

Berufsakademie Mosbach
University of Cooperative Education
Lohrtalweg 10
74821 Mosbach



Objektorientierte Erweiterung von XSLT in Java

— Studienarbeit —

„Studienarbeit für die Prüfung zur Diplom-Medieninformatikerin (BA) an der
Berufsakademie Mosbach“

Name: Holger Rüprich

Geburtsort: Vorwerk

Geburtsdatum: 01.08.1982

Studiengang: Digitale Medien

Studienjahrgang: DM04

Ausbildungsbetrieb: Horn Druck & Verlag
Stegwiesenstraße 6-10
76466 Bruchsal

Betreuer: Stefan Mintert

Abgabedatum: 16.02.2007

Inhaltsverzeichnis

1	Einleitung	1
1.1	Aufgabenstellung	1
1.2	Aufbau der Arbeit	1
2	Grundlagen	2
2.1	HTTP	2
2.1.1	Nachrichten	2
2.1.2	Header-Typen	3
2.2	XML-Schema	4
2.2.1	DTD versus XML-Schema	4
2.2.2	Abstraktes Datenmodell	5
3	XML-Schema-Modellierung	7
3.1	Attribut versus Element	7
3.2	Global versus lokal	8
3.3	General-, Request-, Response- und Entity-Header	10
3.4	Datentypen	10
3.4.1	Einfache Headertypen	11
3.4.2	Einfache Headertypen mit Datum	11
3.4.3	Komplexe Headertypen	13
3.4.4	Komplexe Headertypen mit Einschränkungen	14
3.4.5	Headertypen mit abweichender Grammatik	16
3.5	Ergebnis	18
4	Entwicklung der XSLT-Erweiterung	19
4.1	HTTP-Zugriff	19
4.2	XML erzeugen	19
4.2.1	StringBuffer	19
4.2.2	Document Object Model	20
4.2.3	XML Data Binding	21
4.3	Architektur	24
4.3.1	Archive	24
4.3.2	Namenskonvention	25

4.4	Header Elemente	27
4.4.1	Header erzeugen	27
4.4.2	HttpHeaderCommons	28
4.5	XML an XSLT liefern	29
4.6	Verwendung der Erweiterung	30
5	Fazit und Ausblick	32
A	Literaturverzeichnis	33

1 Einleitung

1.1 Aufgabenstellung

Mit der `document()`-Funktion (XSLT 1) bzw. `doc()`-Funktion (XPath 2) steht in XSLT eine Möglichkeit zur Verfügung, ein externes Dokument per URI in die Verarbeitung des laufenden Programms einzubeziehen. Sofern es sich bei dem URI um einen HTTP-URL handelt, entspricht `document()` damit einem Aufruf der HTTP-Methode GET. Es fehlt jedoch an einem Äquivalent für HTTP-HEAD.

Ziel dieser Studienarbeit ist, eine entsprechende Funktion als Erweiterung für XSLT in Java zu schreiben, lauffähig mit XSLT-Prozessoren Xalan und Saxon. Dazu ist im ersten Schritt für den Kopf einer HTTP-Response ein XML-Schema zu entwerfen. Darin sollen die in HTTP/1.1 definierten Header-Zeilen abbildbar sein, und es sollen möglichst geeignete Daten bzw. Elementtypen gewählt bzw. definiert werden.

Im zweiten Schritt geht es um die Entwicklung einer XSLT-Funktion, die für einen gegebenen URL einen HTTP-HEAD-Request absendet und die erhaltenen Daten in einer XML-Struktur gemäß dem in Schritt 1 entwickelten Schema als Node-Set an das aufrufende XSLT-Programm zurück gibt. So könnte z.B. die letzte Änderung einer Datei anhand des „Last-Modified“ Headerfeldes überprüft werden oder mit Hilfe des „Content-Type“ Headerfeldes der MIME-Typ einer Datei kontrolliert werden.

1.2 Aufbau der Arbeit

Diese Studienarbeit ist im Wesentlichen in vier Teile gegliedert. Im ersten Teil werden die Grundlagen zu HTTP und XML-Schema besprochen. Der zweite Teil geht auf den Entwurf des XML-Schemas für den HTTP Response Header ein. Der dritte Teil erläutert die Implementierung der XSLT-Erweiterung. Wonach der vierte und letzte Teil ein Fazit und Ausblick bietet.

