# Seminar

# Datenbanken und Informationssysteme

## XML-Databinding

Sven Welte

23. Januar 2003

# Abstract

Nachdem sich XML in vielen Bereichen immer mehr durchsetzt, müssen leistungsfähige Verfahren gefunden werden, um mit XML-Dokumenten zu arbeiten. Hierbei sind insbesondere XML-Dokumente mit Schema interessant, wie sie auch bei WebServices auftreten. Häufig muss in diesem Zusammenhang ein XML-Dokument in Geschäftsobjekte einer Zielsprache (z.B. Java) umgewandelt werden. Ziel von XML-Databinding ist es, diese Umwandlung so weit es geht zu automatisieren. Diese Arbeit behandelt die Grundlagen von XML-Databinding und stellt einige Produkte aus dem Javabereich vor. Neben XML-Databinding wird die XML-Serialisierung in Java sowie zwei Speicherlösungen für XML-Daten (Natix, PDOM) betrachtet.

# 1 Einleitung

In der letzten Zeit hat XML in verschiedenen Bereichen stark an Bedeutung gewonnen. So wird XML inzwischen als eine Art „Superschnittstelle“ angesehen, mit der Interoperabilität zwischen verschiedenen Anwendungen erzielt werden soll. Die verstärkte Durchdringung des XML-Datenformates hat einige Bibliotheken und Standards (wie SAX, DOM) hervorgebracht, die es Entwicklern erlauben, XML-Dateien zu bearbeiten und weiterzuverwenden. Diese Standards bieten eine große Flexibilität, was beispielsweise die Manipulation der Struktur eines XML-Dokumentes anbelangt. Häufig wird diese Flexibilität aber nicht benötigt. So kommt es häufig nur darauf an, XML-Dokumente, die einem bestimmten Schema entsprechen, in einen Objektgraphen einer Zielsprache wie z.B. Java umzuwandeln. Diese Umwandlung ist mit den bestehenden Bibliotheken eigentlich nicht besonders schwierig, aber dennoch mit einem gewissen manuellen Aufwand verbunden.

Mit XML-Databinding versucht man, diesen manuellen Aufwand möglichst zu verringern, indem man bei diesen Vorgang Codegeneratoren benutzt.

Das Kapitel 2 behandelt die Grundlagen von XML-Databinding. Hier wird insbesondere auf das objektorientierte und hierarchische Datenmodell eingegangen, wobei diese mit dem Datenmodell von XML-Dokumenten verglichen werden.

Das Kapitel 3 behandelt die Vorteile von XML-Databinding in Bezug auf den Softwareentwicklungsprozess.

Kapitel 4 behandelt den kompletten Databindingprozess, wobei die einzelnen Schritte detailliert dargestellt werden. Neben dem Produkt JAXB werden die Databindingprodukte Castor, Zeus und Quick behandelt.

Kapitel 5 behandelt die in der Javaversion 1.4 neu hinzugekommene XML-Serialisierung. Hierbei werden auch die Vorteile des neuen Verfahrens aufgezeigt.

Kapitel 6 stellt zwei Speicherlösungen für XML-Daten vor. Bei Persistent DOM wird auf die gesamte Architektur eingegangen, wohingegen bei der zweiten Speicherungslösung Natix die verwendete Datenstruktur im Vordergrund steht.

Kapitel 7 bietet eine kurze Zusammenfassung verbunden mit einem Ausblick. Hierbei wird auch auf die Schwächen von XML-Databinding näher eingegangen.

# 2 Was ist XML-Databinding?

## 2.1 Prinzip von XML-Databinding

Bevor in den kommenden Kapiteln auf die Details von XML-Databinding eingegangen wird, soll hier erst einmal eine grober Überblick über die verschiedenen Bereiche gegeben werden.

Betrachtet man XML-Databinding etwas aus der Ferne, müssen eigentlich nur zwei Aufgaben erfüllt werden:

- Gegeben ist ein XML-Dokument. Aufgabe ist es, dieses Dokument in einen Objektgraphen umzuwandeln, der in der Zielsprache bearbeitet werden kann. Dieser Vorgang wird auch als *Marshalling* bezeichnet.

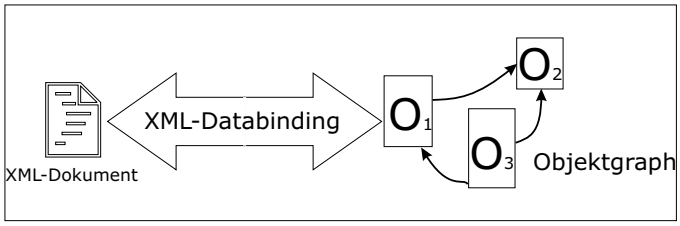
- Gegeben ist ein Objektgraph. Aufgabe ist es, diesen Graphen in ein XML-Dokument umzuwandeln. Dieser Vorgang wird auch als *Unmarshalling* bezeichnet.

Bevor man diesen Umwandlungsvorgang automatisieren kann, muss man sich als erstes Gedanken über die verschiedenen Konzepte auf Seiten des Quelldatenmodells (XML-Dokument) und des Zieldatenmodells

```
<!ELEMENT movies (movie+)>
<!ATTLIST movies
          version    CDATA    #REQUIRED
>

<!ELEMENT movie (title, cast, director?, producer*)>

<!ELEMENT cast (actor+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT producer (#PCDATA)>

<!ELEMENT actor (#PCDATA)>
<!ATTLIST actor
          headliner    (true | false)    'false'
>
```

**Abb. 1: movies.dtd**

(Objektgraph) machen. Hierzu wird als Beispiel das XML-Dokument in Abb. 2 verwendet. Wir betrachten eine mögliche Umsetzung in das Java Objektmodell.




