Elmar Geese
Markus Heiliger

XML mit VB und ASP

Internetlösungen für VB- und Web-Entwickler

Galileo Press

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
Ein Titeldatensatz für diese Publikation
ist bei der Deutschen Bibliothek erhältlich

ISBN 3-934358-50-0

1. Auflage 2000

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch **Eppur se muove** (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Lektorat Judith Stevens, Bonn **Korrektorat** Lisa Alexin, Bonn **Einbandgestaltung** Barbara Thoben, Köln **Titelbild** Barbara Thoben, Köln **Herstellung** Claudia Lucht, Bonn **Satz** reemers publishing services gmbh, Krefeld **Druck und Bindung** Bercker Graphischer Betrieb, Kevelaer

Inhalt

Vorwort

Die Icons in diesem Buch

Wichtige **Hinweise** und **Tips** finden Sie in Abschnitten, die mit diesem Symbol gekennzeichnet sind.

Achtung, Abschnitte mit diesem Symbol sprechen eine **Warnung** aus!

1 XML-Grundlagen

In diesen Kapitel werden die Grundlagen zum Verständnis von XML behandelt. Die wichtigsten XML-Module werden vorgestellt und beispielhafte Anwendungsbereiche erläutert. Es wird kurz auf die Entstehung von XML und die weitere Entwicklung eingegangen.

1.1 Vorbemerkung

Heute ist bereits eine ungeheure Zahl von Ressourcen zu XML verfügbar: Es gibt Dutzende von Büchern, hunderte von Websites mit Tausenden von Dokumenten. Allein die von Microsoft verfügbaren Quellen füllen bereits eine CD. Andere große Softwarefirmen wie SUN oder IBM bieten sehr große Sites mit Informationen und Ressourcen zu XML.

Obwohl daher hinreichend Zugriff auf Basisinformationen zu XML besteht, werden wir auf ein paar Aspekte der XML-Grundlagen eingehen.

- Jede Erweiterung und Neuerung zum Thema XML produziert eine ungeheure Flut von Informationen. Es ist häufig schwer zu erkennen, was Vorschlag ist, oder was bereits zum Standard gehört.
- Die Beschreibungen von XML orientieren sich meist am favorisierten Anwendungsgebiet des jeweiligen Verfassers. Dadurch werden oft mißverständliche oder falsche Ansichten publiziert.
- Praxisbezogene Beschreibungen sind von den verwendeten Softwarewerkzeugen geprägt. Dabei werden leicht spezifische oder auch fehlende Features der Tools mit Features von XML verwechselt.

Wir werden in diesem Kapitel versuchen, einen Überblick über die Aspekte von XML zu geben, die vor allem für Softwareentwickler relevant sind. Das darüber hinaus bestehende und hoffentlich noch wachsende Interesse an XML sollte der Leser anhand der im Web verfügbaren aktuellen Quellen befriedigen. Zu diesem Zweck bietet sowohl die Buch-CD als auch die Website des Buches unter **http://www.tarent.de/xml** aktuelle Informationen und Links zu den interessantesten Quellen.

1.2 Was ist XML?

Das Kürzel XML steht für **EXtensible Markup Language**. Um mit dem am weitesten verbreiteten Irrtum zu beginnen: XML ist, entgegen der landläufigen Meinung, keine Sprache. Es beschreibt die Regeln zur Erzeugung von XML-basierten Auszeichnungssprachen. Seit 1998 wurde eine große Zahl von Sprachen und Spezifikationen zu XML veröffentlicht. Zweck dieser Sprachen ist die Beschreibung von Daten und Dokumenten und deren Verknüpfung, Validierung, Darstellung und Verarbeitung.

1.2.1 Konzept und Ziele von XML

Zunächst einmal geht es bei XML darum, strukturierte Daten in Textdateien zu speichern. Bei der Entwicklung von XML standen folgende Ziele im Vordergrund:

- Es sollte für Menschen wie für Maschinen einfach lesbar sein.
- Es sollte sich für die Speicherung und Übertragung strukturierter Informationen eignen.
- Es sollte Mechanismen für die Suche und Filterung bereitstellen.
- Es sollte (im Gegensatz zu HTML) Inhalt und Darstellung vollkommen trennen.
- Es sollte individuell erweiterbar sein.

Durch die Offenheit des Internets besteht ein steigender Bedarf, Darstellungen an die technischen Möglichkeiten des Clientsystems und an die biologischen oder sozialen Bedingungen des Rezipienten anzupassen. XML bietet die Möglichkeit, Form und Inhalt voneinander zu trennen. So lassen sich unterschiedliche Darstellungen des gleichen Inhalts besser realisieren. So kann der gleiche Inhalt für die Anzeige in einem PC-Web-Browser, einem Info-Terminal oder einem mobilen Gerät aufbereitet werden. Besondere Bedürfnisse der Benutzer sind z. B. die Darstellung für Sehbehinderte und Blinde oder inhaltliche Aggegrationen für Kinder, Hausmänner und Vorstände.

1.2.2 Warum sich XML durchsetzt

Mit Recht beurteilen erfahrene Anwender, Entwickler und Entscheider neue Entwicklungen zurückhaltend oder kritisch. XML hat jedoch von Beginn an eine sehr starke Unterstützung erfahren. Unternehmen wie Microsoft, SUN oder IBM haben jeweils Hunderte von Entwicklern, die sich ausschließlich mit XML-Technologien beschäftigen. Ähnlich wie bei Java spielt hier das Internet eine entscheidende Rolle. Der jahrelange Kampf der Browserkönige Netscape und Microsoft hatte gezeigt, wie kontraproduktiv fehlende Standards sind: Heute noch sind unzählige Webdesigner damit beschäftigt, Webseiten browserkompatibel zu gestalten, weil die Standardisierung nicht mit der technischen Entwicklung und der Verbreitung des Internets Schritt hielt. Das W3C hat daraus gelernt und treibt die Verabschiedung von XML-Standards zügig voran. Die Kernbereiche sind inzwischen verabschiedet, so daß einem Einsatz nichts mehr im Wege steht.

Trotz aller Vorteile und einer sicheren Basis ist XML eine noch recht junge Technologie. Dieses Buch will diesen Entwicklern dabei helfen, sich nicht in der ersten Begeisterung zu verrennen. Man kann nicht immer davon ausgehen, daß alle in den Spezifikationen beschriebenen Features von den XML-Parsern und Prozessoren auch schon unterstützt werden. Die Beispiele dieses Buches sind jedoch alle von den Verfassern erstellt und überprüft.

1.3 Kurze Geschichte der Auszeichnungssprachen

SGML

Auszeichnungssprachen gibt es seit etwa 50 Jahren. Die Idee entstammt dem Druck- und Verlagswesen. Aus dem Auszeichnen von Textpassagen zur Layoutbeschreibung entwickelte sich in diversen Zwischenschritten eine komplexe Sprache namens SGML, die **Standard Generalized Markup Language**. SGML wurde als ISO-Standard normiert.

Einem Masseneinsatz von SGML stand die hohe Komplexität (die Spezifikation umfaßt etwa 500 Druckseiten, die von XML etwa 40) und dadurch der Mangel an entsprechenden Editoren entgegen. Das Internet schuf schließlich durch die Verbreitung von HMTL einen Status Quo. HTML ist einfach zu handhaben, brachte aber den Nachteil mit, daß Darstellung und Inhalt nicht getrennt werden. Hinzu kamen die Schwierigkeiten durch die unterschiedlichen proprietären Erweiterungen von HTML durch verschiedene Browseranbieter. Die Suche nach einer Alternative führte zum Erfolg von XML.

XML ergänzt SGML um die Möglichkeit der individuellen Erweiterung, befreit es von der Verpflichtung, zu jedem Dokument eine Dokumenttypdefinition zu erstellen und führt eine striktere Behandlung der Auszeichnungselemente (Tags) ein. Daher hat es seit seiner ersten Veröffentlichung 1996 eine rapide Verbreitung gefunden.

1.4 Die XML-Sprachfamilie

Es gibt eine Reihe von Modulen, die zum Kern der XML-Technologie gehören. Dies sind zur Zeit:

- XML-1.0
- XSL/XSL Transformations
- XML-Schema, XML-Data
- XPath, XPattern
- XLink/XPointer

XML-1.0

Wie schon erwähnt, ist das XML-1.0-Modul für die Beschreibung der XML-Grammatik zuständig. Es regelt, wie XML-Dokumente ausgezeichnet werden müssen. Alle anderen Module sind an die in XML-1.0 definierten Regeln gebunden. Mehr als wohlgeformte Dateien erstellen kann man mit XML-1.0 alleine jedoch nicht. Die begleitenden Kern-Module setzen sich mit Aspekten wie Formatierung, Strukturbeschreibung, Navigation und Verknüpfung auseinander.

XSL

XSL ist in die Bereiche XSL-Stylesheet und XSL-Transformations gegliedert. **XSL-Stylesheets** können für das Layout von XML-Dateien eingesetzt werden und erweitern die heute bereits durch CSS (Cascading Style Sheets) gegebenen Möglichkeiten. Sowohl CSS-Stylesheets als auch XSL-Stylesheets können mit XML-Dateien verknüpft werden. **XSL-Transformations** definiert Methoden zur Bearbeitung von XML-Dateien. Hier sind z. B. Kontrollstrukturen definiert, die eine sequentielle und rekursive Verarbeitung unterstützen. XSL-Stylesheet-Anweisungen können in die Kontrollstrukturen eingebettet werden.

XPath

Da XML-Dateien hierarchisch organisiert sind, kann in ihnen durch Pfade navigiert werden. Die Navigationsangaben sind nicht in XML codiert, sondern werden als kompakte Ausdrücke formuliert. Die Regeln für diese Ausdrücke sind in **XPath** spezifiziert. In XSL-Transformations wird stark von XPath Gebrauch gemacht. Als XPattern wird häufig die Microsoft-Implementierung von XPath bezeichnet, die in ein paar Punkten von XPath abweicht.

XLink und XPointer

XLink und **XPointer** sind für die Verknüpfung von Dokumenten zuständig. Ziel bei ihrer Entwicklung ist es, die sehr eingeschränkten Linking-Möglichkeiten von HTML zu erweitern. XLink ist für die Adressierung von

Dokumenten, XPointer für die von Dokumentelementen zuständig. Daher setzt XPointer auch weitgehend XPath ein. Die Spezifikation von XLink und XPointer ist noch nicht abgeschlossen.

XML-Schema

Eine sehr wichtige Rolle spielen **Schemata**. Durch die Einführung von **XML-Schema** steht ein einfacher Mechanismus zur Definition von Dokumentklassen zur Verfügung. In Schemas ist festgelegt, welche Daten in welcher Form in XML-Dateien enthalten sind. Eine Applikation (z. B. ein Browser) kann so überprüfen, ob ein Dokument bestimmten Vorgaben in bezug auf Struktur und Daten entspricht. Da **DTDs** (Document Type Definitions) nicht in XML beschrieben sind, gehören sie nicht zur Sprachfamilie. Gleichwohl können sie in XML als Alternative zu Schemas eingesetzt werden.

DTD

Es gibt noch eine ganze Reihe weiterer wichtiger Sprachen, die jedoch für die Erstellung von Webanwendungen nicht unbedingt erforderlich sind, oder deren Spezifikation noch nicht hinreichend abgeschlossen ist. Um hier nicht der Gefahr der Informationsüberflutung zu erliegen, werden wir es zunächst bei den genannten Sprachen belassen.

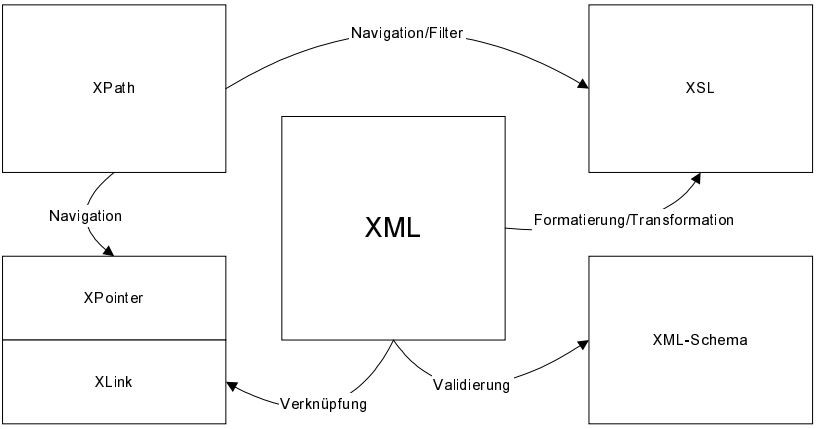

Abbildung 1.1 XML-Module

1.5 Anwendungsbereiche

XML setzt sich in den Bereichen Internet-Anwendungen, E-Business und Content Management zunehmend durch.

XML ist überall dort geeignet,

- wo strukturierte Informationen verwaltet werden
- wo Daten plattformübergreifend übertragen werden sollen. Mit XML lassen sich Dateneinheiten in praktische Pakete packen und verschikken. Dies kann den Transfer zwischen unterschiedlichen Systemen stark vereinfachen. Innerhalb heterogener Systeme können so auch plattformübergreifend einheitliche Standards realisiert werden.
- wo Daten in einer Quelle gehalten, aber in unterschiedlicher Form ausgegeben werden sollen.

Durch die Standardisierung von XML ist sichergestellt, daß unterschiedliche Handelspartner einfach miteinander kommunizieren können. Ein praktisches Beispiel ist ein Bestellvorgang, der allgemeine Bestelldaten sowie eine Stückliste und Artikelbeschreibungen enthält.

Internet-Anwendungen profitieren von der Trennung von Inhalt und Darstellung und von der Möglichkeit der Transformation in HTML. Dadurch ist die XML-Technologie auch für Browser nutzbar, die selbst kein XML unterstützen.

E-Business

E-Commerce und E-Business sind die magischen Wörter des Internets. Besonders in der Business-To-Business-Kommunikation sind aber die proprietären Datenformate der Handelspartner ein großes Problem. Und selbst dort, wo Vereinbarungen über solche Formate bestehen, gestaltet sich die Überprüfung der Daten als aufwendig. Bemühungen um Standards auf diesem Gebiet laufen meist sehr langsam und mit ungesicherten Ergebnissen ab. Daher werden zunehmend vorhandene Standards in XML übertragen, um eine bessere Kommunikation zwischen Handelspartnern umzusetzen. Geschützte Bibliotheken verwalten die Dokumenttyp-Definition, so daß diese aus dem Internet geladen und das Dokument validiert werden kann. Dabei werden sowohl Struktur als auch die Gültigkeit der Eigenschaftswerte geprüft. Unterstützt wird dies unabhängig vom Betriebs- und Anwendungssystem der an der Transaktion beteiligten Partner.

Content Management

Durch die einfachere Implementierung im Vergleich zu SGML setzt sich XML immer stärker im Bereich Content Management durch. Dies wird durch die neuen XML-Datenbanken noch unterstützt. Diese Datenbanken speichern Dokumente als XML-Strukturen, die zu einem Dokument verknüpft sind. Ein Element kann in beliebig vielen Verknüpfungen referenziert werden, dadurch können virtuelle Dokumente erstellt werden. Es besteht direkter Zugriff auf jedes einzelne Dokument-Element.

Strukturierte Objekte

Softwareentwicklung wird in immer stärkeren Maß mit objektorientierten Zielen durchgeführt. Die Modellierung von Anwendungen nach den Objekten der realen Welt macht immer stärker das Dilemma der tabellarischen Datenverwaltung deutlich. Mangels verfügbarer Werkzeuge werden Eigenschaften von Objekten zu Daten flachgeklopft, um diese persistent zu machen. Mit XML wird eine einfach zu beherrschende Technik für eine skrukturierte Persistenzschicht zur Verfügung gestellt. Jeder Software-Entwickler kennt die Problematik im Umgang mit strukturierten Daten. Software-Tools zur Lösung des Problems waren bisher Mangelware. In der Microsoft-Welt wurde in den letzten Jahren der Ansatz der Compound Documents propagiert. Der Nachteil war und ist jedoch die Plattformabhängigkeit und die schwergewichtige Implementierung der COM-Dateien und ihrer Editoren. Es ist anzunehmen, daß dieser Structured-Storage-Ansatz mit der Durchdringung von XML weiter an Bedeutung verliert.

1.6 Verfügbarkeit von XML-Tools

XML ist für die meisten Betriebssysteme verfügbar. Als Softwarekomponenten für die Entwicklung mit XML sind XML-Parser und XSL-Prozessoren erforderlich. Zum Beispiel Microsoft, SUN und IBM stellen XML-Komponenten kostenlos zur Verfügung. Für die Entwicklung unter Windows wird meist die mit dem Internet Explorer 5.0 ausgelieferte COM-Bibliothek MSXML.DLL eingesetzt, die auch einen XSL-Prozessor enthält. Diese Bibliothek hat durch ihre zeitlich sehr frühe Verfügbarkeit eine Art Quasi-Standard in der Windows-Welt erzeugt, der jedoch in einigen Punkten von den späteren W3C-Spezifikationen abweicht. Da XML-Dateien definitionsgemäß wohlstrukturiert sind, sollte eine Konvertierung auf neue Standards kein Problem darstellen. Zudem wird Microsoft seinen Parser erwartungsgemäß abwärtskompatibel halten.

Bei der Veröffentlichung des Internet Explorers 5.0 implementierte Microsoft eine Reihe von XML Features, bevor die entsprechenden Standards verabschiedet waren. Die dadurch vorhandenen Abweichungen von den heutigen Standards haben viele Entwickler verunsichert. Microsoft hat jedoch eine Abwärtskompatibiltät zukünftiger Versionen zugesagt, so daß kaum Gefahr gegeben ist, daß heutige Lösungen in Kürze nicht mehr lauffähig sind. Informationen hierzu hält Microsoft unter **http://msdn.microsoft.com/xml/general/msxmlconform.asp** bereit.

Der Autor verschiedener XML-Spezifikationen und XML-Guru James Clark **(http://www.jclark.com)** stellt einige OpenSource Tools zur Verfügung.

XML-Parser

XML-Parser stehen dem ambitionierten C+- oder Java-Entwickler in vielen Implementierungen zur Verfügung. Wer schnelle Ergebnisse erzielen will oder/und sich sowieso im Microsoft-Umfeld bewegt, ist mit dem Microsoft Parser bestens beraten, weil dieser eine einfach zu implementierende ActiveX-Schnittstelle bereitstellt. Ansonsten findet man auch Parser für die folgenden Entwicklungsumfelder kostenlos im WWW.

- COM-Bibliotheken
- Java-Packages
- Perl
- Python

XSL-Prozessoren Durch die Werkzeuge zur Transformation von XML kann XML-Technologie auch dort eingesetzt werden, wo Endgeräte noch kein XML unterstützen. XSL-Prozessoren sind meist korrespondierend zu den jeweiligen XML-Parsern verfügbar.

Einige Links zu Editoren und Tools finden Sie unter **http://www.ta-rent.de/xml**.

