Erschienen in: Köhler, Werner/Wittenbrink, Heinz/Bormann, Carsten (Hrsg.): XML. - Berlin: TEIA Lehrbuch Verlag, 2003. S. 249-295.

[5] XML IN DER PRAXIS: DOKUMENTE PARSEN, VALIDIEREN UND VERARBEITEN

In dieser Lerneinheit werden die Kenntnisse über die Definition von Dokumenttypen vertieft und Werkzeuge zur Validierung von Dokumenten gegen eine DTD vorgestellt. Mit dem SIMPLE API FOR XML und dem DOCUMENT OBJECT MODEL werden zwei Schnittstellen für XML-Prozessoren eingeführt, die darauf aufbauende Anwendungsprogramme für die Weiterverarbeitung eines Dokuments nutzen können. In dem Zusammenhang werden das ereignisorientierte und das modellorientierte Parse-Verfahren diskutiert.

Im praxisorientierten Teil der Lerneinheit wird der Umgang mit Java-basierten XML-Parsern eingeübt. Dazu werden kurz die wichtigsten Konzepte der Programmiersprache wiederholt, und anschließend werden die von SAX bzw. DOM bereitgestellten Java-Schnittstellen vorgestellt. In eigenen kleinen Programmen wird die Erweiterung der bestehenden Komponenten um weitere Funktionen aufgezeigt.

5.1 Einleitung

In der vorhergehenden Lerneinheit haben wir uns damit befasst, wie Typen von XML-Dokumenten mittels einer DTD definiert werden können. Daraus ergibt sich die Frage, wie überprüft werden kann, ob ein konkretes XML-Dokument einem vorgegebenen Dokumenttyp entspricht und damit GÜLTIG (engl. „valid") ist. Dazu werden in der Praxis so genannte VALIDIERENDE PARSER eingesetzt, das sind Software-Komponenten bzw. selbständige Programme, die ein Dokument lesen und überprüfen können, ob es den in einer gegebenen DTD festgelegten Bildungsregeln entspricht. Ein Parser kann auch ein XML-Dokument auf bloße Wohlgeformtheit überprüfen, ohne zu validieren. In diesem Falle spricht man von einem NICHTVALIDIERENDEN PARSER. Im ersten Teil dieser Lerneinheit werden wir uns damit befassen, wie Parser funktionieren und wie man sie praktisch einsetzt.

Parser bilden auch eine wichtige Komponente bei der Verarbeitung von XML-Dokumenten durch Anwendungsprogramme. Der Parser analysiert das XML-Dokument, stellt fest, ob es wohlgeformt bzw. gültig ist, zerlegt es in seine syntaktischen Bestandteile und stellt diese Bestandteile einem Anwendungsprogramm zur weiteren Verarbeitung zur Verfügung. Damit Anwendungsprogramme Parserkomponenten benutzen können und mit ihnen XML-Dokumente verarbeiten können, gibt es so genannte Programmierschnittstellen (Application Programming Interfaces, APIs). Im zweiten Abschnitt dieser Lerneinheit werden wir die beiden für XML gebräuchlichsten Programmierschnittstellen SAX (Simple API for XML) und DOM (Document Object Model) kennen lernen und jeweils an einem praktischen Beispiel lernen, wie man diese Programmierschnittstellen für die Entwicklung eines Java-Anwendungsprogramms einsetzt.

5.2 Parsen von XML-Dokumenten

Ein Parser liest als Eingabe strukturierte Textdokumente und analysiert deren syntaktischen Aufbau anhand einer bestimmten Grammatik. Wir werden im Folgenden auf die unterschiedlichen Vorgehensweisen zwischen einem VALIDIERENDEN und einem NICHTVALIDIERENDEN Parser eingehen. Darüber hinaus werden wir zwei beim Parsen zur Anwendung kommende Verfahren zur Verarbeitung der Dokumentbestandteile behandeln, die auch später bei der Betrachtung der Programmierschnittstellen DOM und SAX eine Rolle spielen: das ereignisbasierte STREAMING-Verfahren und das TREE-BUILDING-Verfahren.

Validierendes Parsen

Von der Validierung eines XML-Dokuments spricht man, wenn die Struktur einer XML-Instanz gegen eine formale Strukturdefinition getestet wird. Die formale Strukturdefinition liegt meist in Form einer Dokumenttyp-Definition (DTD, *siehe Lerneinheit 4*) vor, möglich sind aber auch andere Strukturbeschreibungen, z.B. ein XML-Schema (siehe *Lerneinheit 10*). Neben der Überprüfung der strukturellen Anordnung der Elemente werden Namen und Typen der Attribute überprüft. Je nach Definition innerhalb der DTD können Attribute obligatorisch oder fakultativ sein. Ihr Wertebereich lässt sich (mit Einschränkungen) vorgeben, die Wertbelegung des Attributs erfolgt entweder in der XML-Instanz oder durch Vorgaben aus der DTD (etwa durch die Definition von Standardwerten). In der XML-Instanz, nicht jedoch in der DTD, wird festgelegt, welchen Elementtyp das Wurzelelement der XML-Instanz, das Dokumentelement, haben soll. Es ist also möglich XML-Dokumente zu definieren, die sich nur auf einen Ausschnitt einer DTD beziehen und die gegen diesen Ausschnitt validiert werden. Gelingt die Strukturüberprüfung einer XML-Instanz gegen eine DTD, so spricht man von einem „gültigen" oder VALIDEN XML-Dokument (jeweils in Bezug auf diese DTD).

Seite 208

Seite 509

Beim validierenden Parsen von XML-Dokumenten ergibt sich das Problem, dass in einem ersten Schritt die DTD geparst und intern repräsentiert werden muss. Dieser Schritt ist insofern problematisch, als die DTD in einer eigenen, nicht XML-konformen Syntax vorliegt. Ein validierender XML-Parser besteht also im Prinzip aus zwei Parsern: einem für die DTD und einem für die XML-Instanz. In einem weiteren Schritt muss dann das XML-Dokument in seine syntaktischen Bestandteile zerlegt werden (die LEXIKALISCHE ANALYSE). Es müssen Tags zerlegt und Elementnamen, Attributnamen und Attributwerte als syntaktische Einheiten identifiziert werden, Entity-Referenzen und Namensraum-Deklarationen aufgelöst und verarbeitet werden. Die lexikalische Analyse erfolgt in der Regel schrittweise (INKREMENTELL). Bereits erkannte Teile werden direkt an den dritten Schritt des Parsens weitergeleitet: die SYNTAKTISCHE ANALYSE. Hierbei wird überprüft, ob die bereits gefundenen Elemente und Attribute der Strukturdefinition innerhalb der DTD entsprechen. Erst wenn das komplette XML-Dokument verarbeitet wurde, kann der Parser entscheiden, ob ein valides Dokument vorliegt. Ein validierender Parser wird also durch die XML-Instanz (Festlegung des Wurzelelements) und die DTD gesteuert.

Nichtvalidierendes Parsen

Jede valide XML-Instanz ist immer auch eine WOHLGEFORMTE XML-INSTANZ (WELLFORMED XML), allerdings nicht zwangsläufig umgekehrt. Das Parsen wohlgeformter XML-Dokumente basiert auf der Basissyntax von XML:

- Für jedes XML-Dokument existiert genau ein Wurzelelement.
- Jedes Element muss ein Starttag und ein Endtag besitzen, d.h. jedes geöffnete Element muss in der Folge auch wieder geschlossen werden. Leere Elemente können abweichend von dieser Regel auch mit dem speziellen EMPTY-ELEMENT-TAG repräsentiert werden, das nicht durch ein Endtag abgeschlossen wird.

- Elemente dürfen sich nicht überschneiden, d.h. jedes andere Element befindet sich entweder komplett innerhalb oder komplett außerhalb des jeweils betrachteten Elements. Für jedes Element gilt also: Das Element umfasst das Starttag eines anderen Elements genau dann, wenn es auch das jeweils zugehörige Endtag umfasst.

Ein Element wird in einem wohlgeformten Dokument also durch seine Verwendung im Kontext und nicht durch eine entsprechende Regel der DTD definiert. Hieraus ergibt sich für den XML-Nutzer ein erheblicher Vorteil gegenüber dem ursprünglichen SGML-Ansatz, der zwingend eine DTD verlangte. Die Syntax der Tags ist ebenfalls genau festgelegt. Jedes Tag wird mit < und > geklammert, der Elementname muss ein gültiger Bezeichner (siehe *Lerneinheit 2*) sein, und ein schließendes Tag wird gebildet, indem dem Elementnamen ein / vorangestellt wird. Das spezielle Empty-Element-Tag wird mit der Zeichenkette „/>" geschlossen, damit es von einem nichtvalidierenden Parser als ein solches erkannt werden kann.

Seite 82

Ein nichtvalidierender XML-Parser muss also nur die Wohlgeformtheit eines XML-Dokuments überprüfen. Entsprechend muss für das Parsen keine DTD angegeben werden. Dadurch vereinfacht sich das Parsen erheblich. Die lexikalische Analyse entspricht dem oben beschriebenen Vorgang, die syntaktische Analyse reduziert sich jedoch auf die Überprüfung der Basissyntax. Daher benötigt ein nichtvalidierender XML-Parser wesentlich weniger Speicherplatz und Rechenzeit als ein validierender Parser.

Streaming und Tree-Building

Wie funktioniert nun die syntaktische Prüfung von XML-Dokumenten auf Wohlgeformtheit im einzelnen? Findet der Parser ein Starttag, so speichert er den Elementnamen, findet er ein weiteres Starttag, speichert er wiederum dessen Elementnamen und so weiter bis er auf ein Endtag stößt. Die gespeicherten Elementnamen bilden einen „Stapel" (engl. „Stack"), wobei der zuerst gefundene Name (das ist immer der Name des

Dokumentelements) zuunterst des Stapels liegt und der zuletzt gefundene Name zuoberst. Findet der Parser nun ein Endtag, so wird dessen Elementname mit dem obersten Elementnamen des Stacks verglichen. Stimmen die Namen überein, wird das Element vom Stack entfernt und der Parse-Vorgang wird fortgesetzt. Ist dies nicht der Fall, so ist das XML-Dokument nicht wohlgeformt und der Parser kann abbrechen. Ist der Stack am Ende des Parse-Vorgangs leer und das Ende der Eingabedaten erreicht, liegt ein wohlgeformtes XML-Dokument vor. Ein nichtvalidierender Parser wird im wesentlichen also nur durch die XML-Instanz gesteuert. Da der Parser das XML-Dokument als Strom von Ereignissen (engl. „event stream") wie dem Auftreten eines Starttags, von Zeichendaten, eines zugehörigen Endtags usw. behandelt und auf dem Stack immer nur temporäre Informationen ablegt, ohne intern eine vollständige Repräsentation des ganzen Dokuments zu erzeugen, spricht man bei diesem Vorgehen auch von STREAMING-Verfahren.