**Prinzip XML-Databinding**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE movies SYSTEM "movies.dtd">

<movies version="1.1">
  <movie>
    <title>Pitch Black</title>
    <cast>
      <actor headliner="true">Vin Diesel</actor>
      <actor headliner="true">Radha Mitchell</actor>
      <actor>Vic Wilson</actor>
    </cast>
    <producer>Tom Engelman</producer>
  </movie>
  <movie>
    <title>Memento</title>
    <cast>
      <actor headliner="true">Guy Pearce</actor>
      <actor headliner="true">C.Anne Moss</actor>
    </cast>
    <director>Christopher Nolan</director>
    <producer>Suzanne Todd</producer>
    <producer>Jennifer Todd</producer>
  </movie>
</movies>
```

**Abb 2: movies.xml**

Auf der XML-Seite gibt es ein Schema mit dem Namen `movies.dtd`. Zu einem Schema gibt es Daten in Form von XML-Dateien (hier `movies.xml`). Häufig werden diese XML-Dateien auch als Instanz eines XML-Schemas bezeichnet. Hier kann die Verwendung des Begriffs *XML-Schema* für Verwirrung sorgen. Unter einem *XML-Schema* versteht man sowohl **die** Dokumentenbeschreibungssprache des W3C (XML-Schema [1]), als auch **eine** Dokumentenbeschreibungssprache wie z.B. DTDs, RelaxNG oder eben XML-Schema [1]. In dieser Ausarbeitung wird XML-Schema bis auf wenige Ausnahmen in der letztgenannten Bedeutung verwendet. Auf Java-Seite stehen dem XML-Schema die Klassen gegenüber, wobei den Daten einer XML-Datei auf der XML-Seite eine Objektinstanz in Java gegenübersteht. Attribute in XML-Dokumenten werden auf die Felder einer Klasse in Java abgebildet, wobei je nach Art des Attributes primitive Datentypen wie `int` und `byte` verwendet werden oder Objekte wie Date und String. Jedes Element eines XML-Dokuments wird auf Java-Seite als eigene Klasse dargestellt.

Diese Abbildung kann aber auch automatisiert werden. Hierzu soll die „Java Architecture for XML Binding“ (JAXB [2]) einmal in groben Zügen vorgestellt werden. Ausgangspunkt ist ein XML-Schema und ein Bindungsschema. Das XML-Schema beschreibt die Struktur der XML-Datei. Das Bindungschema beschreibt die Abbildung der einzelnen Elemente und Attribute des XML-Schemas auf Javaklassen und Javaattribute. Wichtig ist, dass nur Elemente und Attribute in das Bindungsschema aufgenommen werden müssen, die vom oben beschriebenen Standardmapping abweichen. Beide Schemata werden dann von einem Übersetzer dazu verwendet, die den Schemata entsprechenden Javaklassen zu erstellen. Eine erstellte Javaklasse besitzt drei wichtige Methoden.

- `marshall()`: Diese Methode wandelt das Objekt und alle von diesem Objekt referenzierten Objekte in ein XML-Dokument um.

- `unmarshall()`: Diese statische Methode bekommt als Parameter einen Stream übergeben. Aus diesem Stream wird der Objektgraph wieder aus dem XML-Dokument erstellt.

- `validate()`: Auf Seite des XML-Dokuments sorgt das XML-Schema für ein gültiges Dokument. Da ein Objektgraph wieder in ein gültiges Dokument transferiert werden soll, muss die Gültigkeit vorher mit der Methode validate sichergestellt werden.

## 2.2 Datenmodelle

Neben der Abbildung auf Java soll jetzt das Augenmerk auf die Zieldatenmodelle *Hierarchisches Datenmodell* (HDM) und *Objektorientiertes Datenmodell* (OODM) gerichtet werden. Hierzu ist die bisherige Literatur etwas spärlich, was insbesondere darauf zurückzuführen ist, dass XML hauptsächlich eine industriegetriebene Entwicklung ist. Mertz [3] vergleicht das objektorientierte und hierarchisches Datenmodell in Bezug auf XML-Daten miteinander, wobei die wichtigsten Punkte hier aufgeführt werden.

## 2.2.1 Hierarchisches Datenmodell

Das hierarchische Datenmodell (HDM) [20] stellt im Wesentlichen einen Baum dar, bei dem jeder Knoten Daten und Unterknoten enthält. Diese Unterknoten sind von ein- und demselben Typ und können wiederum Elternknoten für weitere Unterknoten sein. Jeder Knoten außer dem Wurzelknoten des Baumes besitzt einen Elternknoten. Zwischen den einzelnen Knoten existieren sog. *Links*, die die Eltern Kind-Beziehung herstellen. In einer relationalen Datenbank würde dies durch Fremdschlüssel modelliert werden. Hierdurch kann dann via navigierendem Zugriff vom Elternknoten ausgehend jeder Kindknoten erreicht werden.

Im Folgenden soll ein kleines Beispiel die Stärken und Schwächen des HDM aufzeigen. Wir haben folgende einfache Hierarchie: (Buch-)Warengruppen->Autoren->geschriebene Bücher->Herausgeber. Möchte man in Erfahrung bringen, wer der Herausgeber von "An Introduction to Database Systems" ist, so könnte eine Anfrage im Pseudocode etwa so aussehen:

Programming/C.J.Date/An Introduction to Database Systems/Publisher?

Durch die stark vorgegebene Struktur des Datenmodells ist es für das Datenbankverwaltungssystem recht einfach, Zugriffe zu optimieren. Jedes Datum ist über einen genau bestimmten Weg zu erreichen. Der eben beschriebene Zugriff wäre somit recht effizient durchführbar, wobei es aber auch Anfragen geben kann, für die das HDM nicht besonders gut geeignet ist. Möchte man nun den Autor des Buches und die Warengruppe in Erfahrung bringen, so sähe eine Anfrage wie folgt aus:

Category?/Author?/An Introduction to Database Systems

Eine solche Anfrage ist nur mit großem Aufwand durchführbar, da hier unter Umständen der komplette Baum durchlaufen werden muss. Das Problem verschärft sich zunehmend, wenn nicht nur einfache 1:n oder n:1 Beziehungen, sondern n:m Beziehungen verwendet werden, welche von einem hierarchischen Datenbanksystem nicht adäquat abgebildet werden können. Dies hat mitunter zu der starken Verbreitung von relationalen Datenbanksystemen geführt, was aber nicht Gegenstand dieser Arbeit ist.

## 2.2.2 Objektorientiertes Datenmodell

Was ein objektorientiertes Datenmodell (OODM) bzw. objektorientierte Datenbanksystem (OODBMS) ist oder nicht ist, soll hier nicht diskutiert werden. Es sollen nur die wesentlichen Unterschiede des OODM zum HDM dargestellt werden.

Das OODM ist recht eng verwandt mit dem hierarchischen Datenmodell. Wichtige Unterscheidungsmerkmale sind:

- Im OODM wird aus dem Baum des HDM ein gerichteter Graph. Dies bedeutet für den Datenzugriff, dass ein Knoten bzw. Objekt über verschiedene Wege erreichbar ist.

- Weiterhin besitzt ein Objekt Attribute, die entweder primitive Datentypen oder komplexe Datentypen wie andere Objekte darstellen können. Hierdurch werden auf die Konstrukte Aggregation von Objekten und Komposition von Objekten möglich.

- Die Unterknoten eines Knotens müssen nicht denselben Typ haben. Hiermit wird die für das OODM wichtige Polymorphie abgedeckt.

- Ein Knoten bzw. ein Objekt besitzt ein Verhalten in Form von Methoden. Verhalten kann an andere Objekte weitervererbt werden. Eine Unterklasse erbt von einer Superklasse alle Attribute und Methoden. Man unterscheidet Einfach- und Mehrfachvererbung.

- Der Status und das Verhalten eines Objekte kann gekapselt werden, wobei eine Kommunikation durch Nachrichtenaustausch stattfindet.

## 2.2.3 Vergleich: XML und HDM/OODM

Vergleicht man nun XML mit dem OODM und dem HDM, so fällt auf, dass XML weder komplett mit dem einen noch mit dem anderen Modell erklärt werden kann.

Ein XML-Dokument besitzt eine Baumstruktur, was es somit eng mit HDM verwandt macht. Nimmt man nun einige Erweiterungen wie IDREF oder XLink [4] hinzu, so entspricht XML eher dem OODM, da so beliebige Verweise möglich werden.

Betrachtet man die Elemente und Subelemente eines XML-Dokuments, so dürfen diese ohne Schema beliebig benannt und angeordnet sein. Ein solches Dokument hätte in diesem Zusammenhang weder mit dem OODM noch mit dem HDM etwas zu tun. Nimmt man nun generell Schemata hinzu, so gewinnen XML-Dokumente, je nach verwendetem Schema, eine stärkere Struktur und Typisierung. In diesem Fall ist das OODM im Vorteil, da dieses Modell auch mit einer etwas schwächeren Struktur zurechtkommt, was insbesondere auf Subclassing und Subtyping zurückzuführen ist. Hierzu sei die Bedeutung von Subclassing und Subtyping einmal an einem Beispiel erklärt. Wir haben die Klassen Rechteck und Quadrat, wobei Quadrat von Rechteck erbt. Subtyping bedeutet in diesem Fall, dass an jedem Vorkommen eines Rechteckes auch ein Quadrat stehen darf. Subclassing besagt, dass jede Methode von Rechteck auch in Quadrat enthalten ist. Weitergehende Informationen sind bei Ernst [5] zu finden.

Dass bei XML-Dokumenten Begriffe wie das Geheimnisprinzip nicht vorhanden sind, liegt in der Natur der Sache, da XML hauptsächlich als Datenaustauschformat genutzt wird.

Zum gegenwärtigen Zeitpunkt können XML-Dokumente auch keinerlei Verhalten in Form von Methoden besitzen. Dies könnte sich aber in Zukunft mit der Entwicklung und Standardisierung von XML-Programmiersprachen ändern. Einige interessante Entwicklungen auf diesem Gebiet sind XDuce [6] oder XMLambda [7].

# 3 Warum ist XML-Databinding sinnvoll?

Im folgende Abschnitt soll auf die Vorteile von XML-Databinding (im besonderen JAXB) im Hinblick auf bestehende Verfahren wie beispielsweise SAX eingegangen werden. Die Grenzen von XML-Databinding werden dann in der Zusammenfassung näher betrachtet. Die Schlüsse, die in diesem Abschnitt über SAX gezogen werden, treffen ebenfalls in etwas abgeschwächter Form auf DOM zu.

Angenommen, ein XML-Dokument, welches beispielsweise das Ergebnis eines Abfrage eines Webservices einer Filmdatenbank beinhaltet, soll mit Java weiterbearbeitet werden. Der Betreiber der Filmdatenbank hat dazu den DTD zur Verfügung gestellt, der das genaue Schema des XML-Dokumentes enthält. Für eine solche Aufgabe ist JAXB ausgezeichnet geeignet.

- Da das Schema für das XML-Dokument vorhanden ist, können mit Hilfe eines Codegenerators alle benötigten Javaklassen automatisch erzeugt werden, die benötigt werden, um alle Informationen, die in dem XML-Dokument enthalten sind, aufzunehmen.

- Die erzeugten Klassen besitzen alle benötigte Funktionalität, um ein XML-Dokument einzulesen oder den aktuellen Objektgraphen wieder als XML-Dokument auszugeben.

- Die von den erzeugten Klassen zur Verfügung gestellte Schnittstelle, bestehend aus den Methoden *marshal*, *unmarshal* und *validate*, ist minimal, einfach zu verstehen und kann auch von Nicht-XML-Spezialisten ohne Probleme benutzt werden. Kenntnisse von SAX oder DOM sind hier nicht erforderlich.

- Von der Laufzeit ergeben sich gegenüber SAX nicht unbedingt Nachteile, da intern von den erzeugten Klassen ebenfalls SAX eingesetzt wird.

- Verglichen mit einer reinen SAX Implementierung werden deutlich weniger Codezeilen benötigt, da nur das Bindungsschema (siehe Kapitel 4.2) benötigt wird, welches im einfachen Falle leer ist. Eine SAX-Implementierung enthält hauptsächlich die Callback-Methoden *startElement* und *endElement*, die den Großteil der Leselogik enthalten. Bei größeren XML-Dokumenten können diese Methoden unter Umständen sehr unübersichtlich werden, da beispielsweise oft mit globalen Variablen gearbeitet wird, um die geparsten Daten festzuhalten. Zusätzlich wird bei komplexen Dokumenten oft ein Automat benötigt, der die aktuelle Parseposition im Dokument als Zustand enthält. Für die Erstellung einer solchen SAX-Implementierung werden schon etwas tiefgreifendere Kenntnisse über SAX und XML benötigt.

- Jeder manuell erzeugte Code durchläuft den kompletten Softwareentwicklungsprozess, ob dies nun geplant war oder nicht. So muss Zeit für Design, Implementierung, Tests, Reviews, (Code-)Verwaltung und Wartung aufgewendet werden, was sich natürlich auch in den Kosten eines Projektes niederschlägt. Ein mit JAXB erstellte Programmmodul benötigt vermutlich weniger Zeit beim Design, da durch das höhere

Abstraktionsniveau, das JAXB im Gegensatz zu SAX mitbringt, Probleme einfacher verstanden werden können. Die Implementierung gestaltet sich recht einfach, da nur das Bindungsschema angepasst werden muss. Den Rest erledigt dann ein Codegenerator. Tests oder Codereviews werden bei JAXB nicht benötigt, wenn man einen korrekt implementierten Codegenerator unterstellt. Da die Wartung ebenfalls wieder einen kleinen in sich geschlossenen Softwareentwicklungsprozess mit Design, Implementierung und Tests darstellt, lässt sich alles bis jetzt Gesagte auch auf die Wartung übertragen.

- JAXB kann man durchaus als einen kommenden Standard ansehen, um XML-Dokumente in einen Objektgraphen umzuwandeln. Eine SAX-Implementierung entspricht mit Sicherheit keinem Standard, was sich zwar kurzfristig nicht negativ auswirken muss, aber langfristig Nachteile mit sich bringt. So kann man mit Sicherheit davon ausgehen, dass sich in 10 Jahren Entwickler noch mit JAXB auskennen, was man dann von der SAX-Implementierung nicht mehr unbedingt behaupten kann.

- JAXB ist in bezug auf XML-Databinding sicher einfacher zu erlernen als SAX. Auch die Fehlersuche gestaltet sich bei JAXB einfacher.

# 4 XML-Databinding in der Praxis

Zu Beginn werden wir eine Standortbestimmung für die einzelnen XML-APIs durchführen, da man sonst durch die Vielfältigkeit dieser APIs den Überblick verlieren kann. Dieser Überblick ist in Abb. 3 gegeben. Als unterste Schicht zum Parsen von XML-Dokumenten ist dort SAX aufgeführt. Zusammen mit dem darauf aufbauenden DOM bilden diese beiden APIs JAXP. Als nächster Abstraktionsgrad ist in Abb. 3 die Java API for XML-Data Binding dargestellt, die Dienste von JAXP in Anspruch nimmt. Auf Basis von JAXB und JAXP können dann weitergehende Dienste wie beispielsweise Webservices realisiert werden.

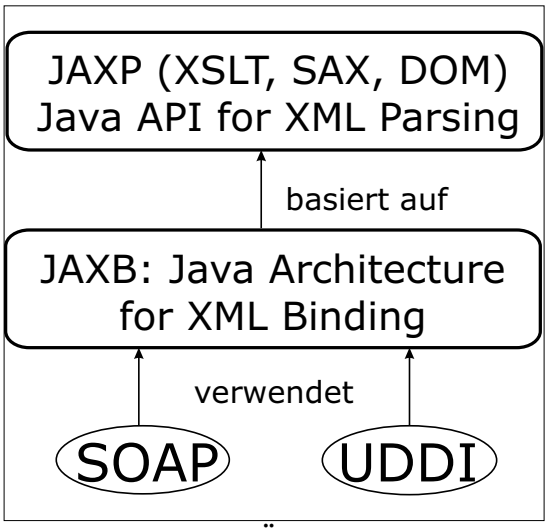


**Abb. 3 Übersicht**

## 4.1 Die Bedeutung des Schemas

Ausgangspunkt für XML-Databinding ist das Schema. Schemaloses XML-Databinding ist nicht sinnvoll oder entspräche dann im Wesentlichen dem Funktionsumfang der darunterliegenden Schichten wie z.B. SAX. In einem Schema befinden sich Festlegungen, die es erlauben, aus einer langen Zeichenkette, der XML-Datei, eine hierarchische typisierte Struktur zu erstellen. Je mächtiger bzw. je stärker typisiert ein Schema ist, desto besser kann eine Abbildung auf einen Java Objektgraphen durchgeführt werden.

Betrachtet man als Schema einmal DTDs, so kennen diese nur Zeichenketten als primitive Typen. Nimmt man als Vergleich ein XML-Schema [1], so kennt dieses eine ganze Reihe an einfachen Typen wie z.B. Zahlen, Daten oder Zeiträume. Dies vereinfacht die Abbildung von XML->Java ganz erheblich, was auch der Grund dafür ist, dass es bis jetzt noch keine endgültige Version von JAXB gibt. Es war nämlich zuerst geplant, JAXB nur mit Unterstützung für DTDs auszustatten, wobei diese Entscheidung zugunsten von XML-Schema [1] revidiert wurde.

## 4.2 Databinding am Beispiel von JAXB

Im Folgenden soll an Hand des bereits oben vorgestellten Beispiels `movies.dtd` der komplette JAXB Entwicklungsprozess durchlaufen werden. Dazu verwenden wir ein minimales Bindungsschema, das in Abb. 4 dargestellt ist. In diesem Schema definieren wir, dass alle automatisch erstellten Klassen zu dem Javapackage dazugehören: `javajaxb.generated.movies` . Weiterhin wird angegeben, dass Movies die Klasse ist, von der an alle anderen Klassen erzeugt werden, was durch den Zusatz `root=true` festgelegt ist. Alle anderen Klassen, für die hier keine Einstellungen vorgenommen wurden, werden nach dem in Kapitel 2 angegebenen Standardmapping umgewandelt. Ruft man nun

```
<?xml version="1.0"?>
<xml-java-binding-schema version="1.0-ea">
  <options package="javajaxb.generated.movies"/>
  <element name="movies" type="class" root="true"/>
</xml-java-binding-schema>
```
**Abb. 4 : Bindungsschema movies.xjs**

den Codegenerator mit dem DTD und dem Bindungsschema als Parameter auf, so werden die benötigten Klassen erzeugt.

