Die kleine XML-Apotheke
Einführendes zum Thema XML, XSL und Datenbanken

Burkhardt Renz
Fachhochschule Gießen-Friedberg Fachbereich MNI
Burkhardt.Renz@mni.fh-giessen.de

Version 1.0, 30. Januar 2001

Unsere kleine XML-Apotheke ist im Original die Lukas-Apotheke am Frankfurter Parlamentsplatz — nebenbei: sie hat die freundlichsten Mitarbeiter der nördlichen Hemisphäre. (Bild: Claudia Fritsch)

An einem einfachen Beispiel, dem einer kleinen Apotheke, wird praktisch demonstriert, wie man XML und XSL in Verbindung mit relationalen Datenbanken einsetzen kann. Wir erstellen XML-Dokumente, stellen sie mit XSL dar und speichern sie in einer Datenbank. Alle Mittel für diese Demonstration sind bewusst einfach gewählt.

Betreten wir also unsere kleine XML-Apotheke und lassen uns überrraschen, welche Mittelchen, Medikamente und Rezepte sie für uns bereithält:

1 XML – Inhalt mit Semantik und Struktur

1.1 Inhalt – Semantik & Struktur – Gestaltung

Werfen wir doch einfach einen Blick in den Inhalt unserer Apotheke:

```
Pantozol|v|Pantoprazol-Natrium-Sesquihydrat|Refluxkrankheit|Byk Gulden
Rhinopront|a|Carbinoxaminhydrogenmaleat|Schnupfen|Mack
Codipront|v|Phenyltoloxamindohydrogencitrat|Reizhusten|Mack
Novalgin|v|Metamizol-Natrium|Schmerzen|Hoechst
Glycilax||Glycerol|Verstopfung|Engelhard
```

Wer sich mit Medikamenten auskennt, wird wahrscheinlich erkennen, worum es sich handelt: eine Liste von Medikamenten, mit Angaben, die durch das Zeichen `"|"` getrennt sind. Für jedes Medikament ist aufgelistet:

- Der Name des Medikaments.
- Ein Kennzeichen, ob das Medikament apothekenpflichtig (`"a"`) oder verschreibungspflichtig (`"v"`) ist.
- Der Wirkstoff, der im Medikament enthalten ist.
- Die Beschwerden, bei denen das Medikament angewendet werden kann.
- Der Hersteller des Medikaments.

Wer sich jedoch nicht mit Medikamenten auskennt, so zum Beispiel ein Computer, kann wenig mit dieser Information anfangen. Der *Inhalt* der kleinen Apotheke allein eignet sich nicht dazu, von einer Software verarbeitet zu werden. Denkt man zum Beispiel daran, diese Liste einem Lieferanten zwecks Bestellung zu geben, so wird er (oder seine Software) sie nur verstehen, wenn der genaue Aufbau der Liste mit ihm vereinbart ist.

Vielleicht wird die Sache besser, wenn wir in das Schaufenster unserer kleinen Apotheke schauen. Nehmen wir für einen Moment an, die kleine Apotheke ist im Internet präsent und stellt sich etwa wie in Abb. 1 dar.

In der *Gestaltung* des Inhalts unserer Apotheke können wir erkennen, dass sich Informationen unterscheiden, wir erkennen eine Gruppierung und wir sehen eine gewisse Struktur: „Pantozol “ etwa hat den Charakter einer Überschrift. Aber: über die Bedeutung dieser Information erfahren wir aus dem Quelltext dieser HTML-Seite nichts.

```
<html>
    <head>
        <title>Die kleine XML-Apotheke</title>
    </head>
    <body>
        <h1>Die kleine XML-Apotheke</h1>
        <h2>Pantozol</h2>
```

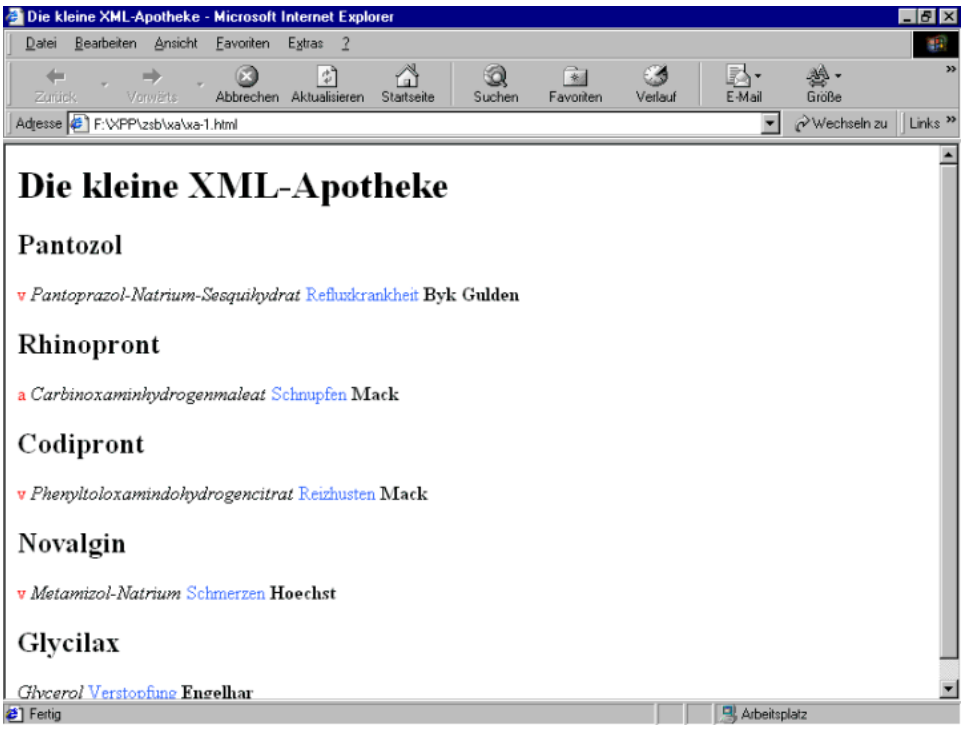

Abbildung 1: Gestaltung der kleinen Apotheke in HTML

```
        <p> <font color="FF000">v</font>
            <i>Pantoprazol-Natrium-Sesquihydrat</i>
            <font color="3366FF">Refluxkrankheit</font>
            <b>Byk Gulden</b>
        </p>
        <h2>Rhinopront</h2>
        <p> <font color="FF000">a</font>
            <i>Carbinoxaminhydrogenmaleat</i>
            <font color="3366FF">Schnupfen</font>
            <b>Mack</b>
        </p>
        <h2>Codipront</h2>
        <p> <font color="FF000">v</font>
            <i>Phenyltoloxamindohydrogencitrat</i>
            <font color="3366FF">Reizhusten</font>
            <b>Mack</b>
        </p>
        <h2>Novalgin</h2>
        <p> <font color="FF000">v</font>
            <i>Metamizol-Natrium</i>
            <font color="3366FF">Schmerzen</font>
            <b>Hoechst</b>
        </p>
        <h2>Glycilax</h2>
        <p> <i>Glycerol</i>
            <font color="3366FF">Verstopfung</font>
            <b>Engelhar</b>
        </p>
    </body>
```

```
</html>
```

Der Grund dafür liegt darin, dass HTML im wesentlichen nur Informationen zur *Gestaltung* des Inhalts bereitstellt. Wenn man es genau nimmt, so ist HTML ein Mix von stilistischen (z.B. `<font>`), (text-)strukturellen (z.B. `<h1>`) und semantischen (z.B. `<title>`) Auszeichnungen des Textes.

Ein Beweggrund für die Entwicklung von XML bestand darin, die verschiedenen Ebenen der Information in einem Dokument zu unterscheiden und zu *trennen*: die *generische* Auszeichnung durch XML soll nicht die Gestaltung des Dokuments bestimmen, sondern soll die *Struktur* des Dokuments wiedergeben und die *Semantik* des Inhalts ausdrücken. XML-Dokumente enthalten so nicht nur den Inhalt, sondern auch die *Meta-Daten*, Informationen *über* den Inhalt des Dokuments.

Entwickeln wir unsere kleine Apotheke also zur kleinen XML-Apotheke weiter:

```
<?xml version="1.0"?>

<apotheke>
    <medikament zugang="v">
        <name>Pantozol</name>
        <wirkstoff>Pantoprazol-Natrium-Sesquihydrat</wirkstoff>
        <anwendung>Refluxkrankheit</anwendung>
        <hersteller>Byk Gulden</hersteller>
    </medikament>
    <medikament zugang="a">
        <name>Rhinopront</name>
        <wirkstoff>Carbinoxaminhydrogenmaleat</wirkstoff>
        <anwendung>Schnupfen</anwendung>
        <hersteller>Mack</hersteller>
    </medikament>
    <medikament zugang="v">
        <name>Codipront</name>
        <wirkstoff>Phenyltoloxamindohydrogencitrat</wirkstoff>
        <anwendung>Reizhusten</anwendung>
        <hersteller>Mack</hersteller>
    </medikament>
    <medikament zugang="v">
        <name>Novalgin</name>
        <wirkstoff>Metamizol-Natrium</wirkstoff>
        <anwendung>Schmerzen</anwendung>
        <hersteller>Hoechst</hersteller>
    </medikament>
    <medikament>
        <name>Glycilax</name>
        <wirkstoff>Glycerol</wirkstoff>
        <anwendung>Verstopfung</anwendung>
        <hersteller>Engelhard</hersteller>
    </medikament>
</apotheke>
```

Das XML-Dokument hat in der ersten Zeile die *XML-Deklaration*, mit der es sich als XML-Dokument identifiziert. Wir sehen dann, dass die Inhalte mit *Bedeutung* belegt werden: Es handelt sich um eine Apotheke, die Medikamente enthält. Diese haben unter anderem einen Wirkstoff.

