



**TU Wien**



**Business Informatics Group**  
Institut für Softwaretechnik und Interaktive Systeme

---

Peter Gerstbach

## **XML Data Binding**

**Bakkalaureatsarbeit**

Betreuender Assistent: Univ.Ass. Mag. Martin Bernauer

Wien, 3. November 2004

## Kurzfassung

Diese Arbeit ist eine Einführung in *XML Data Binding*, einer Technik mit der sich XML-Daten in Objekte einer Programmiersprache umwandeln lassen. Weiters wird die JAXB-Spezifikation von Sun, deren Funktionsumfang und Anpassungsmöglichkeiten besprochen und die Referenzimplementierung anhand von zwei Beispielen getestet. Das erste Beispiel erläutert auf einfache Weise die Funktionsweise von JAXB, das zweite und komplexere Beispiel zeigt neben Details auch die Grenzen und Probleme der Spezifikation.

# Inhaltsverzeichnis

<b>I. Theoretischer Teil</b>	<b>4</b>
<b>1. Einleitung</b>	<b>5</b>
<b>2. Grundlagen von XML Data Binding</b>	<b>6</b>
2.1. Klassen-Generierung . . . . .	7
2.2. Unmarshalling . . . . .	8
2.3. Marshalling . . . . .	9
2.4. Anpassung . . . . .	9
<b>II. Praktischer Teil</b>	<b>10</b>
<b>3. JAXB</b>	<b>11</b>
3.1. Verwendung von JAXB . . . . .	11
3.1.1. Generieren der Klassen und Interfaces . . . . .	11
3.1.2. Verwendung der generierten Klassen und Interfaces . . . . .	13
3.2. Mapping in JAXB . . . . .	16
3.2.1. Einfache Typen . . . . .	18
3.2.2. Komplexe Typen . . . . .	19
3.2.3. Elementgruppen . . . . .	20
3.2.4. Elemente . . . . .	20
3.2.5. Attribute und Attributgruppen . . . . .	20
3.2.6. Inhaltsmodell . . . . .	20
3.2.7. Bezeichner und Namensräume . . . . .	21
<b>4. Beispiel Active XML Schema</b>	<b>23</b>
4.1. Active XML Schema im Überblick . . . . .	23
4.2. Generierung und Mapping . . . . .	23
4.2.1. Fehler und Warnungen . . . . .	23
4.2.2. Erweiterungen . . . . .	25
4.2.3. Typsichere Aufzählung . . . . .	25
4.3. UML Klassendiagramm . . . . .	26
<b>III. Zusammenfassung</b>	<b>28</b>

---

<b>5. Zusammenfassung</b>	<b>29</b>
<b>A. Listings für Active XML Schema</b>	<b>30</b>
A.1. Binding-Declaration . . . . .	30
A.2. Typsichere Aufzählung . . . . .	31
<b>Abbildungsverzeichnis</b>	<b>34</b>
<b>Tabellenverzeichnis</b>	<b>35</b>
<b>Listings</b>	<b>36</b>
<b>Literaturverzeichnis</b>	<b>37</b>

**Teil I.**  
**Theoretischer Teil**

# 1. Einleitung

XML und Java haben sich als wichtige Technologie für die Programmierung von unterschiedlichsten Anwendungen am Markt etabliert. XML ermöglicht es Daten zu strukturieren und unabhängig von der Architektur zu verarbeiten. Durch XML basierte Webservices können Anwendungen über Rechner- und Betriebssystemgrenzen hinweg verteilt werden. Heute verwendet fast jede moderne Anwendung in der einen oder anderen Weise XML. Meistens wird die Verarbeitung von XML mit Hilfe von Low-Level-APIs, wie SAX und DOM, programmiert. Diese Ansätze haben jedoch den Nachteil, dass in der Anwendungslogik viel Wissen über die Struktur und Verarbeitung der XML-Daten steckt. XML Data Binding schafft hier eine direkte Verbindung zwischen XML und Java-Objekten, die eine Programmierung ermöglicht, bei der man sich nicht mit XML Interna beschäftigen muss.

Diese Arbeit gibt eine Einführung in XML Data Binding und erklärt das bekannteste Data Binding-Framework JAXB, die Spezifikation und den Aufbau und die Funktionsweise der Referenzimplementierung von Sun. In einem praktischen Teil wird JAXB anhand zweier Beispiele getestet. Das erste Beispiel veranschaulicht die Vorteile, die die Verwendung von JAXB mit sich bringt. Das zweite Beispiel zeigt auch die Grenzen der derzeitigen Version von JAXB auf, da sie auf dem viel komplexeren Schema für *Active XML* basiert.

## 2. Grundlagen von XML Data Binding

Derzeit gibt es grundsätzlich zwei Arten von Parsern, um auf XML Daten zuzugreifen: SAX- und DOM-basierende Parser [McL02]. Ein *SAX-Parser* liest ein XML-Dokument linear ein und erzeugt Events, wenn es an relevante Stellen kommt, wie z.B. der Beginn eines XML-Elements. Die dokumentspezifische Anwendungslogik steckt in einem Event-Handler, der vom Parser eingebunden wird und die erzeugten Events abarbeitet. Dieser so genannte Call-Back-Mechanismus ist eine einfache und ressourcenschonende Art, XML zu verarbeiten. Einem *DOM-Parser* liegt ein komplettes, gut dokumentiertes Objektmodell für XML zugrunde, das Document Object Model des W3Cs. Ein DOM-Parser erzeugt aus einem XML-Dokument einen DOM-Baum im Speicher. Da XML einen hierarchischen Aufbau hat, eignet sich die Darstellungsweise in einem Baum besonders. Teile dieses Baums lassen sich dann einfach abfragen, hinzufügen und verschieben. Der wahlfreie Zugriff erfordert aber viel Speicher. Da das DOM-Modell sprachen- und plattformübergreifend spezifiziert ist, nutzt es viele Sprachvorteile von Java nicht und lässt dadurch die Handhabung für Java-Programmierer nicht ganz intuitiv wirken.

Bei SAX und DOM spricht man von *Low-Level-APIs*, da sie auf niedrigem Level sehr nahe an den Daten operieren. XML Data Binding ist hingegen eine *High-Level-API*. Data Binding schafft die direkte Verbindung zwischen Komponenten eines XML-Dokumentes und den Objekten einer Programmiersprache. Durch geeignete Methoden der Objekte ist dann ein einfacher Zugriff auf die Daten möglich. Damit sichergestellt werden kann, dass die Umwandlung eines XML-Dokumentes in ein Objekt erfolgreich durchgeführt werden kann, muss das XML mit einem Schema, wie z.B. XML Schema, begrenzt sein. Das Data Binding Framework kann anhand dieses Schemas eine Klassenstruktur erzeugen. Diese kann dann für alle XML Dokumente verwendet werden, die gegen das Schema validierbar sind.

Zur Erläuterung folgt ein kurzes Beispiel:

Listing 2.1: person.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <person xmlns="http://www.gerstbach.at/2004/XMLDataBinding/
   person-1_0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://www.gerstbach.at
   /2004/XMLDataBinding/person-1_0
3 person.xsd">
4   <name>Peter Gerstbach</name>
5   <dateOfBirth>1982-03-23</dateOfBirth>
6   <height>188</height>
```

```
7 </person>
```

Listing 2.2: person.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema targetNamespace="http://www.gerstbach.at/2004/
  XMLDataBinding/person-1_0" xmlns:xs="http://www.w3.org
  /2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified">
3   <xs:element name="person">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="name" type="xs:string"/>
7         <xs:element name="dateOfBirth" type="xs:date"/>
8         <xs:element name="height" type="xs:integer"/>
9       </xs:sequence>
10    </xs:complexType>
11  </xs:element>
12 </xs:schema>
```

Listing 2.1 zeigt ein XML-Dokument, das dazugehörige XML Schema ist in Listing 2.2 abgebildet. Das XML-Dokument enthält ein Element `<person>`, dieses wiederum u.a. das Element `<name>`, in dem der Name einer Person gespeichert ist, usw. Ein Data Binding Framework generiert in einem ersten Schritt aus dem Schema eine Klasse `Person`. In einem weiteren Schritt liest es das XML-Dokument ein und speichert die Werte aus dem Person-Element im Objekt `aPerson` der Klasse `Person`. Der Zugriff auf den Inhalt des Namen-Tags erfolgt dann z.B. über die Methode `getName`, in diesem Fall also: `aPerson.getName()`. Das Objekt enthält nun alle Daten aus dem XML-Dokument, kann nun verändert und wieder zurück in ein XML-Dokument umgewandelt werden.

Während des gesamten Vorgangs kommen zumeist vier Konzepte zur Anwendung (siehe auch Abbildung 2.1):

- Klassen-Generierung (siehe Abschnitt 2.1)
- Unmarshalling (siehe Abschnitt 2.2)
- Marshalling (siehe Abschnitt 2.3)
- Anpassung (siehe Abschnitt 2.4)

In den folgenden Abschnitten werden diese Konzepte näher erläutert.

