

XML

Datenhaltung und -verarbeitung

Teil I

Das SAX-API zum Verarbeiten von XML-Dokumenten

Teil II

Design-Varianten für XML-Daten in einer relationalen Datenbank

Teil III

Document Object Model DOM

Teil IV

Das Transformation API TrAX

Teil V

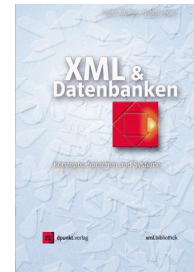
XML-Datenbanken am Beispiel Persistent DOM (PDOM)

Teil VI

Abfragesprachen für XML am Beispiel XQL

Literatur

- [1] "XML & Datenbanken";
Meike Klettke, Holger Meyer;
dpunkt.verlag, Dezember 2002;
ISBN 3-89864-148-1.
- [2] "XQuery"; Wolfgang Lehner,
Harald Schöning;
dpunkt.verlag, 2004;
ISBN 3-89864-266-6.
- [3] "XML in a Nutshell";
Elliotte Rusty Harold, W. Scott Means;
O'Reilly, June 2002;
ISBN 0-596-00292-0.

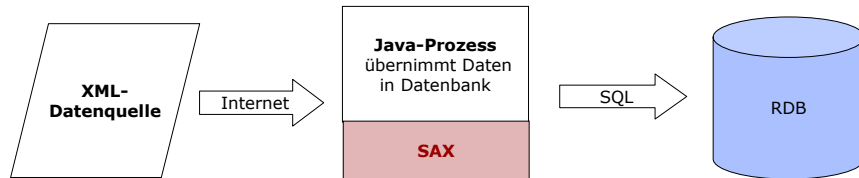


Teil I

SAX

Simple Application Interface for XML

Anwendungsbeispiel SAX



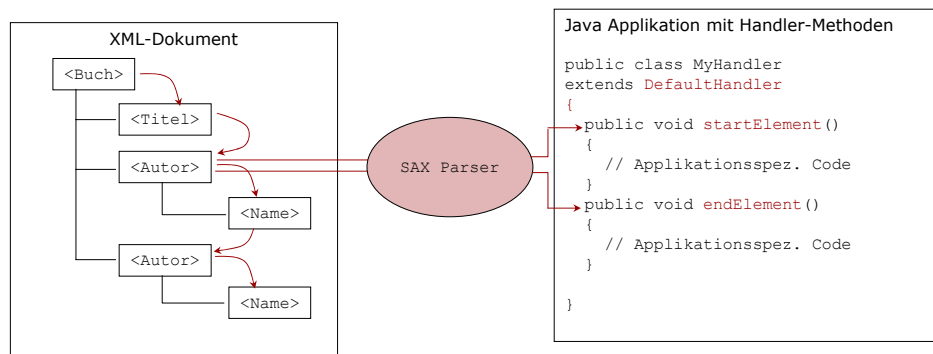
SAX ist ein zentrales Hilfsmittel für das Einlesen XML-Daten aus einem Datenstrom, das Validieren und Weiterleiten an andere Applikationen, Applikationsteile oder eine Datenbank. Obiges Beispiel zeigt ein automatisiertes Übertragen von XML-Daten in eine Datenbank. In der B2B-Kommunikation sind solche Transfers häufig.

- Mit Hilfe von SAX können die XML-Daten in einem Datenstrom elementweise entgegengenommen werden.
- JDBC ermöglicht das Übertragen der Daten mit SQL-Befehlen in die Datenbank

Ein SAX-Parser wird immer auch als erste Stufe beim Erstellen von DOM-Bäumen verwendet (siehe dort).

Verarbeitung von XML mit SAX

- SAX = Simple API for XML
- Der SAX-Parser traversiert ein XML Dokument und ruft für jedes Strukturelement einen benutzer-implementierte Handler-Methode auf



Das SAX API

Das SAX API ist einfach zu benützen und eignet sich sehr gut für das prozessorientierte Verarbeiten von XML-Daten. Es ist auf die Verwendung mit Java als Programmiersprache zugeschnitten, es gibt jedoch viele analoge API's für C++, Perl, PHP, Python usw.

Jedes XML-Dokument stellt eine Baumstruktur dar. SAX realisiert im wesentlichen das Traversieren des XML-Baumes. Genau gesprochen ist es eine *in-order Traversierung*: Sobald der Traversierungsmechanismus in einen Knoten (Element) eintritt, wird die Callback-Funktion `startElement()` aufgerufen. Anschließend wird in weiteren Aufrufen von Callback-Funktionen der Inhalt des Knotens an die Applikation übergeben. Danach wird der erste, weiter unten im Baum liegende Kind-Knoten bearbeitet. Sind alle Kind-Knoten bearbeitet, wird beim Elternknoten die Austrittsfunktion `endElement()` aufgerufen.

Weitere Callback-Funktionen werden aktiviert beim Starten und Verlassen eines Dokumentes, beim Wechseln des Namensraumes, beim Auftreten von Fehlern, für die Verarbeitung von XML 'Processing Instructions' usw.

Der SAX-Parser kann das verarbeitete XML-Dokument validieren. Letzteres muss dafür eine DTD-Deklaration beinhalten, und beim SAX-Parser muss die `setValidating()` Funktion aufgerufen werden.

Das SAX-API eignet sich gut für den Transfer von XML-Daten in eine relationale Datenbank. In den Funktionen `startDocument()` und `endDocument()` kann die Transaktionskontrolle stattfinden. Die `startElement()`-, `characters()`- und `endElement()`-Funktion können SQL-Befehle enthalten.

Im Moment aktuell ist die Version 2 des SAX-API. Die entsprechenden Java Packages sind bei Sun unter JAXP 1.1 zu finden und ab Java 1.4 komplett integriert. Die Validierung von XML-Dateien mit XMLSchema funktioniert mit dem neuesten API JAXP 1.2 (zusätzlich herunterzuladen bei Sun).

SAX ist nicht ein W3 Standard und wird auch nicht vom W3 Konsortium gepflegt.

SAX Parser, Beispiel

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

public class MyHandler extends DefaultHandler
{
    private String elementContent = "";
    public static void main(String argv[])
    {
        try {
            SAXParserFactory factory;
            factory = SAXParserFactory.newInstance();
            factory.setValidating( true );
            SAXParser saxParser = factory.newSAXParser();
            DefaultHandler handler = new MyHandler();
            saxParser.parse( new File(argv[0]), handler );
        } catch (SAXParseException e ) {
            System.out.println( "Dokument ist ungültig."+e.toString());
        } catch ( Throwable t ) {
            t.printStackTrace();
        }
    }
}
```

Obiges Beispiel zeigt einen validierenden SAX Parsers. Die Validierung kann über die Methode `setValidating( true )` oder über die allgemeine Methode `setFeature()` eingeschaltet werden.

`argv[0]` enthält die einzulesende XML-Datei

Wenn für die Validierung ein XMLSchema benutzt werden soll, sind folgende Angaben notwendig:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setValidating(true);
SAXParser saxParser = factory .newSAXParser();
saxParser.setProperty(
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema" );
saxParser.setProperty(
    "http://java.sun.com/xml/jaxp/properties/schemaSource",
    urlOderPfadAufMeinSchemaFile );
saxParser.parse( myfile, myhandler )
```

SAX Parser, Beispiel ff.

```
public void startDocument() { System.out.println( "Verarbeitung gestartet." ); }

public void endDocument() { System.out.println( "Verarbeitung abgeschlossen." ); }

public void startElement(String namespaceURI,String lName,
                        String qName, Attributes attrs) {
    System.out.println( "Element:" + qName );
    for ( int i = 0; i < attrs.getLength(); i++ ) {
        String attName = attrs.getQName(i);
        String attValue = attrs.getValue(attName);
        System.out.println( "Attribut:" + attName + "=" + attValue );
    }
}

public void endElement(String namespaceURI,
                      String lName, String qName ) {
    System.out.println( "Content: " + elementContent );
    System.out.println( "(AnzahlZeichen:" + elementContent.length() + ")");
    elementContent = "";
}

public void characters( char[] buf, int start, int length ) {
    elementContent += new String( buf, start, length );
}

public void error(SAXParseException e) { throw e; }
}
```

Das Beispiel erstellt folgende Ausgabe:

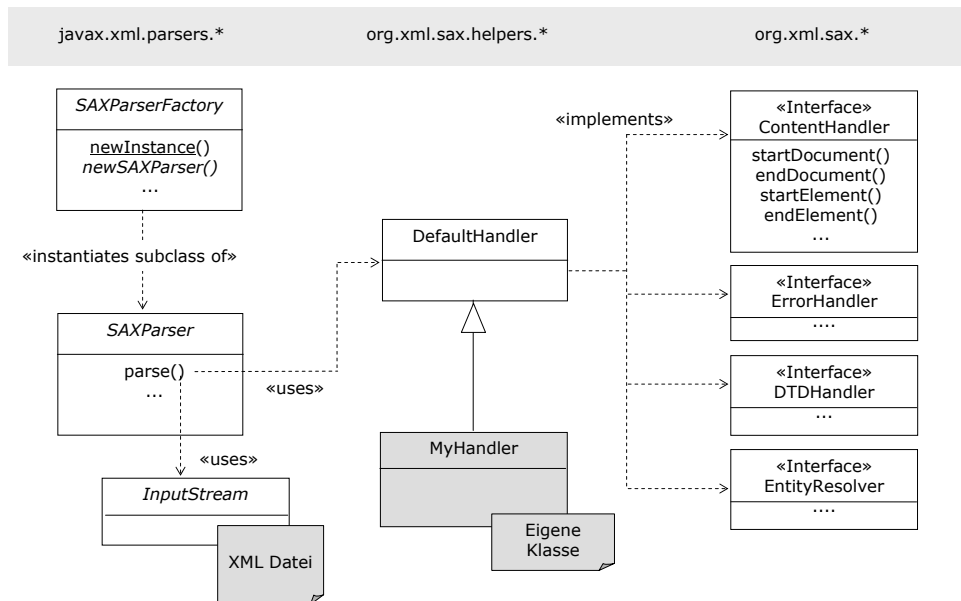
- Startmeldung für das Bearbeiten eines Dokumentes
- von jedem Element den Namen
- von jedem Attribut eines Elementes Namen und Wert
- von jedem Element den Inhalt
- Abschlussmeldung für das Bearbeiten eines Dokumentes
- Fehlermeldung bei Validierungsfehler

Die `characters()`-Methode kann vom Parser *mehrmals* aufgerufen werden innerhalb eines Elementes. Es ist deshalb empfehlenswert, dass die Verarbeitung des Element-Inhaltes in der Methode `endElement()` durchgeführt wird.

Der Parameter `buf` der `characters()`-Methode enthält je nach Implementation des SAXParsers alle bisher geparsten XML-Daten. Mit `start` und `length` kann deshalb der effektiv gemeinte Teil im aktuellen Aufruf bestimmt werden.

Beachte, dass die `error()` -Methode implementiert werden *muss*, um einen Validierungsfehler zu bemerken.

SAX Package Übersicht



SAXParserFactory

Ein Objekt von SAXParserFactory ist zuständig für das Erzeugen von SAXParser Objekten. Mit der statischen Methode newInstance() kann eine Factory erzeugt werden. newInstance() ermittelt die zu erzeugende, konkrete Klasse typischerweise über folgende Mechanismen:

- Über eine Plattformweite defaulteinstellung in der Methode newInstance() selbst.
- Über eine Umgebungsvariable namens javax.xml.parsers.SAXParserFactory, welche den vollständigen Klassenpfad enthält.
- Über das Property File "lib/jaxp.properties" im JRE Verzeichnis.

Mit der SAXParserFactory kann ein SaxParser instantiiert werden.

SAXParser

Das SaxParser-Objekt enthält im wesentlichen einen Satz von parse() Methoden, mit welchen ein XML-Dokument verarbeitet werden kann. Den parse() Methoden wird eine XML-Quelle und ein Handler-Objekt mitgegeben.

DefaultHandler

Die Klasse DefaultHandler stellt für alle Verarbeitungsfunktionen, welche vom SAXParser aufgerufen werden, Default-Implementationen zur Verfügung. Der DefaultHandler implementiert hierzu die vier Interfaces ContentHandler, ErrorHandler, DTDHandler und EntityResolver. Der Applikationsentwickler leitet in der Regel eine eigene Handler-Klasse ab, z.B. MyHandler, und überschreibt darin die ihn interessierenden Methoden.