Die *Struktur* des XML-Dokuments ist ein *Baum* (Abb. 2):

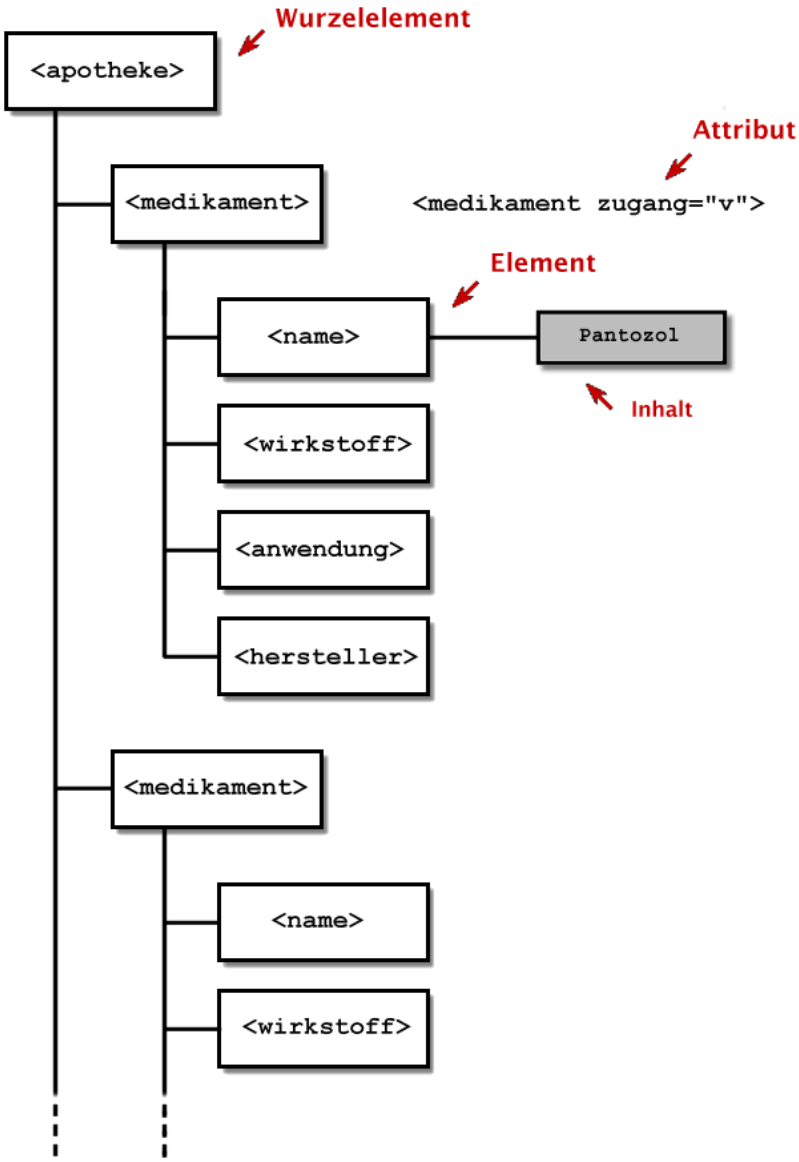

Abbildung 2: Die Struktur der kleinen Apotheke

Die Folgen der Tatsache, dass ein XML-Dokument strukturell ein Baum ist, werden oft übersehen. Gerade im Bereich von Textdokumenten gibt XML einem Text eine andere Struktur, als man sie etwa von Textverarbeitungsprogrammen oder HTML kennt. Betrachten wir ein schematisches Beispiel:

In HTML würde man schreiben

```
<h1>Überschrift Stufe 1</h1>
<h2>Überschrift Stufe 2</h2>
<p>Text</p>
```

In XML verwendet man nicht diese flache Struktur, sondern drückt den Baum aus, der durch die Textgliederung gegeben ist. Das Beispiel verwendet DocBook, eine XML-Anwendung zur Auszeichnung von technischen Textdokumenten (siehe [13]).

```
<Sect1>
    <Title>Überschrift Stufe 1</Title>
    <Sect2>
        <Title>Überschrift Stufe 2</Title>
        <Para>Text</Para>
    </Sect2>
</Sect1>
```

Halten wir fest: XML-Dokumente enthalten nicht nur die Information, sondern auch Struktur und Semantik, kurz Meta-Daten. Dadurch wird der Inhalt für Software-Programme zugänglich, er kann leichter verarbeitet werden. Um unser Beispiel des Lieferanten nochmals aufzugreifen: Natürlich brauchen wir auch beim Austausch von Informationen im XML-Format Vereinbarungen, welche Informationen enthalten sind usw. Aber der Aufbau einer jeden XML-Datei folgt den demonstrierten Prinzipien: also kann die Software, die solche Dateien lesen, verarbeiten, verändern, neu gestalten kann, auch für jedes beliebige XML-Dokument verwendet werden. XML ist so Mittel und Voraussetzung für die generische Verarbeitung *semistrukturierter Daten*.

Wir betrachten nun genauer, nach welchen Regeln XML-Dokumente aufgebaut sind.

1.2 Ordentliche XML-Dokumente

Folgenden Abbildung 3 expliziert, was wir an den Beispielen schon gesehen haben.

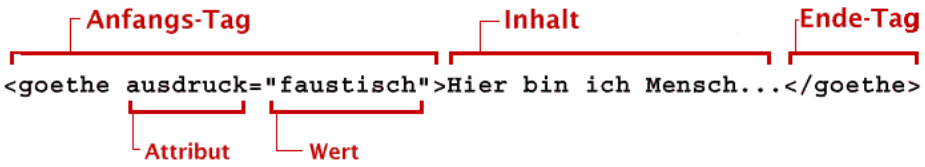

Abbildung 3: Aufbau eines XML-Elements

Es gibt ein paar Regeln, die ein XML-Parser überprüft. Ein Dokument, das diese Regeln einhält, heißt *well-formed*. (In der Literatur wird das Wort

gern mit „wohlgeformt“ übersetzt, ich erlaube mir nicht ganz so hochgestochen von „ordentlichen“ XML-Dokumenten zu sprechen.) Die Regeln sind:

1. Jedes XML-Dokument beginnt mit der *Deklaration*, in der einfachsten Form also mit

   ```
   <?xml version="1.0"?>
   ```

2. Ein XML-Dokument hat genau ein Wurzelelement.

3. Korrespondierende Anfangs- und Ende-Tags passen in der Schachtelung korrekt zusammen. (Es soll sich ja wirklich ein Baum ergeben.)

4. Kommentare werden mit `<!--` und `-->` begrenzt. Um einen XML-Parser nicht zu verwirren, vermeidet man besser `--` innerhalb von Kommentaren.

5. Die Namen von Elementen und Attributen beginnen mit einem Buchstaben.

6. Attribute werden im Anfangs-Tag erklärt.

7. Die Werte von Attributen benötigen korrespondierende Anführungszeichen, also

   ```
   <medikament zugang="a">...
   <medikament zugang='a'>...
   ```

8. Attribute können als Wert nur einfachen Text enthalten.

9. Man braucht `&lt;` und `&amp;` für < bzw. &, an manchen Stellen auch `&quot;` und `&apos;` für " und '.

10. Leere Elemente werden als `<LeeresElement/>` geschrieben.

XML-Parser überprüfen Dokumente danach, ob sie *well-formed* sind. Im Microsoft Internet Explorer 5 (aber auch in anderen Browsern, wie etwa Opera 5) ist ein XML-Parser eingebaut. Wenn wir unser ordentliches XML-Dokument in den Internet Explorer laden, zeigt er den Baum der Elemente des Dokuments an, wie wir in Abbildung 4 sehen.

Wenn wir einen Fehler in unsere kleine XML-Apotheke einbauen, indem wir schreiben

```
<medikament zugang=a>...
```

dann ist das Dokument nicht mehr ordentlich und der Parser im Internet Explorer meckert — zurecht! Siehe Abb. 5.

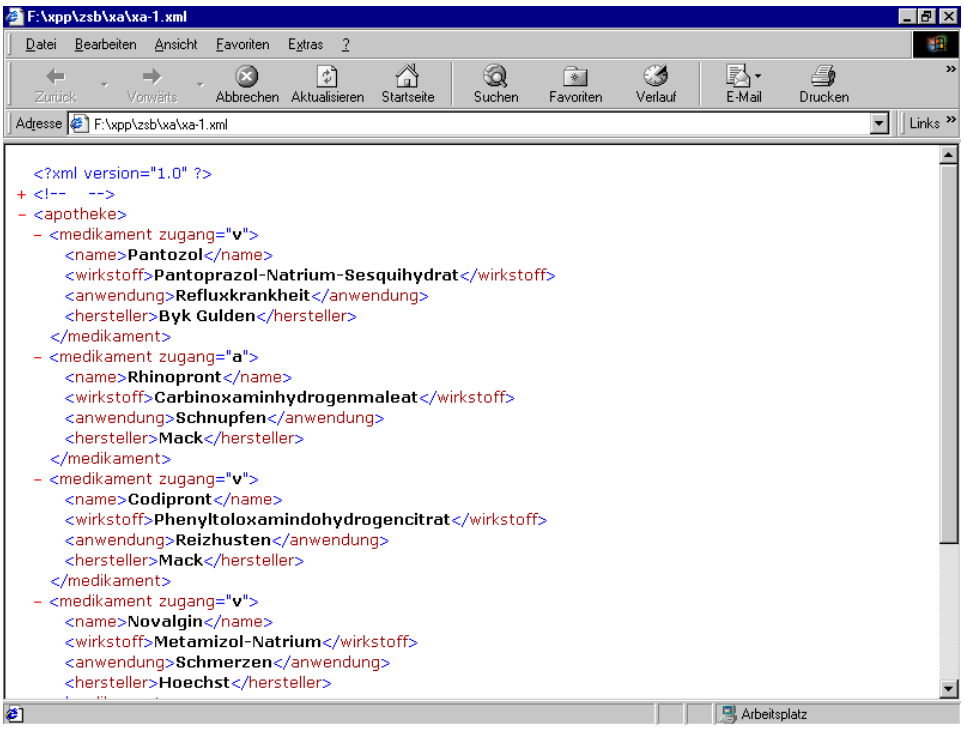

Abbildung 4: XML-Dokument im Browser

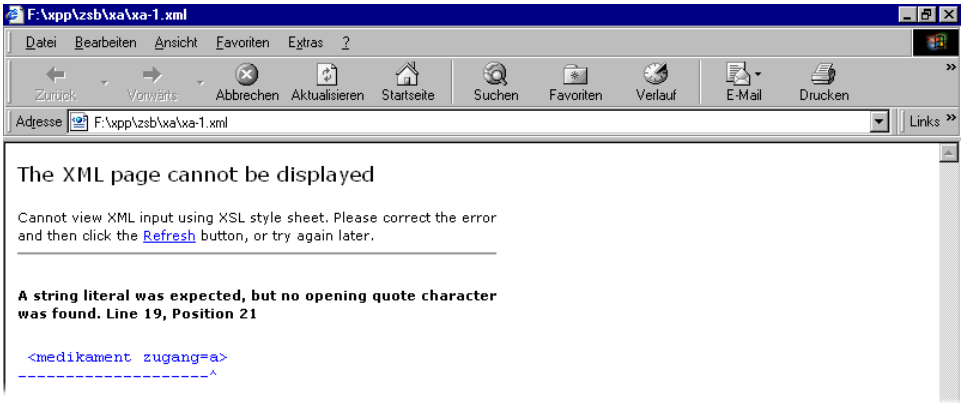

Abbildung 5: Fehler im XML-Dokument!