2 Grundlagen

2.1 HTTP

Das Hypertext Transfer Protokoll (HTTP) ist ein Protokoll für verteilte und gemeinschaftliche Hypermedia-Informationssysteme auf Applikationsebene. Es ist ein generisches, zustandsloses Protokoll, das durch die Erweiterung von Anfrage-Methoden, Fehlercodes und Headern neben Hypertext auch für viele Anwendungen, wie z.B. Nameserver und verteilte Objekt Management Systeme, verwendet werden kann.

HTTP wird seit 1990 von der World Wide Web global information initiative verwendet. Die erste Version, von T. Berners Lee, wird HTTP/0.9 bezeichnet und war ein einfaches Protokoll für rohen Datentransfer über das Internet.

Diese Studienarbeit verwendet die Spezifikation HTTP/1.1. Das Protokoll enthält strengere Voraussetzungen als HTTP/1.0, um die zuverlässige Realisierung seiner Funktionen zu sichern. (vgl. [Fielding u. a., 1999])

2.1.1 Nachrichten

HTTP Nachrichten sind einfache und formatierte Datenpakete, die zwischen HTTP Anwendungen ausgetauscht werden. Jede Nachricht enthält nach [Gourley und Totty, 2002] entweder eine Anfrage des Klienten oder eine Antwort des Servers. Sie besteht aus drei Teilen:

- der `start-line`, welche den Inhalt der Nachricht beschreibt,
- dem `message-header` Block, mit keinem bis beliebig vielen Headern und deren Attribute,
- sowie einem optionalen `message-body` der die Daten der Nachricht enthält.

```
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line = Request-Line | Status-Line
(vgl. [Fielding u. a., 1999])
```

Alle Mechanismen im RFC 2616 sind sowohl als Prosa, als auch in der hier verwendeten Backus-Naur Form beschrieben. (vgl. [Fielding u. a., 1999], 2.1 Augmented BNF)

2.1.2 Header-Typen

[Wong, 2000] erläutert die vier Typen von HTTP Headern des RFC 2616 [Fielding u. a., 1999] wie folgt:

- **General Header** deuten auf eine generelle Information hin, wie z.B. das Datum. Sie werden gleichermaßen von Klienten und Servern verwendet.
- **Request Header** werden ausschließlich vom Klient verwendet, um die eigene Konfiguration sowie das gewünschte Dokumentenformat an den Server zu übertragen.
- **Response Header** werden lediglich vom Server genutzt. Sie enthalten Informationen über das angefragte Dokument und die Konfiguration des Servers.
- **Entity Header** beschreiben Daten, welche zwischen Klient und Server transportiert werden. Sie werden von Servern verwendet, wenn ein angefragtes Dokument zurück geliefert wird und von Klienten, sofern diese Daten mit Hilfe der Methoden POST oder PUT an den Server senden.

Neben den existierenden Headern besteht ebenfalls die Möglichkeit eigenen Headern Semantiken von Request-, Response- oder Entity-Headern zu verleihen, wenn dies alle an der Kommunikation beteiligten Parteien erkennen. Unbekannte Header werden als Entity-Header behandelt. (vgl. [Fielding u. a., 1999])

2.2 XML-Schema

Die Extensible Markup Language (XML) ermöglicht Entwicklern eigene Formate zum Speichern und Austauschen von Informationen zu erstellen. Eine der Hauptaufgaben ist dabei, die Definition von Regeln und Dokumentationen auf deren Grundlage Software entwickelt werden kann. (vgl. [van der Vlist, 2002])

Hierzu boten bereits Document Type Definitions (DTDs) grundlegende Regeln und Strukturmodelle für eine Klasse von XML-Dokumenten. Doch mit der schnellen Verbreitung von XML benötigten immer mehr Applikationen genauere und ausdrucksvollere Möglichkeiten zur Kontrolle von Element- und Attributinhalten. Aus diesem Grund veröffentlichte das W3C am 2. Mai 2001 die dreiteilige XML-Schema-Empfehlung und am 8. Oktober 2004 eine zweite Auflage. (vgl. [Harold und Means, 2002] sowie [Sperberg-McQueen und Thompson, 2007])

Die XML-Schema-Empfehlung besteht einerseits aus einer nicht normativen Einführung (Primer) [Fallside und Walmsley, 2004], welche Entwicklern eine leicht verständliche Beschreibung von XML-Schema bieten soll. Andererseits enthält sie mit den beiden Teilen Strukturen (Structures) [Thompson u. a., 2004] und Datentypen (Datatypes) [Biron und Malhotra, 2004] eine komplette normative Beschreibung der Sprache.