ContentHandler Funktionen

```
setDocumentLocator()  
startDocument()  
endDocument()  
startPrefixMapping()  
endPrefixMapping()  
startElement()  
endElement()  
characters()  
ignorableWhitespace()  
processingInstruction()
```

setDocumentLocator() übergibt ein Objekt der Klasse Locator. Dieses gibt im Wesentlichen Auskunft über den URL des Dokumentes.

startDocument() wird nur einmal und unmittelbar nach setDocumentLocator() aufgerufen. Diese Methode ist deshalb für das Initialisieren einer Transaktion oder Datenbankverbindung geeignet.

endDocument() wird als allerletzte Methode aufgerufen, nachdem das ganze Dokument durchlaufen wurde. Die Methode ist geeignet für das Bestätigen einer Datenbank-Transaktion oder das Schliessen einer Verbindung.

startPrefixMapping() zeigt die Einführung eines Namensraumes an. Zur Verfügung steht der Name des Prefix und der zugehörige URL. Der Prefix Name kann leer sein, dann wird ein Default Namespace eingeführt.

endPrefixMapping() zeigt das Ende der Gültigkeit eines Namespaces.

startElement() zeigt den Beginn eines XML-Elementes an. Zur Verfügung steht Information über den gültigen Namespace-URL, den vollen Namen des Elementes (Name inkl. Prefix) und den lokalen Namen (Name ohne Prefix). In diese Methode wird auch die Information über Attribute des Elementes geliefert, d.h. über vollen Namen, lokalen Namen, URL des Namespace, Typ und Wert des Attributes. Der Typ eines Attributes ist CDATA, ID, IDREF, IDREFS, NMTOKEN, NMTOKENS, ENTITY, ENTITIES, oder NOTATION. Die Verarbeitung der Attribute wird erleichtert durch das Interface `Attributes`, das Methoden für den Zugriff auf alle Informationen der Attribute enthält.

endElement() zeigt das Ende eines Elementes an, d.h. die Verarbeitung von `</element>` im XML-Dokument.

characters() gibt jeweils den Inhalt eines Elementes aus, d.h. Daten vom Typ `#PCDATA`. Der SAXParser ist frei, diesen Inhalt in mehrere aufeinanderfolgende Aufrufe von `characters()` aufzuteilen. Um sicher zu sein, dass der ganze Inhalt eines Elementes verarbeitet wurde, muss also der Aufruf von `endElement()` abgewartet werden. Die von `characters()` gelieferten Teile sind bis dahin von der Applikation aufzubewahren! Ein Zeilenumbruch innerhalb des Textes (was in XML absolut legal ist) kann beispielsweise zu zwei Aufrufen von `characters()` führen.

ignorableWhitespace() wird aufgerufen, wenn zwischen Element-Tags nicht-druckbare Zeichen (Blank, Linefeed, Tab etc.) stehen. Diese werden dann in einem oder mehreren Aufrufen von `ignorableWhitespace()` gemeldet. Validierende Parser *müssen* Whitespace melden.

processingInstruction() wird aufgerufen, wenn ein XML Element der Form `<?target data ?>` in der XML-Datei auftritt. Target ist dabei der Name des Prozessors der zuständig für die Verarbeitung ist. Bis zum Auftreten von `?>` gehört alle Zeichen zu 'data' und werden vom SAX Parser nicht interpretiert.

Error-, DTD-, und Entity-Handler

ErrorHandler Interface

```
fatalError()  
error()  
warning()
```

DTDHandler Interface

```
notationDecl()  
unparsedEntityDecl()
```

EntityResolver Interface

```
resolveEntity()
```

fatalError() wird aufgerufen, wenn der Parser nicht mehr weiterfahren kann. Dies ist beispielsweise der Fall wenn ein Dokument nicht 'well formed' ist.

error() tritt auf, wenn ein Fehler auftritt, aber der Parser trotzdem weiterfahren kann. Beispielsweise wird `error()` aufgerufen, wenn ein Validierungsfehler auftritt. Es ist der Applikation überlassen, das Parsing abubrechen oder weiterzufahren.

warning() wird aufgerufen, wenn eine ungewöhnliche Situation in einem DTD auftritt, zum Beispiel eine doppelte Element-Deklaration.

notationDecl() wird zum Beispiel in Zusammenhang mit folgendem DTD aufgerufen:

```
<!ELEMENT image (#PCDATA) >
<!NOTATION gif SYSTEM "image/gif">
<!ATTLIST image type NOTATION ( gif | jpg ) >
```

Damit wird der Applikation ermöglicht, den Attributwert gif in eine MimeType- Angabe umzuwandeln, indem mit dem Aufruf von notationDecl() die Übersetzung von " gif" zu "image/gif" bekanntgegeben wird.

unparsedEntityDecl() wird zum Beispiel in Zusammenhang mit folgendem DTD aufgerufen:

```
<!NOTATION html PUBLIC "text/html">
<!ENTITY htabe SYSTEM "http://www.hta-be.bfh.ch/"  NDATA html >
<!ELEMENT company EMPTY>
<!ATTLIST company description ENTITY #REQUIRED>
```

Die Funktion unparsedEntityDecl() liefert das Mapping von 'htabe' nach "http://www.hta-be.bfh.ch/". Damit wird der Applikation ermöglicht, beispielsweise im XML Element `<company description="htabe" />` das description Attribut in einen URL umzuwandeln.

resolveEntity() wird aufgerufen, bevor ein URL geöffnet wird. Die Applikation hat damit die Möglichkeit den tatsächlich verlangten URL vom Parser bearbeiten zu lassen, oder allenfalls den URL durch einen anderen URL oder eine andere Quelle zu ersetzen.

Mit den optionalen Interfaces `LexicalHandler` und `DeclHandler` können auch CDATA - Abschnitte, Kommentare und DOCTYPE-Deklarationen in einer XML-Datei erfasst werden, während der Verarbeitung mit SAX:

```
import org.xml.sax.ext.*;

public class myHandler extends DefaultHandler() implements LexicalHandler
{
    public void comment(char[] ch, int start, int length) {...}
    // Beginn von CDATA, Daten via characters()
    public void startCDATA() {...}
    // Ende von CDATA
    public void endCDATA() {...}
}

...

XMLReader r = parser.getXMLReader();
r.setProperty("http://xml.org/sax/properties/lexical-handler", myHandler );
```

Verarbeitung binärer Daten

Wie können binäre Daten in XML behandelt werden?

1. Die binären Daten werden vorerst in eine Darstellung gebracht, welche keine XML-Syntaxzeichen enthält. Hierzu eignet sich beispielsweise eine Base64-Codierung.
2. Die codierten Binärdaten werden in ein XML-Element geschrieben, beispielsweise
`<image>codierte Binärdaten</image>`
3. Die Daten können beim Parsen des XML-Dokumentes wieder in ihre ursprüngliche Form decodiert werden

Die Decodierung kann typischerweise der SAX-Parser übernehmen.

Verarbeitung binärer Daten

3 Bytes (24 Bits) werden in 4 Stücke à 6 Bits zerlegt. Jedes dieser 6-Bit Stücke wird als ASCII-Zeichen (8 Bit) dargestellt. Es wird folgendes Mapping der 6-Bit Stücke in ASCII-Zeichen verwendet:

6-Bit-Stück	ASCII-Zeichen	8-Bit Wert
000000...011001	A...Z	01000001...01011010
011010...110011	a...z	01100001...01111010
110100...111101	0...9	00110000...00111001
111110	+	00101011
111111	/	00101111

Das ASCII Zeichen '=' kann als Füllzeichen verwendet werden

Die Codierung der 6 Bit Stücke ist in RFC 2045 geregelt.

Das Füllzeichen kann verwendet werden, um beispielsweise gleichlange Zeilen zu übertragen.

Alle Nicht- Codierungszeichen werden bei der Decodierung eines Base64-Zeichenstromes übergangen.

Base64 Bibliotheken für Java und andere Sprachen sind vielfältig verfügbar, beispielsweise unter

<http://prdownloads.sourceforge.net/iharder/base64-1.3.5.zip>

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 24 input bits are available in an input group, zero bits are added (on the right) to form an integral number of 6-bit groups. Padding at the end of the data is performed using the "=" character. Since all base64 input is an integral number of octets, only the following cases can arise: (1) the final quantum of encoding input is an integral multiple of 24 bits; here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding, (2) the final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by two "=" padding characters, or (3) the final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be three characters followed by one "=" padding character. Because it is used only for padding at the end of the data, the occurrence of any "=" characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three and no "=" characters are present.

Aufgaben SAXParser

1. Erstellen Sie einen Handler, der `catalog-classic.xml` einliest und daraus eine HTML-Liste mit allen Buchtiteln erstellt.
2. Erstellen Sie einen Handler, der `catalog-generic.xml` einliest und daraus eine HTML-Liste mit allen Buchtiteln erstellt.

Eine Vorlage für den Handler finden Sie unter

`...\xml-db-workshop\SAX\Handler\MyHandler.java`

Zum Kompilieren und Starten von Java-Programmen sollten Sie die DOS-Shell `shell.bat` verwenden. Allenfalls sind die Klassenpfade für ihre Umgebung noch anzupassen.

Die Lösungen finden Sie anschliessend unter

`...\xml-db-workshop\SAX\Loesungen-HTML`

HTML-Dokumentationen

Die Dokumentation ist in der Java Api Doc 1.4 in den Packages `javax.xml.*` und `org.sax.*` zu finden.

Gute Parser sind auch bei Apache unter dem Namen Xerces und bei IBM unter dem Namen xml4j zu finden.

Teil II

Design-Varianten für XML-Daten in einer relationalen Datenbank

Übersicht

- Wird eine relationale Datenbank als "physisches" Speichermedium für XML-Daten verwendet, gibt es zwei grundsätzliche Design-Varianten:
 1. klassisch, nach Begriffen der Applikation
 2. generisch, nach Sprachelementen von XML
 3. gemischt, XML-Fragmente und gewöhnliche Attribute

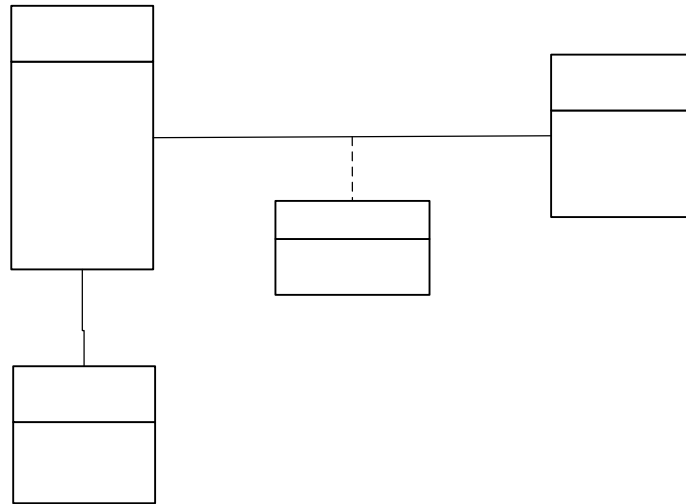
Relationale Datenbank mit klassischem Design:

einfach, hoch optimiert, gut mit Legacy Systems vereinbar, Standard-Schnittstellen wie JDBC und ODBC.

Relationale Datenbank mit generischem XML-Design:

stabile Technologie, einfache oder keine Pflege bei Änderungen an der XML-Datenstruktur, Abfragen sind komplex, Standard-Schnittstellen wie JDBC und ODBC.

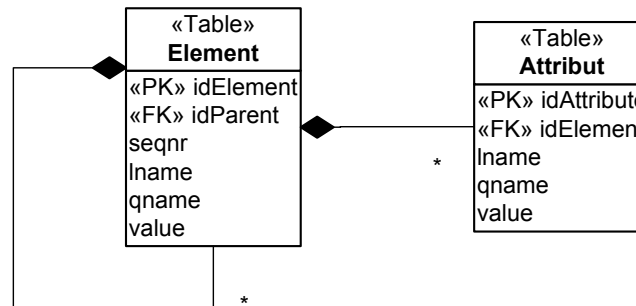
Relationale DB, Variante "Classic"



Obige Abbildung des XML-Bücherkataloges liegt auf der Hand und ist "typisch" relational. Der Vorteil dieser Variante liegt darin, dass die Abfragesprache SQL optimal geeignet ist, um Bücher oder andere Information abzufragen und herauszufiltern. Die Performance wird auch bei grössten Datenbeständen noch sehr gut sein. Der Nachteil der Struktur liegt darin, dass bei stark verschachtelten und rekursiven XML-Strukturen rasch ein unübersichtlicher Wald von Tabellen entsteht, aufgrund der Normalisierungs-Bedingungen.