## 2.1. Klassen-Generierung

Ziel von XML Data Binding ist es, ein XML Dokument in eine Instanz einer Java-Klasse umzuwandeln. Unter *Klassen-Generierung* versteht man das automatische Generieren von Klassen und Interfaces aus Dokumenttyp-Definitionen, z.B. XML-Schema: aus einem XML-Schema werden eine oder mehrere Java-Klassen bzw. Interfaces generiert. Jedes valide XML-Dokument des Schemas kann in ein

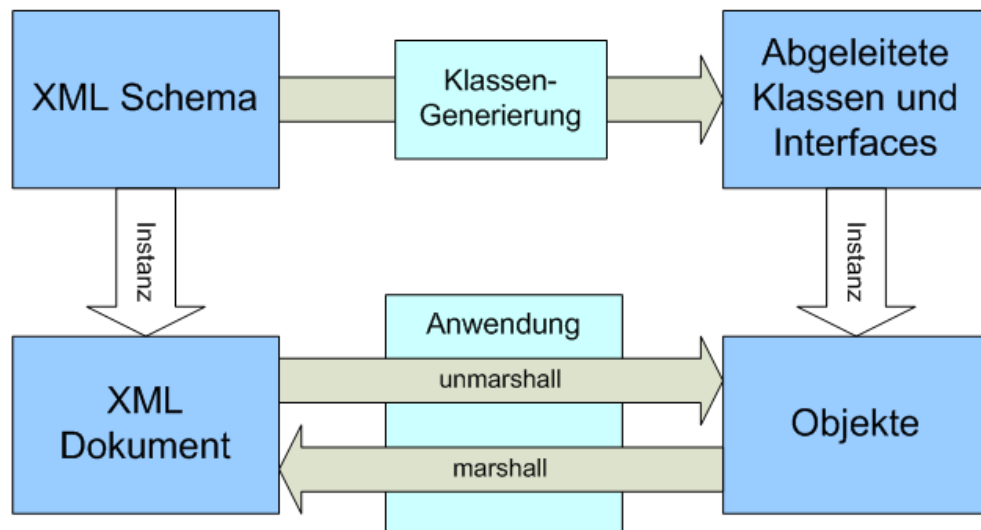


Abbildung 2.1.: Konzepte des XML Data Bindings  
(angelehnt an [OM03])

Java-Objekt der generierten Root-Klasse und wieder zurück umgewandelt werden. Bei den meisten Frameworks erzeugt ein externes Programm aus dem XML-Schema den Java-Quelltext. Die erzeugten Klassen bieten meistens JavaBean-ähnliche Methoden an, um die Daten zu manipulieren, beispielsweise `getName()` und `setName()`.

Die Phase der Generierung verbindet also ein XML Schema (die Beschreibung der Struktur eines XML-Dokuments) mit einer Klasse (die Beschreibung der Struktur eines Java-Objekts). Alle XML-Dokumente (Instanzen des Schemas) sind somit kompatibel mit dem Java-Objekt (eine Instanz der Java-Klasse).

## 2.2. Unmarshalling

Unter Unmarshalling versteht man das Erzeugen von Objekten aus einem XML-Dokument. Ein Objekt repräsentiert das Wurzelement des XML-Dokuments und beinhaltet die weiteren Objekte. Es entsteht ein Objekt-Baum, ähnlich aufgebaut wie die Elemente im XML-Dokument. Dieser Objekt-Baum darf jedoch nicht mit einem DOM-basierten Baum verwechselt werden: die Objekte sind Instanzen der zuvor generierten Klassen. In welcher Form das XML-Dokument bereitgestellt werden kann, ist je nach Implementierung unterschiedlich. Die meisten Frameworks unterstützen neben XML-Dateien bzw. Streams auch DOM-Bäume und SAX-Handler als Eingabe.

Neben dem Unmarshalling eines vorhandenen XML-Dokuments, ist es selbstverständlich auch möglich, diesen ohne Umweg im Speicher zu erzeugen. Dabei können zwei unterschiedliche Ansätze unterschieden werden: Objekt-Erzeugung mittels

- **Konstruktor:** dabei wird das Objekt direkt über den Konstruktor erzeugt.

Für das Person-Beispiel wäre das `Person aPerson = new Person();`; Diese Variante ist besonders einfach für den Programmierer.

- **Factory:** hier übernimmt eine Factory die Erzeugung des Objekts, eine mögliche Implementierung wäre `Person aPerson = ObjectFactory.createPerson();`

Die zwei bekannten Data Binding Frameworks JAXB und Castor unterscheiden sich genau in diesen zwei Varianten: JAXB verwendet Factories während Castor auf *normale* Objekte mit Konstruktoren zurückgreift [Exo04].

## 2.3. Marshalling

Unter Marshalling versteht man das Gegenteil von Unmarshalling, also die Umwandlung von Java-Objekten nach XML. Auch hier gilt wieder, dass das Ziel nicht unbedingt eine Datei sein muss. Ebenfalls muss das Objekt nicht zwingend durch einen Unmarshalling-Prozess erzeugt worden sein. Es wäre ebenso denkbar, dass das Objekt zuvor im Programm direkt (über den Konstruktor bzw. die Factory) neu erzeugt wurde.

Damit das Marshalling eines Objekts zu einem validen XML-Dokument überhaupt durchgeführt werden kann, muss das Objekt schon im Speicher auf seine Gültigkeit geprüft werden. Dazu reichen jedoch die Datentypen, die Java bereitstellt, nicht aus. Schon für einfache Angaben im Schema wie `minInclusive/maxInclusive` oder `minOccurs/maxOccurs` gibt es keine native Unterstützung in Java, ebenso wie für strukturelle Abhängigkeiten. Deswegen stellt das Framework Validierungsmethoden zur Verfügung, mit dem sich ein Objekt bezüglich des Schemas auf Gültigkeit überprüfen lässt. Diese Methode kann explizit aufgerufen werden, implizit beim Marshalling oder auch schon beim Setzen der Felder eines Objekts. Nur wenn die Validierung im Speicher erfolgreich ist, kann das Objekt durch Unmarshalling in ein XML-Dokument umgewandelt werden.

## 2.4. Anpassung

Da viele verschiedene Möglichkeiten denkbar sind, eine XML-Struktur in Java-Komponenten umzuwandeln, ist das Ergebnis je nach Framework meist unterschiedlich. Aus diesem Grund unterstützen viele Frameworks auch ein Mapping, wodurch die Zuordnung von XML- zu Java-Komponenten an die eigenen Vorstellungen angepasst werden kann. Dadurch lassen sich oft auch Klassen bestehender Java-Projekte integrieren. Mappings definieren beispielsweise, welche primitiven Java-Datentypen oder Wrapper-Klassen den Schema-Datentypen zugeordnet werden, wie eine Sequenz realisiert wird (als Array, Collection oder Aggregation von Objekten) oder wie einzelne Bezeichner lauten sollen.

# **Teil II.**

## **Praktischer Teil**

## 3. JAXB

Dieses Kapitel beschreibt das Data Binding Framework JAXB. Der erste Absatz gibt einen Überblick über die Entstehung des Frameworks. Im Abschnitt 3.1 wird die Verwendung der Implementierung von Sun erläutert und anhand eines einfachen Beispiels veranschaulicht. Der Abschnitt 3.2 beschreibt die Spezifikation von JAXB, behandelt werden vor allem die Mappingregeln und Anpassungsmöglichkeiten.

JAXB ist ein Akronym für *Java Architecture for XML Binding*. Genauer genommen ist JAXB kein Data Binding Framework, sondern dessen Spezifikation. Diesen Sachverhalt drückt auch der ausgeschriebene Name aus. Die Spezifikation wird, wie auch Java selbst, durch den Java Community Process entwickelt. Die Vorarbeiten dazu begannen im August 1999, eine Early Access-Version wurde im Juni 2001 veröffentlicht. Diese Version verwendete noch XML-DTDs als Basis des Generierungs-Prozesses. Die Version wurde aber nie fertig implementiert. Da das Design nicht auf eine Unterstützung von XML Schema ausgelegt war, wurde die Architektur und die API noch einmal gründlich überarbeitet, wobei auch Feedback zur Vorversion berücksichtigt wurde [Sos03]. Seit dem 8. Jänner 2003 ist JAXB als Version 1.0 verfügbar [FV03]. Sie basiert nun auf XML Schema (DTDs werden nicht mehr unterstützt) und verwendet einen Interface-Ansatz, um die Kompatibilität unterschiedlicher Binding-Frameworks zu gewährleisten. Die Referenzimplementierung ist in Suns Java Web Services Developer Pack ab Version 1.3 enthalten [Sun03b]. Da bis jetzt keine weitere vollständige Implementierung von JAXB verfügbar ist, bezieht sich dieses Kapitel nur auf die Implementierung von Sun.

### 3.1. Verwendung von JAXB

Um JAXB verwenden zu können, müssen folgende Schritte in dieser Reihenfolge durchgeführt werden:

1. Schreiben des Schemas
2. Generieren der Klassen und Interfaces (siehe Abschnitt 3.1.1)
3. Verwendung der generierten Klassen und Interfaces (siehe Abschnitt 3.1.2)

#### 3.1.1. Generieren der Klassen und Interfaces

Die JAXB-Implementierung von Sun enthält `xjc`, ein Tool zur Code-Generierung, das aus XML-Schema Java-Klassen und Interfaces generiert. Anhand des kurzen Beispiels 3.1 wird der von `xjc` erzeugte Sourcecode analysiert. Das Beispiel zeigt

ein XML-Schema Dokument, das eine Struktur für die Speicherung von Produktinformationen enthält. Ein Element `itemList` enthält Elemente `item`, diese enthalten wiederum die Elemente `productName`, `quantity` und `price`.