Neben dem XML-Schema des W3C gibt es ebenso andere XML-Schemata, wie z.B. RELAX NG [Clark, 2003] oder Schematron [Jelliffe, 2001], auf die im Folgenden nicht weiter eingegangen wird.

2.2.1 DTD versus XML-Schema

Doch worin liegt der Vorteil des W3C XML-Schema gegenüber der DTD? [Harold und Means, 2002] erläutern dies wie folgt:

DTDs unterstützen die einfache Validierung von Element-Schachtelung, Element-Häufigkeitsbeschränkungen, verbotenen Attributen, Attribut-Typen und Standardwerten. Sie bieten allerdings keine genaue Kontrolle über das Format und den Datentyp von Element- und Attribut-Werten. Enthält ein Element oder ein Attribut „character data“, können keine weiteren Einschränkungen festgelegt werden.

Der W3C XML-Schema Standard beinhaltet einfache und komplexe Datentypen sowie namespacebewusste Element- und Attribut-Beschreibungen. Zudem ermöglicht er sowohl Typableitung und Vererbung als auch Element-Häufigkeitsbeschränkungen. Wobei der größte Vorteil in der Einführung von einfachen Datentypen für „parsed character data“ und Attribut-Inhalten liegt. Dadurch können XML-Schemata im Gegensatz zu DTDs spezielle Regeln über den Inhalt von Elementen und Attributen definieren. Zusätzlich zur breiten Spanne an eingebauten einfachen Typen, wie z.B. `string`, `integer` und `dateTime`, können darüber hinaus neue Datentypen deklariert werden.

2.2.2 Abstraktes Datenmodell

Wie XML und XML-Namensräume lassen sich auch XML-Schemata laut [Thompson u. a., 2004] durch ein abstraktes Datenmodell in Form von Informationseinheiten, gemäß Definition im XML-Infoset [Cowan und Tobin, 2001], beschreiben. Der Begriff Schema-Komponente bezeichnet die Bausteine, aus denen das abstrakte Schema-Modell zusammengesetzt ist. Insgesamt definiert die Spezifikation nach [Thompson u. a., 2004] 13 Schema-Komponenten, die sich in Primäre-, Sekundäre- und Hilfskomponenten aufteilen.

Primäre-Komponenten

- Definition von einfachen Typen
`<xsd:simpleType>`
- Definition von komplexen Typen
`<xsd:complexType>`
- Attribut-Deklaration
`<xsd:attribute>`
- Element-Deklaration
`<xsd:element>`

Sekundäre-Komponenten

- Attributgruppen-Definition
`<xsd:attribute-group>`
- Identitätsbeschränkungs-Definitionen
`<xsd:unique>`, `<xsd:key>`, `<xsd:keyref>`
- Elementgruppen-Definition
`<xsd:group>`
- Notations-Deklaration
`<xsd:notation>`

Hilfskomponenten

- Anmerkung
`<xsd:annotation>`
- Elementgruppen
`<xsd:sequence>`, `<xsd:choice>`, `<xsd:all>`
- Partikel
involviert in `<xsd:elements>`, `<xsd:group>`, `<xsd:any>` mit Occurs-Attributen
- Wildcards
`<xsd:any>`, `<xsd:anyAttribute>`
- Attributvorkommen
involviert in `<xsd:attribute>` mit use-Attribut

3 XML-Schema-Modellierung

Im vorherigen Kapitel wurde besprochen, wie HTTP Header aufgebaut sind und die Grundlagen von XML-Schema wurden erläutert. Nun soll ein XML-Schema für den HTTP Response Header modelliert werden. Im Rahmen dieser Arbeit sind dabei die Headerfelder, sowie die Statuszeile von Interesse.

Zwei Punkte gilt es zu klären: Einerseits die Entscheidung zwischen Attribut oder Element und andererseits die Datentypen für die einzelnen Headerfelder. Dieses Kapitel stellt sich diesen und weiteren Fragen und beschreibt anschließend das fertige XML-Schema für den HTTP Response Header.

