Proseminar „Rund um XML“

Thema:

SAX & DOM

von

Alexander Kleinen
alexkleinen@uni-koblenz.de

Christian Bruckhoff
brchrist@uni-koblenz.de

Inhaltsverzeichnis

1. Einleitung

DOM und SAX sind beides APIs (**A**pplication **P**rogramming **I**nterface) zur Auswertung und Interpretation von XML- Daten.
Zwischen ihnen gibt es einige Unterschiede, welche einen unterschiedlichen Nutzen nach sich ziehen.

2. SAX

2.1 Einführung

2.1.1 Begriff

SAX steht für "**S**imple **A**PI for **X**ML". API heißt wiederum "Application Programming Interface" und repräsentiert eine Sammlung von Schnittstellen und Klassen, wie sie zumeist bei objektbasierten Programmiersprachen Verwendung finden. Also frei übersetzt heißt SAX „Einfache Anwendungsprogrammier-schnittstelle für XML".

2.1.2 Funktion von SAX

SAX stellt eine Schnittstelle für viele verschiede XML Parser (z.B. Xerces (Java), MSXML (COM), Ælfred (Java)) dar. Diese Schnittstelle ist aber auch für andere Parser für Programmiersprachen wie C, C++, Perl und Delphi verfügbar.
Da SAX ursprünglich für Java konzeptiert wurde und heute auch auf dieser Ebene weiterentwickelt wird, ist auch viel von dem Code, das zur Bearbeitung von XML geschrieben wurde, in Java geschrieben. Der beste Grund warum XML meist mit Java bearbeitet wird, ist dass beide Sprachen auf das Internet zugeschnitten sind.
SAX ist eine standardisierte Möglichkeit (z.B. neben DOM), wie eine XML Datei durch einen Parser bearbeitet wird.

2.1.3 Parser

Definition Parser: „Ein Parser nennt man ein Programm, das während der syntaktischen Analyse ein Quellprogramm als Eingabe bekommt. Dies erhält jener aus der lexikalischen Analyse vom sog. Scanner und in Form einer Folge von Token. Er hat die Aufgabe, daraus einen Ableitungsbaum zu erstellen und hinsichtlich syntaktischer Korrektheit zu überprüfen.“ (Java und XML für Dummies) Parser sind wichtiges Hilfsmittel, um einer Anwendung XML- Daten verständlich zu machen. Parser lesen die XML- Dokumente, erkennen die Struktur und geben gefundene Informationen an die Anwendung weiter. Zusätzlich kann meist ein Parser erkennen, ob das XML-Dokument seine Gültigkeit besitzt.

2.1.4 Überblick–Parser

Das Document Object Model (DOM) und die Simple API for XML (SAX) sind zwei Parser für XML.
Es wird grundsätzlich in zwei Typen von Parser APIs unterschieden:

- Baumbasierte APIs (z.B. DOM)
- Ereignisgesteuerte APIs (SAX)

Je nach Anwendung und Vorhaben der XML- Dokumente muss man entscheiden, welchen Parser man benutzt. Doch zu Funktion von SAX und seinen Vor- und Nachteilen später mehr.

2.2 SAX-Spezifikationen

Der SAX 1.0 Parser für Java enthält 11 Kern- Klassen und Schnittstellen mit 3 optionalen Hilfsklassen. Aber meist werden nur 3 Schnittstellen vom Parser verwendet.
Die Klassen und Schnittstellen von SAX lassen sich in 5 Gruppen unterteilen:

2.2.1 interfaces implemented by the parser

Ein SAX- konformer XML- Parser braucht in der Regel nur zwei oder drei simple Schnittstellen. D.h. es ist möglich alle Schnittstellen in nur einer Klasse zu realisieren. Gebräuchlich kann man diese Klasse auch als „SAXDriver“ nennen.
Die wichtigsten Schnittstellen im Überblick:

- Parser
  Dies sind die Hauptschnittstellen von einem SAX-Parser. Sie startet ein XML-Parsing und ist auch für die Fehlerbehandlung von Bedeutung

- AttributeList
  Diese Schnittstelle erlaubt dem Benutzer aus einer Vielzahl von Attributen auszuwählen, die der Parser z.B. in der „SAXDriver“ Klasse implementieren kann.

- Locator
  Diese einfache Schnittstelle erlaubt es dem Benutzer die aktuelle Stelle im XML-Dokument zu finden.

2.2.2 interfaces implemented by the application

Eine SAX-Applikation kann mithilfe folgender Schnittstellen realisiert werden.

- DocumentHandler
  Die Schnittstelle enthält Methoden die eine Verarbeitung eines XML-Dokuments erst möglich machen. Z.B. Start-Tags können ausfindig gemacht und weiterverarbeitet werden. Eine weitere Erklärung findet sich in der Skizze (Abb. 1) sowie im Beispiel („MyThreeSins“).

- ErrorHandler
  Wenn eine Anwendung eine Fehlerbehandlung braucht, wird die Implementierung dieser Schnittstelle benötigt.

- DTDHandler
  Falls die Document Type Definition bearbeitet werden sollte, muss man diese Schnittstelle implementieren um Informationen über die NOTATION und ENTITY Deklarationen zu erhalten.

2.2.3 standard SAX classes

SAX besteht zum größten Teil aus Schnittstellen, anstatt aus Klassen. Dennoch beziehen sich die Schnittstellen auf zwei Standard Exception Klassen.
Die folgenden Klassen sind nützlich für den Parser sowie der Anwendung selbst.

- InputSource
  Diese Klasse zeigt nützliche Information über das Dokument, wie z.B. Verfasser, byte stream oder den charachter stream.

- SAXException
  Diese Klasse repräsentiert alle generellen SAX-Exceptions.

- SAXParseException
  Wenn ein XML Dokument syntaktische Fehler enthält, werden sie durch die Fehlerbehandlung aufgefangen.

- HandlerBase
  Eine Standart-Sammlung für den DocumentHandler, ErrorHandler, DTDHandler sowie den EntityResolver.

2.2.4 optional Java- specific classes in the Org.xml.helpers package

Diese Klassen sind nicht Teil von SAX, sondern nur ein optionaler Zusatz. Diese „Hilfs“-Klassen sind deswegen auch in unterschiedlicher Sprache erhältlich.

2.2.5 Java demonstration classes in the nul package

Einfache Demos zur Demonstration von SAX.

2.3 XML Dokumente bearbeiten

2.3.1 Die Simple API zur XML-Bearbeitung verwenden

Der Ansatz von SAX ist grundsätzlich verschieden von DOM. Hier wird kein hierarchisches Baummodell aufgebaut, sondern eine Sequenz von Ereignissen, wie z.B. dem Anfang oder dem Ende eines Elements, erzeugt. Der Vorteil hiervon gegenüber DOM ist, dass man kein vorgegebenes Objektmodell verwenden muss, sondern sich sein eigenes modellieren kann. SAX erwartet vom implementierenden Parser nicht viel (deswegen auch Simple API for XML), außer dass er das Dokument scannen und Ereignisse erzeugen soll. Wie die Ereignisse interpretiert werden, ist dem Anwendungsentwickler überlassen.
SAX ist daher sehr effizient und macht es möglich, eigene Objektmodelle zu erschaffen.