2 Überblick über die XML-Bausteine

In diesem Kapitel wird ein Überblick über die XML-Bausteine geliefert. Es werden die technischen Grundlagen für das Verständnis von XML vermittelt.

2.1 XML-Grundregeln

2.1.1 Wohlgeformte und gültige Dokumente

Das Regelwerk von XML läßt sich in zwei Aufgabengebiete trennen:

- Das eine Aufgabengebiet befaßt sich mit der Struktur, die einem Dokument gegeben sein muß. Man spricht von **wohlgeformten** Dokumenten, wenn diesen Regeln entsprochen wird.
- Das andere Aufgabengebiet befaßt sich mit der Syntax. Wird diesem Teil des Regelwerks entsprochen, spricht man von **validen** Dokumenten.

Dokumente können wohlgeformt sein, aber dennoch syntaktisch invalide. (Der umgekehrte Fall ist natürlich ausgeschlossen.) XML-Parsern ist es möglich, Dokumente zu verarbeiten, die nicht dem gesamten Regelwerk entsprechen – aber wohlgeformt müssen die Dokumente mindestens sein. Auf das Arbeiten mit gültigen Dokumenten wird im Kapitel **DTD und Schema** genauer eingegangen.

2.1.2 Struktur eines XML-Dokuments

XML wird in einer Baumstruktur modelliert.

Nachfolgend die wichtigsten Regeln für den Umgang mit XML:

- Jedes XML-Dokument enthält ein eindeutig benanntes Tag (Root-Tag), das alle anderen datenhaltenden Tags umschließt
- Tags müsssen immer geschlossen werden, im Gegensatz zu HTML, das ›leere‹ Tags zuläßt. Hierbei muß die Benamung des öffnenden und schließenden Tags exakt übereinstimmen (case-sensitive).

```
<Chapter>
</Chapter>
```

- Für Tags, die keinen Wert umschließen, kann anstatt des vorangegangenen Beispiels auch der folgende Shortcut genutzt werden.

```
<Chapter/>
```

- Elemente können geschachtelt werden. Überschneidungen von Elementen sind nicht möglich.

```
<Document>
   <Chapter>XML-Bausteine
      <Heading_2>Der Sprachstandard
      </Heading_2>
   </Chapter>
</Document>
```

- Jedes Element kann Eigenschaften haben.

```
<Document name="XML-Grundwissen" created="01-01-2000">
   <Chapter start="1" end="28240" style="Heading 1">XML Bausteine
      <Heading_2 start="25" end="140" style="Heading 1"> Der
Sprachstandard
      </Heading_2>
   </Chapter>
</Document>
```

- Eigenschaften werden immer in Anführungszeichen eingeschlossen.

```
<Document name="XML-Grundwissen" created="01-01-2000">
</Document>
```

- Eigenschaften können keine Elemente enthalten.
- Reservierte Zeichen können nicht innerhalb des Wertebereichs von Elementen bzw. Attributen genutzt werden.

```
< =&lt;
& =&amp;
> =&gt;
" =&quot;
' =&apos;
```

- Daten, die ungültige, d.h. für XML mißverständliche Zeichen beinhalten, werden in CDATA-Klammern eingeschlossen. Der Parser ignoriert dann die Anweisungen in diesen Blöcken.

2.2 Die Verarbeitung

Die XML-Spezifikation beinhaltet die Vorgaben, nach denen XML verarbeitende Software entwickelt wurde bzw. wird. XML ist für die meisten Betriebssysteme durch native oder Java-Lösungen verfügbar. Diese Basiskomponenten leisten folgende Features:

- Parser lesen und schreiben XML-Strukturen
- Filter selektieren Elemente
- »Prozessoren« interpretieren und transformieren XML-Dateien
- Durch XML-Schemas können Dokumenttypen abgebildet werden
- Parser können XML-Dateien gegen Schemas prüfen

2.3 XSL – Stylesheets

Die **Extensible Stylesheet Language** (XSL) ist für die Aufbereitung der Inhalte von XML-Dokumenten zuständig. Hierbei gibt es keine Beschränkung hinsichtlich des Zielformates. Dies kann sowohl eine Textdatei als auch ein HTML-Dokument sein.

Die XSL-Anweisungen werden zur Laufzeit von einem XSL-Prozessor interpretiert. Dabei werden die Anweisungen auf eine verknüpfte XML-Datei angewendet. Die Zuordnung unterschiedlicher Stylesheets zu einer XML-Datei ermöglicht verschiedene Darstellungen der gleichen Daten.

Durch Filterausdrücke ist es möglich, Untermengen der Daten einer XML-Datei abzubilden.

2.4 XPath

XPath ist die Abfragesprache für XML-Dokumente. In XPath können Filterausdrücke formuliert werden, die Anwendung ermöglicht erst den produktiven Einsatz von XSL sowie die Navigation innerhalb eines XML-Dokuments mit einem XML-Parser.

2.5 DTD

Document Type Definitions (DTD) beschreiben die Struktur, der ein XML-Dokument entsprechen muß, wenn dieses auf ein DTD verweist.

Im Gegensatz zu SGML sind DTDs in XML optional.

Dieselbe Aufgabe wie DTDs können auch XML-Schemas übernehmen.

2.6 Schema

Schemas sind vergleichbar mit DTDs. Sie ermöglichen, daß Dokumente auf ihre Beschaffenheit geprüft werden können. Die Besonderheit ist hierbei, daß Schemas, im Gegensatz zu DTDs, auch offene, d.h. erweiterbare, Modelle beschreiben können.

Ein weiterer großer Vorteil der Schemas ist das Format, in dem sie verfaßt sind. Im Gegensatz zu DTDs liegen sie im XML-Format vor, was zum einen das Erlernen der Syntax erleichtert und zum anderen nicht die Existenz eines entsprechenden Editors voraussetzt. Die Bearbeitung kann in einem XML-Editor erfolgen.

2.7 XLink

Die **XML Linking Language** (XLink) erweitert die Möglichkeiten von XML um die Funktionalität von Links auf nicht im Dokument enthaltene Daten.

```
<Document name="XML Grundwissen" created="01-01-2000">
   <Chapter start="1" end="28240" style="Heading 1">XML Bausteine
      <Heading_2 start="25" end="140" style="Heading 1"> Der
Sprachstandard
      </Heading_2>
      <a xml:link="simple" href="spreadsheet.jpg" show="new"
content-role="basics">basicsspreadsheet
      </a>
   </Chapter>
</Document>
```

Des weiteren werden durch XLink die folgende Arten von Links zur Verfügung gestellt:

- Links, die zu mehreren Zielen führen
- Bi-direktionale Links
- Links für Annotationen, zum Beispiel in schreibgeschützten Dokumenten
- Verknüpfungen zu Datenbanken
- Funktionalitäten, wie sie jetzt (wenn überhaupt) nur durch Skripts realisiert werden können wie ‚expand-in-place', neues Fenster oder Zielrahmen, automatisches Folgen und weitere.

2.8 XPointer

Die **XML Pointer Language** (XPointer) bietet im Vergleich zu XLinks einen detaillierten Zugriff. Hiermit ist nicht nur die Adressierung eines Dokuments, sondern auch die einer bestimmten Stelle im Inhalt des Dokuments möglich.

Sprungmarke für einen XPoint-Link ist hierbei eine ID.

```
<a id="begin-here">This is a sample.</a>
```

Der folgende XPointer-Link, bezogen auf die vorangegangene Codezeile, würde als Ziel den Buchstaben »i« der Wortes »is« haben.

```
id(begin-here).string(2,"i")
```

Duch XPointer werden hinsichtlich XML die folgenden Mehrwerte geschaffen:

- Links, die auf bestimmte Stellen innerhalb von Dokumenten zeigen, ohne daß diese vom Autor gekennzeichnet sein müssen.
- Links können sich dabei sowohl auf einen Bereich beziehen als auch absolut oder relativ angegeben werden.
- Einfach lesbare Adressierung.

2.9 XHTML

Die bekannteste Auszeichnungssprache ist HTML, die als Sprache des Internets (und ausschließlich dort) große Verbreitung gefunden hat. Die Realisierung einer privaten Homepage erfordert sicherlich nicht den Einsatz von XML. Der Einsatz von HTML auf großen Sites ist jedoch problematisch, denn HTML wurde für andere Zwecke entworfen als die, für die es jetzt eingesetzt wird. Es war eigentlich nur für den Austausch von Dokumenten im wissenschaftlichen Bereich gedacht und wird heute mehr als Bindeglied zwischen Information, Daten und multimedialen Elementen verwendet.

Eine Verarbeitung von in HTML codierten Dokumenten durch Parser ist nur sehr begrenzt möglich. Weder die Struktur noch der Inhalt kann schlüssig aus einem HTML-Dokument abgeleitet werden. Recherchen sind nur über Volltext oder Meta Tags möglich. Das typische Aussehen von HTML-Quellcode ist allgemein bekannt:

HTML

```
<font size="2">Hier ist etwas Text in HTML
<p>Dies ist ein Absatz in HTML
<li>Dies ist ein Listeneintrag in HTML
<p>Dies ist ein weiterer Absatz in HTML
</font>
</p>
```

HTML-Browser sind beispiellos tolerant, denn die meisten von ihnen können dieses Fragment problemlos anzeigen, obwohl es unsauber strukturiert ist. Die o.g. Probleme werden durch diese Toleranz jedoch verschärft. Durch XML kann dieses Dilemma auf zwei verschiedenen Wegen gelöst werden.

- Erstellung von »sauber« ausgezeichneten Webseiten durch aus XML generiertem HTML
- Einsatz von XHTML, das HTML ablösen wird

In XHTML wird HTML durch XML beschrieben. Dadurch finden die strikten Vorgaben von XML auf HTML Anwendung, so daß die Interpretation von XHTML-Dateien einfach möglich ist. Das obige Beispiel sähe in XHTML so aus:

XHTML

```
<font size="2">Hier ist etwas Text in HTML
   <p>Dies ist ein Absatz in HTML</p>
   <li>Dies ist ein Listeneintrag in HTML</li>
   <p>Dies ist ein weiterer Absatz in HTML</p>
</font>
```

Die Spezifikation zu XHTML finden Sie unter **http://www.w3.org/TR/xhtml1**. Hier finden Sie auch Hinweise dazu, wie man heute bereits XHTML einsetzen kann, und dennoch von den Browsern verstanden wird.

2.10 Andere Erweiterungen

Mit der einhergehenden Verbreitung von XML wuchs und wächst der Anteil von Sprachbestandteilen und Erweiterungen, wie die Implementierung XML-basierter Sprachen, die für spezielle Anwendungszwecke geschaffen werden.

Festgeschrieben werden diese Erweiterungen in DTDs.

Als Beispiel sind nachfolgend einige Erweiterungen aufgeführt.

2.10.1 SMIL

Durch die **Synchronized Multimedia Integration Language** (SMIL) können in einer XML-basierten Sprache multimediale Daten referenziert werden. Durch SMIL ist es Autoren möglich, Links an Medienobjekte zu binden oder das Layout einer Bildschirmpräsentation zu beschreiben.

Einsatz findet SMIL derzeit z. B. im RealPlayer. Hier können Inhalte mit der SMIL-Syntax erstellt werden, um sie dann mit dem RealPlayer wiederzugeben.

Durch Einsatz der SMIL-Syntax in anderen XML-Dokumenten können dort Timing und Synchronisation realisiert werden.

2.10.2 MathML

Ziel der **Mathematical Markup Language** (MathML) ist es, die Benutzung von mathematischen und wissenschaftlichen Ausdrücken zu erleichtern.

MathML kann zur Kodierung von Sprachsynthetisierung eingesetzt werden und somit als Basis für sprachgesteuerte User Agents genutzt werden.

2.10.3 UXF

Das **UML Exchange Format** (UXF) ist die Portierung der Unified Modelling Language in das XML-Format.

Günde für diese Portierung sind,

- ein standardisiertes Dateiformat für den Datenaustausch zwischen Entwicklungswerkzeugen unabhängig von der zugrundeliegenden Plattform.

- verbesserte Möglichkeiten der Kommunikation zwischen Entwicklern. Insbesonders dann, wenn die Verbindung durch das Inter- bzw. Intranet gewährleistet wird.
- Die Transparenz des Formats, die erweiterten Möglichkeiten der Aufbereitung und Präsentation der Daten.

3 DTD und Schema

In diesem Kapitel werden die Grundbegriffe von Schemata erläutert. Unterschiedliche Sprachen zur Beschreibung von Dokumenttypen werden vorgestellt. Die Ziele beim Einsatz von Schemata werden erläutert.

3.1 Überblick über Schemata

Wann braucht man Schemata?

Eine Stärke von XML ist seine individuelle Erweiterbarkeit. Um eine einfache Handhabung zu ermöglichen, kann auf Dokumenttyp-Definitionen verzichtet werden.

In vielen Anwendungsfällen muß jedoch gewährleistet werden, daß eine Datei einer erwarteten Struktur entspricht, und daß Werte in der Datei im Sinne erwarteter Datentypen gültig sind. Dies ist die Aufgabe von Schemata.

Es gibt verschiedene Schema-Sprachen. Wir werden jedoch überall dort, wo es um grundsätzliche Eigenschaften von Schemata geht, den Begriff Schema verwenden.

Allgemein gesagt, beschreibt ein Schema oder eine DTD die Struktur und das Vokabular von Dokumenten, die in bezug auf dieses Schema gültig sind. Die Verknüpfung einer XML-Datei mit einem Schema ermöglicht dem Parser festzustellen, ob die vorliegende Datei ihrer Typdefinition enspricht.

Schemata und DTDs stellen unterschiedliche formale Grammatiken dar, um Dokumente zu beschreiben. Während die aus der SGML-Welt stammenden DTDs dafür geschaffen wurden, Dokumente zu beschreiben, berücksichtigen Schema-Sprachen zusätzlich den Aspekt der Datenbeschreibung. Gemeinsam ist beiden Wegen der Ansatz: Wenn zwei Dokumente die gleiche Definition verwenden, dann können sie auch mit den gleichen Mitteln verarbeitet werden. Die Definition stellt eine Zusicherung an die verarbeitende Anwendung dar, daß Daten oder Dokument den vereinbarten Regeln entsprechen. Über den Inhalt selbst wird dabei nichts ausgesagt.

Die Möglichkeiten, Informationen über Schemata zur erhalten, sind jedoch noch weit granulierter verfügbar: In Erweiterung der DOM API kennt Microsoft für NODE-Elemente die Eigenschaft `Definition`, die Auskunft über die Spezifikation des Elements im Schema gibt.

Allerdings wird diese Eigenschaft vom Microsoft-Parser nur für Schemata unterstützt, die nach der XML-Data-Spezifikation erstellt wurden, nicht jedoch für XML-Schema oder DTDs. Wir werden die Unterschiede in diesem Kapitel kennenlernen.

3.2 Einsatzmöglichkeiten von Schemata

Schemata definieren Regeln, denen die Dokumente, die das Schema referenzieren, genügen müssen. Eine erfolgreiche Prüfung der Regeln bedeutet, daß ein Dokument sicher verarbeitet werden kann. Dabei müssen Parser nach dem Prinzip »ganz oder gar nicht« verfahren, ein Dokument ist entweder gültig oder nicht.

»Einigermaßen gültig« gibt es nicht, da die XML-Parser beim ersten Fehler die Verarbeitung abbrechen. Dies ist auch logisch, denn sie könnten in der weiteren Verarbeitung sowieso nicht zwischen neuen Fehlern und Folgefehlern unterscheiden. Der erste Fehler wird jedoch qualifiziert mit Zeilenangabe und Position berichtet.

Anwendungsübergreifende Kontrolle über Datenstrukturen

Ein Schema kann von unterschiedlichen Anwendungen genutzt werden, unabhängig von deren Aufgabe im Anwendungssystem, denn es sagt nichts über den Verwendungszweck der Daten aus. Es ist anwendungsneutral und daher universell einsetzbar. Schemata bieten so eine Sicherheitsfunktionalität, die besonders beim Datenaustausch zwischen Anwendungen wichtig ist.

Einsatz von Schemata in Editoren

Anwendungen zum Erstellen von Dokumenten können durch Schema-Definitionen beim Editiervorgang gesteuert werden. In diesem Szenario liest die Editor-Anwendung das Schema und prüft bei der Bearbeitung, ob eine Aktion eines Anwenders ein im Sinne des Schemata gültiges Dokument produziert.

GUI-Erzeugung

Aus Schemata können GUI-Objekte wie z. B. Dialoge abgeleitet werden. Gültige Werte oder Wertbereiche können aus der Spezifikation ermittelt werden.

Vorstellbar sind auch die Prüfung von z. B. Produktkonfigurationen, die als Schemata gepflegt oder generiert werden.

Austausch von Daten

Ein Schema definiert ein eindeutiges Vokabular zur Beschreibung der erwarteten Daten in den Dokument-Instanzen. So kann ein Schema sicherstellen, daß unterschiedliche Anwendungen mit den gleichen Datenformaten arbeiten.

Schemata sichern Transaktionen im Web

Das Schema wird zu diesem Zweck meist auf einer allen Transaktionspartnern zugänglichen Adresse im Internet hinterlegt. Dort wird es vom Ersteller des Schemata verwaltet. Die Dokumente referenzieren dieses Schema, wodurch sichergestellt ist, daß alle Dokumente das gleiche Schema verwenden. Die Veränderung solcher Schemata kann nur noch

mit sehr großer Sorgfalt geschehen, da eine Abwärtskompatibiltät stets gewährleistet sein muß. Haben die XML-Dokumente eine längerfristige Lebensdauer, als bei einer reinen Datenübertragung gegeben, oder sollen sie dauerhaft archiviert werden, ist es empfehlenswert, eine lokale Kopie des Schemata zu erstellen und sicher zu verwahren. Unter Umständen besteht nach einiger Zeit kein Zugriff mehr auf die Web-Ressource des Schemata, und die Dokumente können dann nicht mehr verarbeitet werden.

Auch wenn XML-Dokumente ohne Schemata verarbeitet werden können, kann es dennoch zu Problemen kommen, wenn auf ein benötigtes Schemata nicht mehr zugegriffen werden kann. Schemata können auch Informationen, Hinweise zu ihrer Verarbeitung beinhalten. Diese Informationen können entweder implizit in den Schema-Objekten enthalten sei, oder explizit durch Notationen oder Kommentare transportiert werden. Ein weiterer kritischer Punkt sind in diesem Zusammenhang auch Entitäten, auf die später noch genauer eingegangen wird.

Business-to-Business-Datenübertragung

Bei der Business-to-Business-Datenübertragung zwischen Anwendungssystemen müssen die Vorgaben des jeweiligen Transaktionspartners, dem die XML-Datei zugestellt wird, beachtet werden. Ein Anwendungsbeispiel zeigt den großen Nutzen für den Einsatz von Schemata:

Aufgabe ist die Übertragung von Bestelldaten eines Automobilherstellers an seine Zulieferer. Die Zulieferer setzen unterschiedliche EDV-Systeme ein. Anhand der Dokumenttyp-Definition kann der Hersteller allen Zulieferern eine anwendungsunabhängige Beschreibung der gesendeten Datenformate zur Verfügung stellen. Die Typdefinition wird zentral auf dem Server des Herstellers verwaltet. Der Datenaustausch erfolgt mit XML-Dateien.