Listing 3.1: itemlist.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns="http://www.gerstbach.at/2004/
   XMLDataBinding/itemlist-1_0" targetNamespace="http://www
   .gerstbach.at/2004/XMLDataBinding/itemlist-1_0" xmlns:xs
   ="http://www.w3.org/2001/XMLSchema" elementFormDefault="
   qualified" attributeFormDefault="unqualified">
3 <xs:element name="itemList">
4   <xs:complexType>
5     <xs:sequence>
6       <xs:element name="item" maxOccurs="unbounded" type=
          "itemType">
7         </xs:element>
8       </xs:sequence>
9     </xs:complexType>
10  </xs:element>
11  <xs:complexType name="itemType">
12    <xs:sequence>
13      <xs:element name="productName" type="xs:string"/>
14      <xs:element name="quantity" type="xs:int"/>
15      <xs:element name="price" type="xs:double"/>
16    </xs:sequence>
17  </xs:complexType>
18 </xs:schema>
```

Wird `xjc` nur mit der Schema-Datei als Parameter aufgerufen, also z.B. `xjc itemlist.xsd` werden folgende Artefakte generiert:

- die Interfaces `ItemList`, `ItemListType` und `ItemType`
- die Klassen `ItemListImpl`, `ItemListTypeImpl` und `ItemTypeImpl`
- die Klasse `ObjectFactory`
- und einige weitere Klassen, auf die hier nicht näher eingegangen wird.

Entfernt man alle automatisch generierten Kommentare, ist dies der gesamte Inhalt von `ItemList.java`:

Listing 3.2: ItemList.java

```
1 package at.gerstbach._2004.xmldatabinding.itemlist_1_0;
2 public interface ItemList
3     extends javax.xml.bind.Element, at.gerstbach._2004.
       xmldatabinding.itemlist_1_0.ItemListType
4 {
5 }
```

Der Name des Packages wird durch den Namensraum des Wurzel-Elements bestimmt: aus dem Namensraum `http://www.gerstbach.at/-2004/XMLDataBinding/itemlist-1_0` wird das Paket `at.gerstbach._2004.-xmldatabinding.itemlist_1_0`. Dieser Name kann aber beim Aufruf von `xjc` durch die Angabe des Parameters `-p` verändert werden. Das Interface selbst erweitert ein weiteres Interface von Typ `javax.xml.bind.Element`, das jedoch selbst nur ein leeres, sog. Marker-Interface, ist. Es zeigt an, dass es sich bei dieser Klasse um ein XML-Element handelt. Als nächstes wird das Interface `ItemListType` aus demselben Package erweitert, das die Methodensignaturen für die Getter und Setter deklariert, mit denen auf den Inhalt und die Attribute des zugrunde liegenden Elements zugegriffen werden kann. Dieses Interface hat folgenden Aufbau:

Listing 3.3: ItemListType.java

```
1 package at.gerstbach._2004.xmldatabinding.itemlist_1_0;
2
3 public interface ItemListType {
4     java.util.List getItem();
5 }
```

Für jedes Kind-Element von `ItemList` wurde ein Getter und ein Setter generiert. In diesem Fall nur `getItem()` mit dem Typ `java.util.List`, das für Sequenzen verwendet wird. Eine Methode `setItem` gibt es nicht, da die Klasse `List` ja bereits Methoden für das Verwalten der darin enthaltenen Objekte zur Verfügung stellt. Die Liste enthält Objekte vom Typ `ItemType`:

Listing 3.4: ItemType.java

```
1 package at.gerstbach._2004.xmldatabinding.itemlist_1_0;
2 public interface ItemType {
3     double getPrice();
4     void setPrice(double value);
5     java.lang.String getProductName();
6     void setProductName(java.lang.String value);
7     int getQuantity();
8     void setQuantity(int value);
9 }
```

Im Schema wurde das Element `item` mit dem Typ `itemType` definiert. Für diesen Typ generiert JAXB das Interface `ItemType`. Es besteht aus den Gettern und Settern für die einfachen Datentypen `productName`, `quantity` und `price`. Für das Mapping von XML Schema Datentypen werden sowohl primitive Java-Typen als auch Standard-Klassen verwendet. Der Abschnitt 3.2 erläutert genauer, wie JAXB das Mapping von XML nach Java löst.

Die erwähnten Klassen mit der Endung `-Impl` implementieren die Interfaces. Die Klassen werden jedoch niemals direkt verwendet sondern nur durch die Klasse `ObjektFactory` erzeugt.

### 3.1.2. Verwendung der generierten Klassen und Interfaces

Um mit JAXB Umwandlungen zwischen XML und Java-Objekten durchzuführen, verwendet man die Klasse `JAXBContext` und die generierte Klas-

se `ObjectFactory`. Der Quelltext von Beispiel 3.5 zeigt die genauere Vorgehensweise. Zuerst muss ein neues Objekt der Klasse `JAXBContext` erstellt werden, das mit dem Paketnamen des zu behandelnden XML-Elements verknüpft wird. Bei dieser Klasse handelt es sich um eine Factory mit den Methoden `createUnmarshaller()`, `createMarshaller()` und `createValidator()`, die Objekte zurückgeben, mit denen die Elemente umgewandelt und validiert werden können. Alle Operationen lassen sich jedoch nur auf Elemente anwenden, deren Namensraum mit jenem im `JAXBContext` übereinstimmt.

Die Methode `unmarshal` der Klasse `Unmarshaller` erzeugt aus einer XML-Quelle ein Java-Objekt. Als Quelle werden Objekte des Typs `java.io.File`, `java.io.InputStream`, `java.net.URL`, `org.xml.sax.SAXInputSource`, `org.w3c.dom.Node` und `javax.xml.transform.Source` unterstützt. Kann das XML nicht umgewandelt werden, wird eine `JAXBException` ausgelöst. Das neue Objekt kann nach einer Typ-Umwandlung wie ein herkömmlich erzeugtes Objekt verwendet werden. Im Beispiel werden alle Item-Objekte ausgegeben und dann der Preis um jeweils 20% erhöht.

Mit einem Objekt der Klasse `ObjectFactory` kann man neue Objekte erzeugen, die zu einem späteren Zeitpunkt in XML umgewandelt werden können. Wie schon erwähnt wurde die Klasse `ObjectFactory` automatisch generiert, da sie eine Liste aller Objekte enthalten muss, die sie erzeugen kann. Der Name des zu erzeugenden Objekts findet sich im Namen der create-Methode wieder: für `ItemList` als `createItemList()` oder für `ItemType` als `createItemType()`. Ist das Objekt einmal erzeugt, kann es aber wie üblich verwendet werden. Im Beispiel wird ein neues Item-Objekt angelegt und der Liste hinzugefügt.

Bevor die neue `ItemList` wieder in XML umgewandelt wird, kann man mit dem Objekt des Typs `Validator` und der Methode `validate` das Objekt gegen das Schema validieren lassen. Die Umwandlung zurück nach XML funktioniert nach dem selben Muster wie zuvor. Der Marshaller mit der Methode `marshall` schreibt das XML in einen DOM-Baum oder SAX-Handler oder in ein Objekt vom Typ `java.io.Writer`, `java.io.OutputStream` oder `javax.xml.transform.Result`.

Listing 3.5: ProcessItems.java

```
1 package at.gerstbach._2004.xml databinding.itemprocessor_1_0
   ;
2
3 import java.io.File;
4 import java.util.List;
5
6 import javax.xml.bind.JAXBContext;
7 import javax.xml.bind.Marshaller;
8 import javax.xml.bind.Unmarshaller;
9 import javax.xml.bind.Validator;
10
11 import at.gerstbach._2004.xml databinding.itemlist_1_0.
    ItemList;
12 import at.gerstbach._2004.xml databinding.itemlist_1_0.
    ItemType;
13 import at.gerstbach._2004.xml databinding.itemlist_1_0.
    ObjectFactory;
```

```
14
15 /**
16  * This example shows how to use JAXB to marshall,
17   * unmarshall and validate XML and how to create objects.
18  */
19 public class ProcessItems {
20     public static void main(String args[]) {
21         try {
22             JAXBContext jc = JAXBContext
23                 .newInstance("at.gerstbach._2004.xml databinding.
24                     itemlist_1_0");
25             ObjectFactory objFactory = new ObjectFactory();
26             Unmarshaller unmarshaller = jc.createUnmarshaller();
27             unmarshaller.setValidating(true);
28             Marshaller marshaller = jc.createMarshaller();
29             marshaller.setProperty(Marshaller.
30                 JAXB_FORMATTED_OUTPUT,
31                 new Boolean(true));
32             Validator validator = jc.createValidator();
33             ItemList items = (ItemList) unmarshaller.unmarshal(
34                 new File(
35                     "itemlist.xml"));
36             List itemList = items.getItem();
37             ItemType item;
38             //Print all items
39             System.out.println("Items:");
40             for (int i = 0; i < itemList.size(); i++) {
41                 item = (ItemType) itemList.get(i);
42                 System.out.println(item.getProductName() + ", "
43                     + item.getQuantity() + ", " + item.getPrice());
44             }
45             //increase price
46             for (int i = 0; i < itemList.size(); i++) {
47                 item = (ItemType) itemList.get(i);
48                 item.setPrice(item.getPrice() * 1.2);
49             }
50             //Insert new item
51             ItemType newItem = objFactory.createItemType();
52             newItem.setProductName("a new product");
53             newItem.setQuantity(1);
54             newItem.setPrice(9.50);
```