```
Aufruf: xjc movies.dtd movies.xjs
```

```
package javajaxb.generated.movies;
...
public class Movie extends MarshallableObject
                   implements Element {
  public String getTitle() {...}
  public void setTitle(String _Title) {...}
  public Cast getCast() {...}
  public void setCast(Cast _Cast) {...}
  public String getDirector() {...}
  public void setDirector(String _Director) {...}

  public List getProducer() {...}
  public void deleteProducer() {...}
  public void emptyProducer() {...}

  public void validateThis()
    throws LocalValidationException {...}
  public void validate(Validator v)
    throws StructureValidationException {...}

  public void marshal(OutputStream out)
    throws IOException {...}

  public static Movie unmarshal(InputStream in)
    throws UnmarshalException {...}

  public boolean equals(Object ob) {...}
  public int hashCode() {...}
  public String toString() {...}
}
```

**Abb. 5: Generierte Datei Movie.java**

Betrachtet man nun die generierte Javaquelldatei in Abb. 5, so sind dort alle Elemente zu finden, die auch in DTD definiert waren. So hat ein Film einen `title` und einen `director`, die über get/setMethoden, welche jeweils einen String als Parameter erhalten, verändert werden können. Ein Film darf nach DTD mehrere Produzenten haben, wobei diese von Typ `#PCDATA` also String sind. Dies wird hier über eine Liste realisiert, die mit den Methoden `getProducer`, `deleteProducer` und `emptyProducer` bearbeitet werden kann. Der Unterschied zwischen `emptyProducer` und `deleteProducer` besteht darin, dass bei `emptyProducer` alle Produzenten gelöscht werden, wohingegen bei `deleteProducer` die Assoziation zwischen Film und Produzenten aufgehoben wird und die Produzenten somit noch für einen anderen Film verwendet werden können. Die Methoden `validateThis` überprüft, ob sich die lokalen Attribute des Objekt in einem Zustand befindet, der bezogen auf das ursprüngliche Schema gültig ist. In diesem Falle wird in der Methoden geprüft, ob Titel und Name des Films vorhanden bzw. nicht null sind. Die Methode `validate` erfüllt im Prinzip dieselbe Aufgabe wie `validateLocal` bis auf den Unterschied, dass in dieser Methode alle von diesem Objekt erreichbaren Objekte überprüft werden. Die Methode `marshal` ist dafür zuständig, den aktuellen Objektzustand zurück in ein XML-Dokument zu überführen. Als Parameter wird ein einfacher `OutputStream` übergeben. Es ist wichtig, vor dem Aufruf von `marshal` den Objektgraphen mit der Methode `validate` zu überprüfen, da dies nicht von `marshal` erledigt wird bzw. das Ergebnis von `marshal` dann undefiniert ist (z.B. Exception). Die statische Methode `unmarshal`, welche in der Implementierung des größten Platz einnimmt, liest ein XML-Dokument ein und wandelt es in einen Objektgraph um, der als Rückgabewert zurückgegeben wird. Im Folgenden ein Beispiel für die Anwendung der Methoden `marshal` und `unmarshal`:

```
// Filme einlesen
InputStream in = new FileInputStream (”kinofilme.xml”);
Movies movies = Movies.unmarshal (in);
// Filme bearbeiten
....
// Filme in XML-Datei speichern
OutputStream out = new FileOutputStream (new File(”kinofilme_neu.xml”));
Movies.validate();  // validate wurde von MarshallableRootElement vererbt
Movies.marshal(out);
```

Zusätzlich zu den bereits oben genannten Methoden werden die Methoden `hashCode` und `equals` mit einer sinnvollen Implementierung überschrieben.

## 4.3 Das Bindungsschema

Nachdem nun dargelegt wurde, wie Klassen erzeugt (siehe auch Abb. 6) und die erzeugten Klassen benutzt werden, soll einmal näher auf das Bindungsschema eingegangen werden. Hierzu wird im Wesentlichen der DTD des Bindungsschemas vorgestellt und die einzelnen Elemente näher erläutert. Hier muss noch darauf hingewiesen werden, dass es sich bei dem hier vorgestellten DTD um einen DTD aus dem Early Access Release von JAXB handelt und die Wahrscheinlichkeit sehr groß ist, dass sich alles in Bezug auf das final Release noch verändern wird. Nach Aussage von Sun sind auch die Schnittstellen noch nicht als stabil anzusehen.

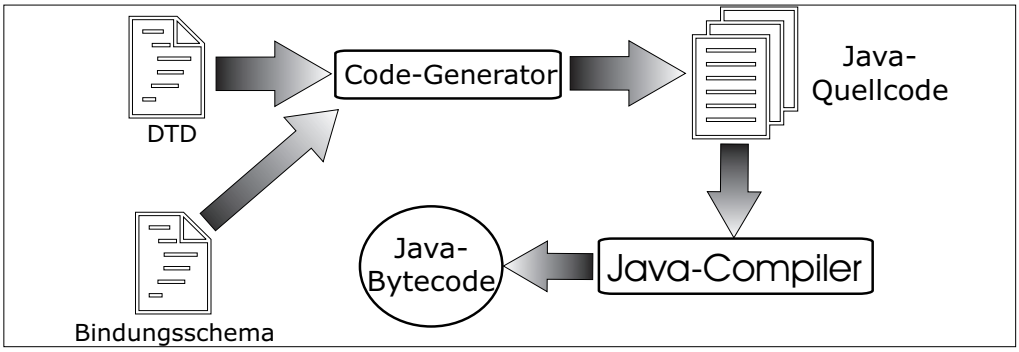



**Abb. 6 Databindingprozess**

```
<!ELEMENT      xml-java-binding-schema
       (options?, (element | interface | enumeration | conversion)* ) >

<!ELEMENT      options EMPTY>
<!ATTLIST      options
               package                  NMTOKEN        #IMPLIED
               default-reference-collection-type       (array | list) "list"
               property-get-set-prefixes               (true | false) "true"
               marshallable                            (true | false) "true"
               unmarshallable                          (true | false) "true" >
```

Package sagt, zu welchem Package die erzeugten Klassendateien gehören sollen. Hätten wir die Option `default-reference-collection-type` auf `array` gesetzt, so würden folgende zwei Methoden für den Produzenten erzeugt werden:

```
public void setProducer (String[] producer);
public String[] getProducer();
```

Die Verwendung der nächsten Option (property-get-set-prefixes) erscheint eher verwirrend, da dann auf get/set Prefixe verzichtet werden muss, womit das obige Beispiel wie folgt aussähe:

```
public void producer (String[] producer);
public String[] producer();
```

Mit `marshallable` und `unmarshallable` kann eingestellt werden, ob die erzeugten Klassen entsprechende Methoden besitzen. So kann es beispielsweise sinnvoll sein, den aktuellen Objektzustand für Debuggingzwecke in eine XML-Datei zu speichern. Eine solche Klasse bräuchte keine Unmarshal-Methode.

Betrachten wir nun `element` genauer:

```
<!ATTLIST element
       name    ID                    #REQUIRED
       type    (value | class)       #REQUIRED
       convert NMTOKEN               #IMPLIED
       class   NMTOKEN               #IMPLIED
       root    (true | false)        #IMPLIED>
```

Hierfür gibt es eine Reihe an Anwendungen:

```
1) <element name="movie-data"  type="class" class="Movies" root=true/>
2) <element name="actor"  type="value"/>
3) <element name="copyrightYear" type="value" convert="int"/>
```

Im ersten Fall ändern wir den Klassenname von "movie-data" in "Movies" um, da der ursprüngliche Name sonst keine gültige Javaklasse darstellen würde. Für den zweiten Fall schaffen wir uns so eine Erleichterung bei der Handhabung der erstellten Objekte. So können wir nun schreiben

```
Cast.getActor().add("Sean Astin");
```

Ursprünglich hätte man dies etwas ausführlicher schreiben müssen:

```
Actor actor = new Actor();
actor.setContent("Sean Austin");
Cast.getActor().add(actor);
```

Im dritten Fall spezifizieren wir, dass das Jahr auf Javaseite als `Integer` dargestellt werden soll. JAXB kümmert sich in diesem Falle dann um die notwendige Konvertierung.

JAXB bietet auch Unterstützung für Aufzählungen. Angenommen, wir möchten für unseren Film ein Genre definieren, dann könnte es im Bindungsschema wie folgt dargestellt werden:

```
movies.dtd
<!-- Genre als Attribut zum Film speichern-->
<!ATTLIST movie
        genre (sci-fi | horror | comedy | drama | mystery |children) 'drama'>

movies.xjs (Bindungsschema)
<enumeration name="Genre"
        members="sci-fi horror comedy drama mystery children"/>
<element name="movie" type="class">
 <attribute name="genre" convert="Genre"/>
</element>
```

Durch diese Definition wird von JAXB eine Klasse Genre erstellt, die nach dem in Java üblichen Enumeration Pattern erstellt ist. Die Klasse `Movie` enthält dann Methoden mit der Signatur

```
public void setGenre(Genre genre);
public Genre getGenre();
```

Weiterhin wird dem Codegenerator durch `convert="Genre"` mitgeteilt, dass er die Klasse `Genre` zur Konvertierung nutzen soll, wobei wir bei dem Thema Konvertierungen wären.

Neben Aufzählungen und den primitiven Datentypen wie beispielsweise `String`, `Integer` oder `Float` kennt JAXB keine weiteren Datentypen. Angenommen, man möchte das Veröffentlichungsdatum eines Films einlesen, dann müsste zu diesem Zweck eine Konvertierungsklasse erstellt werden, die zwei Methoden enthält, die beide statisch sein müssen. Eine der Methoden erhält eine Zeichenkette als Parameter und gibt das gewünschte Objekt zurück. Die andere Methode verfährt genau andersherum und wandelt ein Objekt, in diesem Fall ein Datum, in eine Zeichenkette um. Im Bindungsschema müssen dann nur noch die geschriebenen Methoden angegeben werden. Im Folgenden sei dies an Hand von Code dargestellt:

```
// Konvertierungsklasse in Java
public class DateConversion {
  private static SimpelDateFormat df = new SimpleDateFormat ("yyyy-mm-dd");
  public static Date parseDate (String d) {
    try {
      return df.parse(d);
    }
    catch (Exception e) { return new Date(); }
  }
  public static String printDate(Date d) {
     return df.format(d);
  }
}