Es haben sich verschiedene Industriegruppen gebildet, die zum Teil eigene Schema-Sprachen definiert haben. Es ist zwar zu erwarten, daß proprietäre Sprachen keinen langfristigen Bestand haben werden, die Berücksichtigung solcher Sprachen kann aber dennoch für eine Lösung Bedingung sein.

Dadurch wird es ggf. erforderlich, Dokumente zwischen verschiedenen Schemata zu konvertieren. Dies kann, abhängig von der Komplexität der Aufgabe, durch XSL-Stylesheets oder über das DOM geschehen. In jedem

Falle sollte ein Schema immer hinreichend dokumentiert werden, damit in einem solchen Fall die inhaltlichen Bezüge der Daten verständlich sind.

Dokumentbezogene Schemata

Grundsätzlich kann man zwischen den Anwendungsfällen »Dokumentbezogene Schemata« und »Datenbezogene Schemata« unterscheiden.

Ein dokumentbezogenes Schema bildet die Daten in einer Form ab, wie sie für eine Darstellung des Dokuments günstig ist. Der Aufwand für die Erstellung von Style-Sheets zur formatierten Ausgabe des Dokuments hängt stark von der Strukturierung ab. In der Entwicklung sollte daher bei der Modellierung durch Testdokumente und Stylesheets evaluiert werden, ob das Schema für den geplanten Anwendungsfall geeignet ist. In dokumentbezogenen Schemata ist es häufig sinnvoll, untergeordnete Objekte direkt in die übergeordneten einzubetten, um die Struktur später so einfach wie möglich durchlaufen zu können. Dafür wird eine größere XML-Datei in Kauf genommen.

Datenbezogene Schemata

Datenbezogene Schemata bilden meist die Struktur der Quelldaten 1:1 ab. So werden bei der Abbildung einer Datenbanktabelle die Felder zu Elementen und ihre Eigenschaften zu Attributen.

Strukturierte Objekte werden in der Strukturtiefe abgebildet, mit der sie in der Quelle vorliegen, wobei relationale Beziehungen zu untergeordneten Objekten sowohl eingebettet als auch durch Referenzen zu Elementen speziell dafür erzeugter Auflistungen dargestellt werden können. Dadurch, daß datenbezogene Schemata meist von Anwendungskomponenten bearbeitet werden, können solche Referenzen leichter aufgelöst werden als in Stylesheets.

3.3 Schemata in der Praxis

3.3.1 Einbindung von Schemata in Dokumente

Bevor wir auf die einzelnen Aspekte von Schemata eingehen, hier einige Beispiele, wie Dokumente und Schemata verknüpft werden. Wir werden die verschiedenen Möglichkeiten anhand der DTD-Syntax vorstellen. Dieselben Möglichkeiten sind in der Regel auch mit anderen Schemataspra-chen gegeben und werden später eingeführt.

DTD im Dokument enthalten

Beispiel 1 DTD im Dokument enthalten

```
<!DOCTYPE menuBar [
<!ELEMENT menuBar (menuItem+ )
]>
```

Hier wird die gesamte DTD innerhalb des XML-Dokuments notiert. Der Vorteil ist, daß Schema und Dokument in einer Einheit verwaltet werden. Auf diese Art kann jedoch nicht sichergestellt werden, daß verschiedene Dokumente dieselbe DTD nutzen.

Externe DTDs mit enthaltener interner Entity-Deklaration

Externe DTDs werden in vom Dokument getrennten Dateien notiert. Dadurch können verschiedene Dokumente die gleiche DTD benutzen.

Häufig werden interne und externe Deklarationen gemeinsam verwandt. Ein gängiger Anwendungsfall hierfür ist das Hinzufügen von internen Entities zu externen DTDs.

```
<!DOCTYPE menuBar SYSTEM "menuBar.dtd" [
<!ENTITY copyright "copyright 1999 tarent GmbH">
]>
```

Externe DTD-Deklaration

Die externe DTD ist der häufigste Anwendungsfall:

```
<!DOCTYPE menuBar SYSTEM "menuBar.dtd">
```

3.3.2 Schema-Elemente

Die Gestaltung von Schemata erfolgt durch Deklaration der Schema-Elemente. Die möglichen Schema-Elemente sind in den verschiedenen Schema-Sprachen weitgehend gleich, auf die Unterschiede werden wir später eingehen. Wir sehen uns zunächst die verfügbaren Deklarationselemente an.

Die wichtigsten verfügbaren Deklarationen:

- Elemente
- Attribute
- Modellgruppen
- Attributlisten
- Entitäten

Elemente

Elemente können Attribute, Attributlisten, Modellgruppen oder Text enthalten. Sie entsprechen einem Knoten im XML-Dokument. In einem gültig validierten XML-Dokument wird z. B. der Knoten

```
<menuBar>
```

durch eine Deklaration

```
<!ELEMENT menuBar  (menuItem+ )>
```

beschrieben.

Attribute und Attributlisten

Attribute bilden Eigenschaften von Elementen ab. Sie können, abhängig von der Schemataspache, innerhalb eines Elements oder innerhalb einer Attributliste deklariert sein. Hier die DTD-Schreibweise als Attributliste:

```
<!ATTLIST menuItem  id   CDATA  #REQUIRED>
```

Die Zuweisung des Attributs an das Element erfolgt hier über den Namen der Attributliste. Dies ist für DTD-Schemata typisch.

Modellgruppen

Modellgruppen sind Listen von Elementen, die in einem anderen Element enthalten sein dürfen:

```
<!ELEMENT menuItem  (menuItem+ | link+ )>
```

Entitäten

Entitäten sind ähnlich wie Makros in Progammiersprachen oder wie Textbausteine in Texteditoren zu sehen. Sie verwalten unter einem symbolischen Namen Zeichenfolgen. Sogar binäre Daten können in Entities verwaltet werden. Im folgenden Beispiel wird einfach ein Text hinterlegt:

```
<!ENTITY copyright "copyright 1999 tarent GmbH">
```

Dieser Text kann im Dokument später als `&copyright;` referenziert werden. Die Referenz wird im Dokument in den Wert `copyright 1999 tarent GmbH` aufgelöst.

Entities werden auch innerhalb von DTDs als wiederverwertbare Deklarationsblöcke eingesetzt.

3.3.3 Allgemeine Problematiken bei der Modellierung

Auch wenn wir Schemata im Detail erst im weiteren Verlauf dieses Kapitels kennenlernen werden, sei hier schon einmal auf allgemeine Problematiken bei der Modellierung von Schemata und damit letztlich von XML-Dateien hingewiesen.

Diese Problematiken betreffen vor allem die Entscheidungen:

- Wie tief soll ein Dokument strukturiert sein?
- Wann soll ein Element, und wann soll ein Attribut eingesetzt werden?

Die Entscheidung über die Modellierung von Daten in Elementen und Attributen behandelt der folgende Abschnitt.

Element versus Attribut

Eine XML-Regel, die bestimmt, ob Daten in Elementen oder Attributen modelliert werden sollen, gibt es nicht. Generell gilt, daß Elemente Inhalte beschreiben und Attribute wiederum Elemente. Es ist durchaus möglich, daß Dokumente entsprechend dem einzelnen Anwendungsfall

in ihrer Struktur transformiert werden, zum Beispiel wenn die Erstellung von Stylesheets sonst nur mit unverhältnismäßigem Aufwand möglich wäre. Wir kommen darauf im Kapitel über XSL-Stylesheets zurück.

Abbildung durch Elemente

Bei der Abbildung durch Elemente werden Eigenschaften eines Objekts als untergeordnete Elemente notiert. Dadurch ergibt sich eine einfache Lesbarkeit, jedoch eine tiefere Struktur.

Beispiel 2 Abbildung durch Elemente

```
<Customer>
   <Adress>
      <Created>2000-02-01</Created>
      <Prename>MyPrename</Name>
<Lastname>MyLastname</Lastname>
<Country>MyCountry</Country>
   </Adress>
</Customer>
```

Abbildung durch Attribute

Bei der Abbildung durch Attribute entsteht eine kompakte Darstellung, wobei gelegentlich die Lesbarkeit leidet. Da die direkte Bearbeitung von XML-Dateien ein eher seltener Anwendungsfall ist, ist dieser Aspekt jedoch nachrangig.

Beispiel 3 Abbildung mit Attributen

```
<Customer>
      <Adress Created="2000-02-01" Prename="MyPrename"
       Lastname="MyLastname" Country="MyCountry"/>
</Customer>
```

Kriterien für die Modellierung mit Element und Attributen

Wichtig ist neben dem Aspekt der optimalen fallbezogenen Lösung die Anforderung, ein konsistentes und nachvollziehbares Vorgehen anzuwenden. Außer wenn in Schemata Default-Werte oder Wertbereiche spezifiziert werden sollen, was zur Zeit für Elemente nicht möglich ist, besteht freie Auswahl in bezug auf den zu wählenden Weg. Grundsätzlich ist, mit der o.g. Ausnahme, die Modellierung ausschließlich mit Elementen immer möglich, führt jedoch zu größeren Dateien.

Ein besonders häufiger Fall ist die Abbildung von Anwendungsobjekten, die durch eine XML-Schnittstelle serialisiert werden sollen. Hier ist es sinnvoll, die Struktur im korrespondierenden XML-Objekt analog anzulegen, um den intuitiven Umgang z. B. beim Verfassen von Stylesheets oder der Programmierung mit dem DOM zu unterstützen.

Im folgenden werden Entscheidungstabellen vorgeschlagen, die der Leser sicherlich mit der Zeit mit seinen eigenen Erfahrungen anreichern wird.

Ausschlußkriterien

Zunächst werden die Anforderungen dargestellt, die entweder Elemente oder Attribute erzwingen:

Anforderung	Implementierung als
Sind die Daten strukturiert?	Element
Sollen Default-Werte verwaltet werden?	Attribute

Tabelle 3.1

Wahlfreie Kriterien

Hier werden die Anforderungen dargestellt, die entweder Elemente oder Attribute ermöglichen:

Anforderung	Bevorzugt Implementierung als
Sind die Daten nicht strukturiert?	Attribute
Sollen die Daten ergänzend beschrieben werden?	Element
Soll eine möglichst kompakte Datei erstellt werden?	Attribute
Soll eine möglichst einfach lesbare Datei erstellt werden?	Elemente
Sind die Daten der Quellobjekte mit Eigenschaften modelliert?	Attribute

Tabelle 3.2

DTD versus Schema

Welche Beschreibungssprache sollte gewählt werden?

Zunächst muß eine Entscheidung getroffen werden, welche Beschreibungssprache gewählt werden soll. Zwei grundsätzlich unterschiedliche Ansätze verfolgen, wie schon im ersten Abschnitt dieses Kapitels erwähnt, DTDs und Schemata.

Vor- und Nachteile von DTDs

DTDs sind der Standard. Sie werden von praktisch allen validierenden Parsern erkannt und verarbeitet. Hier gibt es die meiste Literatur, Beispiele und Unterstützung durch Softwarewerkzeuge. Bei der Verarbeitung von SGML-Dokumenten kann auf verfügbare DTDs zurückgegriffen werden. Die DTD-Syntax verfügt nur über wenige Elemente und ist einigermaßen intuitiv erfaßbar und daher recht schnell erlernbar.

Vorteile
Einfache Erstellung
Langjährig etablierter, eindeutiger Standard
Gute Unterstützung durch Parser
Entities werden unterstützt
Nachteile
Nicht in XML formuliert
Große DTDs sind sehr unübersichtlich
Datentypen werden nicht unterstützt
Kein direkter Zugriff über das DOM
Wenige Ausdrucksmittel

Tabelle 3.3

Vor- und Nachteile von Schemata

Während es DTDs aufgrund ihrer SGML-Herkunft schon einige Jahre gibt, sind Schemata so jung wie XML. Es gibt zur Zeit ein paar konkurrierende Spezifikationen, auf die wichtigsten werden wir im nächsten Abschnitt, »Die verschiedenen Schema-Spezifikationen«, eingehen. Das W3C hat

sich für die Unterstützung von XML-Schemata entschieden, dennoch gibt es durchaus sinnvolle Anwendungsfälle für andere Initiativen. Die gemeinsamen Vor- und Nachteile von Schemata in der folgenden Tabelle:

Vorteile
Werden in XML formuliert
Direkter Zugriff über das XML DOM
Größerer Sprachschatz für genauere Beschreibung
Datentypen werden unterstützt
Individuell erweiterbar
Nachteile
Entities werden noch nicht unterstützt
Kein einheitlicher Standard

Tabelle 3.4

Welche Schema-Sprache sollte man einsetzen?

Die interessanteste Frage für den Praktiker ist sicherlich, welche Schema-Sprache man einsetzen sollte. Eine allgemeingültige Antwort kann man auf diese Frage nicht geben. Die verschiedenen Schema-Spezifikationen entstanden zum Teil aus strategischen und historischen Gründen, zum Teil aus den unterschiedlichen Anforderungen, die an sie gestellt wurden.

Wir stehen, im Gegensatz zu den meisten Autoren, die das Thema VB und XML behandeln, den proprietären Erweiterungen Microsofts eher kritisch gegenüber, müssen allerdings zugeben, daß Microsofts Taktik, Verwirrung in der XML-Welt zu stiften, aufgegangen ist. Dies gilt nicht nur, aber auch für Schemata.

Wenn es sein muß: XML-Data

Wer also mit den Microsoft Tools arbeitet und beabsichtigt, dies auch zukünftig ausschließlich zu tun, ist vielleicht mit XML-Data gut bedient, denn XML-Schema wird zur Zeit von Microsoft nicht unterstützt.

Ob XML-Data eingesetzt werden soll, kann von vielen projektbezogenen Faktoren abhängen. Durch erweiterte DOM-Methoden des Microsoft-Parsers kann von einem Element direkt auf seine entsprechende Schema-Definition zugegriffen werden, wenn ein XML-Data-Schema verwendet wird. Ein schönes Feature, dennoch raten die Verfasser vom Einsatz von XML-Data grundsätzlich ab, wenn nicht geplant ist, die Schemata später zu konvertieren.

Die Zukunft: XML-Schema

XML-Schema ist dem W3C zufolge der kommende Standard. XML-Parser, die XML-Schema unterstützen, sind als Java-Implentierungen reichlich verfügbar. Ein Parser mit einer COM-Schnittstelle, wie sie der MSXML-Parser hat, der zugleich XML-Schema beherrscht, ist uns nicht bekannt. Microsofts Dominanz mit seinem ActiveX-Parser ist auf der Windows-Plattform so groß, daß es kaum wahrscheinlich erscheint, daß sich viele Hersteller der Aufgabe, ebensolche XML-Parser zu entwicklen, stellen werden. Kurz gesagt: Was an XML-Parsern mit ActiveX-Schnittstelle für Windows erstellt wird und welche Features diese Parser verfügbar haben, wird von Microsoft bestimmt.

Wer auf Schemata nicht verzichten kann oder will und dem W3C-Standard folgen möchte, sollte auf XML-Schema setzen. Hier ist größte Zukunftssicherheit gegeben und zudem gewährleistet, daß erstellte Schemata von unterschiedlichen Softwaretools verarbeitet werden können. XML-Schema-Tools sind für Windows als C+- Klassenbibliotheken z. B. von IBM verfügbar. Alternativ können Java-Parser unter Windows eingesetzt werden.

Da bei RDF die Entwicklung noch nicht abgeschlossen ist, ist es für den produktiven Einsatz vielleicht noch etwas früh. Dennoch ist RDF überall dort eine Alternative, wo nicht nur Schemata beschrieben, sondern Objekte modelliert werden sollen. RDF ist als Format für die Beschreibung beliebiger Ressourcen besonders für Abbildung von Dokumenttypen und von Dokument-Metadaten geeignet.

Die verschiedenen Schema-Spezifikationen

Um die Unterschiede deutlich zu machen, werden wir ein einfaches Schema in verschiedenen Schema-Sprachen darstellen. Wir werden dafür ein einfaches Anwendungsobjekt durch ein Schema beschreiben. Das Modell ist ebenso auf Dokumente übertragbar.

Beispiel 4 Schema für eine Menüleiste

In unserem Beispiel wird eine einfache Menüleiste beschrieben. Die Menüleiste soll Menüeinträge enthalten, die Untermenüs oder Aufrufe enthalten können.

Etwas formaler (und genauer) ausgedrückt:

- Die Menüleiste enthält ausschließlich Menüeinträge
- Menüeinträge enthalten entweder Menüeinträge oder Aufrufe
- Die Aufrufe enthalten die Eigenschaften des Aufrufs (Quelle, Ziel)
- Mögliche Ziele von Aufrufen müssen `content` oder `nav` heißen

Daraus ergibt sich folgendes Modell:

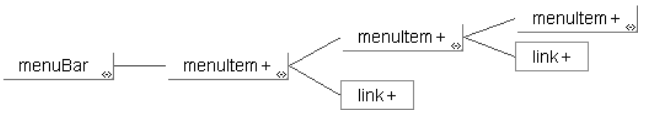

Abbildung 3.1

Wir werden dieses Modell als DTD notieren, die als Referenz für die übrigen Schema-Darstellungen dient. Dafür bilden wir die Objekte (`menuBar`, `menuItem`, `link`) als Elemente ab, die Eigenschaften der Objekte als ihnen zugeordnete Attribute (`name`, `id`, ...).

Die Elemente der DTD sind:

- `menuBar`, als Container für alle Menu-Elemente der 1. Hierarchie, `<!ELEMENT menuBar (menuItem+ )>`
- `menuItem`, als Menüelement und Container für Submenüs `<!ELEMENT menuItem (menuItem+ | link+ )>`
- `link`, als Container für die Beschreibung der Verknüpfung, die beim Aufruf eines Menüelements aufgelöst werden soll `<!ELEMENT link EMPTY>`

Die Attribute eines `menuItem` sind:

- `id`, um eine eindeutige ID für das Element zu verwalten
- `name`, um den angezeigten Namen auszunehmen

In der DTD sieht das so aus:

```
<!ATTLIST menuItem  id   CDATA  #REQUIRED
                    name CDATA  #REQUIRED >
```

Die Attribute eines `link` sind:

- `url`, um eine eindeutige ID für das Element zu verwalten
- `target`, um den angezeigten Namen aufzunehmen

```
<!ATTLIST link  url    CDATA  #REQUIRED
          target  (nav | content )  #REQUIRED >
```

Hier nun die vollständige DTD:

Beispiel 4 Referenz-DTD Menübar

```
<!ELEMENT menuBar  (menuItem+ )>

<!ELEMENT menuItem  (menuItem+ | link+ )>
<!ATTLIST menuItem  id   CDATA  #REQUIRED
                    name CDATA  #REQUIRED >
<!ELEMENT link EMPTY>
<!ATTLIST link  url    CDATA  #REQUIRED
                target  (nav | content )  #REQUIRED >
```

Im Hinblick auf die enthaltenen Informationen ist die DTD sehr kompakt. Wir werden nun diese DTD in anderen Schemataprachen betrachten.