```
60     itemList.add(newItem);
61
62     System.out.println("\nValidator returned "
63         + validator.validate(items)
64         + " after changing and inserting items.");
65
66     //Print all items
67     System.out.println("\nnew Items:");
68     for (int i = 0; i < itemList.size(); i++) {
69         item = (ItemType) itemList.get(i);
70         System.out.println(item.getProductName() + ", "
71             + item.getQuantity() + ", " + item.getPrice());
72     }
73
74     System.out.println("\nXML-Output:");
75     marshaller.marshal(items, System.out);
76
77 } catch (Exception e) {
78     e.printStackTrace();
79 }
80 }
81 }
```

## 3.2. Mapping in JAXB

Die JAXB-Spezifikation beschreibt viele Regeln, wie Implementierungen die XML Komponenten in Java abbilden müssen. Beim Generieren der Java-Quellen werden diese Mappingregeln angewendet. Sie können jedoch auch an die Wünsche des Programmierers angepasst werden. Da JAXB nur XML Schema als Schema-sprache akzeptiert, ist es verständlich, dass sich die Mapping-Regeln stark an der Struktur von XML Schema orientieren. Die folgenden Abschnitte zeigen, wie das Standard-Mapping aussieht und in welchem Umfang es angepasst werden kann.

JAXB führt bei der Beschreibung des Mappings drei neue Begriffe ein, die in der Spezifikation und auch in dieser Arbeit an mehreren Stellen verwendet werden:

- **Java-Content-Interface:** komplexe Typdefinitionen werden an so genannte Java-Content-Interfaces gebunden. Diese Interfaces werden durch einen Namen und einer Zuordnung zu einem Paket identifiziert, sind von einem Basis-Interface abgeleitet und stellen eine Reihe von Java-Properties zur Verfügung, mit denen der Inhalt gelesen und verändert werden kann. Weitere Prädikate repräsentieren die inhaltlichen Bedingungen. Im Beispiel ist das Interface `ItemListType` ein Content-Interface.
- **Java-Property:** jede lokale Schema-Komponente wird innerhalb des Java-Content-Interfaces durch ein Property dargestellt. Es hat einen Namen und einen Basistyp. Wenn mehrere Werte in einer Kollektion gespeichert werden können, hat es einen Kollektionstyp und einige optionale Prädikate um z.B.

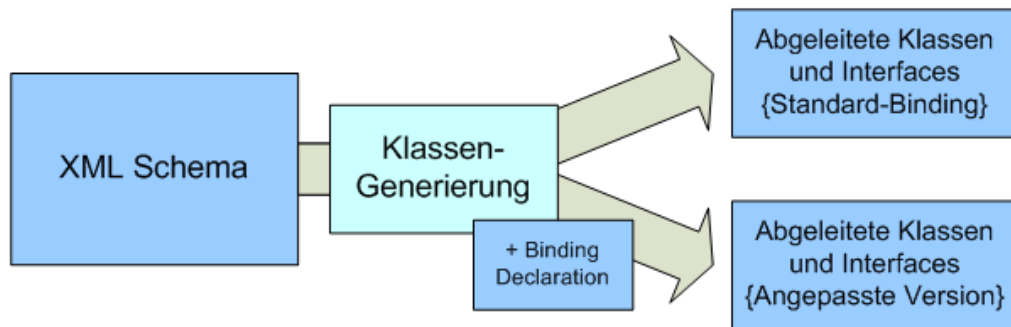


Abbildung 3.1.: Generieren mit und ohne Binding-Declaration

Standard- und Nullwerte speichern zu können. Die Realisierung in Java erfolgt durch Zugriffsmethoden, die sich an dem JavaBeans-Stil orientieren. Im Beispiel hat das Interface `ItemListType` ein Property mit dem Namen `item`. Das Interface `ItemType` hat die Properties `productName`, `quantity` und `price`.

- **Java-Element-Interface:** Elementdeklarationen werden durch Java-Element-Interfaces abgebildet. Elemente mit komplexem Typ unterscheiden sich dabei von Elementen mit einfachem Typ. Element-Interfaces werden im Abschnitt 3.2.4 näher erläutert. Im Beispiel ist `ItemList` ein Element-Interface.

**Anpassung.** Die JAXB-Spezifikation beschreibt, wie oben erwähnt, eine Menge von Regeln, wie XML Schema-Komponenten an Java-Komponenten gebunden werden. Dieses Standardverhalten kann jedoch an eigene Bedürfnisse angepasst werden. Diese Anpassung erfolgt durch die Verwendung einer eigenen XML-Sprache, der Binding-Declaration. Wird eine *Binding-Declaration* angewendet, entsteht eine angepasste Version, im anderen Fall kommt das Standard-Binding zur Geltung. Die Abbildung 3.1 erläutert dieses Konzept. Die Binding-Declaration kann entweder in einem separaten Dokument gespeichert sein oder innerhalb (*inline*) des XML-Schemas, in einem Annotation-Element (siehe Listing 3.6). Wird ein separates Dokument mit den Anpassungen angelegt, können Elemente darin mit einem XPath-Ausdruck an bestimmte Deklarationen im Schema-Dokument gebunden werden. Bei der Inline-Verwendung ist dies nicht notwendig, da die Anpassung schon innerhalb der zu verändernden Schema-Deklaration steht.

Für Elemente der Binding-Declaration gibt es unterschiedliche Gültigkeitsbereiche auf vier Ebenen:

- Global (global scope)
- Schema (schema scope)
- Definition (definition scope)
- Komponente (component scope)

Werte, die im globalen Gültigkeitsbereich definiert werden, gelten für alle Schema-Elemente und rekursiv auch für alle eingefügten und importierten Schemas. Sie werden in der Binding-Declaration mit dem Element `<globalBinding>` eingeleitet. Werte, die im Schema-Gültigkeitsbereich definiert werden, gelten für alle Schema-Elemente, die sich im Ziel-Namensraum des Schemas befinden. Sie werden mit dem Element `<schemaBindings>` eingeleitet. Werte im Definitionsgültigkeitsbereich gelten für alle Typdeklaration und alle globalen Deklarationen, Werte im Komponenten-Gültigkeitsbereich gelten für Elemente. Die beiden letzteren können entweder *inline* beim Element oder in einer externen Datei angegeben werden. In der externen Datei kann durch einen XPath-Ausdruck der Wert mit der zu verändernden Deklaration verknüpft. Weiters werden alle Deklarationen, beginnend beim globalen Gültigkeitsbereich, an die darunterliegende Schicht vererbt.

Listing 3.6: Inline Binding-Declaration

