Kapitel 2
XML – Bausteine und Regeln

Die Regeln, nach denen XML-Dokumente gebildet werden, sind einfach, aber streng. Die eine Gruppe von Regeln sorgt für die Wohlgeformtheit, die andere für die Gültigkeit eines Dokuments.

Die Basis der Sprachfamilie XML ist der XML-Standard 1.0. Sie finden den Text unter *www.w3.org/TR/xml*. Der launige Kommentar von Tim Bray, selbst Mitglied der W3C XML Working Group, ist unter *www.xml.com/axml/axml.html* zu finden und immer noch einen Blick wert. Wir wollen hier zunächst einen kurzen Überblick über die Syntax geben, verknüpft mit einem einfachen Beispiel, und dann schrittweise die einzelnen Aspekte näher beleuchten.

2.1 Aufbau eines XML-Dokuments

Laut der Empfehlung des W3C beschreibt XML eine Klasse von Datenobjekten, die *XML-Dokumente* genannt werden. Das entscheidende Kriterium, ob ein Dokument als XML-Dokument genutzt werden kann, ist, dass es im Sinne des Standards *wohlgeformt* ist. Um wohlgeformt zu sein, muss es die syntaktischen Regeln der XML-Grammatik erfüllen, die in den folgenden Abschnitten beschrieben wird.

Entspricht ein solches Dokument außerdem weiteren Einschränkungen, die in Form eines Dokumentschemas in der einen oder anderen Weise festgelegt sind, wird es als *gültig* bezeichnet. Dabei ist entscheidend, dass sowohl die Wohlgeformtheit als auch die Gültigkeit maschinell geprüft werden können.

2.1.1 Entitäten und Informationseinheiten

Der vage Begriff *Datenobjekt* bezieht sich darauf, dass ein XML-Dokument nicht unbedingt eine Datei sein muss, sondern auch ein Teil einer Datenbank oder eines Datenstromes sein kann, der im Netz »fließt«. In einem bestimmten Umfang werden dabei zugleich die Regeln festgelegt, die für den Zugriff von Computerprogrammen auf solche Dokumente gelten.

Die XML-Spezifikation behandelt sowohl die physikalischen als auch die logischen Strukturen eines XML-Dokuments. Physikalisch bestehen XML-Dokumente aus Spei-

chereinheiten. Zunächst ist ein XML-Dokument nichts anderes als eine Kette von Zeichen. Ein XML-Prozessor startet seine Arbeit mit dem ersten Zeichen und arbeitet sich bis zum letzten Zeichen durch. XML liefert dabei Mechanismen, um diese Zeichenkette in verwertbare Stücke zu zerlegen. Diese Textstücke werden *Entitäten* genannt. Auch das Dokument insgesamt wird als Entität bezeichnet, als *Dokumententität*. Im Minimalfall kann eine Entität auch aus nur einem einzigen Zeichen bestehen.

Jede dieser Entitäten enthält Inhalt und ist über einen Namen identifiziert, mit Ausnahme der Dokumententität, die alle anderen Entitäten in sich einschließt. Für einen XML-Parser ist die Dokumententität, der Container für alle anderen Entitäten, immer der Startpunkt. Liegt ein XML-Dokument als Datei vor, ist die Dokumententität eben diese Datei. Wenn Sie dagegen ein XML-Dokument über einen URL einfließen lassen, ist die Dokumententität der Bytestream, den Sie über einen Funktionsaufruf erhalten.

XML erlaubt, Bezüge auf bestimmte Entitäten in das Dokument einzufügen. Solche Entitätsreferenzen werden vom XML-Prozessor durch die Entität, auf die sie sich beziehen, ersetzt, wenn das Dokument eingelesen wird. Deshalb werden solche Entitäten auch *Ersetzungstext* genannt. Das ist ähnlich den Textbausteinen, die von Textprogrammen verwendet werden.

Der Ersetzungstext, den eine aufgelöste Entitätsreferenz liefert, wird als Bestandteil des Dokuments behandelt. Mit Hilfe solcher Referenzen können Entitäten in einem Dokument mehrfach verwendet werden. Durch solche Referenzen kann ein XML-Dokument auch aus Teilen zusammengesetzt werden, die in verschiedenen Dateien oder an unterschiedlichen Plätzen im Web abgelegt sind.

2.1.2 Parsed und unparsed

Eine weitere Unterscheidung ist hier von Bedeutung. Entitäten können laut Spezifikation »parsed or unparsed data« enthalten. *Parsed data* besteht in jedem Fall aus Zeichen. Diese Zeichenfolgen stellen entweder Markups oder Zeichendaten dar. Als Markups sind die Tags, Entitätsreferenzen, Kommentare, die Begrenzer von CDATA-Blöcken, Dokumenttyp-Deklarationen und Verarbeitungsanweisungen zu verstehen – also alles, was mit einer spitzen Klammer oder einem Ampersand-Zeichen beginnt. Die Bezeichnung *parsed* ist leicht irritierend, weil sie erst zutrifft, wenn ein XML-Prozessor das Dokument verarbeitet hat. Gemeint sind also die Teile des Dokuments, die ein XML-Parser auszuwerten hat.

Als *unparsed data* bezeichnet man dagegen Entitäten, die der Parser überhaupt nicht parsen soll und auch nicht kann, weil sie keine Markups enthalten, mit denen der Parser etwas anfangen könnte. Diese Teile müssen nicht unbedingt Text enthalten. Sie werden zum Beispiel verwendet, um Bilder oder sonstige Nicht-Text-Objekte wie Sounds oder Videos in das Dokument einzubeziehen.

2.1.3 Die logische Sicht auf die Daten

Während die physikalische Struktur eines XML-Dokuments durch die Entitäten bestimmt wird, besteht seine logische Struktur aus einem Baum von Informationseinheiten, der seit der erst 2001 nachgereichten Empfehlung *XML Information Set* als *Infoset* bezeichnet wird. (Darin ist auch die Verwendung von Namensräumen aufgenommen, die für XML erst nach der Spezifikation für XML 1.0 eingeführt wurden.) Die Empfehlung definiert insgesamt elf Typen von Informationseinheiten mit jeweils speziellen Eigenschaften:

- Dokument
- Element
- Attribut
- Verarbeitungsanweisung
- nicht expandierte Entitätsreferenz
- Zeichen
- Kommentar
- Dokumenttyp-Deklaration
- ungeparste Entität
- Notation
- Namensraum

Die wichtigsten Komponenten, in die sich der Inhalt des Dokuments teilen lässt, werden in XML als *Elemente* bezeichnet. Der Baum der Elemente hat seine Wurzel im Dokumentelement, das alle anderen Elemente umschließt. Das XML-Dokument besteht also logisch aus Elementen, die jeweils in einer bestimmten Baumstruktur geordnet sind. Welche Bedeutung die Elemente jeweils haben, beschreibt das Dokument selbst durch seine Tags. Neben den Elementen enthält das Dokument noch Deklarationen, Kommentare, Zeichenreferenzen und Verarbeitungsanweisungen.

Die Grammatik von XML legt fest, wie ein wohlgeformtes XML-Dokument erzeugt werden kann. Sie ist in nicht weniger als 81 Produktionsregeln fixiert. Der harte Kern dessen, was XML ausmacht, findet sich aber konzentriert in den folgenden sechs Regeln, die wir hier zunächst in der Schreibweise der Empfehlung wiedergeben.

```
[1]  document  ::= prolog element Misc*
[39] element   ::= EmptyElemTag
        | STag content ETag
        [
          WFC: Element Type Match ]
        [
          VC: Element Valid ]
```

```
[40] STag      ::= '<' Name (S Attribute)* S? '>'
       WFC: Unique Att Spec ]
[41] Attribute ::= Name Eq AttValue
       VC: Attribute Value Type ]
       WFC: No External Entity References ]
       WFC: No < in Attribute Values ]
[42] ETag      ::= '</' Name S? '>'
[43] content   ::= (element | CharData | Reference | CDSect |
                   PI | Comment)*
```

Diese Regeln, die auch *Produktionen* genannt werden, sind in einer einfachen Extended-Backus-Naur-Form notiert, einer Erweiterung der von Backus und Naur für die Beschreibung von Grammatiken zuerst bei der Niederschrift von Algol 60 verwendeten Notation. Jede Regel in einer solchen Grammatik definiert jeweils ein Symbol in der Form:

```
symbol ::= ausdruck
```

Zusätzlich werden in bestimmten Fällen Einschränkungen in eckigen Klammern angehängt, und zwar entweder mit der Abkürzung `WFC`: für *well-formedness constraint* – also Einschränkungen, die beachtet werden müssen, damit das Dokument von einem Parser als wohlgeformt akzeptiert wird, oder `VC`: für *validity constraint*, also Einschränkungen, die die Gültigkeit des Dokuments betreffen.

Was besagen diese Regeln für den Aufbau eines XML-Dokuments? Zunächst ist festgelegt, dass jedes wohlgeformte XML-Dokument einen Prolog haben kann und aus mindestens einem Element bestehen muss. Der Inhalt eines XML-Dokuments wird also aus logischer Sicht in Elemente zerlegt. Im Anschluss daran sind noch Kommentare oder Verarbeitungsanweisungen erlaubt, was aber in der Praxis nicht unbedingt zu empfehlen ist.

Abbildung 2.1 zeigt den gesamten Aufbau des XML-Dokuments und die möglichen Bezüge auf externe Komponenten.

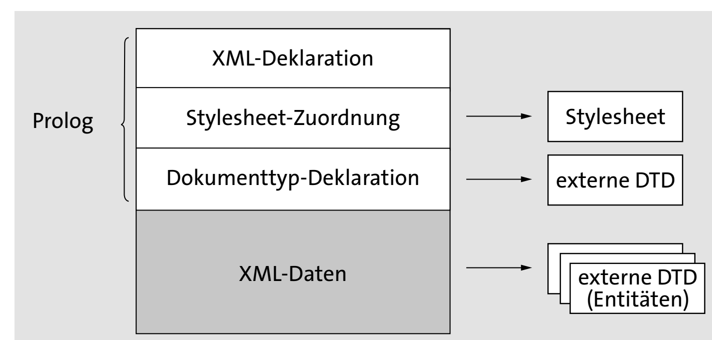

Abbildung 2.1 Aufbauschema eines XML-Dokuments

2.1.4 Der Prolog

Wenn Sie ein XML-Dokument erstellen wollen, beginnen Sie in der Regel mit dem Prolog. Der Prolog ist zwar nicht zwingend vorgeschrieben, aber unbedingt zu empfehlen, weil damit das Dokument sofort als XML-Dokument identifiziert werden kann. Die erste Zeile des Prologs ist meist die sogenannte XML-Deklaration, die zunächst die verwendete XML-Version angibt. In der Minimalform sieht sie so aus:

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

Damit wird die Übereinstimmung des Dokuments mit einer der im Augenblick gültigen Spezifikationen von XML deklariert. Wenn die XML-Deklaration verwendet wird, muss sie in der ersten Zeile des Dokuments stehen, und es dürfen auch keine Leerzeichen davor auftauchen. Das Versionsattribut ist erforderlich.

Neben dem Versionsattribut können in der XML-Deklaration noch zwei weitere Attribute benutzt werden, und zwar `encoding` und `standalone`. Im folgenden Beispiel wird angegeben, dass das Dokument die Zeichencodierung UTF-16 verwendet und dass keine externen Markup-Deklarationen vorhanden sind, auf die für die Verarbeitung des Dokuments zugegriffen werden müsste, also etwa eine externe Dokumenttyp-Definition oder ein XML-Schema.

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
```

2.1.5 Zeichencodierung

Da es bei XML-Dokumenten um Textdaten geht, muss eine Entscheidung getroffen werden, wie Zeichen in Bits und Bytes dargestellt, also codiert werden sollen, und welche Zeichen, also welcher Zeichensatz, in einem bestimmten Dokument maßgeblich sind.

Um XML von vornherein für den internationalen Einsatz zu präparieren, wurde vom W3C, genauso wie auch für die HTML 4.01-Empfehlung, das *Universal Character Set – UCS* – als Basis für die Zeichencodierung in XML-Dokumenten bestimmt, das im Standard *ISO/IEC 10646* festgelegt ist. Dieser Zeichensatz ist, was die verwendeten Zeichencodes betrifft, mit dem Zeichensatz *Unicode* synchronisiert, dem Standard, der vom Unicode-Konsortium – *www.unicode.org* – gepflegt wird. Dieser Standard enthält über ISO 10646 hinaus eine Reihe von Einschränkungen für die Implementierung, die gewährleisten sollen, dass Zeichen unabhängig von Anwendung und Plattform einheitlich verwendet werden. Unicode ist also eine Implementierung der ISO-Norm.

Unicode gibt jedem Zeichen eine eigene Nummer, die unabhängig von Plattformen, Programmen oder Sprachen ist. Text kann mit Unicode weltweit ausgetauscht werden, ohne dass es zu Informationsverlusten kommt.

Zunächst konnte mit einer 16-Bit-Codierung eine Menge von mehr als 65.000 Zeichen abgedeckt werden. Es stellte sich aber schnell heraus, dass diese Menge nicht ausreichen würde, um alle weltweit in Vergangenheit und Gegenwart verwendeten Zeichen zu codieren. Deshalb wurde Unicode um einen sogenannten Ersatzblock erweitert, der über eine Million Zeichen zusätzlich erlaubt. Allerdings sind diese Zeichen keine gültigen XML-Zeichen.

Inzwischen werden drei unterschiedliche Unicode-Codierungen eingesetzt, mit 8, 16 oder 32 Bit pro Zeichen. Sie werden als *UTF-8*, *UTF-16* und *UTF-32* bezeichnet, wobei *UTF* eine Abkürzung für *Unicode* (oder *UCS*) *Transformation Format* ist.

Die `encoding`-Deklaration legt fest, welche Zeichencodierung das Dokument verwendet, damit der XML-Prozessor diese Codierung seinerseits ebenfalls benutzt. Wenn Ihr XML-Editor mit dem ASCII-Code arbeitet, ist diese Angabe nicht unbedingt nötig; der Prozessor wird den Code als Teil des Unicodes UTF-8 auswerten.

XML verwendet UTF-8 als Vorgabe. Diese Codierung wird hauptsächlich für HTML und ähnliche Protokolle verwendet. Dabei werden alle Zeichen in variabel lange Codierungen (1 bis 4 Bytes) umgesetzt. Das hat den Vorteil, dass sich bei den ersten 128 Zeichen der Unicode mit dem 7-Bit-ASCII-Code deckt. Außerdem können einfache Texteditoren so für XML-Dokumente eingesetzt werden.

Die andere Unicode-Codierung, die XML-Prozessoren unterstützen müssen, ist UTF-16. Diese Codierung ist unkomplizierter als UTF-8. Die am häufigsten verwendeten Zeichen werden jeweils mit 16-Bit-Einheiten codiert, alle anderen Zeichen durch Paare von 16-Bit-Codeeinheiten.

UTF-32 hat zwar den Vorteil, dass es fast unendlich viele Zeichen darstellen kann, dieser Vorzug wird aber damit erkauft, dass der Speicherbedarf pro Zeichen doppelt so hoch ist wie bei UTF-16.

Wenn ein anderer Zeichensatz als UTF-8 verwendet werden soll, muss er in der Deklaration angegeben werden. Der Wert für das Attribut `encoding` ist ausnahmsweise nicht fallsensitiv – `UTF-16` ist also ebenso erlaubt wie `utf-16`. (Jede externe Entität kann übrigens eine eigene Zeichencodierung verwenden, wenn eine entsprechende Deklaration angegeben wird.)

Da aber Unicode noch nicht überall verbreitet ist, werden auch andere Codierungen unterstützt. Für Westeuropa kann beispielsweise die verbreitete Codierung *ISO-8859-1 (ISO Latin-1)* verwendet werden.

2.1.6 Standalone or not

Das Attribut `standalone` kann nur einen logischen Wert annehmen. Wird `standalone="no"` verwendet, ist das eine Anweisung für den XML-Prozessor, nach externen Markup-Definitionen Ausschau zu halten, um Referenzen auf externe Entitäten aufzulösen und die Gültigkeit des Dokuments prüfen zu können. Diese Einstellung muss allerdings nicht extra angegeben werden, weil sie Vorgabe ist. Der Wert `"yes"` dagegen bedeutet, dass das Dokument alle Informationen in sich selbst enthält, die für die Verarbeitung benötigt werden.

Beachtet werden muss, dass die Attribute `encoding` und `standalone` zwar optional sind; wenn sie verwendet werden, muss aber die gerade vorgeführte Reihenfolge eingehalten werden, im Unterschied zu »normalen« Elementattributen, bei denen die Reihenfolge keine Bedeutung hat.

Der Prolog kann nach der XML-Deklaration weitere Verarbeitungsanweisungen enthalten, wie zum Beispiel die Verknüpfung mit einem Stylesheet oder eine Dokumenttyp-Deklaration, etwa:

```
<?xml-stylesheet type="text/css" href="formate.css"?>

<!DOCTYPE kontaktdaten SYSTEM "kontakte.dtd">
```

2.1.7 XML-Daten – der Baum der Elemente

Erst hinter dem Prolog beginnen die eigentlichen XML-Daten in Form eines Baums aus Elementen und Attributen. Das erste Element im Dokument ist immer das Wurzelelement, das alle anderen möglichen Elemente in sich einschließt. Mit anderen Worten: Das Dokument hat die Struktur eines Baums aus ineinander verschachtelten Elementen.

Außer für das Wurzelelement gibt es folglich für jedes andere Element genau ein Elternelement, während das Wurzelelement und jedes seiner Kindelemente wieder weitere Kindelemente zum Inhalt haben können. Es sind beliebig tiefe Verschachtelungen erlaubt.

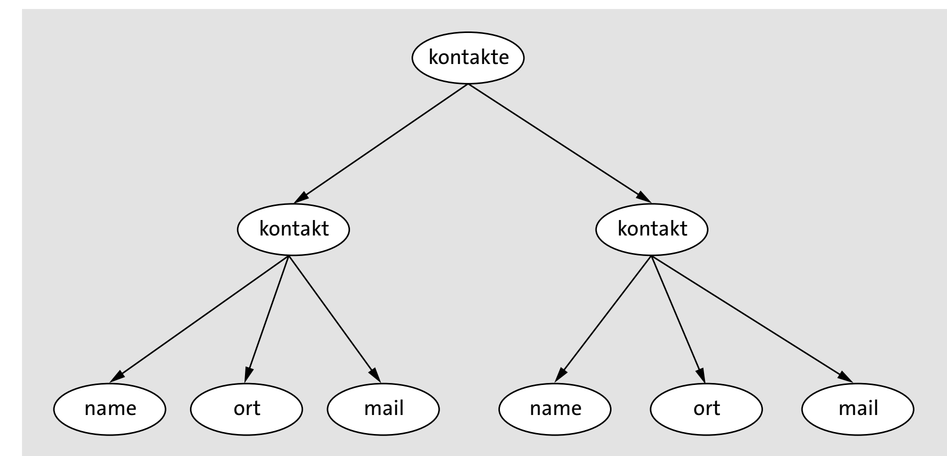

Abbildung 2.2 Baumstruktur eines Dokuments

Diese hierarchische Baumstruktur kann durch einen Graphen dargestellt werden, dessen Knoten durch gerichtete Kanten verbunden sind und dessen Wurzelknoten für keine dieser Kanten der Endknoten ist. Dieser Graph enthält folglich auch keine Zyklen.

Der Baum ist durch die sequenzielle Abfolge der Elemente im XML-Dokument implizit geordnet, die als *Dokumentreihenfolge* bezeichnet wird; darauf wird in Kapitel 5, »Navigation und Verknüpfung«, näher eingegangen. Natürlich kann auch eine ganz flache Struktur wie eine relationale Datenbanktabelle in XML ausgedrückt werden, aber die besonderen Stärken des Modells kommen besonders dann zur Geltung, wenn es um tief gestaffelte Datenstrukturen geht.

2.1.8 Start-Tags und End-Tags

Jedes Element wird jeweils durch ein Start-Tag und ein End-Tag begrenzt. Das XML-Dokument vermischt also die darin enthaltenen Inhalte mit Informationen über diese Inhalte, oder man kann auch sagen, es mischt Informationen und Informationen über diese Informationen. Die Metainformation befindet sich in den Tags, die die Inhalte einschließen. Damit zwischen Inhalt und Markup unterschieden werden kann, werden spezielle Zeichen verwendet, die Beginn und Ende des Markups kennzeichnen und so das Markup vom Inhalt trennen.

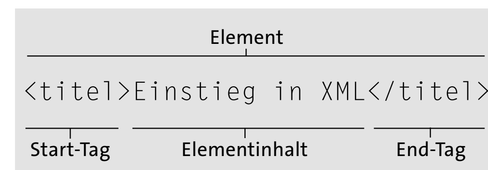

Abbildung 2.3 Ein Element in XML

Vergleicht man eine Gruppe von Datensätzen im CSV-Format oder in einer Datentabelle mit einer Feldnamenzeile mit denselben Datensätzen im XML-Format, wird sofort deutlich, dass das XML-Format zwangsläufig aufwendiger ist. Bei jedem Datensatz werden alle »Feldnamen« erneut angegeben. Dieses Format wäre in den Zeiten, in denen Speicherressourcen noch sehr knapp und die Bandbreiten in den Netzen sehr eng waren, wahrscheinlich wenig attraktiv gewesen.

Der Vorteil von XML ist aber, dass auch jeder einzelne Zweig im Baum der Elemente darüber Auskunft gibt, was die einzelnen Elemente bedeuten. Das erleichtert auch die Zerlegung eines großen XML-Dokuments in kleinere Teile oder umgekehrt das Zusammenfügen von Teildokumenten zu einem Gesamtdokument.

2.1.9 Elementtypen und ihre Namen

Das Start-Tag enthält den Namen des Elementtyps, eingeschlossen in spitze Klammern, beim End-Tag kommt vor den Namen des Elementtyps noch ein Schrägstrich. Als strenge Einschränkung für das Kriterium der Wohlgeformtheit ist festgelegt, dass der Elementtypname im Start-Tag und im End-Tag exakt übereinstimmen muss. Die Wiederholung des Elementtypnamens im End-Tag ist notwendig, damit der Parser die Schachtelung der Elemente sofort korrekt erkennen kann. Nur bei leeren Elementen ist eine vereinfachte Schreibweise erlaubt, bei der auf die Wiederholung des Elementnamens verzichtet wird.

Anders als bei HTML-Tags ist dabei unbedingt auf die Groß- und Kleinschreibung zu achten, weshalb es sinnvoll ist, sich von vornherein für eine einheitliche Schreibweise zu entscheiden, also alle Tags im Ganzen klein bzw. groß zu schreiben oder aber der Groß- und Kleinschreibung den Vorzug zu geben.

Ein Verstoß gegen die Regel der Wohlgeformtheit führt zu einem fatalen Fehler, das heißt, ein XML-Prozessor wird die Verarbeitung des nicht wohlgeformten Dokuments ab der Stelle, an der der Fehler auftritt, verweigern. Diese harsche Reaktion unterscheidet XML wiederum stark von HTML, das für seine eher tolerante Reaktion auf viele Fehler bekannt ist, die dafür sorgt, dass der Webbesucher auch bei Seiten mit kleinen Fehlern nicht leer ausgeht. Diese »Nachgiebigkeit« der Browser bei der Verarbeitung von »unsauberem« HTML-Code sollte für XML mit Vorsatz nicht wiederholt werden, weil diese Situation letztlich dazu geführt hat, dass viel Wildwuchs auf Webseiten zugelassen wurde. Der aber muss von den Browsern mit immer mehr Ausnahmeregelungen aufgefangen werden, was den Code enorm aufbläht.

Zwischen dem Start- und dem End-Tag befindet sich der Inhalt des Elements. Dass die Tags nicht einfach die Namen der Elemente enthalten, sondern die Namen der Elementtypen, weist schon darauf hin, dass Elemente desselben Typs in einem Dokument mehrfach verwendet werden können. Jedes einzelne Element ist also ein Exemplar oder eine Instanz eines bestimmten Elementtyps.

```
<team>
  <person>Hanna Karl</person>
  <person>Kurt Vondel</person>
</team>
```

Im Unterschied zu HTML und auch zu SGML darf das End-Tag nicht fehlen, sonst kann das Dokument eine Prüfung auf Wohlgeformtheit nicht bestehen. Nur bei einem leeren Element ist es erlaubt, eine verkürzte Schreibweise zu verwenden, also `<leer />` statt `<leer></leer>`. Leere Elemente werden zum Beispiel für das Einbinden von Bildern in ein XML-Dokument verwendet.

2.1.10 Regeln für die Namensgebung

Die in den Tags verwendeten Namen für die Typen der Elemente sind frei wählbar, solange nur die Wohlgeformtheit des gesamten XML-Dokuments interessiert. Ein Dokument kann also auch beliebig viele Elementtypen enthalten. Allerdings müssen bei der Wahl des Namens einige Einschränkungen beachtet werden:

- Ein Name muss mit einem Buchstaben, mit Unterstrich oder einem Doppelpunkt beginnen.
- Danach dürfen alle Zeichen verwendet werden, die als Namenszeichen zugelassen sind: Neben den Zeichen für die erste Stelle sind das die Zahlen, der Bindestrich und der Punkt. Auch Umlaute, Akzente etc. sind erlaubt. Allerdings sollte der Doppelpunkt möglichst vermieden werden, weil er als Trennzeichen verwendet wird, wenn mit Namensräumen gearbeitet wird, wovon in Abschnitt 2.9, »Namensräume«, noch die Rede sein wird.
- Die Zeichenfolge `xml` darf in keiner der möglichen Schreibweisen am Beginn eines Namens stehen; diese Zeichenfolge ist für XML reserviert.
- XML-Namen sind *case-sensitive*, es wird also zwischen Groß- und Kleinschreibung unterschieden, so dass `<Name>...</name>` beispielsweise nicht zulässig ist.

Die Länge der Namen ist nicht begrenzt, es sollte aber beachtet werden, dass möglicherweise Anwendungen, die auf die Daten zugreifen, die Länge einschränken. Wenn Sie Namen aus mehreren Wörtern zusammensetzen, haben Sie die Möglichkeit, die einzelnen Wörter mit Bindestrichen oder Unterstrichen zu verbinden. Eine häufig genutzte Variante ist auch die Verwendung von Großbuchstaben am Wortbeginn, entweder bei jedem Wort wie in `<VorName>` – das wird auch *Pascal-casing* genannt, oder erst beim zweiten Wort wie in `<vorName>` – also im *Camel-casing*-Verfahren.

Es ist immer wieder die Rede davon, dass XML Tags erlaubt, die den Inhalt der eingeschlossenen Daten beschreiben, also semantische Tags wie `<Postleitzahl>` oder `<Titel>`. Der Standard legt aber nur fest, dass ganz beliebige Elementtypen bestimmt werden können; `<tag1></tag1>`, `<tag2></tag2>` würde die Wohlgeformtheitsprüfung ebenfalls überstehen. Es kommt also hier darauf an, was die Entwickler mit der durch die Spezifikation angebotenen Freiheit anfangen.

Das Ziel, das der Entwicklung von XML die Richtung gibt, ist jedenfalls eine Identifizierung der Komponenten, aus denen eine Datensammlung oder ein Dokument besteht, mit Hilfe von bedeutungsvollen – also semantischen – Namen, die ein Computerprogramm zwar nicht wie ein Mensch versteht, die dem Programm aber erlauben, sich so zu verhalten, als ob es verstehen würde, was die verwendeten Namen bedeuten. Die XML-Tags haben insofern durchaus Ähnlichkeiten mit den Feldnamen, die bei der Strukturierung von Datenbanken verwendet werden.

Die Tags gehören zu den Auszeichnungen, den Markups, die die Struktur des Dokuments festlegen und im Idealfall zugleich den Inhalt des Dokuments beschreiben. Da XML-Dokumente ein schlichtes Textformat verwenden, bleiben sie für den menschlichen Betrachter im Prinzip lesbar.

2.1.11 Elementinhalt

Abgesehen von den schon angesprochenen leeren Elementen haben Elemente in der Regel einen Inhalt, der aus ganz unterschiedlichen Dingen bestehen kann. In diesem Sinne werden die Elemente auch als *Container* betrachtet. Zunächst kann ein Element wiederum untergeordnete Elemente enthalten. Man spricht dann von *element content*. Das folgende Element `<kontakt>` hat zum Beispiel drei Kindelemente zum Inhalt:

```
<kontakt>
  <name>Hans Maier</name>
  <ort>Hamburg</ort>
  <mail>hmaier@nonet.de</mail>
</kontakt>
```

In diesem Fall beginnt mit dem Element `<kontakt>` also ein Teilbaum innerhalb der gesamten Baumstruktur. Die Art der Anordnung der Kindelemente kann über ein Inhaltsmodell geregelt werden, entweder per DTD oder per XML Schema. Dieses Modell legt beispielsweise fest, dass die Elemente unbedingt in einer bestimmten Reihenfolge auftreten müssen.

Den Blättern des Baums entsprechen dagegen die Elemente, die dann nur noch Zeichendaten enthalten, also Zeicheninhalt oder *character content*. XML lässt auch zu, Elemente und Zeichendaten zu mischen, gemischte Inhalte oder *mixed content* also. Im Unterschied zu dem ersten Fall, bei dem Elemente selbst wiederum nur Kindelemente enthalten, bleibt bei einer solchen Mischung von Zeichendaten und Elementen die Reihenfolge weitgehend ungeregelt – ein Grund, solche Mischcontainer möglichst zu vermeiden.

2.1.12 Korrekte Schachtelung

So schlicht die Kernfestlegungen auf den ersten Blick erscheinen, können doch schon bei den ersten Schritten zur Strukturierung eines Gegenstandsbereichs Probleme auftreten. Eine erste Falle ist eine falsche Schachtelung von Elementen. HTML reagiert auf eine falsche Schachtelung von Tags relativ harmlos, wie Abbildung 2.4 zeigt.

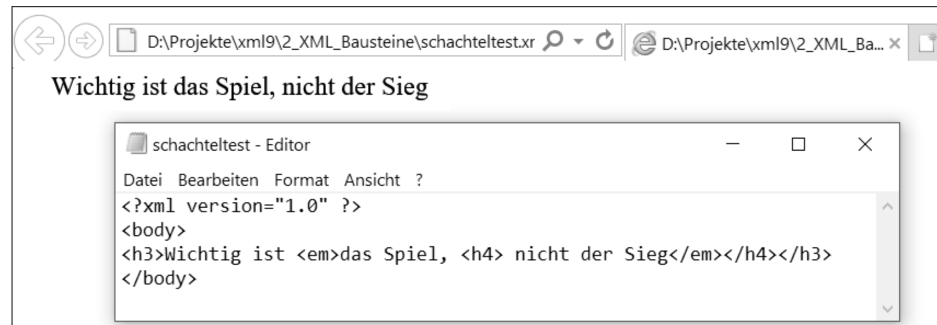

Abbildung 2.4 Der Internet Explorer verdaut die fehlerhaft geschachtelten HTML-Elemente.

Wenn Sie dieselben Tags als XML-Dokument speichern, ist die Reaktion eines XML-Prozessors wiederum sehr kategorisch. Er wird einen fatalen Fehler melden und das Prädikat der Wohlgeformtheit verweigern.

Schachtelungsfehler können sich auch dadurch einschleichen, dass einfach das End-Tag für ein übergeordnetes Element vergessen wird.

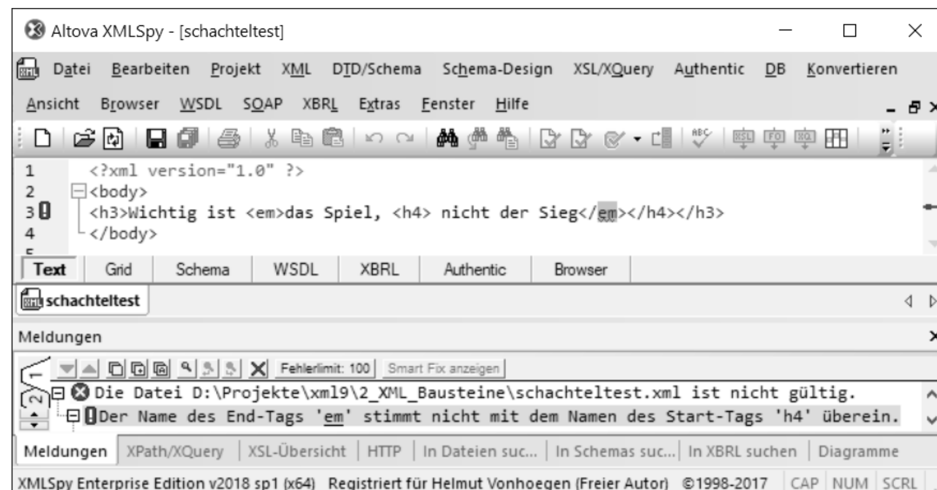

Abbildung 2.5 Reaktion des XMLSpy-Editors auf eine fehlerhafte Schachtelung

2.1.13 Attribute

Alle Elemente, die innerhalb der Struktur auftauchen, können mit beliebig vielen Attributen versehen werden, die jeweils als Paare von Attributnamen und Attributwerten auftreten. Attribute werden verwendet, um bestimmte Eigenschaften oder Besonderheiten eines Elements festzuhalten. Manchmal werden deshalb auch die Attribute mit den Adjektiven verglichen, die Elemente mit den Subjekten. Allerdings

ist dieser Vergleich nicht streng anwendbar, denn Attribute können auch ähnlich wie Unterelemente verwendet werden.

Die Attribute werden jeweils im Start-Tag eines Elements platziert. Sie müssen eindeutig sein, es darf also nicht mehrfach derselbe Attributname in einem einzigen Start-Tag verwendet werden. Ein Ausdruck wie

```
<haus nr="22" nr="33">
```

ist also nicht zulässig – er ergibt ja auch wenig Sinn.

Dagegen darf durchaus derselbe Attributname bei unterschiedlichen Elementen verwendet werden. Obwohl die Attribute denselben Namen verwenden, bleiben sie für einen XML-Prozessor unterscheidbar, weil sie ja zu unterschiedlichen Elementen gehören.

