Dokumenttypen mit XML-Schema definieren

Struktur im Griff

Der XML-Schema-Standard räumt mit den Schwächen von DTDs (Document Type Definitions) auf. XML-Schemas beschreiben die Struktur von Dokumenten exakter und verwenden im Gegensatz zu DTDs selbst XML. Die Java-Version des Xerces-Parsers beherrscht diese neue Technologie. Bernhard Bablok

Die Struktur von XML-Dokumenten wird seit jeher durch DTDs (Document Type Definitions) festgelegt. XML-Schema soll diese nun ablösen und in der Praxis ersetzen. Mit der XML-Schema-Recommendation [1] hat das W3C die Voraussetzung dafür geschaffen. Obwohl die Recommendation bereits aus dem Jahr 2001 stammt, beginnt XML-Schema sich nun langsam durchzusetzen. Dieser Artikel soll dazu einen Beitrag leisten. Er führt zuerst in die Konzepte von XML-Schema ein, um dann kurz zu zeigen, wie der Xerces-Parser in Java damit umgeht.

XML-Schema statt DTD

Eine DTD ist, obwohl sie ein XML-Dokument beschreibt, selbst nicht in XML verfasst. Das sieht zwar eher wie ein Schönheitsfehler aus, es bedeutet aber, dass eigene Parser nötig sind, um solche Definitionen zu verarbeiten. Zudem ist die Syntax recht komplex.

Zweiter Nachteil ist die eingeschränkte Möglichkeit, den strukturellen Aufbau zu beschreiben. Über Content-Modelle lässt sich zwar angeben, dass gewisse Elemente in vorgegebener Reihenfolge (zum Beispiel Kapitel–Titel–Abschnitt) oder als Alternative zu erscheinen haben, aber da stoßen DTDs schon an ihre Grenzen. Eine DTD kann zwar festlegen, dass kein, ein oder mehrere Abschnitte nach dem Titel folgen, aber eine Obergrenze (etwa maximal fünf Abschnitte) anzugeben ist nicht möglich.

Ein dritter Nachteil ist, dass sich mit DTDs der Inhaltstyp der Elemente nicht festlegen lässt: Ein mittels DTD definiertes Postleitzahl-Element ist auch gültig, wenn es statt einer Zahl einen Text enthält. XML-Schema schafft Typsicherheit und löst auch die anderen Schwierigkeiten der DTDs.

Hinter XML-Schema steht ein einfaches Modell von XML-Dokumenten: Es gibt Elemente (Tags), Attribute und das so genannte Content-Modell, das festlegt, welche Elemente in welcher Reihenfolge auftreten dürfen und müssen. XML-Schema führt für Elemente und Attribute auch einen Typ ein.

Drei Schritte zum Schema

Um ein XML-Schema zu schreiben, sind nur drei Dinge erforderlich: Typen definieren, Elemente mit ihren Attributen festlegen und das Content-Modell beschreiben. Die folgenden Abschnitte folgen in etwa diesem Dreischritt. Obwohl XML seine Wurzeln in SGML und HTML hat, dient es schon lange nicht mehr nur dazu, Texte zu schreiben. XML beschreibt heute auch Datenstrukturen und Datenobjekte. Deshalb dient ein (schon recht altes) Projekt des Autors zur Illustration.

Das Plotlib-Toolkit (siehe [4]) ist eine Java-Bibliothek für das Plotten mathematischer Funktionen und wurde zusammen mit Java2D schon einmal in einem Coffee-Shop [5] vorgestellt. Das Toolkit soll in einer seiner nächsten Versionen XML-Dateien einlesen, die Plots aus Linien und Symbolen beschreiben, und diese grafisch darstellen. Doch das ist noch Zukunftsmusik. Fürs Erste soll