```
1 <xs:annotation>
2   <xs:appinfo>
3     <!-- hier stehen die Deklarationen -->
4   </xs:appinfo>
5 </xs:annotation>
```

Die folgenden Abschnitte erläutern die Mapping-Regeln und zeigen, in welcher Art die Anpassungen erfolgen können.

### 3.2.1. Einfache Typen

Das Mapping von einfachen Datentypen in XML Schema (Simple Types) kann sowohl durch primitive Java-Datentypen (z.B. für `xsd:int` und `xsd:float`) als auch durch Klassen (z.B. für `xsd:String` und `xsd:dateTime`) erfolgen. Die Tabelle 3.1 zeigt die genaue Zuordnung.

Insgesamt definiert XML Schema 15 Fassetten, die auf atomare Datentypen angewendet werden können. Fassetten wie `length`, `pattern`, `maxInclusive` und `minInclusive` können in Java jedoch nicht direkt durch die Sprache ausgedrückt werden. Eine Ausnahme bildet dabei die häufig eingesetzte Fasette `enumeration` (z.B. eine Zeichenkette, die nur aus den Werten „rot“, „grün“ oder „blau“ bestehen darf). Für diesen Fall ermöglicht JAXB die Verwendung einer typsicheren Aufzählung. Dafür werden alle möglichen Werte in Konstanten gespeichert. Falls einem Objekt dieses Typs ungültige Werte zugewiesen werden, wird eine `IllegalArgumentException` ausgelöst. Ein Beispiel so einer Klasse wird in Abschnitt 4.2.3 erläutert.

Zusätzlich zu den atomaren Typen kennt XML Schema auch das Konzept der Listentypen und Vereinigungstypen. Listentypen werden in einem Array oder in einer Kollektion gespeichert. Standardmäßig ist der Typ der Kollektion `java.lang.List`, diese Voreinstellung kann jedoch in der Binding-Declaration durch die Angabe des Attributs `collectionType` festgelegt werden. Für Vereinigungstypen wird die erste gemeinsame Oberklasse verwendet, also mindestens die Klasse `Object`.

Wenn gewünscht, können auch eigene Java-Klassen definiert werden, die als Container für bestimmte einfache Datentypen verwendet werden sollen. Dabei

HXML Schema Datentyp	Java-Datentyp
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	java.util.Calendar
xsd:date	java.util.Calendar
xsd:anySimpleType	java.lang.String

Tabelle 3.1.: Java-Mapping von Eingebauten Typen in XML Schema

lassen sich Parse-Methoden angeben, die die Umwandlung von z.B. String in das eigene Format durchführen.

### 3.2.2. Komplexe Typen

Komplexe Datentypen werden mit Hilfe von Java-Properties abgebildet. Der Inhalt eines komplexen Typs wird in die jeweiligen Java-Repräsentationen umgewandelt und als Attribut einem Java-Content-Interface hinzugefügt, das dann das Gegenstück zur Schema-Komponente ist. Dieser Prozess wird für die Teile rekursiv wiederholt, bis für das gesamte Schema eine Java-Repräsentation entstanden ist. Für jede Deklaration eines komplexen Typs entsteht also ein Java-Content-Interface. Im Beispiel erhält das Content-Interface `ItemListType` das Property `item` (also die Methode `getItem()`) vom Typ `java.util.List`.

Für benannte komplexe Typen bekommt dieses Java-Content-Interface den Namen des Typs im Schema. Wurde der komplexe Typ anonym definiert, hängt JAXB in diesem Fall an den Namen die Endung `-Type` an. Wäre im Beispiel 3.1 die Deklaration des Typs vom Element `itemList` innerhalb und ohne Angabe eines Namens durchgeführt worden, hätte JAXB das Content-Interface deshalb ebenfalls `ItemListType` genannt.

### 3.2.3. Elementgruppen

Eine Elementgruppe wird normalerweise nicht an ein eigenes Java-Content-Interface gebunden. Die einzelnen Properties sind stattdessen im Content-Interface des komplexen Typs zu finden. Wenn eine Elementgruppe zweimal referenziert wird, bedeutet dies, dass die entsprechenden Properties in jeweils beiden Interfaces erzeugt werden. Der Grund dafür liegt in der XML Schema Spezifikation, die Elementgruppen nicht als Teil der Ableitungshierarchie sieht.

Bei einfachen Elementgruppen ist dieses Verhalten meist kein Problem. Wenn eine Elementgruppe jedoch weitere Komponenten enthält, würde dies bedeuten, dass alle enthaltenen Komponenten in jedem referenzierenden Element generiert werden würden. Da daraus doppelter Quellcode entstehen würde, kann dieses Verhalten durch Anpassung abgeändert werden, sodass eine Elementgruppe an ein eigenes Java-Content-Interface gebunden wird.

### 3.2.4. Elemente

Eine Element-Deklaration in XML Schema wird durch das schon kurz erläuterte Java-Element-Interface abgebildet. Dieses Interface kann unterschiedlich aufgebaut sein. Ist der Typ der Elementdeklaration ein komplexer Typ, dann erweitert das Interface das Content-Interface, welches den Typ des Elements repräsentiert. Ist der Typ ein einfacher Typ, dann bekommt das generierte Element-Interface ein Property mit dem Namen `value`.

Zusätzlich wird das Interface `javax.xml.bind.Element` erweitert, das eine Unterscheidung zwischen Typen und Elementen ermöglicht. Im Item-Beispiel ist das Interface `ItemList` ein Element-Interface. Es implementiert das Interface `ItemListType` und das leere Marker-Interface `Element`. Weitere Unterscheidungen betreffen u.a. den Gültigkeitsbereich, die Wertebereich-Beschränkung und Nullwertfähigkeit.

### 3.2.5. Attribute und Attributgruppen

Die Deklaration eines Attributes wird auf ein Java-Property abgebildet, und wird, wie bei komplexen Typen, durch Getter und Setter realisiert. Der Typ entspricht den Konventionen für einfache Typen. Für Attributgruppen-Definitionen gibt es keine Entsprechung. Sie werden dort, wo sie referenziert werden, wie einzelne Attribute abgebildet.

### 3.2.6. Inhaltsmodell

Die Elementgruppen `sequence`, `choice` und `all` können mit JAXB auf zwei unterschiedlichen Arten an Java gebunden werden, wobei die erste Art der Standardeinstellung entspricht:

**Element Binding Style.** Bei dieser Art werden Elementdeklarationen in Properties abgebildet. Je nach Aufbau und Art der Elementgruppe werden unterschiedliche Ansätze gewählt:

- **General-Content-Property:** bei diesem Ansatz werden alle Elemente in einer Liste verwaltet. Dadurch kann zusätzlich zum Element auch die Reihenfolge gespeichert werden. Gemischter Inhalt kann ebenfalls leicht dargestellt werden, da auf einfache Art Strings der Liste hinzugefügt werden können. Dieser Ansatz wird bei gemischtem Inhalt angewendet und bei allen Elementgruppen, deren Attribut `maxOccurs` größer ist als eins.
- **Wildcards** werden durch ein Property mit dem Namen `any` dargestellt. Ist das Attribut `maxOccurs` größer als eins, wird eine Collection verwendet.
- **Bindung der Elementdeklarationen an Content Properties:** bei diesem Ansatz wird jede Elementdeklaration einem Property im Content Interface zugeordnet. Die Elemente sind demnach flach und ohne jegliche Reihenfolge in dem Interface vorhanden. Eine allenfalls nötige Reihenfolge muss von der JAXB-Implementierung durch Berechnung hergestellt werden. Falls dies nicht möglich ist, wird eine Exception ausgelöst. Dieser Ansatz wird auf die restlichen Elementgruppen angewendet: wenn das Attribut `maxOccurs` der Elementgruppe den Wert eins hat und keine Wildcard Komponente oder Komponente mit gemischtem Inhalt vorliegt.

Generell kann der Typ von Listen durch Anpassung verändert werden. Standardmäßig ist eine Liste vom Typ `java.util.List` voreingestellt. Es können aber auch andere, von `Collection` abgeleitete Klassen, oder auch Arrays verwendet werden.

**Model Group Binding Style.** Bei dieser Art wird nicht eine Liste verwendet, sondern jede Elementgruppe wird in das Java-Content-Interface aggregiert. Der Vorteil dieser Variante ist, dass die Semantik der Elementgruppen erhalten bleibt, wodurch der Benutzer beim Erbauen von gültigen Strukturen unterstützt wird. Der Nachteil ist jedoch, dass sehr viele Java-Content-Interfaces mit langen Bezeichnern generiert werden. Hier wird die Umbenennung dieser Interfaces durch Anpassung fast zu einem Muss.

### 3.2.7. Bezeichner und Namensräume

Die XML 1.0 Spezifikation definiert alle möglichen Bezeichner für Komponenten. Nur ein Teil dieser XML-Namen sind jedoch in Java erlaubt. Deswegen widmet sich ein eigener Abschnitt in der JAXB-Spezifikation dem Thema Bezeichner und deren Umwandlung zwischen XML und Java. Ein weiterer Teil der Spezifikation behandelt die Bedeutung von Namensräumen und Paketen.

**Bezeichner.** In Java gibt es für Bezeichner 3 einfache Regeln:

- Namen von Klassen und Interfaces beginnen mit einem Großbuchstaben, gefolgt von weiteren Zeichen, die Ziffern sowie Groß- und Kleinbuchstaben sein können. Großbuchstaben werden innerhalb des Namens eingefügt, um bei zusammengesetzten Worten die Lesbarkeit zu verbessern.

- Namen von Methoden beginnen mit einem Kleinbuchstaben und werden sonst wie Klassen- und Interfacenamen gebildet.
- Konstanten werden nur mit Großbuchstaben geschrieben, einzelne Wörter durch den Unterstrich ('\_') getrennt.

Die Definition von XML-Namen ist um einiges offener, so dürfen sie z.B. diverse Interpunktionszeichen enthalten, die in Java nicht erlaubt sind. Um trotzdem eine Umwandlung zu ermöglichen, werden XML-Namen nach folgendem Schema abgeändert:

1. Interpunktionszeichen am Anfang und am Ende werden entfernt
2. Die Zeichenkette wird in eine Wortliste aufgespaltet (Trennzeichen sind Interpunktionszeichen und das direkte Aufeinanderfolgen von Groß-/Kleinbuchstaben bzw. Ziffern)
3. Beginnt ein Wort mit einem Kleinbuchstaben wird der erste Buchstabe in einen Großbuchstaben umgewandelt
4. Je nach Art des Java-Bezeichners wird der resultierende Name durch Verkettung der einzelnen Worte erreicht; bei Methoden wird der Präfix (get, set, usw.) vorangestellt, bei Konstanten werden alle Buchstaben groß geschrieben und durch Unterstriche getrennt

Die Tabelle 3.2 erläutert dieses Konzept anhand von 4 Beispielen.

XML-Name	Klassen-Name	Methoden-Name	Konstanten-Name
mixedCase	MixedCase	getMixedCase	MIXED_CASE
Answer42	Answer42	getAnswer42	ANSWER_42
with-dashes	WithDashes	getWithDashes	WITH_DASHES
other_chars	OtherChars	getOtherChars	OTHER_CHARS

Tabelle 3.2.: XML Namen und Java-Bezeichner

Bei diesem Konzept können Konflikte auftreten, da zwei unterschiedliche XML-Namen zu zwei gleichen Java-Bezeichnern werden können (z.B. würden die Namen „conflict-name“ und „conflictName“, falls es sich um Klassen handelt, beide in „ConflictName“ umgewandelt werden). Ein weiteres Problem entsteht bei der Verwendung von reservierten Java-Bezeichnern. In beiden Fällen wird das Generieren des Java-Quelltexts mit einem Fehler abgebrochen. Der Konflikt muss durch eine manuelle Anpassung des Mappings behoben werden.

**Namensräume.** Der Ziel-Namensraum (target namespace) in einem XML Schema erfüllt denselben Zweck wie ein Paket in Java. In XML Schema lassen sich Type-Definitionen, benannte Model-Groups, globale Elementdeklarationen und globale Attribut-Deklarationen einem Namensraum zuordnen. In Java erhalten Klassen und Interfaces durch Pakete eine eindeutige Zugehörigkeit. Deswegen entspricht in JAXB der Namensraum dem Paketnamen.

## 4. Beispiel Active XML Schema

Dieses Kapitel behandelt ein komplexeres Anwendungsbeispiel für JAXB als in Teil 1 dieser Arbeit: Active XML Schema [SB02]. Es wird das Beispiel, die Generierung der Klassen und das Mapping erläutert und ein Anwendungsbeispiel aufgezeigt.

### 4.1. Active XML Schema im Überblick

XML ermöglicht einfachen Datenaustausch im Web und steht damit im Zentrum vieler e-Business-Konzepte. Jedoch sind Dokumente oft wechselseitig voneinander änderungsabhängig. Diese Änderungsabhängigkeiten sind normalerweise lose gekoppelt, da sie unter der Verantwortung der verschiedenen Kommunikationspartner stehen. Durch aktives Verhalten von Dokumenten können Änderungsabhängigkeiten automatisch berücksichtigt werden. Active XML Schema [SB02] weist einen konzeptionellen Ansatz für aktives Verhalten von XML Dokumenten auf und bietet die Möglichkeit, Dokumente als autonome Einheiten, die reaktives Verhalten aufweisen, zu modellieren.

In diesem Teil der Arbeit wird das Beispiel *Active Jobs* aus [SB02] umgesetzt. Relevante Teile des Quellcodes werden im Text oder im Anhang eingebunden.

### 4.2. Generierung und Mapping

#### 4.2.1. Fehler und Warnungen

Die Beispiele aus Teil 1 ließen sich alle ohne weitere Konfiguration generieren. Bei Active XML Schema ist das nicht mehr der Fall. Beim Versuch das Schema `ActiveJobs.xsd` zu parsen kommt es zu folgender Fehlermeldung:

```
>xjc ActiveJobs.xsd
parsing a schema...
[ERROR] In "strict" mode, the following schema feature is not
  allowed (See App E.2). Use the "-extension" switch:
  "abstract" attribute of <element> is not supported
  line 23 of FR-EventTypes.xsd
```

Failed to parse a schema.

Im Appendix E.2 der JAXB-Spezifikation sind einige XML Schema-Konzepte angeführt, die von JAXB-Implementierungen nicht zwingend unterstützt werden müssen. Die Implementierung von Sun bietet einige dieser optionalen Funktionen als Erweiterung an. Sie lassen sich beim Aufruf von `xjc` durch die Angabe

des Parameters `-extension` aktivieren. Beim neuerlichen Versuch das Schema zu parsen, erscheint der vorige Fehler nicht mehr, es tritt jedoch eine große Anzahl von anderen Warnungen und Fehlern auf. Zwei dieser Fehler werden näher erläutert und deren Korrektur erklärt.

```
[WARNING] warning: "final" attribute of <complexType> is not
supported
line 122 of FR-EventTypes.xsd
[WARNING] warning: <key> identity constraint will be ignored by
JAXB validation
line 31 of MS-ActiveDefinitions_Core.xsd
...
[ERROR] A class/interface with the same name
"at.ac.tuwien.big.axs.samples.jobs.Job" is already in use.
line 12 of Jobs.xsd
```

Die Warnungen machen deutlich, dass die aktuelle Version von JAXB weder mit dem Attribut *final* noch mit *Keys* umgehen kann. Eine Warnung verhindert aber noch nicht das erfolgreiche Generieren der Klassen. Die Attribute und Keys werden lediglich ignoriert.

Anders ist das bei den Fehlern. Hier handelt es sich um Namenkonflikte, die aus den beschränkten Möglichkeiten der Namensgebung in Java resultieren, wie sie schon im Abschnitt 3.2.7 erklärt wurden. Der exemplarische dargestellte Fehler entsteht durch eine zweimalige Verwendung des Bezeichners `Job` in `Jobs.xsd` (Listing 4.1). Beim Binding würde das sowohl das Element `job` (kleingeschrieben), wie auch der komplexe Typ `Job` (großgeschrieben) in eine Datei `Job.java` umgewandelt werden.

Listing 4.1: Ausschnitt aus `Jobs.xsd`

```
12 <xs:element name="job" type="j:Job"/>
13 <xs:complexType name="Job">
14 <xs:sequence>
15 <xs:element name="field" type="xs:string"/>
16 <xs:element name="title" type="xs:string"/>
17 <xs:element name="appDeadline" type="xs:date"/>
18 </xs:sequence>
19 <xs:attribute name="id" type="xs:ID"/>
20 </xs:complexType>
```

Da diese beiden Klassen nicht den selben Namen besitzen dürfen, muss dieser Namenskonflikt durch die Verwendung einer *Binding-Declaration* beigelegt werden. Listing A.1 zeigt eine externe Binding-Declaration für die Datei `Jobs.xsd` und jene, die von ihr eingebunden werden. In Zeile 32 wird der Abschnitt definiert, der sich nur auf die Datei `Jobs.xsd` bezieht und in Zeile 38 derjenige, der nur für das Element mit dem Namen `job` gilt. Die folgende Zeile enthält dann die Anweisung zur Umbenennung der Klasse in `JobElement`. Wird die Generierung nun durchgeführt, lässt es sich erstmals ohne Fehler beenden und es entsteht für den komplexen Typ die Klasse `Job` und für das Element die Klasse `JobElement`. Listing A.1 enthält noch weitere solche Umbenennungen, um alle Namenskonflikte aufzulösen.

### 4.2.2. Erweiterungen

Neben den Fehlern, die das Generieren der Klassen verhindern, wird nun noch die Anwendung herstellerabhängiger Erweiterungen besprochen. Eine dieser Erweiterungen betrifft *Type Substitution*, die von der JAXB Spezifikation nicht unterstützt wird. Im Beispiel-XML-Dokument `ActiveJobs-adt.xml` (Listing 4.2) wird Type Substitution jedoch häufig eingesetzt, z.B. beim Element `eventType`. Das Element `eventType` ist eigentlich vom Typ `EventType`; im XML-Dokument wird es jedoch vom abgeleiteten Typ `OperationEvtTp` überschrieben. Um dieses Verhalten im Binding korrekt abzubilden, muss in der Binding-Declaration die entsprechende Zeile in den globalen Einstellungen eingefügt werden (Zeile 20 in A.1). Diese Erweiterung der JAXB-Implementierung ist noch experimentell und wird voraussichtlich erst in JAXB 2.0 von der Spezifikation standardisiert werden [Sun03a].

Listing 4.2: Ausschnitt aus `ActiveJobs-adt.xml`

```

45     <actm:eventType name="j:TExecAnnounceEvtTp"
46         timeSpec="after" xsi:type="actm:OperationEvtTp">
47         <actm:operationRef interfaceNm="j:JobAnnounce"
48             operationNm="announce">
49             <actm:parameter type="j:Job"/>
48         </actm:operationRef>
49     </actm:eventType>

```

Eine andere mögliche Erweiterung betrifft *Keys*, die in Active XML Schema ebenfalls häufig eingesetzt werden. Keys werden jedoch in JAXB 1.0 noch nicht unterstützt, weder in der Spezifikation noch als herstellerabhängige Erweiterung. Eine entsprechende Fehlermeldung deutet auf diesen Umstand hin. Es kann jedoch erwartet werden, dass in einer zukünftigen Version Keys bei der Validierung der Objekte beachtet werden, da das Konzept in vielen Dokumenttypdefinitionen wesentlich ist.