movies.xjs (Bindungsschema)
<conversion name="Date"
               type="java.util.Date"
               parse="DateConversion.parseDate"
               print="DateConversion.printDate"/>
<element name="movie" type="class">
 <attribute name="genre" convert="Genre"/>
 <attribute name="releaseYear" convert="Date"/>
</element>
```

Neben diesen vorgestellten Möglichkeiten in JAXB gibt es noch eine Reihe anderer Möglichkeiten wie z.B. Interfaces, Konstruktoren, Sequenzen oder Elementreferenzen. Diese können hier aus Platzgründen nicht behandelt werden. Der interessierte Leser sei hier auf die JAXB Dokumentation [8] oder auf [9] verwiesen, woraus auch die hier vorgestellten Beispiele stammen.

## 4.4 Andere Produkte

Bevor die Produkte genauer betrachtet werden, soll kurz dargelegt werden, warum es sinnvoll ist, sich andere Produkte als JAXB anzusehen. Ein Grund ist sicher, dass JAXB im Augenblick eigentlich noch gar nicht offiziell erhältlich ist. Es existiert zwar seit Neuestem JAXB 1.0 beta aber laut Aussage von Sun können sich die Schnittstellen bis zur endgültigen Version noch ändern. Weiterhin ist die gesamte Funktionalität noch nicht ganz fehlerfrei und es wird, wie bei jeder neuen Technologie, noch eine Weile dauern, bis eine gewisse Stabilität eingekehrt ist. Als weiterer wichtiger Punkt wäre anzuführen, dass JAXB zwar die wichtigsten Schemata unterstützt, aber für neuere, eventuell reichhaltigere oder einfachere Schemata wie z.B. RelaxNG keine Unterstützung bietet. Da JAXB bereits vor diesem Abschnitt ausführlich behandelt wurde, soll hier nicht näher darauf eingegangen werden, wobei hervorgehoben werden sollte, dass es sich bei JAXB um das in Zukunft vermutlich wichtigste Produkt handelt.

Anmerkung zu Relax NG [10]: Relax NG wurde entwickelt, da bestehende Schemata wie DTD nicht leistungsfähig genug und XML-Schema [1] recht kompliziert waren. Relax NG bietet deutlich mehr Möglichkeiten als DTDs ohne dabei so Komplex wie XML-Schmea zu sein. Das ISO-Komitee ISO/IEC JTC 1/SC34 hat Relax NG inzwischen zu einem *Draft International Standard* erhoben [21].

### 4.4.1 Zeus

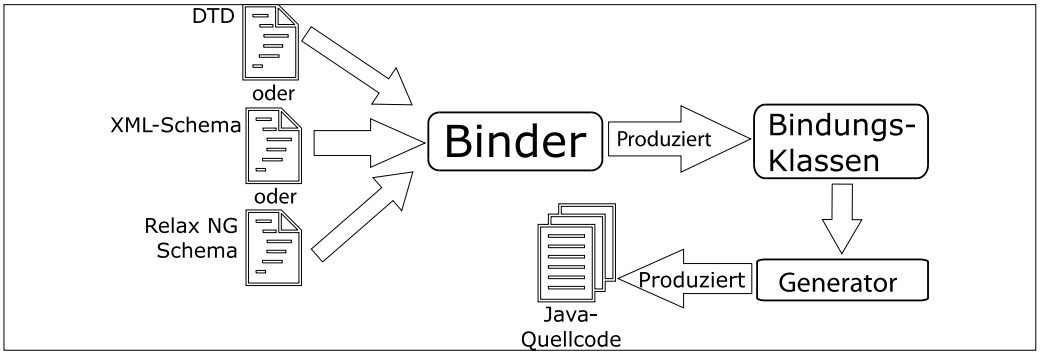



**Abb. 7 Databindingprozess in Zeus**

Verglichen mit JAXB bietet Zeus [11] dieselbe Funktionalität mit den generierten Klassen an. Lediglich der Erstellungsprozess dieser Klassen wird nicht wie bei JAXB in einem Schritt vorgenommen, sondern es sind drei Schritte notwendig. Hierdurch wird der gesamte Prozess (in Abb. 7 dargestellt) zwar etwas schwerer verständlich, aber man gewinnt einiges an Flexibilität hinzu.

Im ersten Schritt wird das Schema, in dem sich die verschiedenen Beschränkungen und Typdefinitionen der später zu erzeugenden Klassen befinden, mit Hilfe eines Binders in sog. Bindungsklassen umgewandelt. Ein solcher Binder muss für jeden zu bearbeitenden Schematyp wie z.B. DTD, XML-Schema oder RelaxNG geschrieben werden. Im Moment existieren Binder für DTDs und XML-Schema. Sinn dieser Architektur ist es, dass später recht einfach zusätzliche Schematypen hinzugefügt werden können, ohne dabei größere Änderungen an Zeus vornehmen zu müssen. Werden in unterschiedlichen Schematypen wie z.B. DTD und XML-Schema dieselben Einschränkungen auf das zu generierende Klassenmodell gemacht, so sind auch die erzeugten Bindungsklassen gleich. Den Bindungsklassen kann also nicht mehr angesehen werden, ob sie aus einem DTD oder einem XML-Schema erzeugt wurden.

Der zweite Schritt wird Transformation genannt. In diesem Schritt werden die Bindungen gefiltert, konfiguiert oder konvertiert. Sinn dieses Schrittes ist es beispielsweise, eine Art PlugIn-System zur Verfügung zu stellen, mit dem dann Schemata anderer Produkte wie z.B. Castor (siehe Kapitel 4.4.2 oder JAXB (hier Bindungsschema) verwendet werden können.

Im dritten Schritt werden die eigentlichen Quelldateien erstellt, welches Aufgabe des Generators ist. Ziel dieses Ansatzes ist es, später vielleicht einmal Quelldateien für andere Sprachen wie beispielsweise C#, C oder C++ zu erstellen, wobei dann nur der Generator ausgetauscht werden muss. Im Augenblick existiert nur ein Generator für Java.

Das Arbeiten mit den erzeugten Klassen unterscheidet sich bis auf kleinere Besonderheiten nicht wesentlich von JAXB. Die größte Abweichung liegt sicher bei dem Unmarshalling vor, da Zeus zu jeder erzeugten Klasse

ein Interface anbietet. Anwendungen verwenden nur das Interface, was aber Probleme bereitet, da die für das Unmarshalling benötigte statische Methode nicht in einem Interface enthalten sein kann. Da aber auf die Implementierung der Klasse nicht zurückgegriffen werden soll, existiert dann für eine Klasse `MyClass` eine zweite Klasse `MyClassUnmarshaller`, mir der das Unmarshalling durchgeführt werden kann.

Ein Vorteil von Zeus ist, dass es in sich geschlossen ist. Dies heißt, dass die erzeugten Klassen keine weiteren Laufzeitbibliotheken außer einem SAX-Parser benötigen.

### 4.4.2 Castor

Das nächste Produkt, was wir uns anschauen werden, ist Castor [12] von der Exolab group. Da die DataBinding Funktionalität bereits entwickelt wurde, als man bei Sun an JAXB noch nicht einmal dachte, weicht Castor in einigen Punkten etwas von JAXB ab. Anstatt DTDs unterstützt Castor nur XML-Schema, wobei diese Unterstützung inzwischen recht ausgereift ist.

```
<class
  name="generated.hr.Office">
  <field name="Id"
         type="integer">
    <bind-xml name="office-id" node="attribute"/>
  </field>
  <field name="Address"
         type="generated.hr.Address">
    <bind-xml name="office-address" node="element"/>
  </field>
```

**Castor Mappingdatei (Ausschnitt)**

Ähnlich wie bei den anderen beiden vorgestellten Produkten gibt es hier ebenfalls einen Codegenerator, der mit Hilfe eines XML-Schemas für jedes XML-Objekt eine Javaklasse und einen dazugehörigen XML-Klassendeskriptor erstellt. Die erstellt Javaklasse enthält die von dem Anwendungsentwickler verwendeten get- und set- Methoden. Im Klassendeskriptor, welcher ebenfalls als Javaklasse vorliegt, sind Informationen wie zur Validierung, XML-Namen oder (XML-)Namensbereichen gespeichert. Ein Aufruf von marshal und unmarshal-Methoden unterscheidet sich nicht besonders von JAXB, wobei bei jedem Marshalling und Unmarshalling die Informationen des Klassendeskriptors benutzt werden.

Was bei JAXB das Bindungsschema ist, stellen bei Castor die sog. Mappingfiles dar. In ihnen ist es möglich, Namen von XML nach Java zu transformieren und umzubenennen. Im Folgenden ein Beispiel, wie ein solches Mapping angewendet wird:

```
Mapping mapping = new Mapping ();
mapping.loadMapping (”mappingfile.xml”);
// Marshalling durchführen
Marshaller marshaller = new Marshaller (new FileWrite ( new File (”output.xml”)));
marshaller.setMapping(mapping);
marshaller.marshal(office);   // office wurde vorher instantiiert und mit Daten versehen
```

Neben den hier dargestellen Möglichkeiten gibt es bei Castor auch Mappings auf relationale Datenbanken (Castor JDO) und Verzeichnisdienste wie LDAP. Hierbei ist zu beachten, dass es sich bei Castor JDO um keine dem JDO[1] Standard konforme Implementierung handelt. Leider erlaubt der zu Verfügung stehende Platz keine genauere Behandlung. Es sei hier aber darauf hingewiesen, dass es mit Castor recht einfach ist, XML-Dateien in einer relationalen Datenbank zu speichern.

### 4.4.3 Quick

Nachdem wir nun verschiedene XML-Databinding Produkte betrachten haben, kommen wir zu einem Produkt, was sich deutlich von den bisher genannten unterscheidet. Quick [19] ist ein Werkzeug, das hauptsächlich die Konvertierung von XML-Daten auf bestehende Javaklassen erleichtern soll. Hierzu müssen eine ganze Reihe an Konvertierungen von Dateiformaten vorgenommen werden, die wir jetzt näher betrachten werden.

Ausgangspunkt für alle Aktionen in Quick ist eine eigene Dokumentenbeschreibungssprache, genannt Quick Document Markup Language (QDML). Diese Sprache besitzt Fähigkeiten, ähnlich der von DTDs und XML-Schemata. Damit QDML keine Insellösung ist, existieren Konvertierungsprogramme, um aus QDML

1 JDO=Java Data Objects

```
<qjml root="po">
  <bean tag="po">
    <targetClass>PurchaseOrder</targetClass>
      <elements>
        <item coin="o" repeating="True">
          <property kind="list" name="orderList"/>
        </item>
      </elements>
  </bean>
  <bean tag="o">
    <targetClass>Order</targetClass>
    <attributes>
        <item coin="o.id">
          <property name="id"/>
        </item>
        <item coin="o.s_id">
          <property name="sku"/>
        </item>
     </attributes>
     <elements>
        <item coin="p_name">
           <property name="productName"/>
        </item>
        <item coin="m_name">
           <property name="manufacturerName"/>
        </item>
        <item coin="p_c">
           <property name="purchasePrice"/>
        </item>
        <item coin="sk">
           <property name="stock"/>
        </item>
     </elements>
  </bean>
  <text label="o.id" tag="id"
        validInherited="True" type="int" />
  <text label="o.s_id" tag="s_id"
        validInherited="True" type="PCDATA" />
  ....
```

**Abb. 9: QJML-Datei**

```
<qdml root="po">
  <bean tag="po">
    <elements>
       <item coin="o"
             repeating="true"/>
    </elements>
  </bean>
  <bean tag="o">
      <attributes>
          <item coin="o.id"/>
          <item coin="o.s_id"/>
      </attributes>
      <elements>
         <item coin="p_name"/>
         <item coin="m_name"/>
         <item coin="p_c"/>
         <item coin="sk"/>
      </elements>
  </bean>
  <text label="o.id" tag="id"/>
  <text label="o.s_id" tag="s_id"/>
  <text tag="p_name"/>
  <text tag="m_name"/>
  <text tag="p_c"/>
  .....
  .....
```

**Abb. 8: QDML-Datei**

```
<po>
  <o id="23" s_id="145-9876-90">
    <p_name>OfficeConnectHub</p_name>
    <m_name>3Com</m_name>
    <p_c>149.99</p_c>
    <sk oh="true" num="22" />
  </o>
  ....
  ....
</po>
```

**Abb. 10 productOrder.xml**

einen DTD zu erstellen und umgekehrt. Neben QDML gibt es eine weitere Beschreibungssprache, genannt Quick Java Markup Language (QJML). Was für JAXB ein Bindungsschema ist, stellt QJML in etwa für Quick dar. Für die Erstellung einer QJML-Datei existiert ein Compiler, der als Eingaben eine QDML-Datei und den Pfad zu den bereits existierenden Javaklassen erhält. Der Compiler versucht dann auf eine möglichst intelligente Art und Weise, eine Beziehung zwischen den einzelnen Javaklassen und deren Attributen und den in der QDML-Datei spezifizierten (XML-)Elementen herzustellen. Je nachdem, wie stark XML-Datei und Javaklassen voneinander abweichen, desto mehr muss von Hand nachbearbeitet werden. Besitzt man nun eine solche QJML-Datei, kann man vom Prinzip bereits Funktionen ähnlich marshalling und unmarshalling nutzen. Eine QJML-Datei hat nur den Nachteil, dass die in ihr enthaltene Information bei jedem (Un-)Marshallvorgang zur Laufzeit neu eingelesen und interpretiert werden muss. Zu diesem Zweck besitzt Quick eine weitere Beschreibungssprache, genannt Quick Internal Markup Languange (QIML). Eine QIML-Datei ist eine XML-Datei, die eine mit einem weiteren Compiler vorkompilierte Version einer QJML-Datei enthält. Sie eignet sich auf Grund ihrer Länge und Komplexität nicht für eine manuelle Nachbearbeitung. Eine solche Datei kann an Stelle einer QJML-Datei zur Laufzeit eingesetzt werden und so das Laufzeitproblem etwas mildern. Will man nun keinerlei zusätzlichen Aufwand zur Laufzeit, so bietet Quick die Möglichkeit an, eine QIML-Datei in eine Javaquelldatei umzuwandeln, die anschließend mit einem Javacompiler in Javabytecode übersetzt werden kann. Existieren zu Beginn des Vorgangs noch keine Javaklassen, so können diese aus einer QJML-Datei erzeugt werden.

Wir nehmen an, ein Unternehmen besitzt verschiedene XML-Dokumente, ähnlich wie in Abb. 10 dargestellt. Bei der Erstellung eines solchen Schemas könnte Platzeffizienz früher eine Rolle gespielt haben, wobei inzwischen genügend Festplattenplatz und Übertragungskapazität existieren um solche Dokumente problemlos zu verwalten. Es fällt also die Entscheidung, das etwas knapp gehaltene Format durch ein

ausführlicheres Format zu ersetzten. Da es sich hier um sehr viele Dokumente handelt, müssen die beiden Formate über längere Zeit gleichzeitig bearbeitet werden könne.

Mit JAXB wäre eine Lösung sehr schwierig, da es nicht möglich ist, zwei unterschiedliche DTDs auf ein und dieselbe Javaklasse abzubilden. Man könnte zwar die Attribute der Javaklasse mit den gewünschten Bezeichnungen erstellen, aber es wäre nur möglich, entweder das neue oder das alte Format zu unterstützen.

Die für dieses Beispiel aus dem DTD generierte QDML-Datei ist in Abb. 8 auszugsweise dargestellt. Aus dieser QDML-Datei ist dann eine QJML-Datei erstellt worden, wobei der dick gedruckte Text verändert wurde. So wurden die Klassennamen und Attribute der Klasse durch sinnvolle Bezeichner ersetzt. Betrachtet man das erste text label, so wäre dort im Original mit folgendem Code für id eine eigene Klasse erzeugt worden:

```
<text label="o.id" tag="id" validInherited="True">
    <targetClass>O_id</targetClass>
</text>
```

Damit in unserem Beispiel nun beide XML-Formate von Quick verarbeitet werden können, gäbe es die Möglichkeit, nun eine Kopie dieser QJML-Datei zu erzeugen und die kurzen Bezeichner durch die geplanten langen Bezeichner zu ersetzen.

In Java würde die Verwendung der erstellten Dateien so aussehen:

```
// PurchaseOrderSchema ist eine aus einer QIML-Datei erzeugte Javaklasse
QDoc purchaseOrderSchema = PurchaseOrderSchema.createSchema();

// Unmarshalling durchführen
QDoc purchaseOrderDoc = Quick.parse(purchaseOrderSchema,
                                    new File (”order.xml”).getAbsolutePath());
// Zugriff auf erzeugten Objektbaum
PurchaseOrder order = (PurchaseOrder) Quick.getRoot(purchaseOrderDoc);

// order bearbeiten
....

// Marshalling durchführen
Quick.express(purchaseOrderDoc, new File (”output.xml”).getAbsolutePath());
```

# 5 XML-Serialisierung in Java

Betrachtet man die bestehende Serialisierung in Java, die bereits seit der Version 1.0 dabei ist, so haben sich mit der Zeit einige Nachteile herauskristallisiert, die mit der neuen XML-Serialisierung größtenteils behoben wurden. Im Folgenden seien die wichtigsten Punkte aufgeführt, die zu der Entwicklung von der XML-Serialisierung, welche auch unter dem Namen *Long-Term Persistence for JavaBeans* [13] bekannt ist, geführt haben.

- Der bestehende Serialisierungsmechanismus hatte den Nachteil, dass er nicht standardisiert war. So waren die erzeugten Dateien von der verwendeten JVM abhängig, was dazu führte, dass die erzeugten Dateien mit JVMs anderer Hersteller nicht gelesen werden konnten. Hinzu kam, dass häufig Dateien, die mit einer JVM eines Herstellers erzeugt wurden, mit einer anderen Version der JVM desselben Herstelles gelesen werden konnten, da sich die Implementierung des Serialisierungsmechanismus geändert hatte. Diese Problematik wurde zusätzlich dadurch verschärft, dass im Laufe der Zeit die zugrundeliegende Klasse beispielsweise zusätzliche Attribute erhielt, welche dann die bisher gespeicherten serialisierten Dateien ungültig machten.

- Mit dem bestehenden Serialisierungsmechanismus wurden immer alle Attribute serialisiert, was dazu führte, dass die erzeugten Dateien oft unnötig aufgebläht wurden. Hier speichert der neue Serialisierungsmechanismus nur die Werte, die von den vorgegebenen Werten abweichen (vorgegebene Werte gibt es nur bei JavaBeans).

- Ein weiterer Vorteil des neuen Serialisierungsmechanismus ist die Fehlertoleranz, die durch die Verwendung von XML-Dateien als Persistenzmedium erreicht wird. Kleinere Fehler in XML-Dateien können mit jedem Texteditor behoben werden. Hierbei wäre es auch durchaus vorstellbar, dass beispielsweise bei einer Änderung eines Attributnamens in einer Javaklasse ein Konvertierprogramm die bisherig gespeicherten XML-Dateien auf den neuen Attributnamen umstellt (z.B. mit XSLT).

Betrachten wir nun im Folgenden die Anwendung in Java und die erzeugte XML-Datei:

```
    // Testframe erzeugen
    javax.swing.JFrame aFrame = new JFrame();
    aFrame.setBounds(new Rectangle (0, 0, 200, 200));
    aFrame.getContentPane().add(new JButton ("label"));
    aFrame.setVisible(true);

    // Frame in XML-Datei serialisieren
    XMLEncoder e = new XMLEncoder(
            new BufferedOutputStream(
                    new FileOutputStream("test.xml")));
    e.writeObject(aFrame);
    e.close();


    // Frame aus XML-Datei deserialisieren
    XMLDecoder d = new XMLDecoder(
            new BufferedInputStream(
                    new FileInputStream("test.xml")));
    JFrame anotherFrame = (JFrame)d.readObject();
    d.close();
```