```
<einrichtung>
  <tisch farbe="blau">Esstisch rund</tisch>
  <stuhl farbe="beige">Vierbeinstuhl</stuhl>
</einrichtung>
```

Werden mehrere Paare von Attributnamen und Attributwerten verwendet, werden sie durch ein Leerzeichen getrennt. Für die Attributnamen gelten dieselben Vorschriften wie für die Elementnamen. Während Elemente andere Elemente enthalten können, ist dies bei Attributen nicht erlaubt. Die Attributwerte wiederum müssen jeweils mit den oberen doppelten Anführungszeichen oder mit dem Apostroph eingeschlossen werden. Die Werte selbst dürfen nur Literale sein. Innerhalb des Literals dürfen die Markup-Zeichen `<`, `>` und `&` nicht verwendet werden. Sie sind durch `<`, `>` und `&`, also durch sogenannte Entitätsreferenzen, zu maskieren. Mehr Details dazu finden Sie in Abschnitt 2.5.

Wenn Sie eine Art von Begrenzungszeichen verwenden, um einen Attributwert einzuschließen, können Sie die anderen Begrenzungszeichen innerhalb des Literals verwenden:

```
<antwort text="Er sagte: ´So geht es nicht´">
```

Auch leere Elemente können mit Attributen versehen werden. Ein Ausdruck wie

```
<leer name="leeres Element"/>
```

ist also erlaubt. Solche leeren Elemente werden zum Beispiel verwendet, um im Dokument Stellen für Daten in einem Nicht-XML-Format zu reservieren, etwa für Bilder, Videos oder Sounds. Dabei werden die Informationen zu diesen Datenquellen über Attribute des leeren Elements festgehalten.

2.2 Die Regeln der Wohlgeformtheit

Bei der Prüfung von XML-Dokumenten wird grundsätzlich zwischen der Prüfung der Wohlgeformtheit und der Gültigkeit unterschieden. Auch wenn ein XML-Dokument mit einer DTD oder einem XML-Schema verknüpft ist, kann es ein XML-Prozessor dabei bewenden lassen, nur die Wohlgeformtheit zu überprüfen.

Wenn auf eine Gültigkeitsprüfung verzichtet wird, besteht allerdings keinerlei Gewähr, dass Daten zum Beispiel in einem bestimmten Datenformat vorliegen. Werden solche Daten von Anwendungsprogrammen weiterverarbeitet, müssen diese Programme folglich selbst dafür sorgen, dass die ungeprüft übernommenen Daten korrekt verarbeitet werden. Das muss aber, beispielsweise bei Lösungen innerhalb eines Intranets, kein Problem sein.

Dass mit Wohlgeformtheit allein nicht unbedingt viel erreicht ist, wird schlagartig durch die folgenden Zeilen deutlich:

```
<tierprodukte>
  <milchprodukt>Käse</milchprodukt>
  <getreidesorte>Weizen</getreidesorte>
</tierprodukte>
```

Hier ist zwar kein syntaktischer Fehler zu finden, aber auf der Ebene der Bedeutungen ist gleich offensichtlich, dass etwas nicht stimmen kann.

2.3 Elemente oder Attribute?

In der Regel sollten Attribute verwendet werden, um Zusatzinformationen zu der Information darzustellen, die das Element selbst enthält. Es liegt aber nicht immer auf der Hand, welche Aspekte eines Dokuments oder einer Datenstruktur besser als Element oder besser als Attribut behandelt werden sollten. Ein Ausdruck wie

```
<an name="Clara Donna" email="cd@clarad.de"/>
```

ist ebenso akzeptabel wie

```
<an>
  <name>Clara Donna</name>
  <email>cd@clarad.de </email>
</an>
```

Prinzipiell sind Attribute immer einem Element zugeordnet; anstelle eines Attributs kann aber häufig auch ein Unterelement verwendet werden. Im Unterschied zu Elementen können Attribute keine Unterelemente oder Unterattribute enthalten. Ein

Nachteil von Attributen ist auch, dass der Zugriff auf die Attributwerte über Anwendungen, etwa mit Hilfe der Schnittstellen des in Kapitel 10, »Programmierschnittstellen für XML«, vorgestellten *Document Object Models* (*DOM*), umständlicher ist als der auf den Inhalt von Elementen. Auch wenn Sie die Daten mit Cascading Stylesheets anzeigen wollen, sind Elemente vorzuziehen.

Ein gewisser Vorteil von Attributen dagegen ist, dass ihre Reihenfolge nicht wie bei Elementen fixiert werden muss.

Wenn es um tatsächlich getrennte oder begrifflich sinnvoll auftrennbare Einheiten in einem Gegenstandsbereich geht, sollte in der Regel mit Elementen gearbeitet werden, zumal dadurch auch der Aufbau von Links auf diese Einheiten ermöglicht wird, etwa durch ID- und IDREF-Attribute in einem XSLT-Stylesheet.

2.4 Reservierte Attribute

Die XML-Spezifikation gibt zwei Attribute vor, die in jedem Element verwendet werden können. Das eine – `xml:lang` – dient zur Identifikation der Sprache, die für die Inhalte eines Dokuments oder einzelner Elemente gilt, das andere Attribut – `xml:space` – erlaubt Festlegungen zur Behandlung von Leerraum im Inhalt eines Elements. Obwohl die Attribute vorgegeben sind, müssen sie innerhalb einer DTD explizit deklariert werden, wenn sie zum Einsatz kommen sollen.

2.4.1 Sprachidentifikation

In den folgenden Elementen

```
<zeile xml:lang="en">to be or not to be</zeile>
<zeile xml:lang="de">Sein oder Nichtsein</zeile>
```

wird durch das `xml:lang`-Attribut erkennbar, welche Sprache für den Inhalt des Elements verwendet wird. Auf diese Weise kann zum Beispiel ein Stylesheet entwickelt werden, das dem Wunsch eines Anwenders entsprechend die Zeile einmal in der einen, einmal in der anderen Sprache ausgibt. Die Einstellung gilt jeweils auch für die Unterelemente.

Als Werte des Attributs werden zwei- oder dreistellige Ländercodes verwendet, die durch *ISO 639* genormt sind. Auch Subcodes für regionale Sprachen sind möglich, etwa "`en-US`" oder "`en-GB`".

2.4.2 Leerraumbehandlung

Mit dem Attribut `xml:space` können Sie versuchen, Einfluss darauf zu nehmen, wie ein XML-Prozessor mit Leerräumen im Inhalt von Elementen umgeht. Leerräume sind Leerzeichen, Tabulatoren und Leerzeilen. Auch diese Einstellung gilt jeweils für das Element und die Unterelemente. Es gibt zwei mögliche Werte:

- `preserve` meldet der Anwendung den Wunsch, dass alle Leerräume, so wie sie vorhanden sind, erhalten bleiben.
- `default` stellt der Anwendung, die die Daten verarbeitet, anheim, deren Vorgabe für den Umgang mit Leerräumen zu verwenden.

Der Inhalt des folgenden Elements sollte also in einer Anwendung genauso erscheinen wie eingegeben:

```
<zeile xml:space="preserve">
T o o o r !
</zeile>
```

Es hängt allerdings von der Anwendung ab, ob sie dem Hinweis folgt oder die Leerzeichen einfach entfernt.

2.5 Entitäten und Verweise darauf

Es ist schon angesprochen worden, dass die speziellen Zeichen, die XML für Markups verwendet, nicht ohne Weiteres im Inhalt eines Elements oder innerhalb eines Attributwerts auftauchen dürfen. Es gibt nun mehrere Möglichkeiten, Zeichen davor zu schützen, von einem XML-Prozessor als Markup-Zeichen ausgewertet zu werden, wenn sie als Inhalt eines Elements oder innerhalb eines Attributwerts benötigt werden.

2.5.1 Eingebaute und eigene Entitäten

Der eine Weg ist die Verwendung von Entitäten und Referenzen auf diese Entitäten. XML bringt fünf solcher Entitäten schon mit (siehe Tabelle 2.1).

Entität	Entitätsreferenz	Bedeutung
`lt`	`<`	< (kleiner als)
`gt`	`>`	> (größer als)
`amp`	`&`	& (Ampersand)

Tabelle 2.1 Entitäten von XML

Entität	Entitätsreferenz	Bedeutung
`apos`	`'`	' (Apostroph oder einfache Anführungszeichen)
`quot`	`"`	" (doppelte Anführungszeichen)

Tabelle 2.1 Entitäten von XML (Forts.)

Ein Firmenname wie »B&B« kann auf diese Weise als `B&B` eingegeben werden.

Zusätzlich zu den eingebauten Entitäten können Sie eigene Entitäten definieren, ähnlich wie es auch in HTML möglich ist. Dies geschieht innerhalb einer Dokumenttyp-Definition. Wie dies gemacht wird, ist in Abschnitt 3.9, »Verwendung von Entitäten«, beschrieben.

Um solche Entitäten zu verwenden, wird dieselbe Schreibweise verwendet wie bei den eingebauten Entitäten. Wird beispielsweise in der DTD ein Kürzel `GB` für Geschäftsbedingungen abgelegt, kann der Ersetzungstext mit `&GB;` referenziert werden.

2.5.2 Zeichenentitäten

Eine weitere Methode, Zeichen indirekt einzubringen – etwa wenn das Eingabegerät bestimmte Zeichen nicht zur Verfügung stellt –, sind Zeichenreferenzen, die mit Hilfe von dezimalen oder hexadezimalen Zahlen arbeiten, die auf den Unicode-Zeichensatz verweisen. Die Schreibweise ist ähnlich wie bei den Entitätsreferenzen, nur wird dem `&`-Zeichen noch ein `#`-Zeichen nachgestellt.

```
&#169;
&#xA9;
```

liefern beide das Copyright-Zeichen.

```
&#x20AC;
```

oder

```
&#8364;
```

liefern das Euro-Symbol.

Werden Entitätsreferenzen verwendet, muss das Dokument nach der Auflösung der Referenzen – also nachdem das Kürzel gegen den Ersetzungstext getauscht worden ist – weiterhin ein wohlgeformtes Dokument sein.

Solche Entitätsreferenzen sind daher nur erlaubt, wenn sie Bezüge auf *parsed data* enthalten; direkte Bezüge auf *unparsed data* sind nicht erlaubt. Solche Bezüge müs-

sen auf einem Umweg über Attributwerte vom Typ `ENTITY` oder `ENTITIES` hergestellt werden. Mehr dazu finden Sie in Abschnitt 3.9.

2.6 CDATA-Sections

Wenn es sich ergibt, dass größere inhaltliche Teile eines XML-Dokuments sehr häufig reservierte Markup-Zeichen benötigen, etwa ein Dokument, das selbst wiederum ein anderes XML- oder HTML-Dokument beschreibt oder Skriptcode enthält, kann es mühsam werden, jedes Mal mit Zeichen- oder Entitätsreferenzen zu arbeiten. In diesem Fall ist es praktischer, CDATA-Blöcke zu verwenden. Hier ein kleines Beispiel:

```
<![CDATA[
  Tags in XML werden immer mit < und > begrenzt.
]]>
```

Ein solcher Block von *Character Data* beginnt mit dem String `<![CDATA[` und endet mit dem sogenannten CDEnd-String `]]>`. Alle Zeichen, die sich dazwischen befinden, werden von einem XML-Prozessor nicht als Markup interpretiert. Sie können also ungehindert die spitzen Klammern oder das Ampersand (&) verwenden. Nur die Zeichenfolge `]]>` selbst darf nicht innerhalb des Textes der CDATA-Section vorkommen.

2.7 Kommentare

Über die Nützlichkeit von Kommentaren muss man Entwicklern keine Vorträge halten, obwohl manchmal eher zu wenige als zu viele verwendet werden. Das gilt auch für XML, obwohl die »sprechenden« Tags das Dokument schon in einem bestimmten Umfang selbst dokumentieren.

Zur eigenen Erinnerung oder zur Information für andere lassen sich Kommentare überall in das XML-Dokument einbetten, sie sind nur nicht innerhalb von Tags oder Deklarationen erlaubt, im Unterschied übrigens zu SGML:

```
<!--
das Dokument muss noch mit einem Schema verknüpft werden
-->
```

Wie in HTML beginnt ein Kommentar mit `<!` und endet mit `>`. Die Zeichenfolge darf nicht innerhalb des Kommentars verwendet werden. Deshalb sind auch Kommentare innerhalb von Kommentaren verboten. Wenn Sie bei einem Test ein Element oder einen Teilbaum von Elementen vorübergehend herausnehmen wollen, können Sie die gesamte Gruppe auskommentieren:


```
<!--
<element>
  <unterelement>
  </unterelement>
</element>
-->
```

Streng beachtet werden muss, dass keine Kommentare vor der XML-Deklaration erscheinen dürfen, wenn diese verwendet wird. Dokumente ohne XML-Deklaration dürfen dagegen mit einem Kommentar beginnen!

2.8 Verarbeitungsanweisungen

Es ist möglich, in das XML-Dokument Verarbeitungsanweisungen – *processing instructions* – für den XML-Prozessor einzubetten. Ihr Inhalt gehört nicht zu den Zeichendaten, aus denen das Dokument besteht.

Die Anweisungen beginnen immer mit `<?` und enden mit `?>`. Sie geben zunächst ein Ziel an, das die Anwendung identifizieren soll, an die sich die Anweisung richtet, dann folgen die Daten der Anweisung selbst.

Welche Anweisungen möglich sind, hängt davon ab, was der verwendete Prozessor versteht und was nicht. Sie sind also nicht durch die XML-Spezifikation vorgegeben. Zweck einer solchen Anweisung kann zum Beispiel die Verknüpfung des Dokuments mit einem Stylesheet sein:

```
<?xml-stylesheet type="text/css" href="praesentation.css"
media="screen"?>
```

Da diese Instruktion in der ursprünglichen Spezifikation von XML noch nicht vorgesehen ist – das Wort »Stylesheet« taucht darin nicht auf –, wurde dafür im Sommer 1999 extra eine kleine Empfehlung namens *Associating Style Sheets with XML documents* über die Zuordnung von Stylesheets zu XML-Dokumenten herausgegeben, in der die `xml-stylesheet`-Anweisung definiert ist.

XML-Prozessoren, wie sie zum Beispiel im Internet Explorer ab Version 5 integriert sind, verstehen eine solche Anweisung und geben ein XML-Dokument in dem zugewiesenen Format aus. Mehr dazu finden Sie in Kapitel 10, »Programmierschnittstellen für XML«.

2.9 Namensräume

Eine der ersten wichtigen Erweiterungen des XML-Standards war die Empfehlung *Namespaces in XML*, die bereits ein Jahr nach der Verabschiedung von XML 1.0 im Februar 1998 veröffentlicht wurde. Die Freiheit, die XML bei der Wahl von Element- und Attributnamen gewährt, wirft überall dort ein Problem auf, wo gleiche Namen mit unterschiedlichen Bedeutungen verwendet werden. Da XML-Dokumente häufig darauf ausgelegt sind, dass sie mit unterschiedlichen Programmen ausgewertet und zu bearbeitet werden, führen Mehrdeutigkeiten schnell zu unerwünschten Ergebnissen.

2.9.1 Das Problem der Mehrdeutigkeit

Ein einfaches Beispiel ist etwa ein Elementname wie `<beitrag>`, der in dem einen Kontext den Mitgliedsbeitrag in einem Verein benennt, in einem anderen Kontext einen Artikel in einer Zeitschrift und in einem dritten Kontext eine Arbeitsleistung, etwa innerhalb eines Projekts. Während bei dem ersten Element ein Programm beispielsweise prüfen soll, ob es Beitragsrückstände gibt, könnte im letzten Fall eine Berechnung der Arbeitszeit angestoßen werden.

Zwar wäre es jedes Mal möglich, den Namen entsprechend zu spezifizieren und die entsprechenden Elemente als `<mitgliedsbeitrag>`, `<textbeitrag>` und `<projektbeitrag>` zu differenzieren, aber erstens führt dies in vielen Fällen zu eher künstlichen Bezeichnungen, und zweitens kann man niemals sicher sein, dass beim Design eines XML-Vokabulars diese Regel immer beachtet wird.

2.9.2 Eindeutigkeit durch URIs

Da XML-Vokabulare aber auch unter dem Gesichtspunkt entwickelt werden, möglichst wiederverwendbar zu sein, lag es nahe, dafür eine andere Lösung zu suchen. Deshalb hat das W3C ein relativ einfaches Verfahren eingeführt, um die Eindeutigkeit von Namen für Elemente und Attribute zu gewährleisten. Dabei werden die einzelnen Namen als Teile eines bestimmten Namensraums behandelt. Namensräume werden als Ansammlungen von Namen vorgestellt, die zu einem bestimmten Gegenstandsbereich gehören. Eine solche Ansammlung benötigt keine bestimmte Struktur, wie es etwa bei einer DTD der Fall ist. Es reicht eine einfache Zugehörigkeit: Der Name `a` gehört zum Namensraum `x`.

Mit der Angabe eines Namensraums wird der Kontext angegeben, in dem ein bestimmter Name seine ganz spezielle Bedeutung erhält. XML-Namensräume werden über eine URI-Referenz, also in der Regel einen URL, identifiziert. (Dabei werden solche Referenzen nur als identisch betrachtet, wenn sie Zeichen für Zeichen gleich sind, also unter Beachtung der Groß- und Kleinschreibung.) Diese URI-Referenz wird

aber nur verwendet, um dafür zu sorgen, dass der Namensraum auch eindeutig ist. Man nutzt also die Tatsache, dass URI-Referenzen immer eindeutig sein müssen, um dem Prozessor einen eindeutigen Namen für den Namensraum zu liefern.

Abbildung 2.6 Gleichlautender Name in verschiedenen Namensräumen

Der Prozessor sieht keineswegs bei dem entsprechenden URL nach, ob dort ein Namensraum abgelegt ist. (Das W3C hinterlegt unter den URLs der von ihm selbst gepflegten Namensräume lediglich Hinweise, aber keine Listen der Namen.) Es handelt sich also im Grunde um eine Formalität, deren Zweck es ist, den Namensraum dauerhaft und eindeutig zu identifizieren. Solange der Schema-Autor einen URL verwendet, dessen Einmaligkeit er kontrollieren kann, weil er ja die Eindeutigkeit seines URLs kennt, gibt es keine Probleme für den XML-Prozessor.

2.9.3 Namensraumname und Präfix

Dass ein Name zu einem bestimmten Namensraum gehört, kann zum Zweck der Vereinfachung durch ein Präfix angegeben werden, das dem Namen vorangesetzt wird, wobei zur Trennung ein Doppelpunkt verwendet wird. Dieses Präfix kann bei der Deklaration eines Namensraums zugewiesen werden.

Das Präfix dient einfach nur als Abkürzung für den Namen des Namensraums; der XML-Prozessor wird diese Abkürzung immer durch den eigentlichen Namensraumnamen, also den URI, ersetzen.

2.9.4 Namensraumdeklaration und QNamen

Um Namensräume in einem XML-Dokument verwenden zu können, müssen sie zunächst deklariert werden. Dies geschieht in Form eines Elementattributs. Der Namensraum ist durch die Deklaration für das betreffende Element und alle seine Kindelemente gültig. Soll der Namensraum also im gesamten Dokument gültig sein, muss das Attribut dem Wurzelelement zugewiesen werden. Es ist aber auch möglich, erst auf einer tieferen Ebene ein Element mit einem Namensraum zu versehen.

Für die Deklaration werden reservierte Attribute verwendet. In dem folgenden Beispiel ordnet die Namensraumdeklaration dem Namensraumnamen `http://mitglieder.com/organisation` das Präfix `mtg` zu:

```
<mitglieder xmlns:mtg="http://mitglieder.com/organisation">
...
</mitglieder>
```

Dieses Präfix kann anschließend verwendet werden, um für ein Element oder Attribut anstelle eines einfachen Namens einen qualifizierten Namen – *QName* – zu verwenden:

```
<mtg:beitrag>100</mtg:beitrag>
```

In diesem Fall nennt das Element nicht nur seinen Namen, sondern gibt zugleich an, zu welchem Namensraum es gehört. Der qualifizierte Name beugt der Gefahr der Mehrdeutigkeit eines Namens vor, indem er diesen durch die Zuordnung zu einem Namensraum erweitert. Er besteht in diesem Fall aus dem Präfix, das die Bindung an den Namensraum herstellt, dem Doppelpunkt als Trennzeichen und dem lokalen Namen des Elements.

Auch bei Attributnamen kann so verfahren werden. Wenn der Attributname aber zum gleichen Namensraum gehört wie der Name des Elements, zu dem das Attribut gehört, darf der Attributname nicht mit dem entsprechenden Präfix versehen werden.

Das Präfix kann frei gewählt werden, nur die Zeichenfolge `xml` ist für den Namensraum `http://www.w3.org/XML/1998/namespace` und `xmlns` für die Einbindung von Namensräumen reserviert. Es ist an den Namensraum `http://www.w3.org/2000/xmlns` gebunden. Beide Namensräume müssen nicht deklariert werden.

Innerhalb eines XML-Dokuments werden die mit Präfix versehenen Namen ansonsten wie normale Namen behandelt oder wie Namen, die einen Doppelpunkt als eines der Zeichen enthalten.

2.9.5 Einsatz mehrerer Namensräume

Wenn erforderlich, kann auch mit mehreren Namensräumen gearbeitet werden:

```
<mtg:mitglieder xmlns:mtg="http://XMLbeisp.com/organisation"
                xmlns:pro="http://XMLbeisp.com/abrechnung"
                xmlns:mass="http://XMLbeisp.com/masse">
  <mtg:mitglied>
    <mtg:name>Hansen</name>
    <mtg:beitrag mass:waehrung="EUR">100</beitrag>
    <mtg:projekt>
```



```
      <pro:beschreibung>Haussanierung</pro:beschreibung>
      <pro:beitrag mass:einheit="Std" pro:status="ehrenamtlich">20
      </pro:beitrag>
    </mtg:projekt>
  </mtg:mitglied>
</mtg:mitglieder>
```

Mit Hilfe der Präfixe lassen sich die unterschiedlichen Elemente und Attribute exakt dem jeweils gültigen Namensraum zuordnen. Allerdings ist es beim Einsatz mehrerer Namensräume oft praktisch, einen dieser Namensräume als Default-Namensraum, also als Vorgabe, zu verwenden. Dies geschieht dadurch, dass bei der Deklaration kein Präfix zugeordnet wird. In der folgenden Variante des letzten Beispiels wird der erste angegebene Namensraum als Vorgabe verwendet:

```
<mitglieder xmlns="http://XMLbeisp.com/organisation"
            xmlns:pro="http://XMLbeisp.com/abrechnung">
  <mitglied>
    <name>Hansen</name>
    <beitrag waehrung="EUR">100</beitrag>
    <projekt>
      <pro:beschreibung>Haussanierung</pro:beschreibung>
      <pro:beitrag einheit="Std">20</pro:beitrag>
    </projekt>
  </mitglied>
</mitglieder>
```

Bei den Elementen, die zum vorgegebenen Namensraum gehören, kann dann auf ein Präfix verzichtet werden, was die Dokumente lesbarer macht und die Schreibarbeit verringert. Trotzdem handelt es sich bei diesen Elementnamen um qualifizierte Namen, auch wenn das sonst verwendete Präfix gleichsam unsichtbar geworden ist. Darauf wird in Kapitel 4, »Inhaltsmodelle mit XML Schema«, noch einmal eingegangen werden, wenn es um die Validierung von Dokumenten geht, die Namensräume verwenden. Werden Attribute ohne Präfix benannt, gehören sie dagegen nicht zum vorgegebenen Namensraum. Sie erben also nicht den vorgegebenen Namensraum des Elements, zu dem sie gehören.

Die Voreinstellung auf einen bestimmten Namensraum, die beispielsweise innerhalb des Wurzelelements vorgenommen worden ist, kann auf einer tieferen Ebene auch wieder überschrieben werden, indem erneut das Attribut `xmlns` ohne Präfixzuordnung verwendet wird. Die neue Vorgabe gilt dann aber nur für diese Ebene und eventuelle Kindelemente dieser Ebene. Soll eine Vorgabe ganz aufgehoben werden, kann auch mit dem Wertepaar `xmlns=""` gearbeitet werden. Dieses Verfahren

kann allerdings nicht für die Aufhebung von Namensraumzuordnungen verwendet werden, die mit einem Präfix arbeiten.

2.10 XML-Version 1.1

Das W3C hat 2004 die Version XML 1.1 verabschiedet, um der Weiterentwicklung von Unicode besser zu entsprechen. In der XML-Version 1.0 sind zwar für Elementinhalte und Attributwerte fast alle Unicode-Zeichen zugelassen, für Element- und Attributnamen, für Aufzählungen vorgegebener Attributwerte und für die Formulierung des Ziels von Verarbeitungsanweisungen sind aber nur die Zeichen zugelassen, die bereits in Unicode 2.0 enthalten sind. Um kommende Erweiterungen von Unicode zuzulassen, führt die Version XML 1.1 hier eine Lockerung ein und lässt auch in den Namen alle Zeichen zu, die nicht ausdrücklich verboten sind.

Die zweite Änderung betrifft die Behandlung von Leerraum. Um bisherige Umständlichkeiten beim Datenaustausch mit Großrechnern von IBM und damit kompatiblen Systemen zu vermeiden, wird den bisher für Leerraum verwendeten Zeichen – Space, Tab, CR, LF – noch ein NEL-Zeichen (Newline, U+0085 bzw. #x85) hinzugefügt. Zudem wird das Unicode-Zeilentrennzeichen U+2028 bzw. #x2028 unterstützt. Mit Ausnahme der erwähnten Leerraumzeichen dürfen Steuerzeichen im Codebereich #x1 bis #x1F und #x7F bis #x9F allerdings nur in Form von Zeichenreferenzen verwendet werden, damit XML-Dateien auch weiterhin von normalen Texteditoren angezeigt und bearbeitet werden können. Verboten in jeder Form bleibt dagegen das NUL-Zeichen (#x00).

Um den binären Vergleich von zwei Zeichenfolgen zu vereinfachen, soll außerdem eine Unicode-Normalisierung dafür sorgen, dass für jedes Zeichen eine einheitliche Schreibweise verwendet wird. Das ist bisher nicht unbedingt der Fall. Umlaute können beispielsweise durch ein einzelnes Zeichen oder als Kombination von zwei Zeichen dargestellt werden. Da diese Normalisierung aber in einem XML-Parser nicht einfach zu implementieren ist, kann die Umsetzung dieser Anforderung noch geraume Zeit dauern.

Obwohl die Änderungen von Version 1.0 zu 1.1 insgesamt gering bleiben, sind die entsprechenden XML-Dokumente nicht kompatibel. Eine XML 1.1-Datei ist für einen Version 1.0-Parser unter Umständen also nicht wohlgeformt.

Insgesamt folgt aus all dem, dass die Version 1.1 wohl über die nächsten Jahre in der praktischen Arbeit mit XML-Daten kaum eine Rolle spielen wird. Aus diesem Grund wird in diesem Buch auch weiterhin von der Version 1.0 ausgegangen. Dies gilt umso mehr, als in der 2008 erschienenen 5. Edition von XML 1.0 bereits einige der Lockerungen aus XML 1.1 in Bezug auf die erlaubten Element- und Attributnamen übernommen wurden.

Kapitel 10
Programmierschnittstellen für XML

Transformationen mit XSLT sind eine Möglichkeit, ein XML-Dokument auszuwerten – die Verwendung von Programmierschnittstellen für den Zugriff auf XML-Daten eine andere. DOM und SAX waren die ersten Lösungen auf diesem Gebiet.

In den vorangegangenen Kapiteln ist mehrfach die Rede von XML-Prozessoren gewesen, die mit XML-Dokumenten etwas anstellen, ohne dass gesagt worden ist, wie das tatsächlich geschieht. Wie können Programme mit XML-Dokumenten verfahren, das heißt sie verarbeiten, auswerten oder auch erzeugen?

Hierfür werden in diesem Kapitel zunächst die beiden bisher hauptsächlich verwendeten Techniken vorgestellt, die unter den Namen *DOM (Document Object Model)* und *SAX (Simple API for XML)* bekannt geworden sind. Außerdem werden einige grundlegende XML-Klassen im .NET Framework von Microsoft beschrieben, die zum Lesen und Schreiben von XML-Daten zur Verfügung stehen.

Allerdings kann an dieser Stelle nicht mehr als eine Einführung gegeben werden. Alles andere würde den Rahmen sprengen, wie schon ein Blick auf die umfangreichen Referenzen zu DOM und SAX zeigt.

10.1 Abstrakte Schnittstellen: DOM und SAX

In beiden Fällen werden für die Entwickler abstrakte Programmierschnittstellen definiert, die dafür gedacht sind, eine Anzahl von Standardverfahren für den Zugriff auf die in XML-Dokumenten enthaltenen Informationen zur Verfügung zu stellen.

Während es sich bei DOM um eine vom W3C verabschiedete Gruppe von Empfehlungen handelt, ist SAX, das hier in der Version 2 vorgestellt wird, ein De-facto-Standard, der von der Entwicklergemeinde schon seit längerer Zeit akzeptiert wird; *www.sax-project.org* ist die offizielle Website für SAX.

DOM war von Anfang an als eine plattform- und sprachunabhängige Schnittstellenbeschreibung für den Zugriff auf XML-Dokumente (und gleichzeitig auf HTML) konzipiert.

SAX wurde zunächst nur als Java-API entwickelt, die aktuelle Version liegt aber auch in Varianten für andere Entwicklungsumgebungen vor. Seit dem Microsoft Parser *MSXML 3* gibt es auch Varianten von SAX für C++ und Microsoft Visual Basic, allerdings werden den Java-Namen der SAX-Schnittstellen Präfixe wie `IMX` oder `ISAX` vorangesetzt. Unterstützung für SAX gibt es auch in anderen Sprachen, etwa Perl, Python oder Pascal.

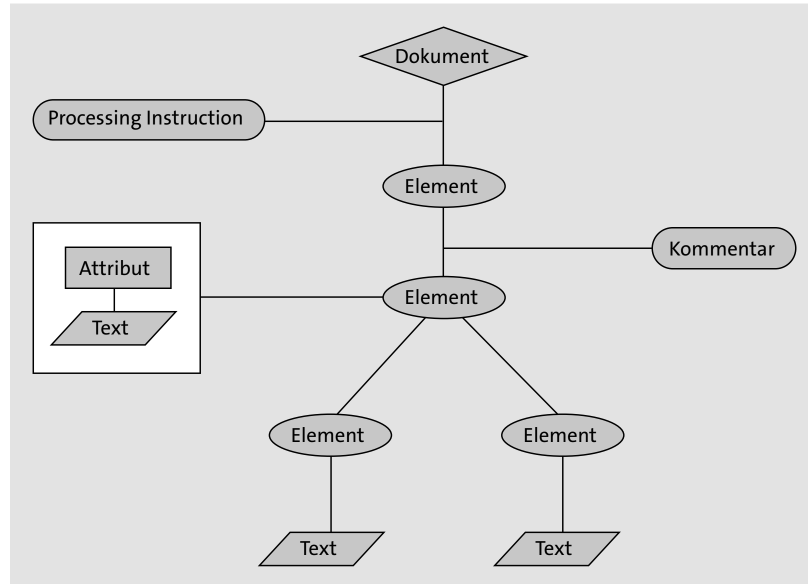

Abbildung 10.1 Allgemeine Struktur des DOM-Baumes

Das abstrakte Datenmodell, das einem XML-Dokument zugrunde liegt, wird in beiden Fällen auf ein Objektmodell abgebildet, das es den Programmierern erlaubt, auf der Basis der logischen Struktur des Dokuments zu arbeiten, anstatt sich mit eckigen Klammern und mit Zeichenreferenzen herumzuplagen, die das Dokument zunächst anbietet.

Datenstrom versus Knotenbaum

Ansonsten sind die beiden Techniken vom Ansatz her grundverschieden. SAX besteht aus Streaming-Schnittstellen, die die Informationseinheiten, aus denen ein XML-Dokument zusammengesetzt ist, in eine Abfolge von Methodenaufrufen teilen. Dagegen besteht DOM aus einem Satz von Schnittstellen, die das dem XML-Standard zugrunde liegende Infoset in einen hierarchischen Baum von Objekten abbilden, die *Knoten* genannt werden. Um eine wahlfreie Abfrage und Manipulation dieser Knoten zu ermöglichen, wird der Baum im Speicher aufgebaut und dort bereitgehalten. Bei sehr großen Dokumenten kann der Speicher also durchaus zum Engpass werden, was in einer entsprechenden Anwendung berücksichtigt werden sollte.

SAX stellt dagegen Verarbeitungsmöglichkeiten für XML-Dokumente zur Verfügung, die durch Ereignisse gesteuert werden. Ein Dokument wird dabei sequenziell durchgearbeitet. Beim Auftauchen eines Start-Tags wird beispielsweise eine entsprechende Methode aufgerufen, und das Programm trifft daraufhin die Maßnahmen, die der Entwickler vorgesehen hat.

Die Größe des Dokuments kann bei dieser Vorgehensweise deshalb nicht zum Problem werden, weil immer nur eine Zeile aus dem Dokument verarbeitet wird. SAX erlaubt eine schnelle Verarbeitung des gesamten Dokuments in einem Durchgang.

Eine Schwäche dieses Ansatzes ist allerdings, dass eine gezielte Bearbeitung bestimmter Stellen im Dokument oder Sprünge von einer Stelle zur anderen nicht möglich sind. Für solche Aufgaben ist eher die DOM-Technologie geeignet, die es erlaubt, in dem im Speicher aufgebauten Knotenbaum nicht nur frei herumzuturnen, sondern auch neue Knoten einzufügen. Der Baum besteht, wie schon angesprochen, in diesem Fall aus Objekten im Sinne der objektorientierten Programmierung, also aus ansprechbaren Einheiten, in denen jeweils Daten mit Eigenschaften und Methoden gekoppelt sind. Die Eigenschaften geben Auskunft über die Art der Daten oder bestimmen diese; die Methoden betreffen das, was mit den Daten geschehen kann.

Beide Technologien können aber durchaus auch im Verbund eingesetzt werden, etwa indem mit SAX das Dokument für die weitere Bearbeitung mit DOM präpariert wird.

DOM und SAX liefern beide abstrakte Schnittstellen, bestimmen also nicht im Detail, wie eine bestimmte Sprache diese Schnittstellen verwendet.

10.2 Document Object Model (DOM)

Das DOM ist vom W3C entwickelt worden, um den Zugriff auf Webdokumente – in HTML oder XML – zu vereinheitlichen, nachdem sich in dem Gerangel um die Vorherrschaft auf dem Browsermarkt im Zusammenhang mit Erweiterungen zu HTML ein unerfreulicher Wildwuchs breitgemacht hatte. Die Objekthierarchien, die damals für JavaScript und andere Skriptsprachen entwickelt wurden, um mehr Dynamik in die starren Webseiten einzubauen, drohten auseinanderzudriften.

Die leitende Vorstellung war dabei, dass ein XML-Prozessor ein beliebiges XML-Dokument über allgemein anerkannte Schnittstellen als Knotenbaum aufbereitet. Andere Anwendungen, die diese Schnittstellen ebenfalls unterstützen, greifen auf den im Speicher verfügbaren Baum in gleicher Weise zu, um Informationen auszulesen oder um Informationen hinzuzufügen oder zu ändern.

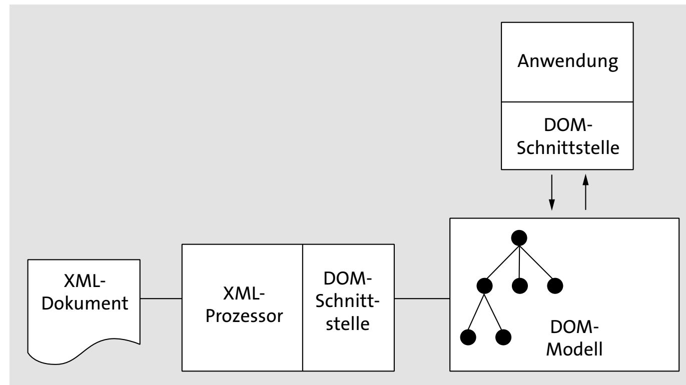

Abbildung 10.2 Architektur von DOM-Anwendungen

10.2.1 DOM Level

In den Empfehlungen des W3C sind die Schnittstellen, die das DOM für die Darstellung und Manipulation eines XML-Dokuments zur Verfügung stellt, mit Hilfe der sprachenunabhängigen *Interface Definition Language (IDL)* definiert, die von der Object Management Group entwickelt wurde.

Zusätzlich wurden für Java und *ECMAScript* spezielle Sprachabbildungen der IDL-Spezifikationen bereitgestellt. Bindungen für andere Sprachen regeln, wie die Schnittstellen jeweils in einer bestimmten Entwicklungsumgebung implementiert werden.

Die DOM Level-1-Spezifikation von Oktober 1998 berücksichtigte noch keine Namensräume; DOM Level 2 wurde im November 2000 in Form von sechs separaten Modulen verabschiedet, wobei die für XML wichtigsten Schnittstellen in der *Core Specification* zusammengefasst sind. DOM-Implementierungen müssen diesen Kern unterstützen, aber nicht unbedingt alle anderen Module, die sich zum Beispiel mit Ansichten, Styles oder Ereignissen befassen. Neben der Unterstützung von Namensräumen brachte Level 2 auch neue Module zur Unterstützung von Cascading Stylesheets, zur Ereignisbehandlung und weiterentwickelte Methoden für die Bewegung im Knotenbaum.

Seit 2004 ist DOM Level 3 in drei separaten Empfehlungen spezifiziert. Insbesondere die *Load-and-Save*-Spezifikation bringt nützliche Erweiterungen in Form von neuen Schnittstellen für den Umgang mit dem `Document`-Objekt – Dinge, die bisher den verschiedenen Implementierungen von DOM überlassen wurden, worauf im weiteren Verlauf noch eingegangen wird.

Für HTML5 und die entsprechende Version von XHTML wurden weitere Anpassungen vorgenommen, die jetzt unter dem Namen *DOM Living Standard* von der WHATWG Community unter *https://dom.spec.whatwg.org/* angeboten werden. Mehr Details dazu finden Sie in Kapitel 15.

10.2.2 Objekte, Schnittstellen, Knoten und Knotentypen

Die Idee, die hinter DOM steht, ist, dass sich ein XML-Dokument nicht bloß als eine Abfolge von Zeichen betrachten lässt, sondern gleichzeitig als eine geordnete Menge von Objekten. Für alle möglichen Komponenten eines XML-Dokuments wird deshalb durch das DOM jeweils eine abstrakte Objektklasse mit einer genau definierten Schnittstelle bereitgestellt, der eine bestimmte Menge von Methoden und Attributen zugeordnet ist.

Die Klasse `Element` stellt beispielsweise Methoden wie `setAttribute()` oder `getAttribute()` zur Verfügung, mit denen Sie Werte der zugeordneten Attribute bestimmen oder auslesen. Gleichzeitig lässt sich von einem Element aus auf die Werte der ihm zugeordneten Attribute zugreifen. Außerdem gibt es bestimmte Festlegungen, welche Eigenschaften das Objekt haben kann und ob diese nur gelesen oder auch verändert werden dürfen. Schließlich ist über das Objektmodell auch festgelegt, in welcher Beziehung eine Objektklasse zu anderen stehen kann, etwa ob ein Objekt Unterobjekte enthalten kann oder nicht.

Den Objektklassen entspricht in der Regel ein bestimmter Knotentyp in der Baumrepräsentation des Objektmodells. Dabei entsprechen die Knotentypen den im *XML Information Set* definierten Informationseinheiten. Nur die dort aufgeführten *Character Information Items* werden nicht als einzelne Knoten gehandhabt, sondern zu Textknoten zusammengefasst, um die Handhabung des Modells nicht zu sehr zu verkomplizieren.

10.2.3 Die allgemeine Node-Schnittstelle

Zentraler Punkt des Kernmoduls von DOM ist die `Node`-Schnittstelle, die als Basisschnittstelle für alle Knotentypen verwendet wird. Sie repräsentiert einen einzelnen Knoten im Dokumentenbaum.

Alle Objekte implementieren diese Schnittstelle, für die ein allgemeiner Satz von Methoden, Attributen und Konstanten definiert ist – erforderlich, um beliebige Knoten anzusteuern, abzufragen oder zu verändern. Die Konstanten werden den speziellen Knotentypen zugeordnet, um Abfragen mit dem `nodeType`-Attribut zu vereinfachen.

Das bedeutet also, dass alle anderen Knotentypen von `Node` abgeleitet sind; sie erben von `Node` alle Methoden, können aber noch zusätzliche enthalten.

Der Vorteil dieser Technik besteht darin, dass Sie sich in dem Knotenbaum mit den für `Node` definierten Methoden bewegen können, egal welcher Knotentyp gerade erreicht ist. Die dadurch erreichte Einheitlichkeit vereinfacht das Klettern im Knotenbaum. Nur bei den Zugriffen, die ausschließlich für einen bestimmten Knotentyp möglich sind, kommen die speziellen Schnittstellen dieser Knotentypen ins Spiel.

Die IDL-Definition von `Node` sieht in der WHATWG-Spezifikation so aus:

```
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  const unsigned short ATTRIBUTE_NODE = 2; // historical
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4; // historical
  const unsigned short ENTITY_REFERENCE_NODE = 5; // historical
  const unsigned short ENTITY_NODE = 6; // historical
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short COMMENT_NODE = 8;
  const unsigned short DOCUMENT_NODE = 9;
  const unsigned short DOCUMENT_TYPE_NODE = 10;
  const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short NOTATION_NODE = 12; // historical
  readonly attribute unsigned short nodeType;
  readonly attribute DOMString nodeName;

  readonly attribute DOMString? baseURI;

  readonly attribute Document? ownerDocument;
  readonly attribute Node? parentNode;
  readonly attribute Element? parentElement;
  boolean hasChildNodes();
  [SameObject] readonly attribute NodeList childNodes;
  readonly attribute Node? firstChild;
  readonly attribute Node? lastChild;
  readonly attribute Node? previousSibling;
  readonly attribute Node? nextSibling;

           attribute DOMString? nodeValue;
           attribute DOMString? textContent;
  void normalize();

  [NewObject] Node cloneNode(optional boolean deep = false);
  boolean isEqualNode(Node? node);

  const unsigned short DOCUMENT_POSITION_DISCONNECTED = 0x01;
  const unsigned short DOCUMENT_POSITION_PRECEDING = 0x02;
  const unsigned short DOCUMENT_POSITION_FOLLOWING = 0x04;
  const unsigned short DOCUMENT_POSITION_CONTAINS = 0x08;
  const unsigned short DOCUMENT_POSITION_CONTAINED_BY = 0x10;
  const unsigned short DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;
  unsigned short compareDocumentPosition(Node other);
  boolean contains(Node? other);
```



```
  DOMString? lookupPrefix(DOMString? namespace);
  DOMString? lookupNamespaceURI(DOMString? prefix);
  boolean isDefaultNamespace(DOMString? namespace);

  Node insertBefore(Node node, Node? child);
  Node appendChild(Node node);
  Node replaceChild(Node node, Node child);
  Node removeChild(Node child);};
```

Listing 10.1 IDL-Definition von Node

Allerdings kann nicht bei jedem Knotentyp jede Methode angewandt werden. Beispielsweise können zu einem Textknoten keine Knoten hinzugefügt werden; die entsprechende `appendChild`-Methode würde zu einem Fehler, zu einer Ausnahme, einer `DOMException`, führen. Im Zweifelsfall kann auch zunächst der spezielle Knotentyp abgefragt werden, bevor eine bestimmte Methode angewendet wird.

Auch sind nicht alle Attribute bei allen Knoten abfragbar; zum Beispiel hat ein Kommentarknoten keinen Namen. Eine Abfrage des Namens führt aber nicht zu einem Fehler, sondern liefert einfach nur den Wert `null`.

10.2.4 Knotentypen und ihre Besonderheiten

Zusätzlich zu dieser allgemeinen `Node`-Schnittstelle definiert DOM, wie schon angedeutet, noch spezielle Schnittstellen für die einzelnen Knotentypen, die weitere Mechanismen anbieten können. Die Schnittstelle `Element` erbt zum Beispiel von der allgemeinen `Node`-Schnittstelle alles, was verwendbar ist, gibt aber zusätzlich die Möglichkeit, etwa Attribute über ihren Namen nach ihrem Wert zu befragen.

Von Bedeutung für den Umgang mit dem DOM ist insbesondere auch die Frage, welche Beziehungen die Knotentypen zueinander haben können. Tabelle 10.1 listet die verschiedenen Knotentypen auf und gibt zugleich an, ob und, wenn ja, welche Kindknoten jeweils möglich sind.

DOM-Knoten	Mögliche Kinder
`Document`	(maximal ein) `Element`, `ProcessingInstruction`, `Comment`, (maximal ein) `DocumentType`
`DocumentFragment`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`DocumentType`	keine

Tabelle 10.1 Liste der DOM-Knoten

DOM-Knoten	Mögliche Kinder
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	keine
Comment	keine
Text	keine
CDATASection	keine
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	keine

Tabelle 10.1 Liste der DOM-Knoten (Forts.)

10.2.5 Zusätzliche Schnittstellen

Neben den Schnittstellen für die verschiedenen Knotentypen spezifiziert DOM eine `Nodelist`-Schnittstelle, um geordnete Knotenlisten handhaben zu können, etwa die Liste der Kinder eines Elements.

Für ungeordnete Knotenlisten, die über den Knotennamen zusammengestellt werden können, wird die `NamedNodeMap` zur Verfügung gestellt. Diese Listen sind dynamische Objekte, die sich automatisch anpassen, wenn beispielsweise ein Element eingefügt oder gelöscht wird.

Obwohl die Handhabung solcher Listen teilweise denen von Arrays ähnelt, handelt es sich also nicht um Arrays, sondern um aktuelle Ansichten von dem betreffenden Teil des Knotenbaums. Das muss beim Zugriff über Indizes beachtet werden, da sich der Index eines nachfolgenden Knotens sofort ändert, wenn davor ein Knoten eingefügt oder entfernt wird.

Speziell für die Verarbeitung von Zeichenfolgen ist eine abstrakte Schnittstelle `CharacterData` definiert, die keine direkte Entsprechung in einem Knotenbaum hat, aber einen Satz von Methoden und Eigenschaften enthält, die von den Knotentypen `Text`, `Comment` und `CDATASection` geerbt werden. Außerdem werden Schnittstellen für die Fehlerbehandlung angeboten wie `DOMException`.

10.2.6 Zugriff über Namen

Eine Reihe von Knotentypen kann nicht nur über die verschiedenen Klettermethoden wie `firstChild`, `nextSibling` etc., sondern auch direkt über den Namen angesprochen werden. Dazu wird die Eigenschaft `Node.nodeName` verwendet. Bei den Knotentypen, die keinen individuellen Namen haben, liefert `Node.nodeName` vorgegebene Werte, die in Tabelle 10.2 zu finden sind. Der Name eines Knotens kann immer nur gelesen, nicht verändert werden. Allerdings lässt sich diese Einschränkung umgehen, indem ein neuer Knoten eingefügt wird und die Inhalte des anderen Knotens dorthin kopiert werden.

Die Tabelle enthält außerdem eine Liste der möglichen Werte, die über die Eigenschaft `Node.nodeValue` abgelesen werden können.

nodeType	nodeName	nodeValue
Attr	Name des Attributs	Wert des Attributs
CDATASection	#cdata-section	Inhalt der CDATA-Section
Comment	#comment	Inhalt des Kommentars
Document	#document	null
DocumentFragment	#document-fragment	null
DocumentType	Dokumenttyp-Name	null
Element	Tag-Name	null
Entity	Name der Entität	null
EntityReference	Name der referenzierten Entität	null
Notation	Name der Notation	null
ProcessingInstruction	Ziel	gesamter Inhalt ohne das Ziel
Text	#text	Inhalt des Textknotens

Tabelle 10.2 Liste der Knotentypen

10.2.7 Verwandtschaften

Die Beziehungen der Knoten im Knotenbaum werden durch die entsprechenden Attribute der `Node`-Schnittstelle ausgedrückt, die die Eltern-Kind-Beziehungen betreffen. Die Eigenschaft `parentNode` gibt zum Beispiel an, von welchem Knoten der Knoten abstammt; `previousSibling` gibt den vorhergehenden Knoten auf derselben Ebene an. Wird eine dieser Eigenschaften bei einem Knotentyp abgefragt, bei dem die

entsprechende Beziehung nicht möglich ist, liefert das Attribut immer den Wert `null`. Bei einem Kommentarknoten oder einem Processing-Instruction-Knoten ergibt `firstChild` also immer `null`.

Mit Hilfe von DOM lassen sich auch neue Knoten in den Knotenbaum einfügen. Dafür gibt es zwei allgemeine Verfahren. Mit `Node.appendChild` lässt sich ein neuer Knoten an eine Folge von Tochterknoten anhängen, während sich mit `Node.insertBefore` genau bestimmen lässt, an welcher Stelle in einer Liste von Tochterknoten ein Knoten eingefügt werden soll.

Es ist auch möglich, einen bestehenden Knoten mit `Node.removeChild` zu entfernen oder ihn mit `Node.replaceChild` durch einen neuen Knoten zu ersetzen.

Auch das Klonen eines Knotens und eventuell all seiner Unterknoten kann mit `Node.cloneNode` erreicht werden, allerdings wird dieser Klon zunächst nicht in den Knotenbaum eingefügt, sondern muss anschließend mit `Node.appendChild` oder `Node.insertBefore` an einer bestimmten Stelle eingefügt werden.

10.2.8 Das Dokument als DOM-Baum

Um zu zeigen, wie DOM ein XML-Dokument in einen Knotenbaum umsetzt, verwenden wir hier eine etwas veränderte Version des Lagerbeispiels aus dem Kapitel 3 über DTDs.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="lagerliste.xslt"?>

<Lager>
  <Artikel nr="7777" wg="Jalou">
    <Bezeichnung>Jalousie Cxs</Bezeichnung>
    <Bestand>100</Bestand>
    <Preis>198</Preis>
    <Absatz>120</Absatz>
  </Artikel>
  <Artikel nr="7778" wg="Jalou">
    <Bezeichnung>Jalousie Cxx</Bezeichnung>
    <Bestand>200</Bestand>
    <Preis>174</Preis>
    <Absatz>330</Absatz>
  </Artikel>
  <Artikel nr="5554" wg="Rollo">
    <Bezeichnung>Rollo PCx</Bezeichnung>
    <Bestand>150</Bestand>
    <Preis>95</Preis>
    <Absatz>200</Absatz>
  </Artikel>
```



```
  <Artikel nr="7999" wg="Rollo">
    <Bezeichnung>Sunset</Bezeichnung>
    <Bestand>200</Bestand>
    <Preis>120</Preis>
    <Absatz>200</Absatz>
  </Artikel>
  <Artikel nr="8444" wg="MK">
    <Bezeichnung>Markise Blue Sk</Bezeichnung>
    <Bestand>160</Bestand>
    <Preis>287,5</Preis>
    <Absatz>80</Absatz>
  </Artikel>
</Lager>
```


Listing 10.2 lagerdaten.xml

Wenn ein XML-Parser dieses Dokument als DOM ausgibt, wird im Speicher ein entsprechender Baum aufgebaut, der dem logischen Modell des Dokuments entspricht. Beim Einlesen in den Speicher wird jedem erstellten Knoten gleichzeitig der entsprechende Knotentyp zugewiesen. Mit Hilfe dieser Metadaten wird damit festgelegt, welche Merkmale und Funktionen für jeden einzelnen Knoten zur Verfügung stehen.

Abbildung 10.3 zeigt, wie das lineare XML-Dokument durch den Knotenbaum repräsentiert wird.

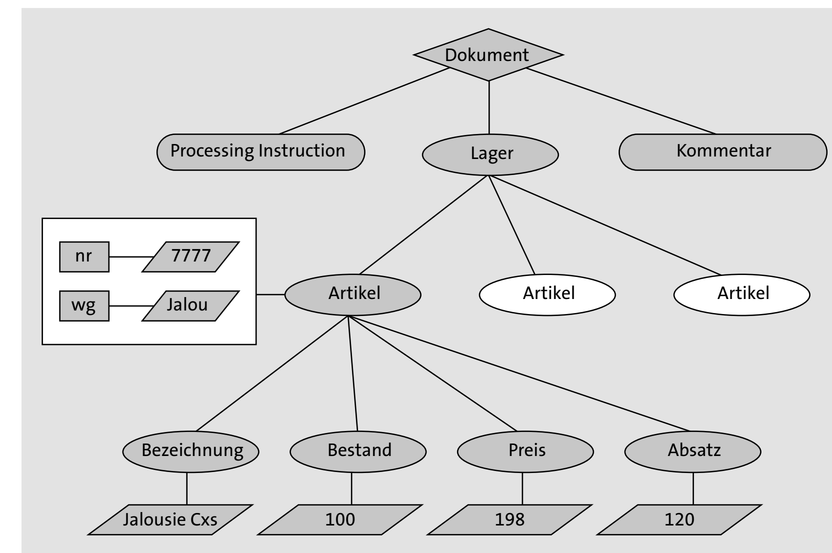

Abbildung 10.3 DOM-Baum für das Lagerbeispiel

10.2.9 Document – die Mutter aller Knoten

Die Wurzel des Baumes bildet der `Document`-Knoten. Dieser Knoten repräsentiert das gesamte Dokument und liefert den primären Zugang zu seinen Daten. Es ist der einzige Knoten, der nicht Kindknoten eines anderen ist, während alle anderen Knoten jeweils einen übergeordneten Knoten haben. Wie ein großer Container schließt dieser Knoten alle anderen in sich ein, und deshalb sind an ihn auch alle Fabrikmethoden gebunden, die für die Erzeugung der anderen Knoten benötigt werden. Die IDL-Definition listet die Fabrikmethoden auf:

```
interface Document : Node {
  [SameObject] readonly attribute DOMImplementation implementation;
  readonly attribute DOMString URL;
  readonly attribute DOMString documentURI;
  readonly attribute DOMString origin;
  readonly attribute DOMString compatMode;
  readonly attribute DOMString characterSet;
  readonly attribute DOMString inputEncoding; // legacy alias of .characterSet
  readonly attribute DOMString contentType;

  readonly attribute DocumentType? doctype;
  readonly attribute Element? documentElement;
  HTMLCollection getElementsByTagName(DOMString localName);
  HTMLCollection getElementsByTagNameNS(DOMString?
    namespace, DOMString localName);
  HTMLCollection getElementsByClassName(DOMString classNames);

  [NewObject] Element createElement(DOMString localName);
  [NewObject] Element createElementNS(DOMString?
    namespace, DOMString qualifiedName);
  [NewObject] DocumentFragment createDocumentFragment();
  [NewObject] Text createTextNode(DOMString data);
  [NewObject] Comment createComment(DOMString data);
  [NewObject] ProcessingInstruction createProcessingInstruction
   (DOMString target, DOMString data);

  [NewObject] Node importNode(Node node, optional boolean deep = false);
   Node adoptNode(Node node);

  [NewObject] Attr createAttribute(DOMString localName);
  [NewObject] Attr createAttributeNS(DOMString? namespace, DOMString name);

  [NewObject] Event createEvent(DOMString interface);
```



```
  [NewObject] Range createRange();

  // NodeFilter.SHOW_ALL = 0xFFFFFFFF
  [NewObject] NodeIterator createNodeIterator(Node root, optional
   unsigned long whatToShow = 0xFFFFFFFF, optional NodeFilter? filter = null);
  [NewObject] TreeWalker createTreeWalker(Node root, optional unsigned long
   whatToShow = 0xFFFFFFFF, optional NodeFilter? filter = null);
};
```

Listing 10.3 IDL-Definition für Node

Die mit diesen Methoden erzeugten Knoten verraten jeweils über den Wert des Attributs `ownerDocument`, wo sie als Unterknoten angesiedelt sind.

Der Knoten `Document` ist nicht identisch mit dem Wurzelelement des Dokuments. Dieses Wurzelelement wird vielmehr dargestellt durch einen `Element`-Knoten, der Kind des `Document`-Knotens ist. Auf derselben Ebene wie der erste `Element`-Knoten können noch ein `DocumentType`-Knoten und solche für Kommentare und Processing Instructions auftauchen.

10.2.10 Elementknoten

In Abbildung 10.3 entspricht der erste Knoten vom Typ `Element` dem Wurzelelement `<Lager>` im XML-Dokument. Auch die untergeordneten Elemente für die verschiedenen Artikel werden durch Knoten vom Typ `Element` repräsentiert. Jeder Artikelknoten hat wiederum eine Liste von Kindknoten, die untereinander eine geordnete Geschwisterbeziehung haben, die durch die Dokumentreihenfolge vorgegeben ist. Das erste Kind des Artikelknotens ist deshalb der Knoten für das Element `<Bezeichnung>`, das letzte Kind ist das Element `<Absatz>`.

10.2.11 Textknoten

Die Zeichendaten, die den Inhalt der Elemente der unteren Ebene ausmachen, werden als Textknoten behandelt. Damit Anwendungen von beliebigen Plattformen dabei keine Probleme haben, gibt es in DOM für die Darstellung von Zeichendaten einen Standarddatentyp `DOMString`, der als eine Sequenz von 16-Bit-Einheiten definiert ist. Für die Zeichencodierung wird UTF-16 verlangt.

