>
```

Neben der schon aus Beispiel 10 bekannten Tabellendefinition wird hier zunächst in Zeile 2 eine Namensvorlage mit dem Namen „seitenkopf" aufgerufen. Im Gegensatz zu Mustervorlagen muss eine aufgerufene Namensvorlage auch irgendwo im Stylesheet angegeben definiert werden. Bei unserem Beispiel könnte die Definition so aussehen:

```
<xsl:template name="seitenkopf">
  <h1>Eine Beispieladressliste</h1>
</xsl:template>
```

Man kann also Mustervorlagen für sämtliche Unter-Tags des derzeit abzuarbeitenden Eingabe-Tags aufrufen (mit Hilfe des Tags `<xsl:apply-templates/>`) oder durch Angabe des Attributs **select**, das wiederum ein XLink-Muster annimmt, die Mustervorlagenverarbeitung auf bestimmte Vorlagen eingrenzen[4].

Will man eine Namensvorlage aufrufen, so geschieht dies durch den Tag

```
<xsl:call-template name="Name der Vorlage"/>
```

[4] Es ist auch möglich die Liste der Tags, auf die Vorlagen angewendet werden sollen, nach verschiedenen Gesichtspunkten zu sortieren. Da diese Funktionalität in unserem Projekt jedoch nicht benötigt wurde, will ich nicht näher darauf eingehen.

Neben den genannten Möglichkeiten mit Vorlagen kommt es oft vor, dass man den gleichen Tag — oder auf Tags des gleichen Musters — verschieden formatieren möchte, etwa im HTML-`<head>` als `<title>`-Tag und im Text des zu erstellenden Dokuments als Hauptüberschrift (etwa mit einem `<h1>`-Tag).

Zu diesem Zweck kann man Vorlagen auch in einem spezifischen selbstdefinierten „Modus" anwenden. Dazu gibt man beim Aufruf und bei der Definition der Vorlage das optionale zusätzliche Attribut

```
<xsl:template mode="Modusname der Vorlage"/>
```

Durch die Verwendung von Modi ist es möglich, für ein einziges Muster viele verschiedene Formatierungen für die unterschiedlichsten Anwendungsfälle zu definieren.

Durch die genannten weitreichenden Möglichkeiten der XSL-Vorlagen kann man bereits sehr komplexe Stylesheets erzeugen. In einigen Fällen reichen aber diese Möglichkeiten noch nicht aus. Deshalbe werde ich in den folgenden Abschnitten noch auf einige andere Möglichkeiten mit XSL-Stylesheets eingehen.

4.4. Bedingte Verarbeitung

Eines der wesentlichsten Merkmal von XSL, die nicht durch die ausschließliche Verwendung von Vorlagen erreicht werden kann ist bedingte Verarbeitung. Man will oft abhängig von der Eingabe — ob aus der Eingabedatei oder zusätzlichen Eingabedaten wie Datenbanken oder andere XML-Dateien — ein unterschiedliches Layout ausgeben. Dazu benötigt man Mittel, um Bedingungen zu prüfen und um dann abhängig vom Wahrheitswert der Bedingung etwas auszugeben.

XSL bietet für die bedingte Verarbeitung zwei verschieden Konstrukte:

1. einseitige Bedingungsverarbeitung, die nur bei Bewahrheitung der Bedingung zu einer Ausgabe oder Vearbeitung des betreffenden Stylesheet-Teils führen

2. Fallunterscheidung, die bei Bewahrheitung eines Falls ausgewertet werden

Als Bedingungen können hier Vergleiche und „Boolsche Funktionen" vorkommen. Vergleiche sind von der Form „ist gleich", „ist ungleich", „ist kleiner als", „ist größer als", „ist kleiner-gleich" oder „ist größer-gleich", wobei die letzteren vier Vergleiche natürlich nur bei Zahlenwerten Sinn machen, die ersten beiden Vergleichstypen jedoch auch für Zeichenketten und ganze Tags möglich sind. Bei dem Vergleich zweier Tags wird dann der Inhalt der Tags ohne deren Attribute verglichen.

Die in XSL standardmäßig enthaltenen Boolschen Funktionen sind unter anderem Negation (`not(bedingung)`), Disjunktion und Konjunktion zweier Bedingungen (`bed1 and bed2` und `bed1 or bed2`), sowie die Funktionen für Wahrheit und Falschheit (`true()` und `false()`).

Neben den Boolschen Funktionen gibt es in XSL auch standardmäßig viele andere Funktionen, auf die ich in Abschnitt 4.7.4 weiter eingehen werde.

4.4.1. Einseitige Bedingungsverarbeitung

Will man einen Abschnitt nur dann ausgeben, wenn eine Bedingung erfüllt ist, so verwendet man das XSL-Konstrukt

```
<xsl:if test="Bedingung">
  bedingter Abschnitt
</xsl:if>
```

Zur Verdeutlichung führen wir Beispiel 11 fort und erweitern es in Beispiel 12 um die Möglichkeit, die Straßenangabe wegzulassen.

Beispiel 12 *Manchmal kann man bei einer Addressdatenbank wie in Beispiel 10 keine Angaben über Straße und/oder Wohnort machen. In einem solchen Fall will man das Feld einfach leer lassen. Bei der Darstellung als HTML-Dokument kann es zu Problemen führen, keinerlei Inhalt in einer Tabellenzelle zu haben*[5]*, oder man will dann einfach etwas wie „keine Angaben" stehen haben.*

Wir wollen nun die in Beispiel 11 beschriebenen Vorlagen um eine Überprüfung der Straße erweitern, sodass bei einem nicht vorhanden `<strasse>` *-Tag der Text „keine Angaben" im auszugebenen HTML-Dokument steht. Dazu modifizieren wir die Vorlage für den Tag* `<person>` *wie folgt:*

