

Inhaltsverzeichnis

1. XML in der Praxis: Dokumente kodieren, editieren, parsen und validieren	??
Überblick	??
XML-Dokumente editieren mit dem Emacs	??
XML-Dokumente Bearbeiten mit dem Emacs	??
Parsen von XML Dokumenten: Streaming vs. Tree-Building	??
Validieren von XML Dokumenten	??
Was bedeutet Validierung ?	??
Validierendes Parsen von XML-Dokumenten	??
Nicht-Validierendes Parsen von XML-Dokumenten	??
Validierung von XML-Dokumenten in der Praxis	??
SAX und DOM	??
Was sind Programmierschnittstellen (APIs) ?	??
SAX als ereignisbasiertes API	??
Content Handler: die Verarbeitung von XML mit SAX	??
W3C DOM	??
Zeichensätze und Entities	??
Grundlegende Begriffe	??
Character Entities	??
Der UNICODE Standard	??
Ausblick	??

Kapitel 1. XML in der Praxis: Dokumente kodieren, editieren, parsen und validieren

LE 5.1 Einführung in die Benutzung des Emacs PSGML-Modes Es werden die Funktionen des PSGML-Modes und des TDTD-Modes erklärt, die im Verlauf des Kurses eine Rolle spielen werden. In einer Übung (5-1) wird ein XML-Dokument mit dem Emacs editiert; in der Übung 5-2 die entsprechende DTD.

Überblick

In dieser Lerneinheit befassen wir uns mit den praktischen Aspekten der Verwendung von XML-Dokumenten. Praxis bedeutet im Zusammenhang mit XML, aber auch generell im Bereich des Informationsmanagements, einerseits die Erstellung und andererseits die Verwendung der (Text-)daten.

Die wichtigste Form der Datenerstellung bildet die manuelle Erfassung der Texte, wobei bereits beim Schreiben der Texte die XML-Information erfasst wird. Diesen Prozess werden wir detailliert anhand des Editors Emacs betrachten.

Überblick Parsen etc.

XML-Dokumente editieren mit dem Emacs

Einführung in die Benutzung des Emacs

Der Editor Emacs ist ein außerordentlich mächtiger, programmierbarer Editor. Er wurde 1975 von Richard Stallman entwickelt und wird seitdem von einer Vielzahl von Programmierern stetig aktualisiert, erweitert und verbessert. Der Editor ist als "freie Software" lizenziert, was im Wesentlichen bedeutet, dass jeder den Editor verwenden, weiterentwickeln und gratis oder gegen Entgelt weitergeben darf.

Der Emacs stammt aus der Unix-Welt, d. h. er steht für die verschiedensten Varianten dieses Betriebssystems, z. B. auch Linux, zur Verfügung und gehört meist zum Standardinstallationsumfang. Für Rechner mit Microsoft Betriebssystemen und Apple-Rechnern existiert jedoch ebenfalls eine Vielzahl von Programmdistributionen. Wir werden uns bei der Betrachtung der Installation insbesondere auf die Windows-Versionen konzentrieren, für die Benutzung des Editors ist das verwendete Betriebssystem ohnehin irrelevant.

Installation

Ein Prinzip der freien Software ist, dass zu den Programmen auch der vollständige Quellcode mitgeliefert wird. Daneben werden aber meist auch bereits einfach installierbare Versionen für verschiedene Betriebssystemen angeboten. Diese Binärdateien werden zusammen mit weiteren benötigten Dateien in einer komprimierten und gepackten Datei zusammengestellt.

Zum Lieferumfang des Emacs-Editors gehören viele Hundert Dateien, die in einem eigenen Verzeichnis entpackt werden sollen. Ist dies erfolgt, gibt es unter Windows die Möglichkeit mit dem mitgelieferten Programm `addpm.exe` eine Programmgruppe zu erstellen und den Editor über das Start-Menü aufzurufen. Alternativ hierzu kann Emacs auch durch Aufruf des Programms `runemacs.exe` ausgeführt werden.

Texte erstellen und Bearbeiten

Nach dem Start des Editors werden allgemeine Informationen angezeigt. Alle Informationen, aber auch die Menüeinträge, die Hilfen etc. sind auf englisch verfasst. Bedauerlicherweise ist es anfangs relativ aufwändig die Arbeit mit dem Emacs zu erlernen. Da dies selbst für die einfachsten Aufgaben gilt, betrachten wir jetzt das öffnen, verändern und speichern von Dateien. Anhand dieser elementaren Aufgaben lernen wir die Konventionen der Interaktion mit dem Emacs und ihrer Beschreibung kennen.

Die wichtigsten Kommandos zur Bearbeitung von Textdateien sind über die Menüpunkte File und Edit zu erreichen. Zusätzlich hierzu sind die Befehle, wie alle Kommandos, über die Tastatur aufrufbar, allerdings werden andere Tastaturkürzel verwendet als die, die den Windows-Benutzern vertraut sind. Zusätzlich hierzu können alle Befehle über eine spezielle Kommandozeile (Mini-Buffer) eingegeben werden.

Um die Interaktion mit dem Emacs zu erlernen empfiehlt es sich eine Textdatei anzulegen, zu schreiben, zu speichern und den Editor zu verlassen. Diese Schritte werden wir uns, insbesondere um die Notation der Befehlseingaben kennen zu lernen, nachfolgend in den drei Alternativen ansehen:

1. Menü: Nach dem Aufruf des Emacs öffnen wir eine Datei, z. B. `text.txt`. Die geschieht durch die Auswahl des Menüpunktes **File** → **Open File**. Existiert eine derartige Datei nicht, wird sie neu angelegt. Wir erstellen einen Beispieltext und speichern diesen durch die Auswahl des Menüpunktes **File** → **save** (current Buffer). Zum Verlassen des Editors wählen wir **File** → **Exit Emacs**.
2. Tastaturkürzel: Mit dem gleichzeitigen Drücken der Steuerungstaste (Auf deutschen Tastaturen **Strg** auf englischen Tastaturen **Ctrl** CONTROL) und einer weiteren Taste werden die meisten Tastaturkürzel eingeleitet. Das Pressen der Steuerungstaste wird in den Beschreibungen des Emacs mit C- gekennzeichnet. Nach dem Bindestrich folgt dann die gleichzeitig zu drückende Taste. **C-x** bedeutet z. B., **Strg** zu drücken, das x dazu. Das Erstellen einer Datei wird mit **C-x C-f** veranlasst, das Speichern der Datei mit **C-x C-s** und das verlassen des Editors durch **C-x C-z**.
3. Auf das gleichzeitige Drücken der Taste **Alt** und der Taste x wird in den Emacs-Anleitungen mit **M-x** Bezug genommen. **M-x** leitet die Befehlseingabe über den sogenannten Mini-Buffer ein. Unsere Aufgabenstellung wird auf diesem Weg über folgende Schritte ausgeführt: **M-x find-file**, **M-x save-buffer** **M-x save-buffers-kill-emacs**.

Bei der Betrachtung dieser einfachen Interaktionen zeigt sich bereits, dass die Arbeit mit dem Emacs eine gewisse Einarbeitungszeit verlangt. Würde diese genutzt steht - kostenlos - ein äußerst mächtiges Softwareprodukt zur Verfügung.

XML-Dokumente Bearbeiten mit dem Emacs

Der Emacs stellt ein große Menge von Bearbeitungsmodi für verschiedene Dokumenttypen zur Verfügung. Es gibt Modi für Textdokumente, Für die Erstellung von XML-Dokumenten existieren zwei interessante Modi: `tdtd` und `psgml`. Der Modus `tdtd` unterstützt den Autor bei der für die Erstellung, Bearbeitung und Betrachtung

von Dokumenttypdefinitionen; der Modus psgml erlaubt die DTD-geleitete Editierung von XML-Instanzen.

Beide Modi gehören derzeit nicht zum Standarddistribution des Emacs, müssen also auch installiert werden, wenn bereits eine vollständige Emacs-Installation durchgeführt wurde. Hierzu werden die Pakete ttdt und psgml benötigt, die frei über das WWW verteilt werden. Die gepackten Distributionen werden in dem Verzeichnis site-lisp entpackt und bei Bedarf kompiliert. In einem weiteren Schritt müssen in der Datei default.el oder der Datei .emacs benutzerspezifische Einstellungen vorgenommen werden. Der Installationsprozess und die zu tätigen Einstellungen sind relativ komplex, so dass wir diese nicht genauer betrachten wollen. In dem Kursmaterial wird eine derartige XML-Editierumgebung vorkonfiguriert bereitgestellt.

Der Modus ttdt

Tabelle 1-1. ttdt-Befehle

Befehlsname	Emacs-Tastaturkürzel	Beschreibung
Elementdefinition	C-c C-e	Fügt eine Elementdefinition an der Cursorposition ein; Der Benutzer muss den Namen des Elements, eine Umschreibung, einen Kommentar und das Inhaltsmodell bestimmen.
Attributsdefinition	C-c C-a	Fügt eine Attributsdefinition an der Cursorposition ein. Der Benutzer kann nacheinander den dazugehörigen Elementnamen, den Attributnamen, einen Kommentar, den Datentyp und den Status des Attributs eingeben. Weitere Attribute folgen mit C-c C-a.
Entitätsdefinition	C-c C-%	Fügt eine Entitätsdefinition an der Cursorposition ein. Zunächst wird der Name bestimmt, dann eine Umschreibung, ein Kommentar und schließlich der Entitätswert.