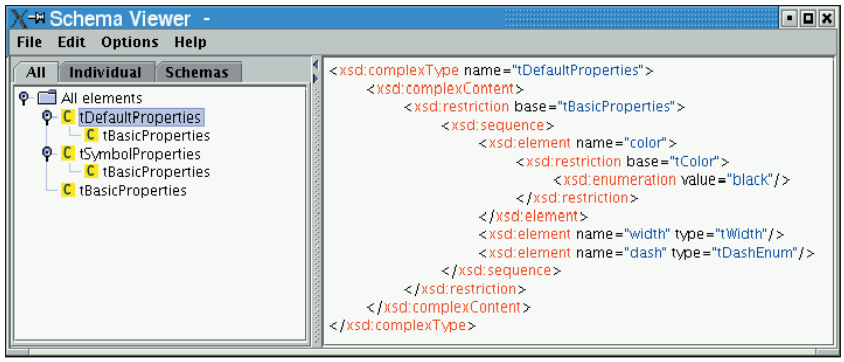

Abbildung 1: Der komplexe Typ aus Listing 2 in einem Browser für XML-Schemas [8]. Er stellt die Struktur als Baum dar und hebt Elemente farblich hervor.

es genügen, den XML-Schema-Standard kurz vorzustellen.

Einfache und komplexe Typen

Ähnlich wie Java (und andere objektorientierte Sprachen) unterscheidet auch XML-Schema zwischen einfachen und komplexen Typen (simple types, complex types). Beispiele für einfache Typen sind Zahlen, Zeichenketten oder ein Datum. Davon gibt es eine ganze Reihe, zum Beispiel »string«, »integer«, »long« oder »duration«. Komplexe Typen unterscheiden sich von den einfachen dadurch, dass sie Attribute besitzen oder Subelemente enthalten.

Typen gehorchen einer Hierarchie. An oberster Stelle steht »anyType«, analog zu »java.lang.Object«. Jede Typdefinition nimmt Bezug auf einen Basistyp, wobei der Bezug auf »anyType« üblicherweise weggelassen wird, da er implizit ist. Listing 1 zeigt die Typdefinition des simplen Typs »tSizeEnum«. Dessen Basistyp ist der vordefinierte Typ »string«. Der eigene Typ ist eine Einschränkung davon, denn er erlaubt nur eine ausgewählte Anzahl (»enumeration«) symbolischer String-Konstanten (»SMALL«, »BIG«, »LARGE«).

Das Beispiel veranschaulicht die übliche Methode: So genannte Constraining Facets schränken die Basistypen ein und bilden eigene Typen. Neben dem verwendeten »enumeration« gibt es weitere einschränkende Facets wie »length«, »pattern« oder »maxInclusive«. Das »pattern«-Facet ist für Typen nützlich, die zwar eine bestimmte Struktur aufweisen (etwa KFZ-Kennzahlen), bei denen aber ein eigener komplexer Typ zu aufwändig wäre.

Typen zusammensetzen

Auch einfache Typen können zusammengesetzt sein, und zwar mittels Listen (etwa: »<liste>a b c</liste>«) oder Unions. Wie Letzteres funktioniert, demonstriert der Typ »tSize« in den Zeilen 9 bis 11. Größenangaben gibt man im Plotlib-Toolkit nämlich über die erwähnten symbolische Konstanten oder direkt in Pixeln an. Die Union »tSize« erlaubt daher die beiden Alternativen »tSizeEnum« oder »positiveInteger« (also einen eingebauten Typ).

Einfache Typen sind die Basis für Attribute und komplexe Typen. Die Typdefinition komplexer Elemente legt deren Content-Model fest, definiert also, welche Typen (einfache oder komplexe) sie in welcher Reihenfolge enthalten. Als Beispiel für einen komplexen Typ ist in dem Listing 2 die Definition der Klasse »tBasicProperties« aus dem Plotlib-Toolkit abgedruckt. Die Klasse besteht aus den drei Attributen »color«, »width« und »dash«, die die Farbe, Breite und das Muster von Linien in einer Java2D-Grafik festlegen.

Das XML-Schema-Fragment für den Typ »tBasicProperties« verwendet das Content-Model »sequence« (bei DTDs war dies das Komma). Das »minOccurs«-Attribut beim Element »color« legt mit dem Wert »0« ein optionales Element fest. Ein höherer Wert wäre ebenso möglich. Als Obergrenze gibt es das »maxOccurs«-Attribut mit dem Default-Wert »1«.