2.3.2 Anwendungsgebiete

Dadurch dass DOM das Dokument komplett im Speicher ablegt und erst dann Operationen auf selbigen möglich sind, ist SAX die bessere Alternative sobald man größere Dokumente auslesen muss.
Beispiel:
Man möchte in einem Dokument mit allen Büchern der Staatsbibliothek ein bestimmtes Buch suchen. Wenn man diese Suche nicht an einem Großrechner durchführen will, so kann man diese Problemstellung leichter mit SAX lösen. Man registriert beim Parser einen ContentHandler, in dem man nun einfach den Namen eines Buches mit dem des gesuchten vergleichen und dieses dann in

einem Objekt gekapselt der Applikation zukommen lassen kann. Was einem Entwickler natürlich klar sein muss, ist, dass er mit SAX keine Dokumente oder Datensätze verändern oder neu erstellen kann. Es ist ja auch eine API zum parsen, nicht zum modifizieren/ erstellen von XML. Dafür müsste man sich wiederum eine Baumstruktur aufbauen und dafür ist das Document Object Model prädestiniert.

2.3.3 Eigenschaften von SAX-Parsern

Ein SAX Parser ist ein sehr primitiver Parser und übernimmt hauptsächlich die Funktion eines Scanners. Da bei XML die Klammerung schon von Haus aus vorhanden ist, ist die normalerweise schwierigste Funktion eines Parsers trivial. Bei einem gültigen XML Dokument besteht immer eine Art Baumstruktur, die der Parser nur noch durch einen Scanner erfassen und auswerten muss. Bei Xerces, dem SAX Parser des Apache Projects, ist hauptsächlich die Klasse XMLDocumentScanner für diese Aufgabe vorgesehen. Diese ist für die Erkennung der XML Grammatik vorgesehen und wird von den Klassen XMLEntityHandler und XMLDTDScanner unterstützt. Das eigentliche Parsen übernehmen bei Xerces die Klassen DOMParser und SAXParser, welche die jeweiligen Interfaces implementieren.
Auf den SAX- Parser wird im Beispiel eingegangen.

SAX ist so einfach, dass die Essenz davon in nur einem Satz zusammengefasst werden kann:
„Beginne am Anfang und arbeite dich durch, bis du zum Ende gelangst“ (Zitat: JAVA und XML für Dummies)

2.4 Struktur von SAX

SAX besteht hauptsächlich aus 2 Komponenten: dem Parser und dem Interface. Der Parser ("XMLReader") versorgt dem Interface ("ContentHandler") mit den Informationen aus dem XML-Dokument, indem dessen Methoden aufgerufen werden.

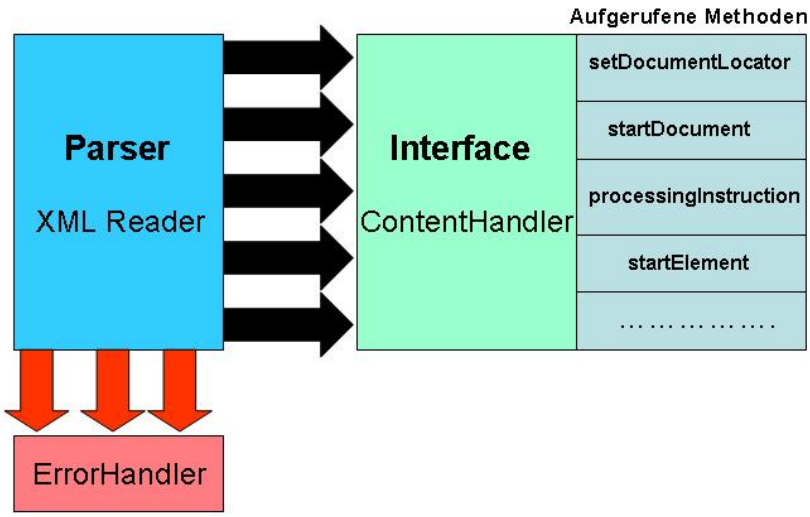


Abb. 1 SAX- Struktur

2.4.1 Beschreibung: Parser

Der Parser wird in SAX als "XMLReader" bezeichnet. Er stellt neben den Methoden zum Parsen des Dokumentes auch die Möglichkeit bereit, den Parser zu konfigurieren. So kann man zum Beispiel die Gültigkeitüberprüfung aktivieren und deaktivieren.
Erstellt wird eine Instanz des XMLReader mithilfe der Klasse XMLReaderFactory aus dem Paket "org.xml.sax.helpers". Dazu später mehr.

2.4.2 Beschreibung: Interface

Bei dem Interface handelt es sich um den "ContentHandler" und wird in der Regel vom Anwender selbst implementiert. Die implementierten Methoden werden während des Parsens vom XMLReader aufgerufen. Davor muss der ContentHandler dem XMLReader mitgeteilt werden.
Bsp.:

```
parser.setContentHandler(new MyContentHandler());
```

2.5 Ereignis und Rückruf

2.5.1 Ereignisse behandeln

→ SAX- Programme sind ereignisgesteuert.
Ereignisgesteuerte Programmierung besteht aus drei Teilen:

1. **Registrierung**
   Sie registrieren Ihren Wunsch benachrichtigt zu werden, wann immer ein Ereignis eintritt. Sie registrieren diesen Wunsch bei einem anderen Stück Programmcode- einem weiteren Objekt, normalerweise etwas, das sie importiert haben (wie z.B. einem Stück Code, das Bestandteil der API von jemand anderem ist). Dieses Objekt hält dann hinter den Kulissen Ausschau, ob das von Ihnen spezifizierte Ereignis eintritt.

2. **Ereigniseintritt**
   Ein bestimmtes Ereignis geschieht. (Der Parser entdeckt etwas Interessantes.)

3. **Rückruf**
   Das andere Stück Programmcode nimmt einen Rückruf vor. Eine Ihrer Methoden wird aufgerufen.

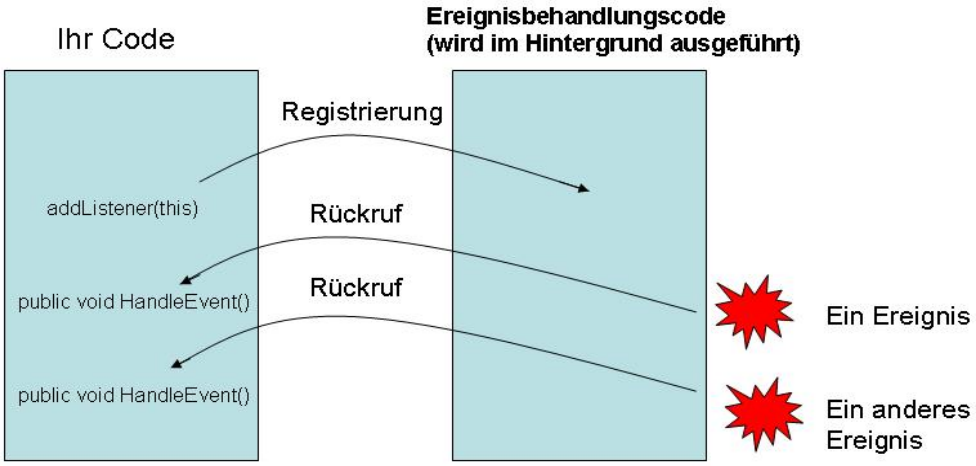


Abb.2 Ereignis und Rückruf

2.5.2 SAX-Ereignisse

Jedes SAX- Programm besitzt zwei unerlässliche Codeteile:

1. Ein Stück Code, den man schreibt- die Behandlungsroutine. (Die Behandlungsroutine kann eine vorgefertigte DEFAULTHANDLER- Klasse erweitern.)
2. Ein Stück Code, den man normalerweise nicht schreibt- den Parser.
   Der Parser spielt eine wichtige Rolle. Die Java 1.4 API besitzt einen integrierten Parser. Man erzeugt eine Instanz dieses Parsers und registriert dann eine Behandlungsroutine bei dieser Parserinstanz. Man weist der Instanz an, Ihre Behandlungsroutine zurückrufen, wenn ein Ereignis eintritt.