Der Modus psgml

Der XML-Modus psgml ermöglicht die Erstellung der XML-Datei unter Beachtung der durch die DTD gegebenen Strukturvorgaben. Bei einer entsprechenden Installation des Modus werden zusätzlich zu dieser Hilfe auch die XML-Auszeichnungen (die Elemente, Kommentare etc.) farbig hervorgehoben.

Der Modus psgml stellt eine Vielzahl von Kommandos bereit, von denen (wie generell beim Emacs oder bei anderen komplexen Software-Produkten) meist nur wenige benötigt werden. Die nachfolgende Tabelle stellt die wichtigsten Kommandos zusammen.

Tabelle 1-2. PSGML-Befehle

Befehlsname	Emacs-Tastaturkürzel	Beschreibung
sgml-insert-element	C-c C-e	Fügt ein Element an der Cursorposition ein; Je nach Modus wird eine Selection-Box mit erlaubten Elementnamen oder ein Menu-Dialog mit inkrementeller Namensvervollständigung (TAB -Taste) geöffnet
sgml-tag-region	C-c C-r	Fügt ein Element an der markierten Stelle ein. Die betroffene Umgebung beginnt vor dem markierten Text und endet hinter der Markierung.
sgml-insert-end-tag	C-c /	Beendet das aktuelle Element, d. h. das an der aktuellen Stellen passen-den End-Element wird eingefügt.
sgml-split-element	C-c Return	Beendet das aktuelle Element, d. h. das an der aktuellen Stellen passen-den End-Element wird eingefügt.
sgml-validate	C-c C-v	Externe Validierung, z.B. durch onsgmls; Viele Applikationen liefern emacs-compatible Fehlermeldungen, die den direkten Sprung zur Fehlerstelle ermöglichen.

Befehlsname	Emacs-Tastaturkürzel	Beschreibung
sgml-insert-attribute	C-c +	Befindet sich der Cursor über einen Start-Element können die zulässigen Attribute des Elementes eingegeben werden. Dieser Vorgang ist vergleichbar mit dem der Einfügung von Elementen.

Neben dieser Editierfunktionalität bietet der Modus psgml noch eine weitere interessante Möglichkeit: Durch den Aufruf des Menüpunktes **DTD**→**info** werden verschiedene Dokumentationen der aktuellen DTD generiert, z.B. eine Beschreibung der zur Verfügung stehenden Elemente, ihrer Inhaltsmodelle, ihrer Attribute und den Kontexten in denen sie verwendet werden dürfen.

Parsen von XML Dokumenten: Streaming vs. Tree-Building

Die allgemeine Zielsetzung des *Parsens* ist der Aufbau einer Repräsentation von *Struktur* und *Inhalt* eines XML-Dokuments. Der Parser überführt dabei die *linear* angeordnete XML-Datei in eine Baumstruktur. Die Baumstruktur ist beschreibbar durch eine *kontextfreie* Grammatik. Je nachdem, ob das XML-Dokument validiert werden soll, oder nicht, arbeiten die gängigen XML-Parser in unterschiedlichen Modi.

Man kann zwischen zwei grundsätzliche Parseverfahren unterscheiden: *Streaming* und *Tree-Building*. Für beide Ansätze existieren Implementationen in Form von standardisierten Programmierschnittstellen: *SAX* (*Simple API for XML*) und *DOM* (*Document Object Model*). Beim Streamingverfahren wird während des Parsens des Dokuments immer nur eine partielle Repräsentation des XML-Dokuments aufgebaut. Hierdurch wird es möglich nahezu beliebig große XML-Dokumente zu verarbeiten. Streaming erweist sich besonders dann als nützlich, wenn nur lokale Berechnungen (Transformationen) auf der Baumstruktur durchgeführt werden müssen, etwa beim direkten Umsetzen eines XML-Dokuments in HTML.

Im Gegensatz hierzu wird im Tree-Building-Modus der vollständige Strukturbaum im Hauptspeicher des Rechners aufgebaut. Dieses Vorgehen schränkt die Größe der verarbeitbaren Dokumente ein, hat jedoch den Vorteil, dass unmittelbar auf alle Teilstrukturen des XML-Dokuments zugegriffen werden kann. Dies ist beispielsweise für Sortierungen nützlich.

Validieren von XML Dokumenten

Was bedeutet Validierung ?

Von der *Validierung* eines XML-Dokumentes spricht man, wenn die Struktur einer XML-Instanz gegen eine formale Struktur-Definition getestet wird. Die formale Struktur-Definition liegt meist in Form in Form einer DTD

(Document Type Definition) vor [siehe LE 4], möglich wären aber auch andere Strukturbeschreibungen, z.B. ein XML-Schema [siehe LE 11]. Neben der Überprüfung der strukturellen Anordnung der Elemente, wird Name und Typ der Attribute überprüft. Je nach Definition innerhalb der DTD können Attribute obligatorisch oder fakultativ sein. Ihre 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 Aufzählungstypen als Entity). In der XML-Instanz, nicht jedoch in der DTD, wird festgelegt, welches der Elemente das Wurzel-Element der XML-Instanz sein 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 *validen* XML-Dokument.

Validierendes Parsen von XML-Dokumenten

Beim validierenden Parsen von XML-Dokumenten ergibt sich das Problem, dass in einem ersten Schritt die DTD geparkt und intern repräsentiert werden muss. Dieser Schritt ist insofern problematisch, als die DTD in einer eigenen, nicht XML-konformen Syntax beschrieben wird. Ein validierender XML-Parser besteht also im Prinzip aus zwei Parsern: einem für die DTD und einem für die XML-Instanz. In nächsten Schritt muss dann das XML-Dokument in seine syntaktischen Bestandteile zerlegt werden (die *lexikalische Analyse*). Es müssen Tags zerlegt und Elementnamen identifiziert werden, Attributnamen und Attributwerte erkannt werden, Entitäten und Namespaces aufgelöst und verarbeitet werden. Die lexikalische Analyse erfolgt in der Regel schrittweise (*inkrementell*). Bereits erkannte Teile werden direkt weitergeleitet an den dritten Schritt des Parsens: 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 vorlag, oder nicht. Ein validierender Parser wird also durch die XML-Instanz (Festlegung des Wurzel-Elements) und die DTD gesteuert.

Nicht-Validierendes Parsen von XML-Dokumenten

Jede valide XML-Instanz ist immer auch eine *wohlgeformte XML-Instanz* (*wellformed XML*), allerdings nicht zwangsweise umgekehrt. Der Parsevorgang für wohlgeformte XML-Dokumente basiert auf der Basissyntax von XML: für jedes XML-Dokument existiert genau ein Wurzelement, sich überlappende Tags sind nicht zugelassen und jedes geöffnete Tag muss in der Folge auch wieder geschlossen werden. Ein Element-Tag 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 traditionellen SGML-Ansatz, der zwingend eine DTD verlangt. Die interne Syntax der Tags ist ebenfalls genau festgelegt. Jedes Tag wird mit < und > geklammert, der Elementname muss ein gültiger Bezeichner (siehe LE 2) sein und ein schließendes Tag stellt dem Elementnamen ein oder voran.

Ein nicht-validierender XML-Parser muss also nur die Wohlgeformtheit eines XML-Dokuments überprüfen. Entsprechend muss für den Parsevorgang keine DTD angegeben werden. Dadurch vereinfacht sich das Parsen erheblich. Die lexikalische Ana-

lyse entspricht dem oben beschriebenen Vorgang, die syntaktische Analyse reduziert sich jedoch auf die Überprüfung der Basissyntax. Dies kann durch die Nutzung eines einfachen *rekursiven* Stack-gesteuerten Parser erfolgen.

Anmerkung: Der Begriff Rekursion bezeichnet in der Mathematik, Informatik und Linguistik Verarbeitungsverfahren, die ein Problem schrittweise solange in kleinere gleich strukturierte Teilprobleme zerlegen, bis eine einfache Lösung für das entstandene Teilproblem gefunden ist. Die Gesamtlösung wird anschließend aus der Summe der Teillösungen errechnet. Für die Verarbeitung des Problems wird eine rekursive Vorschrift (lat. *programm* !!) definiert. Bezogen auf das Parsen von XML-Dokumenten bedeutet dies: alle wohlgeformten XML-Dokumente stellen Baumstrukturen dar. Auf der obersten Ebene existiert ein Wurzelement. In dieses eingebettet ist eine Menge von Tochterelementen. Betrachtet man diese Teile einer XML-Datei separat, erkennt man, dass auch diese Teilstrukturen wieder Bäume sind. Dies gilt ebenfalls für deren Tochterelemente und für deren Tochterelemente und so weiter. Erst wenn man auf der Ebene der Textdaten angelangt ist, existieren keine weiteren Tochterelemente mehr. Ein rekursiver Parser macht sich diese Form der Selbstähnlichkeit zunutze. Beginnend mit dem Wurzelement (einem Baum) werden erst alle Teilbäume geparkt, 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 auch Beispielprogramm bei DOM]

Findet der Parser ein öffnendes Tag, so speichert er den Elementnamen auf dem Stack, wird ein schließendes Tag gefunden, so wird dieser mit dem obersten Element des Stacks verglichen. Stimmen die Namen überein, wird das Element vom Stack entfernt und der Parsevorgang 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 Parsevorgangs leer und die Datei ist beendet, so ist liegt ein wohlgeformtes XML-Dokument vor. Ein nicht-validierender Parser wird also nur durch die XML-Instanz gesteuert.