Bei komplexen XML-Typen funktioniert die Vererbung ähnlich wie bei Java-Klassen. So erweitert der Typ »tSymbolProperties« aus dem Plotlib-Toolkit den Typ »tBasicProperties« um zwei Attribute (Größe und Symbolform). Die Zeilen 9 bis 18 in Listing 2 zeigen, wie die Erweiterung in einer XML-Schema-Definition funktioniert (Abbildung 1).

Vererbung und Polymorphismus

Java-Klassen können Methoden ihrer Basisklasse überschreiben, Instanzvariablen verdecken dieselben Namen der Basisklasse. XML-Schema bietet einen ähnlichen Mechanismus: Typen können über eine Einschränkung (restriction) den Basistyp verändern. Wie das aussieht, zeigt das Beispiel in den Zeilen 20 bis 34. Die Vererbung funktioniert aber nicht völlig analog zu Java, da eingeschränkte Typen ein Subset und keine Erweiterung definieren.

Ein weiteres objektorientiertes Paradigma ist der Polymorphismus. Er besagt, dass Objekte abgeleiteter Klassen auch von einem Code verarbeitet werden, der nur die Basisklasse kennt. Sind bestimmte Methoden in der abgeleiteten

Listing 1: Definition einfacher Typen

```
01 <xsd:simpleType name="tSizeEnum">
02   <xsd:restriction base=="xsd:string">
03     <xsd:enumeration value="SMALL"/>
04     <xsd:enumeration value="BIG"/>
05     <xsd:enumeration value="LARGE"/>
06   </xsd:restriction>
07 </xsd:simpleType>
08
09 <xsd:simpleType name="tSize">
10   <xsd:union memberTypes="tSizeEnum
   positiveInteger"/>
11 </xsd:simpleType>
```

Listing 2: Definition komplexer Typen

```
01: <xsd:complexType name="tBasicProperties">
02:   <xsd:sequence>
03:     <xsd:element name="color" type="tColor"
    minOccurs="0"/>
04:     <xsd:element name="width" type="tWidth"/>
05:     <xsd:element name="dash"
    type="tDashEnum"/>
06:   </xsd:sequence>
07: </xsd:complexType>
08:
09: <xsd:complexType name="tSymbolProperties">
10:   <xsd:complexContent>
11:     <xsd:extension base="tBasicProperties">
12:       <xsd:sequence>
13:         <xsd:element name="size" type=
    "tSize"/>
14:         <xsd:element name="type" type=
    "tSymbolEnum"/>
15:       </xsd:sequence>
16:     </xsd:extension>
17:   </xsd:complexContent>
18: </xsd:complexType>
19:
20: <xsd:complexType name="tDefaultProperties">
21:   <xsd:complexContent>
22:     <xsd:restriction base=
    "tBasicProperties">
23:       <xsd:sequence>
24:         <xsd:element name="color">
25:           <xsd:restriction base="tColor">
26:             <xsd:enumeration value=
    "black"/>
27:           </xsd:restriction>
28:         </xsd:element>
29:         <xsd:element name="width" type=
    "tWidth"/>
30:         <xsd:element name="dash"  type=
    "tDashEnum"/>
31:       </xsd:sequence>
32:     </xsd:restriction>
33:   </xsd:complexContent>
34: </xsd:complexType>
```

Klasse nicht implementiert, verwendet das Laufzeitsystem die Methoden der Basisklasse. Die überschriebenen Methoden stammen aber weiterhin von der abgeleiteten Klasse.
Ähnlich sieht es bei XML-Dokumenten aus. In den Instanzdokumenten taucht nur der Basistyp auf, qualifiziert mit einem zusätzlichen Attribut. Dies erlaubt einfache und kompakte Formulierungen für XML-Schemas:

```
<meinBasisTyp
  xsi:type="meinNamespace:meinSubTyp">
...
</meinBasisTyp>
```

Zu diesem Zweck gibt es den Namespace »xsi« (XML-Schema Instance), dessen »type«-Attribut den Typ spezifiziert. Ob dieses Feature nützlich ist oder zur Übersichtlichkeit beiträgt, ist im Einzelfall zu entscheiden.