```
<xsl:template match="person">
  <tr>
    <td><xsl:apply-templates select="name"/></td>
    <td>
      <xsl:if test="strasse=""">keine Angaben</xsl:if>
      <xsl:apply-templates select="strasse"/>
    </td>
    <td><xsl:apply-templates select="ort"/></td>
  </tr>
</xsl:template>
```

Wie man sieht, überprüfen wir, ob der Inhalt des Tags `<strasse>` *gleich der leeren Zeichenkette ist. Falls die Überprüfung zu einem wahren Ergebnis führt, wird der Text „keine Angaben" an der entsprechenden Stelle ins Ausgabedokument geschrieben. Der darauffolgende Aufruf der Vorlage für den Tag* `<strasse>` *liefert dann entsprechend keinen Text in der Ausgabe. Liefert die Überprüfung eine falsches Ergebnis, so wird der Text „keine Angaben" nicht ins Ausgabe-Dokument geschrieben, der anschließende Vorlagenaufruf liefert den Wert des Tags* `<strasse>` *(in dem hoffentlich die richtige Straße und Hausnummer drin steht!).*

[5] Der Netscape Navigator™ beispielsweise vereinfacht Tabellen mit leeren Zellen derart, dass er die leeren Zellen überhaupt nicht zeichnet, also auch keinen Zellenrahmen und -hintergrund.

4.4.2. Auswahlbedingungen

Neben der in Abschnitt 4.4.1 vorgestellten einseitigen Bedingungsverarbeitung ist es oft viel sinnvoller, die bedingte Verarbeitung durch eine Fallunterscheidung zu realisieren. Dazu sieht XSL die folgende Konstruktion vor:

```
<xsl:choose>
  <xsl:when test="Bedingung 1">
    bedingter Abschnitt für Bedingung 1
  </xsl:when>
  <xsl:when test="Bedingung 2">
    bedingter Abschnitt für Bedingung 2
  </xsl:when>
  ...
  <xsl:otherwise>
    bedingter Abschnitt, falls keine Bedingung erfüllt ist
  </xsl:otherwise>
</xsl:choose>
```

Wie man sieht, kann man bei dieser Form der bedingten Verarbeitung beliebig viele Verarbeitungsfälle mit jeweiliger Bedingung und höchstens einen Fall haben, der zur Verarbeitung kommt, wenn keine der Bedingungen erfüllt ist. Wichtig ist hierbei, dass die einzelnen Bedingungen der Fälle sich nicht gegenseitig ausschließen müssen, d.h. es ist durchaus möglich, dass mehrere Fälle verarbeitet werden, dann natürlich in der Reihenfolge der Angabe.

4.5. Platzhalter

Platzhalter sind XSL-Bausteine, die mit einem Wert belegt werden können. Bei der Verarbeitung werden alle verwendenden Auftretungen dann durch ihren Wert ersetzt. Man unterscheidet in XSL zwei verschiedene Arten von Platzhaltern:

- Variablen, die im XSL-Stylesheet mit einem Wert belegt werden können
- Parameter, die im Stylesheet deklariert und von außen mit einem Wert belegt werden

4.5.1. Variablen

Mit Hilfe von Variablen ist es möglich Textbausteine im Vorfeld der eigentlichen Verarbeitung zu erstellen. Besonders sinnvoll ist dies bei Textteilen, die sehr kompliziert aufgebaut sind, häufiger verwendet werden und bei der Zusammensetzung von Bedingungsteilen für die bedingte Verarbeitung (vgl. Abschnitt 4.4).

Definiert werden Variablen mit Hilfe von:

```
<xsl:variable name="Name der Variablen">
  Wert der Variablen
</xsl:variable>
```

oder durch den leeren Tag

```
<xsl:variable name="Name der Variablen" select="Wert der Variablen"/>
```

Die erste Variante kommt meist zur Verwendung, wenn man weitere XSL-Konstrukte, wie beispielsweise einen Vorlagenaufruf, benutzen will. Bei der zweiten Variante kann man hingegen besser Werte anderer Variablen, Tags und XSL-Funktionen verwenden.

Um eine definierte Variable irgendwo zu verwenden, benutzt man den Variablennamen mit einem vorangestellten „$". Zusätzlich kann man den Variablennamen noch mit geschweiften Klammern „{}" umklammern, um ihn eindeutig von den folgenden Zeichen zu trennen.

4.5.2. Parameter

Will man bei der Verarbeitung Daten von außen mit einfließen lassen, so benutzt man Parameter. Im Gegensatz zu Variablen, die auch innerhalb von Vorlagen definiert werden können, muss ein Parameter außerhalb der Vorlagen deklariert werden. Dies geschieht mit:

```
<xsl:param name="Name des Parameters"/>
```

Es ist abhängig von der verwendeten Verarbeitungs-Software, wie XSL-Parameter zur Verarbeitung spezifiziert werden. Näheres dazu finden Sie im Abschnitt 5.2.

Ein deklarierter Parameter wird im Stylesheet genauso verwendet wie eine Variable.

Beispiel 13 *Bei den FAQ-Dokumenten des Projekts „Elektronisches Publizieren" setzten wir sogenannte „Erweiterungen" getypt durch den Tag* `<erweitert>` *ein, die annotierende Textabschitte an der aktuellen Position ermöglichten. Diese sollten im initialen Ausgabedokument zunächst nicht erscheinen, sondern nur als Verweis gekennzeichnet werden. Dieser Verweis erzeugte beim Aufruf ein neues Ausgabedokument aus der ursprünglichen FAQ-Datei, das nur noch diese Erweiterung anzeigte. Dazu verwendeten wir den Parameter*

```
<xsl:param name="erweitert"/>
```

der mit einer Zahl belegt wurde, falls nur eine „Erweiterung" anzuzeigen war. Diese Zahl entspricht der Nummer des entsprechenden `<erweitert>` *-Tags bei deren Durchnummerierung im FAQ-Dokument.*

In der Vorlage zum Wurzel-Tag der FAQ-Dokumente benutzten wir dann eine Fallunterscheidung, um zwischen den beiden Layouts (initiales Dokument oder „Erweiterung") zu entscheiden:

```
<xsl:template match="/faq">
  <xsl:choose>
    <xsl:when test="${erweitert}!="">
      <xsl:call-template name="erweitertlayout"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="normallayout"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

In der Namensvorlage mit Namen „erweitertlayout" wird das Dokument dann so weiterverarbeitet, dass die „Erweiterung" mit der Nummer, die gleich der Zahl in `$erweitert` *ist, angezeigt wird.*

4.6. XLink

Wie schon in Abschnitt 4.3 erwähnt, kommt bei der Spezifikation von Tags des Ausgangsdokuments in Stylesheets die Sprache XLink zum Einsatz. Im folgenden will ich kurz auf einige Möglichkeiten dieser Sprache zu sprechen kommen.