Validierung von XML-Dokumenten in der Praxis

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

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 System ist die Komplexität ihrer Bedienung. Ein preisgünstige und sehr leistungsfähige Alternative stellt die Verwendung des Emacs in Kombination mit psgml und (o)nsxmls dar. Das psgml Modul ist in der Lage komplexe DTDs zu parsen und den Emacs so zu konfigurieren, dass er den Benutzer bei

der Eingabe des XML-Dokuments leitet. Der externe onsgmls-Parser ermöglicht die anschließende Validierung des XML-Dokuments [siehe hierzu auch LE 2].

Eine weitere vergleichsweise einfache Möglichkeit eine XML-Datei zu überprüfen bieten einige WebBrowser (z.B. Mozilla, Konquerer, aber auch der InternetExplorer). Auch in diese Systemen sind die XML-Parser integriert und werden aktiviert, sobald eine XML-Datei geladen wird. Leider sind die eingebauten Parser nicht allen Fällen konform mit XML 1.0. So reagiert z.B. der InternetExplorer bei die 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 InternetExplorer als valides XML-Dokument akzeptiert.

Neben den integrierten Systemen existieren eine Reihe von "Standalone"-Parsern, also Parsern, die unabhängig von einer sie einbettenden Anwendung aufgerufen werden können. Beispiele hierfür sind onsgmls oder msxml, der Microsoft XML-Parser, der auch im InternetExplorer verwendet. Für den Benutzer bedeutet dies, das die gewohnte Maus-basierte 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

Im Folgenden werden wir den nsgmls-Parser verwenden. Dieser SGML-Parser basiert auf den Arbeiten von James Clark. Der Parser wurde über die Jahre schrittweise verbessert und ist inzwischen Teil des freien OpenJade-Pakets. Die Parser trägt hier den Namen onsgmls. Nsgmls ist vergleichsweise schnell und bietet derzeit die beste Unterstützung von XML 1.0. Der Parser wird durch folgende Befehlsfolge gestartet:

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

In diesem wird die Datei **yourfile.xml** geparkt. Neben der Eingabe-Datei wird noch Datei **xml.dcl**, die XML-Deklaration, an den Parser übergeben. Dies ermöglicht dem SGML-Parser die XML-Basissyntax zu verarbeiten. Durch die Kommandozeilenschalter **-s** und **-w** wird der Parser konfiguriert: **-s (silent)** unterdrückt die Ausgabe des ESIS-Ergebnisbaums, **-wxml (warning)** schaltet den Parser in den XML-Warnungsmodus.

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

```
nsgmls.exe:xml.dcl:1:W: SGML declaration was not implied  
nsgmls.exe:tasx.dtd:20:25:W: #PCDATA in seq group
```

Man erkennt, dass die Meldungen des Parsers den Namen der Datei angeben, gefolgt von Zeile und 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 eine kurzen (englischen) Beschreibung der Warnung. Die

zweite Meldung des Parsers bedeutet also, dass für der Datei **tax.dtd** in Zeile 20, Spalte 25 eine Warnung vom Type **#PCDATA in seq group** ausgegeben werden muss. In der Tat wird in der DTD an dieser Stelle mixed content definiert und der Parser informiert den Benutzer über diesen Umstand. Ein Warnmeldung des Parsers führt also nicht zum Abbruch der Verarbeitung, sondern informiert den Anwender über mögliche Probleme innerhalb der geparsten Dateien.

Beispiel 1-1. Fehlerhaftes End-Tag

Welche Ergebnisse liefert nsgmls wenn echte Fehler in der XML-Datei existieren ? Im ersten Schritt verstoßen wir gegen die Basissyntax von XML. In der XML-Datei wird der Name eines schließenden Tags verändert (aus **</session>** wird **</ession>**), so daß die notwendige geschachtelte Klammerung fehlerhaft ist. Neben den schon beschriebenen Warnungen liefert nsgmls drei Meldungen:

```
nsgmls.exe:t2.xml:123:8:E: end tag for element "ession" which is not open
nsgmls.exe:t2.xml:124:6:E: end tag for "session" omitted, but OMITTAG NO was specified
nsgmls.exe:t2.xml:4:0: 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 123, Spalte 8 ein schließendes Element **ession** identifiziert wurde, für das kein entsprechendes öffnendes Element existiert. Die zweite Meldung zeigt an, dass in Zeile 124, Spalte 6, ein schließendes Element für das geöffnete Element **session** nicht gefunden wurde. Dies ist nicht erlaubt ist, da die geladene XML-Deklaration die Auslassung von Tags verbietet (**OMMITTAG NO**). Die dritte Meldung ist schließlich weder eine Warnung, noch eine Fehlermeldung! Hier wird die Position (Zeile 4, Spalte 0) des öffnenden Elements ausgegeben, für das das schließende Element fehlt. In der Datei findet man an dieser Position das Element **<session>**. 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 gelieferte Fehlermeldung die genaue Position des Fehlers an.

Es ist wichtig zu verstehen, dass der Parser schon bei nur einem einzigen Fehler in der XML-Datei eine Menge von Fehlermeldungen ausgeben kann. Ein Fehler kann die Gesamtstruktur einer XML-Datei zerstören. Tritt der Fehler innerhalb der ersten Zeilen einer grossen XML-Datei auf, können potentiell eine große Anzahl von sogenannten *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.

Beispiel 1-2. undefiniertes Attribut

Im zweiten Versuch verstoßen wir gegen die Vorgaben der DTD, indem wir ein obligatorisches Attribut auslassen. Der Parser liefert das folgende Ergebnis:

```
nsgmls.exe:t2.xml:121:38:E: required attribute "e-id" not specified
```

Die Meldung zeigt klar an, dass in Zeile 121, Spalte 8 ein Attribut fehlt. Der Parser gibt zusätzlich an, welches Attribut fehlt, in diesem Fall **e-id**. Fügt man an der gleichen Stelle ein nicht definiertes Attribut ein, meldet der Parser:

```
nsgmls.exe:t2.xml:121:28:E: there is no attribute "zuviel"
```

Auch hier ist Fehlermeldung sehr klar. In der DTD ist kein Attribut mit dem Namen zuviel definiert, also darf es auch nicht in Zeile 121, Spalte 28 verwendet werden.

Beispiel 1-3. Element-Tag nicht mit > abgeschlossen

Im dritten Versuch erzeugen wir fehlerhaftes XML indem wir ein Tag nicht ordnungsgemäß mit > schließen. In die gleiche Fehlerklasse fällt die Auslassung eines " am Ende eines Attributwertes. Fehler dieser Art können zu einer Menge von Folgefehlern führen. In diesem Fall werden folgende Fehlermeldungen geliefert:

```
nsgmls.exe:t2.xml:121:46:E: "1" is not a member of a group specified for any attribute  
nsgmls.exe:t2.xml:121:47:E: unclosed start-tag requires SHORTTAG YES
```

Die zweite Meldung zeigt uns die eigentliche Ursache und den genauen Fundort des Fehlers (Zeile 121, Spalte 47) an. Die erste Meldung ist komplizierter: es wird moniert, dass der Wert "1" kein Attribut des Elementes ist. Da der Parser das schliessende > des Elements nicht findet, muss er annehmen, dass der Inhalt des Elements (in diesem Fall die Zahl 1) ein Attribut bezeichnet. Dies ist nicht der Fall und daraus resultiert diese Fehlermeldung.

Beispiel 1-4. DTD nicht gefunden

Als letztes Beispiel für einen gängigen Fehler verändern wir die DOCTYPE-Deklaration zu Beginn des XML-Dokuments. Wir ändern den Namen der DTD von **taxs-dtd** nach **task.dtd**. Diese Datei existiert nicht. Der Parser muss also versuchen das Dokument gegen eine nicht existierende DTD zu validieren. Nach Aufruf des von nsgmls werden für die Beispieldatei 126 (!) Fehlermeldungen ausgegeben, z.B.:

```
nsgmls.exe:t2.xml:3:5:E: element "taxs" undefined  
nsgmls.exe:t2.xml:4:14:E: there is no attribute "s-id"  
nsgmls.exe:t2.xml:4:16:E: element "session" 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. Nsgmls bricht die Ausgabe bei mehr als 200 Fehlermeldungen ab.

Beispiel 1-5. Prüfen der Wohlgeformtheit

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

```
nsgmls.exe:t1.xml:2:0:E: no document type declaration; will parse
without validation
```

Das Dokument wird trotzdem vollständig geparkt und dabei auf seine Wohlgeformtheit hin überprüft. Treten keine weiteren Fehlermeldungen mehr auf, ist das Dokument wohlgeformt.

SAX und DOM

Im folgenden Abschnitt der LE stellen wir die Programmierschnittstellen SAX und DOM vor. Zunächst wird der Begriff Programmierschnittstelle (API) genauer erklärt. Danach wird die Verwendung von SAX und DOM in der Programmiersprache Java anhand einiger leicht nachzuvollziehender Beispielprogramme dargestellt.

Was sind Programmierschnittstellen (APIs) ?