Für die Manipulation von Zeichenketten eines Textknotens können die Attribute und Methoden der schon angesprochenen abstrakten Schnittstelle `CharacterData` genutzt werden, die der Textknoten beerbt. Beispielsweise lassen sich mit `substringData` Teile der Zeichendaten extrahieren oder mit `insertData` Zeichenfolgen in den

Textknoten einfügen. Die Länge der Zeichendaten kann über das `length`-Attribut abgefragt werden.

10.2.12 Attributknoten sind anders

Eine Besonderheit von DOM ist die Art und Weise, wie die Attribute von Elementen behandelt werden. Zwar erbt auch die `Attr`-Schnittstelle die allgemeinen Merkmale der `Node`-Schnittstelle, aber Attribute werden nicht als Knoten im eigentlichen Sinne – mit eigener Identität – betrachtet. Sie sind insofern auch keine Kinder des Elementknotens, zu dem sie gehören. Sie sind in diesem Sinne nicht Teil des Knotenbaums und lassen sich deshalb auch nicht mit den Methoden erreichen, mit denen sonst der Knotenbaum durchwandert wird.

Attribute werden als Eigenschaften der Elemente gehandhabt, zu denen sie gehören. Um auf den Attributwert zuzugreifen, kann entweder `Node.attributes` verwendet werden oder eine der speziellen Methoden, die für den Elementtypknoten definiert sind, wie `getAttribute`, `setAttribute` etc.

Gibt es zu einem Element mehrere Attribute, so besteht zwischen diesen Attributen auch keine Geschwisterbeziehung, wie es zwischen Elementen auf derselben Ebene der Fall ist. In unserem Beispiel haben die `<Artikel>`-Elemente zwei Attribute, `nr` und `wg`, aber die Reihenfolge hat hier keine Bedeutung, die Liste ist ungeordnet. Die Attribute `parentNode`, `previousSibling` oder `nextSibling` liefern deshalb immer nur das Ergebnis `null`. Allerdings ist es möglich, über das Attribut `Attr.ownerElement` herauszufinden, zu welchem Element ein Attribut gehört.

Die einem Attributnamen zugeordneten Attributwerte werden in Textknoten oder auch in Knoten mit Entitätsreferenzen repräsentiert. Wenn dem Attribut im XML-Dokument explizit ein Wert zugewiesen worden ist, ist das auch der effektive Wert des Attributs. Ist über DTD oder XML Schema ein Vorgabewert bestimmt, wird dieser Wert eingesetzt, wenn kein expliziter Wert im Dokument zu finden ist. Ist weder eine Vorgabe noch eine explizite Wertzuweisung erfolgt, wird auch kein Attributknoten für dieses Attribut im DOM erzeugt.

10.2.13 Dokumentfragmente

Außerhalb des Knotenbaums ist noch eine Schnittstelle definiert, deren IDL-Definition sehr spartanisch ist:

```
interface DocumentFragment : Node {
};
```

Diese Schnittstelle übernimmt also einfach die `Node`-Definition und fügt ihr nichts hinzu. Sie kann verwendet werden, um einen Teil des Dokumentenbaums zunächst

auszuschneiden und dann an anderer Stelle einzufügen. Das Dokumentfragment ist gewissermaßen ein Hilfscontainer für die vorübergehende Aufnahme von Knoten. Wird der Inhalt des Dokumentfragments dann irgendwo eingefügt, werden nur seine Kindelemente verwendet.

10.2.14 Fehlerbehandlung

Die Schnittstelle `DOMException` ist im DOM für die Behandlung von Ausnahmesituationen definiert. Für den Fall, dass eine bestimmte Operation nicht ausgeführt werden kann, ist die Ausgabe von Fehlercodes vorgesehen, wobei für die verschiedenen Methoden, die zu Problemen führen können, bestimmte Konstanten als Fehlerwerte vorgegeben werden. Allerdings ist diese Art der Ausnahmebehandlung nur ein Vorschlag, jede konkrete DOM-Implementierung kann auch anders verfahren.

10.3 DOM-Implementierungen

Mit den abstrakten Schnittstellen des DOM lässt sich so lange nichts Praktisches anfangen, wie keine konkreten Anwendungen und Anwendungssysteme greifbar sind, die das DOM implementieren. Wie diese Implementierung genau aussehen muss, schreibt das DOM nicht im Detail vor, abgesehen von den schon erwähnten Sprachbindungen für Java und ECMAScript.

Diese Implementierungen müssen auch Dinge regeln, zu denen sich die Spezifikation nicht verbindlich äußert. Wie zum Beispiel ein XML-Dokument gespeichert oder geladen wird, überließ die Spezifikation auf Level 1 und 2 der jeweiligen Anwendung, die die DOM-Schnittstelle implementiert. Solche Erweiterungen des W3C DOM werden durch die DOM-Spezifikationen durchaus zugelassen.

Die Implementierungen müssen auch nicht alle Module übernehmen, die seit Level 2 ja auf verschiedene Spezifikationen verteilt sind. Deshalb ist in DOM auch eine Schnittstelle `DOMImplementation` definiert, die über die Methode `DOMImplementation.hasFeature` Abfragen darüber erlaubt, ob die konkrete Implementierung bestimmte Merkmale von DOM unterstützt. Tabelle 10.3 enthält die seit DOM Level 2 vorgegebenen Merkmalsnamen.

Modul	Name
Core	`"Core"`
XML	`"XML"`
HTML	`"HTML"`

Tabelle 10.3 Liste der den DOM-Modulen zugeordneten Namen

Modul	Name
Views	"Views"
Style Sheets	"StyleSheets"
CSS	"CSS"
CSS2	"CSS2"
Events	"Events"
User Interface Events	"UIEvents"
Mouse Events	"MouseEvents"
Mutation Events	"MutationEvents"
HTML Events	"HTMLEvents"
Range	"Range"
Traversal	"Traversal"

Tabelle 10.3 Liste der den DOM-Modulen zugeordneten Namen (Forts.)

10.4 Die MSXML-Implementierung von DOM

Eine der häufig genutzten DOM-Implementierungen ist Bestandteil des XML-Parsers von Microsoft, der unter der Bezeichnung *Microsoft XML Core Services (MSXML)* bereitgestellt wird. Er basiert auf COM, also dem *Component Object Model*, das für die Windows-Plattform entwickelt worden ist, um das Zusammenspiel von Softwarekomponenten zu ermöglichen. Seit der Version 4 unterstützt diese Komponente weitgehend die vom W3C vorgegebenen Standards: DOM Level 1 und 2, XPath 1.0, XML Schema 1.0 und XSLT 1.0. Auch SAX 2.0 wird inzwischen vollständig abgedeckt. Die Version MSXML 6.0 wurde in die Windows-Betriebssysteme ab XP mit Service Pack 3 integriert. Die älteren Versionen von MSXML wurden noch separat installiert. Einige Features von MSXML 3 und 4 werden allerdings in der Version 6.0 nicht mehr angeboten. Dies betrifft insbesondere XDR Schemas und die Verwendung von XML-Dateninseln in Webseiten.

Die Tatsache, dass MSXML in DOM definierte Schnittstellen implementiert, schließt nicht aus, dass zu den einzelnen Schnittstellen auch zahlreiche Erweiterungen gegenüber dem W3C DOM in Form zusätzlicher Methoden und Eigenschaften eingebaut sind.

MSXML unterstützt XSLT-Transformationen, XSL-Stylesheets, Abfragen mit XPath-Ausdrücken, die Verwendung von Namensräumen, einen differenzierteren Einsatz von Datentypen, das asynchrone Laden vom Server sowie das Laden und Sichern von XML-Dokumenten. Zudem werden weitere Hilfsschnittstellen definiert, um bestimmte Grundoperationen zu erleichtern.

10.4.1 Schnittstellen in MSXML

Microsoft setzt vor die vom W3C verwendeten DOM-Schnittstellennamen eine Art Präfix, in der Regel `IXMLDOM`, statt `Node` also `IXMLDOMNode`. Tabelle 10.4 zeigt die Namen der Schnittstellen, die als *Kernschnittstellen* bezeichnet werden.

MSXML-Name	DOM-Name	Bedeutung
`IXMLDOMDocument/ DOMDocument`	`Document`	Oberster Knoten im XML-DOM-Baum.
`IXMLDOMDocument2`	–	Eine Erweiterung von `IXMLDOMDocument`, die das Laden von XML-Schemas und die Validierung des eingelesenen Dokuments unterstützt, außerdem die Nutzung von XPath.
`IXMLDOMDocument3`	–	Erweitert `IXMLDOMDocument2` um die Methoden `importNode()` und `validateNode()`.
`IXMLDOMNamedNodeMap`	`NamedNodeMap`	Erlaubt den Zugriff auf eine Sammlung von Attributen über den Namen, wobei Namensräume unterstützt werden.
`IXMLDOMNode`	`Node`	Steht für einen einzelnen Knoten im Knotenbaum und ist die Basisschnittstelle für den Zugriff auf die Daten im DOM. `IXMLDOMNode` erweitert die `Node`-Schnittstelle durch die Unterstützung für Datentypen, Namensräume, DTDs und XML-Schemas.
`IXMLDOMNodeList`	`NodeList`	Erlaubt wiederholte und indizierte Zugriffe auf eine aktuelle Sammlung von Knoten.

Tabelle 10.4 Liste der Kernschnittstellen in MSXML

MSXML-Name	DOM-Name	Bedeutung
IXMLDOMParseError	–	Gibt detaillierte Informationen über den letzten Fehler wieder. Neben dem Fehlercode werden die Zeilennummer, die Zeichenposition und eine Fehlerbeschreibung geliefert.
IXMLHTTPRequest	–	Erlaubt die Kommunikation mit HTTP-Servern.

Tabelle 10.4 Liste der Kernschnittstellen in MSXML (Forts.)

Tabelle 10.5 zeigt die Implementierung weiterer DOM-Schnittstellen.

MSXML-Name	DOM-Name	Bedeutung
IServerXMLHTTPRequest/ ServerXMLHTTP	–	Stellt Methoden und Eigenschaften zur Verfügung, um eine HTTP-Verbindung zwischen Dateien oder Objekten auf unterschiedlichen Webservern herzustellen. Die Schnittstelle ist eine Ableitung von IXMLHTTPRequest.
IXMLDOMAttribute	Attr	Steht für ein Attribut.
IXMLDOMCDATASection	CDATASection	Maskiert Textblöcke, damit kein Text darin als Teil des Markups interpretiert wird.
IXMLDOMCharacterData	CharacterData	Ermöglicht Methoden zur String-Manipulation, die von den abgeleiteten Knotentypen wie Text und Comment genutzt werden können.
IXMLDOMComment	Comment	Repräsentiert den Inhalt eines XML-Kommentars.
IXMLDOMDocumentFragment	Document-Fragment	Steht für ein Objekt, das verwendet wird, um Cut & Paste-Operationen in einem Knotenbaum vornehmen zu können.
IXMLDOMDocumentType	DocumentType	Enthält Informationen, die mit der Dokumenttyp-Deklaration zusammenhängen.

Tabelle 10.5 Liste der zusätzlichen Schnittstellen in MSXML

MSXML-Name	DOM-Name	Bedeutung
IXMLDOMElement	Element	Repräsentiert ein Element.
IXMLDOMEntity	Entity	Repräsentiert eine geparste oder ungeparste Entität in einem XML-Dokument.
IXMLDOMEntityReference	Entity-Reference	Repräsentiert eine Entitätsreferenz.
IXMLDOMImplementation	DOMImple-mentation	Erlaubt – unabhängig von einer Dokumentinstanz – die Prüfung, ob die DOM-Implementierung bestimmte Schnittstellen unterstützt.
IXMLDOMNotation	Notation	Enthält eine Notation, die in einer DTD oder einem XML-Schema deklariert worden ist.
IXMLDOMProcessing-Instruction	Processing-Instruction	Repräsentiert eine Verarbeitungsanweisung.
XMLSchemaCache	–	Repräsentiert einen Satz von Namensraum-URIs.
IXMLDOMSchema-Collection/ XMLSchemaCache	–	Repräsentiert ein XMLSchemaCache-Objekt.
IXMLDOMSchema-Collection2/ XMLSchemaCache	–	Ist eine Erweiterung von IXMLDOM-SchemaCollection.
IXMLDOMSelection	–	Repräsentiert eine Knotenliste, die zu einem XPath-Ausdruck passt.
IXMLDOMText	Text	Repräsentiert den Textinhalt eines Elements oder Attributs.
IXSLProcessor	–	Wird für die Ausführung von Transformationen mit geladenen XSLT-Templates benutzt.
IXSLTemplate	–	Repräsentiert ein geladenes XSL-Stylesheet.

Tabelle 10.5 Liste der zusätzlichen Schnittstellen in MSXML (Forts.)

10.4.2 Erweiterungen für Laden und Speichern

Für die praktische Nutzung von DOM sind insbesondere die Erweiterungen der `document`-Schnittstelle wichtig, die das Einlesen und Speichern von XML-Dokumenten betreffen, wie in Tabelle 10.6 gezeigt.

Erweiterung	Bedeutung
`load`	Lädt ein XML-Dokument von dem angegebenen Ort.
`loadXML`	Lädt die angegebenen Zeichendaten als XML-Dokument.
`save`	Speichert ein XML-Dokument am angegebenen Ort.

Tabelle 10.6 Erweiterungen der »document«-Schnittstelle

10.4.3 Erweiterungen der Node-Schnittstelle

Ansonsten kann der größte Teil des Zugriffs auf die Daten des XML-Dokuments über die Abfrage der Eigenschaften und die Nutzung der Methoden der `IXMLDOMNode`-Schnittstelle geleistet werden, die ebenfalls gegenüber der `Node`-Schnittstelle des W3C erweitert worden ist. Tabelle 10.7 listet die Attribute und Tabelle 10.8 die Methoden von `IXMLDOMNode` auf. Die Erweiterungen gegenüber dem DOM-`Node` des W3C sind jeweils mit dem *-Zeichen gekennzeichnet.

Attribut	Beschreibung
`attributes`	Liefert eine Liste der Attribute des Knotens.
`baseName*`	Gibt den lokalen Namen eines durch einen Namensraum qualifizierten Namens zurück, also beispielsweise `farbe`, wenn das Element `<form:farbe>` ist.
`childNodes`	Gibt eine `IXMLDOMNodeList` zurück, deren Länge von der Anzahl der Kindknoten abhängt.
`dataType*`	Spezifiziert den Datentyp für diesen Knoten. Kann nur bei DTDs angewendet werden.
`definition*`	Gibt die Definition des Knotens in der DTD oder dem XML-Schema zurück.
`firstChild`	Enthält den ersten Kindknoten dieses Knotens.
`lastChild`	Gibt den letzten Kindknoten zurück.
`namespaceURI*`	Gibt den URI für den verwendeten Namensraum zurück.

Tabelle 10.7 Attribute von IXMLDOMNode

Attribut	Beschreibung
`nextSibling`	Enthält den nächsten Geschwisterknoten dieses Knotens.
`nodeName`	Gibt den qualifizierten Namen zurück, wenn es sich um einen Knoten für ein Element, ein Attribut, einen Dokumenttyp, eine Entität oder eine Notation handelt. Alle anderen Knotentypen geben einen fixierten String zurück (siehe Tabelle 10.2 in Abschnitt 10.2.6, »Zugriff über Namen«).
`nodeType`	Gibt den DOM-Knotentyp in Form eines numerischen Codes an (siehe Tabelle 10.2 in Abschnitt 10.2.6).
`nodeTypedValue*`	Enthält den Wert des Knotens, ausgedrückt in dem für den Knoten definierten Datentyp.
`nodeTypeString*`	Gibt den Knotentyp in Form eines Strings zurück, beispielsweise `element` oder `attribute`.
`nodeValue`	Enthält den Text, der dem Knoten zugeordnet ist.
`ownerDocument`	Gibt die Wurzel des Dokuments zurück, das den Knoten enthält.
`parentNode`	Enthält den Elternknoten.
`parsed*`	Gibt an, ob der Knoten und die dazugehörigen Unterknoten bereits geparst sind oder nicht.
`prefix*`	Gibt das Namensraumpräfix zurück.
`previousSibling`	Enthält den vorherigen Geschwisterknoten dieses Knotens.
`specified*`	Wird bei Attributknoten verwendet, um anzugeben, ob der Attributwert explizit eingetragen oder vom Default-Wert einer DTD oder eines Schemas stammt.
`text*`	Repräsentiert den Textinhalt eines Knotens oder den verketteten Textinhalt des Knotens und seiner Abkömmlinge.
`xml*`	Enthält die XML-Repräsentation des Knotens und all seiner Abkömmlinge in Form eines Unicode-Strings.

Tabelle 10.7 Attribute von IXMLDOMNode (Forts.)

Methoden	Beschreibung
appendChild	Fügt einen neuen Kindknoten als letztes Kind des Knotens an.
cloneNode	Erzeugt einen neuen Knoten aus einer Kopie dieses Knotens.
hasChildNodes	Ist *wahr*, wenn der Knoten Kinder hat.
insertBefore	Fügt einen Kindknoten genau vor dem angegebenen Knoten oder am Ende der Knotenliste ein.
removeChild	Entfernt den angegebenen Kindknoten und liefert ihn zugleich als Ergebnis.
replaceChild	Ersetzt den angegebenen alten Knoten durch einen neuen Knoten.
selectNodes*	Wendet das angegebene Suchmuster auf den aktuellen Knotenkontext an und gibt eine Liste der passenden Knoten als IXMLDOMNodeList aus.
selectSingleNode*	Wendet das angegebene Suchmuster auf den aktuellen Knotenkontext an und liefert den ersten Knoten, der dazu passt.
transformNode*	Verarbeitet diesen Knoten und seine Kinder mit Hilfe des angegebenen XSLT-Stylesheets und liefert die entsprechende Transformation, die sich daraus ergibt.
transformNodeTo-Object*	Verarbeitet diesen Knoten und seine Kinder mit Hilfe des angegebenen XSLT-Stylesheets und gibt die entsprechende Transformation in dem Ergebnisdokument aus.

Tabelle 10.8 Methoden von IXMLDOMNode

10.5 Fingerübungen mit DOM

Wir können in diesem Abschnitt nur einige kleine Demonstrationen unterbringen, die zeigen sollen, wie mit DOM in der Praxis gearbeitet werden kann. Zugang zur Dokumentation von MSXML finden Sie über *msdn.microsoft.com/en-us/library/ms763742(VS.85).aspx*. Sie enthält eine ausführliche Referenz zu allen Objekten, Eigenschaften, Methoden und Ereignissen von MSXML DOM, mit zahlreichen Codebeispielen, meist gleich in Varianten für C/C++ und JavaScript.