Wie schon an den bisherigen Beispiele zu sehen war, werden einfache Tags in Vorlagen u.ä. einfach durch die Angabe des Tag-Namens spezifiziert:

```
<xsl:template match="Name des Tags"/>
```

Es ist auch möglich mehrere Tags zu bestimmen, sodass das entsprechende XSL-Konstrukt für jeden der spezifizierten Tags gilt. Dabei werden die Tags durch das Zeichen „|" getrennt:

```
<xsl:template match="Name des ersten Tags|Name des zweiten Tags"/>
```

Man kann auch Tags spezifizieren, die innerhalb eines anderen Tags vorkommen müssen. Es ist also möglich, Tags abhängig von ihren übergeordneten Tags — innerhalb der Baumstruktur — zu formatieren. Hierbei spezifiziert man die Auftretungsreihenfolge in der Baumstruktur dadurch, dass man den Namen des übergeordneten Tags mit dem nachgestellten Zeichen „/" vor den Namen des gewünschten Tags stellt. Dies lässt sich mit beliebiger Verschachtelungstiefe fortsetzen:

```
<xsl:template match="Name des übergeordneten Tags/
    Name des eigentlichen Tags"/>
```

Will man einen „Pfad" in der Baumstruktur von der Wurzel ab spezifizieren, so stellt man ein weiteres „/" vor den Ausdruck:

```
<xsl:template match="/Name des Wurzel-Tags/Name des eigentlichen Tags"/>
```

Auch die Spezifikation eines Attributs eines Tags wird unter Verwendung des Zeichens „/" durchgeführt. Dem Attribut wird dabei das Zeichen „@" vorangestellt und der Tag mit nachgestellten „/" davorgestellt:

<**xsl:template match**="*Name des Tags/@Name des Attributs*"/>

Es ist auch möglich an Stelle des Names eines Tags einen Platzhalter für einen beliebigen Tag zu verwenden, nämlich mit Hilfe des Zeichen „*":

<**xsl:template match**="*"/>

Will man alle Tags eines bestimmten Namens unabhängig von der Position in der Baumstruktur — aber in der Reihenfolge ihres Auftretens im Dokument — spezifizieren, so stellt man die Zeichen „//" davor:

<**xsl:apply−templates match**="*//Name des Tags*"/>

Die hier gezeigten Möglichkeiten von XLink sind natürlich nur ein kleiner Ausschnitt, dessen, was die Sprache bietet. Sie sind jedoch ausreichend, um alle entsprechenden Konstrukte in den Stylesheets des Projektes „Elektronisches Publizieren" zu verstehen.

4.7. Sonstige XSL-Konstrukte

4.7.1. Werte

Manchmal will man nicht den durch eine Vorlage formatierten, sondern den echten Inhalt eines Tags oder den Wert eines Attributs oder eines Platzhalters als Text in das Ausgabedokument einsetzen. Dazu gibt es das XSL-Konstrukt

<**xsl:value−of select**="*Objekt*"/>

„*Objekt*" kann dabei ein XLink-Muster zur Spezifikation von Tags oder Attributen oder ein Platzhalter sein. Auch die Verwendung von XSL-Funktionen ist möglich.

4.7.2. Text

Da die Verwendung von einfachem Text, der im Ausgabedokument erscheinen soll, zwischen den XSL-Tags oft zu ungewünschten Leerzeichen durch Umbrüche führt (vgl. Abschnitt 2.5), gibt es die Möglichkeit reinen Text durch den XSL-Tag <**xsl:text**> einzuklammern. Wird innerhalb einer Vorlage einfacher Text immer in einen solchen Tag eingeschlossen, so werden keine Leerzeichen zwischen den XSL-Tags mit ins Ausgabedokument übernommen, sondern nur solche, die in <**xsl:text**>-Tags stehen.

4.7.3. Nummerierung

Kapitel und Abschnitte einer wissenschaftlichen Arbeit sollen oft durchnummeriert werden. Auch bei anderen Vorlagen ist es manchmal nötig die nummerische Position eines Tags zu wissen. Dazu dient das XSL-Konstrukt

```
<xsl:number/>
```

Es liefert die Nummer des aktuellen Tags innerhalb des übergeordneten. Dadurch ist es sehr einfach möglich, Kapitel und darin enthaltene Abschnitte durchzunummerieren. Beispiel 14 verdeutlicht dies.

Beispiel 14 *Wir betrachten uns die Struktur einer einfachen wissenschaftlichen Arbeit:*

```
<diplom>
  <kapitel titel="Einleitung">
    <abschnitt titel="Grundlagen">
    </abschnitt>
    <abschnitt titel="Spezialitäten">
    </abschnitt>
  </kapitel>
  <kapitel titel="Schluss">
    <abschnitt titel="Fazit">
    </abschnitt>
    <abschnitt titel="Ausblicke">
    </abschnitt>
  </kapitel>
</diplom>
```

Nun wollen wir die Überschriften der Kapitel und Abschnitte — die ja im jeweiligen `titel` *-Attribut stehen — mit vorangestellter Kapitel- bzw. Abschnittsnummer im Ausgabedokument erzeugen. Dazu dient uns die Vorlage*[6]*:*

```
<xsl:template match="kapitel|abschnitt">
  <h1>
    <xsl:number/>
    <xsl:text>. </xsl:text>
    <xsl:value-of select="@title"/>
  </h1>
  <xsl:apply-templates/>
</xsl:template>
```

Abbildung 2 zeigt die aus den Beispieldaten erzeugte Ausgabe im Web-Browser.