```
<!--test.xml-->
<java version="1.0" class="java.beans.XMLDecoder">
 <object class="javax.swing.JFrame">
   <void property="name">
     <string>frame1</string>
   </void>
   <void property="bounds">
     <object class="java.awt.Rectangle">
       <int>0</int>
       <int>0</int>
       <int>200</int>
       <int>200</int>
     </object>
   </void>
   <void property="contentPane">
     <void method="add">
       <object class="javax.swing.JButton">
         <void property="label">
           <string>Hello</string>
         </void>
       </object>
     </void>
   </void>
   <void property="visible">
     <boolean>true</boolean>
   </void>
 </object>
 </java>
```

Der Aufbau dieser Datei ist eigentlich recht einfach. Jedes der verwendeten Tags (`object` und `void`) stellt Methodenaufrufe dar. Wird `object` verwendet, so ist das Ergebnis ein Ausdruck, der an das nächsthöhere Tag als Parameter weitergegeben wird. In der Beispieldatei ist `JButton` ein Parameter für das nächsthöhere Element `add`. Elemente von Typ `void` sind einfache Methodenaufrufe des nächsthöheren Objektes. Parameter sind die von diesem Tag umschlossenen Objekte oder primitive Datentypen (z.B. `int` bei `Rectangle`). Das Element `void` kann zwei verschiedene Attribute besitzen. Das Attribut `method` kennzeichnet einen einfachen Methodenaufruf. Speziell für JavaBeans wurde das Attribut `property` eingeführt, bei dem dann die entsprechenden set/get Methoden des JavaBeans verwendet werden.

# 6 Speicherungslösungen für XML-Daten

XML-Dokumente werden inzwischen in vielen unterschiedlichen Bereichen eingesetzt. Mit der Nutzung von XML-Dokumenten entsteht auch das Problem von deren Speicherung.

Hier einige Vorteile:

Für den Nutzer stellt sich eine XML-Datei als eine Ansammlung von Tags wie <begin> und </begin> sowie textueller Daten zwischen diesen Tags dar. Aus Sicht des Nutzers ist eine solche Datei einfach zu verstehen und er kann Änderungen an den Daten selbst durchführen. Parsergeneratoren lassen sich für eine solche Datei ebenfalls recht einfach erstellen.

Diesen Vorteilen stehen aber einige eklatante Nachteile entgegen:

- XML-Dokumente werden durch die sich ständig wiederholende Nutzung von Tags recht aufgebläht. So besteht ein Element immer mindestens aus einem einleitenden Tag und einem schließenden Tag. Sehr oft sind diese Strukturierungshilfen sehr viel größer als die eigentliche umschlossene Information.

- XML-Dokumente müssen, bevor sie von einer Applikation bearbeitet werden können, in eine für die Applikation verständliche interne Struktur umgewandelt werden (z.B. Aufbau von DOM-Bäumen).

- Deswegen ist es schwierig, sehr große XML-Dokumente zu bearbeiten. Zum einen dauern Aktionen wie Parsen sehr lange, zum anderen kann es sogar sein, dass Dokumente bearbeitet oder durchsucht werden müssen, deren Größe die Größe des Hauptspeichers oder die des virtuellen Speichers übersteigt.

Aufgrund dieser Nachteile wurden schon recht früh Speicherungslösungen wie XML-Datenbanken entwickelt. Im Folgenden soll ein Überblick über zwei Lösungen namens Persistent-DOM (PDOM) und Natix gegeben werden. Die Betrachtungen für PDOM sind eher auf die Architektur eines solchen Systems ausgerichtet, wohingegen bei Natix nur die verwendete Baumstruktur im Vordergrund steht.

## 6.1 PDOM

Persistent DOM (PDOM [14]) wurde 1998 an dem GMD-IPSI entwickelt. Ziel dieses Projektes war es, neben schemabehafteten auch schemalose XML-Dokumente zu persistieren. Der Zugriff auf die gespeicherten XML-Dokumente erfolgt über eine DOM Schnittstelle, was bedeutet, dass der Anwender im Idealfall gar nicht merkt, dass er mit einer PDOM-Implementierung arbeitet.

Dieses Teilkapitel inkl. aller Abbildungen gehen auf Huck et al [14] zurück.

### Architektur

In PDOM werden der Applikation zwei verschiedene APIs zur Verfügung gestellt. Die DOM API stellt dem W3C-Standard entsprechende Funktionen zur Verfügung und ermöglicht so eine recht schnelle Integration von PDOM in bestehende Anwendungen. Zusätzlich zu dieser API existiert noch eine besondere PDOM API, über die die darunter liegende Datenbank verwaltet wird. Hier sind Funktionen zum Transaktionsmanagement (z.B. Commit, Rollback) und Datenbankmanagement (z.B. Create, Drop) vorhanden. Beide APIs greifen auf den gleich noch näher erläuterten *Presistent Object Manager* zurück, der für Caching und das Transaktionsmanagement zuständig ist.

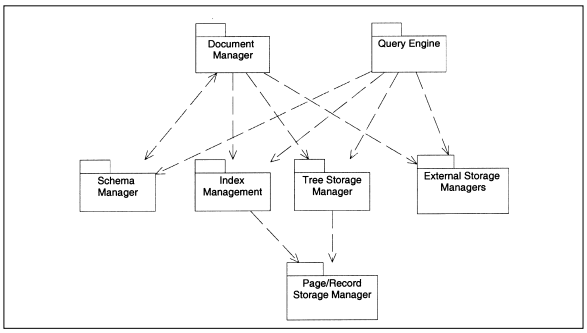



**Abb. 11 Architektur**

### Dateiorganisation

Der Dateiaufbau ist in Abbildung 12 dargestellt. Eine solche Datei besteht aus mehreren Knotenseiten, die jeweils 128 DOM-Objekte aufnehmen. Diese Knotenseiten haben neben der Beschränkung auf höchstens 128 DOM-Objekte eine variable Größe. Am Anfang der Datei befinden sich zwei Zeiger, von denen der eine auf ein Wörterbuch zeigt und der andere auf den sog. Knotenindex. Das Wörterbuch wird verwendet, um die

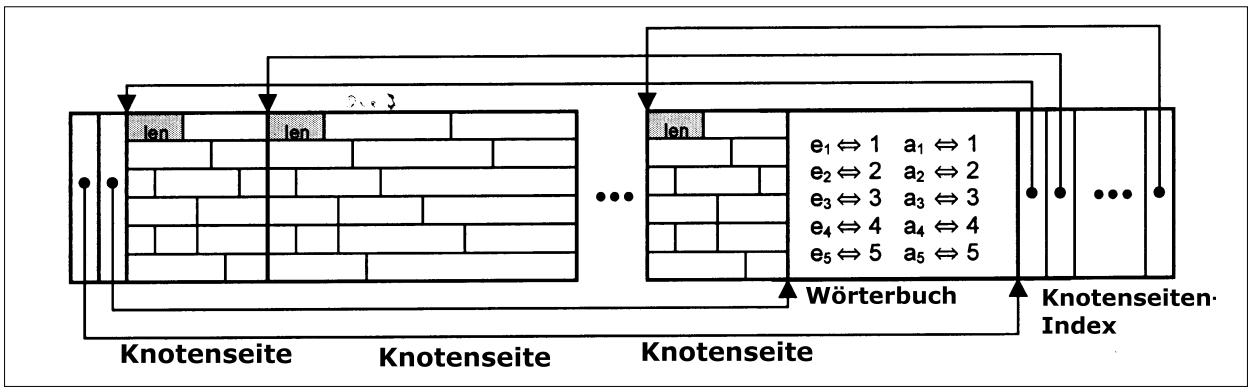



**Abb. 12 Dateiorganisation**

Elementnamen zu komprimieren. So wird jedem Elementnamen bei der Speicherung eine Zahl zugeordnet. Beim Retrieval kann so aus der Zahl wieder ein Elementname erstellt werden.

Um auf ein Objekt in einer Knotenseite zuzugreifen, bedarf es einer eindeutigen Adresse (OID). Diese OID wird nach folgender Gleichung berechnet: OID=SI*128+I. SI bezeichnet hier den Seitenindex. Die physikalische Adresse der Knotenseite ist über den Knotenseitenindex[Seitenindex] ermittelbar. I ist die Position des Objektes innerhalb der referenzierten Knotenseite. Damit bei der Addition von SI und I eine eindimensionale Größe entstehen kann, muss vorher SI noch um 7 nach links geshiftet werden. Die OID eines Objektes ist während der gesamten Lebenszeit eines (persistenten) Objektes gleich, unabhängig von der physikalischen Position innerhalb der Datei.

Dieses Dateiformat hat einige Vorteile:

- Kompaktheit: Durch die Verwendung eines Wörterbuches kann die Dateigröße reduziert werden. Da es sich bei den Informationen, die in XML-Dokumenten gespeichert sind, größtenteils um Textdaten handelt, können diese Informationen auch gut komprimiert werden. PDOM verwendet dazu den in Java vorhandenen GZIP-Algorithmus, der Kompressionsraten auf diesen Daten von bis zu 60% erreicht.

- Schneller wahlfreier Zugriff: Durch die oben genannte Kompaktheit kann Platz gespart werden, was sich auch auf die Anzahl der I/O Operationen auswirkt. Auf einer Knotenseite sind 128 DOM-Objekte gespeichert. Üblicherweise werden Objekte, die zusammen gespeichert werden, auch zusammen wieder aufgerufen, was hier besonders schnell geht, da die Objekte sich wahrscheinlich auf der selben Seite befinden.

- Der Wiederanlauf bei einem Absturz innerhalb einer Transaktion wird durch die Verzeigerung am Anfang der Datei erleichtert. Die Einbringstrategie ist damit Atomic. Für eine genaue Beschreibung des sog. Schattenspeicherkonzeptes sei der interessierte Leser auf [15] verwiesen.

## Serialisierung und Deserialisierung von Objekten

Einer der Vorteile bei der Benutzung von PDOM ist die Möglichkeit, beliebig große XML-Dokumente zu verwalten. Damit dies erreicht werden kann, darf die Speicherverwaltung von DOM-Objekten nicht komplett durch die Java Speicherbereinigung erfolgen.

Hierzu wird, wenn eine Knotenseite geladen wird, an Stelle jedes DOM-Objektes nur ein Proxy-Objekt geladen. Greift man nun über ein Proxy-Objekt auf ein DOM-Objekt zu, so wird dieses DOM-Objekt aus dem Bytestrom deserialisiert und in einen Cache eingereiht. Zusätzlich wird im Proxy-Objekt hinterlegt, dass sich nun das DOM-Objekt im Hauptspeicher bzw. im Cache befindet. Ein Proxy-Objekt besteht damit im Wesentlichen nur aus der OID des referenzierten DOM-Objektes und der Position des DOM-Objektes im Cache (falls dieses geladen ist). Müssen nun, auf Grund von Speichermangel oder weil neue DOM-Objekte in den Cache eingelagert werden sollen, alte DOM-Objekte entfernt werden, so werden lediglich die entsprechenden Cacheverweise in den Proxy-Objekten entfernt. Fordert nun eine Anwendung ein nicht mehr im Cache befindliches Objekt an, so kann dieses jederzeit nachgeladen werden, da die Anwendung nur das Proxy-Objekt referenziert.

Als Cachestrategie wird eine abgewandelte LRU-Strategie verwendet, die auf die Besonderheiten von Baumtraversierung abgestimmt ist. Bei reinem LRU liegt die Idee zu Grunde, dass das älteste Element das

Element ist, welches in Zukunft am wenigsten gebraucht wird. Diese Annahme ist bei der Baumtraversierung nicht erfüllt, da das älteste Element der Baumknoten ist, der, wenn der Unterbaum abgearbeitet ist, direkt wieder gebraucht wird. Wenn man eine nach Änderungszeiten von Objekten geordnete Warteschlange für die abgewandelten LRU-Strategie zu Grunde legt, so werden einige der ältesten Elemente (i.d.R. Elemente mit Position 1-3) vor jedem Verdrängen erneut angefasst, was zu einer Veränderung der Warteschlangenposition führt. Diese Strategie hat sich in praktischen Experimenten im allgemeinen Fall als nicht schlechter als einfaches LRU erwiesen. Manche Zugriffsmuster wie z.B. Baumtraversierung können mit dieser Strategie deutlich besser befriedigt werden.

## Sichere Dateiänderungen

Im folgenden Abschnitt soll darauf eingegangen werden, wie Änderungen sicher zurück in die Datenbank bzw. Datei eingebracht werden.

Zu Beginn ist die Datei in Zustand (a). Befinden sich nun DOM-Objekte im Cache, die verdrängt werden müssen und verändert wurden, so werden die veränderten Knotenseiten mit den veränderten DOM-Objekten an das Ende der Datei geschrieben (b). Wird nun die Transaktion durch ein commit abgeschlossen, so wird an das Ende der Datei ein neuer Knotenseitenindex angelegt und die Verzeigerung am Anfang der Datei auf den neuen Knotenseitenindex verändert (c). Mit Änderungen im Wörterbuch wird auf dieselbe Art und Weise verfahren (d).