1.3 Die Document Type Definition und gültige XML-Dokumente

Kommen wir auf unser Beispiel mit dem Lieferanten von Medikamenten zurück: wir haben gesehen, wie wir eine Liste von Medikamenten erstellen können, wir können sie einem Partner zuleiten und er kann sie in seinem Browser ohne weitere Umstände verwenden. Allerdings legen die Regeln

für ordentliche Dokumente nichts, aber auch gar nichts über den Inhalt des XML-Dokumentes fest. Alle 10 Regeln machen nur eine Aussage, wie das Dokument syntaktisch aufgebaut ist — aber kein Wort darüber, welche Elemente in welcher Kombination und Schachtelung vorkommen dürfen; keine Festlegung der „Grammatik“ unserer Sprache.

Das ist durchaus gewollt: anders als HTML, wo eine feste Menge von Tags fixiert ist, ist XML eine Meta-Sprache: wir können für den jeweiligen Anwendungszweck selbst definieren, welchen Inhalt unsere Klasse von XML-Dokumenten darstellen soll. Man spricht von *XML-Anwendungen* oder auch *XML-Sprachen*. Die Struktur, die wir unserer kleinen Apotheke gegeben haben, ist also eine XML-Anwendung.

Haben wir nun eine solche Struktur mit unserem Lieferanten vereinbart, dann wird der Kontakt nur funktionieren, wenn wir beide uns auch daran halten. In der *Document Type Definition*, kurz DTD, kann man festschreiben, wie eine Klasse von XML-Dokumenten aufgebaut sein muss. Und man kann einen XML-Parser verwenden, um zu überprüfen, ob diese DTD von einem konkreten XML-Dokument auch tatsächlich eingehalten wird.

Bei XML-Dokumenten, die einer DTD entsprechen und sie einhalten, spricht man von *gültigen XML-Dokumenten*: Sie sind gültig in Bezug auf die DTD, nach der sie sich richten.

Am besten machen wir uns eine DTD für unsere kleine Apotheke. Dazu schreiben wir eine Datei `xa.dtd`:

```
<!ELEMENT apotheke      (medikament+)>
<!ELEMENT medikament    (name, wirkstoff?, anwendung+, hersteller )>
<!ATTLIST medikament    zugang (f | a | v) 'f'>
<!ELEMENT name          (#PCDATA)>
<!ELEMENT wirkstoff     (#PCDATA)>
<!ELEMENT anwendung     (#PCDATA)>
<!ELEMENT hersteller    (#PCDATA)>
```

In der DTD werden die erlaubten Elemente festgelegt:

- ihre Gruppierung mit `"()"`,
- ihre Reihenfolge mit `","`,
- ihre Anzahl:
  1 (ohne weitere Angabe),
  0 oder 1 (mit `"?"`),
  1 oder mehr (mit `"+"`) oder
  0 oder mehr (mit `"*"`).
- Man kann mit dem Symbol `"|"` auch Wahlmöglichkeit zwischen Elementen erlauben.

Außerdem legt man in der DTD die Attribute eines Elements fest, in unserem Beispiel kann das Attribut `zugang` von `medikament` die angegebenen Werte haben, wobei der Wert `"f"` automatisch gilt, wenn kein anderer angegeben ist.

Die Bezeichnung `#PCDATA` steht für *parsed character data*, also für den Inhalt unserer Elemente, den wir als Text angeben.

Nun müssen wir in unserem XML-Dokument auch noch den Bezug auf die DTD eintragen. Dazu ergänzen wir unser Dokument um die folgende Zeile, die im *Prolog* des Dokuments auf die XML-Deklaration folgt:

```
<!DOCTYPE apotheke SYSTEM "xa.dtd">
```

Natürlich gibt es auch eine Möglichkeit, eine DTD zu verwenden, die man nicht selbst erstellt hat, sondern die im Internet zur Verfügung steht. In der Regel ist dies der eigenen Definition einer DTD ohnehin vorzuziehen: mit der Durchsetzung von XML haben sich eine ganze Reihe von Standards für bestimmte Klassen von XML-Dokumenten entwickelt, oder sie wurden aus der SGML-Welt übernommen. *Der* Einstieg in die Welt der XML-Anwendungen ist `http://www.oasis-open.org`, siehe [7]. Erwähnenswert als wichtige Exempel von XML-Sprachen sind etwa die *CML Chemical Markup Language* (`http://www.xml-cml.org`) und die *Mathematical Markup Language (MathML)*, die ihre Homepage bei `http://www.w3.org/Math/` hat.

Ein validierender XML-Parser prüft also nicht nur, ob ein Dokument ordentlich ist, sondern auch seine Übereinstimmung mit der DTD. Wenn wir unser Dokument verändern, so dass es nicht mehr der Definition entspricht, etwa indem wir ein zusätzliches Element `<preis>` einbauen, dann sollte ein validerender Parser es nicht mehr akzeptieren. Aber ach: Internet Explorer 5 beachtet die DTD nicht. Verwenden wir also einen anderen, wirklich validierenden Parser: *xerces*. Um präzise zu sein: wir verwenden *xalan*, den Prozessor für XSL-Transformationen, und lassen ihn eine triviale Transformation durchführen, er soll einfach die XML-Datei lesen und unverändert wiedergeben. *xalan* nun verwendet *xerces*. Beide Werkzeuge sind Teil des Apache XML Projekts [1].

Die Meldungen beim Parsen des gültigen Dokuments `xa-2.xml` und des nicht gültigen Dokuments `xa-2f.xml` sind in der Abbildung 6 wiedergegeben.

1.4 XML-Parser: DOM und SAX

Das Web Consortium hat nicht nur XML standardisiert, sondern auch das *XML Document Object Model*, kurz DOM. Wir haben gesehen, dass jedes XML-Dokument ein Baum ist, und so beschreibt das DOM nichts anderes als einen Baum von Objekten, sowie die Zugriffe auf die verschiedenen Elemente des Baumes und Methoden zum Traversieren des Baumes.

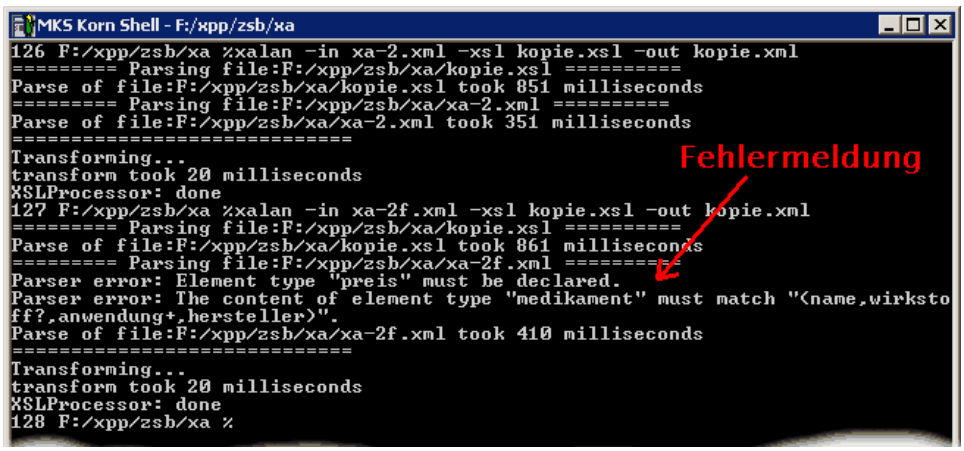

Abbildung 6: Validierendes Parsen mit *xalan* und *xerces*

Das DOM ist also eine standardisierte Schnittstelle zur Repräsentation eines XML-Dokuments als Baum. Die Standardisierung des *Document Object Models* ist von wesentlicher Bedeutung für den Erfolg von XML: indem der Zugriff auf XML-Dokumente, das API für die Manipulation von XML-Dokumenten standardisiert wird, können Werkzeuge kooperieren, wird der Standard wirklich offen und für jeden verwendbar.

Wie arbeitet also ein XML-Parser? Er liest die XML-Datei und baut im Speicher den Baum der Objekte auf. Dazu verwendet er das DOM. Diesen Typ von Parser nennt man auch *DocumentBuilder*. Nun kann das generierte Modell Ausgangspunkt für jede Art Transformationen sein. Andere Werkzeuge können den Parser verwenden, weil und insofern er die standardisierte Schnittstelle implementiert.

Wir stellen uns das Ergebnis eines XML-Parsers vor als die Speicherrepräsentation des XML-Baums, wie wir ihn in Abbildung 2 gesehen haben.

Die Methoden, mit denen im DOM auf den Baum zugegriffen wird, sind im wesentlichen Methoden des Interface *Node*. Ferner gibt es Interfaces wie *Document*, *Element*, *Text* oder *Attr*, die *Node* um spezielle Methoden des jeweiligen Typs von Knoten erweitern. Nennen wir einige Methoden von *Node*, sie geben einen Eindruck, wie die Schnittstelle „gestrickt ist“:

- *getNodeName()*,
  getNodeValue(),
  getNodeType()
- *getParentNode()*,
  getChildNodes(),
  getFirstChild(),
  getNextSibling()

- *insertBefore(Node newChild, Node refChild),*
  replaceChild(Node newChild, Node oldChild)
  removeChild(Node oldChild)
  appendChild(Node newChild)
- usw. usf.