Die Benutzung von XML-Parsern ist in der Regel an die Verwendung einer spezifischen Programmiersprache (z.B. Java, Perl, PHP etc.) und den hierfür verfügbaren Parse-Bibliotheken gebunden. Um den Einsatz der unterschiedlichen Parse-Bibliotheken zu vereinfachen wird dem Entwickler eine dokumentierte *Programmierschnittstelle* (API = *Application Programming Interface*) zur Verfügung gestellt. Eine Programmierschnittstelle besteht aus einer Reihe von öffentlichen Methoden, Variablen und Konstanten. Für den Programmierer abstrahiert die Schnittstelle die internen Abläufe der Programmbibliothek, die eigentliche Implementation der Funktionen ist *transparent*. Die Konzeption einer Schnittstelle erlaubt es so, alternative Bibliotheken in den entwickelten Programmen zu nutzen, ohne den Programmcode der Anwendung 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 Hinblick auf deren tatsächlich durchgeführte Funktion (*Semantik* des APIs).

Für die Verarbeitung von XML Dokumenten existieren eine Reihe konkurrierender APIs, wobei die derzeit umfassendste Infrastruktur für die Programmiersprache Java verfügbar ist. Aktuell verwendete APIs sind unter anderem:

- SAX, Simple API for XML: SAX1 und SAX2
- DOM, Document Object Model: DOM Level 0, DOM Level 1 und DOM Level 2
- JDOM: OpenSource API, vergleichbar mit DOM, jedoch optimiert für die Verwendung mit Java.

- JAXP: Java API for XML Processing von SUN, abstraktes Interface, das SAX, DOM und XSL-T kombiniert. Die jeweiligen APIs sind als PlugIns durch JAXP ladbar.
- Spezifische Parser-APIs : proprietäre Schnittstellen, die von den Entwicklern der jeweiligen Parser definiert werden.

Im weiteren Verlauf dieser LE werden die meisten Beispiele aus dem Java-Umfeld stammen. Es muss jedoch betont werden, dass auch für andere aktuelle Programmiersprachen SAX- und DOM-Bibliotheken zur Verfügung stehen (siehe auch den nächsten Abschnitt 5.4.3). Die Verwendung von SAX und DOM erfolgt in diesen Sprachen analog zu den vorgestellten Java-Beispielen. Natürlich muss man die Anwendung an die Syntax der jeweiligen Programmiersprache anpassen. Das grundlegende Verwendungskonzept bleibt jedoch gleich.

SAX als ereignisbasiertes API

Wie im Abschnitt 5.2 bereits 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 kein Wissen über vorangehende oder nachfolgende Elemente hat. Aus dieser Tatsache resultiert ebenfalls, dass ein SAX-Parser kein Wissen über die strukturelle Einbettung eines Elements hat. Der Programmierer muss sich selbst um den Aufbau von internen Repräsentation 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 XML-*dev* Mailingliste. Die aktuelle Versionsnummer der Bibliothek ist SAX 2.0. Wenn in der Folge dieser LE von SAX geredet wird, so ist damit, falls nicht anders gekennzeichnet, immer SAX 2.0 gemeint. Die Ursprungsversion SAX 1.0 wird von dem API weiterhin unterstützt. Allerdings sind die meisten Methoden der Version 1.0 als "deprecated" markiert, werden also in der Zukunft nicht weiterentwickelt und sollten deshalb nicht mehr verwendet werden. Neben Java unterstützen inzwischen eine Reihe weiterer Programmiersprachen SAX 1.0 und SAX 2.0. Damit erreicht SAX einen sehr hohen Verbreitungsgrad hat sich so als de facto Standard etabliert. Die folgenden Parser implementieren SAX 2.0:

- Xerces-J: ist Teil des Apache XML Projekts, unterstützt auch JAXP.
- AElfried2: ist Teil des GNU JAXP Projekts.
- Crimson: ist Teil des Apache XML Projekts, wird als Bestandteil des Java Development Kits (JDK 1.4) von SUN ausgeliefert, unterstützt auch JAXP.
- Oracle's Java XML Parser: ist Bestandteil der Oracle Produkte.
- XP 0.5 von James Clark : ein schneller XML-Parser, der jedoch nicht weiterentwickelt wird (SAX 2.0 wird durch externe Erweiterungen unterstützt).

SAX folgt bei der Verarbeitung der XML-Daten einem *ereignisorientierten* Ansatz (*Event-Based Processing*). Im Verlauf des Parsens werden, in Abhängigkeit von Inhalt und Struktur des Eingabedokuments, Ereignisse (Events) generiert. Die wiederum lösen den Aufruf von Verarbeitungsmethoden (Callback-Methoden) aus. Im

Gegensatz zu Tree-Building-basierten Ansätzen, wie etwa DOM, ist bei SAX die eigentliche Verarbeitung des XML-Dokuments also in das Parsen integriert.

Die Callback-Methoden sind wiederum definiert als Teil von größeren Interfaceklassen. Diese werden als *EventHandler* oder *ContentHandler* bezeichnet. SAX definiert vier Hauptklassen, 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 Entitäten (*EntityResolver*). Mit Hilfe dieser *EventHandler*-Klassen kann SAX eine Vielzahl von unterschiedlichen Ereignissen verarbeiten. So sind z.B. im *ContentHandler* Methoden definiert, die

- auf den Anfang/das Ende des Parsevorgangs
- auf Processing Instructions
- auf den Wechsel des Namespaces
- auf Starttags und Endtags
- auf den Inhalt eines Elements
- auf das Auftreten von Whitespaces (Return, Tabulator und Newline)

reagieren sollen. Im Folgenden wird schrittweise eine einfache Java Anwendung entwickelt werden, in der SAX verwendet wird, um ein XML-Dokument zu parsen.

Einige Vorbemerkungen zu der Verwendung von Java

Um die nachfolgenden 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 1.4) kostenlos zur Verfügung. Die Hauptkomponenten sind dabei der Compiler **javac** und der Interpreter **java**. Mit dem Compiler wird ein Programm übersetzt, mit dem Interpreter kann man ein Programm starten. Der Programmierer kann ein Java-Programm in mehrere separate Programmdateien aufteilen. Innerhalb jeder einzelnen Programmdatei existieren zwei Blöcke: der optionale **import**-Block und der obligatorische **class**-Block.

Um die Funktionen (in Java *Methoden* genannt) von externen Programm-Bibliotheken (in unserem Fall beispielweise Methoden aus SAX, DOM oder Xerces) verwenden zu können, müssen diese in das Java-Programm importiert werden. Dies geschieht für jede Bibliothek einzeln durch eine **import** Anweisung. Import-Anweisungen stehen normalerweise am Anfang eines Java-Programms. So kann man auf einen Blick erkennen, welche externen Bibliotheken das Programm verwendet. Die folgende Anweisung

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

importiert den SAXParser der Java-Version des XML-Parsers Xerces. In anderen Programmiersprachen gibt es vergleichbare Anweisungen. In Perl-Programmen muss man beispielsweise die Anweisung `use XML::Xerces` eingeben, um Xerces benutzen zu können.

Java-Programme werden in sogenannten *Klassen* organisiert. Entsprechend steht im Anschluss an die Import-Anweisungen die Vereinbarung des Klassennamens. Diese erfolgt mit Hilfe der `class` Anweisung. Hierbei ist es wichtig die folgende Konvention zu beachten: die Datei in der ein Java-Programm gespeichert wird muss *exakt* den gleichen Namen tragen, wie der mit der class Anweisung vereinbarte Klassenname. Als Endung muss für die Datei `.java` gewählt werden. So muss z.B. eine Klasse, die mit der Anweisung `class MySAXParser` vereinbart worden ist in der Datei `MySAXParser.java` abgespeichert werden. Innerhalb der folgenden Beispielprogramme werden wir immer nur eine Klasse pro Datei definieren. Entsprechend umfasst der class-Block in allen Beispielen den gesamten Rest der Programmdatei. In Java verwendet man die geschweiften Klammern (`{` und `}`), um den Anfang und das Ende des Blocks zu markieren.