XML-Data

Der Internet Explorer versteht XML-Schema nicht

Um möglichst schnell Schema nutzen zu können, implementierte Microsoft im Internet Explorer eine Untermenge der XML-Data-Spezifikation. Zum jetzigen Zeitpunkt wird auch nur diese vom Microsoft Parser mit vollem Funktionsumfang unterstützt. XML-Schema verstehen der Microsoft-Parser und -Browser nicht.

Der IE5 validiert bei Anzeige einer XML-Datei nicht. Eine Validierung im Browser muß explizit durch ein Skript angestoßen werden. Auch wenn XML-Data nicht der Favorit des W3C bei der Entscheidung zu einer einheitlichen Schema-Sprache ist, kommt man, wenn man die Features des Microsoft-Parsers vollständig nutzen will, nicht am Einsatz von XML-Data vorbei. Ein gängiger Kompromiß ist der Einsatz von DTDs.

Hier die Darstellung unserer DTD im sogenannten IE5 Subset von XML-Data:

Beispiel 5 Menübar als XML Data Subset

```
<?xml version ="1.0"?>
<Schema name = "menubar.dtd"
    xmlns = "urn:Schemata-microsoft-com:xml-data"
    xmlns:dt = "urn:Schemata-microsoft-com:datatypes">
   <ElementType name = "menuBar" content = "eltOnly" order = "seq">
      <element type = "menuItem" minOccurs = "1" maxOccurs = "*"/>
```

```
   </ElementType>

   <ElementType name = "menuItem" content = "eltOnly" order = "one">
      <AttributeType name = "id" dt:type = "string" required =
"yes"/>
      <AttributeType name = "name" dt:type = "string" required =
"yes"/>
      <attribute type = "id"/>
      <attribute type = "name"/>
      <element type = "menuItem" minOccurs = "1" maxOccurs = "*"/>
      <element type = "link" minOccurs = "1" maxOccurs = "*"/>
   </ElementType>

   <ElementType name = "link" content = "empty">
      <AttributeType name = "url" dt:type = "string" required =
"yes"/>
      <AttributeType name = "target" dt:type = "enumeration"
dt:values = "nav content" required = "yes"/>
      <attribute type = "url"/>
      <attribute type = "target"/>
   </ElementType>
</Schema>
```

XML-Schema

XML-Schema ist der vom W3C favorisierte Standard und wird sich langfristig durchsetzen. Jedoch wird es noch einige Zeit dauern, bis eine flächendeckende Unterstützung durch Parser gegeben ist. Die vorhandenen Implementierungen haben eher experiementiellen Charakter. Die Unterschiede zwischen dem vom Microsoft unterstützen XML-Data und XML-Schema sind jedoch nicht so groß, daß eine Konvertierung zwischen den Sprachen nicht möglich wäre, denn beides sind schließlich XML Sprachen. Auch kann man davon ausgehen, daß Parser noch einige Zeit zu DTDs abwärtskompatibel sein werden. Die Darstellung der DTD in XML-Schema macht die Unterschiede zu XML-Data deutlich:

```
<?xml version ="1.0"?>
<schema name = "menubar"
  xmlns="http://www.w3.org/1999/05/06-xmlschema-
1/structures.xsd">
   <elementType name = "menuBar" model = "open">
      <sequence>
      <elementTypeRef name = "menuItem" minOccur = "1" maxOccur =
"*"/>
      </sequence>
   </elementType>

   <elementType name = "menuItem" model = "open">
      <choice>
         <elementTypeRef name = "menuItem"
                       minOccur=         1"               maxOccur =
"*"/>
      <elementTypeRef name = "link" minOccur = "1" maxOccur = "*"/>
      </choice>
      <attrDecl name = "id" required = "true">
         <datatypeRef name = "string"/>
      </attrDecl>
      <attrDecl name = "name" required = "true">
         <datatypeRef name = "string"/>
      </attrDecl>
   </elementType>

   <attrDecl name = "id" required = "true">
      <datatypeRef name = "string"/>
   </attrDecl>

   <attrDecl name = "name" required = "true">
      <datatypeRef name = "string"/>
   </attrDecl>

   <elementType name = "link" model = "open">
      <empty/>
      <attrDecl name = "url" required = "true">
         <datatypeRef name = "string"/>
```

```
    </attrDecl>
    <attrDecl name = "target" required = "true">
       <datatypeRef name = "ENUMERATION">
          <enumeration>
             <literal>nav</literal>
             <literal>content</literal>
          </enumeration>
       </datatypeRef>
    </attrDecl>
</elementType>

<attrDecl name = "url" required = "true">
    <datatypeRef name = "string"/>
</attrDecl>

<attrDecl name = "target" required = "true">
    <datatypeRef name = "ENUMERATION">
       <enumeration>
          <literal>nav</literal>
          <literal>content</literal>
       </enumeration>
    </datatypeRef>
</attrDecl>
</schema>
```

Darstellung als RDF

RDF stellt den, im Vergleich zu den bisher beschriebenen Schema-Sprachen, allgemeinsten Ansatz zur Beschreibung von Metadaten zur Verfügung. Das Ziel von RDF ist nicht auschließlich die Beschreibung von Dokument-Schemata, sondern die Beschreibung beliebiger Ressourcen. Die Ziele von RDF:

- Bessere Recherche nach Web-Dokumenten
- Möglichkeit der Indexierung und Katalogisierung
- Beschreibung virtueller Dokumente
- Profiling

Für die Beschreibung von Schemata durch RDF wurde RDF-Schema erstellt. RDF-Schema ist ein Satz von RDF-Objekten mit dem Ziel, Dokument und Objektbeschreibungen zu erstellen.

Durch Integration Digitaler Signaturen soll RDF ein Standard für E-Commerce und kollaborative Anwendungen werden.

Das Konzept von RDF ähnelt dem eines Klassensystems einer objektorientierten Programmiersprache. Klassen können wiederverwendet und auch verfeinert werden. Eine Sammlung von Klassen einer Applikation in einer RDF-Datei wird als Schema bezeichnet. Für diesen Zweck stellt RDF eine eigene Spezifikation zur Verfügung, die die für die Beschreibung von Schemata erforderlichen Klassen enthält: RDF-Schema. Hier unser Beispiel in RDF codiert.

```
<?xml version ="1.0"?>
<DCD xmlns:RDF = "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <ElementDef Type = "menuBar" Content = "Closed"
            Model = "Elements">
      <Group RDF:Order = "Seq">
         <Group Occurs = "OneOrMore">
            <Element>menuItem</Element>
         </Group>
      </Group>
   </ElementDef>

   <ElementDef Type = "menuItem" Content = "Closed"
              Model = "Elements">
      <AttributeDef Name = "id" Occurs = "Required"/>
      <AttributeDef Name = "name" Occurs = "Required"/>
      <Group RDF:Order = "Alt">
         <Group Occurs = "OneOrMore">
            <Element>menuItem</Element>
         </Group>
         <Group Occurs = "OneOrMore">
            <Element>link</Element>
         </Group>
      </Group>
   </ElementDef>
```

```
   <ElementDef Type = "link" Content = "Closed" Model =
             "Empty">
      <AttributeDef Name = "url" Occurs =
                                                "Required"/>
      <AttributeDef Name = "target" Datatype =
                         "enumeration" Occurs = "Required">
         <Values>nav content</Values>
      </AttributeDef>
   </ElementDef>
</D
```

4 Das Document Object Model

Das Document Object Model (DOM) wurde vom W3C entwickelt, um ein plattform- und sprachunabhängiges Interfacemodell für objektorientierte Dokumente bereitzustellen. Im Normalfall versteht man unter einem Dokument einen Text mit den dazugehörigen Auszeichnungen. In XML umfaßt dieser Begriff nun aber auch enthaltene Daten, die in einer strukturierten Form vorliegen und dadurch eindeutig zu adressieren sind.

Aufgrund der in XML-Dokumenten enthaltenen Struktur und der Möglichkeit, diese durch Objekte abzubilden, kann man von einer objektorientieren Dokumentstruktur sprechen.

Das DOM spezifiziert das Interface für den Zugriff auf diese objektorientierte Dokumentstruktur.

4.1 Das W3C DOM

4.1.1 Was ist das DOM und was ist es nicht?

Das DOM ist ein plattform- und sprachunabhängiges Interface für den Zugiff auf die Dokumentstruktur und hat somit keine binäre Ausprägung. Damit sollte klar sein, daß es sich auch beim Microsoft DOM nicht um »das« DOM handelt, sondern um eine Komponente, die das DOM-Interface implementiert.

Des weiteren stellt das DOM auch kein Regelwerk für die XML-Semantik dar. Diese ist in der entsprechenden XML-Spezifikation des W3C festgehalten.

Zugriffskomponenten, die das Interface implementieren, stellen damit alle notwendigen Funktionen für die Manipulation von XML-Dokumenten zur Verfügung.

4.1.2 Vom Dokument zum DOM

Die im XML-Dokument enthaltene Struktur wird innerhalb des DOMs durch geschachtelte Objekte dargestellt. Somit entsteht ein Objektmodell mit einer Baumstruktur, das durch die verschiedenen Hirarchiestufen Knotenpunkte (Nodes) erhält.

```
<specifications>
<spec>Canonical XML</spec>
<spec>Extensible Stylesheet Language</spec>
<spec> Infoset WD of May 8, 1999</spec>
   .
</specifications>
```

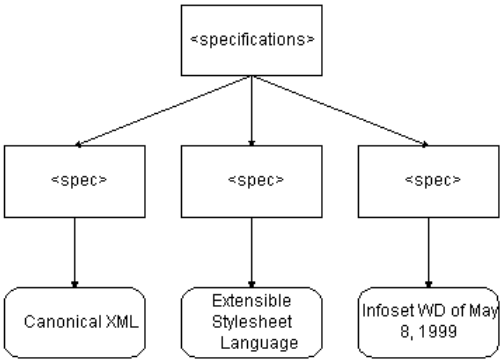

Abbildung 4.1 Objektstruktur des vorangegangenen XML-Dokuments im DOM

An jedem Objekt innerhalb der Baumstruktur, an dem eine Aufteilung in weitere darunterliegende Objekte erfolgt, sprechen wir von einem Node-Objekt. Alle Node-Objekte enthalten eine Auflistung der Objekte, die in der untergeordneten Hirarchiestufe liegen. Abgesehen vom Wurzelobjekt enthält jedes Node-Objekt eine Referenz auf das ihm übergeordnete Node-Objekt.

4.1.3 Objekttypen im DOM

Obwohl das DOM auch die kleinste Informationseinheit durch ein generisches Node-Objekt darstellt, kann man verschiedene Objekttypen innerhalb des DOM unterscheiden.

Die nachfolgende Grafik zeigt einen Ausschnitt der zur Verfügung stehenden Objekttypen.

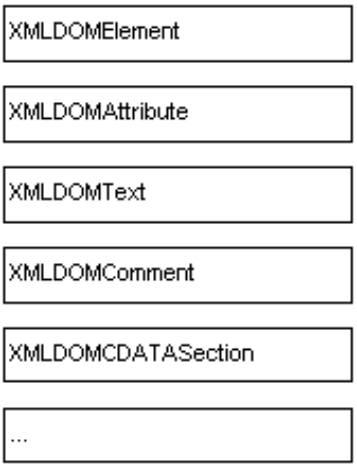

Abbildung 4.2 Eine kleine Auflistung von Objekttypen innerhalb des DOM

4.2 Das Microsoft DOM

Microsoft liefert mit der MS XML-Library eine ActiveX-Komponente aus, die das DOM in der Version Level 1 implementiert.

4.2.1 Die Basis-Objekte im Microsoft DOM

Alle nachfolgend beschriebenen Objekte implementieren die Basis-Interfaces, die durch das W3C spezifiziert sind. Hinzu kommen einige Microsoft-spezifische Erweiterungen für die Unterstützung von Namespaces, DataTypes, XML-Schemas, XSL-Operationen und dem asynchronen Laden von XML-Dateien.

Bevor wir genauer auf einzelne Objekte innerhalb des Microsoft DOM eingehen, zeigt die nachfolgende Tabelle einen kurzen Abriß der enthaltenen Objekte.

Objektname	Beschreibung
XMLDOMDocument	Dieses Objekt repräsentiert das XML-Dokument als solches.
XMLDOMNode	Dieses Objekt repräsentiert ein einzelnes Element innerhalb des Dokumentenbaums. Hierbei unterstützt es Datentypen, Namespaces, DTDs und XML-Schemas.
XMLDOMNodeList	Dieses Objekt dient dem Zugiff per Name oder Index auf eine Liste mit XMLDOMNode-Objekten.
XMLDOMNamedNodeMap	Dieses Objekt dient dem Zugiff auf die Attribute eines Elements.
XMLDOMParseError	Dieses Objekt liefert detaillierte Informationen über den letzten aufgetretenen Fehler. Hierzu zählen Zeilennummer, Position des Zeichens etc.
XMLHTTPRequest	Dieses Objekt ermöglicht es, eine HTTP-Verbindung zu einem Web-Server herzustellen und Standard-HTTP-Funktionen wie Put oder Get auszuführen.
XTLRuntime	Dieses Objekt beinhaltet Methoden für die Transformation von XML-Dateien mittels XSL.

Tabelle 4.1

XMLDOMDocument

Das **XMLDOMDocument**-Objekt repräsentiert die XML-Datei selbst. Es ist das einzige DOM-Objekt, das von außen instanziiert werden kann. Auf die anderen im DOM enthaltenen Objekte kann nur über das **XMLDOM**-Document zugegriffen werden.

Durch das Laden eines XML-Streams bzw. einer XML-Datei erfolgt deren Validierung, sofern diese nicht deaktiviert ist.

XMLDOMNode

Hierbei handelt es sich um ein Basisobjekt, dessen Eigenschaften und Methoden von vielen anderen Objekten innerhalb des DOMs implementiert wurden.

Über die **nodeType**-Eigenschaft des Objekts ist es möglich, den Namen des tatsächlichen Objekttyps zu erfahren.

XMLDOMNodeList

Dieses Objekt stellt eine Collection von **XMLDOMNode**-Objekten dar. Es wird meistens zusammen mit dem **childNodes**-Property des **XMLDOMNode**-Objekts genutzt.

XMLDOMNamedNodeMap

Das **XMLDOMNamedNodeMap**-Objekt ist eine weitere Collection innerhalb des Microsoft DOM. Es wird für die Auflistung von Attributen eines **XMLDOMNode**-Objekts genutzt.

XMLDOMParseError

Sollte das **DOMDocument**-Objekt beim Parsen des XML-Codes einen Fehler finden, so wird dieser durch das **XMLDOMParseError**-Objekt beschrieben.

Hierbei liefert das Objekt Informationen über die Zeilennummer des Fehlers, das Zeichen, die Fehlerbeschreibung etc.

XMLHTTPRequest

Dieses Objekt ermöglicht den Aufbau einer Verbindung zu einem Webserver. Ebenso wie die hauptsächlichen Methoden **open()** für den Aufbau

einer Verbindung und **send()** für die Übermittlung stehen natürlich auch Methoden und Eigenschaften für die Auswertung der Antwort des Webservers zur Verfügung.

XTLRuntime

Die Methoden dieses Objektes sind einzig und allein für die Transformation von XML-Dateien durch XSL vorgesehen und sind daher auch nur innerhalb des XSL-Codes zugreifbar. Für den Zugriff mittels Visual Basic gibt es keine Möglichkeit.

4.2.2 Die erweiterten Objekte im Microsoft DOM

Nachdem Sie im vorherigen Abschnitt über die Basis-Objekte innerhalb des Microsoft DOMs informiert wurden, kommen wir nun zu den erweiterten Objekten.

Die erweiterten Objekte implementieren hierbei Interfaces, die durch die Basis-Objekte des DOM definiert wurden.

Auch hierbei erhalten Sie nachfolgend einen kurzen Abriß über die Funktion der hierzu gehörigen Objekte.

Objektname	Beschreibung
XMLDOMCharacterData	Dieses Objekt stellt Methoden für die Textbearbeitung zur Verfügung und wird daher auch von anderen Objekten innerhalb des DOM implementiert.
XMLDOMComment	Dieses Objekt repräsentiert den Inhalt eines Comments.
XMLDOMDocument-Fragment	Dieses leichtgewichtige Objekt repräsentiert ein Fragment aus dem Objektbaum des DOMs.
XMLDOMEntityReference	Dieses Objekt repräsentiert eine Entity innerhalb des XML-Codes.
XMLDOMImplementation	Dieses Objekt stellt Methoden zur Verfügung, die unabhängig von der Instanz des DOM sind.
XMLDOMNotation	Dieses Objekt repräsentiert die Deklaration der Notation innerhalb einer DTD oder eines Schemas.
XMLDOMText	Dieses Objekt repräsentiert den Text eines Elements oder eines Attributes.

Tabelle 4.2

Objektname	Beschreibung
XMLDOMElement	Dieses Objekt repräsentiert jedes Element innerhalb des Dokuments.
XMLDOMCDATASection	Dieses Objekt repräsentiert den Wertebereicht eines Elements, das in eine CDATA-Klammer eingeschlossen ist. Diese Daten wurden beim Parsen nicht berücksichtigt.
XMLDOMAttribute	Dieses Objekt repräsentiert ein einzelnes Attribut eines Elements.
XMLDOMDocumentType	Dieses Objekt repräsentiert die Document Type Definition bzw. den Verweis auf eine entsprechende Datei.
XMLDOMEntity	Dieses Objekt repräsentiert eine Entity im DTD-Bereich einer XML-Datei.
XMLDOMProcessing-Instruction	Dieses Objekt repräsentiert eine Processing Instruction in der XML-Datei.

Tabelle 4.2

XMLDOMCharacterData

Das **XMLDOMCharacterData**-Objekt repräsentiert keinen Node innerhalb des XML-Codes. Vielmehr dient es als Hilfsobjekt für andere Objekte, die einen Node mit langen Textpassagen repräsentieren. Hierfür stellt dieses Objekt Methoden für die Bearbeitung zur Verfügung.

XMLDOMComment

Dieses Objekt repräsentiert den gesamten Inhalt innerhalb der XML-Comment-Tags <!-- und -->. Spezielle Methoden weist dieses Objekt nicht auf und ist deckungsgleich mit dem **XMLDOMCharacterData**-Objekt.

XMLDOMDocumentFragment

Das leichtgewichtige **XMLDOMDocumentFragment**-Objekt repräsentiert einen Teilbaum der Objektstruktur innerhalb des XML-Dokumentes. Hierbei stellt es z. B. für das Einfügen von Nodes spezielle Methoden zur Verfügung, die dieses Objekt für Entwickler interessant werden lassen.