Das Registrieren und Rückruf Szenario macht SAX ereignisgesteuert. Das triviale ist nun, dass ein SAX- Ereignis nicht greifbar ist. Ein SAX- Ereignis erinnert Sie nicht an einen Tastendruck oder einen Mausklick. In SAX durchsucht der Parser ein XML- Dokument von oben bis unten. Wann immer der Parser auf etwas Interessantes stößt, löst er ein Ereignis aus und ruft die Behandlungsroutine auf. Danach liegt es bei der Behandlungsroutine, etwas mit dieser interessanten Begegnung zu tun.

2.6 Beispiel: „MyThreeSins“

2.6.1 Hinzufügen der Xerces Klassen mit Eclipse

Um ein XML Dokument in JAVA zu bearbeiten benötigen wir neben den Standard Tools für JAVA noch zusätzlich einen XML Parser. Für dieses Beispiel wurde Xerces von Apache verwendet. Diesen kann man unter http://xml.apache.org herunterladen.
Nachdem die herunter geladene Datei in ein Verzeichnis entpackt wurde, müssen wir dem JAVA Compiler noch sagen, wo er die neuen Klassen von Xerces findet, die wir benutzen wollen. Diese befinden sich im Hauptverzeichnis von Xerces.
Diese kann man unter z.B. Eclipse unter *Window → Properties → Java → Build Path → User Libraries* hinzufügen. Nun kann man dort mit *New…* ein neues User Library erstellen. Danach wird das Verzeichnis markiert und man kann nun mit *Add JARs…* alle Bibliotheken von Xerces hinzufügen.
Nachdem wir dies geschafft haben erstellen wir uns ein neues Java Projekt. Nach der Eingabe des Namens findet man nach dem Klick auf *Next* > unter der Registerkarte *Libraries* den Button *Add Library...*. Dort können wir nun unser User Library von Xerces hinzufügen.
Nun können wir anfangen zu programmieren…

2.6.2 Code

XML-Dokument „MyThreeSins.xml“

```
<? xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Sin SYSTEM "MyThreeSins.dtd">

<MyThreeSins>
   <Sin rank="favorite">Faulheit</Sin>
   <Sin>Gefräßigkeit</Sin>
   <Sin>Neid</Sin>
</MyThreeSins>
```

CallSAX.java

```
/**
 * @author Alexander Kleinen
 */
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import java.io.File;
import java.io.IOException;

public class CallSAX {

 public static void main(String[] args)
       throws SAXException,
                   ParserConfigurationException,
                   IOException

 {
       SAXParserFactory factory =
                   SAXParserFactory.newInstance();
             SAXParser saxParser =factory.newSAXParser();
             XMLReader xmlReader = saxParser.getXMLReader();
             xmlReader.setContentHandler(new MyContentHandler());
             xmlReader.parse
                   (new File("MyThreeSins.xml").toURL().toString());

 }
```

MyContentHandler.java

```
/**
 * @author Alexander Kleinen
 */

import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;

public class MyContentHandler extends DefaultHandler{

  public void startDocument()
  {
        System.out.println("Dokumentstart");
  }
  public void startElement(String uri,
                           String localName,
                           String qualName,
                           Attributes attribs)
  {
        System.out.print("Start- Tag: ");
        System.out.println(qualName);
        for (int i=0; i<attribs.getLength();i++)
        {
        System.out.print("Attribut: ");
        System.out.print(attribs.getQName(i));
        System.out.print(" = ");
        System.out.println(attribs.getValue(i));
        }
  }
  public void characters
              (char[] charArray, int start, int length)
  {
        String charString =
              new String(charArray, start, length);
        charString = charString.replaceAll("\n", "[cr]");
        charString = charString.replaceAll(" ", "[blank]");
        System.out.print(length+ " Zeichen: ");
        System.out.println(charString);

  }
  public void endElement(String uri,
                         String localName,
                         String qualName)
        {
        System.out.print("End- Tag: ");
        System.out.println(qualName);
        }
  public void endDocument()
  {
        System.out.println("Dokumentende.");
  }
}
```

```
Dokumentstart
Start- Tag: MyThreeSins
0 Zeichen:
1 Zeichen: [cr]
3 Zeichen: [blank][blank][blank]
Start- Tag: Sin
Attribut: rank = favorite
8 Zeichen: Faulheit
End- Tag: Sin
0 Zeichen:
1 Zeichen: [cr]
3 Zeichen: [blank][blank][blank]
Start- Tag: Sin
12 Zeichen: Gefräßigkeit
End- Tag: Sin
0 Zeichen:
1 Zeichen: [cr]
3 Zeichen: [blank][blank][blank]
Start- Tag: Sin
4 Zeichen: Neid
End- Tag: Sin
0 Zeichen:
1 Zeichen: [cr]
End- Tag: MyThreeSins
Dokumentende.
```

Entsprechende Ausgabe bei Aufruf

2.6.3 Erklärung zum Beispiel „MyThreeSins“:

a) Die CallSAX- Klasse:
In dem Beispiel befindet sich das Wesentliche innerhalb des Body der Hauptmethode. Es gibt fünf Anweisungen innerhalb des Body, und man kann sagen, dass diese Anweisungen verwendet, wieder verwendet und wieder verwendet werden können. D.h. man diese SAX- Anweisungen universell weiterbenutzen.

Drei Anweisungen erzeugen einen XML- Parser:

Man definiert im Body der Hauptmethode drei neue Objekte:
1. `factory`
2. `saxParser`
3. `XML-Reader`

Erklärungen:
1. Die erste Anweisung erzeugt ein Factory-Objekt.
   Ein Factory ist ein Objekt, das zur Erzeugung andere Objekte verwendet wird. Und wozu benötigen Sie derartige Factorys? Man ruft keinen Konstruktor auf, da man sich nicht sicher sein kann, welches Objekt welcher Klasse konstruiert wird.

Die 2. und 3 .Anweisungen konstruieren einen richtigen Parser:

`XMLReader` ist eine Schnittstelle, und `saxParser` ist eine abstrakte Klasse.
`XMLReader` listet Methoden auf, die implementiert werden müssen,
wogegen `saxParser` die Basisklassse bildet, die der Parser erweitern soll.

b) Das XML- Dokument abrufen:
`new File(„MyThreeSins.xml“).toURL.toString();`
Dies besteht aus Aufrufen des File- Konstruktor, die `toURL`- Methode und die `toString`- Methode
1. Der FileKonstruktor findet die Datei auf der Festplatte.
2. Die Methode `toURL` stellt Ihre Datei als webfähige Ressource dar.
3. Die Methode `toString` verwandelt diese abstrakte URL- Angelegenheit in eine echte Zeichenfolge. Diese wird dann in dem Parser eingelesen.

c) Das Dokument bearbeiten:
- **Die Handler- Klasse:**
  Die Handler-Klasse enthält Methoden mit Namen wie `startDocument`, `startElement` und `endElement`. Dieses Handler- Objekt wird beim Parser registriert. Wenn ein Ereignis stattfindet, erhält das Handler-Element einen Rückruf. Dieser Verlauf wird in Abb.2 (Ereignis und Rückruf) dargestellt.

- **Die Registrierung:**
  In den Worten der SAX- Handler- Objekte passiert nichts, falls der Handler sich nicht beim Parser registriert. Codezeile dazu:
  `Xml.Reader.setContentHandler(new MyContentHandler());`
  Es wird eine neue Instanz von `MyContentHandler` erzeugt. Man kann ebenfalls sagen: "Wann immer etwas interessantes geparst wird, ruf eine der Methoden des MyContentHandler- Objekts auf".