Demgegenüber baut das TREE-BUILDING-Verfahren zusätzlich eine Repräsentation des verarbeiteten XML-Dokuments als Baumstruktur (engl. „tree structure") auf. Alle wohlgeformten XML-Dokumente stellen Baumstrukturen dar. Auf der obersten Ebene existiert ein Wurzelelement. In dieses eingebettet ist eine Menge von Kindelementen. Betrachtet man diese Teile einer XML-Datei separat, erkennt man, dass auch diese Teilstrukturen wieder Bäume sind. Dies gilt ebenfalls für deren Kindelemente und für deren Kindelemente und so weiter. Erst wenn man auf der Ebene der Textdaten angelangt ist, existieren keine weiteren Kindelemente mehr. Ein Parser macht sich diese Form der Selbstähnlichkeit zunutze. Beginnend mit dem Wurzelelement (einem Baum) werden erst alle Teilbäume geparst, von diesen wiederum die Teilbäume usw. Diese Zerlegung wird solange fortgesetzt, bis die Ebene der Textdaten erreicht ist. Nun kann die Baumstruktur der jeweiligen Teilbäume aufgebaut werden und anschließend in den Gesamtbaum integriert werden (siehe dazu auch SCHRITT 4: REKURSIVE AUSGABEROUTINE).

Seite 255

Aufgrund der Selbstähnlichkeit der Baumstruktur setzen Parser hier in der Regel „rekursive Funktionen" ein. Der Begriff Rekursion bezeichnet in der Mathematik, Informatik und Linguistik Verarbeitungsverfahren, die sich im Laufe der Verarbeitung selbst wieder aufrufen. BEISPIEL 5.1 zeigt in Pseudo-Code eine rekursive Verarbeitungsvorschrift für die Ausgabe einer Baumstruktur.

▶ Beispiel 5.1: Beispiel für rekursive Methode

```
Funktion ausgabe(IN Knoten):
  schreibe Name(Knoten)
  für alle Kinder(Knoten): ausgabe
```

Um die rekursive Verarbeitung eines XML-Dokuments zu verdeutlichen, betrachten wir die in ABBILDUNG 5.1 gezeigte Baumstruktur.

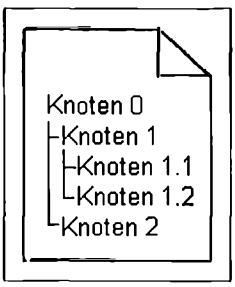

Abbildung 5.1: Beispiel für eine hierarchische Dokumentstruktur

Der Ablauf bei der Verarbeitung dieses Dokuments durch das Programm aus BEISPIEL 5.1 würde dann (wieder in Pseudo-Code) wie folgt aussehen:

```
ausgabe(Knoten 0):
  schreibe "Knoten 0"
  für Knoten 1, Knoten 2: ausgabe

  ausgabe(Knoten 1)
    schreibe "Knoten 1"
    für Knoten 1.1, Knoten 1.2: ausgabe

    ausgabe(Knoten 1.1)
      schreibe "Knoten 1.1"

    ausgabe(Knoten 1.2)
      schreibe "Knoten 1.2"

  ausgabe(Knoten 2)
    schreibe "Knoten 2"
```

Die Ausführung des Programms beginnt mit dem Aufruf der Funktion `ausgabe` mit dem Parameter `Knoten 0`, dem Wurzelknoten des Baums. Als erstes wird der Name dieses Knotens ausgegeben. Anschließend werden die Kindknoten ermittelt, und die Funktion `ausgabe` wird für jeden dieser Kindknoten einzeln aufgerufen. Als Parameter der Funktion wird diesmal jedoch nicht der Wurzelknoten des Baums angegeben, sondern der jeweils betrachtete Kindknoten des aktuellen Knotens.

Auf jeden der beiden Kindknoten des Wurzelknotens wird also nacheinander die Funktion `ausgabe` angewendet. Das bedeutet: Es wird jeweils der Name des Kindknotens geschrieben, und für jeden Kindknoten des jeweiligen Kindknotens (das sind also die „Enkel" des Wurzelknotens) wird wiederum dieselbe Funktion aufgerufen.

Diese Rekursion wiederholt sich mit den Nachfahren des Wurzelknotens so lange, bis ein Knoten keine Kindknoten mehr hat. Dann nämlich ist der entsprechende Funktionsaufruf beendet und die Verarbeitung setzt genau an der Stelle wieder ein, wo die Funktion aufgerufen wurde. Im Falle des ersten Kindknotens des Wurzelelements, `Knoten 1` bedeutet das, es wurden der Knotenname „Knoten 1" und die Namen der Kindknoten von `Knoten 1` geschrieben („Knoten 1.1" und „Knoten 1.2"). Nun ist die Aufgabe des Funktionsaufrufs `ausgabe(Knoten 1)` also beendet, und die Verarbeitung wird genau da fortgesetzt, wo der Aufruf stattfand: mitten in der Zeile `für alle...` bei der Verarbeitung von `Knoten 0`. Da die Knoten dort in der angegebenen Reihenfolge verarbeitet werden, folgt als nächstes also der Aufruf der Funktion `ausgabe` mit dem Parameter `Knoten 2`.

Da `Knoten 2` keine Kindknoten hat, wird lediglich der Name des Knotens ausgegeben, und der Funktionsaufruf ist anschließend beendet. Da nun auch für `Knoten 0` nichts mehr zu tun ist, kehrt schließlich auch die Funktion `ausgabe(Knoten 0)` wieder zurück. Die rekursive Verarbeitung der Baumstruktur ist damit beendet.

Die Rekursion bildet ein mächtiges Instrument zur Verarbeitung von strukturierten Daten. In vielen Fällen stellt sie jedoch nicht die effizienteste Möglichkeit zur Lösung eines solchen Problems dar. Daher versucht man oft, diese rekursiven Funktionsaufrufe zu umgehen und statt dessen sogenannte **SCHLEIFEN** zu verwenden. In den Programmbeispielen dieser Lerneinheit finden sich daher nur selten Funktionsaufrufe, wie sie hier vorgeführt wurden. Die Schleifenkonstrukte lassen sich allerdings leicht in rekursive Funktionsaufrufe umwandeln.

5.3 Validierung von XML-Dokumenten in der Praxis

Noch vor wenigen Jahren gab es lediglich eine Handvoll brauchbarer XML-Parser. Gerade validierende Systeme waren nicht ohne weiteres frei verfügbar. Diese Situation hat sich im Laufe der Zeit erheblich verbessert. Im Prinzip gibt es heute für nahezu jede relevante Programmiersprache einen XML-Parser. Die Mehrzahl dieser Programme unterliegen der GNU Public License (GPL), sind damit frei verfügbar und liegen im Quellcode vor. Zumindest die Verarbeitung wohlgeformter XML-Dokumente gelingt damit in den meisten Programmiersprachen problemlos.

www.gnu.org/licenses/licenses.html

Integrierte XML-Systeme und Webbrowser

Der Einsatz eines integrierten XML-Editors stellt die einfachste Möglichkeit dar, ein XML-Dokument zu validieren. Kommerzielle Systeme, wie z.B. XML-Spy, arbeiten mit einem eingebauten XML-Parser. Ein simpler Tastendruck startet die Validierung. Treten Fehler auf, springt der Editor an die entsprechende Stelle im XML-Dokument, markiert die Fehlerstelle farbig und gibt eine entsprechende Fehlermeldung aus. Ein Nachteil derartiger integrierter Systeme ist neben den hohen Anschaffungskosten die Komplexität ihrer Bedienung. Eine preisgünstige und sehr leistungsfähige Alternative stellt der in *Lerneinheit 1* vorgestellte Editor Emacs mit entsprechenden Zusatzmodulen dar.

Seite 16

Eine weitere vergleichsweise einfache Möglichkeit, eine XML-Datei zu überprüfen, bieten einige Webbrowser (z.B. Mozilla, Konqueror, Internet Explorer). Auch in diese Systeme sind XML-Parser integriert und werden aktiviert, sobald ein XML-Dokument eingelesen wird. Leider sind die eingebauten Parser nicht in allen Fällen konform mit XML 1.0. So reagiert z.B. der Internet Explorer bei der Validierung eines XML-Dokuments nicht auf die fehlerhafte Verwendung von Attributen. Sowohl die Benutzung nicht definierter Attribute als auch die Auslassung von obligatorischen Attributen (`#REQUIRED`) wird ignoriert. Das fehlerhafte Dokument wird vom Internet Explorer als quasi-valides XML-Dokument akzeptiert, obwohl es tatsächlich nicht valide ist.

Neben den integrierten Systemen gibt es eine Reihe von eigenständigen („standalone") Parsern, also Parsern, die unabhängig von einer sie einbettenden Anwendung aufgerufen werden können. Beispiele hierfür sind NSGMLS oder MSXML, der Microsoft XML-Parser, der auch im Internet Explorer verwendet wird. Für den Benutzer bedeutet dies, dass die gewohnte mausbasierte Interaktion mit dem Computer verlassen werden muss und ein Kommandozeileninterpreter zum Einsatz kommt. Alle Anweisungen an den Rechner erfolgen hier durch Eingaben über die Tastatur.

Praktische Validierung mit nsgmls

www.jclark.com/sp/

Im Folgenden werden wir den Parser NSGMLS verwenden. Dieser SGML-Parser basiert auf den Arbeiten von James Clark und kann in der jeweils neuesten Version von seiner Website heruntergeladen werden. Der Parser ist dort in dem Paket SP enthalten. Er wurde über die Jahre schrittweise weiter verbessert und ist inzwischen Teil des Open-Source-Projekts OPENJADE. Der Parser trägt dort den Namen ONSGMLS und ist entsprechend im Paket OPENSP zu finden. Gegenüber anderen Parsern ist nsgmls vergleichsweise schnell.

http://openjade.sourceforge.net/

Beim Umgang mit NSGMLS ist zu beachten, dass wir es mit einem genuinen SGML-Parser zu tun haben, der neben anderen SGML-Anwendungen auch XML 1.0 als Teilmenge von SGML unterstützt. Aus der Unterstützung von SGML resultieren gewisse Eigenheiten, z. B. bei Fehlermeldungen, auf die wir an den entsprechenden Stellen hinweisen werden. Für das Parsen verwenden wir das folgende Beispieldokument:

▶ Beispiel 5.2: Beispieldokument Planetensystem

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE planetensystem SYSTEM "planetensystem.dtd">
<planetensystem stern="Sonne">
  <planet>
    <name>Erde</name>
    <durchmesser>12756 km</durchmesser>
    <entfernung>149.6 Mill. km</entfernung>
    <umlaufzeit>365d 6h</umlaufzeit>
  </planet>
</planetensystem>
```

Da wir im Umgang mit nsgmls zeigen wollen, welche Meldungen der Parser bei Fehlern ausgibt, benutzen wir für den ersten Aufruf des Parsers die folgende DTD, in der ein kleiner Fehler eingebaut ist:

▶ Beispiel 5.3: Fehlerhafte DTD Planetensystem

```
<!ELEMENT planetensystem (planet)+>
<!ATTLIST planetensystem
          stern CDATA #REQUIRED>
<!ELEMENT planet (name, durchmesser, entfernung, umlaufzeit)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT durchmesser (#PCDATA)>
<!ELEMENT entfernung (#PCDATA, (min | max)?)*>
<!ELEMENT max (#PCDATA)>
<!ELEMENT min (#PCDATA)>
<!ELEMENT umlaufzeit (#PCDATA)>
```

Aufrufen des Parsers nsgmls

Der Parser wird durch folgende Befehlsfolge gestartet:

```
nsgmls -s -wxml xml.dcl planetensystem.xml
```

In diesem Beispiel wird die Datei `planetensystem.xml` geparst. Durch die Kommandozeilenoptionen `-s` und `-w` wird der Parser konfiguriert: `-s` (`silent`) unterdrückt die Standardausgabe, `-wxml` (`warning`) schaltet den Parser in den speziellen Warnungsmodus für XML. Die Datei `xml.dcl` enthält die SGML-Deklaration für XML und muss als Argument für das Parsen von XML-Dokumenten zwingend mitübergeben werden. Sie ermöglicht dem SGML-Parser NSGMLS, die XML-Basissyntax zu verarbeiten und befindet sich normalerweise in dem Unterverzeichnis PUBTEXT der Distribution.

Ist das XML-Dokument valide, so liefert die Eingabe scheinbar kein erkennbares Ergebnis. Der Parser verarbeitet das XML-Dokument und kehrt kommentarlos zur Eingabeaufforderung zurück. Eventuell werden einige Warnungen ausgegeben, z.B.:

```
nsgmls.exe:xml.dcl:1:W: SGML declaration was not implied
nsgmls.exe:planetensystem.dtd:7:30:W: #PCDATA in seq group
```

Man erkennt, dass die Meldungen des Parsers den Namen der Datei angeben, gefolgt von der Position (Zeile, ggf. Spalte) in der ein mögliches Problem existiert, gefolgt von dem Buchstaben W, der anzeigt, dass es sich hierbei um eine Warnung handelt. Abgeschlossen wird die Meldung mit einer kurzen (englischen) Beschreibung der Warnung.

Die erste Meldung des Parsers ist rein SGML-spezifisch und kann in unserem Zusammenhang ignoriert werden. Die zweite Meldung ist eine Warnung, dass in der Datei `planetensystem.dtd` in Zeile 9, Spalte 30 der Typ `#PCDATA` in einer Sequenz verwendet wird. In der Tat wird in der DTD an dieser Stelle `mixed content` definiert, und der Parser informiert den Benutzer über diesen Umstand. Eine Warnmeldung des Parsers führt

also nicht zum Abbruch der Verarbeitung, sondern informiert den Anwender über mögliche Probleme innerhalb der geparsten Dateien. Hier sei noch einmal betont, dass es sich bei den aufgeführten Warnungen um SGML-WARNUNGEN handelt. Hinsichtlich XML ist die zweite Meldung als FEHLER zu interpretieren, weil für die Deklaration von MIXED CONTENT ausschließlich der Konnektor | erlaubt ist. Wir ändern deshalb die Zeile 9 in der DTD und schreiben stattdessen:

```
<!ELEMENT entfernung (#PCDATA|min|max)*>
```

Im Folgenden werden wir in das Beispieldokument einige Fehler einbauen, um zu sehen, wie der Parser darauf reagiert.

Fehlerhaftes Endtag

Im ersten Schritt verstoßen wir gegen die Basissyntax von XML und verändern im Dokument den Namen eines Endttags
(aus `</planetensystem>` wird `</system>`),
so dass die Schachtelung der Elemente fehlerhaft ist. Neben den schon beschriebenen Warnungen liefert nsgmls drei Meldungen:

```
nsgmls.exe:planetensystem.xml:10:8:E: end tag for
   element "system" which is not open
nsgmls.exe:planetensystem.xml:10:10:E: end tag
   for "planetensystem" omitted, but OMITTAG NO
   was specified
nsgmls.exe:planetensystem.xml:3:0: start tag was
   here
```

Der Aufbau der Fehlermeldungen entspricht den schon besprochenen Warnungen, allerdings wird das W (warning) durch ein E (error) ersetzt. Wie man erkennt, zeigt die erste Fehlermeldung an, dass in Zeile 10, Spalte 8 ein Endtag `system` identifiziert wurde, für das kein entsprechendes Starttag existiert. Die zweite Meldung zeigt an, dass in Zeile 10,

Seite 58

Spalte 10, das Endtag für das Element `planetensystem` weggelassen (engl. „omitted") wurde. Wir erinnern uns *(siehe Lerneinheit 1)*, dass es in SGML zwar erlaubt ist, Endtags wegzulassen, dies für XML jedoch in keinem Falle erlaubt ist. Der SGML-Deklaration für XML (`xml.dcl`) entnimmt der Parser die Information, dass für das geparste Dokument die Auslassung von Tags nicht erlaubt ist (`OMITTAG NO`). Die dritte Meldung schließlich ist weder eine Warnung, noch eine Fehlermeldung. Hier wird die Position (Zeile 3, Spalte 0) des Starttags ausgegeben, für welches das Endtag fehlt. In der Datei findet man an dieser Position das Tag `<planetensystem>`. Der Parser beschreibt also den einen Fehler mehrmals in unterschiedlicher Weise und gibt sehr genaue Angaben darüber aus, wo in der Datei der Fehler auftritt. Damit wird die Fehlersuche sehr einfach: in der Regel gibt die erste Fehlermeldung die genaue Position des Fehlers an.

Es ist wichtig zu verstehen, dass der Parser schon bei nur einem einzigen Fehler in dem XML-Dokument eine Menge von Fehlermeldungen ausgeben kann. Ein Fehler kann die Gesamtstruktur eines XML-Dokuments zerstören. Tritt der Fehler innerhalb der ersten Zeilen eines großen XML-Dokuments auf, kann potentiell eine große Anzahl von so genannten FOLGEFEHLERN entstehen. Aus diesem Grund sollte man immer versuchen, die erste ausgegebene Fehlermeldung des Parsers zu bearbeiten und den dort gemeldeten Fehler zu entfernen. Ist der Fehler erkannt und behoben, sollte man anschließend erneut parsen und so den nächsten Fehler aufspüren.

Undefiniertes Attribut

Im zweiten Schritt verstoßen wir gegen die Vorgaben der DTD, indem wir für das Element `planetensystem` das obligatorische Attribut `stern` auslassen. Der Parser liefert das folgende Ergebnis:

```
nsgmls.exe:planetensystem.xml:3:15:E: required
   attribute "stern" not specified
```

Die Meldung zeigt klar an, dass in Zeile 3, Spalte 15 ein Attribut fehlt. Der Parser gibt zusätzlich an, welches Attribut fehlt, in diesem Fall `stern`. Fügt man an der gleichen Stelle ein nicht definiertes Attribut ein, meldet der Parser:

```
nsgmls.exe:planetensystem.xml:3:37:E: there is no
   attribute "zuviel"
```

Auch hier ist die Fehlermeldung sehr klar. In der DTD ist kein Attribut mit dem Namen `zuviel` definiert, also darf es auch nicht in Zeile 3, Spalte 37 verwendet werden.

Tag nicht mit > abgeschlossen

Im dritten Schritt erzeugen wir fehlerhaftes XML, indem wir ein Tag nicht ordnungsgemäß mit > schließen. Wir löschen dazu die Endmarkierung des Starttags für das Element `umlaufzeit` und schreiben also:

```
<umlaufzeit 365d 6h</umlaufzeit>
```

In die gleiche Fehlerklasse fällt auch die Auslassung eines Anführungszeichens (") am Ende eines Attributwertes. Fehler dieser Art können zu einer Menge von Folgefehlern führen. In diesem Fall werden folgende Fehlermeldungen ausgegeben:

```
nsgmls.exe:planetensystem.xml:8:14:E: "365d" is
   not a member of a group specified for any
   attribute
nsgmls.exe:planetensystem.xml:8:19:E: "6h" is not
   a member of a group specified for any attribute
nsgmls.exe:planetensystem.xml:8:21:E: unclosed
   start-tag requires SHORTTAG YES
```

Die eigentliche Ursache (`unclosed start-tag`) und den genauen Fundort des Fehlers (Zeile 8, Spalte 21) zeigt uns die dritte Meldung an. Die ersten beiden Meldungen sind etwas komplizierter und wiederum SGML-spezifisch: Da der Parser das schließende „>" des Starttags nicht findet, versucht er die auf den Elementnamen folgende Zeichenkette (eigentlich der textuelle Inhalt des Elements: in diesem Fall die Zeichenfolge „`365d 6h`") sinnvoll zu interpretieren. In diesem Fall interpretiert der Parser die Zeichenfolgen „`365d`" und „`6h`" jeweils als Attributwerte, die aber in der DTD durch ein entsprechendes Aufzählungsattribut (NMTOKEN-Gruppe) hätten definiert werden müssen. Da dies nicht der Fall ist, resultiert daraus die Fehlermeldung. Mit der dritten Fehlermeldung hat der Parser das fehlende > des Starttags erkannt. Da in SGML der schließende Tagbegrenzer (hier das Zeichen >) durchaus weggelassen werden kann, z.B. wenn ein anderes Tag unmittelbar folgt, diese Art der „Abkürzung" von Tags aber in XML nicht erlaubt ist, gibt der Parser hier die Beschreibung „`... requires SHORTTAG YES`" aus.

DTD nicht gefunden

Als letztes Beispiel für einen gängigen Fehler verändern wir die Dokumenttyp-Deklaration zu Beginn des XML-Dokuments. Wir ändern den Namen der DTD von `planetensystem.dtd` nach `system.dtd`. Diese Datei existiert nicht. Der Parser muss also versuchen, das Dokument gegen eine nicht existierende DTD zu validieren. Nach Aufruf von nsgmls werden für die Beispieldatei eine Reihe von Fehlermeldungen ausgegeben, wie z. B.:

```
nsgmls.exe:planetensystem.xml:3:22:E: there is no
   attribute "stern"
nsgmls.exe:planetensystem.xml:3:29:E: element
   "planetensystem" undefined
nsgmls.exe:planetensystem.xml:4:8:E: element
   "planet" undefined
```

Da die DTD nicht existiert, ist keines der Elemente und Attribute der DTD definiert. Entsprechend wird für jedes einzelne verwendete Element eine Fehlermeldung erzeugt. Je nach Größe der Datei kann solch ein Fehler zu einer großen Anzahl Fehlermeldungen führen. Der Parser bricht per Voreinstellung die Ausgabe bei mehr als 200 Fehlermeldungen ab.

Prüfen der Wohlgeformtheit

Selbstverständlich kann man mit nsgmls auch lediglich die Wohlgeformtheit von XML-Dokumenten überprüfen. Hierzu muss die Dokumenttyp-Deklaration aus dem XML-Dokument entfernt werden. Der Parser quittiert den fehlenden Bezug zur DTD zwar mit einer Fehlermeldung:

```
nsgmls.exe:planetensystem.xml:2:0:E: no document type
   declaration; will parse without validation
```

Das Dokument wird trotzdem vollständig geparst und dabei auf seine Wohlgeformtheit hin überprüft. Treten keine weiteren Fehlermeldungen mehr auf, ist das Dokument wohlgeformt. Alternativ hierzu kann man nsgmls auch in den nichtvalidierenden Modus schalten. Dies bewirkt die Kommandozeilenoption `-wno-valid`.

5.4 Verarbeitung von XML-Dokumenten unter Verwendung von XML-APIs

Wir haben im vorangehenden Teil der Lerneinheit gesehen, wie man einen Parser einsetzen kann, um die Wohlgeformtheit bzw. Gültigkeit von XML-Dokumenten zu überprüfen. Nun werden wir sehen, wie man Parser in Anwendungsprogrammen einsetzen kann, um XML-Dokumente zu verarbeiten. Zunächst wird erläutert, was unter einer Programmierschnittstelle zu verstehen ist. Anschließend werden die beiden für XML gängigsten Programmierschnittstellen **SAX** und **DOM** vorgestellt und Danach wird die praktische Verwendung von SAX und DOM in der Programmiersprache Java anhand einiger leicht nachzuvollziehender Beispiele dargestellt.

Was sind Programmierschnittstellen (APIs)?

Die Benutzung von XML-Parsern ist häufig an die Verwendung einer spezifischen Programmiersprache (z.B. Java, C, C++, Perl, Python etc.) und die hierfür verfügbaren Parser-Bibliotheken gebunden. Um den Einsatz unterschiedlicher Parser-Bibliotheken zu ermöglichen und zu vereinfachen, wird dem Entwickler eine dokumentierte PROGRAMMIERSCHNITTSTELLE (APPLICATION PROGRAMMING INTERFACE, API) zur Verfügung gestellt. Eine Programmierschnittstelle besteht aus einer Reihe von öffentlichen Funktionen, Variablen und Konstanten. Für den Programmierer abstrahiert die Schnittstelle die internen Abläufe der Programmbibliothek. Die Konzeption einer Schnittstelle erlaubt es so, alternative Bibliotheken in den entwickelten Programmen zu nutzen, ohne den Code des Anwendungsprogramms verändern zu müssen. Die einzige Voraussetzung hierbei ist, dass die neu eingesetzte Programmbibliothek die exakt gleiche Schnittstelle hat, und zwar sowohl in Hinblick auf die zur Verfügung gestellten Methoden (SYNTAX des APIs), als auch in Hinsicht auf deren tatsächlich durchgeführte Funktion (SEMANTIK des APIs).

DOM und SAX , die beiden gängigsten Programmierschnittstellen für XML, sind hierbei noch eine Abstraktionsstufe höher anzusiedeln. Insbesondere DOM definiert eine völlig plattformunabhängige und programmiersprachenunabhängige Schnittstelle, die erst durch so genannte „Language Bindings" an eine konkrete Programmiersprache „gebunden" wird. Mittlerweile trifft das auch für SAX zu, obwohl SAX anfangs als reines Java-API entwickelt worden ist. Neben Implementierungen von DOM oder SAX für die meisten gängigen Skriptsprachen und Programmiersprachen gibt es – vor allem für Java – eine Reihe weiterer APIs, u.a. JDOM und JAXP, auf die wir aber im Rahmen dieser Lerneinheit nicht weiter eingehen können.

Im weiteren Verlauf dieser Lerneinheit werden wir ausschließlich Beispiele aus dem Java-Umfeld verwenden. Es sei deshalb ausdrücklich betont, dass auch für andere Programmiersprachen SAX- und DOM-Bibliotheken zur Verfügung stehen, die genauso funktionieren und deren Verwendung in den jeweiligen Sprachen analog zu den vorgestellten Java-Beispielen erfolgt. Natürlich verwendet man bei der Entwicklung eines Anwendungsprogramms die spezifische Syntax der jeweiligen Programmiersprache, das grundlegende Verwendungskonzept der Bibliotheken bleibt jedoch gleich.

SAX und DOM im Vergleich

SAX und DOM werden im allgemeinen als alternative Ansätze für die Verarbeitung von XML-Dokumenten über eine definierte Programmierschnittstelle gesehen. Ein wesentlicher Unterschied zwischen beiden APIs ist, dass DOM vor der Weiterverarbeitung eines XML-Dokuments dessen vollständige Umsetzung in eine Baumstruktur vorsieht, während SAX dies nicht tut. Der Ansatz von DOM kann als MODELLORIENTIERT bezeichnet werden, gegenüber dem EREIGNISORIENTIERTEN Ansatz von SAX. Während SAX aber auch nicht viel mehr zur Verfügung stellt als eine solche ereignisorientierte Schnittstelle, bietet DOM ein viel weitreichenderes Spektrum an Verarbeitungsmöglichkeiten.

Ein SAX-Parser liest ein XML-Dokument als Eingabestrom (INPUT STREAM) und erzeugt während dieses Einlesevorgangs „Ereignisse" (EVENTS) für die syntaktischen Merkmale des Dokuments. Jedes syntaktische Merkmal – wie Anfang/Ende des Dokuments, Anfang/Ende eines Elements, Antreffen einer Verarbeitungsanweisung usw. – ist ein besonderes Ereignis, das durch das implementierte Interface an das Anwendungsprogramm weitergegeben werden kann. Ein Anwendungsprogramm kann also unmittelbar auf ein Ereignis des Eingabestroms reagieren. Bei diesem Streaming-Verfahren wird während des Parsens des Dokuments immer nur eine partielle Repräsentation des XML-Dokuments in den Speicher eingelesen. Hierdurch wird es möglich, nahezu beliebig große XML-Dokumente zu verarbeiten, da die Größe des Hauptspeichers nur eine geringe Rolle spielt. Streaming erweist sich besonders dann als nützlich, wenn nur lokal begrenzte Transformationen auf der Baumstruktur durchgeführt werden müssen, etwa beim direkten Umsetzen eines bereits geeignet strukturierten XML-Dokuments in HTML.

Im Gegensatz hierzu liest ein DOM-Parser das XML-Dokument vollständig in den Speicher und baut dabei ein Baummodell des Dokuments auf (Tree-Building). Dadurch ist bei der Verarbeitung von großen XML-Dokumenten ein DOM-Parser wesentlich langsamer als ein SAX-Parser. Darüber hinaus schränkt das Tree-Building-Verfahren auch die Größe der verarbeitbaren XML-Dokumente ein. Es hat jedoch den Vorteil, dass nach dem Einlesen unmittelbar auf alle Teilstrukturen des XML-Dokuments zugegriffen werden kann und diese verändert werden können. Dies ist beispielsweise für Sortierungen nützlich.

Verwendung von Java

Um im Folgenden die Programmbeispiele nachvollziehen zu können, ist es nicht unbedingt notwendig, Java zu kennen; hilfreich ist es allemal. Für Java-Neulinge sollen hier noch ein paar Hinweise zum prinzipiellen Aufbau von Java-Programmen gegeben werden, von denen wir hoffen, dass sie das Verständnis für die Beispiele erhöhen. Wenn Sie bereits über Java-Kenntnisse verfügen, können Sie diesen Abschnitt überspringen.

Java ist eine von Sun Microsystems entwickelte OBJEKTORIENTIERTE Programmiersprache. Sun stellt dem Programmierer eine komplette Programmierumgebung (das Java Development Kit, JDK, die im April 2003 aktuelle Version trägt die Versionsnummer 1.4) kostenlos zur Verfügung. Die Hauptkomponenten sind dabei der Compiler `javac` und der Interpreter `java`. Mit dem Compiler wird ein Programm in eine für die maschinelle Verarbeitung geeignetere Repräsentation (Bytecode) übersetzt, mit dem Interpreter kann man ein so vorliegendes Programm dann ausführen.

Das Klassenkonzept von Java

Java-Programme werden in sogenannten KLASSEN organisiert. Die Klassen bestehen aus einer Deklaration und einem Code-Block (Körper), in dem die Eigenschaften (Variablen) und Methoden (Funktionen) der Klasse definiert werden. In Java verwendet man die geschweiften Klammern { und} , um den Anfang und das Ende eines Code-Blocks zu markieren. Die Vereinbarung des Klassennamens erfolgt mit Hilfe des Schlüsselworts `class`. Komplexere Java-Programme bestehen aus vielen Klassen und benutzen so genannte Klassenbibliotheken, die wiederum in Paketen organisiert sind. In den Klassenbibliotheken und Paketen sind Gruppen von Klassen zusammengefasst, die einem gemeinsamen Zweck dienen. Die Klassenbibliotheken von Java selbst befinden sich z.B. in einem Paket, das `java` heißt und aus vielen kleineren Paketen besteht. Wir werden gleich sehen, wie in ein neu entworfenes Programm Klassen aus anderen Paketen importiert werden können.

An dieser Stelle ist zunächst einmal wichtig, dass Klassen, die über Paketgrenzen hinweg von anderen Klassen genutzt werden sollen, mit dem Schlüsselwort `public` (deutsch „öffentlich") gekennzeichnet werden müssen. Pro Datei darf es nur eine solche öffentliche Klasse geben. Hierbei ist zusätzlich die folgende Konvention zu beachten: Die Datei, in der eine veröffentlichte Java-Klasse gespeichert wird, MUSS EXAKT den gleichen Namen tragen wie der mittels `public class` vereinbarte Klassenname, ergänzt um die Dateiendung `.java`. So muss z.B. eine Klasse, die als `public class MySAXParser` deklariert worden ist, in der Datei `MySAXParser.java` abgespeichert werden. Innerhalb der folgenden Beispielprogramme wird daher immer nur eine Klasse pro Datei definiert.

Import von Klassen-Definitionen

Um Funktionen von externen Programm-Bibliotheken verwenden zu können (in unserem Fall beispielweise Methoden von Xerces für SAX und DOM), müssen diese Funktionen bzw. Bibliotheken in das Java-Programm importiert werden. Dies geschieht für jede Bibliothek einzeln mit einer Import-Anweisung, eingeleitet durch das Schlüsselwort `import`. Import-Anweisungen stehen immer am Anfang eines Java-Programms. So kann man auf einen Blick erkennen, welche externen Bibliotheken das Programm verwendet. Die folgende Anweisung importiert die Klasse SAXParser der Java-Version des XML-Parsers Xerces:

```
import org.apache.xerces.parsers.SAXParser;
```

In anderen Programmiersprachen gibt es vergleichbare Anweisungen. In Perl-Programmen muss man beispielsweise die Anweisung `use XML::Xerces` eingeben, um Xerces benutzen zu können.

Um Konflikte bei der Benennung von Klassen zu vermeiden, verwendet Java ein Namensraum-Konzept, das der Grundidee von Namensräumen in XML sehr ähnlich ist. Die Bildung eines vollständig qualifizierten Namens erfolgt auch hier unter Verwendung einer Zeichenkette, die üblicherweise einen auf den Urheber der Klasse eingetragenen Domain-Namen darstellt. Anders als in URIs werden die geordneten Komponenten des Domain-Namens allerdings in umgekehrter Richtung angegeben. So ist beispielsweise für das Apache-Projekt derzeit (Stand: April 2003) der Domain-Name `apache.org` eingetragen. Für seine Namenshierarchie wurden zwei weitere Komponenten `xerces` und `parsers` gewählt, die zur weiteren Aufgliederung des Namensraums dienen (und die wie auch die URI-Referenzen in XML-Namensräumen keine gültige Adresse im Internet repräsentieren müssen). Die vollständige Bezeichnung für die bereits erwähnte Klasse `SAXParser` lautet somit `org.apache.xerces.parsers.SAXParser`. Die umgekehrte Notation kann bei der Installation von Java-Klassen im lokalen Dateisystem zur Bildung von Verzeichnisnamen herangezogen werden, indem die einzelnen durch Punkt getrennten Komponenten auf Verzeichnisebenen gleichen Namens

abgebildet werden. So könnte sich die Implementierung der Klasse `SAXParser` in der Datei `org\apache\xerces\parsers\SAX\Parser.java` befinden, während sich die Implementierung einer Hilfsklasse `org.apache.xerces.readers.UTF8Reader` zum Einlesen von Dokumenten in UTF8-Kodierung in der Datei `org\apache\xerces\readers\UTF8Reader.java` befinden würde.

Ein besonderer Namensraum beginnt mit der Zeichenkette `java`. Er bezeichnet Bestandteile der Sprache, die in jeder Java-Umgebung bereits vorhanden sind und nicht explizit verfügbar gemacht werden müssen. Ein Beispiel hierfür ist die Klasse `java.lang.String`, die zur Repräsentation und Manipulation von Zeichenketten dient.

Implementierung von Klassen

Innerhalb des Körpers der Klasse werden schließlich die einzelnen Methoden der Klasse aufgeführt. Für jede Methode deklariert man den Wertebereich für ihren Rückgabewert (z.B. `int` für eine Methode, die ganzzahlige Werte liefert, oder `void` für Methoden, die keinen Rückgabewert liefern). Im Anschluss daran wird der Name der Methode festgelegt, gefolgt von der Vereinbarung der Parameterliste. Diese kann natürlich auch leer sein.

Hier ein Beispiel für den Kopf der Definition einer Methode, die keinen Rückgabewert liefert (`void`), den Namen `setDocumentLocator` trägt und einen Parameter mit Namen `loc` vom Typ `Locator` hat:

```
public void setDocumentLocator(Locator loc)
```

Der Methodendefinition vorangestellt ist der optionale Modifikator `public`. Hierdurch wird festgelegt, dass die Methode öffentlich ist und von anderen Klassen genutzt werden darf. Im Gegensatz dazu schränkt der Modifikator `private` die Nutzung der Methode auf die Klasse ein, in der die Methode selbst definiert ist.

In Java ist es möglich, sogenannte INTERFACE-KLASSEN zu definieren. Diese Klassen implementieren selbst keine Funktionen, sie geben lediglich eine Menge von Methodendefinitionen vor, die der Programmierer in seiner Anwendung umsetzen muss, wenn er ein Modul entwickelt, das dieser Schnittstellenbeschreibung entspricht. Damit wird sichergestellt, dass sich eine Klasse an die vorgegebenen Vereinbarungen hält, wenn sie das Interface implementiert, und wiederum von anderen Klassen verwendet werden kann. Genau dieser Ansatz kommt auch bei SAX zum Einsatz. SAX definiert die Namen, Parameter und Rückgabewerte einer Reihe von Methoden, die für die Verarbeitung XML-spezifischer Ereignisse vorgesehen sind. Der Anwender der Schnittstelle muss diese dann selbst programmieren. Um festzulegen, dass eine Klasse der Definition einer Interface-Klasse genügt, verwendet man das Schlüsselwort `implements`:

```
public class SimpleContentHandler implements
   ContentHandler
```

Hier wird die Klasse `SimpleContentHandler` definiert, die den Definitionen der Interface-Klasse `ContentHandler` entsprechen muss.

Soweit der kurze Überblick zu Java. Alle erwähnten Sprachmittel von Java werden in den folgenden Beispielen verwendet.

5.5 Simple API for XML (SAX)

Wie bereits oben beschrieben, handelt es sich bei SAX, dem SIMPLE API FOR XML, um einen Streaming-basierten Ansatz. Mit SAX geparste XML-Dokumente werden also schrittweise (inkrementell) verarbeitet und können so (nahezu) beliebig groß sein. Der SAX-Parser baut keine Repräsentation des XML-Dokuments im Hauptspeicher auf, sondern verarbeitet lediglich die jeweils lokalen Elemente. Dies bedeutet auch, dass ein SAX-Parser keine Kenntnis über vorangehende oder nachfolgende Elemente hat. Der Programmentwickler muss sich selbst um den Aufbau von internen Repräsentationen des geparsten XML-Dokuments kümmern.

SAX war ursprünglich als API für Java geplant und entwickelt worden. Das API ist das Ergebnis einer langen öffentlichen Diskussion auf der Mailingliste XML-DEV. Die aktuelle Versionsnummer des APIs ist SAX 2.0 (auch **SAX2** genannt). Wenn in der Folge dieser Lerneinheit von SAX geredet wird, so ist damit, falls nicht anders gekennzeichnet, immer SAX 2.0 gemeint. Die Ursprungsversion SAX 1.0 (auch **SAX1** genannt) wird von dem API weiterhin unterstützt. Allerdings sind die meisten Methoden der Version 1.0 als „deprecated" (d.h. abgelehnt, missbilligt) markiert, werden also nicht weiterentwickelt und sollten deshalb nicht mehr verwendet werden. Neben einigen Java-Implementierungen der SAX-API existiert inzwischen eine ganze Reihe von Implementierungen von SAX 1.0 oder SAX 2.0 in weiteren Programmiersprachen. Damit erreicht SAX einen sehr hohen Verbreitungsgrad und hat sich als De-facto-Standard etabliert. Neben vielen anderen implementieren auch die folgenden Parser SAX 2.0:

- Xerces-J: ist Teil des XML-Projekts von Apache, unterstützt auch JAXP.
- AElfred2: ist Teil des GNU-Projekts JAXP.
- Crimson: ist Teil des XML-Projekts von Apache, wird als Bestandteil des Java Development Kits (JDK 1.4) von Sun ausgeliefert, unterstützt auch JAXP.
- Oracles Java XML Parser: ist Bestandteil von Oracle-Produkten.
- XP 0.5 von James Clark: ein schneller XML-Parser, der jedoch nicht weiterentwickelt wird (SAX 2.0 wird durch Module anderer Entwickler unterstützt).

SAX folgt bei der Verarbeitung der XML-Daten einem EREIGNISORIENTIERTEN Ansatz. Im Verlauf des Parsens werden, in Abhängigkeit von Inhalt und Struktur des Eingabedokuments, Ereignisse (Events) generiert. Diese wiederum lösen den Aufruf von CALLBACK-FUNKTIONEN aus. Im Gegensatz zu Ansätzen, die wie z.B. DOM die Repräsentation des Quelldokuments als Baumstruktur erfordern, ist bei SAX die eigentliche Verarbeitung des XML-Dokuments also in das Parsen integriert.

Die Callback-Funktionen sind wiederum definiert als Teil von Interface-Klassen, den so genannten Event-Handlern. SAX definiert vier Interfaces, die für unterschiedliche Aspekte der Verarbeitung zuständig sind: die Behandlung des Dokumentinhalts (CONTENTHANDLER), die Reaktion auf Fehler (ERRORHANDLER), die Manipulation der DTD (DTDHANDLER) und die Auflösung von Entity-Referenzen (ENTITYRESOLVER). Mit Hilfe dieser vier Klassen kann SAX eine Vielzahl von unterschiedlichen Ereignissen verarbeiten.

Im Folgenden werden wir schrittweise ein einfaches Java-Programm entwickeln, das einen SAX-Parser zum Einlesen eines XML-Dokuments verwendet. Dabei werden wir insbesondere auf die ContentHandler-Schnittstelle eingehen, aber auch die Methoden der übrigen Interfaces erklären.

SAX und Xerces

Grundlage des hier vorgestellten Anwendungsprogramms ist der SAX-konforme XML-Parser XERCES2 aus dem Apache-Projekt. Über die angegebene Xerces-Homepage kann die neueste Version des Parsers in Form einer Archivdatei (als zip oder als gezipptes tar-Archiv) heruntergeladen werden. Die zip-Datei enthält als Java-Archive (jar-Dateien) zwei Bibliotheken (`xercesImpl.jar, xercesParserAPIs.jar`), die wir für unsere Beispielanwendung benötigen.

http://xml.apache.org/xerces2-j/

http://xml.apache.org/dist/xerces-j/

Diese zwei Bibliotheken müssen so installiert werden, dass der Java-Interpreter sie finden und laden kann. Folgende Möglichkeiten stehen dazu zur Verfügung:

- Anpassen der Umgebungsvariable `CLASSPATH`.
- Nutzung der Option `-classpath` beim Aufruf des Java-Interpreters `java` über die Kommandozeile.
- Kopieren der genannten Dateien mit der Endung „`.jar`“ in das Unterverzeichnis `%JAVA_HOME%\jre\lib\ext`.

Die zuletzt genannte Option installiert die Bibliotheken systemweit. %JAVA_HOME% steht hierbei für das Installationsverzeichnis der Java-Entwicklungsumgebung bzw. der Laufzeitumgebung für Java-Programme (z. B. `C:\JDK1.4.2`). Alle Java-Programme, die die in %JAVA_HOME% installierte virtuelle Maschine verwenden, können so nach der Installation auf den XML-Parser zugreifen.

Die Klassenbibliothek `xercesImpl.jar` enthält die eigentliche Implementierung des XML-Parsers, während in `xercesParserAPIs.jar` die implementierten APIs definiert worden sind. Der Paketname für das SAX-API lautet `org.xml.sax`. Wir werden im Folgenden die Klassen des SAX-APIs, die wir für unser Beispielprogramm benötigen, über die `import`-Anweisung in unser Programm einbinden.

Ein minimaler SAX-Parser

Die grundlegende Klasse, die wir aus dem SAX-API für unsere Beispielanwendung verwenden wollen, ist das Interface `org.xml.sax.XMLReader`. Diese Schnittstelle legt alle SAX2-konformen Methoden, Eigenschaften und Konfigurationsmöglichkeiten fest (siehe dazu auch den folgenden Abschnitt). Als abstrakte Interface-Klasse kann sie selbst nicht instanziiert werden, sondern muss durch eine konkrete Klasse implementiert werden. Bei Xerces nutzt man dazu die Klasse `SAXParser` (`org.apache.xerces.parsers.SAXParser`). Die entsprechende Zuweisung für die Instanziierung der Interface-Klasse lautet:

```
XMLReader parser = new SAXParser();
```

Mit dem Operator `new` wird hier ein konkretes OBJEKT der Klasse `SAXParser` angelegt und geeignet initialisiert. Anschließend wird es der Variablen `parser` zugewiesen, um später im Programm auf Eigenschaften und Methoden des Objekts zugreifen zu können. Die Deklaration der Variablen `parser` als `XMLReader` besagt, dass das zugewiesene Objekt alle Eigenschaften dieser Klasse aufweisen muss, d.h. es muss das entsprechende Interface implementieren.

Das Interface XMLReader definiert eine Methode parse (java.lang.String URI), mit der ein XML-Dokument geparst werden kann, das über den gegebenen URI bezogen wird. BEISPIEL 5.4 zeigt ein minimales Programm, das die Klasse SAXParser verwendet.

▶ Beispiel 5.4: Java-Beispielprogramm MinimalSaxParser

```
import java.io.IOException;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.apache.xerces.parsers.SAXParser;

public class MinimalSaxParser {

    public static void main (String[] argv) {
  try {

      // setup parser and parse document
      XMLReader parser = new SAXParser();
      parser.parse(argv[0]);