Elemente und Attribute

Bis jetzt ging es um Typen von Elementen und Attributen, die eigentliche Deklaration von Elementen wurde noch nicht beschrieben. Um Elemente und Attribute zu definieren, bietet XML-Schema die Tags »<element>« und »<attribute>«. Komplexe Typen definieren über das »<attribute>«-Tag einfach die erlaubten Attribute.
Elemente, die nicht selbst Teil anderer Elemente sind (also innerhalb eines komplexen Typs definiert sind), heißen globale Elemente. In der Schema-Definition sind sie unterhalb der Wurzel »<schema>« verankert (Listing 3). Da jedes XML-Dokument ein Wurzel-Tag hat, ist mindestens ein globales Element notwendig. Das Listing zeigt den ersten Teil der kompletten Schema-Definition der Plotlib-Bibliothek. Die Zeilen 3 bis 15 reichern das Schema mit Metadaten (»annotation«) an, neben dem »documentation«-Tag ist auch ein »appInfo«-Tag verfügbar. Annotations sind prinzipiell auch an anderen Stellen erlaubt, etwa in Element-Definitionen.
Das Wurzel-Tag im Listing ist »plots« (Zeile 19), die zugehörige Typdefinition (Zeilen 23 bis 33) besagt, dass innerhalb eines »plots« mindestens ein »plot«-Subelement auftauchen muss. Die Zeilen 25 bis 30 zeigen weitere Möglichkeiten von XML-Schema. Der Typ des »plot«-Elements ist nämlich nicht explizit definiert, sondern anonym an Ort und Stelle, ähnlich den anonymen inneren Klassen bei Java. Außerdem – und auch das erhöht die Lesbarkeit – sind Elemente und Attribute in benannte Gruppen ausgelagert und werden in den Zeilen 27 und 28 nur referenziert.

Xerces-J: Überall dabei

Auch wenn dieser Artikel immer wieder Parallelen zu Java gezogen hat und das Beispiel aus dem Java-Umfeld stammt, sind die Java-Programmierer bis jetzt etwas zu kurz gekommen Die letzten Abschnitte konzentrieren sich deshalb auf den XML-Parser Xerces, der im Java-Umfeld ein Quasistandard ist.
Ein »find / -name "xerces*.jar"« fördert zumindest auf Systemen, die viele freie Java-Produkte installiert haben, eine Unmenge von Xerces-Jar-Dateien zu Tage. Aus diesem Grund ist es empfehlenswert, keine Xerces-Datei fest in den systemweiten »CLASSPATH« aufzunehmen – zu groß ist die Gefahr, dass Programme die falsche Version verwenden. Der Download von Xerces-J von der