```

[WARNING] warning: <key> identity constraint will be ignored by
JAXB validation
line 31 of MS-ActiveDefinitions_Core.xsd

```

```

[WARNING] warning: <keyref> identity constraint will be ignored by
JAXB validation
line 38 of MS-ActiveDefinitions_Core.xsd

```

### 4.2.3. Typsichere Aufzählung

Die JAXB-Spezifikation sieht für die Fassade `Enumeration` eine typsichere Aufzählung vor, die durch eine eigene Klasse realisiert wird. In Active XML Schema kommen einige dieser Typen vor, stellvertretend wird hier der Typ `MutationOp` behandelt. Dieser ist in der Datei `MS-ActiveDefinitions_Core.xsd` wie folgt definiert:

```

<xs:simpleType name="MutationOp">
    <xs:restriction base="xs:string">

```

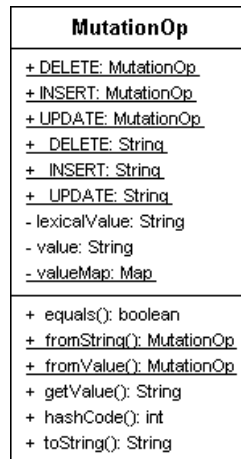


Abbildung 4.1.: Klassendiagramm MutationOp

```

<xs:enumeration value="insert" />
<xs:enumeration value="update" />
<xs:enumeration value="delete" />
</xs:restriction>
</xs:simpleType>

```

Damit der Compiler daraus eine typsichere Aufzählung generiert, müssen in der Binding-Declaration die Basistypen aller gewünschten Aufzählungen angegeben werden (siehe Zeile 18 in Listing A.1). Beim Generieren entsteht nun die Java-Klasse `MutationOp`, die im Anhang als Listing A.2 zu sehen ist. In der Klasse wurden für die erlaubten Zeichenketten *insert*, *update* und *delete* jeweils eigene Konstanten (`_INSERT` usw.) definiert. Durch die Verwendung dieser Konstanten und der Methode `fromString()` kann ein gültiges Objekt erzeugt werden, z.B. `MutationOp.fromString(MutationOp._UPDATE)`, bei einem nicht erlaubten Wert wird eine `IllegalArgumentException` ausgelöst, wie z.B. beim Aufruf von `enum = MutationOp.fromValue("not defined");` Listing A.3 zeigt diese Aufrufe, eingebettet in einem kurzen Programm. Zur Veranschaulichung zeigt Abbildung 4.1 ein UML-Klassendiagramm der generierten Klasse.

### 4.3. UML Klassendiagramm

Nachdem alle Einträge der Binding-Declaration erläutert worden sind, ist nun das Generieren der Klassen möglich. Insgesamt werden durch den JAXB-Compiler 177 Klassen und Interfaces generiert. Abbildung 4.2 zeigt einen Ausschnitt aus dem UML-Klassendiagramm der generierten Klassen, dieser wird in den folgenden Absätzen erläutert.

Ausgangspunkt ist das Element `activeDocumentType`. Da dieses Element durch einen anonymen komplexen Typ definiert wurde, nennt JAXB die Klasse `ActiveDocumentTypeType`. Die im Element enthaltenen Elemente `passiveBehavior` und `activeBehavior` sind ebenfalls anonym definiert, sie werden deswegen als innere Klassen erzeugt. Das Element `activeBehaviour` besitzt ein Element `eventType` mit einem Attribut `name` vom Typ `QName`.

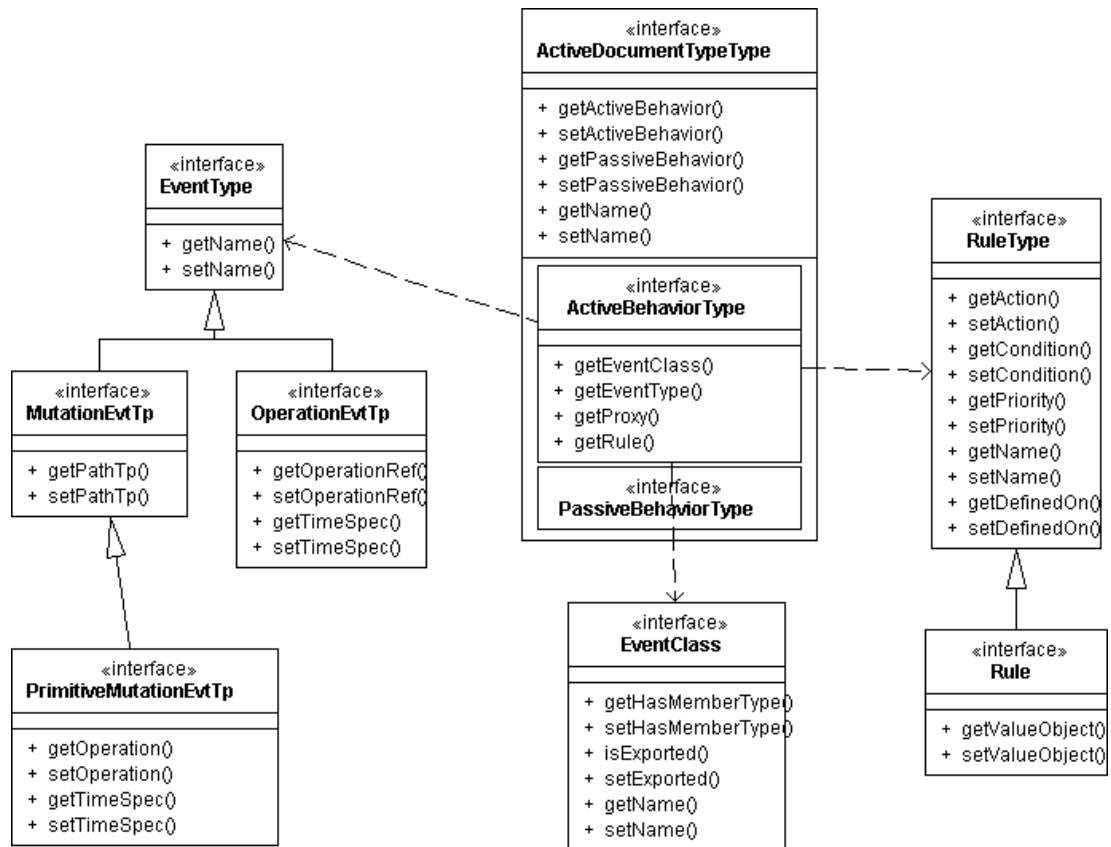


Abbildung 4.2.: Auswahl aus dem Klassendiagramm der generierten Interfaces von *Active XML*

Dieses Attribut referenziert ein Element vom Typ `eventType`. Die im Diagramm eingezeichnete Abhängigkeit zwischen `ActiveBehaviourType` und `EventType` wurde manuell eingefügt, da Keys in JAXB nicht unterstützt werden. Die Methode `getEventType()` liefert lediglich ein Objekt vom Typ `javax.xml.namespace.QName` zurück.

Vererbungsbeziehungen können im Klassendiagramm gut dargestellt werden: der komplexe XML Schema-Typ `OperationEvtTp` erweitert `EventType`, im Klassendiagramm ist diese Vererbung 1:1 vorhanden. Die weiteren Methoden von `ActiveBehaviorType` sind `getEventClass()`, `getProxy()` und `getRule()`. Sie sind nach dem selben Prinzip wie `getEventType()` aufgebaut. Auch hier ist die Abhängigkeit im Klassendiagramm manuell hinzugefügt worden.

**Teil III.**  
**Zusammenfassung**

## 5. Zusammenfassung

*XML Data Binding* ermöglicht auf eine neuartige Weise die Programmierung von Anwendungen, die XML benutzen. Da kein Spezialwissen zu XML und Programmierschnittstellen wie DOM und SAX benötigt wird, können Programmierer schnell und einfach XML-Daten in Anwendungen integrieren, diese bearbeiten, validieren und wieder ablegen. Da alle XML-spezifischen Teile in einer eigenen Software ausgelagert sind, kann durch XML Data Binding die Entwicklungszeit und die Fehleranfälligkeit verringert werden. Durch die Aufteilung entsteht eine neue Aufgabe im Entwicklerteam: neben dem Schema-Design und der Programmierung der Anwendungslogik müssen die Binding-Klassen generiert werden. Dazu ist ein fundiertes Wissen über XML, die verwendete Schema-Sprache, das Data Binding-Werkzeug und die Zielanwendung notwendig. Denn erst wenn die generierten Klassen gut verständlich und einfach zu verwenden sind, werden die Vorteile von XML Data Binding sichtbar.

Mit *JAXB* existiert ein umfangreiches Werkzeug für XML Data Binding. Aus einfach aufgebauten Schema-Dateien lassen sich bereits nach kurzer Einarbeitungszeit praktische Klassen generieren, die den Blick von der komplizierten XML-Programmierung zurück auf die Anwendungslogik möglich macht. Je komplexer das Schema jedoch wird, desto schwieriger fallen die Entscheidungen, in welcher Art und Weise das Binding angepasst werden soll. Bei dem komplexen Beispiel *Active XML Schema* erzeugt JAXB eine große Anzahl an Interfaces und Klassen, die einen nicht vernachlässigbar hohen Einarbeitungsaufwand mit sich bringen. Dafür kann sich der Programmierer auf diese Klassen konzentrieren und muss nicht mehr das gesamte Schema im Kopf haben. JAXB bietet schon in der Version 1.0 einen sehr großen Funktionsumfang. Die fehlende Unterstützung von Keys schränkt mögliche Anwendungsfälle jedoch stark ein.

Bereits vor JAXB existierten schon viele Werkzeuge, die hier nicht näher erläutert wurden. [Sos03] beschreibt neben JAXB auch die Vor- und Nachteile anderer Data Binding-Werkzeuge, wie *Castor*, *JBind*, *Quick* und *Zeus*. Eine sehr umfangreiche und stets aktuelle gehaltene Auflistung diverser XML-Anwendungen gibt [Bou04].

# A. Listings für Active XML Schema

## A.1. Binding-Declaration

Listing A.1: binding.xjb