  } catch (IOException e) {
      System.err.println (e.getMessage());
  } catch (SAXException se) {
      System.err.println (se.getMessage());
  }
    }
}
```

Die Import-Anweisungen binden die externen Bibliotheken ein: IOException und SAXException werden gebraucht, um auf Fehler in der Verarbeitung reagieren zu können, XMLReader ist die Interface-Klasse, die die SAX-Methoden definiert, und SAXParser ist deren Umsetzung durch Xerces. Unsere im BEISPIEL 5.4 neu definierte Klasse trägt den Namen MinimalSaxParser, innerhalb derer eine einzige

Methode namens main definiert wird. Die Methode main ist per Konvention die Methode, die beim Starten eines Java-Programms aufgerufen wird. Ihr werden mit dem Schlüsselwort argv die beim Programmaufruf angegebenen Argumente als Parameter übergeben. Die beiden eckigen Klammern in der Parameterdeklaration der Methode zeigen an, dass es sich dabei um ein sogenanntes ARRAY handelt, ein Feld gleichartiger Objekte (hier sind es Zeichenketten, repräsentiert durch die Klasse java.lang.String). Der Zugriff auf einzelne Komponenten eines solchen Arrays erfolgt ebenfalls über die Klammernotation, wobei innerhalb der eckigen Klammern der Index des gewünschten Objekts in dem Feld angegeben werden muss. Die Zählung der Array-Komponenten beginnt bei 0, also ist argv[0] hier das erste beim Programmaufruf übergebene Argument.

Für das Parsen des Dokuments sind lediglich die folgenden zwei Zeilen verantwortlich:

```
XMLReader parser = new SAXParser();
parser.parse(argv[ 0] );
```

Zunächst wird der Parser erzeugt und initialisiert wie oben beschrieben. Anschließend wird mit der Methode parse das beim Programmaufruf namentlich übergebene XML-Dokument geparst.

Der so definierte minimale SAX-Parser muss in der Datei MinimalSaxParser.java gespeichert werden und kann anschließend mit der Anweisung javac MinimalSaxParser.java kompiliert werden. Das ausführbare Programm erwartet als Parameter den Dateinamen bzw. URI eines XML-Dokuments, das daraufhin vom SAX-Parser geparst wird. Der Programmaufruf lautet:

```
java MinimalSaxParser test.xml
```

Im Beispiel wird eine Datei test.xml verarbeitet. Ist das XML-Dokument fehlerfrei, kehrt das Programm ohne weitere Ausgabe zur Eingabeaufforderung zurück. Andernfalls wird eine entsprechende Fehlermeldung erzeugt und ausgegeben.

Im nächsten Schritt soll unsere minimale Beispielanwendung um Callback-Methoden erweitert werden. Ziel ist es, die in einem XML-Dokument vorkommenden Starttags auszugeben, wobei wir auf die Ausgabe von Attributen und Namensraum-Präfixen verzichten. Um die Ausgabe etwas anschaulicher zu gestalten, sollen die zu den Starttags zugehörigen Zeilennummern mit ausgegeben werden. Dazu werden wir eine Klasse `SimpleContentHandler` definieren, die den `ContentHandler` des SAX-APIs implementiert. Das Interface `org.xml.sax.ContentHandler` definiert die folgenden elf Callback-Methoden, die vom Parser beim Auftreten der entsprechenden Ereignisse aufgerufen werden (siehe TABELLE 5.1).

Callback-Methode	Erklärung
`setDocumentLocator`	Wird genau einmal zu Beginn des Parsens aufgerufen. Das dabei übergebene Locator-Interface stellt in der Folge der Verarbeitung Dateiinformationen zur Verfügung.
`startDocument`	Wird genau einmal zu Beginn des Parsens aufgerufen.
`endDocument`	Wird genau einmal am Ende des Parsens aufgerufen, und zwar unabhängig davon, ob das Parsen erfolgreich war oder nicht.
`processingInstruction`	Wird aufgerufen, wenn eine Processing-Instruction (PI) geparst wurde. Diese Methode wird nicht beim Parsen der XML-Deklaration aufgerufen, da diese keine zum Dokument gehörige PI darstellt.
`startPrefixMapping`	Wird aufgerufen, wenn eine Präfixbindung für einen Namensraum beginnt (d.h. wenn der Parser auf die Deklaration eines Namensraum-Präfixes trifft).
`endPrefixMapping`	Wird aufgerufen, wenn eine Präfixbindung für einen Namensraum endet (d.h. das Element mit der korrespondierenden Deklaration des entsprechenden Präfixes wurde geschlossen).
`startElement`	Wird aufgerufen, wenn ein Starttag geparst wurde.
`endElement`	Wird aufgerufen, wenn ein Endtag geparst wurde.
`characters`	Wird aufgerufen, wenn beim Parsen des Elementinhalts Zeichendaten (#PCDATA) angetroffen werden.
`ignorableWhitespace`	Wird aufgerufen, wenn nicht relevante Folgen von Whitespaces (Wagenrücklauf, Zeilenschaltung, Leerzeichen und Tabulatoren) geparst wurden.
`skippedEntity`	Wird aufgerufen, wenn eine Entity übersprungen wurde. Nichtvalidierende Parser sind gemäß XML 1.0 nicht dazu verpflichtet, Referenzen auf externe Entities aufzulösen und können diese überspringen.

Tabelle 5.1: Callback-Methoden von ContentHandler

Definition und Nutzung der Callback-Methoden

Die in BEISPIEL 5.5 gezeigte einfache Implementierung der Schnittstelle `ContentHandler` benutzt lediglich zwei der elf Callback-Methoden: `setDocumentLocator` und `startElement`. In `setDocumentLocator` wird das vom Parser erzeugte Locator-Objekt in der lokalen Variable `ourLocator` zwischengespeichert. Damit wird sichergestellt, dass der Parser im Verlauf des Parsens auf Dateiinformationen, wie etwa die aktuelle Zeilen-nummer oder den aktuellen Spaltenindex, zugreifen kann. Die Funktion `startElement` wird immer dann aufgerufen, wenn der Parser auf ein Starttag trifft. Die Methode besteht lediglich aus einer Ausgabeanweisung, die die aktuelle Zeilennummer gefolgt von dem in < und > geklammerten Elementnamen auf dem Standardausgabekanal ausgibt. Auch wenn wir in unserem Beispiel für die Ausgabe der Starttags nur diese beiden Methoden benötigen, müssen wir trotzdem sämtliche elf Methoden der ContentHandler-Schnittstelle in unserer Beispielklasse implementieren. Der Code-Block für die neun Methoden, die wir nicht verwenden, bleibt einfach leer.

▶ Beispiel 5.5: Java-Beispielprogramm SimpleContentHandler

```
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.Attributes;

public class SimpleContentHandler implements ContentHandler {
    private Locator ourLocator = null;


    // CALLBACKS
    public void setDocumentLocator(Locator locator) {
        ourLocator = locator;
    };
```

```
    public void startDocument() {};
    public void endDocument() {};
    public void processingInstruction(String target, String data) {};
    public void startPrefixMapping(String prefix, String uri) {};
    public void endPrefixMapping(String prefix) {};

    public void startElement(String namespaceURI, String localName,
          String qualifiedName, Attributes atts) {
          System.out.println (ourLocator.getLineNumber()
          + ":\t<" + localName +">");
    };

    public void endElement(String namespaceURI, String localName,
       String qualifiedName) {};

    public void characters(char[] text, int start, int length) {};

    public void ignorableWhitespace(char[] text, int start, int length) {};
    public void skippedEntity(String name) {};
}
```

Um diese Beispielklasse während des Parsens nutzen zu können und in Aktion zu sehen, modifizieren wir unser oben erstelltes Programm `MinimalSaxParser` dahingehend, dass wir zunächst eine Instanz der Klasse `SimpleContentHandler` erzeugen und bei dem SAX-Parser registrieren. Der Einfachheit halber kopieren wir dazu die Datei `MinimalSaxParser.java` unter dem Namen `ExtendedSaxParser.java` und ändern dann in der neu erzeugten Datei den Klassennamen entsprechend zu `ExtendedSaxParser`. Zur Registrierung eines Content-Handlers stellt das Interface `XMLReader` die Methode `setContentHandler` zur Verfügung. Wir brauchen unser Programm also nur um eine einzige Zeile zu erweitern:

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

Damit unser Programm leicht verständlich bleibt, fügen wir noch ein paar zusätzliche Kommentarzeilen ein. Das Ergebnis findet sich in BEISPIEL 5.6.

▶ Beispiel 5.6: Java-Beispielprogramm ExtendedSaxParser

```
import java.io.IOException;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.apache.xerces.parsers.SAXParser;

public class ExtendedSaxParser {

 public static void main (String[] argv) {
  try {

      // setup parser and parse document
      XMLReader parser = new SAXParser();
      // register SimpleContentHandler
      parser.setContentHandler(new SimpleContentHandler());
      // parse the file
      parser.parse(argv[0]);

  } catch (IOException e) {
       System.err.println (e.getMessage());
  }    catch (SAXException se) {
       System.err.println (se.getMessage());
  }
 }
}
```

Nun kann die Datei mit `javac ExtendedSaxParser.java` kompiliert werden. Der Programmaufruf erfolgt dann analog zum minimalen Parser mit

```
java ExtendedSaxParser test.xml
```

Die registrierte Instanz der Klasse `SimpleContentHandler` sorgt jetzt dafür, dass jeweils alle Starttags (ohne Attribute und Namensraum-Präfixe) mit der Zeilennummer ihres Auftretens im XML-Dokument auf dem Bildschirm ausgegeben werden. So wird es möglich, die schrittweise Verarbeitung des XML-Dokuments zu beobachten. Das Beispiel zeigt, wie einfach die Verarbeitung von XML-Dokumenten mit dem SAX-Modell ist.

Weitere Methoden zur Ereignisbehandlung

Wie bereits oben erwähnt, definiert SAX neben `ContentHandler` noch drei weitere Schnittstellen zur Behandlung von Ereignissen: die Interfaces `ErrorHandler`, `EntityResolver` und `DTDHandler`. Die folgenden Tabellen geben eine Kurzübersicht über die jeweiligen Callback-Methoden der einzelnen Klassen.

Für die Behandlung von Parse-Fehlern stellt das Interface `ErrorHandler` folgende Methoden zur Verfügung:

Tabelle 5.2: Callbacks des Interface ErrorHandler

Callback-Methode	Erklärung
`error(SAXParseException exception)`	Wird aufgerufen, wenn ein behebarer Fehler auftritt (recoverable error gemäß XML 1.0).
`fatalError(SAXParseException exception)`	Wird aufgerufen, wenn ein nicht-behebbarer Fehler auftritt (non-recoverable error gemäß XML 1.0).
`warning(SAXParseException exception)`	Wird aufgerufen, wenn eine Warnung auftritt (warning gemäß XML 1.0).

Die Registrierung eines ErrorHandler-Objekts erfolgt durch die Methode `setErrorHandler` der Klasse `XMLReader`.

Das Interface `EntityResolver` ist die Grundlage für die anwendungsspezifische Behandlung externer Entity-Referenzen. Die Klasse stellt lediglich eine Callback-Funktion zur Verfügung:

Tabelle 5.3: Callbacks von Entity Resolver

Callback-Methode	Erklärung
`resolveEntity(String publicid, String systemId)`	Wird aufgerufen, wenn eine Referenz auf eine externe Entity aufgelöst werden muss.

Ein SAX-konformer XML-Parser ruft diese Methode immer dann auf, wenn eine Referenz auf eine externe Entity aufgelöst werden soll. Als Ergebnis liefert die Methode ein Objekt der Klasse `InputSource`, eine Referenz auf die Datenquelle, aus der der Parser die Entität lesen kann. Die Klasse `InputSource` kapselt die möglichen unterschiedlichen Datenquellen von SAX in einem einzigen Objekt, so dass ein direkter Zugriff auf PUBLIC- und SYSTEM-Identifier, die Datenquelle als Byte-Stream (mit der gewählten Zeichenkodierung) und/oder als Character-Stream (ebenfalls mit der gewählten Zeichenkodierung) möglich ist. Instanzen der Klasse `InputSource` können in SAX alternativ zu URIs verwendet werden. Die Registrierung eines ENITITYRESOLVERS erfolgt durch die Methode `setEntityResolver` der Klasse `XMLReader`.

Das Interface `DTDHandler` erlaubt schließlich die Reaktion auf Ereignisse, die beim Parsen der DTD auftreten.

Tabelle 5.4: Callbacks des Interface DTDHandler

Callback-Methode	Erklärung
`notationDecl,`	Wird aufgerufen, wenn innerhalb der DTD eine Notation angetroffen wird.
`unparsedEntityDecl`	Wird aufgerufen, wenn innerhalb der DTD eine nicht-geparste Entity angetroffen wird.

Konfiguration des SAX-Parsers

SAX erlaubt eine Konfiguration der Arbeitsweise des XML-Parsers durch das Ein-/Ausschalten einfacher Merkmale (Features), indem so genannte FEATURE-FLAGS gesetzt werden. Der Status eines Features kann durch entsprechende Get/Set-Methoden abgefragt und verändert werden (getFeature, setFeature). BEISPIEL 5.7 zeigt, wie durch die Abfrage eines FEATURE-FLAGS festgestellt werden kann, ob der SAX-Parser validierend oder nichtvalidierend arbeitet.

▶ Beispiel 5.7: Lesen eines SAX-Features