Besonders im Zusammenhang mit *bestehenden* Datenbeständen dürfte die Variante "Classic" meistens vorgegeben sein, um XML-Daten aufzunehmen oder solche zu erzeugen.

Relationale DB, Variante "Generic"



Obige, generische Struktur ermöglicht das Abspeichern *beliebiger* XML-Dokumente in einer relationalen Datenbank, ohne dass neue Tabellen definiert, oder bestehende in ihrer Struktur geändert werden müssen.

Die Tabelle Element beinhaltet alle XML-Elemente, unabhängig von ihrem Namen. 'idElement' ist ein datenbank-interner Primärschlüssel. Ein Element hat meistens ein Elternelement. Das Feld 'idParent' zeigt auf dieses. Die 'seqNr' rettet die Reihenfolge, in der Elemente in einem XML-Dokument vorkommen. 'qName' bezeichnet den qualifizierten Elementnamen. Der eigentliche Dateninhalt (PCDATA) von Elementen ist in der Spalte 'value' abgelegt. Für Elemente, die lediglich andere Elemente enthalten ist 'value' leer. Die Attribut-Tabelle enthält die Namen und den Wert von Attributen.

Obige Struktur ist nur eine vereinfachte Darstellung. Folgende Aspekte sind nicht berücksichtigt:

- Namespaces, Notationen und Entity Referenzen
- mixed content Datenmodell
- DTD Information, Elementtypen, Attributtypen
- Index- und Cluster-Strukturen für die Performance-Optimierung
- Integritätsbedingungen, z.B. kaskadiertes Löschen von abhängigen Elementen

Die generische Struktur ist an sich sehr pflegeleicht, weil sie unabhängig von einem bestimmten Content-Modell ist. Aufwendig ist allerdings das gezielte Abfragen und Modifizieren von Daten. SQL alleine bietet eine etwas spartanische Unterstützung. Es sollten Programmbibliotheken erstellt werden, welche das Traversieren von Elementbäumen in der relationalen Datenstruktur ermöglichen. Stored Procedures können dabei für die Lösung von Performance-Problemen nützlich sein.

Die generische Struktur ist eine mögliche Ausgangslage für die Hersteller von XML-Datenbanken, welche über das DOM-API und XPath den selektiven Zugriff auf Daten erlauben. Ausgehend von einem gegebenen Knoten kann beispielsweise sehr einfach die Menge aller direkten Kindknoten, der erste Kindknoten, der letzte Kindknoten etc. aus der Datenbank entnommen werden.

Stored Procedures haben den Vorteil, dass alle Queries auf dem Datenbank-Server ausgeführt werden können und damit recht performant sind. Gewisse proprietäre Erweiterungen von SQL, zum Beispiel bei Oracle, unterstützen rekursive Abfragen auf Tabellen und damit die XML-Ausgabe. Beispiel:

```
select *
from Element
start with idElem = 3
connect by idParent = prior idElem
order by seqnr
```

Bei grösseren Datenbeständen könnte z.B. eine Indextabelle (Hilfstabelle) der folgenden Form erstellt werden

```
ElementIndex( idElement, qNameElement, idChild, qNameChild, valueChild )
```

In diesem Index werden für ein bestimmtes Element alle Unterelemente mit ihrem Wert festgehalten. Enthält beispielsweise ein Element vom Typ 'book' zwei Unterelemente (auf beliebiger Stufe), eines vom Typ 'title' und eines vom Typ 'isbn', so enthält der Index zwei Einträge bezüglich dem 'book'-Element. Möchte man nun z.B. alle 'book'-Elemente suchen, die ein Unterelement mit Namen 'title' und dem Wert 'XML für Anfänger' haben, kann dies mit einer einzigen Abfrage auf der Index-Tabelle geschehen. Die Anzahl Elemente ist damit erheblich eingeschränkt. Die definitive XML-Ausgabe kann durch Traversieren der gefundenen Elemente erstellt werden.

Als weitere Verbesserung wäre auch das Erstellen von Clustertabellen denkbar. Eine solche hätte die Form

```
Cluster ( idElement, qNameElement, subTree )
```

Für bestimmte wichtige und häufig gebrauchte Elemente, zum Beispiel 'book', könnte darin in *einem* Feld der ganze restliche Element-Baum abgelegt sein. Der Zugriff auf ein bestimmtes Element mit all seinen Kind-Elementen benötigt dann nur eine einzige SQL-Abfrage.

Cluster- und Indextabellen können für bestimmte Abfragen auch kombiniert werden. Abfragen sind dann auch bei grossen Datenmengen sehr effizient. Beispiel:

```
select distinct subTree
from Index i, Cluster c
where i.idElement = c.idElement
and i.qNameChild = 'title'
and i.valueChild = 'XML für Anfänger'
```

Das Nachführen von Indices und Clustern kann mit Hilfe von Trigger-Mechanismen bewerkstelligt werden. Die eigentlichen Arbeitstabellen für das Ändern von Daten sollten 'Element' und 'Attribute' sein.

Demo Generic DB Modell

- Übertragen von XML-Daten mit einem SAX-Parser in die generische Datenbank-Struktur: `GenericDBHandler.java`
→ Arbeiten mit einem Stack
- Funktioniert dieser Handler auch für Mixed Content?
- Was passiert beim Einlesen von Mixed Content?
- Kann die originale XML-Datei aus dem Datenbank-Inhalt rekonstruiert werden?

Mixed Content ist beispielsweise folgendes XML-Fragment:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE catalog SYSTEM "mixed-content.dtd">
<catalog>
  <book>
    <description>
      Dieses Buch handelt von <keyword> XML </keyword> und
      <keyword> Java </keyword> in sehr fundierter Weise
    </description>
  </book>
</catalog>
```

Das Charakteristische an Mixed Content ist, dass innerhalb eines XML-Elementes sowohl gewöhnlicher Text *und* weitere XML-Elemente vorkommen dürfen. Bei gewöhnlichem Content können unterhalb eines XML-Elementes nur Text *oder* weitere XML-Elemente vorkommen. Der DTD für obige Mixed Content Daten ist wie folgt:

```
<!ELEMENT catalog (book*)>
<!ELEMENT book (description)>
<!ELEMENT description (#PCDATA|keyword)* >
<!ELEMENT keyword (#PCDATA)>
```

Das wichtigste Beispiel von mixed-content ist HTML !

Arbeiten mit einem Stack

<u>Verarbeitetes Element</u>	<u>idElement</u>	<u>idParent</u>	<u>Stack</u>
			null
<catalog>	1	null	1 null
<book>	2	1	2 1 null
<title>	3	2	3 2 1 null
</title>			2 1 null
</book>			1 null
<book>	4	1	4 1 null
<title>	5	4	5 4 1 null
</title>			4 1 null
</book>			1 null
</catalog>			null

Es ist Stackzustand jeweils *nach* der Verarbeitung des Start- oder Ende-Tags gezeigt.

Beim Aufruf von `startElement()` wird jedem Element eine ID zugeordnet. Ausserdem wird dem Element der aktuelle Wert auf dem Stack als Parent ID zugeordnet, bevor seine eigene ID auf den Stack geschoben wird. Beim Aufruf von `endElement()` wird das oberste Element vom Stack entfernt.

Aufgaben Generic DB Modell

1. Wie könnte man das Datenmodell anpassen, um Mixed Content aufzunehmen?
2. Wie müsste `GenericDBHandler.java` angepasst werden um mixed content aufzunehmen?
3. Nehmen Sie die entsprechenden Anpassungen vor.
4. Beschreiben Sie das Vorgehen (Java + SQL-Befehle) um eine Abfrage der folgenden Art mit dem generischen Datenmodell durchzuführen:
Zeige alle Bücher mit dem Titel 'Learning XML'. Gehen Sie davon aus, dass das XML-Element `<title>` nicht zwingend ein unmittelbares Kinde-Element von `<book>` ist, sondern weiter unten im Baum liegen kann.

Relationale DB, Variante "Blended"

- XML-Fragmente, welche in sich strukturiert sind, aber häufig als Ganzes von und zur Datenbank transferiert werden, können in ein einziges Tabellenfeld abgelegt werden.
- Normale SQL-Attribute für Primärschlüssel, Fremdschlüssel und für das Suchen relevante Information.
- Vorteil: Kompakte Tabellenstruktur, keine Änderung der Tabelle bei Änderung an den XML-Inhalten.

«Table» Catalog
«PK» idBook
book : XML

Catalog	idBook	book
	1	<book><title>...</title>...</book>
	2	<book><title>...</title>...</book>

für die Variante Blended ist eine Unterstützung der Datenbank mit XML-Konstrukten äusserst nützlich. Beispielsweise, wenn nur Bücher mit einer bestimmten Bedingung herausgesucht werden sollen, oder wenn nur bestimmte Teile der Buchinformation wie Titel + ISBN ausgegeben werden sollen.

Aufgabe

- Welche Auswirkungen hat die Variante "Blended" auf die Gestaltung eines SAX-Parsers bezüglich
 - Einfachheit in der Programmierung?
 - Generizität?

Teil III

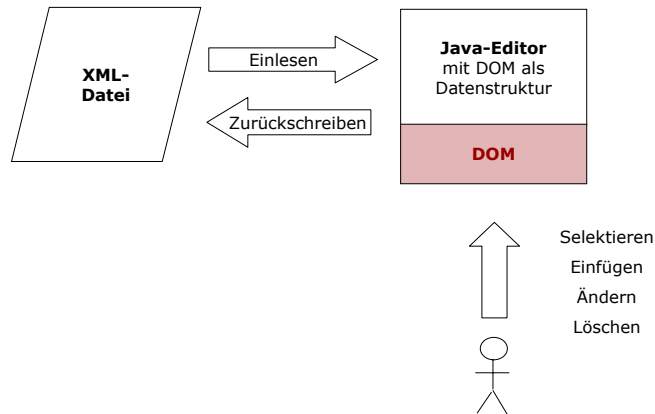
DOM

Document Object Model

Informationsquellen

- <http://www.w3.org/DOM/>
- <http://java.sun.com/xml/>

Anwendungsbeispiel DOM

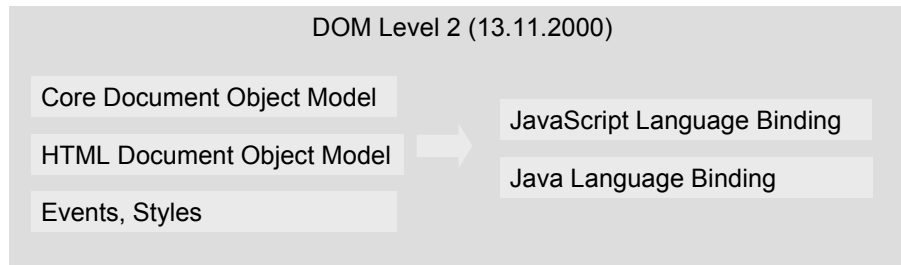


DOM ist eine Struktur, die häufig für das Modifizieren von XML-Daten oder für das Interaktive Arbeiten mit XML-Daten benutzt wird.

- Mit einem DOM-Parser wird aus einer XML-Datei ein DOM-Baum aufgebaut. Dieser enthält die XML-Daten (Nodes) als Objekte bestimmter, vordefinierter Klassen. Die hierarchische Anordnung (Eltern-Kind Beziehungen) in der XML-Datei und im DOM-Baum sind äquivalent.
- Der DOM-Baum kann über einen Editor bearbeitet werden.
- Nach der Bearbeitung des DOM-Baumes wird dieser auf eine XML-Datei zurückgeschrieben.
- Vor dem Einlesen/Zurückschreiben kann eine Transformation des DOM-Baumes in einen anderen DOM-Baum durchgeführt werden. Typischerweise ist das beim Arbeiten mit einer relationalen Datenbank der Fall. Für das Editieren ist eine eher hierarchische Anordnung der XML-Daten wünschenswert, hingegen ist für das Übertragen von und in die Datenbank eine flache, datensatz-orientierte Anordnung notwendig.

DOM Architektur

- Darstellung von XML-Dokumenten als Objektbaum
- Standard Interfaces für Zugriff auf Attribute
- Standard Methoden für Traversierung und Modifikation



Während SAX eine Verarbeitung-Packet ist, dient DOM sowohl der Bearbeitung von XML-Daten, ist aber gleichzeitig auch ein generisches Datenmodell. SAX erlaubt ein Stream-Verarbeitung, DOM baut ein ganzes Dokument als speicherresidente Struktur auf.

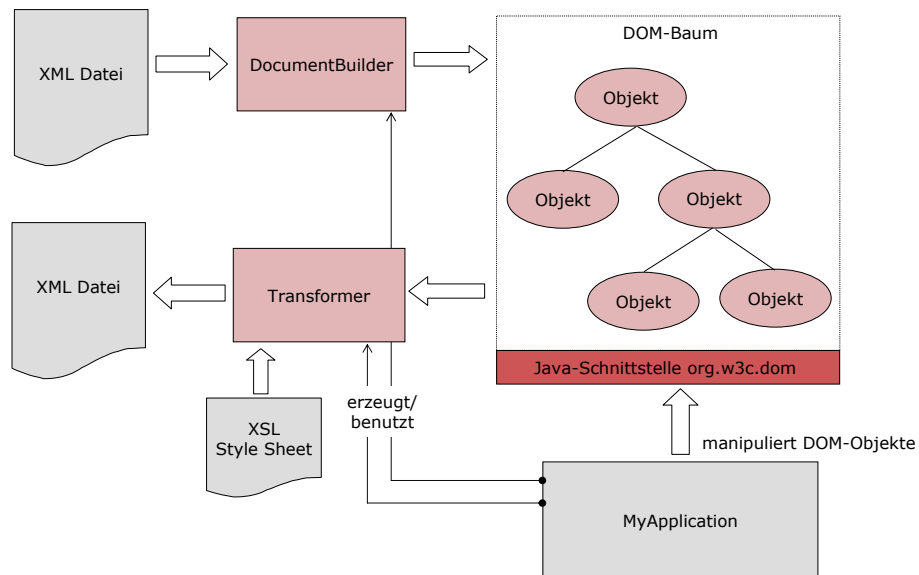
Core DOM beschreibt alle Knotentypen und Dokumente in der abstrakten Sprache IDL. Das wichtige Interface 'Node' sieht beispielsweise so aus:

```
interface Node {
    readonly attribute Node      parentNode;
    readonly attribute NodeList  childNodes;
    Node appendChild(newChild);
    Node insertBefore(newChild, refChild);
    Node removeChild(oldChild);
    replaceChild(newChild, oldChild);
    readonly attribute DOMString nodeName;
    attribute DOMString nodeValue;
};
```

In den Language Bindings für JavaScript und Java wird diese abstrakte Definition entsprechend umgesetzt.

DOM Level 3 ist in Bearbeitung. Definiert unter anderem Aufgaben und Spezifikation des Parsers.

Arbeiten mit DOM



- DOM Spezifikation
- Applikation
- Hersteller-Spezifisch

Eine XML Datei wird von einem DocumentBuilder-Objekt (intern meist via SAX Parser) eingelesen und als Baum von Objekten in Java dargestellt. Die Objekttypen (Klassen) des DOM-Baumes sind selbst in einer Vererbungshierarchie organisiert. Zuoberst in der Hierarchie ist der allgemein Typ 'Node' (siehe nächste Folie). Ein DocumentBuilder-Objekt wird zum Beispiel im Package javax.xml.parsers von der Klasse DocumentBuilderFactory zur Verfügung erzeugt.

Der DOM-Baum kann mit den Mitteln der Programmiersprache beliebig modifiziert und anschliessend via Transformer-Objekt wieder als XML-Datei zurückgeschrieben werden. Die Klasse TransformerFactory kann hierzu ein Transformer-Objekt erzeugen, das den DOM-Baum exakt in XML transformiert (Identity Transformation). Soll der DOM-Baum via Style-Sheet modifiziert und zurückgeschrieben werden, so stellt beispielsweise Apache im Package org.apache.xalan.transformer die Klasse TransformerImpl zur Verfügung, welche eine Unterklasse von javax.xml.parser.Transformer ist (Siehe Apache Xalan Projekt). In Java 1.4 ist das Arbeiten mit DOM 2.0 komplett integriert.

Arbeiten mit DOM, Beispiel

```
public static void main( String argv[] )
{
    ...
    DocumentBuilderFactory f;
    f = DocumentBuilderFactory.newInstance();
    f.setValidating(true);
    DocumentBuilder b = f.newDocumentBuilder();
    Document xmldoc = b.parse( argv[0] );
    visit( xmldoc ); // Lesen und bearbeiten des Baumes
    TransformerFactory tf;
    tf = TransformerFactory.newInstance();
    StreamSource xsl = new StreamSource( argv[1] );
    Transformer t = tf.newTransformer( xsl );
    DOMSource source = new DOMSource( xmldoc );
    File outFile = new File( argv[2] );
    StreamResult result = new StreamResult( outFile );
    t.transform( source, result );
}
```

Dieses Beispiel liest eine XML-Datei ein. Anschliessend werden mit der Funktion `visit()` rekursiv alle Objekte traversiert und der Nodename sowie der Nodewert ausgegeben. Namen aus dem DOM-Standard sind fettgedruckt. Anschliessend wird der DOM-Baum in ein File (`argv[2]`) geschrieben. Zu beachten ist, dass die Funktionen zum Einlesen und Zurückschreiben des DOM-Baumes nicht Teil des DOM-Standards sind. In diesem Beispiel wurde dafür das Transformation API von Java 1.4 verwendet.

Source Beispiel siehe Verzeichnis `DOM\Visitor` Dateien `DOMVisitor.java`, `DOMVisitor2.java` und `test-data-with-all-nodetypes.xml`, sowie `TrAX\Vorlagen\DOMTransform.java` und `TrAX\Vorlagen\XSLTransform.java`

Arbeiten mit DOM, Beispiel

- Die `visit`-Funktion traversiert den DOM-Baum und gibt die Knoten aus:

```
public static void visit( Node node )
{
    ...
    System.out.println(
        node.getNodeName() + "=" + node.getNodeValue() );

    // hier könnte der Knoten auch modifiziert werden

    NodeList children = node.getChildNodes();
    for ( int i=0; i < children.getLength(); i++ ) {
        Node child = children.item(i);
        visit( child );
    }
}
```

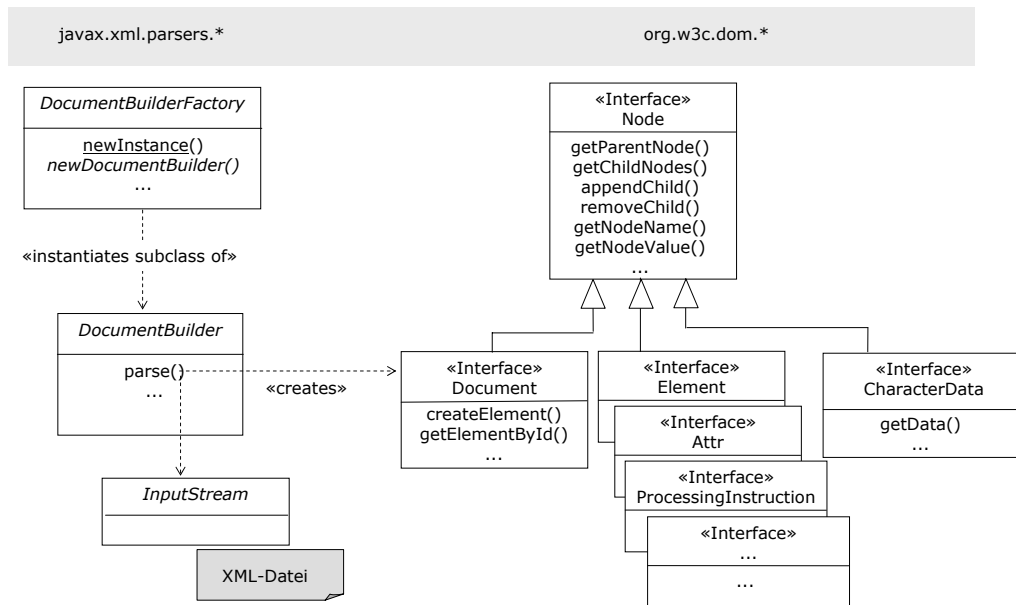
Zu Beachten bei der Ausgabe dieses Programmes:

- Der erste ausgegebene Knoten ist das Dokument selber
- Der zweite ausgegebene Knoten ist häufig die DOCTYPE Deklaration

Die Attributknoten werden in diesem Programm nicht ausgegeben.

Ausführlicheres Beispiel siehe `DOMVisitor.java` und `test-data-with-all-nodetypes.xml`

DOM Package Übersicht



Es existieren folgende, von Node abgeleiteten Interfaces:

- Node
 - Attr
 - CharacterData
 - CDATASection
 - Comment
 - Text
 - Document
 - DocumentFragment
 - DocumentType
 - Element
 - Entity
 - EntityReference
 - Notation
 - ProcessingInstruction

DOM Knoten, wichtige Eigenschaften

Interface	getNodeName()	getNodeValue()	getAttributes()
Attr	name of attribute	value of attribute	null
CDATASection	"#cdata-section"	content of the CDATA Section	null
Comment	"#comment"	content of the comment	null
Document	"#document"	null	null
DocumentFragment	"#document-fragment"	null	null
DocumentType	document type name	null	null
Element	tag name	null	NamedNodeMap
Entity	entity name	null	null
EntityReference	name of entity referenced	null	null
Notation	notation name	null	null
ProcessingInstruction	target	content excluding target name	null
Text	"#text"	content of the text node	null

Viele Knoten besitzen überflüssige Eigenschaften. So hat zum Beispiel ein Element nie einen "Value", sondern nur der dafür vorgesehene Text-Knoten. Ein Element-Knoten kann mehrere Textknoten nacheinander enthalten, je nachdem, wie geparsed wurde. Mit der Funktion `Node.normalize()` können nebeneinanderliegende Textknoten zusammengefasst werden.

XPath und DOM

- DOM stellt keine Möglichkeit für das Suchen nach Knoten zur Verfügung. Jedoch gibt es im Java SDK 1.5 ein XPath API für DOM.
- Das Resultat einer XPath-Abfrage ist eine Liste von *Referenzen* auf Knoten. Die Knoten können also über das DOM-API *modifiziert* werden:

```
Document doc = ...;
XPath xpath = XPathFactory.newInstance().newXPath();
NodeList nodeList = (NodeList)
    xpath.evaluate( query, doc, XPathConstants.NODESET );
for( int i = 0; i < nodeList.getLength(); i++ ) {
    Node n = nodeList.item(i);
    // Knoten n verarbeiten, z.B. n.setNodeValue(...)
}
```

Die resultierende Nodeliste enthält alle DOM-Knoten, welche dem XPath-Ausdruck genügen.

Soll das Resultat einer Abfrage für eine Transformation mit XSL verwendet werden, oder soll das Resultat als Ganzes weiterverwendet werden, müssen die abgefragten Knoten kopiert und unter einem einzigen Resultatknoten zusammengefasst werden. Eine Kopie ist deshalb notwendig, weil ein Knoten nur einen einzigen Elternknoten haben darf. Unterstellt man einen Knoten einem neuen Elternknoten, wird er automatisch beim alten Elternknoten entfernt. Beispiel:

```
protected Element queryByCopy( String query ) throws Exception
{
    Element qresult = doc.createElement( "xpath-query-result" );
    XPath xpath = XPathFactory.newInstance().newXPath();
    NodeList nodeList = (NodeList)
        xpath.evaluate(query, doc, XPathConstants.NODESET);
    for( int i = 0; i < nodeList.getLength(); i++ ) {
        Node n = nodeList.item(i);
        Node copy = n.cloneNode( true );
        qresult.appendChild( copy );
    }
    return qresult;
}
```

Der Aufruf `doc.createElement()` erzeugt ein neues Element. Dieses ist jedoch *nicht* ein Kind des Dokumentknotens, sondern ein freier Knoten.

Der folgende Code verwendet die Funktion `queryByCopy()` und gibt das Resultat mit Hilfe eines Transformers auf die Konsole aus.