Nummerierung kann aber auch zu ganz anderen Zwecken als zur Abschnittsnummerierung benutzt werden. Im Projekt „Elektronisches Publizieren" beispielsweise, benutzen wir Nummerierung zur Komposition der Parameter bei den „Erweiterungen" (vgl. Beispiel 13). Dazu nummerieren wir alle „Erweiterungen" im Dokument durch und können so

[6] Der Einfachheit halber verzichte ich in dem Beispiel darauf, Kapitel- und Abschnittsüberschriften verschieden zu formatieren.

Abbildung 2: Nummerierte von Überschriften im Netscape Navigator™

jede „Erweiterung" als einzelnes Ausgabedokument auf Anfrage ausgeben. Der XSL-Tag `<xsl:number/>` wird dazu mit dem zusätzlichen Attribut `level="any"` ausgestattet, welcher eine Nummerierung auf aller Tag des gegebenen Typs ausweitet. So erhält jeder `<erweitert>`-Tag eine eindeutige Nummer.

4.7.4. XSL-Funktionen

Wie schon in Abschnitt 4.4 erwähnt, gibt es eine Vielzahl von Funktionen in XSL. Eine XSL-Funktion besteht aus einem klein geschriebenen Namen, gefolgt von einer geklammerten Liste von Parametern der Funktion. Die einzelnen Parameter werden dabei durch Kommata getrennt, Zeichenkettenparameter zusätzlich in „" " eingeklammert, wenn sie nicht mit Hilfe eines Platzhalters oder als Wert eines Tags des Ausgangsdokuments angegeben werden. Es gibt auch Funktionen, die keinerlei Parameter erhalten.

Bei der Verarbeitung werden zunächst die Parameter der Funktion durch ihren textuellen Inhalt oder den spezifizierten XLink ersetzt, je nach Art des Parameters der Funktion. Anschließend wird die gesamte Funktion durch den von ihr berechneten Wert — der natürlich abhängig von den Parametern ist — ersetzt. Dann erst wird die entsprechende Vorlage oder Variablendefinition weiter ausgewertet.

Da die Anzahl der Funktionen die XSL bietet den Umfang dieser Arbeit sprengen würde, beschränke ich mich auf die im Projekt „Elektronisches Publizieren" verwendete Funktion

```
dokument("Dateiname")
```

die die Baumstruktur einer XML-Datei, die unter dem Dateinamen „*Dateiname*" zu finden ist, zurückliefert.

Wie schon in Abschnitt 2.2 erwähnt, verweisen im Projekt „Elektronisches Publizieren" die FAQ-Dokumente — und auch alle anderen Dokumenttypen — auf einen Autor und einen Designer, die in einem Person-Dokument, das in dem FAQ-Dokument mit Hilfe seiner ID spezifiziert ist, genauer beschrieben sind.

Bei der Verarbeitung der FAQ-Dokumente benützen wir nun diese IDs, um daraus Dateinamen für die korrespondierenden XML-Dateien zusammenzusetzen. Mit Hilfe der `dokument()`-Funktion können wir dann auf die Baumstruktur dieser externen Dateien zugreifen und die für uns wichtigen Daten wie Vor- und Nachname des Autors bzw. Designers in unser Ausgabedokument bringen.

4.7.5. Präformatierter Text

Zum Schluss möchte ich nochmals auf die besondere Behandlung von Leerzeichen, wie bereits in Abschnitt 2.5 erwähnt, eingehen. Dort sagte, ich dass es grundsätzlich egal ist, wieviele „Leerzeichen" man zwischen zwei Worten zu deren Trennung anbringt. Bei der Verarbeitung werden diese zu einem einzigen Leerzeichen — und insbesondere ohne Zeilenumbruch — zusammengefasst. Das ist leider nur die halbe Wahrheit.

In HTML gilt eigentlich die gleiche Behandlung von Leerzeichen, wie in XML. Dies nützen auch die Verarbeitungsprogramme, in dem sie die überschüssigen Leerzeichen bei der Verarbeitung von XML nach HTML *nicht* eliminieren, sondern diese Aufgabe an den entsprechenden Web-Browser weiterreichen. Bei der Erstellung von Vorlagen, hat dies den Vorteil, dass man zur Darstellung von präformatiertem Text einfach den entsprechenden Tag aus dem HTML-Befehlssatz — nämlich `<pre>` — benützt, um solcherlei Layout zu erhalten.

Kommt als Ausgabeformat PDF [2] oder ein anderes Format, das keine SGML-artige Sprache ist, zum Einsatz, so werden wirklich standardmäßig mehrere Leerzeichen zu einem zusammengefaßt. Um bei solcherlei Formaten, einen präformatierten Text als solchen wiederzugeben, gibt es spezielle XSL-Konstrukte [1, Abschnitt 7.15].

4.8. Der nötige Rahmen

Wie schon in Abschnitt 1.4 erwähnt, ist das Format einer XSL-Datei fest vorgegeben. Bisher habe ich allerdings nur Tags vorgestellt, die eine bestimmte Funktionalität innerhalb von XML haben.

Wie bei jeder anderen XML-Datei auch ist es aber wichtig, dass eine XSL-Datei einen eindeutigen Wurzel-Tag besitzt. Auch dieser ist in der Spezifikation der Sprache vorgegeben. Er heißt:

```
<xsl:stylesheet version="1.0">
```

und muss alle `<xsl:template>`-, `<xsl:variable>`- und `<xsl:param>`-Tags umschließen.

Auch wenn der oben genannte Wurzel-Tag in seiner Minimalfassung ausreicht, ist es doch üblich, in ihm die verwendeten Tag-Namensräume und die dafür gewählten Namensvorsätze anzugeben.