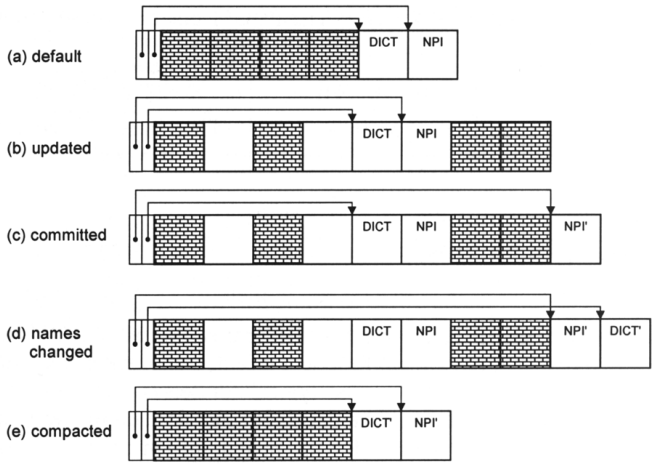



**Abb. Realisierung sicherer Dateiänderungen**

Mit der Zeit werden die gespeicherten Daten fragmentiert, was sich ebenfalls an der gestiegenen Dateigröße ablesen lässt.

Hierzu existieren in PDOM zwei verschiedene Kompaktierungstechniken. Bei der einfachen Kompaktierung wird nur die Dateigröße reduziert, indem alle nicht gelöschte Knotenseiten mit einem neuen Knotenseitenindex und einem Wörterbuch in eine neue Datei geschrieben werden. Eine vollständige Kompaktierung wird dann ausgeführt, wenn der DOM-Baum erneut durchlaufen wird und die einzelnen DOM-Objekte in eine neue Datei eingefügt werden. Mit der vollständigen Kompaktierung werden somit alle Lücken entfernt, die beim Löschen von DOM-Objekten entstanden sind.

## Performanz Evaluierung

| | Einmalige Traversierung | | Median von 10 Traversierungen | |
|---|---|---|---|---|
| | Zeit[ms] | Speicher[KB] | Zeit[ms] | Speicher[KB] |
| 1. IDOM, in Memory | 17115 | 38219 | 412 | 38219 |
| 2. PDOM, Cache | 6208 | 43176 | 705 | 43176 |
| 3. PDOM, kein Cache | 3065 | 839 | 2133 | 840 |

In obiger Tabelle sind die Ergebnisse eines Experiments [14] dargestellt. Aufgabe war es, ein etwa 7.5MB großes aus 327.145 DOM-Objekten bestehendes XML-Dokument (37 Stücke Shakespeares im XML-Format) einzulesen und den daraus entstehenden DOM-Baum ein oder mehrere Male zu durchlaufen. Die binäre PDOM-Datei wurde von PDOM innerhalb von 33sek. erstellt und wurde für alle Läufe mit PDOM verwendet.

Bei der einmaligen Traversierung stellt man fest, dass PDOM (ohne Cache) um den Faktor 4.5 schneller ist und auch nur 2.2% des Speichers benötigt als normales DOM (IDOM). Die Laufzeitreduzierung lässt sich hauptsächlich auf den deutlich geringeren Parseaufwand zurückzuführen. PDOM (mit Cache) schneidet hier schlechter ab, da bei der einmaligen Traversierung kein einziger Cache-Hit vorkommt.

Bei dem Median von 10 Traversierungen fällt auf, dass normales DOM um 75% schneller ist als PDOM (mit Cache). PDOM (mit Cache) muss hier zusätzliche Aufgaben wie Cachemanagement, Verwaltung von Proxy-Objekten verrichten, was neben dem Overhead für synchronisierte Methodenaufrufe, zur Verlängerung der Laufzeit führt. Auch die Speicheranforderungen von PDOM (mit Cache) sind größer, was sich hauptsächlich auf zusätzliche Dateipuffer, Indexstrukturen und Cacheverwaltung zurückführen lässt.

## 6.2 NATIX

Natix verwendet ähnlich wie PDOM für die Speicherung von XML-Dokumenten ein hybrides Verfahren. Im Folgenden wollen wir uns nur mit der eigentlichen Datenstruktur auseinandersetzen. Dieser Abschnitt basiert auf Kanne und Moerkotte [16].

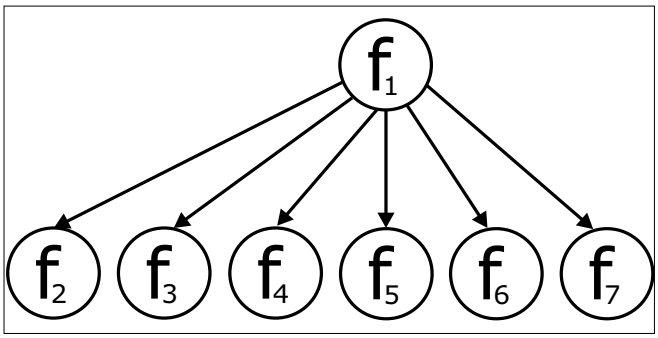



**Abb. 14 Logische Struktur**

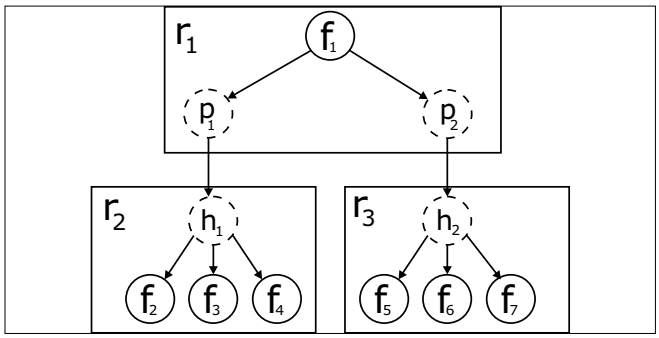



**Abb. 13 Physikalische Struktur**

Zu Beginn soll ein Überblick über das physikalische Datenmodell von NATIX gegeben werden. Hierzu sei auf die Abbildung 14 verwiesen. In dieser Abbildung ist ein Baum dargestellt, in dem der Elternknoten $f_1$ die Kindknoten $f_2 \ldots f_7$ besitzt. Dieser Baum wird auf einzelne Datensätzen $r_1 \ldots r_3$ verteilt, die dann in einer Datei gespeichert werden. Jede dieser Datensätze hat eine feste Größe und ist intern noch einmal in einzelne Slots unterteilt. In jedem Slot kann ein kompletter Unterbaum abgelegt werden. Ist ein Datensatz für einen Unterbaum zu klein, so wird dieser unter Zuhilfenahme von sog. Proxyknoten und Hilfsknoten auf mehrere Datensätze verteilt. In dem Datensatz, dessen Größe zu klein war, werden an Stelle des Unterbaums Proxyknoten (hier $p_1, p_2$) eingefügt, in denen die neue Position, bestehend aus Satznummer und Slot des Unterbaums, enthalten ist(in Abb. 13 Datensatz $r_1$). Die einzelnen Unterbäume haben, nachdem sie in einen neuen Datensatz eingefügt wurden, keine Wurzel, weshalb diese durch einen Hilfsknoten (hier $h_1, h_2$) ersetzt wird.

## Effiziente Speicherverwaltung

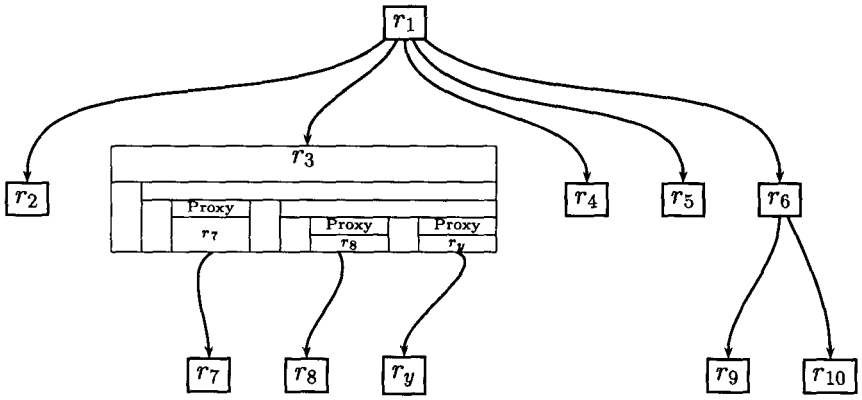



**Abb. 15 Mehrwegbaum**

Nachdem ein Überblick über die Grundzüge des physikalischen Datenmodells gegeben wurde, soll im Folgenden geklärt werden, wie ein zu großer Baum „geeignet" auf mehrere Datensätze verteilt wird.

Betrachtet man den Datenbaum als einen Mehrwegbaum, in dem die Datensätze als Knoten auftreten und jeder Datensatz einen kleinen Teil des gesamten logischen Datenbaums darstellt, so kann man diesen Baum

durchaus mit der Datenstruktur eines B-Baums vergleichen (siehe Abb. 15). Der einzige Unterschied ist, dass beim einfachen B-Baum die Schlüsselwerte aus einfachen Werten wie Zahlen oder Zeichenfolgen bestehen, wohingegen hier die strukturellen Eigenschaften des Datenbaums verwendet werden.

Der verwendete Algorithmus, welcher in im folgenden dargestellt ist, weicht deshalb auch nicht besonders von dem Algorithmus für B-Bäume ab.

Algorithmus:

1. Finde den Datensatz *r*, in die der Knoten eingefügt werden soll
2. Falls nicht genügend freier Platz in *r* ist, versuche *r* zu verschieben.
   Falls immer noch kein Platz ist, so splitte den Datensatz:
   a. Finde den Separator durch rekursiven Abstieg in den Unterbaum von r
   b. Verteile die daraus entstehenden Partitionen auf mehrere Datensätze
   c. Füge den Separator in den Elterndatensatz ein, indem diese Prozedur rekursiv aufgerufen wird
3. Füge den neuen Knoten ein

## Bestimmen des Einfügeortes

Man nehme an, es soll ein Knoten $f_n$ unter einen Knoten $f_1$ eingefügt werden. Wenn sich keine Proxy-Knoten in dem Datensatz befinden, so ist der Einfügeort gleich dem Datensatz, in dem sich f1 befindet. Gibt es nun aber Proxyknoten ($p_a$, $p_b$), so kann $f_n$ zusätzlich auch unter die Hilfsknoten ($h_a$, $h_b$) eingefügt werden, wie es auch in Abb. 15 dargestellt ist. Vom Prinzip könnte sich das System nun den Datensatz $r_a$, $r_b$ oder $r_c$ als Einfügeort wählen, was aber nicht unbedingt immer von Vorteil sein muss, da bestimmte Daten oft zusammen gespeichert werden sollten. Dies kann beispielsweise für Eltern-Kind-Navigation von Wichtigkeit sein, da sonst unter Umständen sehr viele Seiten geladen werden müssen. Hierzu sei $p(\ ^*_{DTD}\ )$ $N$ eine bijektive Abbildung zwischen Elementnamen des DTDs und den natürlichen Zahlen. Seien nun den Elementnamen $e_1, e_2$ $_{DTD}$ die Werte $p(e_1)=i$ und $p(e_2)=j$ zugeordnet. In einer Splitmatrix S, welche dem Algorithmus als zusätzlicher Parameter übergeben wird, ist das gewünschte Einfügeverhalten hinterlegt. So bedeutet $s_{ij}=0$, dass $e_1$ und $e_2$ niemals zusammen in einem Datensatz untergebracht werden dürfen. $s_{ij}=$ bedeutet, dass $e_1$ und $e_2$ so lange wie es geht in einem Datensatz untergebracht werden sollen. Andere Werte für $s_{ij}$ überlassen dem Algorithmus die Entscheidung.

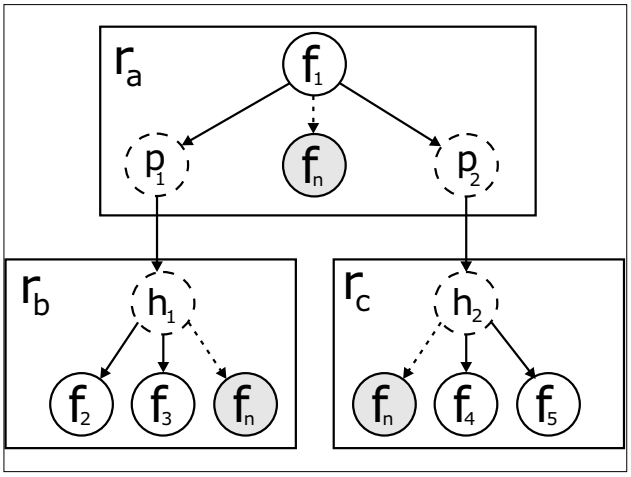



**Abb. 15 Einfügepositionen**

## Splitten eines Datensatzes

Bei einem normalen B-Baum wird der mittlere Schlüssel aller in einem Knoten gespeicherten Schlüsselwerte benutzt, um den Knoten in zwei Paritionen aufzuteilen. Derselbe Vorgang muss nun in der vorliegenden Baumstruktur durchgeführt werden, nur mit dem Unterschied, dass anstatt eines einzigen Knotens ein kleiner Teilbaum als Separator fungiert.

## Bestimmen eines Separators

Angenommen, man wählt einen Knoten *d*, welcher bestimmte Eigenschaften hat, die später noch genauer erläutert werden. Der Separator *S* ist dann definiert als die Menge an Knoten, die auf dem Pfad von *d* nach der Wurzel des Datensatzes liegen, wobei d nicht in der Menge enthalten ist. In Abb. 17 ist $d=f_7$ und $S=\{f_6, f_1\}$. Durch diesen Separator *S* und den Knoten *d* wird der Datensatz in drei disjunkte Partitionen partitioniert: die linke Partition *L*, die rechte Partition *R* und den Separator *S* selbst. Die rechte Partition *R* besteht aus einer Menge an Bäumen, die wie folgt definiert ist: *R*={Unterbaum von d; Unterbäume von d's rechten Geschwistern; alle Unterbäume von Knoten, die rechte Geschwister von Knoten aus S sind}. Die linkte Partition *L* besteht aus den übriggebliebenen Knoten. Für Abb. entstehen so die Partitionenen $R=\{f_7,\ldots, f_{14}\}$ und $L=\{f_2,\ldots, f_5\}$.

Nachdem nun geklärt ist, wie die linke und die rechte Partition erzeugt werden, bleibt noch die Bestimmung eines geeigneten d's. Das *d* sollte so gewählt sein, dass die Partitionen *L* und *R* möglichst gleich groß sind.

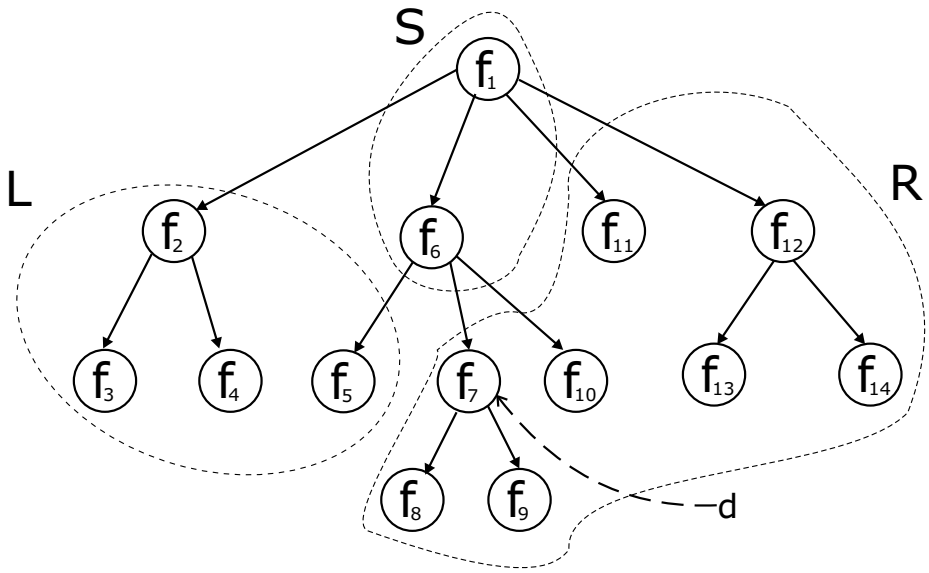



**Abb. 17 Einteilung mit Separator**

Weiterhin sollten die abgespalteten Unterbäume nicht zu klein sein. So würde ein Unterbaum bestehend aus nur zwei Knoten nicht besonders effizient gespeichert, da zusätzlich noch ein Proxy und Hilfsknoten notwendig wären. Die minimale Größe von Unterbäumen und auch das Verhältnis von linker zu rechter Partition können mit Konfigurationsparametern eingestellt werden. So könnten bestimmte Datenfolgen, wenn sie eingefügt werden, zu einer Degenerierung des Baums führen. Ist dies vorher bekannt, so kann dies verhindert werden, indem beispielsweise beim Split die linke Partition immer besonders klein gehalten wird . Der Algorithmus zur Bestimmung von *d* wählt die "physikalische Mitte" und beginnt auf diese Weise den rekursiven Abstieg in den Unterbaum, bis entweder ein Blatt gefunden ist oder der restliche Unterbaum kleiner ist als die minimal erlaubte Größe für Unterbäume. "Physikalische Mitte" ist hier eher ein Gedankenkonstrukt, da auf den Knoten einer Baumebene keine Ordnung definiert ist und somit auf andere Ordnungskriterien wie z.B. die absolute Speicheradresse zurückzugreifen ist.