Listing 3: »plotlib.xsd«

```
001: <?xml version="1.0" encoding="UTF-8"?>
002: <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
003:   <xsd:annotation>
004:     <xsd:documentation xml:lang="en">
005:
006:       $Author: Bablokb $
007:       $Revision: 1.2 $
008:
009:       Schema definition for plots generated and processed by
010:       the Plotlib-Toolkit.
011:       Copyright (c) 1999-2003 by Bernhard Bablok, mail@bablokb.de
012:       Released under the terms of the GPL version 2
013:
014:     </xsd:documentation>
015:   </xsd:annotation>
016:
017: <!-- Toplevel element ------------------------------------------ -->
018:
019:   <xsd:element name="plots" type="tPlots"/>
020:
021: <!-- Complex type: tPlots (1st/2nd/3rd level) ------------------ -->
022:
023:   <xsd:complexType name="tPlots">
024:     <xsd:sequence>
025:       <xsd:element name="plot" minOccurs="1" maxOccurs="unbounded">
026:         <xsd:complexType>
027:           <xsd:group          ref="gPlottable"/>
028:           <xsd:attributeGroup ref="agBasicMetadata"/>
029:         </xsd:complexType>
030:       </xsd:element>
031:     </xsd:sequence>
032:     <xsd:attributeGroup ref="agPlotMetadata"/>
033:   </xsd:complexType>
034:
035:   <xsd:group name="gPlottable">
036:     <xsd:choice minOccurs="1" maxOccurs="unbounded">
037:       <xsd:element name="symbol"       type="tSymbol">
038:       <xsd:element name="symbols"      type="tSymbols">
039:       <xsd:element name="line"         type="tLine">
040:       <xsd:element name="lines"        type="tLines">
041:       <xsd:element name="border"       type="tBorder">
042:       <xsd:element name="grid"         type="tGrid">
043:       <xsd:element name="axis"         type="tAxis">
044:       <xsd:element name="tickMarks"    type="tTickMarks">
045:       <xsd:element name="barPlot"      type="tBarPlot">
046:       <xsd:element name="linePlot"     type="tLinePlot">
047:       <xsd:element name="functionPlot" type="tFunctionPlot">
048:       <xsd:element name="imagePlot"    type="tImagePlot">
049:       <xsd:element name="scatterPlot"  type="tScatterPlot">
050:     </xsd:choice>
051:   </xsd:group>
052:
053: ...
054:
172: </xsd:schema>
```

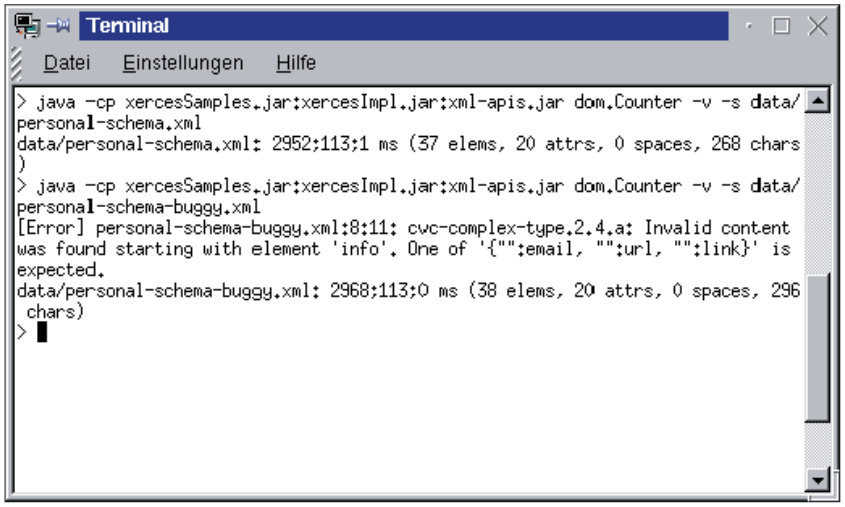

Abbildung 2: Mit dem bei Xerces als Beispiel mitgelieferten »dom.Counter« lassen sich beliebige XML-Dateien schnell und einfach validieren.

Homepage des Apache-XML-Projekts [6] ist auch noch per Modem erträglich, das Paket ist 3,5 MByte klein. Ausgepackt sind es dann 28 MByte.

Das Paket enthält eine ausführliche Dokumentation und einige Beispiele, die einen schnellen Start in die Java-XML-Programmierung erlauben. Eine regelrechte Installation ist nach dem Download gar nicht notwendig. Es genügt schon, die beiden Dateien »xercesImpl.jar« und »xml-apis.jar« auf der Kommandozeile mit »-cp« in den aktuellen Classpath aufzunehmen.

Validieren mit Xerces

Xerces ist ein „fully conforming" XML-Schema-Prozessor, erfüllt also vollständig die Spezifikation. Das spielt allerdings nur beim Validieren von XML-Dokumenten eine Rolle. Je nach Anwendungsfall kann es durchaus sinnvoll sein, darauf zu verzichten, eine Datei zu validieren. Etwa wenn man mit fehlerhaften Dokumenten leben kann oder weil ein Programm sie generiert hat und sie deshalb fehlerfrei sind.