MSXML soll auf den folgenden Seiten mit Hilfe von JavaScript genutzt werden, also innerhalb von HTML-Seiten, so dass Sie gleich die Ergebnisse im Internet Explorer testen können. Beachten Sie, dass über die Internetoptionen die Ausführung von Scripts und ActiveX-Steuerelementen zugelassen ist. Der Microsoft Edge-Browser

unterstützt MSXML nicht mehr, Sie können aber über die Edge-Option MIT INTERNET EXPLORER ÖFFNEN zum Ziel kommen.

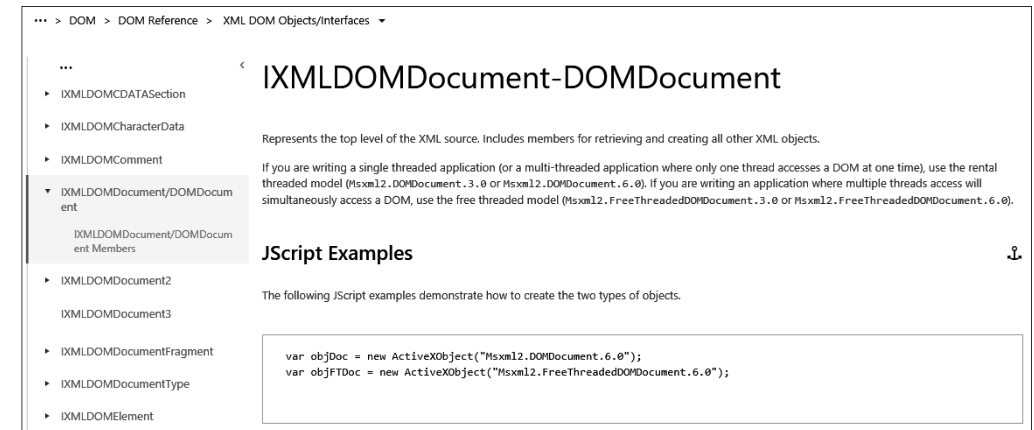

Abbildung 10.4 MSDN-Dokumentation zu MSXML

10.5.1 Daten eines XML-Dokuments abfragen

Das erste Beispiel zeigt, wie aus dem oben aufgeführten Lagerbestandsdokument die Daten eines bestimmten Artikels abgefragt werden können. Dafür ist in der HTML-Seite ein kleines Formular vorgesehen, das die Artikelbezeichnung abfragt, ein entsprechendes Skript zur Datenabfrage aufruft und dann die gefundenen Werte ausgibt.

Das Skript soll dazu eine Funktion `XMLDokumentAbfragen()` zur Verfügung stellen, die das XML-Dokument zunächst einliest und daraus einen DOM-Baum aufbaut. An diese Funktion wird als Parameter die Bezeichnung des gesuchten Artikels übergeben.

Am Anfang werden einige Variablen deklariert, die Mehrzahl zum Zwischenspeichern der von den DOM-Knoten abgelesenen Attributwerte.

```
<script language="JavaScript">

    function XMLDokumentAbfragen(artikelbezeichnung)
    {
    var xmldoc, lagerKnoten, artikelKnoten, bezeichnungKnoten;
    var bestandKnoten, preisKnoten, absatzKnoten;
    var knotenListe;

    xmldoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
    xmldoc.load("lagerdaten.xml");
    ...
```

Einlesen des XML-Dokuments

Zum Laden des XML-Dokuments wird zunächst eine Instanz für ein neues `DOMDocument`-Objekt erzeugt, die an eine Objektvariable `xmldoc` gebunden wird, um das Objekt im weiteren Code einfacher ansprechen zu können.

Da in diesem Fall ein *ActiveXObject* zum Einsatz kommt, wird eine entsprechende *ProgID* benötigt. `Msxml2` liefert dabei den Projektnamen, `6.0` gibt die MSXML-Version an (die Versionen ab 3.0 werden zum Projekt `Msxml2` gezählt). Die etwas verwirrende Schreibweise mit den hinzugefügten Projekt- und Versionsnummern von MSXML hängt damit zusammen, dass MSXML 6.0 auch parallel zu älteren MSXML-Versionen genutzt werden kann. Deshalb müssen die verwendeten Klassen eindeutig über eine entsprechende ProgID identifiziert werden.

Abbildung 10.5 Anzeige der eingelesenen Daten

Die neue Objektvariable `xmldoc` wird anschließend mit Hilfe der oben schon angesprochenen `load`-Methode mit dem vollständigen Inhalt des XML-Dokuments initialisiert. Statt des einfachen lokalen Pfads, den wir hier verwenden, kann auch ein URI

angegeben werden. Der Parser deserialisiert die Daten aus dem XML-Dokument als Vorgabe asynchron; Anwendungen müssen also nicht unbedingt warten, bis der Aufbau des DOM-Baumes im Speicher abgeschlossen ist. Soll diese Vorgabe außer Kraft gesetzt werden, müsste vorher die Zeile

```
xmldoc.async = false
```

eingefügt werden. Wenn Sie für Testzwecke hinter `load` die Zeile

```
alert(xmldoc.xml)
```

einfügen, zeigt das Skript bei der Ausführung die vollständigen Daten an.

Start mit dem Wurzelelement

Im nächsten Schritt geht es darum, nach dem Parsen des gesamten Dokuments Zugriff auf die einzelnen Knoten zu erhalten. Dazu muss zunächst der Knoten aufgesucht werden, der das Wurzelelement des Dokuments liefert, in diesem Fall also das Element `<Lager>`. Dies geschieht mit der Zeile

```
lagerKnoten = xmldoc.documentElement;
```

Es wird also bei dem übergeordneten Dokumentknoten der Wert der Eigenschaft `documentElement` abgefragt. Damit ist der Zugangspunkt zu den untergeordneten Elementknoten erreicht. Da unser Skript nur die Daten eines bestimmten Artikels ausgeben soll, wird die Bezeichnung des Artikels beim Aufruf des Skripts als Parameter an die Funktion übergeben.

Suche nach einem bestimmten Element

Im nächsten Schritt muss nun der Artikelknoten angesteuert werden, dessen Element `<Bezeichnung>` dem eingegebenen Artikelnamen entspricht. Dafür wird eine Schleife benötigt, die die Knotenliste der Artikel abarbeiten kann, um die Werte zu vergleichen. Um diese Knotenliste zu gewinnen, wird die Eigenschaft `childNodes` verwendet, die eine `IXMLDOMNodelist` zurückgibt.

```
knotenListe= lagerKnoten.childNodes;
```

Auf die Knoten in dieser Liste kann über Indizes zugegriffen werden, die von 0 aus zählen. Die Anzahl der Knoten in der Liste kann über die Eigenschaft `length` abgefragt werden. Dieser Wert wird benötigt, um die `for`-Schleife zu stoppen. In der Schleife wird die Übereinstimmung mit dem im Formular eingegebenen Artikelnamen geprüft.

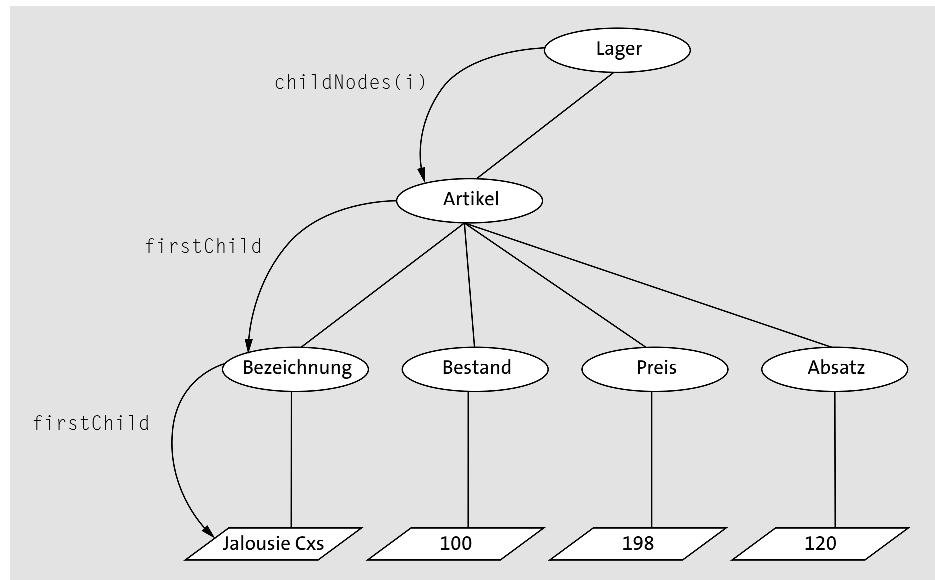

Abbildung 10.6 Ein paar Schritte im Knotenbaum führen zur Artikelbezeichnung.

Beachten Sie, dass in der `if`-Bedingung das doppelte Gleichheitszeichen verwendet wird, um die Übereinstimmung der Artikelbezeichnung zu prüfen. Das einfache Gleichheitszeichen nutzt JavaScript nur für die Wertzuweisung, nicht für den Wertvergleich.

```
for (var i = 0; i < knotenListe.length; i++)
{
  if  (lagerKnoten.childNodes(i).firstChild.firstChild.
       nodeValue == artikelbezeichnung)
```

Um nun an den Text des Elements `<Bezeichnung>` heranzukommen, wird mit Hilfe des Attributs `firstChild` der Knotenbaum abwärts durchgegangen, wie Abbildung 10.6 verdeutlicht.

Auswertung der Daten

Der Text der Artikelbezeichnung wird schließlich über `nodeValue` zurückgegeben. Wenn ein Textknoten gefunden ist, der dem angegebenen Artikelnamen entspricht, kann über den in der `for`-Schleife erreichten Indexwert dann genau auf den gesuchten Artikelknoten zugegriffen werden.

```
artikelKnoten = lagerKnoten.childNodes(i);
```

Von hier aus geht die Reise weiter. Zunächst wird das erste Kind des Artikelknotens angesteuert, dann die drei Geschwisterknoten dieses Kindknotens.


```
bezeichnungKnoten = artikelKnoten.firstChild;
bestandKnoten = bezeichnungKnoten.nextSibling;
preisKnoten = bestandKnoten.nextSibling;
absatzKnoten = preisKnoten.nextSibling;
```

Danach werden die Knotenwerte der verschiedenen Textknoten an die im Formular vorgesehenen Ausgabeelemente übergeben.

```
document.abfrage.bestand.value =
 bestandKnoten.firstChild.nodeValue;
document.abfrage.preis.value =
 preisKnoten.firstChild.nodeValue;
document.abfrage.absatz.value =
 absatzKnoten.firstChild.nodeValue;
```


Damit die Schleife nicht weiter durchlaufen wird, nachdem der gesuchte Artikel gefunden ist, steht am Ende der Anweisungen zu `if` noch die Anweisung `break`.

Das Skript wird aus dem Formular heraus durch Anklicken einer Schaltfläche gestartet, nachdem zunächst die Artikelbezeichnung abgefragt worden ist.

Datenausgabe

Anschließend werden die Daten für den gefundenen Artikel mit Hilfe von benannten `<input>`-Elementen angezeigt, wobei das Attribut `readonly` verwendet wird. Um die Beschriftungen und die Textfelder gleichmäßig auszurichten, wird noch ein kleines Stylesheet für das Element `<label>` definiert.

Daten eines Artikels abfragen

Artikelbezeichnung: Sunset [Daten anzeigen]
Bestand: 200
Preis: 120
Absatz: 200

Abbildung 10.7 Die gesuchten Daten zu einem Artikel

Der Quellcode sieht insgesamt so aus:

```
<html>
  <head>
    <meta http-equiv="content-type"
          content="text/html;charset=UTF-8">
    <title>Daten eines bestimmten Elements anzeigen
    </title>
    <script language="JavaScript">
```

```
    function XMLDokumentAbfragen(artikelbezeichnung)
    {
      var xmldoc, lagerKnoten, artikelKnoten, bezeichnungKnoten;
      var bestandKnoten, preisKnoten, absatzKnoten;
      var knotenListe, gefunden;
      xmldoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
      xmldoc.load("lagerdaten.xml");
      lagerKnoten = xmldoc.documentElement;
      knotenListe= lagerKnoten.childNodes;
      for (var i = 0; i < knotenListe.length; i++)
      {
        if  (lagerKnoten.childNodes(i).firstChild.firstChild.
             nodeValue == artikelbezeichnung)
        {
          gefunden = i;
          artikelKnoten = lagerKnoten.childNodes(gefunden);
          bezeichnungKnoten = artikelKnoten.firstChild;
          bestandKnoten = bezeichnungKnoten.nextSibling;
          preisKnoten = bestandKnoten.nextSibling;
          absatzKnoten = preisKnoten.nextSibling;
          document.abfrage.bestand.value = bestandKnoten.
            firstChild.nodeValue;
          document.abfrage.preis.value = preisKnoten.firstChild.
            nodeValue;
          document.abfrage.absatz.value = absatzKnoten.firstChild.
            nodeValue;
        break;
       }
      }
    }

  </script>
  <style type="text/css" media="screen"><!--

   label {
     font-size: medium;
     width: 4cm }

  --></style>

 </head>
```



```
  <body>
    <form name="abfrage">
      <h2>Daten eines Artikels abfragen</h2><br>
      <label>Artikelbezeichnung: </label>
      <input type="text" name="artbez" size="13">
      <input type="button" value="Daten anzeigen"
       onclick="XMLDokumentAbfragen(document.abfrage.artbez.value)">
      <br>
      <label>Bestand: </label>
      <input type="text" name="bestand" readonly size="6"><br>
      <label>Preis: </label>
      <input type="text" name="preis" readonly size="10"><br>
      <label>Absatz: </label>
      <input type="text" name="absatz" readonly size="6"><br>
    </form>
  </body>

</html>
```

Listing 10.4 dom_001.html

10.5.2 Zugriff über Elementnamen

Mit `Node`-Attributen wie `firstChild`, `lastChild`, `nextsibling` oder `childNodes`, die im letzten Beispiel verwendet worden sind, lässt sich der Knotenbaum vom Wurzelknoten aus bis zu jedem beliebigen Elementknoten durchwandern. Attribute wie `parentNode` und `previousSibling` erlauben es, den Knotenbaum auch in umgekehrter Richtung von jedem beliebigen Elementknoten aus wieder hinaufzuklettern. Die Abfolge ist durch die *Dokumentreihenfolge* bestimmt, die die Ordnung des Baumes regelt.

Eine Alternative zu diesem Schritt-für-Schritt-Verfahren ist die Verwendung der `getElementsByTagName`-Methode des `DOMDocument`-Objekts. Diese Methode liefert ohne Zwischenschritte direkt eine Sammlung von Elementen, die sich über den angegebenen Namen identifizieren lassen. Das Ergebnis ist ein `IXMLDOMNodeList`-Objekt. Die einzelnen Knoten dieser Liste lassen sich über einen Index, der mit 0 startet, wahlfrei ansprechen.

In unserem Beispiel könnte mit Hilfe dieser Methode gleich nach dem Laden der Lagerdaten eine Liste der Artikel erzeugt werden, um darin den gewünschten Artikel aufzuspüren. Der vordere Abschnitt des Skripts kann dann so aussehen:

```
function XMLDokumentAbfragen(artikelbezeichnung)
{
   var xmldoc, lagerKnoten, artikelKnoten, bezeichnungKnoten;
   var bestandKnoten, preisKnoten, absatzKnoten;
   var knotenListe;
   xmldoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
   xmldoc.load("lagerdaten.xml");
   knotenListe = xmldoc.getElementsByTagName("Artikel");
   for (var i = 0; I < knotenListe.length; i++)
   {
    if (knotenListe.item(i).firstChild.firstChild.nodeValue ==
        artikelbezeichnung)
    {
      artikelKnoten = knotenListe.item(i);
```

Listing 10.5 dom_002.html

Die einzelnen Knoten der über die Suche mit dem Elementnamen »Artikel« erzeugten Knotenliste werden in diesem Fall über die `item()`-Methode angesteuert.

10.5.3 Zugriff auf Attribute

Vielleicht ist es Ihnen aufgefallen, dass wir bei der Ausgabe der Artikeldaten bisher die beiden Attribute vernachlässigt haben. Es ist oben schon einiges zur Sonderstellung der Attribute im DOM-Baum gesagt worden. Wie lassen sich die Attributwerte in unserem Beispiel aber in die Ausgabe einfügen?

Die beiden Attribute `nr` und `wg` sind Eigenschaften des Artikelknotens und keine Kinder desselben. Zunächst muss also der Elementknoten für den Artikel angesteuert werden, und von dort aus lässt sich mit der `Node`-Eigenschaft `attributes` eine Liste der dem Element zugeordneten Attribute und ihrer Werte erzeugen.

Diese Liste ist ein Objekt vom Typ `IXMLDOMNamedNodeMap`, das der DOM-Schnittstelle `NamedNodeMap` entspricht. Diese Schnittstelle wiederum macht es möglich, auf die Attributwerte mit Hilfe der `getNamedItem`-Methode zuzugreifen. Die Erweiterungen der Abfragefunktion sehen so aus:

```
<script language="JavaScript">

  function XMLDokumentAbfragen(artikelbezeichnung)
  {
    ...
    var artikelAttribute;
    ...
    artikelKnoten = lagerKnoten.childNodes(i);
```



```
    artikelAttribute = artikelKnoten.attributes
    ...
    document.abfrage.nr.value =
        artikelAttribute.getNamedItem("nr").value;
    document.abfrage.wg.value =
        artikelAttribute.getNamedItem("wg").value;
    ...
  }

</script>
<form name="abfrage">
 ...
<label>Nr: </label>
  <input type="text" name="nr" readonly size="6"><br>
<label>Warengruppe: </label>
  <input type="text" name="wg" readonly size="10"><br>
...
```

Listing 10.6 dom_003.html

Der Internet Explorer zeigt das Ergebnis innerhalb des Formulars an, wie in Abbildung 10.8 dargestellt.

Daten eines Artikels abfragen
Artikelbezeichnung: Sunset [Daten anzeigen]
Nr: 7999
Warengruppe: Rollo
Bestand: 200
Preis: 120
Absatz: 200

Abbildung 10.8 Die kompletten Daten zu einem Artikel

10.5.4 Abfrage über einen Attributwert

Nachdem diese Hürde genommen ist, liegt es nahe, statt der Abfrage über die Artikelbezeichnung eine Abfrage über die Artikelnummer einzurichten. Dazu muss der Funktion die Artikelnummer als Parameter übergeben werden. Für jeden Artikelknoten wird geprüft, ob die gewählte Artikelnummer mit der Artikelnummer des aktuellen Knotens übereinstimmt. Ist das der Fall, werden die Daten des gefundenen Knotens ausgewertet.