```
public static void main()
{
    ...
    doc = builder.parse( ... );
    String query = "//book[title='XML Bible']";
    Element qresult = queryByCopy( query );
    DOMSource source = new DOMSource( qresult );
    StreamResult result = new StreamResult( System.out );
    transformer.transform( source, result );
    ...
}
```

Aufgaben DOM

1. Ergänzen Sie die Klasse `DOMVisitor` so, dass auch Attribute von Elementen ausgegeben werden und für jeden Knoten der Knoten-Typ. Entsprechende Funktionen sind im Java API im Interface `org.w3c.dom.Node` und `org.w3c.dom.Element` zu finden.

Prüfen Sie die Ausgabe, beispielsweise anhand von `catalog-classic.xml`.

2. Erstellen Sie ein Programm, dass einen XQuery-Ausdruck entgegennimmt, auf einen DOM-Baum anwendet, und das Resultat auf Konsole ausgibt
(Erst ab JDK 1.5: `PATH` und `CLASSPATH` und anpassen)

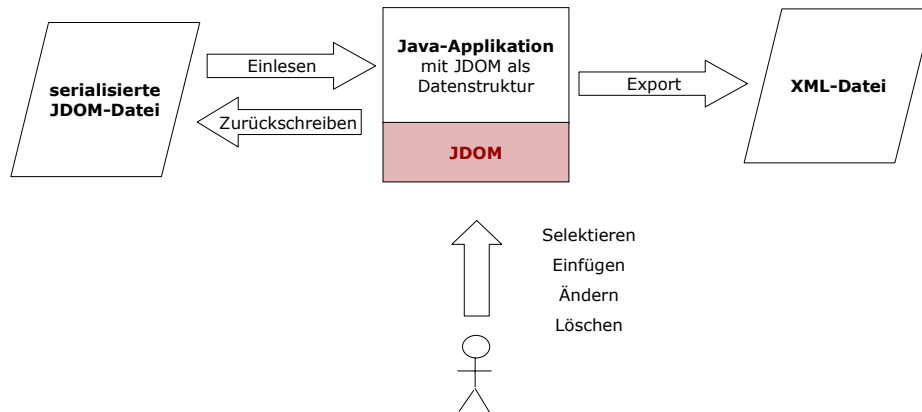
Vorlagen finden Sie unter

```
... \xml-db-workshop\DOM\Visitor\Vorlagen  
und  
... \xml-db-workshop\DOMXPath\Vorlagen
```

HTML-Dokumentationen

In der Java Api Doc 1.4 in den Packages `javax.xml.*` und `org.w3c.dom`

JDOM



Der Anwendungsbereich ist naturgemäss ähnlich wie für DOM. JDOM ist ein Alternativ-Produkt zu DOM und je nach Vorhaben besser oder weniger gut geeignet als eine bestimmte DOM-Implementation.

Vergleich von JDOM mit DOM

JDOM

1. Auf dem Weg zur Java Specification
2. Nur java-bezogen
3. Direkt instanzierbare Klassen
4. Klassen sind unabhängig
5. Klassen sind serialisierbar
6. Strenge, überprüfte Baumstruktur
7. Input aus Datei oder DOM-Baum
8. Output in Datei, SAX-Handler, DOM-Baum
9. XSL-Transformationen über TrAX-API von Sun (javax.transform)

DOM

1. W3C Standard
2. Auf beliebige Sprache bezogen
3. Interfaces verstecken Implementation
4. Interfaces mit Vererbungshierarchie
5. Implementationsabhängig
6. Strenge, überprüfte Baumstruktur
7. Input/Output/Transformation über DOM- und SAX-Parser, Transformation API's von Drittherstellern (Apache, IBM, Sun)

Vorteile/Nachteile

JDOM

- + Self Contained Package
- + Explizite Serialisierbarkeit macht Zusammenarbeit mit eigener Datenbank möglich
- + Verwendung von `java.util.List` für die Traversierung des Baumes statt proprietäre `NodeList`
- Ein einziger Hersteller

DOM

- + Entwickler kann Implementationen verschiedenster Hersteller auswählen (Apache, Oracle, ...), mit oder ohne Datenbank als Speichermedium.
- + Implementationen von DOM in C++/Java/JavaScript und andere Sprachen
- überflüssige Methoden wegen Vererbungshierarchie, z.B. `getChildNodes()` für Text-Knoten

Teil IV

Trax Transformation API for XML

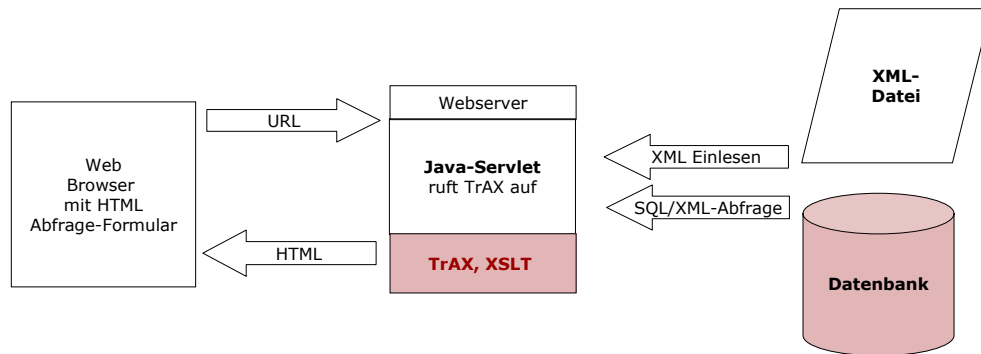
Informationsquellen

- <http://xml.apache.org/xalan-j/trax.html>http://java.sun.com/xml/xml_jaxp.html
- http://java.sun.com/xml/jaxp-1_1-spec.pdf

Anwendungsbeispiel TrAX

Ausgabe von HTML-formatierten Daten in einen Browser

- aus einer Datenbank
- aus einer XML-Datei



TrAX definiert eine Java-Schnittstelle, wie XML-Daten eingelesen, mit XSL-Stylesheets transformiert und wieder ausgegeben werden können. TrAX ist ein rein von Sun definiertes Interface, im engeren Sinne also kein Standard, wird aber nahezu überall eingesetzt.

Daten für die Transformation können aus verschiedenen Quellen stammen: Typischerweise ein XML-Datenstrom oder ein DOM-Baum.

Das Resultat der Transformation steht ebenfalls in verschiedenen Varianten zur Verfügung: Typischerweise als XML-Datenstrom oder als DOM-Baum.

Im obigen Beispiel schickt ein Webbrowser bestimmte Abfragebedingungen für die Datenbank an ein Java-Servlet. Dieses führt die SQL-Abfragen über JDBC gegen die Datenbank durch. Aus dem Resultat wird ein DOM-Baum oder ein XML-Datenstrom erzeugt. Dieser wird vom Transformator in HTML umgewandelt und an den Browser weitergeschickt.

Entscheidend ist, dass die aus der Datenbank gewonnenen Daten vorerst im XML-Format bereitstehen. Erst die XML-Darstellung wird anschliessend durch eine Transformation mit Hilfe von TrAX und XSLT in HTML umgewandelt. Das ermöglicht eine gute Trennung von Datenbank-Zugriffen und Erzeugung der Präsentation.

Die Erzeugung von XML-Elementen aus SQL-Abfragen ist relativ einfach. Viele Datenbanksysteme unterstützen es im Rahmen der SQL-Syntax, zum Beispiel:

```
select * from Book FOR XML
```

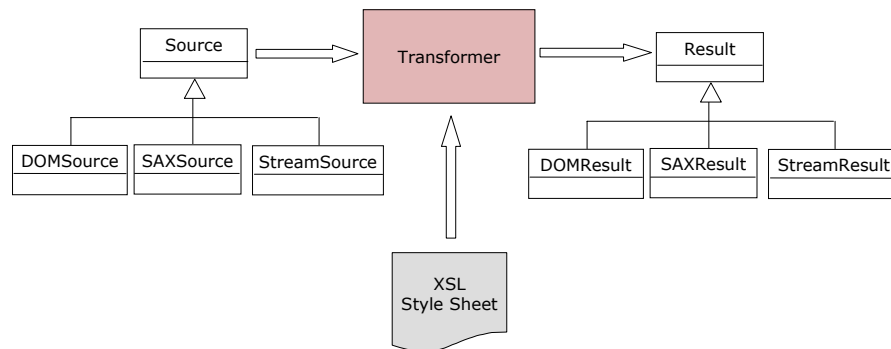
Möglich ist auch eine selber gestrickte Abfrage der Art:

```
select "<title>" + title + "</title>" from Book where ...
```

Häufig wird TrAX auch für eine Transformation von XML nach XML benutzt, beispielsweise um eine flache, datensatzorientierte XML-Darstellung in eine verschachtelte, hierarchische XML-Darstellung umzuwandeln. Dies kann für den Gebrauch der XML-Daten in einem Editor oder für den Transport zu anderen Applikationen notwendig sein. Beim Datenaustausch zwischen Unternehmen hat die hierarchische Darstellung den Vorteil, dass sie weniger mit Schlüsselreferenzen zwischen den Elementen arbeiten muss. Siehe Beispiel `catalog-classic-2-catalog-generic.xsl`

Zweck

- Transformation einer beliebigen XML-Eingabestruktur mit XSLT in eine beliebige XML-Ausgabestruktur.
- Ein- und Ausgabe-Strukturen sind wahlweise kombinierbar.



Die von SUN definierten Packages für Transformationen sind :

```
javax.xml.transform
javax.xml.transform.sax
javax.xml.transform.dom
javax.xml.transform.stream
```

DOM-Source: Zeigt auf ein Node-Objekt unter welchem ein ganzer Baum weiterer DOM-Objekte angehängt ist.

DOM-Target: Ein Node-Objekt an welches das Resultat der Transformation angehängt wird. Muss entweder vom Typ Document, Element oder DocumentFragment sein. Wenn null, wird ein neues Document-Objekt erzeugt. Der erste Knoten unter dem Document-Objekt ist das Root-Element der XM-Daten.

SAXSource: In diesem Objekt ist im Wesentlichen ein Character-Strom, Byte-Strom oder ein URL auf eine XML-Datei verpackt.

SAXResult: In diesem Objekt ist im Wesentlichen ein SAX-Handler verpackt, dessen Funktionen während dem Abarbeiten der transform()-Funktion aufgerufen werden.

StreamSource: Ein Datenstrom von welchem Daten im XML-Format gelesen werden.

StreamResult: Ein Datenstrom in welchen Daten im XML-Format geschrieben werden.

DOM -> XML, Beispiel

```
public static void main( String argv[] )
{
    DocumentBuilderFactory df;
    df = DocumentBuilderFactory.newInstance();
    df.setValidating(true);
    DocumentBuilder b = df.newDocumentBuilder();
    Document xmldoc = b.parse( argv[0] );
    // ... manipulate DOM tree here ....
    TransformerFactory tf;
    tf = TransformerFactory.newInstance();
    StreamSource xsl = new StreamSource( argv[1] );
    Transformer t = tf.newTransformer( xsl );
    DOMSource source = new DOMSource( xmldoc );
    File outFile = new File( argv[2] );
    StreamResult result = new StreamResult( outFile );
    t.transform( source, result );
}
```

Dieses Beispiel liest eine XML-Datei aus argv[0] als DOM Baum ein, transformiert diesen gemäss dem Style in argv[1], und speichert das Transformationsresultat als XML Datei in argv[2].

Codiertes Beispiel siehe DOMTransform2.java

Transformer-Einstellungen

- Mitgeben der DOCTYPE – Deklaration:

```
transformer.setOutputProperty( "method", "xml" );  
transformer.setOutputProperty( "doctype-system",  
                                doc.getDoctype().getSystemId() );
```

- Ausgabe-Formatierung mit Zeilenumbrüchen:

```
transformer.setOutputProperty( "method", "xml" );  
transformer.setOutputProperty( "indent", "yes" );
```

- Mitgeben von Parametern für das Stylesheet
(\$-Variablen in XSL):