XMLDOMEntityReference

Dieses Objekt stellt eine einzelne Entität dar, wie sie im XML-Code festgelegt ist.

XMLDOMImplementation

Dieses Objekt hat den Charakter eines Hilfsobjekts. Seine Methoden sind unabhängig von einer laufenden Instanz des DOM.

XMLDOMNotation

Das **XMLDOMNotation**-Objekt enthält die Notation, die innerhalb einer DTD oder eines XML-Schemas spezifiziert ist.

XMLDOMText

Dieses Objekt repräsentiert den Wertebereich eines Nodes. Hierbei ist es möglich, innerhalb eines XMLDOMNode-Objekts mehrere dieser Nodes zu erzeugen.

Beim Ladevorgang eines XML-Dokuments erfolgt eine Normalisierung dieser Objekte. Das heißt, daß innerhalb eines **XMLDOMNode**-Objektes nur ein **XMLDOMText**-Objekt enthalten ist.

Sollten Sie diese Normalisierung innerhalb eines geladenen und bereits bearbeiteten XML-Dokumentes durchführen wollen, so befindet sich hierfür am **XMLDOMElement**-Objekt die **normalize**-Methode.

XMLDOMElement

Dieses Objekt repräsentiert jedes Element innerhalb des XML-Codes. Hierbei unterstützt es alle notwendigen Funktionen für die Manipulation des Elements selbst und der zugehörigen Attribute.

XMLDOMCDATASection

Dieses Objekt repräsentiert den Wert, der sich innerhalb der **CDATA**-Tags (![...]]) im XML-Code befindet.

XMLDOMAttribute

Durch das **XMLDOMAttribute** wird ein Attribut eines Elementes dargestellt. Die **XMLDOMAttribut**-Objekte eines **XMLDOMNode**-Objektes werden hierbei in einer **XMLDOMNamedNodeMap**-Collection gehalten.

XMLDOMDocumentType

Es enthält alle Information bezüglich enthaltener Entities oder DTD-Deklarationen und kann nur über das **ReadOnly**-Property des **DOMDocument**-Objekts zugegriffen werden.

XMLDOMEntity

Innerhalb des DTD-Bereichs können Sie Entities für die Benutzung als Konstanten deklarieren. Fünf solcher Entities sind bereits definiert (predifined Entities) und bedürfen somit keiner expliziten Deklaration.

Zeichen	Entity-Konstante
&	&amp;
'	&apos;
>	&gt;
<	&lt;
"	&quot;

Tabelle 4.3

XMLDOMProcessingInstruction

Processing Instructions innerhalb des XML-Codes werden durch dieses Objekt dargestellt.

5 Abfragen in XML

XML stellt eine eigene Abfragesprache zur Verfügung, die als XPath bezeichnet wird. XPath dient zur Positionierung auf Elementebene in XML-Dateien. Sie kann sowohl im DOM als auch in XSL-Stylesheets eingesetzt werden.

Die Syntax, mit der XML-Dateien auf Ebene der DOM-Elemente abgefragt werden können, ist in der XPath-Spezifikation festgelegt. Man kann mit XPath-Ausdrücken alle Elemente adressieren. Für den Zugriff innerhalb von XML-Elementen (zum Beispiel eine Zeichenposition in einem Text-Node) ist XPointer gedacht. Da dies von den Parsern noch nicht implementiert ist, gehen wir nicht näher darauf ein. Die XPointer-Spezifikation ist auf der Buch-CD enthalten oder beim W3C einzusehen.

Der Microsoft-Parser unterstützt eine erweiterte Version von XPath, die auch unter dem Namen XSLPattern bekannt ist. Hierbei handelt es sich um eine XPath-Variante, die um Elemente von XQL erweitert wurde.

XPath-Ausdrücke werden von XML-Parsern und von XSL-Prozessoren verstanden, daher sind sie sowohl in XSL-Stylesheets als auch im XML-DOM nutzbar.

Die Idee hinter XPath steckt schon im Namen. Da XML-Dokumente mit Verzeichnisstrukturen verglichen werden können, können wir auf ein Objekt oder eine Gruppe von Objekten durch die Angabe eines Pfades verweisen. Das funktioniert so ähnlich wie das gute alte dir-Kommando unter DOS.

5.1 Die Syntax

XPath unterstützt eine sehr einfache und kurze Schreibweise für die Erstellung von Abfragen in XML-Dokumenten. Da vieles durch die intuitive Syntax von XPath selbsterklärend ist, beginnen wir mit einer kurzen Liste von Beispielen, bevor wir im einzelnen auf XPath eingehen.

Alle Beispiele werden in bezug auf den globalen Kontext der XML-Datei ausgeführt, da die Ausdrücke mit dem Kontext-Operator `//` beginnen. Im Abschnitt Kontext-Operatoren gehen wir auf die verschiedenen Operatoren näher ein.

//field	Alle Elemente, die den Namen `field` haben, werden von diesem Ausdruck zurückgeliefert, ganz gleich, wo sie sich in der Hierarchie des Dokuments befinden.
//field[@name]	Alle Elemente, die den Namen `field` haben und ein Attribut `name` enthalten, werden von diesem Ausdruck zurückgeliefert, ganz gleich, wo sie sich in der Hierarchie des Dokuments befinden.
//field[@name='Comment']	Alle Elemente, die den Namen `field` haben und ein Attribut `name` enthalten, dessen Wert »Comment« ist, werden von diesem Ausdruck zurückgeliefert, ganz gleich, wo sie sich in der Hierarchie des Dokuments befinden.
//field[2]	Selektiert das zweite auftretende Element mit Namen `field` innerhalb der XML-Datei.

Tabelle 5.1

Die Ergebnismenge dieser Abfragen ist abhängig von der Zugriffsmethode des DOMs, und von der Umgebung (XML oder XSL), in der sie ausgeführt wird.

Die DOM-Methode `selectNodes` liefert entweder ein Node-Set zurück oder `NULL`.

Die DOM-Methode `selectsingleNode` liefert entweder ein Node-Element zurück oder `NULL`.

In XSL wird immer ein Node-Set zurückgeliefert oder `NULL`.

Node-Sets enthalten die durch einen XPath-Ausdruck generierte Ergebnismenge. Hierbei bleibt sowohl die Reihenfolge als auch die Struktur erhalten, wie sie auch im gesamten XML-Dokument vorliegt.

5.2 Operatoren in XPath

5.2.1 Vergleichs-Operatoren

Innerhalb eines XPath-Ausdrucks können fast alle gebräuchlichen Vergleichsoperatoren eingesetzt werden. Dadurch sind auch sehr komplexe Abfragen möglich. Diese werden jedoch leicht unübersichtlich, wenn sie im globalen Kontext ausgeführt werden.

Die folgende Tabelle zeigt eine Aufstellung der in XPath zur Verfügung stehenden Vergleichs-Operatoren. Neben der vom Microsoft-Parser genutzten Darstellung der Operatoren gibt es noch die des W3C und eine Kurzform (ebenfalls W3C-compliant). Microsoft unterstützt aber mit seinen neueren Parsern die offizielle Schreibweise.

Operator	W3C	Shortcut	Beschreibung
And	and	&&	Logisches und
Or	or	||	Logisches oder
not()	not		Verneinung
=	eq		Gleich
!=	ne		Ungleich
<	lt		Kleiner als
<=	le		Kleiner gleich
>	gt		Größer als
>=	ge		Größer gleich
|			Liefert den vereinigten Wert zweier Nodes.

Tabelle 5.2

Hier einige Beispiele zu den Vergleichsoperatoren:

- **folder[folder and file]**
  liefert uns alle `folder`, die auch mindestens ein `file` enthalten
- **folder[not(folder) and not(file)]**
  liefert uns alle `folder`, die weder `folders` noch `files` enthalten

- **folder[@size gt 1000]**
  liefert uns alle folder, deren Size-Attribut größer als 1000 ist
- **folder[index() le 2]**
  liefert uns die ersten drei folder

Wie in allen Programmiersprachen üblich, ist auch in XPath eine Rangfolge der Wertigkeiten in der Spezifikation verankert:

1.	( )	Gruppierung
2.	[ ]	Filterausdruck
3.	!	Methodenaufruf
4.	/ //	Pfadoperationen
5.	any all	Zuweisungsoperationen
6.	= != lt le gt ge eq ne ieq	Vergleichsoperationen
7.	|	Vereinigungen
8.	not()	Logische Verneinung
9.	And	Logisches und
10.	Or	Logisches oder

Tabelle 5.3

5.2.2 Numerische Operationen

Berechnungen in XPath sind möglich, erleichtern jedoch auch nicht die Übersicht und sollten daher zurückhaltend eingesetzt werden. Zudem bremsen sie die Verarbeitung.

Die folgenden Zeichen stehen für numerische Operationen zur Verfügung.

Operator	Beschreibung
+	Addition
-	Subtraktion
*	Multiplikation

Tabelle 5.4

Operator	Beschreibung
Div	Division
Mod	Modulare Division

Tabelle 5.4

5.2.3 Kontext-Operatoren

Die Kontext-Operatoren sind ein wesentliches Mittel zur Navigation in XML-Dokumenten. Insbesondere beim Verfassen von XSL-Stylesheets und der Anwendung von Templates in XSL ist es wichtig zu kontrollieren, in welchem Kontext man sich befindet. Der Kontext wird durch das Element beschrieben, auf dem der Parser im Moment der Ausführung des XPath-Ausdrucks positioniert ist. Der //-Operator führt die Abfrage in allen Kontexten des Dokuments aus.

/	Selektiert die Kind-Nodes des Nodes, in dessen Kontext man sich befindet.
//	Selektiert alle im Dokument enthaltenen Nodes.
.	Selektiert den Node, in dessen Kontext man sich befindet.
..	Selektiert den übergeordneten Node des Nodes in dessen Kontext man sich befindet.
*	Selektiert alle Nodes, unabhängig ihrer Bezeichnung (Wildcard).
@	Präfix für den Zugriff auf Attribute.
@*	Selektiert alle Attribute, unabhängig von ihrer Bezeichnung (Wildcard).
:	Separator für Namespace-Bezeichnung und Element-Bezeichnung.
!()	Ordnet eine Funktion dem spezifizierten Node zu.
()	Dient dem Gruppieren von Ausdrücken.
[]	Dient dem Anwenden von Filterausdrücken.
current()	Liefert den aktuellen Kontext-Node.
id()	Liefert den Node mit der angegebenen ID.

Tabelle 5.5

5.2.4 Wechseln des Kontexts mit der anchestor-Funktion

Die **anchestor**-Funktion liefert den nächstmöglichen Vorfahren zurück, der dem im Argument übergebenen Suchausdruck entspricht. Dabei besteht die Rückgabemenge entweder aus dem gefundenen Node oder Null.

Es ist nicht möglich, die ancestor-Funktion innerhalb eines XPath-Ausdrucks rechts von einem Kontext-Operator zur verwenden, da sich `anchestor` nur auf das aktuelle Element beziehen kann.

Um innerhalb eines Dokuments das letzte vorangehende strukturgebende Element zu finden, wäre der folgende Ausdruck möglich:

```
anchestor(*[@outlinelevel])
```

Vorausgesetzt, alle strukturgebenden Elemente würde ein Outline-Level haben, nicht strukturgebende Elemente keines.

Die vorangehende Überschrift der Ebene 1 liefert uns folgender Ausdruck:

```
anchestor(heading[@level='1'])
```

5.2.5 Einsatz der context-Funktion

Die **context**-Funktion ist nützlich, um Werte aus dem aktuellen Element in den Pfadausdruck einzubeziehen. So liefert der folgende Ausdruck alle Dateien in einer Version, deren **extension**-Attribut dem der Datei im aktuellen Kontext entspricht

```
//file[.=context()/@extension]
```

Die **context**-Funktion kann auch mit einem Index-Parameter inititalisiert werden. Dabei liefert

- der Index (0) die Wurzel des Dokuments
- der Index (-1) den aktuellen Kontext, ist also gleichbedeutend mit `context()`
- der Index (-2) den Vorfahren, also gleichbedeutend mit `anchestor()`
- jeder weitere negative Index zeigt auf den in der Hierarchie jeweils höheren Kontext

5.2.6 Einsatz der id-Funktion

Die id-Funktion setzt voraus, daß in einer dem XML-Dokument zugeordneten DTD ein Attribut mit dem Typ ID spezifiert wurde. Über diese ID können dann Querverweise verwaltet oder das Element mit einer bestimmten ID geladen werden.

Wir betrachten zunächst folgende XML-Datei als Beispiel eines Verzeichnissystems:

```
<?xml version="1.0" encoding="iso8859-1"?>
<drive>
  <folder id="1" name="doc">
    <file id="11" name="myFile.ext">
  </folder>
  <folder id ="2" name="favorites">
   <file link="11"/>
  </folder>
</drive>
```

Der erste **doc**-Ordner, enthält eine Datei mit Namen **myFile.ext**. Der zweite Ordner ist ein **Favorites**-Ordner, der nur Verweise auf Dateien enthält. Den Verweis drücken wir durch das **link**-Attribut aus, dessen Wert die ID der referenzierten Datei enthält. Die Benamung spielt für die Fumktionalität keine Rolle, entscheidend ist hier nur die Deklaration der ID in der DTD.

```
//file[@id='2']/id(@link)
```

liefert uns jetzt das Element

```
<file id="11" name="myFile.ext">
```

zurück.

5.2.7 Verwendung von Wildcards

In XPath existiert als Wildcard nur das *-Zeichen. Der * kann sowohl als Wildcard für Element- als auch für Attribut-Name oder -Wert eingesetzt werden:

*[@name]	Selektiert alle im Kontext des aktuellen Nodes befindlichen Elemente, die ein Attribut mit der Bezeichnung »name« enthalten.

Tabelle 5.6

5.2.8 Verwendung des Index

Mittels des in eckigen Klammer eingeschlossenden Index kann in einem Node-Set explizit auf ein Node zugegriffen werden. Der Index in Node-Sets beginnt bei 0.

Bei allen XPath-Bedingungen, die Bedingungen enthalten, die in eckige Klammern gefaßt sind, ist zu beachten, daß diese eine höhere Bedeutung haben als Ausdrücke, die / oder // enthalten.

Im folgenden Beispiel bedeutet dies, daß die Bedingung

```
//comment()[3]
```

alle Kommentare mit dem Index 3 zurückliefert. Dies geschieht, da die Bedingung wie folgt interpretiert wird.

- Erst alle Kommentare filtern
- Dann alle mit dem Index (3) filtern

Durch Klammern verdeutlicht bedeutet dies: `//(comment()[3])`

Möchte man jedoch den 3., im XML-Dokument enthaltenen Kommentar zurückgeliefert bekommen, so muß die Bedingung folgendermaßen aussehen.

```
(//comment())[3]
```

(//field)[3]	Selektiert das field-Element mit dem Index 3 innerhalb aller field-Elemente.

Tabelle 5.7

5.2.9 Typunterscheidung von Nodes

Da es manchmal wünschenswert ist, eine Selektion über den Node-Typ zu machen, stellt XPath die folgenden Funktionen zur Verfügung.

Operator	Beschreibung
node()	Liefert das Node-Element
text()	Liefert den Text des Elements, Leerzeichen bleiben erhalten

Tabelle 5.8

Operator	Beschreibung
comment()	Für Comment-Nodes
processing-instruction()	Für Processing-Instruction-Nodes
*	Für alle Node-Typen abhängig vom Pfad (Element/Attribut)
value()	Liefert den Wert des Elements

Tabelle 5.8

Diese Operatoren sind größtenteils Erweiterungen zur W3C-Spezifikation. Der Einsatz von **text()** ist in jedem Fall dem von **value()** vorzuziehen, da letzteres nicht korrekt mit Leerzeichen umgeht.

5.3 Weitere XPath-Funktionen

5.3.1 Boolesche Funktionen

Die in XPath enthaltenen booleschen Funktionen dienen der Wertkonvertierung in einen booleschen Wert bzw. der einfachen Lieferung solcher.

Operator	Beschreibung
Boolean()	Konvertiert das Argument in einen booleschen Wert.
false()	Liefert den Wert false zurück.
not()	Liefert den Wert true zurück, wenn das Argument den Wert false hat.
true()	Liefert den Wert true zurück.

Tabelle 5.9

5.3.2 Node-Set-Funktionen

Die Node-Set-Funktionen dienen dem erweiterten Filtern und Selektieren in Node-Sets.

Operator	Beschreibung
count()	Liefert die Anzahl der Nodes im Node-Set.
end()	Selektiert den letzten Node im Node-Set.
local-name()	Liefert den lokalen Teil des erweiterten Namens.
position()	Liefert den Index des aktuellen Node im Node-Set.

Tabelle 5.10

5.3.3 Numerische Funktionen

Die numerischen Funktionen in XPath dienen der Wertkonvertierung in Zahlen.

Operator	Beschreibung
number()	Konvertiert das Argument in einen numerischen Wert.

Tabelle 5.11

Damit ein Type-Casting mit dem MSXML-Parser funktioniert, erwartet dieser ein XML-Data-Schema, in dem die Datentypen der Elemente beschrieben werden. Man kann dieses Verhalten mit einem Trick umgehen, den wir im Kapitel »Transformation mit XSL-Stylesheets« anwenden. Es erscheint nicht unbedingt sinnvoll, bei der Entwicklung zu XML-Schema noch auf das proprietäre XML-Data zu setzen.

Eine weitere Auseinandersetzung mit XPath beinhalten die Kapitel zum XML-DOM und zur Transformation mit XSL-Stylesheets. Wir werden dort XPath in einigen Beispielen anwenden.

6 Grundtechniken für die Erstellung der Web-Anwendung

In diesem Kapitel wird auf die erforderlichen Grundtechniken zur Realisation von Web-Anwendungen eingegangen. Die Vor- und Nachteile der etablierten Konzepte für Web-Anwendungen werden gegenübergestellt.

6.1 XML-Applikation versus herkömmliche Web-Anwendung

Was unterscheidet unsere XML-Applikation von herkömmlichen Web-Anwendungen, und worin liegen die Vorteile einer solchen Lösung? Im Vergleich dazu die üblichen Wege für die Umsetzung von Web-Anwendungen.

6.1.1 Statische Webseiten

Statische Webseiten sind mit entsprechenden HTML-Editoren einfach zu erstellen. Sie werden im professionellen Umfeld nur noch als Rahmen für dynamische Sites eingesetzt.

Vorteile
Einfache Erstellung
Geringe Serverlast bei vielen Zugriffen
Nachteile
Aufwendig zu pflegen
Der Zugriff auf aktuelle Daten ist nicht möglich
Für mittlere und große Sites nicht praktikabel
Jede Aktualisierung der Site muß aufwendig im Quellcode der Seiten durchgeführt werden
Verknüpfung von Inhalt und Layout

Tabelle 6.1 Vor- und Nachteile statischer Seiten

6.1.2 Datenbankgestützte Dynamische Webseiten

Datenbankgestützte Dynamische Webseiten werden in der Regel mit Scriptererweiterungen des WebServers realisiert. Als Scripts kommen Perl, JavaScript, VBScript und proprietäre Sprachen zum Einsatz.