```
try {
  String id ="http://xml.org/sax/features/validation";
  if (parser.getFeature(id)) {
    System.out.println("Parser validiert");
  } else {
    System.out.println("Parser validiert nicht");
  }
} catch (SAXNotRecognizedException e) {
  System.out.println("Unbekannte Eigenschaft");
} catch (SAXNotSupportedException e) {
  System.out.println("Wird derzeit nicht unterstützt");
}
```

Im diesem Beispiel wird durch die Methode `getFeature` der Wert des Feature-Flags `validation` abgefragt und in Abhängigkeit von dessen Wert eine Ausgabe erzeugt. Wie man an dem Beispiel erkennt, werden Eigenschaften des Parsers durch URIs spezifiziert (z.B. `http://xml.org/sax/features/validation`). Der Umfang der Eigenschaften und ihre Verwendung ist dabei abhängig von der eingesetzten SAX-Bibliothek. Alle Feature-Flags sind vom Datentyp `boolean`, können also die Wahrheitswerte `true` und `false` annehmen. Es wird allerdings in der Regel kein Vorgabewert definiert. Lediglich zwei Features werden durch den Standard vorgegeben und müssen von jedem SAX2-konformen Parser umgesetzt werden:

```
http://xml.org/sax/features/namespaces
http://xml.org/sax/features/namespace-prefixes
```

Diese Feature-Flags müssen (mindestens) auslesbar sein. Das Feature-Flag `namespaces` muss per Voreinstellung den Wert `true` liefern, das Feature-Flag `namespace-prefixes` muss den Wert `false` liefern. Hierdurch wird sichergestellt, dass der Parser eine minimale Unterstützung von Namespaces gewährleistet. Neben den beiden genannten Features werden noch elf weitere optionale Features für SAX2 definiert.

Ein weitere Möglichkeit zur Konfiguration des Parsers in SAX2 bieten PROPERTIES. Diese können genau wie Features durch entsprechende Methoden ausgelesen und gesetzt werden (`getProperty`, `setProperty`). Im Gegensatz zu den Features, die nur eingeschaltet oder ausgeschaltet werden können – das Flag ist gesetzt (Wahrheitswert „true") oder nicht gesetzt (Wahrheitswert „false") –, speichern Properties OBJEKTE, die das Parseverhalten im Einzelnen beeinflussen. Durch die Properties wird es beispielsweise möglich, Erweiterungen, bzw. Ergänzungen der in SAX 2.0 vordefinierten Methoden zur Behandlung von Ereignissen zu implementieren. Alle Bezeichner für Properties beginnen mit der Zeichenkette `http://xml.org/sax/properties/`. SAX2 definiert vier Properties (DECLARATION-HANDLER, DOM-NODE, LEXICAL-HANDLER und XML-STRING), die alle optional sind und daher nicht notwendigerweise von der eingesetzten Bibliothek unterstützt werden müssen.

Greift ein Anwendungsprogramm auf ein Feature oder eine Property zu, die nicht DEFINIERT ist, wird eine Exception des Typs `SAXNotRecognizedException` ausgelöst. Dieser Ausnahmetypus tritt z.B. auch dann auf, wenn durch Schreibfehler in dem verwendeten URI die gesuchte Eigenschaft nicht erkannt wird. Versucht ein Anwendungsprogramm auf ein Feature oder eine Property zuzugreifen, die zwar definiert ist, vom Parser aber nicht UNTERSTÜTZT wird, so wird eine Exception des Typs `SAXNotSupported Exception` ausgelöst. Dieser Fehler tritt beispielsweise dann auf, wenn wie in BEISPIEL 5.8 versucht wird, einen nicht validierenden Parser durch Setzen des Feature-Flags VALIDATION auf `true` zum Validieren zu bringen.

▶ Beispiel 5.8: Setzen eines SAX-Features

```
import java.io.IOException;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.apache.xerces.parsers.SAXParser;

public class MinimalSaxParser {

    public static void main (String[] argv) {
  try {

      // setup parser and parse document
      XMLReader parser = new SAXParser();
      parser.setFeature
      ("http://xml.org/sax/features/validation", true);
      parser.parse(argv[0]);

      } catch (IOException e) {
      System.err.println (e.getMessage());
      } catch (SAXException se) {
      System.err.println (se.getMessage());
      }
    }

}
```

5.6 Document Object Model (DOM)

www.w3.org/DOM/

http://xml.coverpages.org/dom.html

Das DOCUMENT OBJECT MODEL (DOM) wird vom W3C definiert. Im Gegensatz zu SAX ist DOM ein „echter" W3C-Standard (siehe auch nebenstehende Adressen).

DOM ist ursprünglich aus einer Erweiterung von DYNAMIC HTML (DHTML) hervorgegangen. Die ursprüngliche Idee war, ein allgemeines Format zur Repräsentation von HTML-Dokumenten in Browsern festzulegen sowie festzuschreiben, wie Browser mittels Java und JavaScript HTML-Dokumente verändern können. Jedoch hat sich DOM schnell von dieser speziellen Aufgabe zu einer allgemeinen Spezifikation von Sprachmitteln zum Zugriff auf Bestandteile und zur Manipulation von Bestandteilen von HTML- und XML-Dokumenten entwickelt. Mittlerweise ist DOM neben XML-Schema zu einem der komplexesten XML-Standards angewachsen.

Überblick

DOM ist in unterschiedliche funktionale Module aufgeteilt, deren Anforderungen in unterschiedlichen so genannten DOM-LEVELS definiert werden:

- LEVEL 1: definiert den DOM-Kern (DOM Core), sowie Spezialisierungen für HTML und XML (Verarbeitungsanweisungen, CDATA-Abschnitte und Entities). Level 1 beinhaltet Methoden zur Bearbeitung von Dokumentinhalten und zur Navigation innerhalb der Dokumente.
- LEVEL 2: ergänzt Level 1 um die Unterstützung von Namensräumen und CASCADING STYLE SHEETS (CSS) sowie um ein Ereignismodell, das die Reaktionen auf Benutzereingaben und Änderungen der Baumstruktur eines Dokuments behandelt, sowie die Unterstützung unterschiedlicher Darstellungsarten (Views) von Dokumenten.
- LEVEL 3: ergänzt Level 2 um Anpassungen an XML Infoset, XML Base und XPath, bringt u.a. weitere Möglichkeiten der Ereignisbehandlung, Methoden zum Speichern/Laden sowie zum Validieren von Dokumenten. Level 3 ist noch keine Empfehlung, sondern im Status eines WORKING DRAFT.

Da die Definition eines Standards durch das W3C ein sehr aufwändiger Prozess ist, gibt es eine umfangreiche Dokumentation zu den jeweiligen Teildefinitionen der unterschiedlichen DOM-Level. Diese im Detail darzustellen, würde den Rahmen dieser Lerneinheit sprengen. Entsprechend konzentrieren wir uns im weiteren Verlauf auf die Kernfunktionalität der DOM-Stufe 1 (DOM Level 1). In der Literatur findet man neben den hier aufgeführten Levels außerdem noch die Bezeichnung „DOM Level 0". Dabei handelt es sich um eine zusammenfassende inoffizielle Bezeichnung für die ersten Dokumentmodelle, die in verschiedenen Webbrowsern zur Realisierung von DHTML eingeführt wurden. Die Bezeichnung deutet die Vorläuferschaft dieser Entwicklungen an, hat aber darüber hinaus nichts mit den vom W3C veröffentlichten Standards zu tun.

Im Kern realisiert DOM eine Schnittstelle für Programmiersprachen, mit der XML-Dokumente auf eine Baumstruktur, deren Knoten als Objekte definiert sind, abgebildet und innerhalb dieser verarbeitet werden können. Mit den von DOM definierten Schnittstellen und Methoden erhalten in solchen Programmiersprachen entwickelte Anwendungsprogramme die Möglichkeit, XML-Dokumente zu lesen, zu erstellen, in ihnen zu navigieren, auf ihre Inhalte zuzugreifen und diese zu verändern. Um DOM anwenden zu können, ist eine Implementierung in einer konkreten Programmiersprache erforderlich. Solche Implementierungen (Bindungen von DOM an eine konkrete Programmiersprache) bezeichnet man auch als „Language Bindings". Sie sind mittlerweile in vielen Programmiersprachen (z.B. JavaScript, Perl, Python, aber auch Java, C, C++) realisiert. Das „Java Language Binding" wurde als normativer Appendix in die DOM-Spezifikation aufgenommen.

DOM repräsentiert ein XML-Dokument in einer hierarchisch geordneten Baumstruktur (dem **DOM-Tree**). Die Knoten des Baumes sind dabei Objekte, die wiederum Eigenschaften (**Properties**) und Funktionen (**Methoden**) besitzen. Der DOM-Parser verarbeitet ein XML-Dokument vollständig und überführt es in den DOM-Tree. Erst dann kann ein Anwendungsprogramm darauf zugreifen.

Alle Knoten eines DOM-Baums sind abgeleitet von der generischen Interface-Klasse `Node` (deutsch „Knoten"). In den abstrakten Methoden dieser Klasse werden alle wesentlichen Eigenschaften eines Knotens repräsentiert, z.B. der übergeordnete Knoten (`parentNode`), die Kind-

knoten (childNodes), der erste (firstChild) und letzte (lastChild) Kindknoten, der vorherige (previousSibling) und der nachfolgende (nextSibling) Geschwisterknoten. Auch der Knotentyp (nodeType) wird festgelegt. TABELLE 5.5 zeigt eine Übersicht über die verschiedenen Knotentypen im DOM.

Tabelle 5.5: DOM Knotentypen

Knotentyp	Interface-Klasse	Erklärung
DOCUMENT_NODE	Document	Repräsentiert die Wurzel des Baumes.
DOCUMENT_TYPE_NODE	DocumentType	Beschreibt den Dokumenttyp eines XML-Dokuments. Im Wesentlichen speichert dieser Knoten die Liste der für das Dokument definierten Entities (als NamedNodeMap). Über die NamedNodeMap kann man auf eine Menge von Knoten über deren Namen zugreifen.
ELEMENT_NODE	Element	Element repräsentiert ein XML-Element sowie dessen Attribute. Auf die Attribute kann in ihrer Gesamtheit als Menge oder jeweils einzeln über deren Namen zugegriffen werden.
ATTRIBUTE_NODE	Attr	Attr repräsentiert ein Attribut eines Elementknotens.
TEXT_NODE	Text	Text repräsentiert den textuellen Inhalt (#PCDATA) eines Elements.
CDATA_SECTION_NODE	CDATASection	CDATASection repräsentiert den in CDATA eingebetteten Text eines XML-Dokuments. CDATA-Abschnitte werden vom XML-Parser nicht geparst, sondern unverändert in den DOM-Tree übernommen.
COMMENT_NODE	Comment	Repräsentiert einen Kommentar.
PROCESSING_INSTRUCTION_NODE	Processing Instruction	Repräsentiert eine Processing Instruction. Ziel und Daten der Processing Instruction werden unverändert übernommen.
NOTATION_NODE	Notation	Notation repräsentiert die Deklaration einer Notation. Öffentlicher Bezeichner und Systembezeichner werden unverändert übernommen.
ENTITY_NODE	Entity	Entity repräsentiert eine Entity. Je nach Typ wird diese geparst oder unverändert übernommen.
ENTITY_REFERENCE_NODE	Entity Reference	EntityReference repräsentiert eine Referenz auf eine Entity.
DOCUMENT_FRAGMENT_NODE	Document Fragment	Ein DocumentFragment repräsentiert einen Ausschnitt aus einem DOM-Tree. Hierbei ist es nicht notwendig, dass der Ausschnitt wohlgeformt nach XML 1.0 ist.

Ein minimaler DOM-Parser mit Xerces

Ähnlich wie bei den Erläuterungen zu SAX soll auch in diesem Abschnitt eine einfache DOM-Anwendung schrittweise entwickelt werden. Wir verwenden wieder Xerces, den XML-Parser der Apache Software Foundation. Der Grundaufbau des Programms entspricht dabei dem SAX-Beispiel: Zuerst müssen die notwendigen Bibliotheken eingebunden werden, dann kann der Parser erzeugt werden, und schließlich wird mit der Methode `parse` das XML-Dokument verarbeitet. Das Dokument liegt dann im Speicher als DOM-Tree vor, der anschließend rekursiv durchlaufen werden kann.

- **Schritt1: Importieren der Bibliotheken**

Für die Nutzung von DOM müssen die entsprechenden Bibliotheken des JAVA LANGUAGE BINDING importiert werden. Diese sind im Paket `org.w3c.dom` definiert. Für die jeweiligen Knotentypen existieren entsprechend benannte Interface-Klassen. So findet man z.B. die Definition des DOM-Knoten unter `org.w3c.dom.Node`. Für die Anwendung werden die Klasse DOMParser aus der Xerces-Distribution, die benötigten Schnittstellen für die DOM-Knotentypen und alle zur Fehlerbehandlung notwendigen Exception-Klassen importiert.