Für viele Zwecke ist die Verwendung des DOMs zu aufwändig, oft genügt es für den Einsatz von XML-Dokumenten, dass man die XML-Datei einfach durchläuft und ihren Inhalt verwertet: man traversiert den Baum in der Reihenfolge der Knoten, wie sie in der XML-Datei gegeben ist. Eine Schnittstelle für diese Technik ist SAX, *The Simple API to XML*, von David Megginson. [1] Der *SAXParser* erzeugt beim Durchlaufen der XML-Datei Ereignisse und stößt Methoden an, die auf diese Ereignisse reagieren.

Man verwendet einen SAX-Parser, in dem man eine Klasse schreibt, die die Klasse `DefaultHandler` erweitert. Die Grundidee besteht einfach darin, den Polymorphismus ojektorientierter Sprachen zu benutzen. Der DefaultHandler hat Methoden, die vom SAX-Parser an bestimmten Stellen beim Durchlaufen des XML-Dokuments „angesprungen“ werden (wir sehen gleich welche Stellen das sind) — wir implementieren nun in unserer Ableitung von DefaultHandler die von uns gewünschte Funktionalität an der entsprechenden Stelle und – voilà!

Folgende Abbildung 7 zeigt, welche Ereignisse der SAXParser „feuert“.

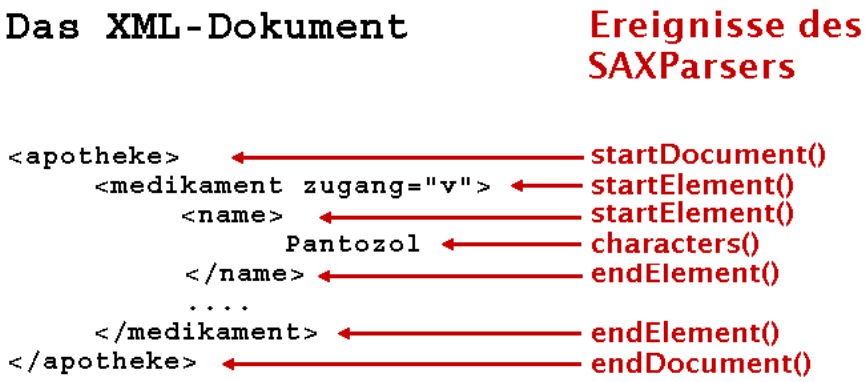

Abbildung 7: Methoden des SAXParsers

[1] Nicht unerwähnt soll bleiben, dass David Megginson auch ein lehrreiches Buch über die Strukturierung von Dokumenten verfasst hat: David Megginson: Structuring XML Documents, Prentice Hall 1998.

Schön und gut, wird der geneigte Leser vielleicht denken. Aber wir wollen den Inhalt unserer XML-Dokumente ja auch darstellen, präsentieren. Jetzt ist die *Gestaltung* von *Inhalt* und *Struktur & Semantik* getrennt, aber wo bleibt sie dann? Das Rezept heißt XSL:

2 XSL – Transformation und Gestaltung

2.1 CSS – Cascading Style Sheets

Die einfachste Art, eine XML-Datei im Browser darzustellen, sind *Cascading Style Sheets*, wie man sie auch mit HTML einsetzen kann.

Eine CSS-Datei definiert die Gestaltung der einzelnen Elemente einer XML-Datei. In bereits zwei Ausprägungen des Standards (CSS1 und CSS2) können Eigenschaften der Textdarstellung, Schriftarten, Farben, Rahmen eingestellt werden. (Eine umfassende Darstellung findet man in [5].) Uns interessieren in diesem Zusammenhang nicht so sehr die vielen Möglichkeiten der Gestaltung, sondern die Art und Weise, wie ein Cascading Style Sheet einer XML-Datei zugewiesen werden kann. Dies geschieht durch die Anweisung `<?xml-stylesheet...>` im Prolog der XML-Datei. Man schreibt etwa:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/css" href="xa.css"?>
...
```

Damit wird einem Programm, das in der Lage ist, eine XML-Datei mit Cascading Style Sheets darzustellen, die Information über die Präsentation der Elemente gegeben. Ein Beispiel für eine CSS-Datei, die die kleine XML-Apotheke darstellt, könnte etwa so aussehen:

```
apotheke
{
    background-color: #ffffff;
    width: 100%;
    font-family: Verdana, sans-serif;
}
medikament
{
    display: block;
    margin-bottom: 30pt;
    margin-left: 0;
}
name
{
    display: block;
    color: #FF0000;
    font-size: 16pt;
}
wirkstoff
{
    color: #0000FF;
    font-size: 12pt;
```

```
    margin-left: 12pt;
}
anwendung, hersteller
{
    display: block;
    color: #000000;
    margin-left: 12pt;
}
```

Das Ergebnis dieses (zugegebenermaßen extrem simplen Stylesheets) wird im Browser so (Abb. 8) dargestellt. Zur Abwechslung verwenden wir mal Opera 5 als Browser.

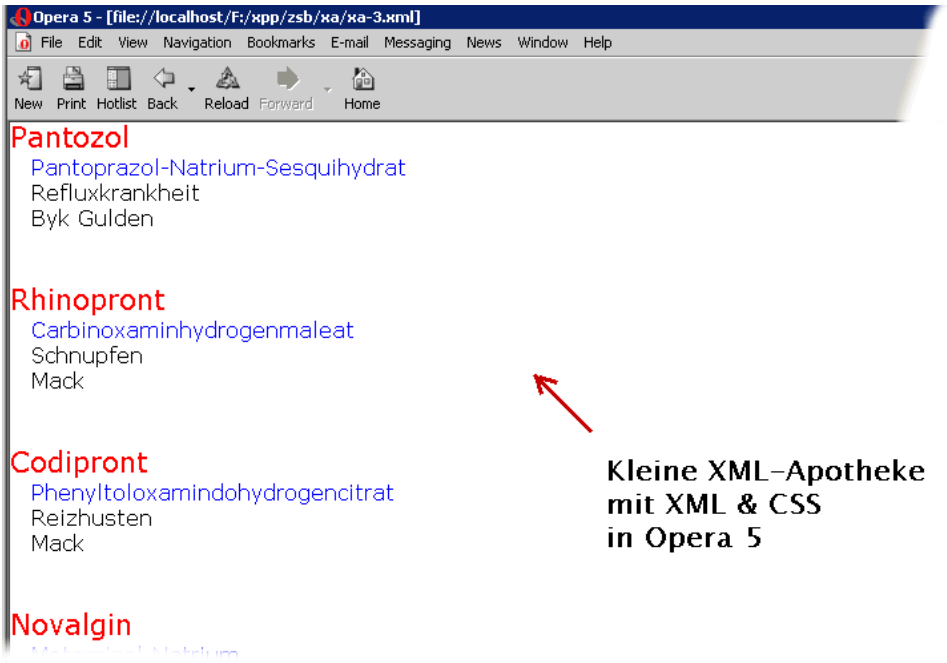

Abbildung 8: Gestaltung der kleinen Apotheke mit CSS

Wichtig ist diese Beobachtung: Die Struktur des XML-Dokuments wurde durch das Cascading Style Sheet nicht verändert. Durch die CSS-Datei wird lediglich festgelegt, welche Präsentation für die jeweiligen Elemente gewählt werden soll. Cascading Style Sheets können deshalb nur in einem Kontext verwendet werden, in dem keine Manipulationen an Struktur und Inhalt eines XML-Dokuments beabsichtigt oder notwendig sind.

Anders ist es mit XSL, der *eXtensible Stylesheet Language*: Der erste Teil des Standards, kurz *xslt* genannt, befasst sich mit dem Thema: wie kann man die *Transformation* von XML-Dokumenten in andere Dokumente, etwa wieder XML-Dokumente, aber auch andere Formate steuern? Gehen wir also weiter zu:

2.2 XSL-Transformationen – Webseiten mit XML und XSL

In diesem Abschnitt wollen wir eine Webseite in HTML erzeugen, in dem wir eine XSL-Transformation auf die XML-Datei mit unserer kleinen Apotheke anwenden. Beginnen wir doch gleich mit unserem Beispiel, es zeigt dann gleich, wie XSL-Transformationen funktionieren.

Die XSL-Transformation wird definiert in einer XSL-Datei, die in unserem Beispiel so aussieht:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output indent="yes" method="html"/>

<xsl:template match="/">
    <html>
        <head>
            <title>Die kleine XML-Apotheke</title>
        </head>
        <body>
            <h1>Die kleine XML-Apotheke</h1>
            <table width="720">
                <xsl:apply-templates/>
            </table>
        </body>
    </html>
</xsl:template>

<xsl:template match="medikament">
    <tr>
        <td><xsl:number/></td>
        <xsl:apply-templates/>
    </tr>
</xsl:template>

<xsl:template match="name | wirkstoff | anwendung | hersteller">
    <td><xsl:value-of select="."/></td>
</xsl:template>

</xsl:stylesheet>
```

Wir können mit dem XSL-Prozessor *xalan* diese XSL-Datei auf unser XML-Dokument anwenden und erhalten dadurch eine HTML-Datei. Der Aufruf von *xalan* lautet:

```
xalan -in xa-1.xml -xsl xa-html.xsl -out xa.html
```

Und das Ergebnis ist folgende HTML-Datei:

```
<html>
    <head>
        <title>Die kleine XML-Apotheke</title>
    </head>
    <body>
        <h1>Die kleine XML-Apotheke</h1>
```

```
    <table width="720">
    <tr>
        <td>1</td>
        <td>Pantozol</td>
        <td>Pantoprazol-Natrium-Sesquihydrat</td>
        <td>Refluxkrankheit</td>
        <td>Byk Gulden</td>
    </tr>
    <tr>
    usw...
    usw...
    usw...
    </tr>
    </table>
    </body>
</html>
```

Im Browser sehen wir dann folgendes Bild:

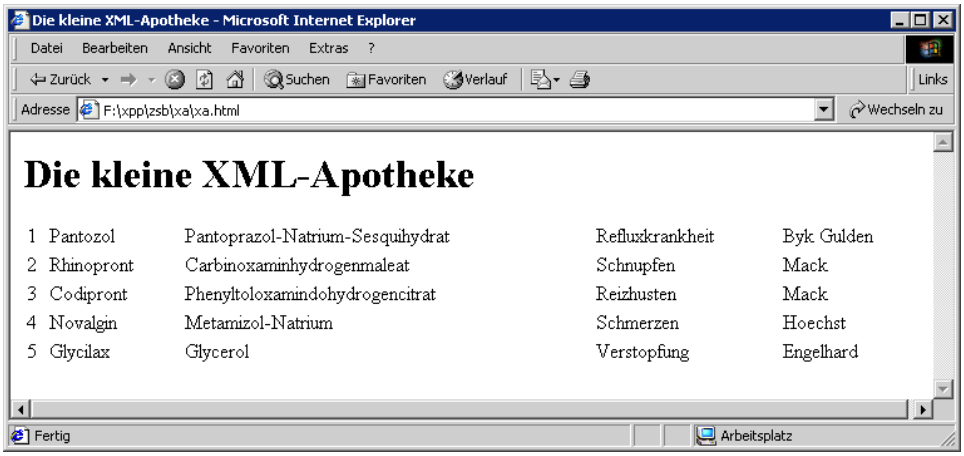

Abbildung 9: Die kleine Apotheke — mit *xslt* erzeugt

Im Ergebnis: Durch die Transformation, die in der XSL-Datei definiert wurde, hat der XSL-Prozessor eine neue Datei erzeugt, eine HTML-Datei, die nicht mehr die „reine“ XML-Information (also Inhalt samt Struktur & Semantik) beinhaltet, sondern *die* Information, die zur Darstellung, zur *Präsentation*, benötigt wird, zum Beispiel im Format HTML.