```
transformer.setParameter( name, wert )
```

Aufgaben TrAX

1. In der Vorlage `DOMTransform.java` wird eine XML-Datei eingelesen und dieselbe XML-Datei wieder ausgegeben (Identitätstransformation). Bauen Sie eine Transformation mit einem ihrer eigenen Stylesheets ein.

HTML-Dokumentationen

In der Java Api Doc 1.4 zu finden

Gute Parser sind auch bei Apache unter dem Namen Xalan und bei IBM unter dem Namen xml4j zu finden.

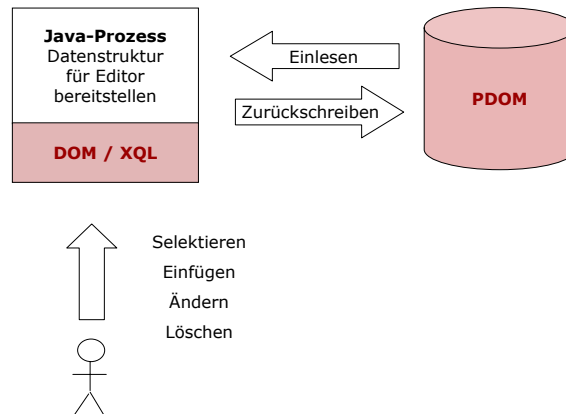
Teil V

XML-Datenbanken am Beispiel Persistent DOM (PDOM)

Informationsquellen

- <http://www.rpbourret.com/xml/> zu finden
- <http://www.infonyte.de>

Anwendungsbeispiel PDOM/XQL



PDOM bezeichnet eine Technologie, bei der ein DOM-Dokument und seine DOM-Objekte auf ein Speichermedium abgelegt werden. Gegenüber der Ablage in einer XML-Datei besteht der Vorteil, dass nicht bei jedem Einlesen in die Applikation das ganze DOM-Dokument geparsed und geladen werden muss. Eine PDOM-Datenbank stellt (transparent für den Anwendungsentwickler) einen Speicherverwaltungsmechanismus zur Verfügung, so dass nur effektiv benötigte Knoten sich im Hauptspeicher befinden müssen. Ausserdem stellen PDOM-Datenbanken meist mehr oder weniger ausgefeilte ACID-Mechanismen zur Verfügung.

PDOM-Produkte bieten häufig auch eine Abfragesprache an (XQL, XPath, XQuery). Damit ist das Auffinden bestimmter Elemente wesentlich einfacher, flexibler und performanter als einzig durch die Methoden der DOM-Interfaces.

Übersicht

Definition einer XML Datenbank: Die XML-Struktur von Daten ist *in* der Datenbank erkennbar/abgebildet und muss nicht durch eine Applikationslogik zusammengebaut werden.

1. XML-Enabled Databases
relationale Systeme mit XML-Erweiterungen in SQL
2. Native XML-Datenbanken
Speicherung von XML-Elementen in Objektdatenbanken, generischen Tabellen usw.
3. Persistente DOM-Bäume
Speicherung von DOM-Knoten (abgeschlossene Menge von Klassen) in einer Datenbank als serialisierte Java-Objekte.

XML-Enabled Databases

Relationale Datenbanken können z.B. flache XML-Ausgaben erzeugen mit der FOR XML-Erweiterung (SQL-Server, Oracle) in Abfragen.

Beispiel

```
select * from Publisher where idPub = 3 FOR XML AUTO
```

Ausgabe

```
<root><row idPub="3" name="Addison Wesley" url="http://www.addison-wesley.com/" /></root>
```

Ansätze für die Unterstützung von XPath sind vorhanden, z.B. bei SQL Server.

Native XML-Datenbanken

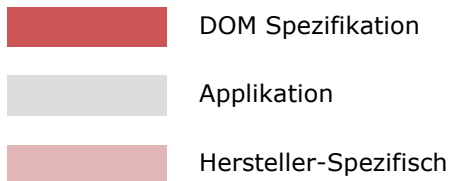
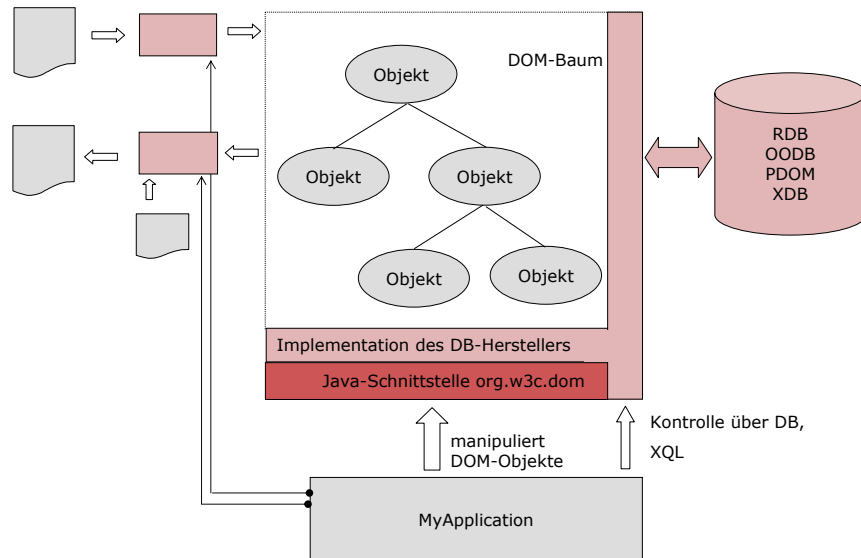
äusserlich komplett XML basiert, Abfragesprache XPath- oder XQL-ähnlich, eigene Modifikationssprache. Innerlich relationale oder objektorientierte Datenbanksysteme. Als Beispiel sei hier www.exceloncorp.com erwähnt. Das generische RDB-Design kann bereits als Native XML-Datenbank gelten.

Persistente DOM Bäume

Einfach zu verwenden, flexible Abfragesprache XQL oder XPath, limitierte Datenmenge, ACID-Regeln und Transaktionskonzept weniger ausgefeilt als bei RDB's.

Im Prinzip enthält ein persistenter DOM-Baum dieselbe Information wie eine korrespondierende XML-Datei. Ein DOM-Baum im Speicher kann jedoch stückweise mit seiner entsprechenden PDOM-Datei synchronisiert werden: Objektbasiertes Caching erlaubt es, dass nicht der ganze DOM-Baum gelesen und wieder geschrieben werden muss. Ausserdem entfällt bei Einlesen das Parsen und allenfalls das Validieren der XML-Datei.

Arbeiten mit DOM und Datenbanken



Was bringt eine Datenbank für die Verwaltung von DOM-Dokumenten:

- Es muss nicht ein ganzes Dokument in die Applikation geladen werden, sondern nur die benötigten Knoten
- Auf die Knoten eines Dokumentes kann von verschiedenen Applikationen zugegriffen werden
- Die Selektion von Knoten mittels Abfragesprachen wie XPath, XQL, XQuery ist präziser und schneller mit einer Datenbank
- Eine Datenbank hat Recovery-Möglichkeiten
- Eine Datenbank kann mit sehr grossen Dokumenten umgehen

PDOM = Persistent DOM

- Als Grundlage für die Navigation und Bearbeitung eines XML-Dokumentes dient das Java DOM API.
- Als Grundlage für die Speicherung (Persistenz) dient der Java Serialisierungs-Mechanismus:
 - Jedes DOM-Objekt bekommt eine eindeutige ID.
 - Bei der Speicherung wird ein DOM-Objekt in einen Byte-Array serialisiert und dieser in der PDOM-Datei des Dokumentes abgelegt.
 - Ein zusätzlicher Index enthält die Information, wo in der Datei ein Objekt mit einer bestimmten ID abgelegt ist.
 - Bei Bedarf wird der Byte-Array anhand seiner ID wieder in die Applikation deserialisiert.
 - Referenzen auf andere Objekte (Kind-Knoten) sind als Liste von ID's im Elternknoten implementiert.

Die Bezeichnung 'Persistente DOM-Bäume' widerspiegelt die Sichtweise des Entwicklers, welcher XML-Daten in Form eines Java DOM-Baumes sehen will. Die Behandlung eines persistenten DOM-Baumes unterscheidet sich für ihn nicht von der Behandlung eines transienten DOM-Baumes, ausser dass er vielleicht ab und zu Änderungen in die Datenbank committen muss. Ausserdem wird er zu Beginn der Applikation ein Objekt erzeugen oder referenzieren (laden), dass den Document-Node repräsentiert. Mit diesem Objekt kann er anschliessend nach Belieben den DOM-Baum traversieren und modifizieren.

Als Speichermedien kommen verschiedene Möglichkeiten in Frage:

- Die billigste Variante ist sicher eine einfache Serialisierung des gesamten DOM-Baumes in eine Datei. Diese Datei wird bei Applikationsstart komplett geladen und am Schluss der Applikation oder zu gewünschten Zeitpunkten komplett auf die Datei zurückgeschrieben.
- Als Verbesserung kann ein knotenbasiertes Laden und Schreiben eingeführt werden (wie oben beschrieben). Das Cache-Management bleibt dabei der Applikation verborgen. Mit dieser Verbesserung muss allerdings auch eine Verwaltung der Objektreferenzen (OID's) und ein Garbage-Collection eingeführt werden.
- Relationale Datenbanken eignen sich ebenfalls zum Ablegen von DOM-Knoten, insbesondere wenn aus einem grossen Datenbestand jeweils nur Teilbäume in die Applikation geladen werden sollen. Der zeitliche Overhead für das Laden von Knoten wird im Vergleich zu serialisierten DOM-Bäumen allerdings recht hoch sein. Dagegen wird man von einem sehr guten Transaktionssystem profitieren können (Concurrency-Control, Recovery-Sicherheit, Atomarität).

PDOM, Beispiel Infonyte DB

- Implementation von org.w3c.dom
- Speichern/Laden eines DOM-Baumes
- Konversion XML->PDOM, PDOM->XML
- Selektion von Teilbäumen, Knoten, Attributen mit XQL
- Java API und Commandline Tools

Produkt

Infonyte (www.infonyte.de) ist ein Spin Off des German National Research Center for Information Technology in Darmstadt, siehe <http://www.darmstadt.gmd.de/IPSI/>

Kosten

980 € für Developer-Lizenz, 180 € für Runtime-Lizenz.

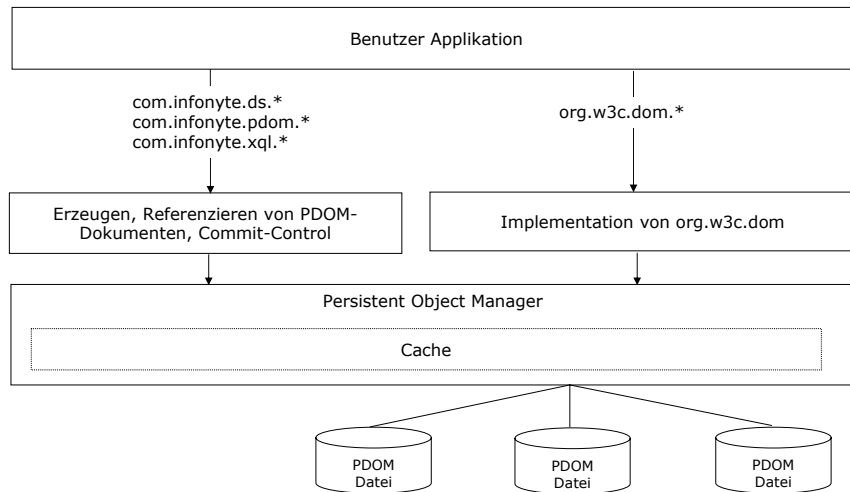
Anwendung

- Datenverarbeitung im Middleware-Bereich.
- Ein mitgeliefertes Servlet ermöglicht das Abfragen von PDOM-Dateien mit XQL über WWW-Formulare oder einen URL mit Parametern.
- Bei einer Dateigrösse von 10 Mbyte ergeben sich Traversierungszeiten von grob 0.5 sec. Die maximale Grösse einer PDOM Datenbank dürfte also zwischen 10 MB und 100 MB liegen.
- Infonyte PDOM ist vorallem für abfragende Applikationen mit eher kleiner Modifikationshäufigkeit geeignet.

Technologie

Rein Java-basiert. Kann als Referenzimplementation für XQL betrachtet werden. Die Definition von Abfragesprachen in der XML Welt ist noch im Fluss. Das von Infonyte verwendete XQL (XQL98, XQL99) ist eine der Hauptgrundlagen für XQuery, dem Standardisierungs-Vorschlag des W3C Konsortiums.

API-Struktur von Infonyte



- Die Funktionen im Package `com.infonyte.ds` dienen der Abstraktion des Speichermediums (Datei, Datenbank, Memory).
- Die Funktionen im Package `com.infonyte.pdom` dienen dem Erzeugen oder Referenzieren von persistenten DOM-Bäumen. Ausserdem enthält es Funktionen für die Commit-Funktionalität und das Komprimieren von PDOM-Files.
- Mit Hilfe des Package `com.infonyte.xml` können XQL Anfragen über DOM-Bäume durchgeführt werden.
- Das Package `org.w3c.dom` enthält alle Funktionen zum Bearbeiten eines DOM-Baumes unabhängig davon, ob er persistent oder transient ist.

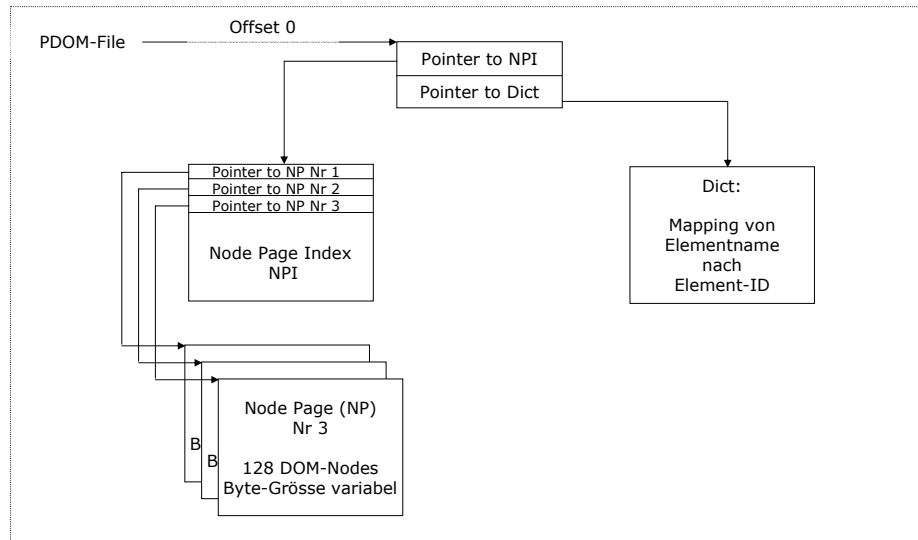
Eigenschaften von Infonyte PDOM

- Caching
- Commit-Mechanismus
- Garbage Collection
- Defragmentierung
- GZIP Kompression
- Locking auf Datei-Ebene
- Multithreading

Ein technischer Übersichtsartikel der Entwickler ist zu finden unter:

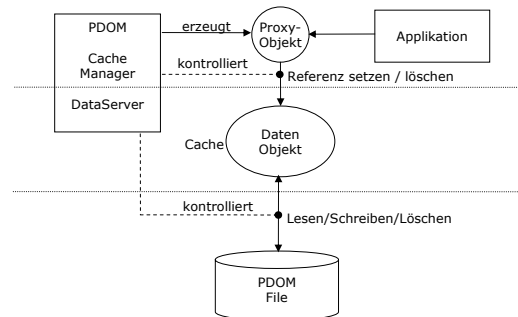
<http://www.ipsi.fhg.de/oasys/reports/ftp/pdf/P1999-27.pdf>

Infonyte PDOM-File



- PDOM-Dateien sind in drei Bereichen organisiert. Ganz am Anfang einer Datei stehen zwei Pointer: einer auf einen Dictionary, welcher Elementnamen und Attributnamen in entsprechende numerische Schlüsselwerte übersetzen kann. Ein Pointer zeigt auf den *Node Page Index* (NPI). Dieser enthält eine Liste von Pointern auf *Node Pages*. Eine Node Page ist ein zusammenhängender Block von 128 Objekten. Innerhalb einer Node Page sind *Objekte* sequentiell aneinandergehängt. Wurden eines oder mehrere Objekte auf einer Node Page geändert, wird zum Commit-Zeitpunkt die ganze Node Page am Schluss des Dokumentes neu aufgebaut und der entsprechende Zeiger im Node Page Index nachgeführt. Häufiges Ändern von Objekten führt daher zu einer grossen Platzverschwendung im vorderen Teil der Dateien. Es wird deshalb eine **Garbage-Collection**- und eine **Defragmentierungs**-Methode bereitgestellt. Die Garbage-Collection-Methode entfernt nicht mehr referenzierte Objekte innerhalb einer Node-Page (Verkleinerung der Node-Page). Die Defragmentierungs-Methode kopiert alle noch benötigten Node Pages in eine neue PDOM-Datei. Die Pointer im NPI werden entsprechend angepasst. In der Version 2.0 müssen nicht mehr gebrauchte DOM-Knoten explizit gelöscht werden. Ansonsten verbleiben sie im PDOM-File und können dort wieder gefunden werden, wenn die Node-ID bekannt ist.
- Jeder DOM-Node bekommt bei der Erzeugung eine eindeutige ID (OID = Object ID). Bei Infonyte hat die OID mit dem Ort der Speicherung folgenden Zusammenhang: $OID = \text{Pagenummer} * 128 + \text{Objektnummer}$. Aus jedem OID kann also sofort berechnet werden, auf welcher Node-Page er sich befindet und das wievielte Objekt er auf dieser Node Page ist. Enthält ein DOM-Knoten Referenzen auf andere Knoten, werden diese als Liste von OID's im DOM-Knoten abgelegt. Das Wurzelobjekt hat immer die OID 0.
- Enthält eine Datenbank umfangreiche Textinformation, kann eine **komprimierte Speicherung** einer Node-Page erfolgen (GZIP-Algorithmus).
- Weitere Details unter <http://www.ipsi.fhg.de/oasys/reports/ftp/pdf/P1999-27.pdf>

Infonyte Caching und Commit-Mechanismus



- Der **Caching**-Mechanismus erlaubt das Arbeiten mit grossen Dateien und einer vollständigen Kontrolle über die Cache-Grösse und das Aufrufen des Garbage-Collectors. Das Caching ist so organisiert, dass für jedes Datenobjekt ein Proxy-Objekt erzeugt wird. Die Applikation sieht nur das Proxy-Objekt. Der Cache-Manager kann die Referenz auf das eigentliche Objekt im Proxy-Objekt beliebig manipulieren (freigeben oder neu belegen). Im Proxy-Objekt wird nur über Methoden auf das Datenobjekt zugegriffen. Diese Methoden (alle aus dom.w3c.org) sind so implementiert, dass sie auf das Datenobjekt zugreifen, wenn es vorhanden ist, resp. vom Cache-Manager anfordern, wenn es neu zu laden ist.
- Der **Commit**-Mechanismus erlaubt das gezielte Synchronisieren des Caches mit den Disk-Dateien. Modifizierte DOM-Objekte werden zum Commit-Zeitpunkt am Ende der Dateien neu plaziert. Erst wenn dies gelungen ist, werden die Verweise auf den neuen Speicherort eines Objektes, am Anfang der Disk-Dateien, nachgeführt. Da dies die einzige kritische Operation ist, wird der Zeitbereich für den ein Systemabsturz zu korrupten Dateien führen kann, sehr klein. Ein Rollback kann ebenfalls durchgeführt werden: Der Cache wird ohne Zurückschreiben gelöscht.
- Der Zugriff von zwei verschiedenen Prozessen (Java Virtual Machines) wird über ein Lockfile mit dem Namen `dateiname.***` synchronisiert.
- Innerhalb desselben Prozesses (Java Virtual Machine) können mehrere Threads gleichzeitig lesend auf die Datenbank zugreifen. Sobald ein Thread Änderungen durchführt, blockiert er alle anderen Leser- und Schreiber-Threads.

Erstellen einer PDOM-Datei