Namensräume sind eine Gliederungsmöglichkeit in XML, die ich bisher bewusst nicht erwähnt habe, und auch hier nicht ausreichend erklären werde. Der Grund dafür ist, dass man als normaler Anwender damit so gut wie nie in Berührung kommt und sie das Verständnis der Sprache XML dementsprechend nur unnötig erschweren.

Da wir in unserem Projekt bisher nur HTML-Dokumente erstellt haben, deklarieren alle XSL-Dateien im `<xsl:stylesheet>`-Tag den Namensraum der Hypertext Markup Language der Version 4.01 [12] ohne Namensvorsatz. Dies macht es möglich, alle HTML-Tags im Stylesheet wie gewohnt zu schreiben.

Außer der Hypertext Markup Language wird in allen Stylesheets die eXtensible Stylesheet Language verwendet. Damit man die XSL-Tags besser erkennen und von den HTML-Tags unterscheiden kann ist es üblich den Namensraum von XSL mit dem Namensvorsatz „xsl" zu deklarieren. Die Stylesheets des Projekts „Elektronisches Publizieren" werden — neben der grundlegenden Deklarierung nach Abschnitt 2.1 — also wie folgt eröffnet:

```
<xsl:stylesheet version="1.0"
                xmlns="http://www.w3c.org/TR/REC-html40/loose.dtd"
                xmlns:xsl="http://www.w3c.org/1999/XSL/Transform">
```

Die für die Namensräume angegebenen Werte müssen nicht notwendigerweise auch URLs auf echte Ressourcen im Internet darstellen, sondern sind lediglich weltweite Platzhalter für den jeweils spezifizierten Namensraum.

5. Verwendete Programme

5.1. Web-Server

Das wichtigste Programm bei der elektronischen Publikation im Internet ist wohl der Web-Server, der die Dokumente an die Konsumenten ausliefert. Nicht aus Kostengründen, sondern vielmehr wegen der großen Verbreitung und Erweiterbarkeit, war uns im Projekt „Elektronisches Publizieren" von Anfang an klar, dass wir als Web-Server das Programme Apache einsetzen.

Der Web-Server Apache wurde und wird von der „Apache Software Foundation" entwickelt (http://www.apache.org). Da der Apache „freie Software"[7] ist, ergibt sich als angenehmer Nebeneffekt, dass er für unser Projekt kostenfrei verfügbar ist.

Apache gibt es in verschiedenen Versionsständen für Windows und fast alle Unix-Plattformen. Da die Windows-Version derzeit noch mehr aus Studie, denn als zum professionellen Einsatz konzipierte Version zu sehen ist, war auch klar, dass wir den Apache für Linux, ein ebenfalls „freies" Unix-Betriebssystem, verwenden. Zum Einsatz kam Version 1.3.19.

Apache Version 1.3.19 ist normalerweise in einer derzeitigen Linux-Installation enthalten und bedarf eigentlich keinerlei großartiger Anpassungen. Auf die Konfiguration benötigter Zusatzmodule werde ich in den zugehörigen Abschnitten eingehen.

5.1.1. Modul für eine Java-Servlet-Engine

Die von uns verwendete Software zur XML-Datenverarbeitung Cocoon (vgl. Abschnitt 5.2) ist in der plattformübergreifenden Programmiersprache Java [4] geschrieben. Deshalb erfordert sie einen Web-Server mit einer sogenannten Java-Servlet-Engine, die die Kommunikation von Cocoon mit dem Web-Server, der die Daten ja ausliefern soll und an den die Anfragen ja auch gestellt werden, herstellt.

Zu diesem Zweck ist unser Projekt-Web-Server mit dem Module JServ ausgestattet. Dieses Modul stellt für den Web-Server die nötige Schnittstelle zur Kommunikation mit Java-Servlets dar und für Cocoon als Java-Servlet die sogenannte Laufzeitumgebung, in der es seine Arbeit verrichtet.

Das Module JServ benötigt außerdem eine sogenannte Java Runtime Environment (JRE, deutsch: „Java-Laufzeitumgebung") in der es die Servlets ausführt. Dabei kam bei uns die JRE Version 1.3 von Blackdown (http://www.blackdown.org) zum Zuge, da die Firma Sun keine eigene Java-Laufzeitumgebung für das Betriebssystem Linux bietet.

Zum Einsatz kam bei uns Version 1.1.2 des Modules JServ. Das Modul entstammt den „Apache JServ Project" (http://java.apache.org/jserv/) und ist unter der gleichen Lizenz wie Apache selbst verfügbar.

[7] Der Begriff der „freien Software" soll hier nicht weiter erklärt werden. Eine genaue Definition findet man unter [8].

5.1.2. Modul für die Skriptsprache PHP

Für die Datenbankanbindung wurde das Web-Server-Modul für die Skriptsprache PHP benötigt. Da diese Modul nicht zum Themengebiet dieser Arbeit gehört, wollte ich es hier nur kurz erwähnen. Genaueres zu seiner Verwendung finden Sie in der Projektarbeit zur Datenbank [14].

5.2. Cocoon

Das Kernprogramm unseres Verarbeitungsprozesses für XML-Dateien ist das Java-Servlet Cocoon aus dem „Apache XML Projekt" (http://xml.apache.org), das alle Programme dieses Projektes zusammenfasst. Es enthält unter anderem:

- dem XML-Parser Xerces (http://xml.apache.org/xerces-j/)
- das XSL-Transformationsprogramm Xalan (http://xml.apache.org/xalan-j/)
- dem PDF-Formatieren Fop (http://xml.apache.org/fop/)

Auf die einzelnen Bestandteile will ich nicht weiter im Detail eingehen. In unserem Projekt wurden eigentlich nur Xerces und Xalan benutzt, da wir lediglich XML-Dateien formatieren wollten, wofür man einen Parser zum Einlesen der Daten und ein XSL-Transformationsprogramm zur Verarbeitung der Daten benötigten.