Hier nun die geänderte Funktion und das geänderte Formular, das zunächst die Artikelnummer abfragt:

```
<script language="JavaScript">

  function XMLDokumentAbfragen(artikelnr)
  {
    var xmldoc, lagerKnoten, artikelKnoten, bezeichnungKnoten;
    var bestandKnoten, preisKnoten, absatzKnoten;
    var attribute, artikelAttribute, knotenListe;
    xmldoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
    xmldoc.load("lagerdaten.xml");
    lagerKnoten = xmldoc.documentElement;
    knotenListe = lagerKnoten.childNodes
    for (var  i = 0; i < knotenListe.length; i++)
    {
      attribute = lagerKnoten.childNodes(i).attributes;
      if (attribute.getNamedItem("nr").value == artikelnr)
      {
        artikelKnoten = lagerKnoten.childNodes(i);
        artikelAttribute = artikelKnoten.attributes;
        bezeichnungKnoten = artikelKnoten.firstChild;
        bestandKnoten = bezeichnungKnoten.nextSibling;
        preisKnoten = bestandKnoten.nextSibling;
        absatzKnoten = preisKnoten.nextSibling;
        document.abfrage.wg.value =
          artikelAttribute.getNamedItem("wg").value;
        document.abfrage.bezeichnung.value = bezeichnungKnoten.
          firstChild.nodeValue;
        document.abfrage.bestand.value = bestandKnoten.firstChild.
          nodeValue;
        document.abfrage.preis.value = preisKnoten.firstChild.
          nodeValue;
        document.abfrage.absatz.value = absatzKnoten.firstChild.
          nodeValue;
        break;
      }
    }
  }

</script>
...
<form name="abfrage">
  <h2>Daten eines Artikels abfragen</h2><br>
```



```
  <label>Artikelnummer: </label>
  <input type="text" name="artikelnr" size="13">
  <input type="button" value="Daten anzeigen"
    onclick="XMLDokumentAbfragen(document.abfrage.artikelnr.value)">
  <br>
  <label>Artikel: </label>
  <input type="text" name="bezeichnung" readonly size="25">
  <br>
  ...
</form>
```

Listing 10.7 dom_004.html

Daten eines Artikels abfragen

Artikelnummer: 8444 [Daten anzeigen]
Artikel: Markise Blue Sk
Warengruppe: MK
Bestand: 160
Preis: 287,5
Absatz: 80

Abbildung 10.9 Abfrage über die Artikelnummer

10.5.5 Fehlerbehandlung

Bei den bisher gelisteten Skriptfunktionen haben wir uns um den Ausnahmefall zunächst nicht gekümmert, um die Sache so übersichtlich wie möglich zu halten. Aber wenn eine Datei eingelesen wird, kann es immer auch zu Fehlern kommen. Die angegebene Datei ist vielleicht irrtümlich gelöscht worden, oder eine Netzverbindung ist gerade nicht verfügbar. Wie also mit möglichen Fehlern umgehen?

Wir haben schon erwähnt, dass die Spezifikation für DOM mit `DOMException` eine Ausnahmebehandlung definiert hat, die mit vorgegebenen Fehlercodes arbeitet. Die MSXML-Implementierung verwendet dagegen für die Fehlerbehandlung ein spezielles `IXMLDOMParseError`-Objekt, das im Fehlerfall einen Fehlercode zur Auswertung liefert.

Neben dem Fehlercode lassen sich aber noch weitere Informationen zu einem Fehler über die anderen Attribute des `IXMLDOMParseError`-Objekts abfragen, die hilfreich bei der Suche nach der Ursache sein können.

Der folgende Code liefert zum Beispiel, wenn die Daten nicht geladen werden können, einen Grund – reason – für den Fehler, dass die angegebene Ressource nicht auffindbar ist:

```
xmldoc.load("lagerdaten.xml");
if (xmldoc.parseError.errorCode != 0)
  {
    alert ("Fehler beim Einlesen des XML-Dokuments " + xmldoc.
      parseError.reason);
  }
```

Tabelle 10.9 zeigt alle Eigenschaften des IXMLDOMParseError-Objekts.

Eigenschaft	Bedeutung
errorCode	Fehlernummer des letzten Parser-Fehlers. 0 bedeutet, dass kein Fehler vorliegt.
filepos	Gibt die absolute Dateiposition an, an der sich der Fehler ereignete.
line	Gibt die Zeilennummer an, bei der der Fehler aufgetreten ist.
linepos	Gibt die Zeichenposition innerhalb der Zeile mit dem Fehler an.
reason	Erklärt die Ursache des Fehlers.
srcText	Quelltext der Zeile, die den Fehler verursacht hat
url	URL des XML-Dokuments, das den Fehler enthält

Tabelle 10.9 Eigenschaften von »IXMLDOMParseError«

10.5.6 Neue Knoten einfügen

Das DOM erlaubt nicht bloß das Ablesen der Inhalte des Knotenbaums, Sie können auch neue Knoten hinzufügen oder Werte von bestehenden Knoten ändern. Dabei werden Methoden des DOMDocument-Objekts verwendet wie createElement, createAttribute, createTextNode etc., um neue Knoten zu erzeugen. Mit Hilfe allgemeiner Node-Methoden wie appendChild, insertBefore etc. lassen sich diese neuen Knoten anschließend in den Knotenbaum einbauen.

Um die Stelle zu finden, wo der jeweilige Knoten vorgesehen ist, wird mit Hilfe der Attribute firstChild, lastChild, nextSibling etc. im Baum herumgeklettert. Es ist dafür nicht ungünstig, wenn Sie zur Orientierung eine grafische Darstellung des DOM-Baumes als Landkarte verwenden können, sonst kann es schon einmal vorkommen, dass die Verwandtschaftsbeziehungen zwischen den Knoten etwas durcheinandergeraten sind und ein neuer Knoten an der falschen Stelle eingehängt wird.

Das folgende Beispiel soll dafür sorgen, dass neue Artikel im Lagerbestand eingegeben werden können. Wir bleiben bei dem bisher verwendeten Zugang über den Internet Explorer, obwohl das in diesem Fall den Nachteil hat, dass die save-Methode, mit der der DOM-Baum wieder in eine Datei zurückgeschrieben, also serialisiert werden kann, nicht anwendbar ist, da die üblichen Sicherheitseinstellungen eines Browsers das Speichern nicht zulassen.

Die gestellte Aufgabe ist im Prinzip eine typische Problemstellung für eine dynamische Webseite unter Verwendung von Active Server Pages (ASP), Java Server Pages (JSP) oder PHP. Da die Beschreibung solcher Lösungen aber den Rahmen dieser Einführung völlig sprengen würde, begnügen wir uns hier mit einem kleinen Trick, um wenigstens kontrollieren zu können, ob die neuen Knoten richtig eingefügt werden. Über das Formular für die Dateneingabe wird der Inhalt des erweiterten DOM-Baumes an ein winziges ASP-Skript übergeben, das in einem virtuellen Verzeichnis eines Webservers unter IIS installiert ist. Dieses Skript gibt die kompletten Daten dann als Antwort an den Browser zurück, der sie als XML-Daten anzeigt.

Hier zunächst die vollständige HTML-Seite mit dem Skript und dem Formular:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;
     charset=UTF-8">
    <title>Neuen Artikel einfügen</title>
    <script language="JavaScript">

    function XMLDokumentEingabe()
    {
      var xmldoc, lagerKnoten;
      var neuerKnoten, aktKnoten, neuesAttribut, neuerText,
       namedNodeMap;
      xmldoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
      xmldoc.load("lagerdaten.xml");

      lagerKnoten = xmldoc.documentElement;
      neuerKnoten = xmldoc.createElement("Artikel");
      aktKnoten = lagerKnoten.appendChild(neuerKnoten);

      neuesAttribut = xmldoc.createAttribute("nr");
      neuesAttribut.value = String(document.eingabe.nr.value);
      namedNodeMap = aktKnoten.attributes
      namedNodeMap.setNamedItem(neuesAttribut);

      neuesAttribut = xmldoc.createAttribute("wg");
```

```
      neuesAttribut.value = String(document.eingabe.wg.value);
      namedNodeMap = aktKnoten.attributes
      namedNodeMap.setNamedItem(neuesAttribut);

      neuerKnoten = xmldoc.createElement("Bezeichnung");
      aktKnoten = lagerKnoten.lastChild.appendChild(neuerKnoten);
      neuerText = xmldoc.createTextNode(document.eingabe.artbez.value);
      aktKnoten.appendChild(neuerText);

      neuerKnoten = xmldoc.createElement("Bestand");
      aktKnoten = lagerKnoten.lastChild.appendChild(neuerKnoten);
      neuerText = xmldoc.createTextNode(document.eingabe.bestand.value);
      aktKnoten.appendChild(neuerText);
      neuerKnoten = xmldoc.createElement("Preis");
      aktKnoten = lagerKnoten.lastChild.appendChild(neuerKnoten);
      neuerText = xmldoc.createTextNode(document.eingabe.preis.value);
      aktKnoten.appendChild(neuerText);

      neuerKnoten = xmldoc.createElement("Absatz");
      aktKnoten = lagerKnoten.lastChild.appendChild(neuerKnoten);
      neuerText = xmldoc.createTextNode(document.eingabe.absatz.value);
      aktKnoten.appendChild(neuerText);
      document.all.daten.value = xmldoc.documentElement.xml;
      document.eingabe.submit();
    }

  </script>

  <style type="text/css" media="screen"><!--

   label {
     font-size: medium;
     width: 4cm }

  --></style>
</head>
<body>
  <form name="eingabe" action="http://dellprof/xmldaten/
    rueckgabe.asp" method="POST">
    <h2>Neuer Artikel:</h2><br>
    <label>Artikelbezeichnung: </label> <input type="text"
     name="artbez" size="25"><br>
    <label>Artikelnr: </label> <input type="text" name="nr"
```



```
     size="6"><br>
    <label>Warengruppe: </label> <input type="text" name="wg"
     size="10"><br>
    <label>Bestand: </label> <input type="text" name="bestand"
     size="6"><br>
    <label>Preis: </label> <input type="text" name="preis"
     size="10"><br>
    <label>Absatz: </label> <input type="text" name="absatz"
     size="6"><br>
    <input type="hidden" name="daten">
    <input type="button" value="Daten anzeigen"
     onclick="XMLDokumentEingabe()"><br>
  </form>
 </body>
</html>
```

Listing 10.8 lagerdatenerweitern.html

Die Webseite bietet ein Formular für die Eingabe eines neuen Artikels an. Nach Eingabe der Daten wird über die Schaltfläche die Funktion `XMLDokumentEingabe()` aufgerufen.

10.5.7 Neue Elementknoten

Um das Einfügen neuer Knoten zu ermöglichen, werden Variablen deklariert, die wiederholt verwendet werden können:

```
var neuerKnoten, aktKnoten, neuesAttribut, neuerText, namedNodeMap;
```

Zunächst wird ein neuer Elementknoten für einen weiteren Artikel erzeugt:

```
neuerKnoten = xmldoc.createElement("Artikel");
```

Dieser neue Knoten wird unterhalb des Wurzelelements `<lager>` in die Liste der Artikelknoten eingebaut, und zwar am Ende der bestehenden Liste.

```
aktKnoten = lagerKnoten.appendChild(neuerKnoten);
```

Die dafür verwendete Methode `appendChild` liefert den eingefügten neuen Knoten, der der Variablen `aktKnoten` zugewiesen wird, so dass darauf wieder Bezug genommen werden kann.

Statt den neuen Artikel hinter dem bisher letzten Artikel einfach anzuhängen, könnte er zum Beispiel vor dem bisher ersten Artikel eingefügt werden oder auch an einer beliebigen Stelle, wenn eine bestimmte Reihenfolge erwünscht ist. Dafür ist die

Methode `insertBefore` zuständig, die gleich zwei Parameter hat. Der erste gibt an, wer das neue Kind ist, der zweite bezeichnet den Knoten des Kindes, vor dem der neue Knoten eingefügt werden soll. Der Ausdruck

```
aktKnoten = lagerKnoten.insertBefore(neuerKnoten, lagerknoten.firstChild);
```

wäre eine Alternative.

10.5.8 Neue Attributknoten

Für den neuen Artikel werden zwei neue Attributknoten benötigt; auch diese werden zuerst über eine Methode des `DOMDocument`-Knotens erzeugt:

```
neuesAttribut = xmldoc.createAttribute("nr");
neuesAttribut.value = String(document.eingabe.nr.value);
```

Anders als bei den Textknoten von Elementen kann der Attributwert in einem einfachen Verfahren zugewiesen werden. Hier wird die im Formular eingetragene Artikelnummer dem neuen Attribut zugewiesen. Wie aber kann nun das neue Attribut an den Elementknoten gebunden werden? Die Methoden `appendChild` oder `insertBefore` sind ja nicht anwendbar, weil, wie schon beschrieben, Attributknoten keine Kindknoten der Elemente sind, sondern ihnen quasi nur lose, in einer ungeregelten Liste, zugeordnet werden.

Stattdessen wird mit der `Node`-Eigenschaft `attributes` ein `NamedNodeMap`-Objekt für die Attribute des aktuellen Knotens erzeugt. Diesem Objekt kann über die `setNamedItem`-Methode das neue Attribut zugewiesen werden.

```
namedNodeMap = aktKnoten.attributes
namedNodeMap.setNamedItem(neuesAttribut);
```

10.5.9 Unterelementknoten und Textknoten

Innerhalb des neuen Artikelknotens sind nun die einzelnen Elemente an der Reihe. Das Verfahren ist für alle vier Unterknoten gleich. Der neue Elementknoten wird erzeugt und wird diesmal eine Etage tiefer angehängt:

```
neuerKnoten = xmldoc.createElement("Bezeichnung");
aktKnoten = lagerKnoten.lastChild.appendChild(neuerKnoten);
```

Zu jedem Unterelement gehört als Kind ein Textknoten, der einen Wert aus dem Formular aufnehmen soll. Der Textknoten wird gleich mit dem entsprechenden Wert erzeugt und dann an den aktuellen Knoten angehängt:

```
neuerText = xmldoc.createTextNode(document.eingabe.artbez.value);
aktKnoten.appendChild(neuerText);
```


Nachdem alle Unterelemente erzeugt und mit ihren Textwerten eingebaut sind, wird der aktuelle Stand des DOM-Baumes ab dem Wurzelelement an ein im Formular verstecktes Feld übergeben, wobei mit Hilfe der `xml`-Eigenschaft noch dafür gesorgt wird, dass diese Daten als Unicode-String übergeben werden:

```
document.all.daten.value = xmldoc.documentElement.xml;
```

Mit der Methode

```
document.eingabe.submit();
```

werden die Daten in diesem Fall an den Webserver verschickt, ohne dass noch einmal eine Schaltfläche gedrückt werden muss.

10.5.10 Request und Response

Dem Eingabeformular ist, anders als bei den bisherigen Beispielen, eine Aktion als Eigenschaft zugeordnet. Damit wird festgelegt, dass die Daten des Formulars an das schon erwählte kleine ASP-Skript auf dem angegebenen Webserver zur Weiterverarbeitung gereicht werden, und zwar mit der Methode `post`, die dafür sorgt, dass das ASP-Skript die übergebenen Daten wie eine Benutzereingabe erhält und verarbeiten muss:

```
<form name="eingabe" action="http://dellprof/xmldaten/rueckgabe.asp"
  method="POST">
```

Das kleine Skript auf dem Server behandelt die ankommenden Daten als eine Anforderung, einen *Request*, von Seiten des Webbrowsers, also des Clients.

```
##rueckgabe.asp##
<%@ LANGUAGE="VBSCRIPT" %>
<%
Response.ContentType = "text/xml"
Response.Write "<?xml version=" & Chr(34) & "1.0" & Chr(34) & "?>" & Chr(13)
 & Chr(10)
Response.Write Request("daten")
%>
```

Die Aufgabe des Skripts ist es einfach nur, die empfangenen Daten, die die neuen Elemente enthalten, wieder an den Webbrowser zurückzuschicken. Dazu wird das `Response`-Objekt verwendet, das die Methode `Write` zur Verfügung stellt. Vorher wird noch der MIME-Typ der Daten mit `"text/xml"` festgelegt und eine Zeile mit der XML-Deklaration vorangestellt.

Neuer Artikel:

Artikelbezeichnung: Rollo Venice
Artikelnr: 5322
Warengruppe: Rollo
Bestand: 100
Preis: 120
Absatz: 20
Daten anzeigen

Abbildung 10.10 Die Maske mit Daten für einen neuen Artikel

Der Internet Explorer zeigt das erweiterte XML-Dokument mit dem angehängten Artikel in Form einer wohlgeformten XML-Datei an, die Sie, wenn Sie wollen, auch aus dem Internet Explorer heraus mit der Dateiendung *.xml* lokal speichern können.

```
<?xml version="1.0" ?>
- <Lager>
  + <Artikel nr="7777" wg="Jalou">
  + <Artikel nr="7778" wg="Jalou">
  + <Artikel nr="5554" wg="Rollo">
  + <Artikel nr="7999" wg="Rollo">
  + <Artikel nr="8444" wg="MK">
  - <Artikel nr="5322" wg="Rollo">
      <Bezeichnung>Rollo Venice</Bezeichnung>
      <Bestand>100</Bestand>
      <Preis>120</Preis>
      <Absatz>20</Absatz>
    </Artikel>
  </Lager>
```

Abbildung 10.11 Die Anzeige des erweiterten XML-Dokuments im Internet Explorer

10.6 Alternative zu DOM: Simple API for XML (SAX)

In vielen Fällen ist der Aufbau eines Knotenmodells im Speicher nicht erforderlich, um an die Informationen zu gelangen, die von einer Anwendung benötigt werden. Das gilt immer dann, wenn es möglich ist, ein XML-Dokument einfach ganz oder auch nur bis zu einer bestimmten Stelle sequenziell durchzugehen, um das zu finden, was gesucht wird. Wenn Sie zum Beispiel wissen wollen, wie viele Artikel zu einer bestimmten Warengruppe gehören, ist es nicht notwendig, einen Knotenbaum aufzubauen. Hier bietet es sich als eine mögliche Lösung an, mit *SAX* zu arbeiten.