Innerhalb des Klassenblocks werden schließlich die einzelnen Methoden der Klasse aufgeführt. Für jede Methode definiert 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 Methodenname festgelegt, gefolgt von der Vereinbarung die Parameterliste. Diese kann natürlich auch leer sein. Die Definition

```
public void setDocumentLocator(Locator l)
```

definiert also eine Methode die keinen Rückgabewert liefert (`void`), den Namen `setDocumentLocator` trägt und einen Parameter mit Namen `l` vom Typ `Locator` hat (`Locator l`). 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 Methoden vor, die der Programmierer in seiner Anwendung umsetzen muss. Damit wird sichergestellt, dass sich eine Klasse an vorgegebene Standards hält 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. Der Anwender muss diese dann selbst programmieren. Um festzulegen, dass eine Klasse der Definition einer Interface-Klasse genügt, verwendet man die `implements` Anweisung:

```
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 besprochenen Java-Elemente werden in den folgenden Beispielen verwendet. Beginnen wir nun mit der Entwicklung einer einfachen XML-Anwendung.

SAX und Xerces

Grundlage der Anwendung ist der SAX-konforme XML-Parser Xerces-2 von Apache¹. Der Parser wird in Form einer zip-Datei ausgeliefert, die die Bibliotheken (xercesImpl.jar, xercesParserAPIs.jar) als jar-Archive enthält. Diese müssen so installiert werden, dass der Java-Interpreter die Bibliotheken finden und laden kann. Hierzu muss der Klassenpfad erweitert werden:

- indem die Umgebungsvariable CLASSPATH angepasst wird, oder
- durch Nutzung des Kommandozeilenschalters **-classpath** von java.exe oder
- indem man die jar-Dateien in das Unterverzeichnis \$JAVA_HOME\jre\lib\ext kopiert.

Die letzte Option installiert die Bibliotheken systemweit. \$JAVA_HOME steht hierbei für das Installationsverzeichnis der Java Entwicklungsumgebung, bzw. der Java Runtime-Umgebung (z.B. C:\JDK1.3.1_02). Alle Java Anwendungen, 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 eigentlich Implementierung des XML-Parsers, während in xercesParserAPIs.jar die implementierten APIs definiert worden sind. Das SAX API liegt als Klassenpaket unter org.xml.sax vor. Auf dieser Ebene umfasst es 17 Klassen. Im Unterpaket org.xml.sax.helpers sind 9 Klassen definiert, das Unterpaket org.xml.sax.ext enthält weitere 2 Klassen. Für eine genauere Darstellung der SAX Klassen siehe auch <http://www.saxproject.org/apidoc/overview-summary.html>².

Ein minimaler SAX-Parser

Grundlage der SAX-Anwendung ist die **XMLReader** Interface-Klasse, welche in **org.xml.sax.XMLReader** definiert ist. Das Interface legt alle SAX2 konformen Methoden, Features-flags und Properties fest (siehe auch den folgenden Abschnitt [5.4.2.3]). Als abstrakte Klasse kann sie nicht instanziiert werden sondern muss entsprechend in der jeweiligen Parser-Bibliothek als konkrete Klasse implementiert werden. Bei Xerces nutzt man dazu die Klasse SAXParser (**org.apache.xerces.parsers.SAXParser**). In XMLReader ist die Methode **parse(java.lang.String URI)** definiert, mit der eine XML-Datei durch den XMLReader geparkt werden kann. Die minimale SAX Anwendung besteht also aus den Zeilen:

```
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 Interfaceklasse, die die SAX-Methoden definiert und `SAXParser` ist deren Umsetzung durch Xerces. Die Klasse trägt den Namen `MinimalSaxParser`, innerhalb derer eine Methode namens `main` definiert wird. Die Methode `main` ist standardgemäß die Methode mit der ein Java-Programm gestartet wird. Für das Parsen der Datei sind lediglich die beiden Zeilen

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

verantwortlich. Zunächst wird der Parser erzeugt und initialisiert. Anschließend wird mit der Methode `parse`, die an das Programm übergebene XML-Datei geparst.

Der so definierte minimale SAX-Parser muss in der Datei **MinimalSaxParser.java** gespeichert werden und kann anschließend mit dem Befehl **javac MinimalSaxParser.java** kompiliert werden. Das ausführbare Programm erwartet als Parameter eine XML-Datei. Diese wird durch den SAXParser geparst:

```
java MinimalSaxParser test.xml
```

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

Content Handler: die Verarbeitung von XML mit SAX

Im nächsten Schritt soll die minimale Anwendung um Callback-Methoden erweitert werden. Hierzu soll ein `ContentHandler` implementiert werden, der die Start-Tags ausgibt. Das Interface `org.xml.sax.ContentHandler` definiert 11 Callback-Methoden, die auf spezifische Events reagieren können:

Tabelle 1-3. Callback-Methoden von `ContentHandler`

<code>setDocumentLocator</code>	Wird genau einmal zu Beginn des Parsens ausgelöst. Das <code>Locator</code> -Object stellt in der Folge der Verarbeitung Datei-Informationen zur Verfügung.
---------------------------------	---

startDocument	Wird genau einmal zu Beginn des Parsens ausgelöst.
endDocument	Wird genau einmal am Ende des Parsens ausgelöst, und zwar unabhängig davon, ob das Parsen erfolgreich war oder nicht.
processingInstruction	Wird ausgelöst, wenn eine Processing Instruction (PI) geparkt wurde. Dieser Event wird nicht ausgelöst durch die PI, die die XML-Deklaration definiert (<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>)
startPrefixMapping	Wird ausgelöst, wenn ein Namespace beginnt.
endPrefixMapping	Wird ausgelöst, wenn ein Namespace endet.
startElement	Wird ausgelöst, wenn ein Start-Tag geparkt wurde.
endElement	Wird ausgelöst, wenn ein End-Tag geparkt wurde.
characters	Wird ausgelöst, wenn der Inhalt eines Elements geparkt wurde (#PCDATA).
ignorableWhitespace	Wird ausgelöst, wenn nicht relevante Folgen von Whitespace (Return, Space und Newline) geparkt wurden.
skippedEntity	Wird ausgelöst, wenn eine Entität übersprungen wurde. Nicht-validierende Parser sind gemäß XML 1.0 nicht dazu verpflichtet Entitäten aufzulösen und können diese überspringen.

Definition und Nutzung der Callback-Methoden

Die folgende einfache Implementation von ContentHandler füllt lediglich zwei Callback-Methoden: **setDocumentLocator** und **startElement**. In setDocumentLocator wird das aktuelle Locator-Object in der lokalen Variable LOCATOR zwischengespeichert. Damit wird sichergestellt, dass der Parser im Verlauf des Parsens auf Dateiinformationen, wie etwa die aktuelle Zeilennummer oder den aktuell Spaltenindex zugreifen kann. Mit **startElement** reagiert der SAX-Parser jeweils auf die öffnenden Tags. Die Methode besteht lediglich aus einer Ausgabeanweisung, die die aktuelle Zeilennummer gefolgt von dem in <> geklammerten Elementnamen auf der Standardausgabe ausgibt.

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

```
public class SimpleContentHandler implements ContentHandler {
    private Locator LOCATOR = null;

    // CALLBACKS
    public void setDocumentLocator(Locator l) {
LOCATOR = l;
    };

    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 rn, Attributes atts) {
System.out.println (LOCATOR.getLineNumber() + ":\t<" + localName + ">");
    };

    public void endElement(String namespaceURI, String ln, String rn) {};

    public void characters(char[] chars, int start, int end) {};

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

Um den einfachen ContentHandler während des Parsens nutzen zu können, muss eine Instanz von diesem bei dem SAXParser registriert werden. Hierzu stellt das Interface XMLReader die Methode **setContentHandler** zur Verfügung. Entsprechend wird der minimale SAXParser um die folgende Zeile erweitert:

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

Speichert man den so erweiterten SAX-Parser unter **ExtendedSaxParser.java** ab, kann dieser anschließend mit **javac ExtendedSaxParser.java** kompiliert werden. Der Aufruf der Applikation erfolgt dann analog zum minimalen Parser mit

```
java ExtendedSaxParser test.xml
```

Der registrierte ContentHandler sorgt jetzt dafür, dass jeweils die Zeilennummer und zugehörige Starttag auf dem Bildschirm ausgegeben werden. So wird es möglich,

die schrittweise Verarbeitung der XML-Datei zu beobachten. Das Beispiel zeigt, wie vergleichsweise einfach die Verarbeitung von XML-Dateien mit dem SAX Modell ist.

Weitere Content Handler

Wie bereits weiter oben erwähnt definiert SAX noch drei weitere EventHandler-Klassen: den ErrorHandler, den EntityResolver und den DTDHandler. Die folgenden Tabellen geben eine Kurzübersicht über die jeweiligen Callback-Methoden der einzelnen Klassen.

Für die Behandlung von Parsefehlern stellt die ErrorHandler-Klasse folgende Methoden zur Verfügung:

Tabelle 1-4. Callbacks von ErrorHandler

Error(SAXParseException ex)	Wird ausgelöst, wenn ein behebbarer Fehler auftritt (recoverable error, gemäß XML 1.0).
FatalError(SAXParseException ex)	Wird ausgelöst, wenn ein nicht-behebbarer Fehler auftritt (non-recoverable error, gemäß XML 1.0)
warning(SAXParseException ex)	Wird ausgelöst, wenn eine Warnung auftritt. (warning, gemäß XML 1.0)

Der Registrierung eines ErrorHandler erfolgt durch die Methode **setErrorHandler** der Klasse XMLReader.

Der EntityResolver ist Grundlage für die applikations-spezifische Behandlung externer Entitäten. Die Klasse EntityResolver stellt lediglich eine Callback-Methode zur Verfügung:

```
InputSource resolveEntity(String publicId, String systemId)
```

Ein SAX-konformer XML-Parser ruft diese Methode immer dann auf, wenn eine externe Entität aufgelöst werden soll. Als Ergebnis liefert die Methode ein Objekt der Klasse InputSource, eine Referenz auf die Datenquelle, aus der die SAX-Anwendung den eigentlichen Wert der Entität lesen kann. Die InputSource-Klasse kapselt die unterschiedlichen möglichen Datenquellen von SAX in einem einzigen Objekt, so dass Public- und System-Identifizierer, ein Eingabe-Byte-Stream (mit dem gewählten Character-Encoding) und/oder ein Eingabe-Character-Stream (ebenfalls mit dem gewählten Character-Encoding) direkt zugreifbar sind. InputSource-Objekte können in SAX alternativ zu URI/URLs verwendet werden. Der Registrierung eines EntityResolvers erfolgt durch die Methode **setEntityResolver** der Klasse XMLReader.

Der DTDHandler schließlich erlaubt die Reaktion auf Ereignisse, die beim Parsen der DTD auftreten.

Tabelle 1-5. Callbacks des DTDHandler

notationDecl	
---------------------	--

unparsedEntityDecl	
--------------------	--

Konfiguration des SAX-Parsers: Feature-Flags und Properties

SAX erlaubt die genauere Konfiguration der Arbeitsweise des XML-Parsers durch eine Menge sogenannter Feature-Flags. Diese können durch entsprechende Get/Set-Methoden Eigenschaften abgefragt und verändert werden (**getFeature**, **setFeature**). Anhand der Flags ist es beispielweise möglich festzustellen, ob der SAXParser validierend oder nicht-validierend arbeitet:

```
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 Attribute spezifiziert, deren syntaktischer Aufbau einer URI entspricht (z.B. <http://xml.org/sax/features/validation>). Der Satz der Feature-flags 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 Defaultwert definiert. Lediglich zwei Feature-flags 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 Featwewerte müssen (mindestens) auslesbar sein. **Namespaces** muss den Wert **true** liefern, **namespace-prefixes** muss den Wert **false** liefern. Hierdurch wird sichergestellt, dass der Parser eine minimale Unterstützung von Namespaces gewährleistet, welche durch höhere APIs genutzt wird. Neben den beiden genannten Feature-flags werden noch 11 weitere optionale Feature-flags für SAX2 definiert.

Ein weitere Möglichkeit zur Konfiguration des Parsers in SAX2 bieten *Properties*. Diese können genau wie Feature-Flags ausgelesen und gesetzt werden (**getProperty**, **setProperty**). Im Gegensatz zu den Feature-Flags speichern Properties *Objekte*, die das Parseverhalten im Einzelnen beeinflussen. Durch die Properties wird es beispielsweise möglich, Erweiterungen, bzw. Ergänzungen der vordefinierten SAX2 Event-Handler zu implementieren. Alle Properties müssen

im Namespace `http://xml.org/sax/properties/` liegen. Vier Standard-Properties (`declaration-handler`, `dom-node`, `lexical-handler` und `xml-string`) sind festgelegt, alle sind jedoch optional und müssen deshalb nicht notwendigerweise von der eingesetzten Bibliothek unterstützt werden.

Greift eine Applikation versucht auf ein nicht *definiertes* Feature-flag oder eine nicht definierte Property zu, wird eine **SAXNotRecognizedException** ausgelöst. Dieser Ausnahmetypus tritt relativ häufig auf, da damit auch Schreibfehler in den verwendeten URIs quittiert werden. Versucht man auf ein Feature/Property zuzugreifen, das nicht unterstützt wird, so wird eine **SAXNotSupportedException** ausgelöst. Dieser Fehler tritt beispielsweise dann auf, wenn versucht wird, einen nicht validierenden Parser durch Setzen des Feature-flag **validation** auf **true** zum Validieren zu bringen:

```
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());
    }
    }
}
```

W3C DOM

Das *Document Object Model* (DOM) wird vom W3C definiert. SAX steht als PublicDomain Software zur allgemeinen Verwendung bereit und hat sich als Quasi-Standard etabliert. Im Gegensatz dazu ist DOM ein "echter" W3C-Standard ist (siehe auch <http://www.w3.org/DOM/>³ oder <http://xml.coverpages.org/dom.html>⁴). DOM ist, genau wie SAX, nicht spezifisch auf die Verwendung mit einer bestimmten Programmiersprache ausgelegt.

Die Grundidee bei der Entwicklung von DOM war es, eine allgemeines Format zur Repräsentation von Dokumenten festzulegen, und zwar unabhängig von einer konkreten Programmiersprache und der verwendeten Plattform. Ursprünglich wurde DOM für den Einsatz in WebBrowsern entwickelt. Als Repräsentationsformat wird es zur internen Speicherung dynamisch erzeugter HTML-Seiten verwendet.

Inzwischen existieren Anbindungen, sogenannten *Bindings*, für eine größere Anzahl von Programmiersprachen (z.B. Java, Perl, Python, JavaScript, CORBA). In der jeweiligen Programmiersprache werden die Vorgaben von DOM auf die sprachinternen Repräsentationen abgebildet. Das DOM bildet so für den Programmierer eine genormte Zwischenschicht, die wie bei SAX, von der eigentlichen Implementation abstrahiert.

DOM fokussiert die Repräsentation von Dokumenten und definiert nur wenige Beschränkungen hinsichtlich der Methoden, die diese Strukturen manipulieren. Für den Anwender bedeutet dies, dass auch dann wenn zwei Bibliotheken DOM-konform sind, sie sehr einen unterschiedlichen Methodenumfang und Methodenaufbau haben können. Trotz dieses Nachteils ist DOM auf einer Vielzahl von Plattformen lauffähig und kann in unterschiedlichen Programmierumgebungen genutzt werden.

Das DOM API ist ausgelegt auf die Verarbeitung von XML und HTML. Entsprechend ist DOM in drei Teile aufgeteilt: *Core*, HTML und XML. Das *DOM Core* definiert eine Menge von Objekten und Methoden, die es erlauben beliebige strukturierte Dokumente zu repräsentieren. Im Prinzip kann man bereits mit dem DOM Core alle XML/HTML-Dokumente wiedergeben. Andererseits bieten die Funktionen von DOM Core nur ein geringen Abstraktionsgrad (das DOM Core ist ein *low level interface*), was bei der Entwicklung komplexerer XML-Anwendungen zu Problemen führen kann. Entsprechend bieten DOM-XML/HTML dem Entwickler höhere Erweiterungsfunktionen (*high level interface*) an, wodurch es vergleichsweise einfach wird, diese Dokumenttypen mit DOM zu parsen, zu erzeugen, zu bearbeiten und zu speichern. Die unterschiedlichen DOM-Versionen sind organisiert in sogenannten *DOM-Levels*:

- *Level 1*: definiert DOM Core, sowie die HTML und XML Dokumentmodelle. Level 1 beinhaltet Methoden zur Bearbeitung von Dokumentinhalten und zur Navigation innerhalb der Dokumente
- *Level 2*: definiert das StyleSheet Objektmodell und Methoden zur Verarbeitung von StyleSheets, die an Dokumente gebunden sind. Zusätzlich ist ein Eventmodell definiert und die Unterstützung von Namespaces ist integriert.
- *Level 3*: Definiert Methoden zum Speichern/Laden von Dokumenten. Die Beschreibung von Dokumenttypen (z.B. durch DTDs oder Schemas) ist möglich und damit gleichzeitig die Validierung der Dokumente. Level 3 definiert Sichten auf die Dokumente, sowie die Formatierung von Dokumenten.

Da die Definition eines Standards durch das W3C einen sehr aufwendigen Prozess darstellt, existieren eine Vielzahl von Dokumenten zu den jeweiligen Teildefinitionen von DOM-Level 1,2,3. Diese im Details darzustellen würde den Rahmen dieser LE sprengen. Entsprechend fokussieren wir im weiteren Verlauf den DOM-Level1 Core.

DOM repräsentiert das 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 die Anwendung darauf zugreifen .

Alle Knoten eines DOM-Baumes sind abgeleitet von der generischen Interfaceklasse **org.w3c.dom.Node**. In einem Node werden alle wesentlichen Eigenschaften des Knotens repräsentiert, z.B. der übergeordnete Knoten (**parentNode**), die Kinderknoten (**childNodes**), der erste (**firstChild**) und letzte (**lastChild**) Kindknoten, der vorher-

rige (**previousSibling**) und nachfolgende (**nextSibling**) Schwesterknoten. Auch der Knotentyp (**nodeType**) wird festgelegt. Folgenden Typen werden in DOM definiert:

Tabelle 1-6. DOM Knotentypen

org.w3c.dom.Document	Repräsentiert die Wurzel des Baumes
org.w3c.dom.DocumentType	Beschreibt den Dokumenttyp eines XML-Dokuments. Im Wesentlichen speichert dieser Knoten die Liste der für das Dokument definierten Entitäten (als NamedNodeMap). Über die NamedNodeMap kann man auf eine Menge von Knoten über ihren Namen zugreifen.
org.w3c.dom.Element	Element repräsentiert ein XML-Element, sowie dessen Attribute. Die Attribute können in ihrer Gesamtheit als Menge oder jeweils einzeln durch ihren Namen ausgelesen werden.
org.w3c.dom.Attr	Attr repräsentiert ein Attribut eines Element-Knotens.
org.w3c.dom.Text	Text repräsentiert den Inhalt eines Elements
org.w3c.dom.CDATASection	CDATASection repräsentiert den in CDATA eingebetteten Text eines XML-Dokuments. CDATA Sektionen werden vom XML-Parser nicht geparkt, sondern unverändert in den DOM-Tree übernommen.
org.w3c.dom.Comment	Repräsentiert einen Kommentar.
org.w3c.dom.ProcessingInstruction	Command repräsentiert eine Processing-Instruction. Ziel und Daten der Processing-Instruction werden unverändert übernommen.
org.w3c.dom.Notation	Notation repräsentiert eine Notation. publicID und systemID werden unverändert übernommen.
org.w3c.dom.Entity	Entity repräsentiert eine Entität. Je nach Typ wird dieser geparkt oder unverändert übernommen.
org.w3c.dom.EntityReference	EntityReference repräsentiert die Referenz auf eine Entität.
org.w3c.dom.DocumentFragment	Ein DocumentFragment repräsentiert einen Ausschnitt aus einem DOM-Tree. Hierbei ist es nicht notwendig, dass der Ausschnitt wohlgeformt nach XML 1.0 ist.