Cocoon stellt nun — ähnlich dem Module JServ (vgl. Abschnitt 5.1.1) für Java-Servlets — einen Rahmen für die in ihm enthaltenen Programme dar, sodass diese perfekt miteinander agieren können. Dazu wird das Cocoon-Servlet als Verarbeitungsprogramm für XML-Dateien in der Konfiguration des Web-Servers eingetragen:

```
Action cocoon /servlet/org.apache.cocoon.Cocoon
AddHandler cocoon xml
```

Dieser liefert dann Anfragen nach XML-Dateien nicht mehr an den Besucher der Webseite direkt aus, sondern übergibt die Datei an Cocoon, der diese verarbeitet und die Ausgabe mittels des Web-Servers an des Besucher schickt.

Die Verarbeitung einer XML-Datei mit Cocoon bedeutet dann, dass Cocoon die Datei mit Hilfe von Xerces parst und damit die Baumstruktur zur Weiterverarbeitung aufstellt. Dann wendet Cocoon mit Hilfe von Xalan das in der Deklaration der XML-Datei angegebene Stylesheet auf die Baumstrukur an und schickt die Ausgabe zum Versenden an den Besucher an den Web-Server Apache.

Während wir zu Anfang des Projektes noch Version 1.8 von Cocoon verwendeten, läuft inzwischen Version 1.8.2, die kleine Geschwindigkeitsvorteile gebracht hat. Geschwindigkeit ist leider eine der Schwächen von Cocoon, die jedoch in der verwendeten Programmiersprache Java und nicht in der Programmiertechnik der Entwickler begründet ist. Ein leistungsstarker Rechner ist also Voraussetzung für eine stark frequentierte

Web-Site eines Projektes, das Cocoon verwenden will. Auch Cocoon ist unter der Lizenz der „Apache Software Foundation" kostenfrei verfügbar.

6. Diskussion

Nachdem nun in den vorherigen Abschnitten alle Techniken, die in unserem Projekt „Elektronisches Publizieren" zur Erzeugung und Verarbeitung von XML-Dokumenten benötigt wurden, beschrieben wurden will ich nun die Ergebnisse meiner Arbeit zusammenfassen und einige Ausblicke auf ungenutzte Möglichkeiten in unserem Projekt und in XML geben.

6.1. Fazit

Im Projekt „Elektronisches Publizieren" haben wir für verschieden Dokumenttypen XML-Formate erstellt. Dabei war es unter anderem auch meine Aufgabe, einen einheitlichen Stil zu gewährleisten und für eine ausgefeilte Interaktionsmöglichkeit zwischen den Dokumenttypen zu sorgen.

Der einheitliche Stil der Dokumente ist an der in allen Typen vorhanden Gliederung in Dokumentkopf (spezifiziert durch den `<head>`-Tag) und Dokumentinhalt zu erkennen. Weiterhin haben alle Wurzel-Tags der verschiedenen Dokumentarten eine eindeutige ID (vgl. Abschnitt 2.2), was auch der Interaktion zugute kommt. Es existieren einheitliche Werkzeuge für Verweise auf andere Dokumente und zur Textformatierung (vgl. Anhang B, Zeilen 27ff).

Unter der Interaktion der Dokumente verstehe ich die Möglichkeit, auf andere Dokumente des Projektes zu verweisen, sei es nun, um sie im Text als Quelle oder weiterführendem Informationsverweis zu spezifizieren oder um Daten aus diesen anderen Dokumenten mit in das Ausgabedokument einzubringen. Als Beispiel für die Dateneinbringung sei hier das Prinzip der Autor- und Designerspezifikation in den Dokumentköpfen erwähnt (vgl. Abschnitt 2.2). Verweise zu Quellen und weiterführenden Informationen können mit verschiedenen Tags spezifiziert werden, sodass es möglich ist, Verweise auf jeden Dokumenttyp anders in der Ausgabe darzustellen. Fernen kann bei verschiedenen Verweis-Tags auch auf Teile des Gesamtdokuments verwiesen werden (vgl. Anhang B, Zeilen 26–70).

Eine weitere Aufgabe meinerseits war es, bei der komplizierten Erstellung des Layouts zu helfen, wie beispielsweise der Anzeige der „Erweiterungen" in einem neuen Fenster (vgl. Beispiel 13). Auch die einfache Lösung des Problems der Darstellung von vorformatiertem Text (vgl. Abschnitt 4.7.5) war erst nach langer Recherche in den Quellen möglich.

Abschließend möchte ich noch einem der Projektteilnehmer danken, der mir mit seinem weitreichenden Wissen über XML, XSL und Cocoon stets hilfreich zur Seite stand. Obwohl seine Projektarbeit [15] eigentlich keinerlei Überschneidungen mit vorliegenden Arbeit zeigt, hat er mir stets helfen können, wenn ich bei einem Problem keine Lösung fand. Vielen Dank, Timo Scheuer.

6.2. Ausblicke

Die Sprache XML — und natürlich auch alle damit zusammenhängenden Sprachen und Konstrukte des W3C — deckt ein sehr weit gefächertes Gebiet ab. Neben den hier im Projekt verwendeten Eigenschaften bei der Dokumentenverwaltung gibt es unzählige andere. Als Beispiele seien hier nur die Anwendung als Oberflächenbeschreibungssprache im KDE-Projekt (http://www.kde.org) oder als Format zur Speicherung von Programmkonfigurationen genannt.

Die Möglichkeiten zur Gestaltung der Dokumente wurden in diesem Projekt maximal skizziert, meist beschränkten wir und auf einen kleinen Teilaspekt, dessen was unterstützt wird. Eine erste interessante Weiterentwicklung wäre beispielsweise die Unterstützung anderen Ausgabeformate wie PDF zum komfortablen Ausdruck oder WML, um Inhalte mobil zur Verfügung zu stellen.

Auch „hinter den Kulissen" gibt es einige Möglichkeiten zur Verbesserung und Verfeinerung. Das W3C hat inzwischen einen Nachfolger für das etwas angestaubte DTD-System entwickelt: XML-Schema. Mit dieser XML-artigen Sprache ist es möglich, sehr fein das Format einer XML-Datei zu definieren. Da XML-Schema zur Zeit der Projektentwicklung jedoch noch in der Entwicklung begriffen war, wurde es hier noch nicht verwendet.