```
...
DataServerFactory dsf = new FileDataServerFactory();
PDOMFactory pdf = new PDOMFactory (dsf);
File store = new File( argv[1] );
PDOM pdom = pdf.create( store );
PDOMParserFactory prsf = new PDOMParserFactory();
PDOMParser prs = prsf.newParser();
prs.setWhitespacePolicy( PDOMParser.SKIP_ALL_WS );
File xmlfile = new File( argv[0] );
prs.parse( xmlfile, pdom );
// Erstelle DOM Dokument nach W3 Standard
Document doc = pdom.getDocument();
pdom.commit();
....
```

Eine Infonbyte-Datenbank hat zwei Architektur-Schichten

1. Ein `DataServer` abstrahiert das verwendete Speichermedium, beispielsweise File, Datenbankserver usw. In der gegenwärtigen Version stehen nur Files als Speichermedium zur Verfügung. Eigene `DataServer` können jedoch entwickelt und von Infonbyte benutzt werden. Der `DataServer` handelt mit `NodePages` (Segmenten). Er kann diese bereitstellen, beschreiben, lesen, löschen, committen, rollbacken. Der `DataServer` realisiert das Cache-Management.
2. `PDOM` repräsentiert ein `org.w3c.Document` und stellt einige Erweiterungen bereit, beispielsweise um DOM-Knoten mit einer eindeutigen ID zu versehen, nach einer bestimmten ID zu suchen oder einen Knoten einer bestimmten ID zu löschen. Mit der Methode `getDocument()` erhält man exakt ein `org.w3c.Document`-Objekt. `PDOM` stellt im Weiteren `commit`- und `rollback`-Methoden zur Verfügung. Ein `PDOM`-Objekt wird immer mit einer Referenz auf einen bestimmten 'Store' erzeugt. Ein 'Store' ist der externe Name für ein Dokument. Im Sinne des Cache-Managements stellt `PDOM` die Proxy-Objekte zur Verfügung. Werden mehrere `PDOM`-Objekte auf denselben 'Store' erzeugt, hat jedes `PDOM`-Objekt einen eigenen Satz von Proxy-Objekten, diese zeigen jedoch alle auf dieselben effektiven DOM-Nodes.

Der `PDOMParser` erstellt aus einem XML-File ein `PDOM`-File. Der Parser ist per default nicht validierend, es kann jedoch mit `PDOMParser.setSAXParser()` ein validierender Parser verwendet werden.

Zu obigem Beispiel:

`argv[0]` ist der Name des bestehenden XML Files

`argv[1]` ist der Name des zu erzeugenden `PDOM` Files

Komplettes Beispiel siehe `PDOM\CreatePDom.java`

Öffnen einer PDOM-Datei

```
// FileDataServer erstellen
DataServerFactory dsf = new FileDataServerFactory();
// PDOM Factory aufgrund dieses DataServers
PDOMFactory pdf = new PDOMFactory(dsf);
// File Objekt erzeugen
String pdomfile = new File( pdomFileName );
// PDOM Objekt erstellen für eine bestimmte Datei.
// die open-Methode erzeugt via DataServerFactory
// selber ein DataServer Objekt
PDOM pdom = pdf.open( pdomfile );
// DOM Document zum Abfragen und bearbeiten abholen
Document pdoc = pdom.getDocument();
```

Komplettes Beispiel siehe `PDomDocumentManager.java`.

Suchen von Elementen mit XQL

- Das Bearbeiten (Einfügen, Ändern, Löschen) von Elementen geschieht mit den Methoden im Package `org.w3c.dom`
- Das Suchen nach Elementen wäre ebenfalls mit den Methoden von `org.w3c.dom` möglich, z.B. mit `getElementsByTagName()`
- XQL bietet jedoch wesentlich mehr Möglichkeiten.

Komplettes Beispiel siehe `PDomDocumentManager.java`.

Suchen von Elementen mit XQL