- **MyContentHandler:**
  Das Programm stellt die Namen von Elementen, die Werte von Attributen und die Zeichen innerhalb von Elementen dar. Das Programm macht dies mit Methoden wie `startElement, characters` und `endElement`. Wenn das XML-Dokument geparst wird, durchsucht der Parser das Dokument von oben bis unten. Wenn der Parser ein interessantes Objekt bemerkt(der Start eines Elements, Zeichen in einem Element usw.), ruft er die passende Methode auf. Um genau zu sein, nimmt der Parser einen Rückruf an die passende Methode vor.

- **Rückruf- Methoden:**
  Der neue `MyContentHandler` erweitert Javas vorgefertigte `DefaultHandler`- Klasse. `MyContenHandler` besitzt außerdem fünf Rückruf-Methoden. Jede Methode sagt dem Computer, was zu tun ist, wenn eine bestimme Art von Ereignis stattfindet. Wenn der Parser beispielsweise dem schließenden tag </sin> begegnet, ruft der Computer die Methode endElement zurück.

→Die Methode `startDocument`
Wenn der Parser mit dem Parsen beginnt, nimmt er einen Rückruf an die Methode startDocument vor. Im Grunde bedeutet ein Aufruf der Methode: "Hallo, Handler! Ich beginne, das Dokument MyThreeSins.xml zu parsen." Reaktion ist: Er stellt das Wort „DokumentStart" auf dem Bildschirm dar.

→Attribute und die Methode `startElement`
Die Methode `startElement` wird immer dann aufgerufen, wenn der Parser auf ein Start- Tag stößt. Wenn der Parser z.B. `<Sin rank="favorite">`
sieht ruft er `startElement` auf.

Parameter `qualName`: Name des Elements.
Parameter `attributes`: Ein Attribute Objekt besitzt sehr viele nützliche Methoden- z.B. `getQName` und `getValue`.
Wenn man nichts mit den XML- Namensbereichen zu tun hat, gibt `getQName` ganz schlicht und einfach den Attribut- Namen zurück.
Die Methode `getValue` gibt den Attributwert zurück.

→ Die Methode `characters`
Diese Methode verarbeitet das XML- Dokument als `CharArray`. Sie stellt z.B. Zeichenlänge von Tags, sowie Attribute dar. Außerdem werden Leerzeichen gezählt und als [blank] dargestellt.

→ Methode `endElement`
Die Methode `endElement` wird immer dann aufgerufen, wenn der Parser auf ein End- Tag stößt. Diese Methode besitzt wie auch `startElement` den Parameter `qualName`. Der Paramter `attributes` ist nicht enthalten, da End- Tags niemals Attribute haben können.

→ `endDocument`
Ist der Parser mit dem Durchsuchen des Dokuments fertig, ruft er die Methode `endDocument` auf.

2.7 Wann wird die SAX-API verwendet.
Was also ist gut an SAX? Wann sollten man die SAX- API verwenden?

1. Man verwendet SAX, wenn das XML- Dokument lineare Daten repräsentiert.
Ein SAX- Parser beginnt am Anfang und arbeitet sich Schritt für Schritt bis zum Ende durch. Im Gegensatz zu anderen APIs betrachtet der SAX- Parser das Dokument nicht als eine Sammlung von verschachtelten Elementen. Für Elemente innerhalb von Elementen ist SAX nicht die richtige API.
Wenn anderseits die Verschachtelungen sehr gering sind und es Sinn macht das Dokument linear abzuarbeiten, ist SAX eine gute Wahl.

2. SAX bei sehr großen Dokumenten.
Die SAX-API hat im Grunde einen Tunnelblick. Wenn SAX mit einem bestimmten Start Tag beschäftigt ist, sind alle umgebenen Tag (einschließlich des End-Tags, der zum Start-Tag gehört) unsichtbar. Die Kurzsichtigkeit kann hinderlich, aber auch von Vorteil sein. Egal wie groß das Dokument ist, der SAX- Parser sieht nur einen kleinen Bruchteil des Dokuments auf einmal und ist deshalb Arbeitsspeichersparend.

3. DOM

3.1 Definition

DOM heißt **D**ocument **O**bject **M**odel, und ist eine vom W3C (**W**orld **W**ide **W**eb **C**onsortium) definierte programmiersprachenunabhängige Programmschnittstelle (API).
Es handelt sich hierbei um eine abstrakte Schnittstellenbeschreibung in OMG IDL (**O**bject **M**anagement **G**roup **I**nterface **D**efinition **L**anguage), die erst in eine konkrete Sprache implementiert werden muss, bevor sie angewendet werden kann.

Das DOM definiert Schnittstellen und Methoden zum erzeugen, durchsuchen, zugreifen, ändern und löschen von Dokumentinhalten.
Dokumente werden unabhängig von der internen Datenstruktur der jeweiligen Implementation nach außen objektorientiert repräsentiert.
Jedes XML-Dokument hat eine eindeutige Struktur, welche in jeder DOM Implementation genau gleich aussieht.

3.2 Spezifikationen

Die DOM Spezifikation wird vom W3C kontrolliert. Da die Spezifikation eines solchen Standards eine Menge Zeit in Anspruch nimmt, beschloss das W3C diese Stufe für Stufe aufzubauen. Jede dieser Stufen enthält nur so viele Spezifikationen wie nötig sind und lässt alles weg, was weiterer Informationen bedarf. Diese werden dann in späteren Stufen eingefügt.

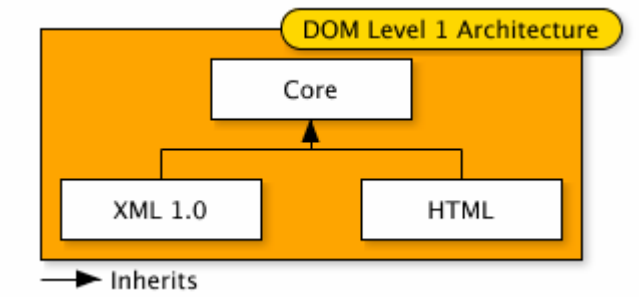
Derzeit gibt es folgende Stufen der DOM-Spezifikation:

- DOM Level 0
- DOM Level 1
- DOM Level 2
- DOM Level 3

3.2.1 DOM Level 0

DOM Level 0 ist kein Standard des W3C. Es ist lediglich eine Definition der Funktionalität, wie sie z.B. im Internet Explorer 3.0 oder Netscape Navigator 3.0 zu finden ist
Die W3C Spezifikation von DOM Level 1 baut auf diese auf.