Vorteile
Einfachere Pflege der Inhalte
Strukturierte Verwaltung der Daten in Tabellen
Verwaltung von Meta-Informationen zu den Inhalten
Nachteile
stark Skript-lastig
viel Code für die Aufbereitung der Datenbankdaten in HTML erforderlich
Anwendungen mit vielen und langen Skripts sind schlecht wartbar
Hohe Serverlast bei vielen Zugriffen
weitgehend statisches Seitenlayout

Tabelle 6.2 Vor- und Nachteile datenbankgestützer Seiten

6.1.3 Webseiten mit aktiven Client-Komponenten

Aktive Client-Komponenten werden als Applets oder ActiveX-Komponenten erstellt. Sie werden sowohl auf statischen wie auf dynamischen Seiten eingesetzt. Es kann zwischen zwei Arten von aktiven Komponenten unterschieden werden:

- Player, die als Abspielgeräte für Medienformate genutzt werden (z. B. Media Player, Flash, ...)
- Anwendungskomponenten, die Programmfunktionalität bereitstellen (z. B. Edit-Controls, Verschlüsselungs-Applets)

Player werden in der Regel als sicher angesehen und bedenkenlos eingesetzt. Bei Anwendungskomponenten sind Internet-Nutzer mit Recht vorsichtig und kritisch eingestellt.

Vorteile
Kapselung des Codes für Anwendungssteuerung
Kapselung des Codes für die Darstellung
Gute Möglichkeiten zur Anwender-Interaktion
Nachteile
Massive Sicherheitsprobleme beim Einsatz von ActiveX-Komponenten
Mögliche Sicherheitsprobleme beim Einsatz von Applets
Hoher Entwicklungsaufwand
Abhängigkeit von der Laufzeitumgebung des lokalen Systems (DLLs, JDK-Version)

Tabelle 6.3 Vor- und Nachteile von Webseiten mit Anwendungskomponenten

6.1.4 Websites mit aktiven Server-Komponenten

Aktive Server-Komponenten werden als Servlets, ActiveX-Komponenten oder als externe Programme eingebunden. Eine besonderes hohe Integration mit dem WebServer kann durch Server-Erweiterungen realisiert werden, die das API (Application Programming Interface) des Servers nutzen. Die APIs von Webservern wie IIS, Fasttrack oder Apache sind für diesen Zweck vorgesehen.

6.1.5 Scripting

Skripts sind für die interaktive Gestaltung einer Web-Anwendung unerläßlich.

Man kann davon ausgehen, daß die große Mehrheit der Anwender die Ausführung von Skripts zuläßt. Die Skripts werden vom Browser ausgeführt. Das von Netscape eingeführte JavaScript ist auf den wesentlichen Browsern verfügbar.

Auf der Serverseite übernimmt Scripting die Aufgabe des »Glue Between«, des verbindenden Elements, das aktive Komponenten miteinander verknüpft.

6.1.6 Anforderung an Web-Anwendungen

Eine Web-Anwendung wird anderen Anforderungen unterworfen als eine Desktop-Anwendung.

- Die Anwendungen müssen mit einer geringen Bandbreite zur Übermittlung der Daten auskommen.
- Das lokale Speichern von Daten ist nicht möglich.
- Unterschiedliche Clientsysteme müssen bedient werden.
- Besondere Sicherheitsaspekte müssen berücksichtigt werden.
- Die lokale Verarbeitung von Daten ist nur eingeschränkt möglich

Als Web-Benutzerschnittstelle wird meist ein Web-Browser eingesetzt.

Zur Zeit sind überwiegend die Web-Browser von Microsoft und Netscape im Einsatz. Die HTML-Implementierung dieser Browser weicht voneinander ab, so daß Web-Anwendungen häufig verschiedene Seiten gleichen Inhalts vorhalten, die für den jeweiligen Browser-Typ optimiert sind.

Zunehmend wird das Internet auch von anderen Endgeräten genutzt, die in der Regel nicht über den Leistungsumfang eines PC-basierten Browsers verfügen. Dies können z.B. Informationsterminals, Verkaufsterminals oder mobile Endgeräte sein.

Gemeinsam gilt für alle Endgeräte:

- Nur wenige verschiedene Kontrollelemente sind verfügbar
- Die Benutzerführung ist auf ein Anwendungsfenster reduziert. In eingeschränktem Umfang können zusätzlich modale Dialogfenster eingesetzt werden.

6.1.7 Architektur einer Web-Anwendung

Web-Anwendungen werden in einer mehrschichtigen Architektur realisiert

Die Benutzerschnittstelle dient der Darstellung der Daten und zur Verarbeitung der Benutzerereignisse.	
Als Transportschicht werden Internet-Techniken eingesetzt.	
Die Web-Anwendung dient als Kommunikationsplattform zwischen dem Web-Client und Datenbanken, sowie den Komponenten, die die Verarbeitungslogik beinhalten.	
Die Web-Anwendung wird auf dem Web-Server ausgeführt.	
Anwendungsbibliotheken stellen der Web-Anwendung Verarbeitungslogik zur Verfügung	Anwendungsbibliotheken stellen der Web-Anwendung Dienste für den Zugriff auf die Daten zur Verfügung
Datenbanken, Dateisysteme und Archive verwalten die Daten und weitere Ressourcen (Dokumente, Inhalte, Grafiken, ...) der Anwendung.	

Tabelle 6.4

6.2 Lösung in XML

Probleme bei heutigen Webanwendungen

Alle genannten Ansätze (und ihre Mischformen) haben ein gemeinsames Problem:

Eine Trennung zwischen Inhalt und Darstellung ist kaum möglich.

Meist enthalten die einzelnen Seiten gleichzeitig Inhalt, Layoutanweisungen und sogar die Geschäftsregeln (Business Logic) der Anwendung. Insbesondere ASP-Anwendungen ufern schnell in sehr unübersichtliche Seiten aus und sind entsprechend aufwendig in der Wartung.

Vorteile der Lösung in XML

Unter anderem wurde XML zur Lösung dieser Probleme geschaffen.

Die Vorteile einer XML-Lösung sind:

- Darstellung durch Transformation von XML in HTML
- Datenaustausch durch Verteilung von XML-Dateien
- Bessere Strukturierung der Anwendung durch klare Trennung von Inhalt und Darstellung
- Zukunftssichererheit der Anwendung durch Einsatz eines offenen Standards
- Einfache Anpassung für unterschiedliche Endgeräte

Konzept einer XML-Lösung

Eine XML-basierte Anwendung fügt zwischen Datenzugriff und grafischer Aufbereitung der Daten eine weitere Schicht ein. Alle Daten werden zunächst in XML konvertiert, und dieses wird für die Anzeige in HTML umgewandelt. Dieser Zwischenschritt entkoppelt Daten und Darstellung.

Während bei bisherigen Anwendungen die »Verpackung« der Daten in HTML nur mit proprietären Ansätzen gelöst werden konnte, stehen jetzt öffentliche Standards für diese Aufgabe zur Verfügung.

Wo zuvor umfangreiche Skripts die Erzeugung der Seite übernahmen, werden jetzt XSL-Stylesheets eingesetzt.

Das Vorgehen bei einer XML-basierten Anwendung ist im allgemeinen wie folgt:

- Transformation der Daten in XML durch Werkzeuge der datentragenden Systemen, oder eigene Programme.
- Transformation der so gewonnenen XML-Dateien in verschiedene Ausgabeformen durch XSL-Stylesheets.

Auf diese beiden Kernbereiche von XML-Anwendungen wird in den beiden folgenden Kapiteln, »Erzeugen von XML aus Datenbanken« und »Transformation mit XSL-Stylesheets«, eingegangen.

7 Erzeugen von XML aus Datenbanken

In diesem Kapitel stellen wir unterschiedliche Ansätze für die Generierung von XML aus Datenbanken vor. Wir zeigen, wie Abfrageinhalte und Ergebnismengen in XML-Dateien geschrieben werden und wie in einer Webanwendung mit dynamisch erzeugten XML-Fragmenten umgegangen wird.

7.1 Übersicht über das Projekt xmlDbLayer

XML aus Datenbanken zu generieren ist eine Aufgabe, mit der wir in unseren XML-Anwendungen häufig konfrontiert werden. Es lieg also nahe, eine flexible und wiederverwertbare Komponente hierfür zu erstellen. Diese Komponente wird es uns ermöglichen, sowohl »ADO-XML« als auch ein für die Erstellung von Stylesheets handlicheres Format zu erzeugen.

Um die Schnittstelle möglichst unabhängig zu gestalten, wird unser Generator mit SQL-Statements gefüttert und gibt uns XML-Fragmente zurück. Die Fragmente können auf verschiedenene Arten zurückgegeben werden:

- als XML-String
- als XML-Datei
- als XML-Node-Objekt
- als XML-Document-Objekt

Damit sind wir für praktisch alle Anwendungsfälle gerüstet.

Für die Datenbankschnittstelle setzen wir ADO ein, das ab Version 2 Ergebnismengen auch im XML-Format speichern und laden kann. Genau dafür ist das Format, das wir in diesem Buch als »ADO-XML« bezeichnen, auch geeignet. Stylesheets können hiermit nur schwierig realisiert werden. Zudem können hierarchische Beziehungen nicht dargestellt werden. Unser Generator kann als Entwurfsvorlage für eigene XML-Generatoren eingesetzt werden, die weitere Features enthalten oder andere XML-Darstellungen oder Datenquellen bedienen.

7.2 Die Beispiel-Datenbank

SQL-DDL-Skripts auf der CD

Wir setzen bei unserem Beispiel eine **Access.mdb** ein. Die SQL-Skripte für die Generierung der Tabellen sind auf der Buch-CD enthalten, so daß das Beispiel auch auf einem SQL-Server nachvollzogen werden kann. In unserer Datenbank verwalten wir Dokumente. Es sind die Dokumente, die zu diesem Buch gehören. Wir haben die Dokumente nach Kategorien unterschieden, die wir in unserem Beispiel Dokumenttypen nennen. Dieses Modell ist auf andere Dokument- oder Datensammlungen einfach übertragbar und erweiterbar.

Die zentrale Tabelle unserer Datenbank ist `T_Documents`. Sie verwaltet einen Standardsatz beschreibender Attribute zu den Dokumenten. Hierzu gehören eine eindeutige **pkDocID**, die Primärschlüssel der Tabelle ist, Angaben zum Speicherort der Datei sowie zum Bearbeitungsstatus. In unserem Beispiel sollen Dokumente nach ihrer Erstellung für den Zugriff freigegeben werden. Die diesen Prozeß begleitenden Statusinformationen werden ebenfalls in dieser Tabelle verwaltet. Wir werden unterschiedliche Sichten für Editoren und für Leser zur Verfügung stellen, die sich am Status der Dokumente orientieren. So werden den Lesern nur freigegebene Dokumente angezeigt, oder es wird erkennbar, ob ein Dokument in der Bearbeitung ist.

Mögliche Erweiterung: Dokumentspezifische Attribute können durch eine ergänzende Tabelle DocumentType_Att abgebildet werden, wobei DocumentType der Name des Dokumenttyps wäre.

Name	Typ	Größe
Name	Text	50
pkDocId	Long Integer	4
Path	Text	50
Comment	Text	255
ToBeDeployed	Yes/No	1
DocType	Long Integer	4
Category	Long Integer	4
Status	Long Integer	4
Used	Long Integer	4
Checked	Date/Time	8
CheckedBy	Text	50

Tabelle 7.1 Feldliste der Tabelle T_Document

In der Tabelle T_Dependency können wir Abhängigkeiten zwischen Dateien erfassen. Je nach Anwendungssituation können die Abhängigkeiten auch als Parent-Child-Beziehung interpretiert werden. In unserem Beispiel dienen uns die Abhängigkeiten für die Zuordnung von XML-Dateien und zugehörigen Stylesheets, aber Zuordnungen wie Buch-, Kapitel-, oder Dokument-Abschnitt sind denkbar.

Zusätzliche Daten werden in den vier Lookup-Tabellen verwaltet, die zu T_Document alle in 1:n- Beziehung stehen. Die Beziehungen und Felder werden hier grafisch dargestellt:

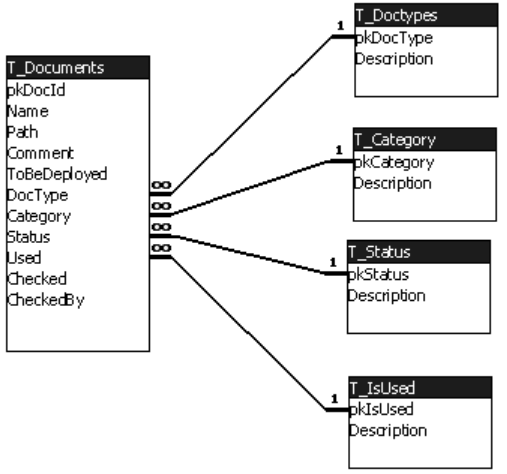

Abbildung 7.1 Diagramm Dokumente und Lookups

7.3 Datenzugriffsschicht

Der Zugriff auf die Daten wird in unserer Anwendung durch eine eigene Schicht (auch als **Layer** oder **Tier** bezeichnet) abgebildet.

Anwendungsklassen

Wir werden uns verschiedene Abfragen erstellen, die Views auf unsere Daten darstellen. Die Abfragen werden in anwendungsspezifischen Klassen gebildet, die wir im folgenden als Anwendungsklassen bezeichnen. Sie modellieren die realen Objekte der Anwendung (in unserem Fall Dokumente). Die Anwendungsklassen enthalten unter anderem SQL-Statements und Methoden, die das Einfügen von Parametern in die Statements vereinfachen.

Datenzugriffsklassen

DbResultset stellt die Schnittstelle zur Datenbank bereit und integriert diese mit unserem XML-Dienstleister, der Klasse **XMLDoc**. Die für die Datenquelle spezifischen Dinge werden durch die Anwendungsklasse gekapselt. Alle Anwendungsklassen nutzen **DbResultset**, um ihre Statements abzusetzen und XML zu generieren. Zur Erzeugung der Ergebnismenge rufen wir die Methode **FetchXMLDoc** auf.

Zusammen mit der Klasse DbLayer wird so die Datenbankschnittstelle gebildet. Diese ist für die Verwaltung der Datenbank-Verbindung zuständig.

Die Klasse **DbLayer** übernimmt die Erzeugung und Verwaltung der Verbindungen zur Datenquelle. Da wir uns hier auf ADO-Datenzugriff beschränken, genügen zum Aufbau einer Verbindung die Informationen:

```
Username
Password
DSN
```

Diese werden als Parameter an die **ADOLogin**-Methode übergeben, die versucht, eine Verbindung zur Datenquelle aufzubauen. War dies erfolgreich, wird ein Aufruf der Methode **CreateDbResult** zukünftig immer ein **DbResultset**-Objekt instanziieren und zurückgeben. Da hinreichend in anderen Quellen beschrieben, sparen wir uns hier den Abdruck des Logins und wenden uns direkt der Klasse **XMLDoc** zu. Wir kommen in deren Kontext auf die Klasse DbResultset zurück.

7.3.1 Die Klasse XMLDoc

In diese Klassen werden wir die Transformation einer **DbResultset**-Ergebnismenge in XML vornehmen. **XMLDoc** stellt dafür entsprechend den eingangs des Kapitels erwähnten XML-Formaten verschiedene Methoden zur Verfügung:

- GetXMLFragmentString
- GetXMLDocumentString
- GetXMLFragment
- GetXMLDocument

Darüber hinaus wird die Transformation durch eine Methode

```
ApplyStylesheet
```

unterstützt.

Über die Eigenschaft

```
ADOResultset
```

wird eine **XMLDoc**-Instanz mit einer ADO-Ergebnismenge initialisiert. Erst nach der Initialisierung sie die anderen Methode verfügbar. Die Initialisierung wird beim ersten Aufruf der Eigenschaft **XMLDoc** der Klasse **DbResult** geleistet. Zur Veranschaulichung eine typische Aufrufsequenz in VbScript:

```
'Erzeugen einer Instanz unserer Anwendung
Set myDbLayer = CreateObject("xmlDbLayer.DbLayer")
'Login an die Datenbank
myDbLayer.Login "","","xmlBook"
'Erstellen eines Resultset-Containers
Set myResultset = myDbLayer.CreateDbResultset()
'Ausführen einer Suche
Set myDoc=myResultset.FetchXMLDoc("Select * From T_Documents Where
pkDocId<3")
'Schreiben des Ergebnisses in eine Datei
Set pFSO=CreateObject("Scripting.FileSystemObject")
Set pStream=pFSO.CreateTextFile("C:\Test.xml")
pStream.Write MyDoc.XMLDocumentString
pStream.CloseSet myResultset=Nothing
myDbLayer.Logout
Set myDbLayer=Nothing
```

7.4 Erstellen des Projekts xmlDbLayer

Wir legen jetzt das VB-Projekt **xmlDbLayer** an. Als Projekttyp wählen wir ActiveX DLL aus, damit wir auch diese Komponente problemlos auf dem Webserver einsetzen können. Dafür sind noch weitere Projekteinstellungen erforderlich, wie sie auf der folgenden Abbildung erkennbar sind:

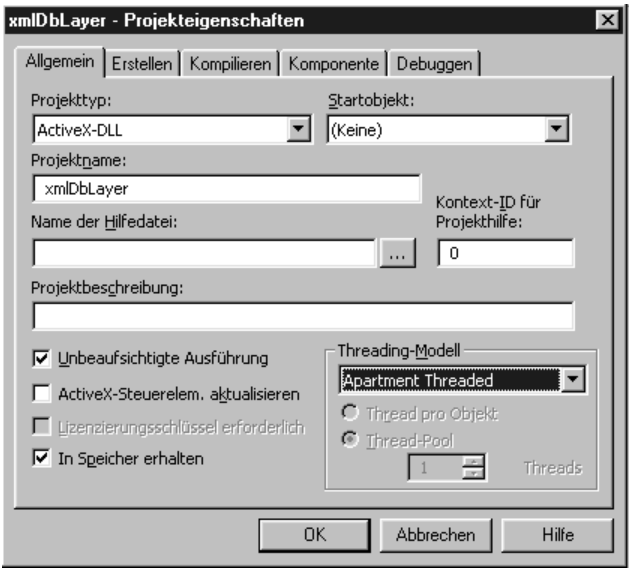

Abbildung 7.2 Projekteinstellungen für eine ActiveX-DLL für den Einsatz mit dem IIS/PWS

Die Einstellungen

- Unbeaufsichtigte Ausführung
- Threading Model: Apartment Threaded

sind Voraussetzung für den Einsatz mit ASP.