```
// der DOMParser von Xerces
import org.apache.xerces.parsers.DOMParser;

// Java-Interfaces für die Knotentypen des DOM
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

// Fehlerbehandlung bei fehlerhaftem Parsing/
// Zugriff auf Datei
import org.xml.sax.SAXException;
import org.xml.sax.SAXNotRecognizedException;
import org.xml.sax.SAXNotSupportedException;
import java.io.IOException;
```

■ **Schritt 2: Erzeugung und Konfiguration des DOM-Parsers**

Nach den Import-Anweisungen folgt die Klassendefinition. In diesem Fall wird die Klasse `ValidatingDOM` definiert. Das Programm muss entsprechend in der Datei `ValidatingDOM.java` abgespeichert werden. Eine Instanz der Klasse `DOMParser` wird erzeugt und durch Setzen des entsprechenden Feature-Flags `validation` auf den Wert `true` in den validierenden Arbeitsmodus geschaltet. Wie man unten sieht, wird an dieser Stelle auf SAX-Ausnahmen reagiert. Wie lässt sich dies erklären? Wie bereits dargelegt, kann man mit SAX XML-Dokumente einfach und effizient parsen. Für die Entwickler des Xerces-DOM-Parsers lag es damit natürlich nahe, SAX für das Parsen der XML-Datei zu nutzen, wodurch sich der Aufwand für die Implementierung der Klasse `DOMParser` auf die Entwicklung von Routinen zum Aufbau des DOM-Trees reduziert. Der DOMParser basiert also auf dem SAXParser; entsprechend können auch SAX-Ausnahmen auftreten, und diese müssen in der DOM-Anwendung dann verarbeitet werden.

```
...

//  Eine validierende DOM Anwendung
public class ValidatingDOM {

    public ValidatingDOM (String xmlFile) {

//  Erzeugt den Xerces DOM Parser
      DOMParser parser = new DOMParser();

      // Aktiviert Validierung
      try {
      parser.setFeature
      ("http://xml.org/sax/features/validation", true);
      } catch (SAXNotRecognizedException e) {
      System.err.println (e);
      } catch (SAXNotSupportedException e) {
      System.err.println (e);
      }
...
    }
}
```

■ **Schritt 3: Parsen des Dokuments und Aufruf der Ausgaberoutine**
Im nächsten Schritt wird das angegebene XML-Dokument (Parameter `xmlFile`) mit der Methode `parse` geparst. Treten keine Fehler auf, liegt das Dokument anschließend als DOM-Tree im Speicher vor. Die Methode `getDocument` liefert den Wurzelknoten des Dokuments, und dieser wird in der Variablen `document` zwischengespeichert. Die Methode `traverse` werden wir im folgenden Schritt 4 des Beispiels programmieren.

```
...
  //  Parsen des Dokuments
  //  Durchlaufen des DOM-Trees mit traverse
  try {
      parser.parse(xmlFile);
      Document document = parser.getDocument();
      traverse (document);
  } catch (SAXException e) {
      System.err.println (e);
  } catch (IOException e) {
      System.err.println (e);
  }

...
```

■ **Schritt 4: Rekursive Ausgaberoutine**
Ausgehend vom Wurzelknoten des DOM-Trees besteht jetzt die Möglichkeit der Manipulation der internen Repräsentation. Im einfachsten Fall kann das Dokument ausgegeben werden. Es ist aber auch möglich, neue Knoten einzufügen, Knoten zu löschen, den Wert der Knoten zu verändern oder Knoten zu verschieben. Dazu ist es notwendig, den DOM-Tree schrittweise zu durchlaufen (zu TRAVERSIEREN), man „hangelt" sich gewissermaßen an der Struktur des Baumes entlang. Dies gelingt am einfachsten durch die Definition einer rekursiven Methode. Wie bereits in STREAMING UND TREE-BUILDING erläutert, sind XML-Bäume selbstähnlich, bestehen also selbst wieder aus Teilbäumen.

Die Traversierungsroutine `traverse` nutzt diese Eigenschaft aus, um die Starttags der Elemente des XML-Dokuments (ohne Attribute und Namensraum-Präfixe) auf dem Bildschirm auszugeben. Zuerst wird der Typ des aktuell betrachteten Knotens (node) überprüft. Handelt es sich um einen Elementknoten (`Node.ELEMENT_NODE`), so wird der Name des Knotens ausgegeben. Anschließend wird die Liste der möglichen Kindknoten ermittelt und im Objekt `children` vom Typ `NodeList` (eine geordnete Liste von Knoten) abgespeichert. Ist diese nicht leer (`children.getLength() > 0`), so wird für jeden einzelnen Kindknoten wiederum die Methode `traverse` REKURSIV aufgerufen und der Name des jeweiligen Elementknotens ausgegeben. So durchläuft die Methode schrittweise alle Elementknoten des DOM-Trees, wie bereits in STREAMING UND TREE-BUILDING anhand eines einfachen Beispiels in Pseudo-Code veranschaulicht wurde.

```
...

    // Durchlaufen des DOM-Trees.
    // gibt die Elementnamen aus

    private void traverse (Node node) {
       int type = node.getNodeType();
       if (type == Node.ELEMENT_NODE) {
           System.out.println
           ("<" + node.getNodeName()+">");
       }

       // Verarbeitet die Liste der Kindknoten durch
       // rekursiven Abstieg
       NodeList children = node.getChildNodes();
       for (int i=0; i< children.getLength(); i++)
           traverse (children.item(i));
    }

...
```

■ **Schritt 5: Start der Verarbeitung im Hauptprogramm**

Das Hauptprogramm besteht lediglich aus einer Zeile in der Funktion `main`, in der eine Instanz des validierenden DOM-Parsers erzeugt wird, um das angegebene XML-Dokument zu verarbeiten.

```
...

    // Main Method
    public static void main (String[] args) {
  ValidatingDOM validatingDOM = new ValidatingDOM
   (args[0]);
    }

...
```

■ **Schritt 6: Übersetzung und Aufruf des fertigen Programms**

Das Programm kann nun mit `javac ValidatingDOM.java` übersetzt und anschließend mit folgender Anweisung gestartet werden:

```
java ValidatingDOM test.xml
```

Der DOM-Parser verarbeitet nun die Datei `test.xml` und gibt die Namen der Elemente in Dokumentordnung auf dem Bildschirm aus.

ZUSAMMENFASSUNG

In dieser Lerneinheit haben wir uns mit praktischen Aspekten der Verarbeitung von XML-Dokumenten beschäftigt.

Ein wichtiger Prozess bei der Arbeit mit XML-Dokumenten ist das Parsen. Wir haben die für das Parse-Verfahren wichtigen Aspekte des Streaming und des Tree-Building kennen gelernt und nachvollzogen, was beim validierenden und beim nichtvalidierenden Parsen geschieht. Am Beispiel von nsgmls haben wir exemplarisch einen Parser angewendet und gesehen, welche typischen Fehlermeldungen hinsichtlich der generellen XML-Dokumentsyntax oder einer zu validierenden DTD auftreten.

Mit den Programmierschnittstellen SAX und DOM haben wir die beiden wichtigsten APIs für die Verarbeitung von XML-Dokumenten kennen gelernt.

SAX ist ein ereignisbasiertes API mit Methoden für die Behandlung des Dokumentinhalts, die Reaktion auf Fehler, DTD-Manipulationen und die Auflösung von Entity-Referenzen. Wir haben ein Programm in Java geschrieben, das ein XML-Dokument mit Hilfe eines minimalen SAX-Parsers auf Wohlgeformtheit überprüft und in der erweiterten Version Starttags ausgibt.

DOM ist ein vom W3C definiertes Modell für die Verarbeitung von Dokumenten. Wir haben uns hier auf das Kernmodul von DOM Level 1 beschränkt und am Beispiel des DOM-Parsers von Xerces unter Verwendung verschiedener Bibliotheken ein kleines Programm geschrieben, das die Elementnamen eines Dokuments sukzessive ausgibt.

In der folgenden Lerneinheit werden wir uns mit einer komplexen DTD beschäftigen und die Möglichkeiten der Dokumentstrukturierung, die mit der Modularisierung von DTDs und der Verwendung von Entities gegeben sind, im praktischen Umgang kennen lernen. Dabei lernen wir mit DocBook ein Dokumentenformat kennen, das sich in vielen Bereichen für die Publikation von Büchern und umfangreichen Texten verwenden lässt.