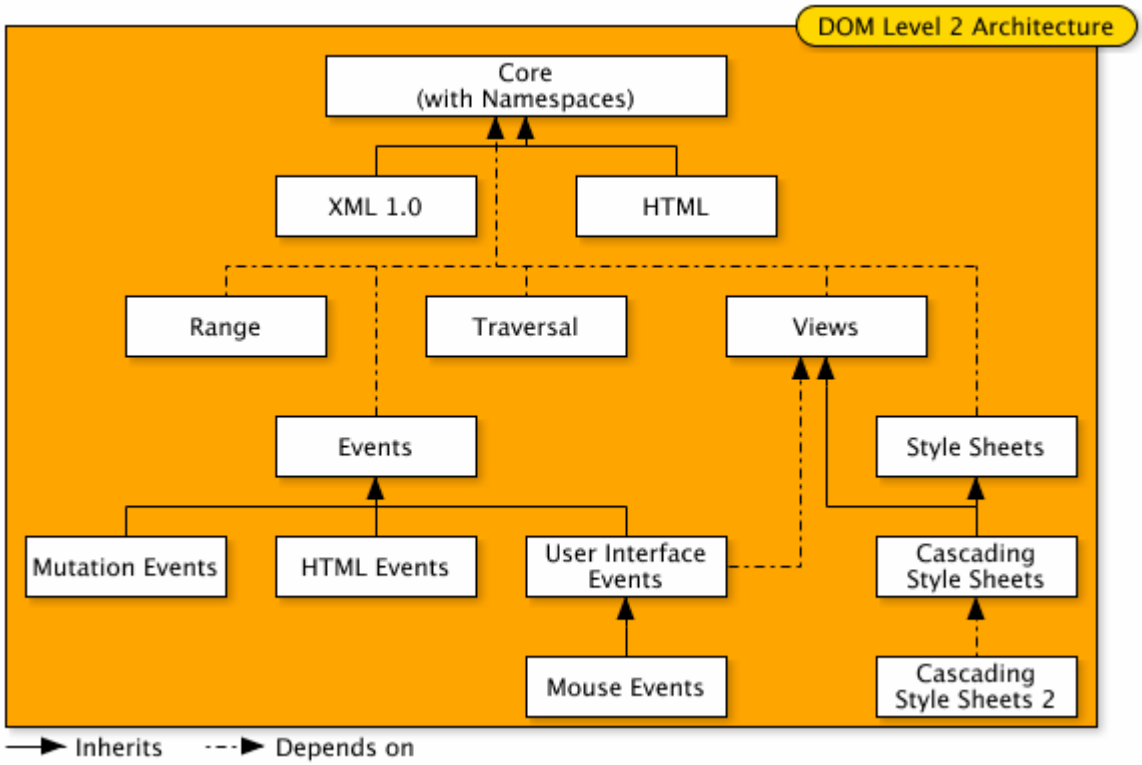
3.2.2 DOM Level 1



DOM Level 1 konzentriert sich auf XML und HTML Objektmodelle. Es enthält die Funktionalität um Dokumente einzulesen und zu manipulieren.
DOM Level 1 wurde am 1. Oktober 1998 als eine Recommendation vom W3C veröffentlicht.

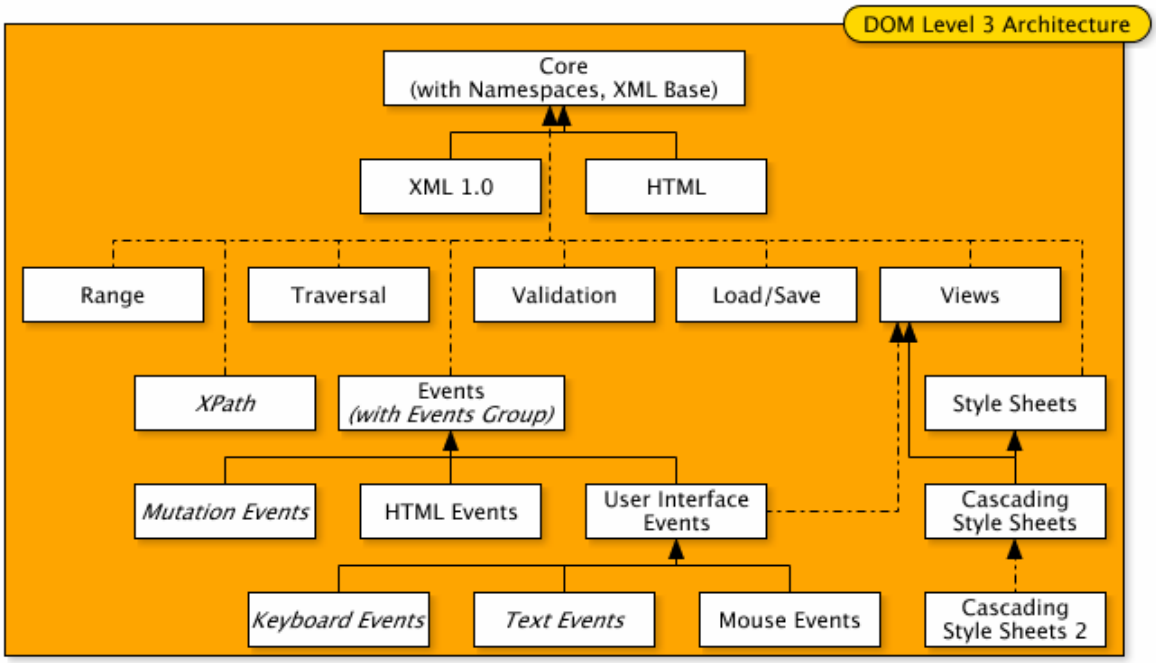
Ein Arbeitsentwurf für eine zweite Edition wurde am 29. September 2000 veröffentlich.

3.2.3 DOM Level 2



DOM Level 2 fügt DOM Level 1 ein Style Sheet Objektmodell hinzu und definiert ebenfalls die Funktionalität diese Style Informationen zu verändern.
Des Weiteren wurde noch ein Eventmodell definiert und die Unterstützung der XML Namespaces bereitgestellt.
Die DOM Level 2 Spezifikation wurde am 13. November 2000 als Recommendation des W3C veröffentlicht:
DOM Level 2 Core spezifiziert eine API um auf den Inhalt und die Struktur von Dokumenten zuzugreifen und diese zu aktualisieren. Die API enthält zusätzlich noch Interfaces welche zu XML gehören.
DOM Level 2 HTML beschreibt eine API, welche es erlaubt, HTML Dokumente in Struktur und Inhalt zu verändern. Dieser Teil der Spezifikation ist derzeit lediglich ein Arbeitsentwurf.
DOM Level 2 Views spezifiziert eine API für den dynamischen Zugriff und die Aktualisierung von der Darstellung eines Dokuments. Die Darstellung ist in diesem Falle eine wechselnde Repräsentation oder Präsentation eines Dokuments.
DOM Level 2 Style gibt eine API an, um auf so genannte content style sheets dynamisch zuzugreifen und diese zu aktualisieren.
DOM Level 2 Events spezifiziert eine API auf um Dokumentevents zuzugreifen.
DOM Level 2 Traversal-Range spezifiziert eine API zum dynamischen traversieren und identifizieren des Inhalts eines Dokuments.

3.2.4 DOM Level 3



DOM Level 3 spezifiziert Kontentmodelle (DTD and Schemas) sowie eine Dokumentvalidierung. Ebenso werden durch DOM Level 3 das Laden und Speichern eines Dokuments, die Dokumentdarstellung und Dokumentformatierung sowie Schlüsselfälle festgelegt. DOM Level 3 baut auf DOM Level 2 auf und unterstützt nun XML 1.1.
DOM Level 3 wurde am 7. April 2004 als Recommendation vom W3C veröffentlicht:
DOM Level 3 Core gibt eine API zum Zugreifen und Aktualisieren des Inhalts, der Struktur und des Designs eines Dokuments an.
DOM Level 3 Events vergrößert die Funktionalität der DOM Level 2 API, indem neue Interfaces und Eventsets hinzugefügt werden.
DOM Level 3 Load and Save beschreibt das Laden und Speichern eines Dokuments, Kontentmodelle (DTD und Schemas) und eine Unterstützung für die Dokumentvalidation in einer API.
DOM Level 3 Views and Formatting spezifiziert eine API für den dynamischen Zugriff und die Aktualisierung von der Darstellung eines Dokuments. Die Darstellung ist in diesem Falle wieder eine wechselnde Repräsentation oder Präsentation eines Dokuments.