Nun fügen wir über **Projekt/Verweise** eine Referenz auf ADO hinzu. Wir nutzen hier die ADO-Version 2.1 (auch mit späteren Versionen ist dieses Beispiel ohne Änderungen ausführbar). Im Projekt sind die Klassen der ADO-Bibliothek nun über **ADODB.Klassenname** einsetzbar.

Implementierung der Klasse DbLayer

Wir erstellen jetzt die Klasse **DbLayer** und erstellen dort eine Instanzvariable für die Datenbankverbindung:

```
Private mConnection As ADODB.Connection
```

Die Methode **Login** initialisiert das **Connection**-Objekt:

```
Public Function Login(Username As String, Password As String, DSN As
String) As Long
mConnection.ConnectionString = "DSN=" & DSN & ";UID=" & Username &
";PWD=" & Password & ";"
mConnection.Open
.....
```

Die nächste Methode werden wir zur Erzeugung von DbResultsets einsetzen, sie erzeugt eine neue **DbResultset**-Instanz. Anschliessend wird die Instanz mit einem Kommunikationskanal zur Datenbank in Form einer Connection versorgt. Die Hilfsvariable, die wir zur Erzeugung der Instanz genutzt haben, wird am Ende der Methode zerstört. Die zurückgegebene Referenz kann nun verwendet werden.

```
Public Function CreateDbResultset() As DbResultset
    Dim pDbResultSet As New DbResultset
    Set pDbResultSet.Connection = mConnection
      Set CreateDbResultset = pDbResultset
    Set pDbResultSet = Nothing
End Function
```

Das letzte Mitglied der Klasse ist die **Logout**-Methode. Mit ihr können wir uns an der Datenbank abmelden und die Verbindung freigeben:

```
Public Sub Logout()
    mConnection.Close
End Sub
```

Die Objektreferenz auf den Verbindungskanal zerstören wir im **Terminate-Event** der Klasse:

```
Private Sub Class_Terminate()
    Set mConnection = Nothing
End Sub
```

Implementierung von DbResultset

Nun wenden wir uns der Klasse **DbResultset** zu, die wir mit der Instancing-Eigenschaft erstellen. Eine Instanz kann nur durch den Aufruf von **Create DbResultset** erzeugt werden. In dieser Methode wird die **DbResultset**-Instanz mit einer Datenbankverbindung versorgt:

```
'Class DbResultset
Option Explicit
Private mConnection As ADODB.Connection
Friend Property Set Connection(NewConnection As ADODB.Connection)
    Set mConnection = NewConnection
End Property
```

Jetzt können wir innerhalb von **DbResultset** die Verbindung für die datenladenden Methoden benutzen. Wurden die Daten erfolgreich geladen, wird das Ergebnis in ein **XMLDoc**-Objekt verpackt und so zurückgegeben. Dafür setzen wir die Methode **FetchXMLDoc** ein, die als Eingangsparameter ein SQL-Statement erhält und ein **XMLDoc**-Objekt zurückgibt.

Implementierung der Klasse xmlDoc

Die Klasse **xmlDoc** muß noch von uns implementiert werden. Sie wird die benötigten XML-Funktionalitäten und die Methoden zur Umwandlung der Ergebnismenge in XML beinhalten. Wir werden ein in der Klasse **DbResult** erzeugtes **Recordset** in einer eigenen Objektreferenz verwalten. Dafür verwenden wir die Modulvariable **mResultset**, die über die Property **ADOResultset** bedient wird.

```
'Class xmlDoc
Option Explicit
Private mResultset As ADODB.Recordset
Public Property Set ADOResultset(NewResultset As ADODB.Recordset)
    Set mResultset = NewResultset
End Property

Public Property Get ADOResultset() As ADODB.Recordset
    Set ADOResultset = mResultset
End Property
```

Neben den Objekten der Datenbank-Schnittstelle gilt es noch die XML-Objekte zu verwalten. Für diese Aufgaben fügen wir dem Projekt eine Referenz auf die **MSXML-DLL** hinzu. Nachdem wir dies erledigt haben, kehren wir zur Klasse xmlDoc zurück. Wir werden eine Objektreferenz auf das XML-Dokument in der **xmlDoc**-Klasse verwalten und ergänzen daher am Ende des Deklarationsteils der Klasse:

```
Private mDocument As MSXML.Document
Public Property Get XMLDocument() As MSXML.DOMDocument
    If mDocument Is Nothing Then
        Set mDocument = LoadXMLDocument(, "<?xml version='1.0'?>" &
XMLFragmentString)
    End If
    Set XMLDocument = mDocument
End Property
```

Schnittstellen-Verbindlichkeit durch Implements-Anweisung

Die Rümpfe der schon eingangs besprochenen Methoden, die wir für die Rückgabe von XML benötigen, werden eingefügt. In einem größeren Projekt kann es sinnvoll sein, eine eigene Klasse zu erstellen, die diese Methodenrümpfe enthält. Diese Klasse wird dann durch eine Implements-Anweisung als verbindliche Schnittstellenbeschreibung verwendet. So kann sichergestellt werden, daß alle XML-generierenden Objekte diese Methode bereitstellen.

Die Methoden und Eigenschaften in der Reihenfolge, in der wir sie vorstellen:

Name	Rückgabetyp	Beschreibung
Property Get XMLDocument()	MSXML.DOMDocument	Liefert Referenz auf ein DOM-Document
Function ApplyStylesheet(Stylesheet As Variant)	String	Wendet Stylesheet auf XML-Datei an und liefert Transformationsergebnis
Function LoadXMLDocument(Optional Filename As String = »«, Optional XMLStream As String = »«)	MSXML.DOMDocument	Lädt Dokument und liefert Referenz auf ein DOMDocument

Tabelle 7.2

Name	Rückgabetyp	Beschreibung
Property XMLRaw()	String	Liefert den ungeparsten XML Stream des Dokuments
Public Property Get XMLFragmentString()	String	Liefert den XMLStream des Root-Elements
Public Property Get XMLDocumentString()	String	Liefert den XMLStream des Dokuments
Public Property Get XMLFragment()	MSXML.IXML-DOMNode	Liefert den XMLNode des Root-Elements

Tabelle 7.2

Die Property **XMLDocument** haben wir oben bereits erläutert, wir fahren hier mit der Methode **ApplyStylesheet** fort.

Die Methode ApplyStylesheet

Der Name der Methode ist recht selbsterklärend, als Parameter werden wir einen Pfad zu einem XSL-Stylesheet unterstützen, alternativ dazu die Übergabe eines XSL-Document-Objekts. So müssen wir ein häufig benutztes Stylesheet nicht immer wieder neu laden und durch den Parser schicken, sondern können es als Objekt im Arbeitsspeicher vorhalten. Der Rückgabewert ist das Ergebnis der Transformation, die das Stylesheet aus dem **XMLDocument** generiert. Hierfür verwenden wir die `transformNode`-Methode des Nodes, die immer ein XML-Document-Objekt erwartet. Sollte das Stylesheet nicht als Dokument übergeben worden sein, erstellen wir ein Dokument und laden den **XMLStream** hinein.

```
Public Function ApplyStylesheet(Stylesheet As Variant) As String
Dim pStylesheet As MSXML.IXMLDOMNode
    If VarType(Stylesheet) = vbString Then
        Set pStylesheet = LoadXMLDocument(CStr(pStylesheet))
        Else
        Set pStylesheet = Stylesheet
    End If
    ApplyStylesheet = mDocument.transformNode(pStylesheet)
    Set pStylesheet = Nothing
End Function
```

Die Methode LoadXMLDocument

In der **ApplyStyleSheet**-Methode benutzen wir die Methode **LoadXMLDocument**. In ihr kapseln wir das Erzeugen eines XML-Dokuments aus einem XML-Text oder XML-Datei. Im weiteren Verlauf der Anwendung werden wir XML als Text erzeugen und können uns dann mit dieser Methode das Dokument verschaffen, wie wir es für die Transformation benötigen. Für die Erzeugung des XML aus dem ADO-Recordset selbst verwenden wir eine private Eigenschaft der xmlDoc-Klasse, die wir im nächsten Abschnitt einführen.

```
Private Function LoadXMLDocument(Optional Filename As String = "",
Optional XMLStream As String = "") As MSXML.DOMDocument
    Dim pDocument As MSXML.DOMDocument
    Set pDocument = New MSXML.DOMDocument
    If Filename <> "" Then
        pDocument.Load Filename
        Else
        pDocument.loadXML XMLStream
    End If
    Set LoadXMLDocument = pDocument
    Set pDocument = Nothing
End Function
```

Die Property XMLRaw

Die eigentliche Aufgabe der Klasse **xmlDoc** ist die Umwandlung der Ergebnismenge aus der Datenquelle in ein einfach zu verarbeitendes XML-Format. Wir werden das XML hier nicht auf dem **XMLDom** erzeugen, sondern als Text erstellen. Dieses Vorgehen ist ressourcenschonender und performanter. Dies ist bei unserem Beispiel zwar nicht relevant, beim Export in XML als Transferformat und allgemein bei hochlastigen Anwendungssituationen aber eine sinnvolle Alternative.

```
Private Property Get XMLRaw() As String
    Dim pFields As ADODB.Fields
    Dim pField As ADODB.Field
    Dim pXML As String
```

Zunächst deklarieren wir die erforderlichen Variablen, dazu benötigen wir zwei ADO-Objekte: die Fields-Collection und das Field-Objekt, aus dem

wir die Attribute des Datenfeldes lesen können. Wir positionieren beim Aufruf immer auf den ersten Datensatz, da wir sonst nicht sicher sein können, auch alle Daten in der dann folgenden Schleife zu erfassen.

```
If Not mResultset.BOF Then mResultset.MoveFirst
```

Nun folgt die Schleife, in der wir XML generieren. Dabei müssen wir berücksichtigen, daß sowohl Benamungen wie Werte in XML Beschränkungen unterliegen. Bei den Feldnamen aus der Datenbank ist dies relativ unkritisch, was den Einsatz in Attribut- oder Nodewerten betrifft. Da wir davon ausgehen, daß Feldnamen in der Datenbank keine in XML ungültigen Zeichen enthalten, werden wir diese in ein Attribut `name` jedes Datensatz-Nodes schreiben:

```
<field name=" & Chr(34) & pField.Name & Chr(34)/>
```

ergibt in der Ausgabe

```
<field name="pkDocId"/>
```

Nun setzen wir uns mit den Feldwerten auseinander, die wir von der Datenbank zu erwarten haben. Hier können wir, abhängig von den Daten, verschiedene Wege gehen:

Unter der Annahme, es sei sichergestellt, daß keine ungültigen Zeichen in den Werten vorkommen können, können wir alle Werte in Attribute schreiben. Dies ergibt eine kompakte Darstellung, da nur »leere« Tags erstellt werden:

```
<field name="pkDocId" value="38794876"/>
```

oder mit dem Feldnamen als Node-Namen:

```
<pkDocId value="38794876"/>
```

Dies ergibt die kürzeste mögliche Schreibweise. Als Node-Namen werden wir die Feldnamen hier jedoch nicht verwenden, da dies die Erstellung von Stylesheets unnötig erschwert.

Wenn wir nicht ausschließen können, daß in den Feldwerten Zeichen enthalten sind, die in Node-Werten oder Attributwerten nicht verwendet werden dürfen, müssen wir anders vorgehen. Die sicherste Art, Werte zu notieren, ist innerhalb von CDATA-Nodes. In Sonderfällen kann es auch dort zu Problemen kommen, nämlich dann, wenn CDATA-Zeichenfolgen

in den Werten enthalten sind, wie zum Beispiel zwei schließende eckige Klammern **]]**. Solche Sonderfälle treten bei binären Daten, mathematischen Formeln und Dokumenten über XML auf. Hier hilft nur das Encodieren der Daten. Hier erwarten wir diese Daten nicht und geben uns mit der Sicherheit der CDATA-Tags zufrieden:

```
<field name="pkDocId"><![CDATA[38794876]]></field>
```

Dies erfordert jedoch weitaus mehr Zeichen und veranlaßt uns daher zu einer differenzierten Vorgehensweise. Bei den von ADO unterstützten Datentypen, die keine unerlaubten Zeichen zulassen, werden wir daher die CDATA-Tags weglassen und die Werte direkt in den Node schreiben:

```
<field name="pkDocId">38794876</field>
```

Die Datentypen, die wir sicherheitshalber verpacken wollen, erkennen wir an den Konstanten der **ADODB.DataTypeEnum**. Es sind:

- adWChar (130)
- adVarChar (200)
- adLongVarChar (201)
- adVarWChar (202)
- adLongVarBinary (205)

Wir setzen dies in einer **Select-Case**-Struktur in der Schleife um, mit der wir durch die Ergebnismenge laufen:

```
Do Until mResultset.EOF
  'Hier beginnt ein Datensatz
  pXML = pXML & "<dataset>" & vbCrLf
  Set pFields = mResultset.Field
  'Durchlaufen der Felder
  For Each pField In pFields
     pXML = pXML & "<field name=" & Chr(34) & pField.Name & Chr(34)
& ">"
       'Fallunterschiedung nach Datentyp
     Select Case pField.Type
          'adWChar 130
          'adVarChar 200
          'adLongVarChar 201
          'adVarWChar 202
```

```
            'adLongVarBinary 205
            'die kritischen Felder
            Case 130, 200, 201, 202, 205
               pXML = pXML & "<![CDATA[" & pField.Value & "]]>"
        Case Else 'unbedenkliche Felder
               pXML = pXML & pField.Value
        End Select
        pXML = pXML & "</field>" & vbCrLf
    Next
    pXML = pXML & "</dataset>" & vbCrLf
    mResultset.MoveNext
    Loop
```

Die Property XMLFragmentString

Wenn wir keine vollständige XML-Datei erzeugen wollen, sondern nur die Ergebnismenge in XML codiert erhalten wollen, benutzen wir die Methode `XMLFragmentString`. Ihr Ergebnis ist nicht für die direkte Darstellung im Browser geeignet, da z.B. Angaben zum Encoding des Dokuments fehlen. Erst die im folgenden Abschnitt beschriebene Methode `XMLDocumentString` erweitert einen `XMLFragmentString` um die erforderlichen Angaben. `XMLFragmentString` gibt zunächst einmal nur den Wert der privaten Eigenschaft XMLRaw zurück:

```
Public Property Get XMLFragmentString() As String
    XMLFragmentString = XMLRaw()
End Property
```

Die Property XMLDocumentString

XMLDocumentString gibt im Gegensatz zu **XMLFragmentString** ein vollständiges XML-Dokument zurück. Es ist für die Anzeige im Browser oder für das Speichern als Datei geeignet.

```
Public Property Get XMLDocumentString() As String
    XMLDocumentString = "<?xml version='1.0' encoding='ISO-8859-
1'?>" & vbCrLf & XMLRaw
End Property
```

Die Property XMLFragment

Die zu **XMLFragmentString** korrespondierende Eigenschaft **XMLFragment** gibt als Ergebnis ein **XMLDocument**-Objekt zurück, das die Wurzel der Ergebnismenge darstellt.

```
Public Property Get XMLFragment() As MSXML.DOMDocument
Set XMLFragment = LoadXMLDocument(, XMLFragmentString)
End Property
```

7.4.1 Erster Test der Bibliothek

Nachdem wir nun die wichtigsten Elemente kennengelernt haben, ist es Zeit für einen kleinen Test des Moduls. Hierfür kompilieren Sie das Projekt. Wir werden den Test mit einer Skriptdatei und dem Windows Skripting Host durchführen.

Wir nutzen für den Test unsere **FetchXMLDoc**-Methode der **DBResultset**-Klasse, der wir ein vollständiges SQL-Statement übergeben. Der Skript-Code kann praktisch unverändert in einer ASP-Seite eingesetzt werden, dabei wird lediglich der `CreateObject`-Aufruf durch `Server.CreateObject` ersetzt. Dabei fällt auf, daß nur ein Objekt mit `CreateObject` erzeugt wird, nämlich die Hauptklasse der Komponente: **DbLayer**. Alle weiteren Objekte können aus **DBLayer** erstellt werden. Neben der besseren Übersicht hat dies auch klare Performance-Vorteile. Da wir im Skript nur die späte Bindung an eine ActiveX-Komponente einsetzen können, muß jedes Objekt mit **CreateObject** erzeugt werden, was einen unnötigen Aufwand für das Dispatchen der Schnittstelle bei jedem Aufruf produziert. Wir erstellen dagegen alle Objekte innerhalb der Komponente, wo wir die performantere frühe Bindung einsetzen können. Wir benötigen daher nur das Top-Level-Objekt DbLayer:

```
Set myDbLayer = CreateObject("xmlDbLayer.DbLayer")
```

Anschließend führen wir das Login an die Datenbank durch, hier ohne Benutzername und Paßwort:

```
myDbLayer.Login "","","xmlBook"
```

Für das Absetzen einer Abfrage erstellen wir einen **Resultset**-Container mit der **Factory**-Funktion:

```
Set myResultset = myDbLayer.CreateDbResultset()
```

Nun können wir eine Suche ausführen:

```
Set myDoc=myResultset.FetchXMLDoc("Select * From T_Documents Where
pkDocId<3")
```

Rückgabe der Werte

```
MsgBox myDoc.XMLFragmentString
MsgBox myDoc.XMLDocumentString
MsgBox myDoc.XMLFragment.xml
MsgBox myDoc.XMLDocument.xml
```

Zerstören der Referenzen

```
Set myDoc=Nothing
Set myResultset=Nothing
myDbLayer.Logout
Set myDbLayer=Nothing
```

7.4.2 Die Klasse Documents

Den Ideen objektorientierter Softwareentwickling folgend, werden wir die Dokumente als Anwendungsobjekt `Documents` implementieren. `Documents` wird `DbResultset` und die nachgeordneten Schnittstellen nutzen, um XML zu erzeugen. Die größte Leistung von Documents besteht dabei in der Erzeugung der SQL-Statements. Die häufig benötigten Aufrufe an die Datenquelle werden durch Methoden gekapselt, denen die jeweiligen Selektionsparameter übergeben werden können. Wir werden dies hier am Beispiel der Dokumentsicht implementieren.

Zur Erzeugung einer Ergebnismenge benötigen wir immer eine DbResultSet-Instanz. Dafür deklarieren wir eine Referenzvariable `mDbResultSet` auf Modulebene und fügen eine Friend-Eigenschaft `DbResultSet` ein.

```
Private mDbResultset As DbResultSet
Friend Property Set DbResultSet(NewDbResultset As DbResultSet)
    Set mDbResultset = NewDbResultset
End Property
```

Eine Documents-Instanz kann genauso wie eine **DBResultset** nicht von außerhalb der **xmlDbLayer**-Komponenten erstellt werden. Zur Erzeugung für den Nutzer unserer Komponente werden »Factory«-Funktionen eingesetzt, die in der **DbLayer**-Klasse notiert sind. Sie initialiseren die ange-

forderten Objekte mit den erforderlichen internen Referenzen. Aus diesem Grund ist die Property `DbResultSet` auch als Friend Property ausgeführt. Dadurch ist die Property nur innerhalb der Komponente sichtbar. Ein DbResultset wird durch die Methode `CreateDbResultset` der Klasse `DbLayer` erzeugt.