## Verteilen der Knoten auf Datensätze

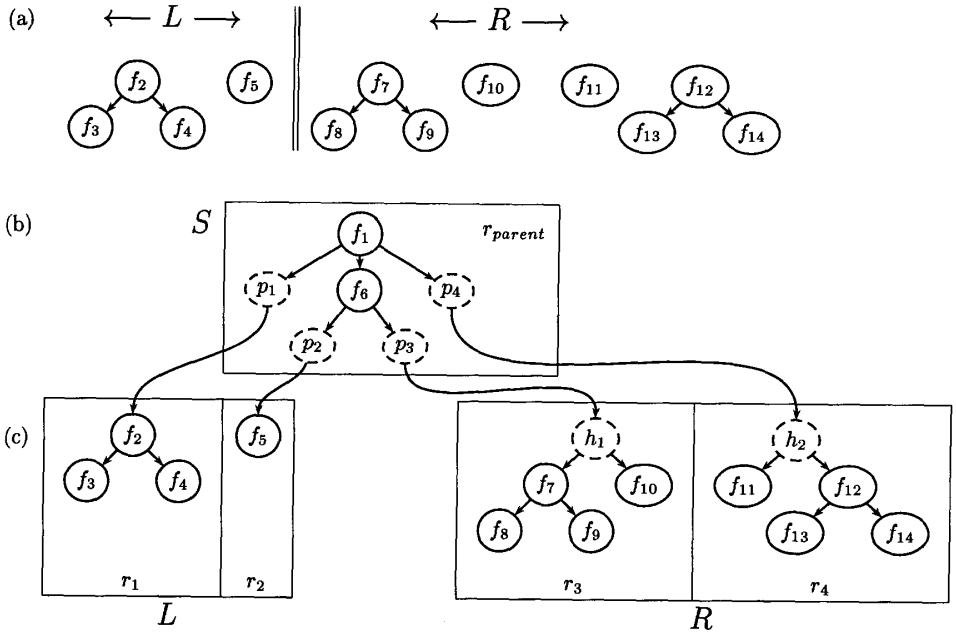



**Abb. 18 Verteilung der Knoten**

Nachdem die Partitionierung feststeht, wird der Separator aus dem Datensatz entfernt. Übrig bleibt dann nur noch ein Wald von Unterbäumen, wie auch in Abb. 18 (a) zu sehen ist. Alle Knoten des Separators und die Proxies für die nun fehlenden Unterbäume werden in den Elterndatensatz eingefügt (siehe Abb. 18(b)). Die Partitionenen L und R werden in eigenen Datensätzen gespeichert, wobei für die Knoten, die im Originalbaum

Geschwister waren, Hilfsknoten eingefügt werden. In Abb 18 (c) sind dies die Knoten $h_1$ und $h_2$. Bei dem Einfügen des Separators in den Elterndatensatz kann es wiederum zu einem Überlauf kommen, der mit demselben Algorithmus rekursiv bearbeitet wird. Hat ein Separator keinen Elterndatensatz, so wird ein neuer Datensatz angelegt.

# 7 Zusammenfassung und Ausblick

In diesem Text wurde versucht, einen Überblick über den aktuellen Stand im Bereich XML-Databinding zu geben. Zu Beginn haben wir uns damit beschäftigt, was XML-Databinding ist. Hierbei sind wir auf die Bedeutung von Typsystemen und dem darunterliegenden Datenmodell eingegangen. Es hat sich gezeigt, dass für jedes Konzept auf XML-Seite wie beispielsweise Elemente, Schema oder Attribute ein entsprechender Gegenpart auf Javaseite existiert. Ein Vergleich mit dem objektorientierten und hierarchischen Datenmodell hat gezeigt, dass eine XML-Datei eine Kombination beider Datenmodelle ist, wobei keines der Datenmodelle komplett in XML wiederzufinden ist. Hier soll noch einmal die besondere Rolle der Verhaltens von XML-Dokumenten hervorgehoben werden. Im Moment ist ein XML-Dokument nur eine Ansammlung von Daten. In Zukunft könnte es aber möglich sein, kleine Sprachkonstrukte, ähnlich wie JavaScript in Html-Dateien, in ein XML-Dokument einzubetten, die dann bestimmte Aufgaben übernehmen. Für die Verarbeitung von XML erweisen sich funktionale Programmiersprachen als sehr geeignet, da diese sehr gut mit rekursiven Strukturen umgehen können (Beispiel XDuce [6] oder XMLambda [7]). Daraufhin haben wir uns eingehender mit den Vorteilen von XML-Databinding beschäftigt. Der Hauptvorteil liegt sicher in der Vermeidung von manuell geschriebenem Code, womit beispielsweise Implementierung, Wartung oder Reviews für diesen Code deutlich vereinfacht werden. Weiterhin kann beispielsweise JAXB leichter erlernt werden als Ansätze, die ohne XML-Databinding auskommen.

Nach dieser recht allgemein gehaltenen Betrachtung von XML-Databinding haben wir uns mit den einzelnen Produkten näher beschäftigt. Hierbei haben wir den Databindingprozess an JAXB genauer betrachtet. JAXB wird in Zukunft im Javabereich sicher das dominante Databindingprodukt sein, da es auf einer offiziellen Spezifikation beruht (siehe Java Specification Requests: JSR31 [17]). Aus einem DTD werden mit Hilfe eines Compilers und eines Bindungsschemas Javaklassen erzeugt, die alle Logik enthalten, um einen Java Objektgraphen zu erstellen. Das Bindungsschema enthält Anpassungen, die der Compiler bei der Erstellung der Javaklassen berücksichtigt. Sinn dieses Vorgangs ist es, Geschäftsobjekte zu erhalten, die dem Inhalt des XML-Dokuments entsprechen.

Neben JAXB bietet Zeus einen ähnlichen Funktionsumfang, wobei der Hauptaugenmerk bei Zeus auf einem flexibleren Erstellungsprozess für die zu erzeugenden Javaklassen liegt. Die von Zeus erstellen Klassen benötigen neben einem SAX-Parser keine weiteren Bibliotheken. Castor war eines der ersten Databindingprodukte auf dem Markt. Die Funktionalität weicht ebenfalls nicht sonderlich von JAXB ab, wobei Castor zusätzliche Funktionalität im Bereich Speicherung von Objekten in relationalen Datenbanken und Verzeichnisdiensten (z.B. LDAP) bietet. Einen ganz anderen Ansatz als die bisherigen Produkte verwendet Quick. Quick gestattet ein deutlich flexibleres Databinding, womit gerade heterogene XML-Dokumente, die auf eine bestimmte (bestehende) Javaklasse abgebildet werden, besonders gut verarbeitet werden können. Dieser Vorteil wird durch einen etwas komplizierteren Erstellungsprozess erkauft.

Persistent DOM (PDOM) ist eine Speicherungslösung, die XML-Dokumente effizient und mit Transaktionsschutz speichert. Hierzu existiert eine DOM-Schnittstelle, die um einige für die Persistenzfunktionalität wichtige Funktionen erweitert ist. XML-Dokumente, die in PDOM gespeichert sind, können einer Anwendung sehr schnell zur Verfügung gestellt werden, da Aufgaben wie Parsen des Dokumentes entfallen. Damit PDOM mit einer reinen InMemory-DOM-Implementierung (IDOM) konkurrieren kann, existiert ein Cachesystem, das den Nachteil der zusätzlichen E/A mit der Festplatte reduziert. Wird auf den Cache verzichtet, so benötigt PDOM deutlich weniger Speicher als IDOM. In einem Experiment benutzte PDOM nur 3 Prozent des Speichers einer IDOM-Implementierung.

Natix ist eine spezielle Speicherungsstruktur für XML-Daten und verwaltet XML-Daten in einer Baumstruktur, die an einen B-Baum erinnert. Für dieses Baumstruktur wurde ein Algorithmus präsentiert, der das Splitting ähnlich wie bei einem B-Baum vornimmt. Einziger Unterschied ist hier, dass dieser Algorithmus nicht mit eindimensionalen einfachen Datentypen, sondern mit Baumstrukturen arbeitet.

XML-Databinding wird mit Sicherheit in einigen Jahren eine etablierte und vielgenutzte Technologie sein, da es eine ideale Ergänzung zu Technologien wie beispielsweise SOAP oder UDDI ist. Weiterhin wird XML inzwischen recht häufig als ein Datentransportformat eingesetzt, was bedeutet, dass XML-Daten aus unterschiedlichen Applikationen gelesen und geschrieben werden müssen. Auch XML-basierte Konfigurationsdateien (z.B. Deployment-Deskriptoren) würden durch XML-Databinding an Bedeutung gewinnen. Im Moment ist XML-Databinding weitgehend auf Java fixiert, da Java als eine der ersten Sprachen

eine ausgesprochen gute Unterstützung für XML bot. Neben Java existieren aber noch andere Implementierungen, wie beispielsweise in .NET, wobei hier Databinding einen deutlich größeren Umfang besitzt. So existieren neben Databinding für XML-Daten auch andere Bindungen wie C#->ADO.NET [18].

XML-Databinding besitzt im Moment noch einige Nachteile:

- So wird eigentlich bei allen Implementierungen der komplette Objektgraph im Speicher erstellt, wodurch man sich ein ähnliches Problem wie bei DOM einhandelt, wenn die zu ladenden Daten den Hauptspeicher übersteigen. Eine Lösungsmöglichkeit wäre eventuell XML-Databinding mit einer Technologie wie PDOM zu vereinigen.

- So gut wie jede Implementierung geht davon aus, dass zu einem Vorkommen in einem XML-Dokument genau eine Klassendatei existiert. Heterogene Strukturen, bei denen mehrere unterschiedliche XML-Dokumente auf eine evtl. sogar bestehende Klasse gemappt werden, werden nicht unterstützt. Quick zeigt in diesem Zusammenhang verschiedene Lösungsmöglichkeiten auf.

- Änderungen zur Laufzeit können von den verschiedenen XML-Databindingprodukten nicht verarbeitet werden. Dies liegt natürlich daran, dass es sich bei den verwendeten Klassen um kompilierte Klassen handelt. Angenommen, es wird nur ein Attribut umbenannt, so muss in einem solchen Falle die komplette Applikation neu deployed werden, was unter Umständen sehr aufwendig sein kann.

- Dokumentenzentrierte XML-Dateien, wie sie beispielsweise in Workflows verwendet werden, eignen sich ebenfalls nicht besonders gut für XML-Databinding, da die zugrundeliegende Dokumentenstruktur sich häufig ändern kann. In einem solchen Fall sind bestehende Technologien wie SAX oder DOM passender.

# Literatur


[1]: World Wide Web Consortium: XML-Schema Spezifikation [2.5.2001] (http://www.w3.org/XML/Schema)

[2]: Sun: Java Architecture for XML Binding (http://java.sun.com/xml/jaxb)

[3] David Mertz: XML and Data Models: Hierarchical, Relational and Object-Oriented, 2001 (http://gnosis.cx/publish/programming/xml_matters_8.html)

[4] World Wide Web Consortium: XML Linking Language (XLink) Version 1.0 [27.7.2001] (http://www.w3.org/TR/xlink)

[5] Michael Ernst: Laboratory in Software Engineering 6.170 (http://web.mit.edu/6.170/archive/Old-Spring01/lectures/6170-lec10.pdf)

[6] XDuce: http://xduce.sourceforge.net

[7] Erik Meijer, Mark Shields: XMLambda: A Functional Language for Constructing and Manipulating XML Documents (Draft), 1999

[8] Sun: Java Architecture for XML Binding (JAXB) Specification 0.90 - Proposed Final Draft, 2002

[9] Brett McLaughlin: Java and XML Data Binding, O'Reilly UK, 2002

[10] OASIS Technical Committee: RELAX NG (http://www.oasis-open.org/committees/relax-ng)

[11] Zeus: http://zeus.enhydra.org

[12] Castor: http://castor.exolab.org

[13] Sun: JSR 057 - Long Term Persistence for JavaBeansTM 1.0 Specification (http://www.jcp.org/aboutJava/communityprocess/first/jsr057)

[14] Gerald Huck, Ingo Macherius, Peter Fankhauser: PDOM: A Leightweight Persistency Support for the Document Object Model). In: Proceedings of the 1999 OOPSLA Workshop Java and Databases: Persistence Options

[15] Theo Härder, Erhard Rahm: Datenbanksysteme: Konzepte und Techniken der Implementierung, Springer, 1999

[16] Carl-Christian Kanne, Guido Moerkotte: Efficient Storage of XML-Data. In: Proceedings of ICDE, California, Seite 198ff, 2000

[17] JSR 31 XML Data Binding Specification (http://www.jcp.org/en/jsr/detail?id=31)

[18] Niel Bornstein: XML Data-Binding: Comparing Castor to .NET (http://www.xml.com/pub/a/2002/07/24/databinding.html)

[19] Quick: http://qare.sourceforge.net/web/2001-12/products/quick/index.html

[20] C. J. Date: An Introduction to Database Systems, Addison Wesley, 1990

[21] Relax NG Draft International Standard (http://www.y12.doe.gov/sgml/sc34/document/0320.htm)