Ein weiterer Punkt bei der Gestaltung des Projektes als öffentliche Informationsbasis ist, dass dazu die Strukturierung noch weiter ausgebaut werden muss, wie ich es auch schon unter [6] angefangen habe, jedoch aufgrund von Zeitmangel nicht zu Ende führen konnte.

6.3. persönliche Abschlußgedanken

Die Arbeit beim Projekt „Elektronisches Publizieren" und die damit verbundene Auseinandersetzung mit dem für mich damals neuen Thema XML hat mir sehr viel Spaß gemacht. Mittlerweile konnte ich die erarbeiteten und erlernten Erkenntnisse auch für ein privates Projekt, die Homepage meines Orchestervereins unter http://www.oho-ormesheim.de, einsetzen und auch weiterentwickeln.

Ich habe auch überlegt, die Erkenntnisse dieser Arbeit bei ihrer Erstellung zu verwenden. Sie ist jedoch nicht im XML-Format erstellt, sondern mit dem Textsatzsystem LaTeX, das mit ausgezeichneten Layouteigenschaften und vielen vorgefertigten Dokumentenvorlagen überzeugen konnte. Eine Entwicklung der Arbeit im XML-Format hätte mich nochmals viel Zeit bei der Erstellung der Vorlagen gekostet, für die ich außerhalb dieser Arbeit keine Verwendung mehr gesehen habe.

Anhang

A. Beispiel eines FAQ-Dokuments

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE faq SYSTEM "faq.dtd">
<?xml-stylesheet href="faq-html.xsl" type="text/xsl"?>
<?cocoon-process type="xslt"?>

<faq id="faq_00001">
  <head>
    <titel alpha="nutzung">Fragen und Antworten zu Friedrich
    Nietzsche.</titel>
    <autor id="per_prossliner_johann_00">
      <datum>
        <jahr>2001</jahr>
        <monat>02</monat>
        <tag>12</tag>
      </datum>
    </autor>
    <design id="per_joerg_brigitte_00" typ="erstellung">
      <datum>
        <jahr>2001</jahr>
        <monat>03</monat>
        <tag>17</tag>
      </datum>
    </design>
    <thema>Zu den Fragen und Antworten selbst</thema>
  </head>
  <body sprache="DE">
    <frage>Wie zu benützen</frage>
    <antwort>
```

```
      <p>Es dürfte ziemlich egal sein, bei welchem Zipfel man
      Nietzsche zu fassen bekommt – Hauptsache man lässt ihn nicht
      gleich wieder los. Falls man Nietzsches Werk als Irrgarten
      erlebt, kommt es nur darauf an, das Wort &quot;Garten&quot; zu
      unterstreichen, nicht das Wort &quot;Irr&quot;.</p>

      <p>Dazu ist zweierlei nötig: Übersicht und Freude. Beides
      liegt nicht auf der Hand, beides muss gewonnen werden. Dazu
      müssen bestimmte
      <erweitert>
        <beschriftung>Flechtwerke entflochten</beschriftung>
        <p>Das ärgert manche, die bereits imstande sind, sich am
        Flechtwerk zu erfreuen. Wo das aber (noch) nicht der Fall
        ist, könnte es Freude bringen. Zur Entflechtung gehört
        Weglassen, Herauspicken, Hervorheben, Verweisen. Ich habe
        mein Bestes getan, damit das nicht willkürlich wird oder
        wirkt.</p>
      </erweitert>
      werden.</p>

      <p>Nicht selten habe ich mit den im Web üblichen – und so
      fabelhaft nützlichen – markierten Links zu einem Hilfsmittel
      gegriffen. Es öffnet sich dort ein neues Fenster, wo ich die
      Leser/User nicht von der jeweiligen Frage weggelockt wollte zu
      einer anderen Frage (von wo sie den Rückweg womöglich nicht
      mehr finden, weil es dort ja weitere Verweisungen gibt). Ein
      kursiver Link mit Fußnote bedeutet also: Dazu gibt es
      zusätzliche Informationen; das Betreffende ist auch über die
      Schlagwortliste zu finden, aber ich empfehle nicht, den
      aktuellen Kontext zu verlassen und sofort dorthin zu
      eilen.</p>
    </antwort>
  </body>
</faq>
```

B. Die DTD für FAQ-Dokumente

```
<!-- DTD fuer Nietzsche - Fragen und Antworten - Brigitte Joerg -->
<!-- Stand 11.03.2001 -->

<!ENTITY % block "p|erweitert|sprache|rede|extrakt_ref|beleg_ref|komm_ref|brief_ref|
    ewerk_ref| bib_ref | link_ref | quelle_ref | per_ref | ort_ref | faq_ref | extern_ref">
<!ELEMENT faq                (head,body,anmerkung*)>
  <!ATTLIST faq              id      ID                  #REQUIRED>
  <!ELEMENT head             (titel,autor+,design+,thema+,sw*,rank?)>
    <!ELEMENT titel          (#PCDATA)>
      <!ATTLIST titel        alpha   CDATA               #IMPLIED>
      <!ATTLIST autor        id      IDREF               #REQUIRED>
        <!ELEMENT datum      (dattext*,jahr?,monat?,tag?)>
          <!ELEMENT dattext  (#PCDATA)>
          <!ELEMENT jahr     (#PCDATA)>
          <!ELEMENT monat    (#PCDATA)>
          <!ELEMENT tag      (#PCDATA)>
    <!ELEMENT design         (datum)>
      <!ATTLIST design       id      IDREF               #REQUIRED
                             typ     (erstellung|modifikation|verschlagwortung)
                                     #REQUIRED>
    <!ELEMENT thema          (#PCDATA)>
    <!ELEMENT sw             (#PCDATA)>
    <!ELEMENT rank           EMPTY>
      <!ATTLIST rank         value   (1|2|3)             #REQUIRED>
  <!ELEMENT body             (frage?,antwort)>
    <!ATTLIST body           sprache (DE|DEO|FR|IT|LT|GR) #REQUIRED
      >
    <!ELEMENT frage          (#PCDATA)>
    <!ELEMENT antwort        (p)*>
      <!ELEMENT p            (%block;|br|ul|ol|hinweis|wichtig|betont)*>
        <!ELEMENT erweitert  (beschriftung|%block;)*>
          <!ELEMENT beschriftung (%block;)*>
        <!ELEMENT br         EMPTY>
        <!ELEMENT ul         (%block;|li)*>
          <!ELEMENT li       (%block;)*>
        <!ELEMENT ol         (%block;|li)*>
        <!ELEMENT sprache    (%block;)*>
```