Dokumentsicht

Die Implementierung der Dokumentsicht zeigt uns exemplarisch, wie in einer Anwendungsklasse die Abfragedefinition erstellt wird und die Definition zur Erzeugung der XML-Objekte genutzt wird. Die Erzeugung der Statements kann in unserem Beispiel-Fall unmittelbar in der Anwendungsklasse geschehen, da wir nur ein Datenbankformat unterstützen müssen. Sollten mehrere Formate unterstützt werden, gehört jeglicher direkt datenzugreifender Code aus der Anwendungsklasse entfernt und in einer eigenen Schicht isoliert. Hier arbeiten wir direkt mit einem SQL-Dialekt, wie er von Access oder dem Microsoft SQL-Server verstanden wird.

Eine SQL-Abfrage ist in der Regel in drei Teile gegliedert:

- Beschreibung der Quelltabellen und Felder sowie deren Verknüpfung
- Optionale Beschreibung der anzuwendenden Selektionsausdrücke (Filter)
- Optionale Beschreibung der Sortierkriterien

Für die Dokumentsicht sind die Quelltabellen und Felder wie folgt beschrieben:

```
SELECT T_Documents.pkDocId, T_Documents.Name, T_Documents.Category,
T_Category.Description, T_Doctypes.Description, T_Documents.Path, T_
Documents.Comment, T_Documents.DocType, T_Documents.Status
FROM T_Category INNER JOIN (T_Doctypes INNER JOIN T_Documents ON T_
Doctypes.pkDocType = T_Documents.DocType) ON T_Category.pkCategory =
T_Documents.Category
```

Sollen alle Dokumente angezeigt werden, werden lediglich noch die Sortierkriterien angefügt:

```
ORDER BY T_Documents.Name
```

Aus den folgenden Parametern wird der Suchausdruck gebildet: **pkDocId**

```
Name
Category
DocType
Status
```

Die Sortierreihenfolge kann mit einem Parameter

```
OrderClause
```

überschrieben werden.

Dies führt in der Anwendung zu einem parameterreichen Methodenaufrüf, den wir wie folgt implementieren:

```
Fetch( _
Optional pkDocId as String="", _
Optional Name as String="", _
Optional Category as String="", _
Optional DocType as String="", _
Optional Status as String="5", _
Optional OrderClause as String=" ORDER BY T_Documents.Name"
) as XMLDoc
```

Die Methode wird bei einem Aufruf

```
Fetch("16")
```

das Statement

```
SELECT T_Documents.pkDocId, T_Documents.Name, T_Documents.Category,
T_Category.Description, T_Doctypes.Description, T_Documents.Path, T_
Documents.Comment, T_Documents.DocType, T_Documents.Status
FROM T_Category INNER JOIN (T_Doctypes INNER JOIN T_Documents ON T_
Doctypes.pkDocType = T_Documents.DocType) ON T_Category.pkCategory =
T_Documents.Category
WHERE T_Documents.pkDocId=16 AND T_Documents.Status=5
ORDER BY T_Documents.Name
```

erzeugen.

Nachdem das Basis SQL-Statement nun verfügbar ist, müssen als nächstes die Abfrageparameter eingefügt werden. Wir prüfen hierfür jeden Parameter und verknüpfen die Parameter zu einem Suchausdruck.

```
'Erstellen der WHERE Klausel
        If Not Name = "" Then pWhere = "T_Documents.Name =" &
Chr(34) & Name & Chr(34)
        If Not Category = "" Then
          If pWhere <> "" Then pWhere = pWhere & " AND "
            pWhere = pWhere & "T_Documents.Category =" & Category &
" "
        End If
        If Not DocType = "" Then
….
```

Nachdem alle Parameter so bearbeitet wurden, erzeugen wir den Abfrageausdruck:

```
        'Erstellung des Statements
        pSQL = pSQL & RTrim(pWhere & " " & Order)
```

Nun werden wir eine **DbResultSet**-Instanz erstellen und mit dem Recordset initialisieren. Dafür deklarieren wir eine Referenzvariable `mDbResultSet` auf Modulebene und fügen eine Friend-Eigenschaft `DbResultSet` ein.

```
Private mDbResultset As DbResultSet
Friend Property Set DbResultSet(NewDbResultset As DbResultSet)
    Set mDbResultset = NewDbResultset
End Property
```

Ein Dokument-Objekt wird über die DBLayer-Klasse durch den Aufruf von **CreateDocuments** erzeugt.

Eine Suche können wir nun mit wesentlich weniger Code-Aufwand durchführen, vor allem aber haben wir die SQL-Schicht aus der Anwendungsschnittstelle so verbannt:

```
Set myDocs = myDbLayer.CreateDocuments()
Set myDoc=myDocs.Fetch("3")
```

liefert jetzt das gleiche Ergebnis wie

```
Set myResultset = myDbLayer.CreateDbResultset()

Set myDoc=myResultset.FetchXMLDoc("Select * From T_Documents Where
pkDocId=3")
```

Wir werden diese Schnittstelle hier nicht weiter ausführen, sondern damit nur eine weitere Anregung dafür liefern, wie eine XML-generierende Schicht gekapselt werden kann.

7.5 ADO-XML

Wir haben eingangs erwähnt, daß unsere Komponente sowohl ein benutzerdefiniertes XML-Format als auch ADO-XML zurückliefern kann. Wir implementieren dies durch die Eigenschaft **ADOXML** der Klasse **xmlDoc**. Zu diesem Zweck speichern wir den Inhalt unseres ADO-Resultsets in eine Datei und liefern den Text der Datei zurück:

```
Public Property Get ADOXML() As String
    Dim pFSO As New Scripting.FileSystemObject
    Dim pStream As Scripting.TextStream
    Dim pXml As String
    Dim pRec As New ADODB.Recordset
    pFSO.DeleteFile "C:\adotemp.xml"
    mResultset.Save "C:\adotemp.xml", adPersistXML
    Set pStream = pFSO.OpenTextFile("C:\adotemp.xml")
    pXml = pStream.ReadAll
    Set pStream = Nothing
    Set pFSO = Nothing
    ADOXML = pXml
End Property
```

Das Vorgehen ist recht einfach, da man sich hierfür ausschließlich zur Verfügung stehender Basiskomponenten bedienen muß:

- ADO
- Microsoft Scripting Runtime

Interessanter als die Erzeugung ist der Vergleich des generierten XMLs mit dem unserer selbstdefinierten Variante. Wir sehen dabei, daß unser Format deutlich kompakter ist. Dies liegt darin begründet, daß ADO-XML eine große Menge beschreibender Daten zurückliefert. Beschrieben wird sowohl die Struktur des Abfrageergebnisses, als auch erweiterte Daten zu den einzelnen Datenfeldern, die aber für viele Verarbeitungszwecke nicht benötigt werden oder zumindest nicht bei jedem Datenzugriff geladen werden müssen. Vorteil der ADO-XML-Lösung ist die Möglichkeit, XML auch in das Resultset zu laden, sofern das XML das ADO-XML-Format hat.

ADO-XML ist in die Bereiche Schema und Daten gegliedert. Der Schema-Bereich liefert Meta-Daten über die Ergebnismenge im XML-Data-Format. Da lediglich Microsoft diese Spezifikation unterstützt, ist es von eher

zweifelhaftem Sinn, sich damit auseinanderzusetzen. Aber auch dies muß jeder selbst entscheiden.

Der Datenbereich liefert, in Reihen gegliedert, die Rückgabedaten.

Nach dem Vergleich mag jeder selbst entscheiden, ob er ADO-XML nutzen möchte oder nicht. Eine Entscheidungshilfe hierfür ist der Versuch, ein Stylesheet für die Anzeige der Daten zu erstellen.

ADO-XML Anzahl Zeichen: 3016	Selbstdefiniertes XML-Format Anzahl Zeichen: 1008
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882' xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882' xmlns:rs='urn:schemas-microsoft-com:rowset' xmlns:z='#RowsetSchema'> <s:Schema id='RowsetSchema'> <s:ElementType name='row' content='eltOnly'> <s:attribute type='pkDocId'/> <s:attribute type='Name'/> <s:attribute type='Path'/> <s:attribute type='Comment'/> <s:attribute type='ToBeDeployed'/> <s:attribute type='DocType'/> <s:attribute type='Category'/> <s:attribute type='Status'/> <s:attribute type='Used'/> <s:attribute type='Checked'/> <s:attribute type='CheckedBy'/> <s:extends type='rs:rowbase'/> </s:ElementType> <s:AttributeType name='pkDocId' rs:number='1'> <s:datatype dt:type='int' dt:maxLength='4' rs:precision='10' rs:fixedlength='true' rs:maybenull='false'/> </s:AttributeType> <s:AttributeType name='Name' rs:number='2' rs:nullable='true' rs:write='true'>	<?xml version='1.0' encoding='ISO-8859-1'?> <datasets> <dataset> <field name="pkDocId">1</field> <field name="Name"><![CDATA[aspExamples.asp]]></field> <field name="Path"><![CDATA[CD\Samples\Asp\Admin]]></field> <field name="Comment"><![CDATA[Shows Asp Examples]]></field> <field name="ToBeDeployed">True</field> <field name="DocType">5</field> <field name="Category">1</field> <field name="Status">2</field> <field name="Used">1</field> <field name="Checked">15.01.2000</field> <field name="CheckedBy"><![CDATA[Elmar]]></field> </dataset> <dataset> <field name="pkDocId">2</field> <field name="Name"><![CDATA[blank.asp]]></field> <field name="Path"><![CDATA[CD\Samples\Asp\Admin]]></field> <field name="Comment"><![CDATA[Liefert leere HTML Seite]]></field> <field name="ToBeDeployed">True</field> <field name="DocType">5</field> <field name="Category">1</field> <field name="Status">3</field>

Tabelle 7.3

ADO-XML Anzahl Zeichen: 3016	Selbstdefiniertes XML-Format Anzahl Zeichen: 1008
<s:datatype dt:type='string' dt:maxLength='50'/> </s:AttributeType> <s:AttributeType name='Path' rs:number='3' rs:nullable='true' rs:write='true'> <s:datatype dt:type='string' dt:maxLength='50'/> </s:AttributeType> <s:AttributeType name='Comment' rs:number='4' rs:nullable='true' rs:write='true'> <s:datatype dt:type='string' dt:maxLength='255'/> </s:AttributeType> <s:AttributeType name='ToBeDeployed' rs:number='5' rs:write='true'> <s:datatype dt:type='boolean' dt:maxLength='2' rs:fixedlength='true' rs:maybenull='false'/> </s:AttributeType> <s:AttributeType name='DocType' rs:number='6' rs:nullable='true' rs:write='true'> <s:datatype dt:type='int' dt:maxLength='4' rs:precision='10' rs:fixedlength='true'/> </s:AttributeType> <s:AttributeType name='Category' rs:number='7' rs:nullable='true' rs:write='true'> <s:datatype dt:type='int' dt:maxLength='4' rs:precision='10' rs:fixedlength='true'/> </s:AttributeType>	<field name="Used">1</field> <field name="Checked">15.01.2000</field> <field name="CheckedBy"><![CDATA[Elmar]]></field> </dataset> </datasets>

ADO-XML Anzahl Zeichen: 3016	Selbstdefiniertes XML-Format Anzahl Zeichen: 1008
<s:AttributeType name='Status' rs:number='8' rs:nullable='true' rs:write='true'> <s:datatype dt:type='int' dt:maxLength='4' rs:precision='10' rs:fixedlength='true'/> </s:AttributeType> <s:AttributeType name='Used' rs:number='9' rs:nullable='true' rs:write='true'> <s:datatype dt:type='int' dt:maxLength='4' rs:precision='10' rs:fixedlength='true'/> </s:AttributeType> <s:AttributeType name='Checked' rs:number='10' rs:nullable='true' rs:write='true'> <s:datatype dt:type='dateTime' dt:maxLength='16' rs:scale='0' rs:precision='19' rs:fixedlength='true'/> </s:AttributeType> <s:AttributeType name='CheckedBy' rs:number='11' rs:nullable='true' rs:write='true'> <s:datatype dt:type='string' dt:maxLength='50'/> </s:AttributeType> </s:Schema> <rs:data> <z:row pkDocId='1' Name='aspExamples.asp' Path='CD\Samples\Asp\Admin'	

Tabelle 7.3

ADO-XML Anzahl Zeichen: 3016	Selbstdefiniertes XML-Format Anzahl Zeichen: 1008
Comment='Shows Asp Examples' ToBeDeployed='True' DocType='5' Category='1' Status='2' Used='1' Checked='2000-01-15T00:00:00' CheckedBy='Elmar'/> <z:row pkDocId='2' Name='blank.asp' Path='CD\Samples\Asp\Admin' Comment='Liefert leere HTML Seite' ToBeDeployed='True' DocType='5' Category='1' Status='3' Used='1' Checked='2000-01- 15T00:00:00' CheckedBy='Elmar'/> </rs:data> </xml>	

Tabelle 7.3

7.5.1 Stylesheets für benutzdefiniertes Format und für ADO-XML

Wir werden nun für jedes Datenformat, benutzerdefiniert oder ADO-XML, ein XSL-Stylesheet für die Ausgabe der Dateien erstellen.

Der Kopfbereich des Stylesheets ist für beide Varianten gleich:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

Das Wurzelelement ist bei der Ausgabe einer HTML-Datei natürlich HTML, zuvor setzen wir das für Westeuropa sinnvolle Encoding auf den Zeichensatz Latin-1.

Es folgen die gestalterischen Anweisungen für die Tabelle und die Spaltenbeschriftungen.

```
<body STYLE="font-family:Arial, helvetica, sans-serif; font-
size:12pt; background-color:#FFFFFF">
<table border="1" style="table-layout:fixed width=600">
<col width="200"></col>
<tr bgcolor="#0000F8">
   <th><font color="#FFFFFF">Name</font></th>
   <th><font color="#FFFFFF">Path</font></th>
   <th><font color="#FFFFFF">Comment</font></th>
</tr>
```

Die Tabelle wird durch eine einfache **For-each**-Schleife erzeugt. Da unser benutzerdefiniertes Format Textfelder in `CDATA` verpackt, lesen wir hier mit einer Kombination aus Pfadausdruck und `cdata()`-Methode.

```
<xsl:for-each select="datasets/dataset">
<tr bgcolor="#FFFFFF">
   <td><font color="#0000F8"><xsl:value-of
select="field[@name='Name']/cdata()"/></font></td>
   <td><font color="#0000F8"><xsl:value-of
select="field[@name='Path']/cdata()"/></font></td>
   <td><font color="#0000F8"><xsl:value-of
select="field[@name='Comment']/cdata()"/></font></td>
</tr>
</xsl:for-each>
```

An dieser Stelle unterscheiden sich die Stylesheets, das für ADO-XML ist wie folgt:

```
<xsl:for-each select="xml/rs:data/z:row">
  <tr bgcolor="FFFFFF">
    <td><font color="#0000F8"><xsl:value-of select="@Name"/></
font></td>
    <td><font color="#0000F8"><xsl:value-of select="@Path"/></
font></td>
    <td><font color="#0000F8"><xsl:value-of select="@Comment"/></
font></td>
  </tr>
</xsl:for-each>
```

Die Unterscheidungen sind einerseits im Pfadausdruck, andererseits im Lesen der Werte. ADO-XML schreibt die Feldwerte in Attribute, unser Format dagegen in CDATA. Hier stellt sich die Frage, wie ADO-XML mit den unerlaubten Zeichen in den Feldwerten umgeht. Um es gleich zu verraten, es verhält sich erwartungsgemäß und encodiert alle gefährlichen Zeichen. Ist im Datenfeld z.B. ein Anführungszeichen enthalten, gibt ADO-XML die Referenz ‚"' zurück. Ein WebBrowser hat damit keine Probleme, da ihm die Entity-Referenz bekannt ist, er wird daher an dieser Stelle ein Anführungszeichen anzeigen. Wird ADO-XML für den Datenaustausch eingesetzt, muß allerdings darauf geachtet werden, daß encodierte Zeichen auch wieder decodiert werden müssen.

Um die Ergebnismenge kleiner zu halten, werden im Datenbereich einer ADO-XML-Datei nur Felder berücksichtigt, die auch Werte enthalten

Wir haben in diesem Kapitel gesehen, daß wir Stylesheets sowohl für ADO als auch für unser Format problemlos erstellen können. Es bleibt jedoch noch eine Eigenheit des ADO-XML zu berichten, die in der Entwicklung besonders lästig ist: Um die Ergebnismenge kleiner zu halten, werden im Datenbereich einer ADO-XML-Datei nur Felder berücksichtigt, die auch Werte enthalten. Leere Felder und solche, die mit dem Wert NULL gefüllt sind, werden von ADO-XML ignoriert. Da es jedoch bisher keinen praktikablen Weg gibt, um festzustellen, welche Attribute ein Element enthält, erschwert dieser Umstand z.B. die Abbildung dynamischer Tabellen. In unserem Format können wir Tabellen nach dem folgenden einfachen Muster erstellen:

- Für jeden Datensatz erzeuge eine Zeile.
- Für jedes Feld erzeuge eine Spalte.

In XSL sieht das so aus:

```
<xsl:for-each select="datasets/dataset">
<tr bgcolor="#FFFFFF">
   <xsl:for-each select="field">
   <td><font color="#0000F8"><xsl:value-of select="text()"/></
font></td>
   </xsl:for-each>
</tr>
</xsl:for-each>
```

Hier müssen wir lediglich noch die Spaltennamen ermitteln und diese in die Ergebnismenge mit einschließen. Dadurch haben wir unabhängig von den Daten ein allgemeines Stylesheet zur Anzeige von Tabellen. Wir erweitern daher das Stylesheet:

```
<xsl:for-each select="datasets/dataset[0]">
   <tr bgcolor="#FFFFFF">
   <xsl:for-each select="field">
      <th>
<font color="#0000F8">
<xsl:value-of select="@name"/>
</font>
</th>
   </xsl:for-each>
   </tr>
</xsl:for-each>
```

Wir haben hier den Ausdruck für das Lesen der Datenfelder nur leicht abgewandelt. Durch das Setzen des Index im ‚select'-Ausdruck sorgen wir zunächst dafür, daß die Schleife zur Erzeugung der Titelzeile nur einmal durchlaufen wird:

```
<xsl:for-each select="datasets/dataset[0]">
```

Die Feldnamen lesen wir aus dem ‚name'-Attribut unserer XML-Datei:

```
<xsl:value-of select="@name"/>
```

Damit haben wir eine vollständiege Tabelle generiert, deren grafische Ausprägung einfach angepaßt werden kann.

Das Kapitel fehlt in dieser PDF-Datei.

Sorry, aber wir wollen ja auch das eine oder andere Buch verkaufen :-)

Index

X