3.2.5 W3C Spezifikationen in ihrer zeitlichen Entwicklung

Specification	Latest Draft	Proposed	Recommendation
DOM Level 1	20. Jul 98	18. Aug 98	01. Oct 1998
DOM Level 1 (SE)	29. Sep 00		
DOM Level 2 Core	10. May 00	27. Sep 00	13. Nov 2000
DOM Level 2 HTML	10. Dec 01	08. Nov 02	09. Jan 2003
DOM Level 2 Views	10. May 00	27. Sep 00	13. Nov 2000
DOM Level 2 Style	10. May 00	27. Sep 00	13. Nov 2000
DOM Level 2 Events	10. May 00	27. Sep 00	13. Nov 2000
DOM Level 2 Traversal-Range	10. May 00	27. Sep 00	13. Nov 2000
DOM Level 3 Requirements	26. Feb 04		
DOM Level 3 Core	07. Nov 03	05. Feb 04	07. Apr 2004
DOM Level 3 Events	07. Nov 03		
DOM Level 3 Load and Save	19. Jun 03	05. Feb 04	07. Apr 2004
DOM Level 3 Validation	05. Feb 03	15. Dec 03	27. Jan 2004
DOM Level 3 XPath	26. Feb 04		
DOM Level 3 Views	26. Feb 04		

3.3 Arbeitsweise des DOM

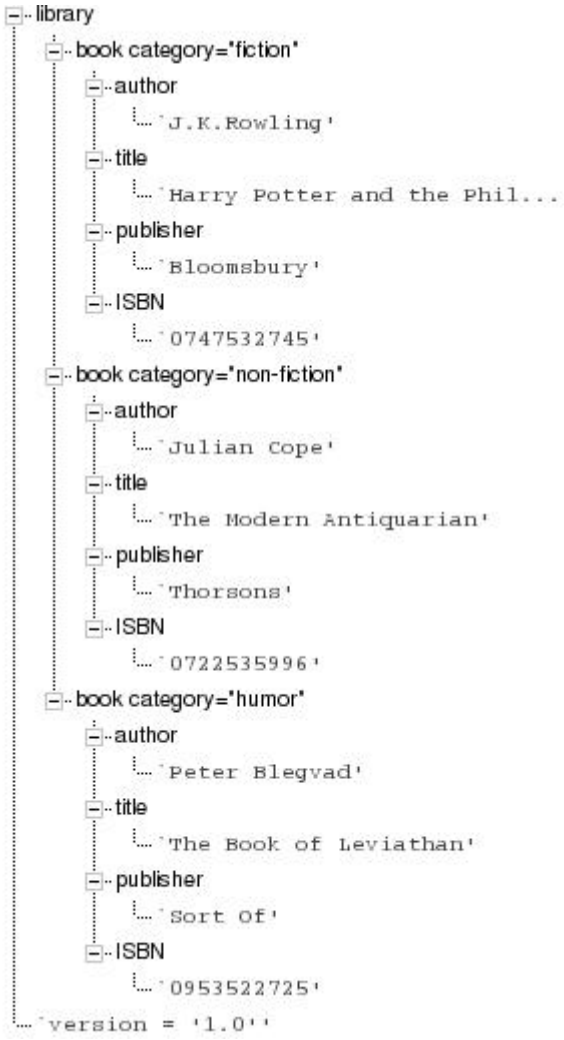
Das Document Object Model baut beim parsen des Dokuments einen Baum auf, den so genannten DOM-Tree. Dieser Baum steht dann im Speicher zur Verfügung und man kann mittels der Werkzeuge, die von der DOM API geliefert werden, auf den Baum zugreifen und diesen verändern.
Da ein Baum über das ganze Dokument aufgebaut ist, ist DOM recht speicherintensiv.

3.3.1 Beispiel

Gegeben ist folgendes XML Dokument:

```
<?xml version="1.0"?>
<library>
  <book category="fiction">
        <author>J.K.Rowling</author>
        <title>Harry Potter and the Philosopher's Stone</title>
        <publisher>Bloomsbury</publisher>
        <ISBN>0747532745</ISBN>
  </book>
  <book category="non-fiction">
        <author>Julian Cope</author>
        <title>The Modern Antiquarian</title>
        <publisher>Thorsons</publisher>
        <ISBN>0722535996</ISBN>
  </book>
  <book category="humor">
        <author>Peter Blegvad</author>
        <title>The Book of Leviathan</title>
        <publisher>Sort Of</publisher>
        <ISBN>0953522725</ISBN>
  </book>
</library>
```

DOM parst nun das Dokument und erstellt dadurch folgenden Baum im Speicher:



3.4 Wichtige Klassen der DOM API

Die wichtigsten Klassen in DOM sind `Node`, `Document`, `Element` und `NodeList`:

- `Node` repräsentiert einen einzelnen Knoten des DOM-Trees
- `Document` repräsentiert das gesamte Dokument. Dieses kann man auch als einen einzigen Knoten betrachten. Dies ist nicht verwunderlich, wenn man weiß, dass `Document` von `Node` abgeleitet ist.
- `Element` repräsentiert jedes XML Element. Diese Klasse ist ebenfalls von `Node` abgeleitet.
- `Node` enthält eine Eigenschaft, die `childNodes` genannt wird, welche vom Typ `NodeList` ist. Diese ist, wie der Name schon sagt, einfach eine Liste von Knoten, der nächsten Hierarchieebene.

3.5 Wann sollte man DOM anstatt XSLT nehmen?

XSLT ist gut geeignet, um Operationen auf einem ganzen Text-String durchzuführen, nicht aber um in Textdaten selbst etwas zu verändern. Sollte man also mit XSLT Substrings eines Dokuments verändern wollen, so ist dies nur sehr schwer oder gar nicht möglich.
Um dies zu verdeutlichen erweitern wir das XML Dokument aus Beispiel 3.3.1 um einige weitere Bücher:

```
<?xml version="1.0"?>
<library>
    <book category="fiction">
        <author>J.K.Rowling</author>
        <title>Harry Potter and the Philosopher's Stone</title>
        <publisher>Bloomsbury</publisher>
        <ISBN>0747532745</ISBN>
    </book>
    <book category="non-fiction">
        <author>Julian Cope</author>
        <title>The Modern Antiquarian</title>
        <publisher>Thorsons</publisher>
        <ISBN>0722535996</ISBN>
    </book>
    <book category="humor">
        <author>Peter Blegvad</author>
        <title>The Book of Leviathan</title>
        <publisher>Sort Of</publisher>
        <ISBN>0953522725</ISBN>
    </book>
    <book category="fiction">
        <author>O'Brien, Flann</author>
        <title>The Third Policeman</title>
        <publisher>Flamingo</publisher>
        <ISBN>0586087494</ISBN>
    </book>
    <book category="fiction">
        <author>Flann O'Brien</author>
        <title>At Swim-Two-Birds</title>
        <publisher>Penguin</publisher>
        <ISBN>0141182687</ISBN>
    </book>
    <book category="fantasy">
        <author>Pinnock, Jonathan</author>
        <title>Professional DCOM Application Development</title>
        <publisher>Wrox</publisher>
        <ISBN>1861001312</ISBN>
    </book>
</library>
```

Wie wir sehen, wurde hier das Namensformat von `<author>` nicht immer eingehalten, so würde Beispielsweise, wenn wir alle Werke von Flann O'Brien ausgeben wollten, sein Werk „The Third Policeman" nicht angezeigt. Dieses Problem könnten wir jetzt zum Beispiel dadurch lösen, das wir den Autornamen nochmals in `<firstname>` und `<lastname>` Elemente unterteilen.
Implementieren wir also die DOM API in ein Java Programm, was dies für uns erledigt. Wir beachten dabei aber lediglich die Struktur des XML Dokuments, nicht aber die DTD!