DOM und Xerces

Ähnlich wie bei den Erläuterungen zu SAX soll auch in dieser LE eine einfache DOM-Anwendung schrittweise entwickelt werden. Wir verwenden wieder Xerces, den XML-Parser von Apache. Der Grundaufbau der Anwendung entspricht dabei dem SAX-Beispiel: erst müssen die notwendigen Bibliotheken eingebunden werden, dann kann der Parser erzeugt werden und schließlich wird mit der `parse` Methode die XML-Datei verarbeitet. Die Datei liegt dann im Speicher als DOM-Tree vor, der anschließend rekursiv durchlaufen werden kann.

Beispiel 1-6. Schritt1: Importierung der Bibliotheken

Für die Nutzung von DOM müssen die entsprechenden Bibliotheken importiert werden. Diese sind im Paket `org.w3c.dom` definiert. Für die jeweiligen Knotentypen existieren entsprechend benannte Klassen. So findet man z.B. die Definition der DOM Knoten unter `org.w3c.dom.Node`. Für die Anwendung werden der DOMParser von Xerces, die benötigten DOM-Knotentypen und alle zur Fehlerbehandlung notwendigen Ausnahmen importiert.

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

// Knotentypen des DOM
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

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

Beispiel 1-7. Schritt 2: Erzeugung und Konfiguration des DOM-Parsers

Nach dem Importblock folgt die Klassendefinition. In diesem Fall wird die Klasse `ValidatingDOM` definiert. Das Programm muss entsprechend in der Datei `ValidatingDOM.java` abgespeichert werden. Der DOMParser wird erzeugt und durch Setzen des entsprechenden Feature-Flags `validation` auf den Wert `true` in den validierenden Arbeitsmodus geschaltet. Wie man sieht werden 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. Der SAXParser baut dabei keine Repräsentation auf. Für die Entwickler der Xerces DOMParsers lag es damit natürlich nahe SAX für das Parsen der XML-Datei zu nutzen, wodurch sich der Aufwand für die Implementation des DOMParsers auf die Entwicklung von Routinen zum Aufbau des DOM-Trees reduziert. Der DOMParser basiert also auf dem SAX-Parser und 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);
}
```

Beispiel 1-8. Schritt 3: Parsen der Datei und Aufruf der Ausgaberroutine

Im nächsten Schritt wird die angegebene XML-Datei (**xmlFile**) mit der Methode **parse** geparkt. Treten keine Fehler auf liegt das Dokument jetzt als DOM-Tree im Speicher vor. Die Methode **getDocument** liefert den Wurzelknoten des Dokuments und dieser wird in der Variablen **document** zwischengespeichert.

```
// 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);
}
}
```

Beispiel 1-9. Schritt4: Rekursive Ausgaberroutine

Ausgehend vom Wurzelknoten des DOM-Trees bestehen jetzt alle Möglichkeiten 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 "hängelt" sich gewissermaßen an der Struktur des Baumes entlang. Dies gelingt am einfachsten durch die Definition einer rekursiven Methode. Wie bereits in LE 2.3 erläutert sind XML-Bäume selbstähnlich, bestehen also selbst wieder aus Teilbäumen.

Die Traversierungsroutine **traverse** nutzt diese Eigenschaft aus, um die Elemente des Baums auf dem Bildschirm auszugeben. Zuerst wird der Typ des aktuell betrachteten Knoten **node** überprüft. Handelt es sich um einen Element-Knoten (**Node.ELEMENT_NODE**), so wird der Name des Knoten ausgegeben. Anschließend wird die Liste der möglichen Kinderknoten berechnet und in der **NodeList children** abgespeichert. Ist diese nicht leer (**children != null**), so wird für jeden einzelnen Kindknoten wiederum die Methode **traverse** (*rekursiv*) aufgerufen

und der Inhalt des Kindes ausgegeben. So durchläuft die Methode schrittweise alle Elementknoten des DOM-Trees.

```
// 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 Kind-Knoten durch rekursiven Abstieg
NodeList children = node.getChildNodes();
if (children != null) {
    for (int i=0; i< children.getLength(); i++)
        traverse (children.item(i));
}
}
```

Beispiel 1-10. Schritt 5: Start der Verarbeitung im Hauptprogramm

Das Hauptprogramm der Anwendung (**main**) besteht lediglich aus einer Zeile, in der eine Instanz des validierenden DOM-Parsers erzeugt wird, die die übergebene XML-Datei verarbeitet.

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

Beispiel 1-11. Schritt 6: Übersetzung und Aufruf der fertigen Anwendung

Das Programm kann nun mit **javac ValidatingDOM.java** übersetzt werden und anschließend mit