Mit dieser Technik ist es leicht möglich, aus Informationen, die als XML-Dateien vorliegen, statische Webseiten zu erstellen. Ein Beispiel für eine ganze Web-Site, die mit dieser Technik erzeugt wird, ist in [2] dargestellt. Meine eigene Website an der FH Gießen-Friedberg mache ich auch mit dieser Methode.

Analyse des Beispiels

Wie funktioniert dieses kleine Beispiel? Gehen wir die XSL-Datei Schritt für Schritt durch:

Sie beginnt mit der bekannten XML-Deklaration. Das bedeutet, dass die XSL-Datei selbst eine XML-Datei ist. Darauf folgt das Wurzelelement des Stylesheets mit der Angabe des Namensraums, was bedeutet, dass in unserer XSL-Datei das Präfix `xsl` für Elemente verwendet wird, die in der Spezifikation des W3C für XSL-Transformationen Version 1.0 standardisiert wurden. Die darauf folgende Zeile legt fest, wie die Ausgabe der Transformation formatiert werden soll.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output indent="yes" method="html"/>
```

Interessant ist der eigentliche Inhalt der XSL-Datei. Typisch ist die Zeile

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

Ein Element `<xsl:template>` definiert eine Regel, die von der XSL-Transformation durchgeführt wird, wenn der Teil der Eingabedatei bearbeitet wird, der durch das Attribut `match=...` angegeben wird. Die erste Regel enthält das Attribut `match="/"`, das das Wurzelelement der Eingabedatei bezeichnet. Also wird das folgende Stück Transformation ausgeführt, wenn das Wurzelelement der Eingabedatei gefunden wird, in unserem Beispiel also beim Element `<apotheke>`.

Da wir aus der XML-Eingabedatei eine HTML-Datei erzeugen wollen, geben wir an:

```
<xsl:template match="/">
    <html>
        <head>
            <title>Die kleine XML-Apotheke</title>
        </head>
        <body>
            <h1>Die kleine XML-Apotheke</h1>
            <table width="720">
                <xsl:apply-templates/>
            </table>
        </body>
    </html>
</xsl:template>
```

Wir schreiben die äußeren Tags der HTML-Datei, geben ihr einen Titel, setzen als Überschrift „Die kleine XML-Apotheke“ ein und machen eine Tabelle. Und wie geht es weiter? Das Geheimnis liegt in der Zeile `<xsl:apply-templates/>`. Dadurch wird der XSL-Prozessor angewiesen, nun alle Kinder des Wurzelelementes zu durchlaufen und die Regeln anzuwenden, die auf sie zutreffen. Die Regeln sind in der Folge definiert:

```
<xsl:template match="medikament">
    <tr>
        <td><xsl:number/></td>
```

```
        <xsl:apply-templates/>
    </tr>
</xsl:template>

<xsl:template match="name | wirkstoff | anwendung | hersteller">
    <td><xsl:value-of select="."/></td>
</xsl:template>
```

Findet der XSL-Prozessor ein Element mit dem Namen `medikament`, dann erzeugt er eine neue Zeile in der HTML-Tabelle und setzt die aktuelle Nummer des Elements ein (`<xsl:number>`). Wenn bei der Transformation ein Element namens `name` oder `wirkstoff` usw. gefunden wird, dann wird eine neue Zelle in der HTML-Tabelle eingesetzt und der Inhalt des Elements in diese Zelle geschrieben (`<xsl:value-of select="."/>`). Und das war schon die ganze Transformation.

2.3 XSL-Transformationen – Das Prinzip

Das Prinzip der Arbeitsweise von XSL-Transformationen *xslt* besteht darin, ein Quell-Dokument zu lesen, die Regeln des XSL Stylesheets auf den Baum des Quell-Dokuments anzuwenden und daraus einen neuen Baum, den Ergebnisbaum zu konstruieren. Aus ihm wird schließlich das Ergebnis-Dokument produziert. Abbildung 10 veranschaulicht das Prinzip.

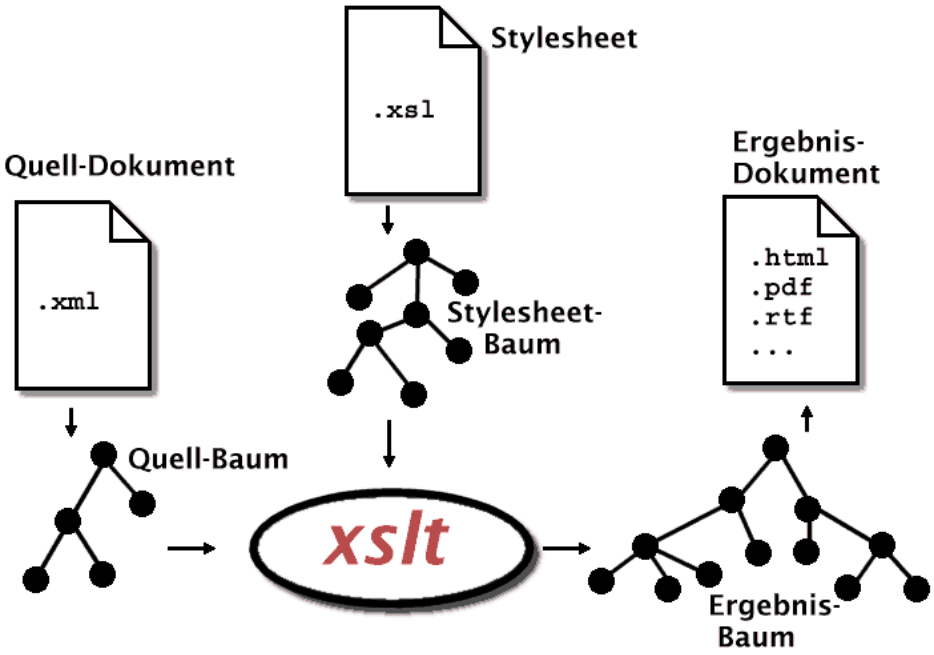

Abbildung 10: Prinzip der XSL-Transformation

xslt hat als Sprache folgende Charakteristika:

- *xslt* ist selbst eine **XML-Anwendung**, XSL Stylesheets werden in XML geschrieben. Dadurch kann der XML-Parser auch für Transformationen verwendet werden, ein erster Beweis für die Wiederverwendung von Werkzeugen der XML-Welt. Es scheint vielleicht zunächst merkwürdig zu sein, eine Programmiersprache in XML zu schreiben, hat man jedoch die nächste Eigenschaft von *xslt* im Auge, kommt man schnell damit zurecht.

- *xslt* ist eine **regelbasierte Sprache** [2]. Typischerweise besteht ein Stylesheet aus einer Folge von Regeln, die durchgeführt werden, wenn bei der Bearbeitung des Eingabebaums eine der Regeln zutrifft. So gesehen ist *xslt* eine deklarative Sprache, mit der festgelegt wird, welcher Ausgabetext erzeugt werden muss, wenn ein bestimmtes Muster in der Eingabedatei auftritt — im Unterschied zu einem sequenziellen Programm, das festlegt, dass Aktionen in einer bestimmten Reihenfolge durchgeführt werden sollen.

- *xslt* ist konzipiert als eine Sprache, die **keine Seiteneffekte** haben soll. Gemeint ist damit, dass eine Funktion keine Veränderung an der Umgebung vornimmt, so dass es möglich ist, die Funktion in jeder beliebigen Situation aufzurufen. Der Gedanke an dieser Forderung beim Design von *xslt* ergibt sich aus dem vorherigen Punkt: es sollte eine deklaratorische Sprache sein. Dabei gab es bei den Entwicklern der Spezifikation sicherlich auch diesen Gesichtspunkt: Transformiert XSL eine aus dem Web übertragene XML-Datei, dann sollte es möglich sein, die Transformation zu beginnen, ehe das gesamte Dokument bekannt ist, d.h. aber eine später übertragene Stelle darf keine Rückwirkung mehr haben, also keinen Seiteneffekt verursachen. Praktisch tritt diese Eigenschaft beim Arbeiten mit *xslt* dadurch auf, dass man den Wert einer Variablen nicht ändern kann.

xslt ist eine ausgewachsene Sprache, wir können hier nur einen Überblick über ihre Elemente geben. Man bekommt durch diese Übersicht aber schon einen Eindruck, wie mächtig *xslt* ist. Für eine umfassende Darstellung siehe [4].

xslt Elemente

Die wichtigsten Elemente von *xslt* kann man in folgende Gruppen einteilen.

- Definition von Regeln und die Art, wie die Regeln ausgeführt werden:

[2] Für die Denkweise bei der Entwicklung von XSL-Transformationen ist es hilfreich, wenn man schon mal mit *awk* gearbeitet hat. Man kann sagen: *xslt* ist ein *awk* für Bäume.

<xsl:template>,
<xsl:apply-templates> und
<xsl:call-template>.

- Elemente, die Einträge in der Ausgabedatei erzeugen, wie
<xsl:value-of>, <xsl:text>
<xsl:element> und <xsl:attribute>, auch
<xsl:copy> und
<xsl:copy-of> zum Kopieren ganzer Teilbäume.

- Variablen und Parameter:
<xsl:variable>,
<xsl:param> und <xsl:with-param>.

- Elemente zur Steuerung von Bedingungen und Schleifen
<xsl:if>, <xsl:choose> und
<xsl:for-each>.

- Elemente zur Nummerierung und zum Sortieren
<xsl:number> und <xsl:sort>.

Ausdrücke und XPath

Innerhalb eines XSL Stylesheets ist es häufig notwendig, bestimmte Elemente im Baum des Eingabe-Dokumentes anzusprechen. Für diese Ausdrücke wurde XPath vom W3C standardisiert, eine eigene Sprache, um innerhalb des Dokument-Baums navigieren zu können. Mit XPath kann man Elemente auf verschiedene Weise benennen, zum Beispiel

- durch explizites Benennen des Namens von Elementen oder von Attributen, wie etwa in einem Ausdruck der Art
<xsl:value-of select="wirkstoff"/>,

- durch die Angabe einer Bedingung, die ein Element erfüllen muss, etwa
<xsl:if test="$var=1">,

- durch die Angabe einer Position eines Elements im Baum, etwa eines *Vaters*, von *Kindern* oder von *Geschwistern*; ein Beispiel dafür ist
<xsl:value-of select=".."/> (Wert des Vaters).

xslt Muster

Regeln werden von xslt angewandt, wenn das zugehörige Muster zutrifft. Was ist nun ein Muster? Es ist ein Ausdruck, den ein Element des Eingabe-Baums erfüllen muss. Beispiele sagen mehr als viele Worte:

- `<xsl:template match="abstract"/>`
  definiert ein Muster, das auf alle Element zutrifft, die den Namen `<abstract>` haben.
- `<xsl:template match="section/para[1]"/>`
  trifft alle Elemente namens `<para>`, die als erstes Kind innerhalb von `<section>` vorkommen.
- `...*[@width]`
  trifft auf alle Elemente zu, die ein Attribut `width` haben.
- `...text()[starts-with(., 'Byk')]`
  alle Elemente, deren Inhalt mit den Buchstaben `Byk` beginnt.
- und viele andere mehr, siehe [4].

xslt Funktionen

Auch was die Funktionen von xslt betrifft, können wir nur andeuten, was mit dieser Sprache alles möglich ist. Es gibt Funktionen für folgende Gebiete:

- Typumwandlungen, wie `format-number()`
- Arithmetik, wie `round()`
- Manipulation von Zeichenketten (String-Funktionen), wie `substring()`
- Aggregration von Werten, wie `sum()`
- Funktionen, wie `position()`, die auswerten, wo im Kontext des Baums sich ein Element befindet
- und viele weitere

2.4 XSL-Transformationen – Weitere Beispiele

In diesem Abschnitt wollen wir noch zwei weitere Beispiele geben, was man mit XSL produzieren kann: Die kleine Apotheke in *pdf* als Exempel für die Verwendung von XSL-Transformationen beim Elektronischen Publizieren. Und eine ganz anders gelagerte Anwendung: die Berechnung der Züge eines Pferds auf einem Schachbrett. Alles nur die Spitze des *xslt*-Eisbergs!

pdf via *xslt*

Für diesen Weg gibt es verschiedene Techniken, die allerdings allesamt zum jetzigen Zeitpunkt noch als "experimentell" bezeichnet werden müssen:

- **fop** ist Teil des bereits erwähnten XML Apache Projekts [1]. Das in Java geschriebene Programm präsentiert eine XML-Datei im Format *xsl fo* als *pdf*-Datei. Diesen Weg werden wir im nächsten Abschnitt noch genauer betrachten.
- **passivetex** ist ein Paket von Sebastian Rahtz [8], der das Satzsystem TEX verwendet, um aus einer Datei im Format *xsl fo pdf* zu erzeugen.
- Das Programm **ufo** geht einen ähnlichen Weg wie Rahtz. (`http://www.unicorn-enterprises.com`)

Wir machen es uns viel einfacher: wir erzeugen einfach aus unserer XML-Datei eine TeX-Datei und nehmen dann dieses Satzsystem, um die *pdf*-Datei zu erhalten:

```
xalan -in xa-1.xml -xsl xa-pdftex.xsl -out XML-Apotheke.tex
texify -p XML-Apotheke.tex
```

Dies ist das Ergebnis: Abb. 11.

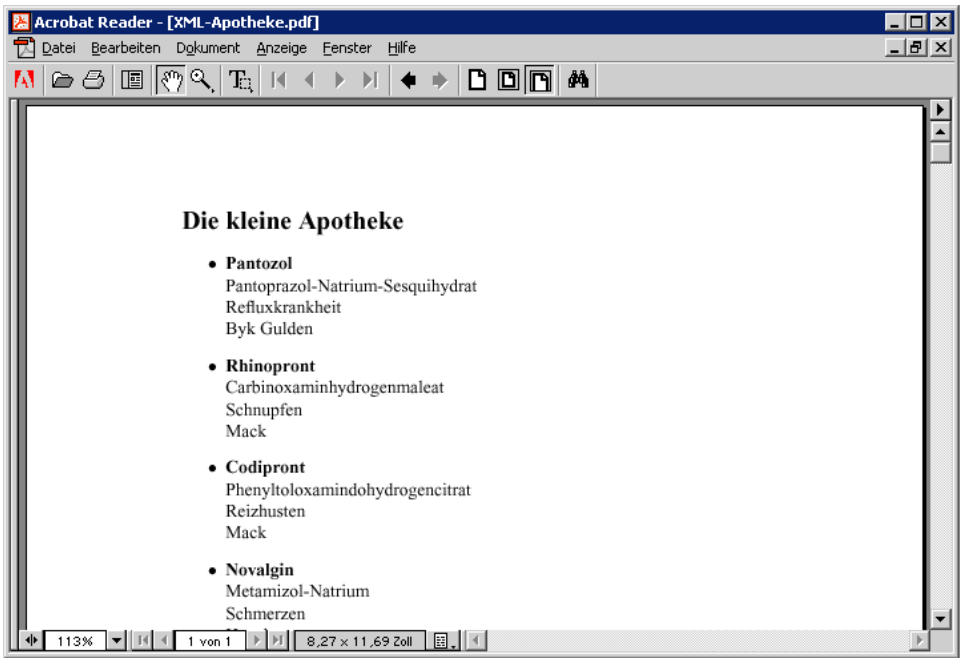

Abbildung 11: Dokument nach *xslt* und TEX

Die Tour des Springers

Das Beispiel ist von Michael Kay [4] und demonstriert die Möglichkeiten von *xslt* als Programmiersprache.

Worum geht's? Der Springer bewegt sich von einem Ausgangsfeld auf dem Schachbrett — und das „Stylesheet" soll einen Weg berechnen, so dass er auf jedem Feld vorbeikommt. Starten wir doch einfach Michael Kays `tour.xsl` mit dem Startfeld `a8`:

```
xalan -xsl tour.xsl -param start "'a8'" -out tour.html
```

Das Ergebnis sieht im Internet Explorer wie in Abbildung 12 aus. Feld `a8` ist in der linken oberen Ecke und trägt die Nummer `01`; wir können den Nummern folgend den Weg des Springers verfolgen.

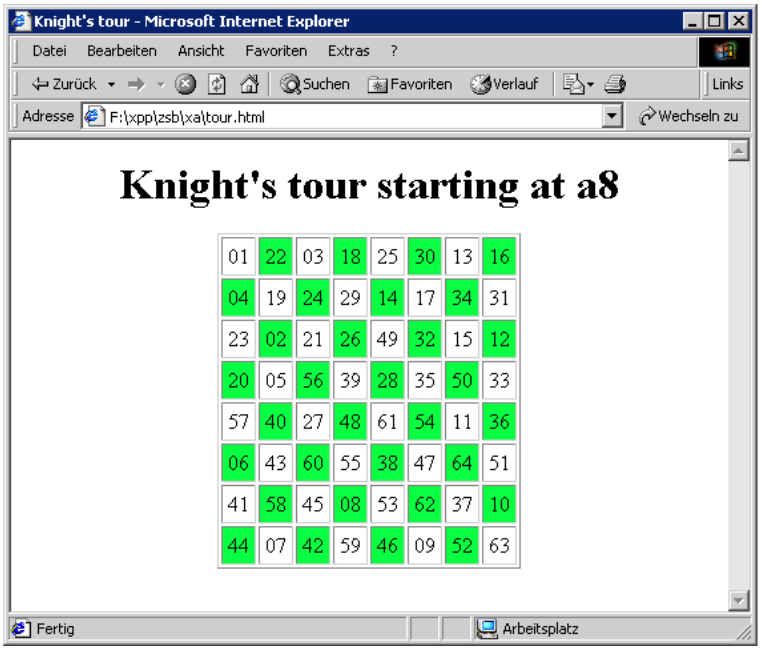

Abbildung 12: Die Tour des Pferds

2.5 Die Perspektive: XSL Formatting Objects

XSL-Transformationen sind „nur" der erste Teil der Spezifikation der Extensible Stylesheets. Der zweite Teil sind die XSL Formatting Objects, die plattform- und medienunabhängig die Gestaltung eines Dokuments beschreiben.

Die Idee: Der *Inhalt* und die *semantische Struktur* der Daten sind im XML-Dokument enthalten. Ein Stylesheet bestimmt, welche Präsentation der Information gewählt werden soll und erzeugt durch eine XSL-Transfor-

mation ein neues XML-Dokument: dieses enthält nun nicht mehr den ursprünglichen Inhalt, sondern die *Gestaltung*, die *Layout-Struktur* der Information. Und die XML-Sprache für genau *diese* Information sind XSL Formatting Objects.

Folgende Abbildung 13 aus [11] veranschaulicht den Prozess.

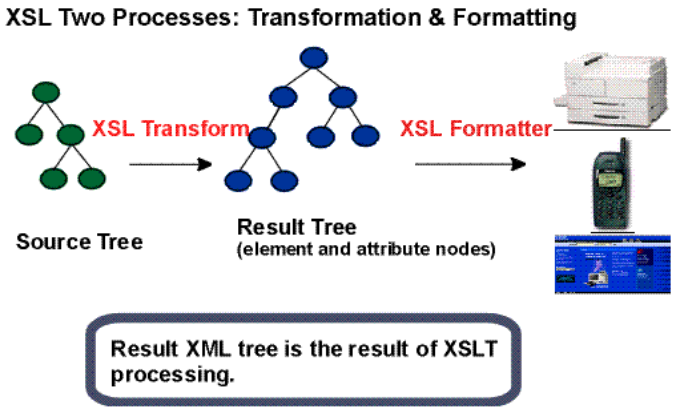

Abbildung 13: Die Idee von *xsl fo* (Formatting Objects)

3 XML und Datenbanken

3.1 Einfache Transformationen: Von XML in die Datenbank und zurück

XML und Datenbanken bilden eine überaus naheliegende Kombination:

- Viele Informationen, die man im Internet verwenden möchte, liegt in Datenbanken, typischerweise relationalen Datenbanken vor. Es handelt sich in der Regel um *datenzentrierte* Informationen, die als Zeilen in Tabellen gesehen werden können.
- Die klassische *dokumentenzentrierte* Sicht von Informationen, wie sie insbesondere im traditionellen Publizieren gang und gäbe ist, bleibt zwar bestehen — XML ist ja auch eine Auszeichnungssprache für Dokumente; die Geschlossenheit der Dokumente löst sich aber auf. Teile von Dokumenten sollen in verschiedenen Kombinationen für unterschiedliche Medien verwendet werden. Nichts naheliegender, als solche (Teil-)Dokumente in Datenbanken zu speichern und abrufbar zu halten.
- Aber auch in der Sicht auf XML-Dokumente ist ein Zusammenhang zu Datenbanken offensichtlich: Man möchte Abfragesprachen haben,

die (ähnlich wie *SQL*) Informationen aus XML-Dokumenten extrahieren. Zu diesem Thema gibt es Standardisierungsbestrebungen beim W3C.

Wir wollen folgendes Szenario durchspielen: Die kleine XML-Apotheke wächst und wächst, immer mehr Medikamente werden verzeichnet und sollen dann wieder — in verschiedenen Formen und für verschiedene Medien — präsentiert werden. Deshalb wollen wir eine relationale Datenbank einsetzen, um die Daten über Medikamente zu verwalten.

Bauen wir uns also eine ganz einfache Datenbank:

```
create table medikament
(
    mid         integer primary key,
    name        varchar(80) not null,
    zugang      char(1),
    wirkstoff   varchar(80),
    anwendung   varchar(80),
    hersteller  varchar(80),
)
```

XML-Dokument in die Datenbank transferieren

Da wir schon mal mit der XML-Datei für die kleine XML-Apotheke begonnen haben, möchten wir das XML-Dokument in die Datenbank bringen. Wie vorgehen? Idee?

Klar: wir schreiben uns eine XSL-Transformation, die aus dem XML-Dokument ein SQL-Skript erstellt. Das SQL-Skript enthält pro Medikament eine `insert`-Anweisung. Nach dem, was wir bisher über *xslt* gelernt haben, ist es eine einfache Übung:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output indent="no" method="text"/>

<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="medikament">
insert into medikament( mid, name, zugang, wirkstoff,
    anwendung, hersteller )
values ( <xsl:number/>,
    '<xsl:value-of select="name"/>',
    '<xsl:value-of select="@zugang"/>',
    '<xsl:value-of select="wirkstoff"/>',
    '<xsl:value-of select="anwendung"/>',
    '<xsl:value-of select="hersteller"/>');
</xsl:template>

</xsl:stylesheet>
```

In der Tat, die Transformation

```
xalan -in xa-2.xml -xsl xa-sql.xsl -out xa-fill.sql
```

erzeugt ein SQL-Skript, das den Inhalt der XML-Datei in die Datenbank übertragt. Wir überzeugen uns:

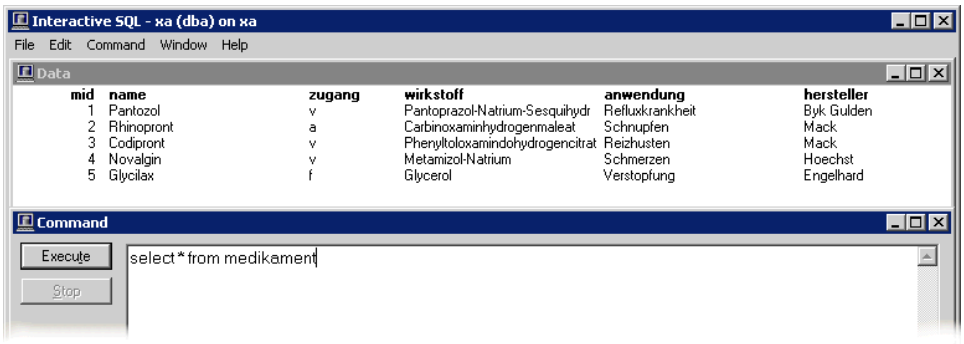

Abbildung 14: Inhalt der Tabelle Medikamente nach der Übertragung

XML-Dokumente aus SQL-Ergebnissen erstellen

Was so schön in der einen Richtung funktioniert hat, muss doch auch umgekehrt gehen: Produzieren wir aus einer Ergebnismenge einer SQL-Anweisung ein XML-Dokument. Studenten der Fachhochschule Gießen[3] haben im Praktikum Softwaretechnik ein Java-Programm geschrieben, das diese Aufgabe erledigt. Die Idee: wir verwenden eine XML-Datei für die Festlegung der Datenquelle und der SQL-Anweisung, außerdem steht dort, wie die Ausgabe strukturiert werden soll. Eine solche Datei könnte etwa so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsInfo>
    <DBInfo>
        <DBUrl>jdbc:odbc:xa</DBUrl>
        <AccountName>dba</AccountName>
        <AccountPassword>sql</AccountPassword>
    </DBInfo>
    <SelectStatement>SELECT hersteller, name
            FROM medikament ORDER BY hersteller
    </SelectStatement>
    <GroupSchema>
        <Group tagname="Apotheke">
            <Group tagname="Hersteller">
                <Field fieldname="hersteller" tagname="Name"/>
```

[3]Die Version des Programms, die ich für dieses Papier verwende, wurde von Florian Schulze, Torben Schindler, Jens Tiefenbach und Frank Hebler entwickelt. Andere Gruppen von Studenten haben ähnliche Programme im Rahmen des Praktikums erstellt.

```
            <Group>
                <Field fieldname="name" tagname="Medikament"/>
            </Group>
        </Group>
    </Group>
  </GroupSchema>
</xsInfo>
```

Diese Datei legt fest, dass ein XML-Dokument `<Apotheke>` aus den Informationen der Datenbank erzeugt werden soll, das die `<Medikament>`e geordnet nach `<Hersteller>`n enthält.

Und in der Tat, das Ergebnis nach dem Aufruf
`java XS -f xa.xs -o xs-out.xml` ist:[4]