```
DocumentFragment qresult = pdoc.createDocumentFragment();
XQLQuery q = new XQLQuery( query );
q.execute( pdoc, qresult );
DOMSource source = new DOMSource( qresult );
StreamResult result = new StreamResult( System.out );
transformer.transform( source, result );
```

`query` bezeichnet den Namen der abzufragenden Elemente.

Obiger Code erzeugt eine Kopie aller DOM-Knoten die von der Abfragebedingung in `query` selektiert wurden. Eine Kopie ist notwendig, weil für die Ausgabe des Abfrageresultates eine Transformation von DOM nach XML mit einem XSL-Stylesheet durchgeführt wird. Diese Transformation verlangt einen Startknoten. Alle abgefragten Knoten müssen also Kinder dieses Start-Knotens sein. Weil ein DOM-Knoten nur ein einziges Elternelement haben kann, müssen notgedrungen Kopien für die Ausgabe erstellt werden, wenn das Originaldokument nicht verändert werden darf.

Ein `DocumentFragment` ist ein Bruchstück aus einem vollständigen DOM-Dokument. Es eignet sich gut zur temporären Aufnahme von DOM-Knoten zwecks späterer Weiterverarbeitung. Theoretisch könnte in obigem Beispiel auch mit einem kompletten `Document` gearbeitet werden. Dessen Erzeugung ist aber tendenziell aufwendiger als nur für ein `DocumentFragment`. Ausserdem muss ein `Document` einen *einzigsten* Root-Knoten besitzen, während ein `DocumentFragment` *mehrere* direkte Kinder haben kann.

Die Methode `q.execute()` führt die Abfrage gemäss `query` durch und erzeugt eine Kopie aller gefundenen Knoten und ihrer Kinder in `qresult`.

Komplettes Beispiel siehe `PDomDocumentManager.java`.

Teil VI

Abfragesprachen für XML am Beispiel XQL

Informationsquellen

- Eintrittseite W3C für Abfragesprachen: <http://www.w3.org/XML/Query>
- Working Draft für XQuery: <http://www.w3.org/TR/xquery/>
- Spezifikation von XQL98: <http://www.w3.org/TandS/QL/QL98/pp/xql.html>
- Spezifikation von XQL99: <http://www.ibiblio.org/xql/xql-proposal.html>
- Übersichtsartikel <http://www.oasis-open.org/cover/xmlQuery.html>

Überblick

- XPath
 - Hauptsächlich für das Ansprechen von XML Elementen im Rahmen von XSL Stylesheets.
- XQL98 / XQL99
 - Selbstständige Abfragesprache für XML Daten.
- XQuery
 - Working Draft einer Allzweck-Abfragesprache vom W3C (30. April 2002).

Gesucht ist eine leicht verständliche Abfragesprache, die es ermöglicht, beliebige Informationsteile eines XML-Dokumentes zu suchen und als XML auszugeben.

Es existieren bereits gute Konzepte und Realisierungen für Abfragesprachen. Der Standardisierungsprozess steckt allerdings noch in den Anfängen. Trotzdem dürften die vorhandenen Produkte ausreichen, um beispielsweise XML-Dokumente mit XQL in einer Verarbeitungskette programmatisch übersichtlich und einfach zu handhaben. Die Performance ist absolut ausreichend für kleinere und mittlere Datenmengen.

Als Backend-Storage für Daten werden XML-Datenbanken und entsprechende Abfragesprachen gegenüber relationalen Systemen einen schweren Stand haben, wie schon vormals die objektorientierten DBs.

XQL

Im folgenden wird auf XQL etwas näher eingegangen. Diese Abfragesprache hat starke Querverbindungen zum Template Matching mit XPath in den XML Style Sheet (XSL). Ausserdem ist sie eine wichtige Grundlage für den Standardisierungsprozess des W3C im Vorschlag von XQuery. Die Datenbank Infonyte bietet eine praktisch komplette Implementation von XQL98 und XQL99, sowie seit Version 2.0 auch für XPath.

Unterschiede zwischen XPath und XQL

- XPath beherrscht verschiedene Datentypen (String, Number, Boolean, Nodeset) und entsprechende Operationen (Arithmetik auf Zahlen, Substring und Transformationen von Strings).
- In XPath ist eine komplette Navigation zwischen verwandten Pfaden (Eltern, Kinder, Geschwister) möglich.
- XQL beherrscht die Mengenoperationen union und intersect.
- XQL enthält ein Patternmatching auf Strings für die Suche nach Elementen und Attributen
- In XQL können Parameter zugewiesen und für die Bildung von Join Operationen verwendet werden.
- In einem XQL-Abfrageresultat kann der *ganze* Pfad bis zu den Resultatknoten dargestellt werden.

XQL, Basiskonstrukte

`/catalog/book`

findet alle Elemente und ihre Kinder vom Typ `book`

`/catalog/book/title`

findet alle Elemente (und ihre Kinder) vom Typ `title`

`/catalog/book/*`

findet alle Elemente *beliebigen* Typs unterhalb `/catalog/book`

`/*/*/isbn`

findet alle Elemente vom Typ `isbn` auf der *dritten* Stufe.

`//isbn`

findet alle Elemente vom Typ `isbn` auf *beliebiger* Stufe. Nicht äquivalent mit vorhergender Abfrage.

Eine XQL-Abfrage ist an sich immer eine Pfadangabe. Als Resultat wird das letzte Element (und alle seine Kinder) ausgegeben. Das Resultat der zweiten Abfrage ist daher:

```
<title>Learning XML</title>
```

bezüglich folgender XML-Datei:

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<catalog>
```

```
  <book>
```

```
    <title>Learning XML</title>
```

```
    <author>Erik T. Ray</author>
```

```
    <publisher>O'Reilly & Associates</publisher>
```

```
    <edition>1st edition</edition>
```

```
    <year>2001</year>
```

```
    <binding>Paperback</binding>
```

```
    <pages>350</pages>
```

```
    <isbn>0596000464</isbn>
```

```
  </book>
```

```
</catalog>
```

Der Pfad, der zu den gesuchten Elementen führt, wird nicht in das Abfrageresultat übernommen.

XQL, Bedingungen

```
/catalog/book[./title = 'Learning XML']
```

findet alle Elemente vom Typ `book`, welche als unmittelbaren Kindnoten ein `title` Element mit dem Wert `'Learning XML'` haben.

```
/catalog/book[./title = 'Learning XML']/isbn
```

findet alle Elemente vom Typ `isbn`, welche zu einem Buch mit dem Titel `'Learning XML'` haben.

```
/catalog/book[./@title = 'Learning XML']/isbn
```

findet alle Elemente vom Typ `isbn`, welche zu einem Buch mit dem Titel `'Learning XML'` haben. Der Titel ist hier ein Attribut des Elementes `book`

```
/catalog/book[title = "XML" $or$ title = "HTML" ]
```

findet alle Bücher mit XML oder HTML als Titel.

Literale Ausdrücke können in einfachen oder doppelten Hochkommas stehen.

Soll ein numerischer Vergleich durchgeführt werden, so muss der rechte Operand des Vergleichsoperators eine Zahl sein.

Für den häufig benötigten, nicht-case sensitiven Vergleich von Strings kann die Funktion `iregMatch()` verwendet werden. `RegMatch` und `iregMatch` sind auch als zweistellige Operatoren `$regmatch$` und `$iregmatch$` implementiert. Es werden reguläre Ausdrücke gemäss Perl5 unterstützt. Siehe hierzu http://www.perl.com/pub/doc/manual/html/pod/perlre.html#Regular_Expressions

XQL, Positionsangaben

```
/catalog/book[index() = 0]
```

findet das *erste* Elemente vom Typ `book`

```
/catalog/book[author[0] = 'Steven Holzner']
```

findet ein Buch mit Steven Holzner als ersten Autor

```
/catalog/book[end()]
```

findet das *letzte* Elemente vom Typ `book`

Einige Möglichkeiten aus der SQL-Welt werden in XQL noch vergeblich gesucht:

- Eine Sortierfunktion existiert nicht. Diese wird jedoch von XSL Style-Sheets angeboten.
- Die einzige Aggregatfunktion in XQL ist `count()`, die Funktionen `sum()`, `avg()`, `min()`, `max()` sind nicht vorhanden. Jedoch bietet zum Beispiel Infonyte die Möglichkeit an, XQL mit eigenen Funktionen zu erweitern.

XQL, Verschiedenes

```
/catalog/book[./title!regMatch( '^L' )]
```

findet alle Bücher mit einem Titelement, das mit 'L' beginnt.

```
/catalog/book[title $contains$ 'XML' ]
```

findet Bücher mit einem Titel der den String XML enthält

```
/catalog/book[title $icontains$ 'xml' ]
```

findet Bücher mit einem Titel der den String XML (case-insensitiv) enthält

```
/catalog/book[author = "Lisa Rein" ] $intersect$
```

```
/catalog/book[author = "Neil Bradley" ]
```

findet alle Bücher, die von beiden Autoren geschrieben wurden.

Einige Möglichkeiten aus der SQL-Welt werden in XQL noch vergeblich gesucht:

- Eine Sortierfunktion existiert nicht. Diese wird jedoch von XSL Style-Sheets angeboten.
- Die einzige Aggregatfunktion in XQL ist count(), die Funktionen sum(), avg(), min(), max() sind nicht vorhanden. Jedoch bietet zum Beispiel Infonyte die Möglichkeit an, XQL mit eigenen Funktionen zu erweitern.

Produkte

- Tamino
- eXcelon XML Platform – Enterprise Edition
- Ozone
- Infonyte

Tamino von Software AG www.softwareag.com

Echte Datenbank, ACID-Eigenschaften korrekt gehandhabt
Hochpreis-Segment

eXcelon XML Platform von Excelon Corporation ww.exceloncorp.com

Hochpreis-Segment
Performante Objektdatenbank als Basis
XPath als Abfragesprache
Eigene XML-Sprache zum Ändern Einfügen, Löschen von Daten
Echte Datenbank, ACID-Eigenschaften korrekt gehandhabt

Ozon von Ozone Database Project www.ozone-db.org

gleichzeitig PDOM und OODB (ODMG 3.0)
Gratis (GNU Library General Public License)
Keine Abfragesprache

Infonyte von Infonyte www.infonyte.de

Niedrigpreis-Segment
PDOM Implementation
XQL und XPath als Abfragesprache
ACID Eigenschaften ansatzweise vorhanden

Ausblick: XQuery

- Allzweck-Abfragesprache des W3C
- Noch kaum Produkte implementiert
- Beispiel einer Verknüpfungsabfrage (Join):

```
<descriptive-catalog>
{
  for $i in document("catalog.xml")//:item,
    $p in document("parts.xml")//part[partno = $i/partno],
    $s in document("suppliers.xml")//supplier[suppno = $i/suppno]
  return
  <item> { $p/description, $s/suppname, $i/price } </item>
  sortby(description, suppname)
}
</descriptive-catalog>
```

Neu gegenüber XQL

- Erzeugen von Elementen
- Kompletter Satz von numerischen, String, Datums-Funktionen

Analog zu SQL

- Join von Elementen
- Gruppierung
- Sortieren der Ausgabe

XQL, Aufgaben

1. Versuchen Sie folgende Abfragen in XQL zu formulieren (Datei `catalog-with-keywords.pdom`)
 1. Alle Bücher mit mehr als 1000 Seiten.
 2. Alle Bücher, welche im Titel 'Companion' an beliebiger Stelle enthalten.
 3. Alle Bücher, welche 'XML' aber nicht 'Learning' als Keyword enthalten.
Benutzung von `$intersect$` und `$not$` Operatoren
 4. Alle Bücher, deren erstes Keyword 'XML' ist.
 5. Alle *Titel* von Büchern mit dem Keyword 'XML'
 6. Alle Bücher welche von den zwei Autoren (oder mehr) "Elliott Rusty Harold" und "W. Scott Means" geschrieben wurden.
 7. Alle Bücher, welche von genau zwei Autoren geschrieben wurden (Funktion `count()` benutzen).
 8. Alle Autoren, welche für Addison-Wesley publizieren.

Übungsumgebung

Zum Erzeugen von PDOM Dateien aus XML und zum Abfragen von PDOM Dateien können Sie die Javaklassen `CreatePDom` und `PDomDocumentManager` aus `xml-db-workshop/PDOM/PDomDocumentManager` verwenden.

Referenzen

1. Eine vollständige Referenz zu XQL finden Sie unter <http://www.w3.org/TandS/QL/QL98/pp/xql.html>
2. Ergänzungen von Infonyte für XQL99 sind zu finden unter xml-db-workshop/PDOM/Infonyte-DB/V2-0-2/doc/manual.pdf
Kapitel 4.7

XQL, Aufgaben ff

2. Was ist der Unterschied zwischen der Abfrage

```
//book[$all$ author = "Elliotte Rusty Harold"]
```

und der Abfrage

```
//book[$any$ author = "Elliotte Rusty Harold"]
```

3. Beschreiben Sie in Textform, wonach folgende Abfragen suchen (Datei catalog-classic.pdom):

```
//book[@publisher = //publisher[@name = "Addison-Wesley"]/@id]
```

```
//book[//publisher[@name = "Addison-Wesley"
```

```
  $or$ @name = "McGraw Hill" ]/@id = @publisher]
```