3.6 DOM Programmierung in JAVA

3.6.1 Der Code

Beginnend importieren wir alle vom Programm benötigten Klassen:

```
import java.io.FileReader;
import java.io.FileWriter;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import org.xml.sax.InputSource;
import org.apache.xml.serialize.XMLSerializer;
```

Seltsam erscheint hierbei nur die Klasse `org.xml.sax.InputSource`. Die Klasse `InputSource` ist aber eine gute Hilfe, mit den Apache Parsern für SAX und DOM, die Eingabe bereitzustellen.
Nun folgt als nächster Schritt die `main`-Methode: Sie erstellt ein neues Objekt DOMTransform. Die Methode `transform()` liest hierbei die Kommandozeilen-argumente ein und gibt das veränderte Dokument aus. Diese sollen bei unserem Programm (in der folgenden Reihenfolge) die Quelldatei, die Zieldatei und optional die Buchkategorie sein.

```
public class DOMTransform
{
 public static void main(String[] args) throws Exception
 {
       DOMTransform transformObj = new DOMTransform();
       transformObj.transform(args);
 }
```

Der folgende Teilausschnitt der Methode `transform()` überprüft lediglich die Anzahl der Kommandozeilenargumente und verpackt diese in Variablen. Gegebenenfalls werden einige Exceptions geworfen.

```
public void transform (String[] args) throws Exception
 {
       if (args.length == 0)
             throw new Exception ("No input file specified");

       String inFile = args[0];

       if (args.length == 1)
             throw new Exception ("No output file specified");

       String outFile = args[1];

       String catFilter = "all";

       if (args.length >= 3)
             catFilter = args[2];

       boolean fAll = false;

       if (catFilter.compareTo ("all") == 0)
             fAll = true;
```

Nun kommen wir zu dem Teil, wo wir das erste Mal mit dem DOM Parser in Berührung kommen. Der Zugriff auf diesen erfolgt durch Code, der eine völlige eigene Implementierung in Xerces fand. Der Parser wurde hier als `DocumentBuilder` Objekt implementiert. Durch die Methode `parse()` wird ein Dokument Objekt Modell generiert und ein Document Objekt zurückgegeben.
Der Grund dafür, dass diese Variante des Einlesens in Xerces eine völlig eigene Implementation fand, liegt einfach daran, dass zu der Zeit aus der das Beispiel stammt, dieser Parser erst Level 1 DOM beherrschte, und es dort keine Regelung dafür gab, wie das Dokument in den Speicher geladen wird.
Seit Level 2 DOM gibt es in neues Interface `DOMImplementation`, welches eine Methode `createDocument()` hat, welche dies standardisiert.
Neuere Versionen von Xerces unterstützen bereits DOM Level 2 und haben dieses Interface bereits implementiert. Wir wollen uns aber in diesem Beispiel auf DOM Level 1 beschränken.

```
System.out.println("Commencing transformation ...");

DocumentBuilderFactory factoryObj=DocumentBuilderFactory.newInstance();
DocumentBuilder domBuilderObj = factoryObj.newDocumentBuilder();

FileReader inFileObj = new FileReader(args[0]);
Document domObj = domBuilderObj.parse (new InputSource(inFileObj));
```

Der nächste Abschnitt des Programms erstellt ein `NodeList` Objekt, welches alle Bücherknoten enthält. Dies geschieht in 3 Schritten:

1. Es müssen alle Instanzen von `<library>` Elementen gefunden und in eine `NodeList` integriert werden. Normalerweise sollte hier nur ein Element vorhanden sein, aber wir sollten es dennoch beachten, dass es auch mehrere `<library>` Elemente geben könnte.
   Alternativ könnte man auch in der DTD sagen, dass es nur ein `<library>` Element geben kann. Dann müssten wir den Parser die Validierung des Dokuments überprüfen lassen. Wir benutzen hier die erste Variante.
2. Davon ausgehend, dass wir mehrere `<library>` Elemente haben können, müssen wir das erste dieser Elemente extrahieren. Dies geschieht durch die Methode `item()`, welche ein `Node` Objekt zurückgibt. Dieses muss in `Element` umgewandelt werden, um im nächsten Schritt Zugriff auf die Methoden des Element Interfaces zu haben.
3. Zum Schluss müssen wir noch die Methode `getElementsByTagName()` auf das `<library>` Element Objekt anwenden, um ein `NodeList` Objekt zu erzeugen, welches alle Instanzen von `<book>` Knoten enthält.

```
NodeList libraryListObj = domObj.getElementsByTagName ("library");
Element libraryObj = (Element) libraryListObj.item(0);
NodeList booksObj = libraryObj.getElementsByTagName ("book");
```

Als nächstes müssen wir die Anzahl der `<book>` Knoten herausfinden um damit eine Schleife erstellen zu können, welche jeden dieser Knoten bearbeitet.
Innerhalb der Schleife müssen wir zuerst das `Node` Objekt extrahieren, welche das einzelne Buch repräsentiert. Dafür verwenden wir die `item()` Methode des `NodeList` Objekts. Der zurückgegebene Wert muss dabei wieder in `Element` umgewandelt werden. Danach können wir den Wert des `category` Attributes extrahieren. Dafür verwenden wir die `getAttribute()` Methode.

```
int numBook = booksObj.getLength();
       for (int intBook = 0; intBook < numBook; intBook++)
       {
             Element bookObj = (Element) booksObj.item(intBook);
             String category = bookObj.getAttribute("category");
```

Wenn nun alle Kategorien ausgegeben werden sollen oder das Buch zu der gewählten Kategorie gehört, so müssen wir das `book` Objekt dementsprechend weiter bearbeiten.
Zuerst sollten wir dabei das `<author>` Element extrahieren. Dies machen wir auf dieselbe Weise, wie wir es bereits mit dem `<library>` Element gemacht haben: Wir erstellen also ein neues `NodeList` Objekt, welches alle `<author>` Elemente enthalten soll. Dies realisieren wir wieder durch die Methode `getElementsByTagName()`. Ausgehend davon, dass nur ein Element `<author>` existiert, können wir das erste dieser Elemente mit der `item()` Methode der `author NodeList` verwenden. Der Wert dieser Methode muss wieder nach `Element` umgewandelt werden.

```
if (fAll || (category.compareTo (catFilter) == 0))
 {
       NodeList authorsObj = bookObj.getElementsByTagName ("author");
       Element authorObj = (Element) authorsObj.item(0);
```

Nun sind wir endlich bei dem <author> Element angekommen, welches wir ja verändern wollten. Wenn das Dokument nun so strukturiert ist, wie wir es uns vorstellen, dann liegt der Text des <author> Knotens in seinem ersten Kind. Diesen Knoten können wir durch die `getFirstChild()` Methode des author Elements extrahieren. Diesen Wert, müssen wir nun in ein Text Objekt umwandeln. Dies ist eine weitere Klasse die von `Node` abgeleitet wird und repräsentiert ein rein textliches Objekt. Um dieses Text Objekt in einen String umzuwandeln benutzen wir seine `getData()` Methode. Nun haben wir einen String, der den Namen des Autors gespeichert hat. Diesen werden wir bald benutzen um zwei neue Kinder anzulegen. Das original Text Objekt können wir nun bedenkenlos löschen. Dazu verwenden wir die `removeChild()` Methode des author Elements.

```
Text textObj = (Text) authorObj.getFirstChild ();
String authorString = textObj.getData ();
authorObj.removeChild (textObj);
```

Nun müssen wir aus dem String, der den Namen des Autors enthält, den Vornamen und seinen Nachnamen extrahieren und jeweils in einer neuen Variable speichern. Dabei verwenden wir folgende Regeln:

- Finden wir in dem Namen ein Komma, gehen wir davon aus, dass die Struktur des Namens ‚Nachname, Vorname' ist.
  Beispiel: Bei ‚Doe, John' wäre der Vorname ‚John' und der Nachname ‚Doe'
- Befindet sich innerhalb des Namens ein Punkt, dann gehen wir davon aus, dass der Name Initialen enthält.
  Beispiel: Bei ‚J.K.Doe' wäre der Vorname ‚J.K.' und der Nachname ‚Doe'
- Ansonsten gehen wir davon aus, dass der Name wie üblich ‚Vorname Nachname' lautet.
  Beispiel: Bei ‚John Doe' Soll der Vorname ‚John' sein und der Nachname ‚Doe'

Das Extrahieren des Namens ist mit diesen Regeln recht einfach gehalten und reicht für unser Beispiel völlig aus, da es hier nicht darum geht komplizierte Namensparser zu implementieren, sondern darum DOM zu verwenden.

```
String firstName = "";
String lastName = authorString;

int comma = authorString.indexOf (',');

if (comma == -1)
{
	int period = authorString.lastIndexOf ('.');

	//Der Name enthält Initialen
	if (period != -1)
	{
		firstName = authorString.substring (0, period + 1);
		lastName = authorString.substring (period + 1);
	}
	else
	{
		int space = authorString.indexOf (' ');

		//Der Name hat das normale Format
		if (space != -1)
		{
			firstName = authorString.substring (0, space);
			lastName = authorString.substring (space + 1);
		}
	}
}

//Der Name enthält ein Komma
else
{
	lastName = authorString.substring (0, comma);
	firstName = authorString.substring (comma + 2);
}
```

Nun haben wir es fast geschafft! Wir müssen nun nur noch die neuen Kinder erstellen, welche den Vornamen und den Nachnamen enthalten sollen. Für jeden Namen benötigen wir die `createElement()` Methode des allumfassenden `Document` Objekts um ein freies `Element` Objekt mit dem entsprechenden Tag Namen zu erzeugen. Diesem `Element` Objekt erstellen wir nun noch einen reinen Textknoten als Kind. Dies machen wir mit der `createTextNode()` Methode von `Document`. Nun müssen wir diesen Knoten noch an das `Element` Objekt anhängen. Dies geschieht durch die Methode `appendChild()`. Nun müssen wir auch noch dieses `Element` Objekt an das author Element anhängen. Dazu verwenden wir wieder die `appendChild()` Methode.

```
 Element firstNameObj = domObj.createElement ("firstname");
 textObj = domObj.createTextNode (firstName);
 firstNameObj.appendChild (textObj);
 authorObj.appendChild (firstNameObj);

 Element lastNameObj = domObj.createElement ("lastname");
 textObj = domObj.createTextNode (lastName);
 lastNameObj.appendChild (textObj);
 authorObj.appendChild (lastNameObj);
}
```

Bisher haben wir die Möglichkeit noch nicht beachtet, dass ein Buch gar nicht zu der Kategorie gehört, die wir am Anfang angegeben haben. Sollte dies der Fall sein, so müssen wir dieses Buch löschen. Dies machen wir wieder durch die Methode `removeChild()`, nur diesmal auf dem library Element. Nun müssen wir noch die Anzahl der Elemente des Baumes um 1 erniedrigen, damit unsere Schleife vom Anfang noch richtig zählt.

```
else
 {
       libraryObj.removeChild (bookObj);

       numBook--;
       intBook--;
 }
} //End for
```

Nun muss nur noch das neue Dokument geschrieben werden. Wir verwenden dazu die in Xerces einmalige Klasse `XMLSerializer`. Dies realisieren wir dadurch, dass wir den Ausgabestrom dieser Klasse auf ein Standard Java `FileWriter` Objekt zeigen lassen. Dies geschieht mittels der `setOutputCharStream()` Methode. Zu guter letzt müssen wir nur noch die Ausgabe durch die Methode `serialize()` starten.

```
       FileWriter outFileObj = new FileWriter (outFile);
       XMLSerializer serializerObj = new XMLSerializer ();
       serializerObj.setOutputCharStream (outFileObj);
       serializerObj.serialize (domObj);

       System.out.println ("Transformation complete");
 }
}
```

3.6.2 Ausführung des Programms

Da unser Code ja nun endlich vollständig ist können wir das Programm jetzt einmal starten, um zu sehen, was geschehen ist.
Wenn dabei die drei Parameter *library.xml javalibrary.xml fiction* eingeben, so erhalten wir als Ausgabe:

```
<?xml version="1.0"?>
<library>
 <book category="fiction">
       <author>
         <firstname>J.K.</firstname>
         <lastname>Rowling</lastname>
       </author>
        <title>Harry Potter and the Philosopher's Stone</title>
        <publisher>Bloomsbury</publisher>
        <ISBN>0747532745</ISBN>
 </book>
 <book category="fiction">
  <author>
       <firstname>Flann</firstname>
       <lastname>O'Brien</lastname>
  </author>
  <title>The Third Policeman</title>
  <publisher>Flamingo</publisher>
  <ISBN>0586087494</ISBN>
 </book>
 <book category="fiction">
  <author>
       <firstname>Flann</firstname>
       <lastname>O'Brien</lastname>
  </author>
  <title>At Swim-Two-Birds</title>
  <publisher>Penguin</publisher>
  <ISBN>0141182687</ISBN>
 </book>
</library>
```

Wie wir sehen, funktioniert das Programm perfekt.
In unserem neuen XML Dokument haben wir nun nicht mehr das Problem, dass wir bei einer Suche nach ‚Flann O'Brien' nicht alle seine Bücher geliefert bekommen.8. Vergleich mit DOM

4. Fazit: Vergleich SAX ↔ DOM

SAX ist wesentlich schneller als DOM, da die Daten sofort an die Applikation geliefert werden, und da nicht erst ein Aufbau einer Datenstruktur erfolgt, kostet dies keine extra Zeit.
Da keinerlei Daten unnötig lange im Hauptspeicher behalten werden, verbraucht SAX im Gegensatz zu DOM fast keinen Speicher.
SAX bietet dafür aber keinen beliebigen Zugriff auf Daten, da das Dokument linear abgearbeitet wird. DOM hingegen ermöglicht eine einfache Manipulation des DOM-Trees (also des XML-Dokuments). Ein Sprung nach oben oder an beliebige Stelle ist nicht möglich.
SAX ist relativ kompliziert zu bedienen. Besonders bei komplexeren Anwendungen, kann die Implementierung Schwierigkeiten bereiten.
Eine Manipulation der Daten ist mit SAX nicht möglich, da das Schreiben von XML nicht unterstützt wird.
Somit sollte man SAX verwenden, wenn das Dokument nur einmal „durchzulesen“ ist, beispielsweise um es anzuzeigen.
DOM ist die besser Alternative bei Anwendungen, die auf das Dokument interaktiv zugreifen oder es mehrfach brauchen, da es ja als DOM-Tree im Speicher zur Verfügung steht.

5. Quellenangaben

- Professional XSL
- Professional XML
- Java und XML für Dummies
- http://www.wikipedia.de/
- http://www.software-kompetenz.de/
- http://www.w3.org/
- http://wwweickel.in.tum.de/lehre/Seminare/Hauptseminar/WS00/DOM/Presentation/paper.html
- http://www.w3schools.com/w3c/w3c_dom.asp
- http://www.saxproject.org
- http://www.oreilly.com/catalog/xmlnut2
- http://safari.oreilly.com