```
<Apotheke>
 <Hersteller>
  <Name>Byk Gulden</Name>
  <Medikament>Pantozol</Medikament>
 </Hersteller>
 <Hersteller>
  <Name>Engelhard</Name>
  <Medikament>Glycilax</Medikament>
 </Hersteller>
 <Hersteller>
  <Name>Hoechst</Name>
  <Medikament>Novalgin</Medikament>
 </Hersteller>
 <Hersteller>
  <Name>Mack</Name>
  <Medikament>Rhinopront</Medikament>
  <Medikament>Codipront</Medikament>
 </Hersteller>
</Apotheke>
```

3.2 XML-Bäume und Datenbank-Tabellen

Eines muss ich natürlich zugeben! Das Beispiel des vorherigen Abschnitts ist ein bisschen geschummelt: Betrachtet man die DTD der kleinen Apotheke genau, so sieht man, dass

- `<zugang>` ein Attribut des Elements `<medikament>` ist. Diese Zuordnung geht in der Datenbanktabelle verloren. Das Feld `zugang` unterscheidet sich in der Datenbank nicht vom Feld `name`, obwohl es sich in der XML-Datei beim Ersten um ein Attribut, beim Zweiten um ein Element handelt.

- `<wirkstoff>` ist in der DTD optional. Dem mag in der Datenbank entsprechen, dass für dieses Feld `NULL`-Werte erlaubt sind. Unsere

[4] Mit dem hier verwendete Programm XS hat man die Möglichkeit, auch komplexere XML-Dokumente aus einer flachen Ergebnistabelle einer SQL-Anweisung zu erzeugen. Es wäre interessant, diesen Ansatz weiterzuverfolgen — vgl. auch [3] und den folgenden Abschnitt.

XSL-Transformation berücksichtigt jedoch nicht, wenn kein Element namens `<wirkstoff>` in der XML-Datei gefunden wird.

- Gravierender: `<anwendung>` darf in der XML-Datei mehrfach vorkommen. Diese „Liste“ von Elementen ist jedoch in der Datenbank nicht berücksichtigt — und ließe sich auch gar nicht berücksichtigen. Wir müssten schon `anwendung` in eine eigene Tabelle mit einer `n-1`-Beziehung zur Tabelle `medikament` auslagern.

Diese Anmerkungen sollen zeigen, dass die Struktur von XML-Dokumenten und relationalen Datenmodellen keineswegs identisch ist. Wenn relationale Datenbanken für die Speicherung von XML-Dokumenten verwendet werden, muss man sich also Gedanken über die Zuordnung der Dokumentstruktur zur Datenbankstruktur machen.

Man unterscheidet zwei Ansätze für diese Zuordnung:

- **Zuordnung durch Vorlagen.** Dies ist die Technik, die oben am Beispiel des Programms `XS` demonstriert wurde: Eine Vorlage legt fest, wie die Ergebnismenge einer SQL-Abfrage als XML-Dokument dargestellt wird. Wir werden diesen Ansatz auch beim Werkzeug XSQL von Oracle finden. Solche Zuordnungen können sehr flexibel sein. Sie werden insbesondere bei der Transformation von Datenbank-Inhalten zu XML-Dokumenten verwendet.

- **Zuordnung durch Datenmodelle.** Bei diesem Ansatz definiert man ein Datenmodell für die XML-Dokumente, das sie auf eine relationale Datenbank abbildet. Man kann sich dabei verschiedene Techniken denken, deren Extreme sind:

  1. Ein XML-Dokument entspricht einem Feld einer Tabelle etwa vom Typ `CLOB`, d.h. die Datenbank hat keinerlei strukturelle Information, für sie ist das gesamte XML-Dokument ein Atom.
  2. Ein Element entspricht einem Feld einer Tabelle, dazu speichert man die Position im Baum. Die Datenbank hat wieder keinerlei strukturelle Information, weil sich der Baum des XML-Dokuments allein durch die gespeicherten Positionsangaben ergibt, nicht aus der Struktur der Datenbank.

Da ist es schon klüger, die Zuordnung aus der Abbildung von Objekten auf Relationen herzuleiten (siehe [3]). Man verwendet dazu Techniken des *object-relational mapping*. Man kann diese Methode auch damit kombinieren, Teile von XML-Dokumenten in Orginalform zu speichern, also mitsamt den Tags.

Wir beenden dieses Thema hier abrupt — leider.

3.3 Ein Beispiel: Was bedeutet das *i* in Oracle 8*i*?

Alle Techniken, die wir an dem einfachen Beispiel der kleinen Apotheke demonstriert haben, finden wir in Oracle auch vor. Die Abbildung 15 aus [6] gibt einen Überblick der Werkzeuge:

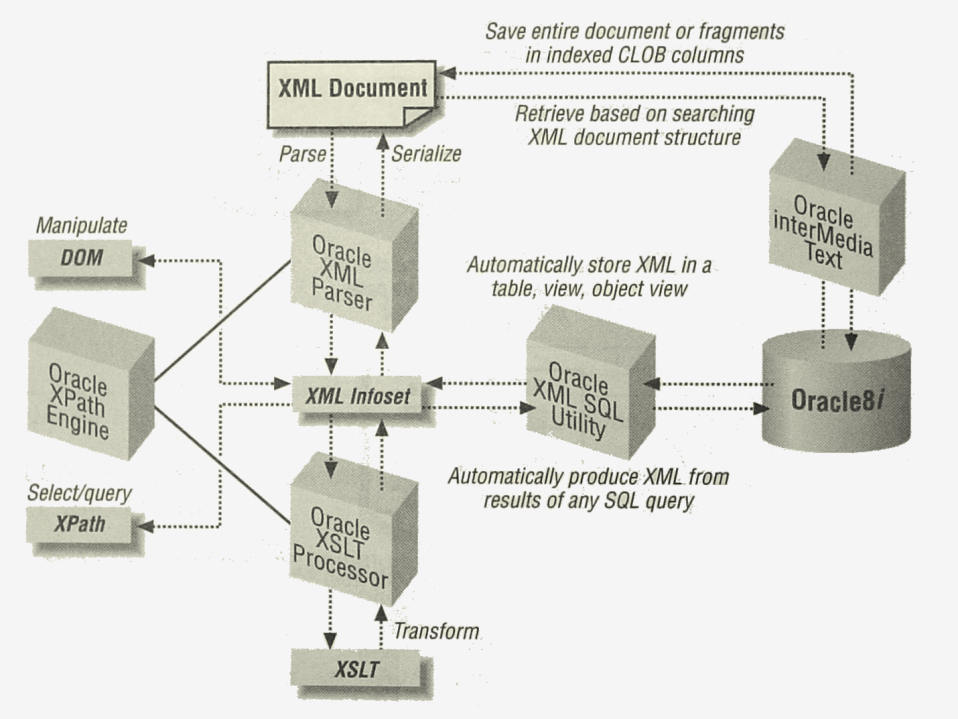

Abbildung 15: Überblick der XML-Technik in Oracle, aus [6] S.22

Oracle XML Parser Den Oracle XML Parser gibt es in Java, PL/SQL und C/C++ auf allen gängigen Betriebssystem. Man kann den Parser als eigenständiges Programm einsetzen, oder auch innerhalb der Datenbank verwenden. Dies geht mit der Java- und der PL/SQL-Version ab Oracle 8*i*.

Die Tatsache, dass der Parser innerhalb der Datenbank verwendet werden kann, ist *die* Erweiterung der Datenbank durch XML-Mechanismen: XML erscheint wie ein Teil der Funktionalität der Datenbank. (Nebenbei bemerkt: Wie Java auch!)

Oracle XSLT Prozessor Auch von diesem Werkzeug gibt es die Versionen in Java und PL/SQL, die auch innerhalb der Datenbank laufen und eine C/C++-Version.

Oracle XPath Engine XPath ist die Spezifikation für das Navigieren und

Adressieren im Baum eines XML-Dokuments, die XPath Engine wird vom Parser und vom XSLT Prozessor verwendet.

Oracle XML SQL Utility bietet eine Reihe von Diensten, um XML-Dokumente in die Datenbank zu speichern und um Ergebnisse von SQL-Abfragen als XML-Dokumente darzustellen.

Oracle interMedia verwaltet komplette Dokumente oder Fragmente von Dokumenten, die indiziert in der Datenbank als `CLOB`s gespeichert werden. interMedia kann in solchen Dokumenten suchen und berücksichtigt dabei die XML-Struktur.

Auf Basis dieser Werkzeuge ist **Oracle XSQL** ein System, das innerhalb von XSQL Pages Ergebnisse von SQL-Abfragen einbezieht. Dadurch entsteht eine Maschine für Web-Publikationen, wie sie in Abbildung 16 dargestellt ist. Oracle XSQL funktioniert außerhalb der Datenbank mit jeder Servlet Maschine, wie etwa Apache JServ oder Tomcat, aber auch innerhalb der Datenbank ab Oracle 8*i* Release 3. XSQL ist Teil des Oracle Internet Application Servers.

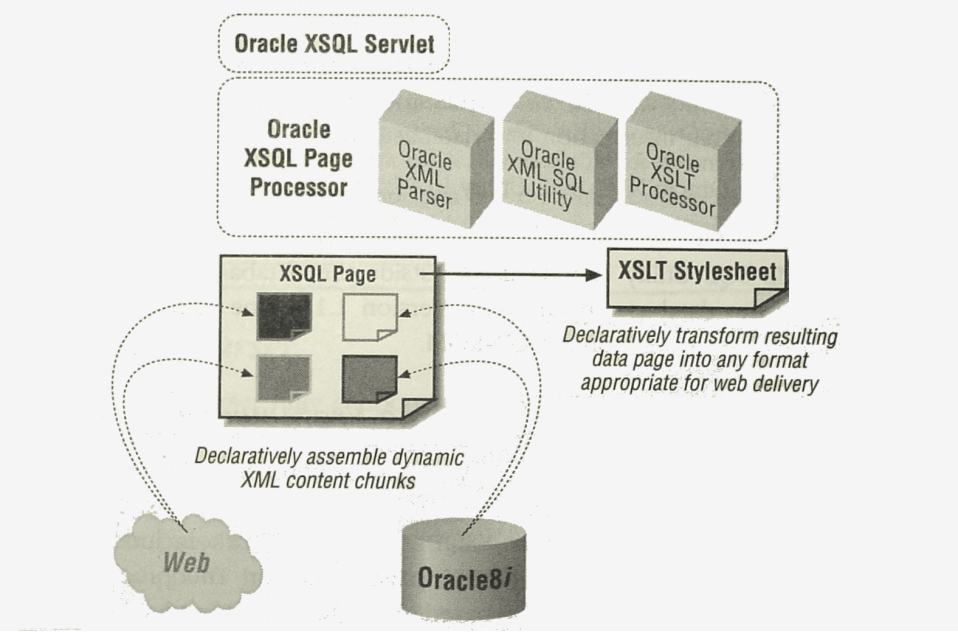

Abbildung 16: Publizieren mit XSQL Pages, aus [6] S.26

Mit diesem kurzen Blick auf die Werkzeuge von Oracle (mehr in [6]) verabschiedet sich die kleine XML-Apotheke[5], — freilich nicht ohne zu

[5] Apropos: Sie verdankt ihren Namen der Wahl des Beispiels, das mir naheliegend erschien, weil das Papier während der Vorbereitung eines Vortrags bei Byk Gulden, einem pharmazeutischen Unternehmen, entstanden ist.

erwähnen, dass natürlich andere Datenbank-Hersteller ähnliche, aber auch ganz andere Konzepte verfolgen.

Die Dinge sind in diesem Bereich noch ziemlich im Fluss!

Quellen

[1] **The Apache XML Projekt:**
http://xml.apache.org.

[2] **Frank Boumphrey, Olivia Direnzo, Jon Duckett, Joe Graf, Dave Hollander, Paul Houle, Trevor Jenkins, Peter Jones, Adrian Kinsley-Hughes, Kathy Kinsley-Hughes, Craig McQueen and Stephen Mohr:** *XML Applications.* Wrox, 1998.

[3] **Ronald Bourrett:** *XML and Databases.*
http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[4] **Michael Kay:** *XSLT — Programmer's Reference.* Wrox, 2000.

[5] **Eric A. Meyer:** *Cascading Style Sheets — The Definitive Guide.* O'Reilly, 2000.

[6] **Steve Muench:** *Building Oracle XML Applications.* O'Reilly, 2000.

[7] **Organization for the Advancement of Structured Information Standards (OASIS):**
http://www.oasis-open.org.

[8] **Sebastian Rahtz:** *PassiveTeX.*
http://users.ox.ac.uk/~rahtz/passivetex/

[9] **Gunther Rothfuss, Christian Ried (Hrsg.):** *Content Management mit XML.* Springer, 2001

[10] **W3C: Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler (Eds.):** *Extensible Markup Language(XML) 1.0 (Second Edition).* W3C Recommendation 6 October 2000
http://www.w3.org/TR/2000/REC-xml-20001006.

[11] **W3C: Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Paul Grosso, Eduardo Gutenberg, Alex Milowski, Scott Parnell, Jeremy Richman, Steve Zilles:** *Extensible Stylesheet Language (XSL) Version 1.0.* W3C Candidate Recommendation 21 November 2000
http://www.w3.org/TR/2000/CR-xsl-20001121.

[12] **W3C: James Clark (Ed.):** *XSL Transformations (XSLT) Version 1.0.* W3C Recommendation 16 November 1999
http://www.w3.org/TR/1999/REC-xslt-19991116.

[13] **Norman Walsh and Leonard Muellner:** *DocBook — The Definitive Guide.* O'Reilly, 1999.