```

1 <jxb:bindings version="1.0" xmlns:jxb="http://java.sun.com/
  xml/ns/jaxb" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:actm="http://big.tuwien.ac.at/axs/metaschema/1.0"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  jxb:extensionBindingPrefixes="xjc">
2 <!-- ##### METASHEMA ##### -->
3 <jxb:bindings schemaLocation="MS-ActiveDefinitions_Core.
  xsd" node="/xs:schema">
4 <jxb:schemaBindings>
5 <!-- Specify package name-->
6 <jxb:package name="at.ac.tuwien.big.axs.metaschema"/>
7 </jxb:schemaBindings>
8 <!-- Resolve collision between case-sensitive XML names
  and case insensitive java classes. -->
9 <jxb:bindings node="//xs:element[@name='eventClass']">
10 <jxb:class name="EventClassElement"/>
11 </jxb:bindings>
12 <jxb:bindings node="//xs:element[@name='eventType']">
13 <jxb:class name="EventTypeElement"/>
14 </jxb:bindings>
15 </jxb:bindings>
16 <!-- ##### FRAMEWORK ##### -->
17 <jxb:bindings schemaLocation="FR-EventTypes.xsd" node="/
  xs:schema">
18 <jxb:globalBindings typesafeEnumBase="xs:string">
19 <!-- enable vendor-extension for type substitution --
  >
20 <xjc:typeSubstitution type="complex"/>
21 </jxb:globalBindings>
22 <jxb:schemaBindings>
23 <!-- Specify package name-->
24 <jxb:package name="at.ac.tuwien.big.axs.framework"/>
25 </jxb:schemaBindings>
26 <!-- Resolve collision between case-sensitive XML names
  and case insensitive java classes. -->
27 <jxb:bindings node="//xs:element[@name='eventClass']">
28 <jxb:class name="EventClassElement"/>
29 </jxb:bindings>
30 </jxb:bindings>

```

```

31 <!-- ##### JOBS ##### -->
32 <jxb:bindings schemaLocation="Jobs.xsd" node="/xs:schema"
   >
33   <jxb:globalBindings>
34     <!-- enable vendor-extension for type substitution --
   >
35     <xjc:typeSubstitution type="complex"/>
36   </jxb:globalBindings>
37   <!-- Resolve collision between case-sensitive XML names
   and case insensitive java classes. -->
38   <jxb:bindings node="//xs:element[@name='job']">
39     <jxb:class name="JobElement"/>
40   </jxb:bindings>
41 </jxb:bindings>
42 </jxb:bindings>

```

## A.2. Typsichere Aufzählung

Listing A.2: Enumerations-Klasse MutationOp

```

1 //
2 // This file was generated by the Java™ Architecture for
  XML Binding (JAXB) Reference Implementation, v1.0.2-b15-
  fcs
3 // See <a href="http://java.sun.com/xml/jaxb">http://java.
  sun.com/xml/jaxb</a>
4 // Any modifications to this file will be lost upon
  recompilation of the source schema.
5 // Generated on: 2004.09.29 at 09:10:19 CEST
6 //
7
8
9 package at.ac.tuwien.big.axs.metaschema;
10
11
12 /**
13  * Java content class for MutationOp.
14  * <p>The following schema fragment specifies the expected
   content contained within this java content object.
15  * <p>
16  * <pre>
17  * <restriction base="{http://www.w3.org/2001/XMLSchema}
   string">
18  *   <enumeration value="insert"/>
19  *   <enumeration value="update"/>
20  *   <enumeration value="delete"/>
21  * </restriction>
22  * </pre>
23  *
24  */
25 public class MutationOp {

```

```
26
27     private final static java.util.Map valueMap = new java.
           util.HashMap();
28     public final static java.lang.String _INSERT = com.sun.
           xml.bind.DatatypeConverterImpl.installHook("insert")
           ;
29     public final static at.ac.tuwien.big.axs.metaschema.
           MutationOp INSERT = new at.ac.tuwien.big.axs.
           metaschema.MutationOp(_INSERT);
30     public final static java.lang.String _UPDATE = com.sun.
           xml.bind.DatatypeConverterImpl.installHook("update")
           ;
31     public final static at.ac.tuwien.big.axs.metaschema.
           MutationOp UPDATE = new at.ac.tuwien.big.axs.
           metaschema.MutationOp(_UPDATE);
32     public final static java.lang.String _DELETE = com.sun.
           xml.bind.DatatypeConverterImpl.installHook("delete")
           ;
33     public final static at.ac.tuwien.big.axs.metaschema.
           MutationOp DELETE = new at.ac.tuwien.big.axs.
           metaschema.MutationOp(_DELETE);
34     private final java.lang.String lexicalValue;
35     private final java.lang.String value;
36
37     protected MutationOp(java.lang.String v) {
38         value = v;
39         lexicalValue = v;
40         valueMap.put(v, this);
41     }
42
43     public java.lang.String toString() {
44         return lexicalValue;
45     }
46
47     public java.lang.String getValue() {
48         return value;
49     }
50
51     public final int hashCode() {
52         return super.hashCode();
53     }
54
55     public final boolean equals(java.lang.Object o) {
56         return super.equals(o);
57     }
58
59     public static at.ac.tuwien.big.axs.metaschema.
           MutationOp fromValue(java.lang.String value) {
60         at.ac.tuwien.big.axs.metaschema.MutationOp t = ((at
           .ac.tuwien.big.axs.metaschema.MutationOp)
```

```
        valueMap.get(value));
61     if (t == null) {
62         throw new java.lang.IllegalArgumentException();
63     } else {
64         return t;
65     }
66 }
67
68 public static at.ac.tuwien.big.axs.metaschema.
    MutationOp fromString(java.lang.String str) {
69     return fromValue(str);
70 }
71
72 }
```

Listing A.3: Anwendung einer Enumerations-Klasse

```
1 package at.ac.tuwien.big.axs.metaschema;
2
3 /**
4  * Class demonstrating typesafe enumeration in JAXB.
5  *
6  * @author Peter Gerstbach, 2004-09-29
7  *
8  */
9 public class EnumTest {
10
11     public static void main(String[] args) {
12         MutationOp enum;
13         try {
14             // assign correct Value
15             enum = MutationOp.fromString(MutationOp._UPDATE
16                 );
17             System.out.println("Enum-Value:␣" + enum);
18         } catch (IllegalArgumentException e) {
19             System.out.println("Exception␣occured:␣" + e);
20         }
21         try {
22             // assign incorrect Value
23             enum = MutationOp.fromValue("not␣defined");
24             System.out.println("Enum-Value:␣" + enum);
25         } catch (IllegalArgumentException e) {
26             System.out.println("Exception␣occured:␣" + e);
27         }
28     }
29 }
```

# Abbildungsverzeichnis

2.1. Konzepte des XML Data Bindings . . . . .	8
3.1. Generieren mit und ohne Binding-Declaration . . . . .	17
4.1. Klassendiagramm <code>MutationOp</code> . . . . .	26
4.2. Auswahl aus dem Klassendiagramm der generierten Interfaces von <i>Active XML</i> . . . . .	27

# Tabellenverzeichnis

3.1. Java-Mapping von Eingebauten Typen in XML Schema . . . . .	19
3.2. XML Namen und Java-Bezeichner . . . . .	22

# Listings

2.1. person.xml . . . . .	6
2.2. person.xsd . . . . .	7
3.1. itemlist.xsd . . . . .	12
3.2. ItemList.java . . . . .	12
3.3. ItemListType.java . . . . .	13
3.4. ItemType.java . . . . .	13
3.5. ProcessItems.java . . . . .	14
3.6. Inline Binding-Declaration . . . . .	18
4.1. Ausschnitt aus Jobs.xsd . . . . .	24
4.2. Ausschnitt aus ActiveJobs-adt.xml . . . . .	25
A.1. binding.xjb . . . . .	30
A.2. Enumerations-Klasse <code>MutationOp</code> . . . . .	31
A.3. Anwendung einer Enumerations-Klasse . . . . .	33

# Literaturverzeichnis

- [Bou04] BOURRET, RONALD: *XML Data Binding Resources*.  
<http://www.rpbouret.com/xml/XMLDataBinding.htm> (Stand: September 2004), 2004.
- [Exo04] EXOLAB GROUP: *Castor Reference, The XML Framework API*.  
<http://www.castor.org/xml-framework.html> (Stand: September 2004), 2004.
- [FV03] FIALLI, JOSEPH und SEKHAR VAJJHALA: *Java Architecture for XML Binding (JAXB) Specification 1.0*.  
<http://java.sun.com/xml/downloads/jaxb.html> (Stand: September 2004), 2003.
- [McL02] MCLAUGHLIN, BRETT: *Java & XML Data Binding*. O'Reilly & Associates, 2002.
- [OM03] ORT, ED und BHAKTI MEHTA: *Java Architecture for XML Binding (JAXB)*.  
<http://java.sun.com/developer/technicalArticles/WebServices/jaxb/> (Stand: September 2004), 2003.
- [SB02] SCHREFL, MICHAEL und MARTIN BERNAUER: *Active XML Schemas*. In: *Proceedings of the International Workshop on Conceptual Modeling Approaches for e-Business (eCOMO)*, 2002.
- [Sos03] SOSNOSKI, DENNIS M.: *Data binding, Part 1: Code generation approaches – JAXB and more*.  
<http://www-106.ibm.com/developerworks/xml/library/x-databdopt/> (Stand: September 2004), Jänner 2003.
- [Sun03a] SUN MICROSYSTEMS: *Java Architecture for XML Binding Readme*.  
<http://java.sun.com/webservices/docs/1.3/jaxb/> (Stand: September 2004), 2003.
- [Sun03b] SUN MICROSYSTEMS: *Java Web Services Developer Pack*.  
<http://java.sun.com/webservices/jwsdp/> (Stand: September 2004), 2003.