Um schnell eine XML-Datei zu validieren, eignet sich das im Xerces-Paket enthaltene Beispiel »dom.Counter«. Abbildung 2 zeigt, wie es sich verwenden lässt. Die fehlerhafte Datei »personal-schema-buggy.xml« unterscheidet sich von dem im »data«-Verzeichnis mitgelieferten Beispieldokument »personal-schema.xml« durch ein zusätzliches Tag mit dem Namen »info«.

Das Xerces-API lässt sich leicht in eigenen Programmen verwenden. Listing 4 zeigt die Grundstruktur eines DOM- und eines SAX-Parsers. Validierung/Schema-Validierung sind nur zwei der Features, mit denen sich der Parser an eigene Bedürfnisse anpassen lässt. Die Methode »setFeature()« stellt seine Eigenschaften ein (Zeilen 9, 10 und 21, 22). Parameter sind ein URI, der das gewünschte Feature angibt, und ein Boolean-Wert, der das jeweilige Feature aus- oder einschaltet. So aktiviert der in »VAL_ID« gespeicherte URI (Zeile 2) die Validierung des Dokuments, während der Inhalt von »S_VAL_ID« (Zeile 3) bestimmt, dass der Parser dazu XML-Schema verwendet.

finally{}

Dieser Coffee-Shop gab eine kompakte Einführung in die XML-Schema-Technologie und stellte Xerces-J vor, eine Java-Bibliothek, die neben der ganzen XML-Palette auch XML-Schema beherrscht. Der XML-Schema Primer [1] beschreibt weitere Aspekte, etwa die wichtigen Namespaces, und geht noch etwas mehr ins Detail. Informativ ist auch der Besuch auf der Schema-Website des W3C [7]. Dort gibt es Links zu praktischen Tools wie Schema-Validatoren, DTD-zu-Schema-Konvertern oder Schema-zu-Java-Generatoren. *(ofr)* ■

Infos

[1] XML-Schema-Primer: [http://www.w3.org/TR/XMLSchema-0]
[2] XML-Schema-Structures: [http://www.w3.org/TR/XMLSchema-1]
[3] XML-Schema-Datatypes: [http://www.w3.org/TR/XMLSchema-2]
[4] Homepage des Plotlib-Toolkits: [http://www.bablokb.de/plotlib/]
[5] Coffee-Shop, „Java2D – Nicht nur für bunte Bilder": Linux-Magazin 03/00, S. 154, [http://www.linux-magazin.de/Artikel/ausgabe/2000/03/Java2D/java2d.html]
[6] Homepage des Apache XML-Projekts: [http://xml.apache.org]
[7] XML-Schema-Homepage: [http://www.w3.org/XML/Schema]
[8] XML-Schema-Browser: [http://www.geocities.com/frakilk/software.html]

Der Autor

Bernhard Bablok arbeitet bei der Allianz Versicherungs AG im Bereich Data-Warehouse-Systeme. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.

Listing 4: Validierung innerhalb von Java-Programmen

```
02: String VAL_ID = "http://xml.org/sax/
    features/validation";
03: String S_VAL_ID = "http://apache.org/xml/
    features/validation/schema";
04:
05: /* DOM ------------------------------ */
06:
07: DOMParser parser = new DOMParser();
08: try {
09:   parser.setFeature(VAL_ID,true);
10:   parser.setFeature(S_VAL_ID,true);
11: }
12: catch (SAXException e) {
13:   System.err.println("could not set parser
    feature");
14: }
15:
16: /* SAX ------------------------------ */
17:
18: SAXParser parser = new SAXParser();
19: XMLReader reader = parser.getXMLReader();
20: try {
21:   reader.setFeature(VAL_ID,true);
22:   reader.setFeature(S_VAL_ID,true);
23: }
24: catch (SAXException e) {
25:   System.err.println("could not set parser
    feature");
26: }
```