```
java ValidatingDOM test.xml
```

gestartet werden. Der DOM-Parser verarbeitet nun die Datei **test.xml** und gibt deren Elemente auf dem Bildschirm aus.

Zeichensätze und Entities

In diesem Abschnitt beschäftigen wir uns zunächst mit der maschinellen Kodierung von Zeichen. Eine wichtige Rolle spielen dabei Character-Entities. Wir beschreiben den UNICODE-Standard und zeigen, wie man Kodierungen im Dokument angibt oder Texte in verschiedene Kodierungen überführt.

Grundlegende Begriffe

Um von einem Zeichen zu seiner Ausgabe auf dem Bildschirm oder im Druck zu kommen, müssen zwei Bedingungen erfüllt sein. Zum einen müssen die Zeichen in maschinenlesbarer Form kodiert werden. Zum anderen müssen die kodierten Zeichen mit Einheiten der Ausgabe, den Schriftarten (engl. *Fonts*) verbunden werden. Diese Prozesse der Zeichenkodierung und der Zeichenausgabe werden wir nun genauer betrachten.

Zeichenkodierung

Zeichenkodierung betrifft die folgenden Aspekte:

- Ein Zeichenvorrat (engl. *Character Repertoire*) muss ausgewählt werden. Man muss sich überlegen, ob man z. B. nur lateinische Buchstaben und deutsche Umlaute verwenden möchte, oder auch bestimmte Sonderzeichen oder Zeichen aus anderen, z. B. außereuropäischen Schriftsystemen.
- Die Zeichen müssen numerische Identifikatoren haben, also Zahlen. Jedes Zeichen darf nur genau einen Identifikator, einen sogenannten *Code Point* erhalten, da sonst die eindeutige Verarbeitung der Zeichen nicht gewährleistet ist. Meist erfolgt die Zuordnung der Zeichen zu den Identifikatoren in Form einer Tabelle, vgl. z. B. die Tabellen des UNICODE-Repertoire⁵. Die Menge aller numerischen Identifikatoren für einen bestimmten Zeichenvorrat nennt man *Coded Character Set*.
- Zwei weitere Schritte sind für die maschinelle Verarbeitung der Zeichen wichtig. Zunächst muss die Anzahl der Bytes pro Zeichen bestimmt werden. Mit einem Byte lassen sich Zahlen mit den Werten 0 bis 255 darstellen. Wenn man nur auf lateinbasierte Schriften zurückgreift und wenig Sonderzeichen verwendet, reicht dies aus. Andere Schriftsysteme, etwa bildbasierte Systeme wie das Chinesische, bestehen aus tausenden von Zeichen und brauchen dementsprechend viele numerische Identifikatoren. Hier kommt man nur mit zwei, drei oder vier Byte aus. Eine *Character Encoding Form* bildet die einzelnen numerischen Identifikatoren ab auf bestimmte Bytelängen. Als zweiter Schritt muss für die Character Encoding Form ein *Character Encoding Scheme* entwickelt werden. Dabei wird z. B. für Mehrbyteeinheiten bestimmt, welches Byte als erstes steht - das 'oberste' oder das 'unterste' - und wie Wechsel zwischen den Bytelängen geregelt werden sollen.

Die Verbindung eines Character Encoding Scheme mit dem dazugehörigen Coded Character Set bezeichnet man als *Character Set* "Zeichensatz". Die Namen der Character Sets sind standardisiert durch einen IANA *charset identifier*, deren genaue Bezeichnung sich in der Liste offizieller Namen von Character Sets⁶ wiederfindet. Beispiele für weit verbreitete Character Sets sind *ASCII* (American Standard Code for Information Interchange) oder die Mitglieder des Standards *ISO-8859*.

Zeichenausgabe

Einen Character Set zu definieren bedeutet noch nicht, Zeichen auf dem Bildschirm oder im Druck darstellen zu können. Bis zur Ausgabe der Zeichen müssen folgende Punkte erfüllt werden:

- Für ein Zeichen müssen die grundlegenden Formtypen der Ausgabe bestimmt werden. Diese Formtypen nennt man *Glyphen*. Ein Zeichen lässt sich durch verschiedene Glyphen darstellen, etwa ein 'A' in Schreibmaschinenschrift versus einem 'A' in handschriftlicher Form, in einer historischen oder regionalen Variante (vergleiche z. B. ein griechisches und ein deutsches 'A') etc.
- Für die Glyphen müssen sogenannte *Glyph Images* bestimmt werden. Sie sind die konkreten Visualisierungen der Glyphen. So kann man ein 'A' in Schreibmaschinenschrift fett drucken oder kursiv.
- Die Sammlung von Glyph Images bezeichnet man als *Font*. Beispiele für weit verbreitete Fonts sind Times, Helvetica, Arial.

Vorsicht ist geboten bei der direkten Übersetzung englischer Terminologie. Fonts werden manchmal als "Zeichensätze" bezeichnet, was sie scheinbar in die Nähe von Character Sets rückt - ein großes Missverständnis!

Die folgende Tabelle fasst die beschriebenen Schritte vom Zeichen zur Zeichenausgabe zusammen:

Tabelle 1-7. Vom Zeichen zur Zeichenausgabe

Zeichen	Numerischer Identifikator	Wiedergabe durch Bytes	Glyphvarianten	Fonts
Der Großbuchstabe A	65	ein Byte: 1000001	Druckschrift	Schriftart 'Times'

Character Entities

Character Entities werden zu unterschiedlichen Zwecken verwendet. Zum einen können mit ihnen Zeichen dargestellt werden, die normalerweise eine bestimmte Bedeutung im XML-Dokument haben, wie die eckigen Klammer < > oder das Kaufmannsund &. Zum anderen kann mit ihnen ein großes Zeichenrepertoire erfasst werden. Character Entities werden oft in separaten Dateien mit der Endung ".ent" definiert. Auch das Docbook-System, mit dem dieser Text erstellt wurde, nutzt solche Entitäten, etwa für mathematische Sonderzeichen.

Die Definition der Character Entities hat das folgende Muster:

Tabelle 1-8. Definitivsmuster für Character Entities

Kaufmannsund	Entitätsname (oder) Entitätsnummer	Semikolon
&	lt (oder) #60	;

<

Im Beispiel wird die linke eckige Klammer definiert. Nach dem Kaufmannsund steht der Entitätsname - wobei Groß- und Kleinschreibung zu beachten ist - oder die Entitätsnummer. Häufig verwendete Zeichen haben Namen wie 'lt' für die linke eckige Klammer. Diese sind leichter zu merken als die Entitätsnummern, sie werden allerdings nicht unbedingt von jedem Programm interpretiert. Die Entitätsnummer wird durch das Doppelkreuz '#' eingeleitet. Folgt danach ein 'x', handelt es sich um eine Zahl im Hexadezimalsystem, ansonsten steht sie im Dezimalsystem.

Der UNICODE Standard

Es gibt eine Vielzahl von Character Sets, die Zeichenvorräte in Abhängigkeit von regionalen, sprachlichen, historischen etc. Varianten erfassen. Viele dieser Character Sets, wie etwa die ISO-8859 Familie, gehen vom ASCII-Zeichensatz aus. ASCII definiert 127 Zeichen, die sich mit 7 Bit darstellen lassen. Die ISO-8859 Familie definiert verschiedene, hauptsächlich europäische Zeichen, deren Identifikatoren oberhalb dieser Nummer im Bereich 128-255 liegen. Diese Character Sets sind also sozusagen Erweiterungen von ASCII.

Das Vorgehen der ISO-8859 Familie hat den Vorteil, dass ein Computer zumindest ASCII verarbeiten kann, auch wenn dem System die Zeichen oberhalb der Nummer 127 nicht bekannt sind. Bei mehr als 255 Zeichen stößt man jedoch schnell an Grenzen. Der auch in XML angewandte Standard ISO/IEC 10646, definiert durch das UNICODE-Konsortium, schafft Abhilfe. UNICODE nimmt keine Überlagerungen von Zeichen an, sondern bestimmt für jedes Zeichen genau einen Identifikator: Der Standard differenziert die Zeichen nicht mehr nach ihrem regionalen oder einem anderen Verwendungszusammenhang und fasst sie soweit wie möglich zusammen. Das *Basic multilingual plane* (BMP) enthält 65535 Zeichen.

Anmerkung: Die Auswahl der in UNICODE zu kodierenden Zeichen ist nicht so einfach, wie es scheinen mag. Sprachspezifische Varianten mit gleicher Bedeutung werden als verschiedene Glyphen eingestuft. Da UNICODE Zeichen, aber nicht Glyphen umfassen soll, sind diese Varianten nicht Teil des Standards. Generalisierungen von regionalen, historischen oder anderen Unterschieden zwischen ähnlichen Zeichen sind oft Gegenstand heftiger Diskussionen zwischen den einzelnen Interessengruppen mit dem Ziel: "Eine Nummer für mein Zeichen!".

Aufbau von UNICODE

UNICODE unterteilt die Zeichendefinitionen in sogenannte *Skripte*. Für die ideographischen Zeichen des asiatischen Raumes gibt es z. B. ein Skript, das chinesische, japanische und koreanische Varianten generalisiert. Die ASCII-Zeichen erhalten die gleichen Identifikatoren, also Code Points zwischen 0-127 wie im ASCII-Standard selbst. Eine Liste der UNICODE Character Sets⁷ verzeichnet sämtliche Code Points.

Der Standard definiert zudem verschiedene Character Encoding Schemes. Ihr Name dient zur Bestimmung des Character Sets, z. B. in einem XML-Dokument. *UTF-8* definiert die Identifikatoren als Folgen von mindestens einem Byte, so dass ASCII-Zeichen die gleichen Bytefolgen und -werte haben wie bei einem ASCII-Dokument.

UTF-16 definiert Folgen von zwei Byte, das bisher selten verwendete UTF-32 Folgen von vier Byte. Enthält ein Text nur ASCII-Zeichen, braucht er mit UTF-8 am wenigsten Speicher. Sind viele über ASCII hinausgehende Zeichen im Text, empfiehlt sich UTF-16. UTF-7 verwendet Folgen von sieben Bits, das achte Bit wird z. B. von Mailservern zu Kontrollzwecken benutzt.

Zeichenkonvertierungen

XML-Dokumente können nicht nur UNICODE-Zeichen enthalten, sondern auch auf andere Character Sets zurückgreifen. Um diese Character Sets zu aktivieren, müssen sie durch eine Verarbeitungsanweisung (engl. "Processing Instruction", vgl. LE 2) im Dokument wiedergegeben werden. Die Angabe geschieht nach folgendem Muster:

Beispiel 1-12. Verarbeitungsanweisungen für Character Sets

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

Diese alternativ zu verwendenden Beispiele definieren zwei japanische Character Sets beziehungsweise UTF-8. Das Attribut encoding kann einen Wert aus der angesprochenen Liste offizieller Namen für Character Sets annehmen. Die Darstellung im Browser sieht so aus:

Beispiel 1-13. Unterschiedliche Character Sets im Browser

Alle drei Dokumente enthalten die gleichen Daten, einen japanischen Satz. Man sieht, dass die Darstellung in jedem Dokument gleich ist, da der Browser auf die selben Fonts zurückgreift. Wenn ein anderes Character Set verwendet wird, werden die Identifikatoren vor der Weiterverarbeitung in das UNICODE-Schema konvertiert. Wenn die Zeichen nicht als numerische Identifikatoren im Dokument eingegeben wurden, sondern als 'Text', kann die Konvertierung fehlerhaft verlaufen, in Abhängigkeit von dem Computersystem (Macintosh, Microsoft etc.) und den entsprechenden Konvertierungstabellen. Verschiedene Konvertierungstools wie Recode⁸ ermöglichen eine Konvertierung der Daten zwischen den meisten gebräuchlichen Character Sets.

Bei einigen Zeichen weicht UNICODE von dem Prinzip 'ein Zeichen, eine Nummer' ab. Dies betrifft z. B. Kombinationen verschiedener Zeichen mittels *Diakritika*. Ein A-Umlaut kann als 'A + zwei Punkte' definiert werden oder als einzelnes Zeichen. Für Such- oder Sortieralgorithmen sind solche Varianten ein großes Problem, da die Eindeutigkeit der Zeichenidentifikation verloren geht. Deshalb hat das W3C ein *Zeichenmodell*⁹ entwickelt, das u. a. die Überführung der Zeichen in eine sogenannte normalisierte Form ermöglicht. *Normalisierung*¹⁰ umfasst z. B. die Umwandlung von kombinierten Zeichen in einzelne Zeichen, wenn dies wie im Falle des A-Umlaut möglich ist. Frei erhältliche Tools wie Charlint¹² erlauben eine automatische Normalisierung.

Ausblick

Fußnoten

1. <http://xml.apache.org/xerces2-j/index.html>
2. <http://www.saxproject.org/apidoc/overview-summary.html>
3. <http://www.w3.org/DOM/>
4. <http://xml.coverpages.org/dom.html>
5. <http://www.unicode.org/charts/>
6. <http://www.iana.org/assignments/character-sets>
7. <http://www.unicode.org/charts/>
8. <http://www.iro.umontreal.ca/contrib/recode/HTML/>
9. <http://www.w3.org/TR/charmod>
10. Die genauen Eigenschaften der Normalisierung sind beschrieben im
 11. <http://www.unicode.org/unicode/reports/tr15/>
des UNICODE-Konsortiums.
 11. <http://www.unicode.org/unicode/reports/tr15/>
 12. <http://www.w3.org/International/charlint/>