```
  <!ATTLIST sprache  sprache (DE|DEO|FR|IT|LT|GR) #REQUIRED
      >
<!ELEMENT rede      (%block;|luecke|auslassung)*>
  <!ATTLIST rede      typ       (direkt | indirekt)  #REQUIRED
                      id        IDREF                #REQUIRED>
  <!ELEMENT luecke EMPTY>
  <!ELEMENT auslassung EMPTY>

<!ELEMENT hinweis   (#PCDATA)>
      <!ELEMENT wichtig   (#PCDATA)>
<!ELEMENT betont    (#PCDATA)>

<!ELEMENT extrakt_ref (%block;|luecke|auslassung)*>
  <!ATTLIST extrakt_ref id IDREF                   #REQUIRED>
<!ELEMENT beleg_ref (%block;)*>
  <!ATTLIST beleg_ref id   IDREF                   #REQUIRED
                           medium CDATA            #REQUIRED>
<!ELEMENT komm_ref (%block;)*>
  <!ATTLIST komm_ref id  IDREF                     #REQUIRED>
<!ELEMENT brief_ref (%block;)*>
  <!ATTLIST brief_ref id   IDREF                   #REQUIRED>
<!ELEMENT ewerk_ref (%block;)*>
  <!ATTLIST ewerk_ref id  IDREF                    #REQUIRED>
<!ELEMENT bib_ref   (%block;)*>
  <!ATTLIST bib_ref   id   IDREF                   #REQUIRED>
<!ELEMENT link_ref  (%block;)*>
  <!ATTLIST link_ref  id   IDREF                   #REQUIRED>
<!ELEMENT quelle_ref (%block;)*>
  <!ATTLIST quelle_ref id  IDREF                   #REQUIRED>
<!ELEMENT per_ref   (%block;)*>
  <!ATTLIST per_ref   id   IDREF                   #REQUIRED>
<!ELEMENT ort_ref   (%block;)*>
  <!ATTLIST ort_ref   id   IDREF                   #REQUIRED>
<!ELEMENT faq_ref   (%block;)*>
  <!ATTLIST faq_ref   id   IDREF                   #REQUIRED>
<!ELEMENT extern_ref (#PCDATA)>
  <!ATTLIST extern_ref href IDREF                  #REQUIRED>

      <!ELEMENT anmerkung (#PCDATA)>
  <!ATTLIST anmerkung id IDREF                     #REQUIRED>
```

Literatur

[1] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, und Steve Zilles. Extensible Stylesheet Language (XSL) Version 1.0, 2000. http://www.w3.org/TR/xsl/.

[2] Adobe Systems Incorporated. *PDF Reference Version 1.3, 2nd Edition*. Addison-Wesley, 2000. http://partners.adobe.com/asn/developer/acrosdk/docs/PDFRef.pdf.

[3] Bert Bos, Håkon Wium Lie, Chris Lilley, und Ian Jacobs. Cascading Style Sheets, level 2, 1998. http://www.w3.org/TR/REC-CSS2/.

[4] Gilad Bracha, James Gosling, Bill Joy, und Guy Steele. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000. http://java.sun.com/docs/books/jls/.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, und Eve Maler. Extensible Markup Language (XML) 1.0, 1998. http://www.w3.org/TR/REC-xml/.

[6] Patrick Cernko. Globale DTDs (im Projekt „Elektronisches Publizieren am Beispiel von F. Nietzsche"), 2001. http://nietzsche.ps.uni-sb.de/projekt/teilprojekte/globale_dtds/.

[7] Steve DeRose, Eve Maler, und David Orchard. XML Linking Language (XLink) Version 1.0, 2001. http://www.w3.org/TR/xlink/.

[8] Free Software Foundation. The Free Software Definition, 2001. http://www.gnu.org/philosophy/free-sw.html.

[9] Dorit Günter. Dokumentation zum Bereich „Werke" im Informationssystem „Nietzsche Online", 2001. http://nietzsche.ps.uni-sb.de/projekt/teilprojekte/werke/.

[10] Elliotte Rusty Harold. *XML Bible*. IDG Books Worldwide, Inc., 1999.

[11] Marc Hofmann. Multimedia-Datenformate (Kapitel 11: SGML), 1995. http://i31www.ira.uka.de/docs/semin94/11_SGML/.

[12] Arnaud Le Hors, Ian Jacobs, und Dave Raggett. HTML 4.01 Specification, 1999. http://www.w3.org/TR/html401/.

[13] Brigitte Jörg. Konzeption eines Frage-Antwort-Katalogs in Zusammenarbeit mit dem Kastell-Verlag, 2001. http://nietzsche.ps.uni-sb.de/projekt/teilprojekte/faq/.

[14] Till Kinstler. Konzept für ein datenbankbasiertes Registersystem, 2001. http://nietzsche.ps.uni-sb.de/projekt/teilprojekte/db/register.pdf.

[15] Timo Scheuer. Konzeption eines Online-Shops für den Kastell-Verlag, 2001. http://nietzsche.ps.uni-sb.de/projekt/index.html (derzeit noch nicht verfügbar).

[16] Wolfgang Schwarz. Überparteilich. *iX - Magazin für Professionelle Informationstechnik*, Seiten 10ff, 10 / 2000.