3.1 Attribut versus Element

Die HTTP Headerfelder sind gemäß RFC 822 alle nach dem gleichen Prinzip aufgebaut:

```
field = field-name ":" [ field-body ] CRLF [Crocker, 1982]
```

Für die Modellierung in XML folgen daraus zwei Möglichkeiten: Entweder wird ein Element `field` mit dem Attribut `name` definiert oder es wird für jedes Headerfeld ein eigenes Element, wie z.B. `<date>`, bestimmt. Für die erste Variante spricht, dass das Schema sehr übersichtlich gehalten wird, da alle Header durch das gleiche Element beschrieben werden. Der Inhalt lässt sich allerdings nur noch bedingt kontrollieren, da keine Spezifikationen für spezielle Headerfelder vorgenommen werden können. Doch wie sieht es mit der Alternative aus?

Vor- und Nachteile der Modellierung eines Headerfeldes als eigenes Element:

- Das Schema wird unübersichtlicher, da viele verschiedene Header existieren.
- + Inhalte können in Abhängigkeit vom Typ ohne die Verwendung von

Kindenlementen spezialisiert werden.

- + Jedes Headerfeld kann einzeln definiert werden und einen eigenen Typ haben.
- + Der XPath Ausdruck in XSLT wird kürzer und ist deutlich leichter zu lesen:

```
/HTTPResponseHeader/field[@name = "Date"]  
/HTTPResponseHeader/field[contains(@name, "Date")]  
/HTTPResponseHeader/Date
```

Ich entschied mich die Headerfelder als eigene Elemente zu definieren, da diese Option mehr Vorteile bietet.

3.2 Global versus lokal

XML-Schema unterstützt eine große Bandbreite an Strukturen (vgl. [Biron und Malhotra, 2004]) und lässt den Schema-Autoren damit viele Freiheiten und Möglichkeiten bei der Entwicklung eigener XML-Schemata. Häufig erschwert dies jedoch die Entscheidung für den optimalen Weg. Zu Beginn der Modellierung des HTTP Response Headers kam schnell die Frage auf, ob Elemente und Typen global oder lokal definiert werden sollten. Im Folgenden hierzu vier Entwurfsprinzipien nach [Costello, 2001] und [Maler, 2002].

Die meisten Prinzipien wurden erstmals von Roger Costello im Rahmen seiner „XML Schema: Best Practices“ Initiative dokumentiert. Darin beschreibt er das Russische-Puppen-, das Salami-Scheiben- und das Venezianische-Jalousien-Prinzip.

Das Russische-Puppen-Prinzip

Element-Deklarationen werden innerhalb anonymer Typ-Definitionen, wie bei den bekannten russischen „Matryoshka“-Puppen, immer tiefer geschachtelt.

Das Salami-Scheiben-Prinzip

Das Instanzdokument wird in individuelle Komponenten zerlegt und jede Komponente wird als eigene Element-Deklaration definiert, die später über Referenzen zusammengefügt werden.

Venezianische-Jalousien-Prinzip

Dieses Prinzip ist das Gegenteil zum Salami-Scheiben-Prinzip. Typ-Deklarationen tragen Namen und werden global definiert. Alle Elemente mit Ausnahme des Top-Level-Elements werden lokal deklariert.

Das Garten-Eden-Prinzip

Eve Maler erweitert diese Prinzipien um das Garten-Eden-Prinzip. Dabei werden alle Elemente global deklariert, Typ-Deklarationen tragen Namen und Inhaltsmodelle binden Elemente wie bei dem Salami-Scheiben-Prinzip ausschließlich über Referenzen ein.

Jedes der vier Prinzipien bringt gewisse Vor- bzw. Nachteile mit sich:

- So erhält man mit dem Russische-Puppen-Prinzip ein sehr kurzes Schema, bei dem sich allerdings keine Komponente wiederverwenden lässt.
- Das Salami-Scheiben-Prinzip ermöglicht demgegenüber die Wiederverwendung einzelner Elemente, deklariert allerdings alle Elemente global. So wird das Schema schnell unübersichtlich und beeinflusst bei Änderungen an einem Element alle anderen die dieses verwenden.
- Ähnlich verhält sich das Venezianische-Jalousien-Prinzip, das im Gegensatz dazu nicht die Elemente sondern die Typ-Definitionen global definiert. Der Vorteil gegenüber dem Salami-Scheiben-Prinzip liegt in der Möglichkeit die Preisgabe von Namespaces über das Element `elementFormDefault` an und aus zu schalten.
- Das Garten-Eden-Prinzip bietet noch mehr Möglichkeiten zur Wiederverwendung von Komponenten, da dabei sämtliche Elemente und Typ-Definitionen global deklariert werden. Das Schema wird auf Grund dessen ebenfalls schnell unübersichtlich, bringt viele Abhängigkeiten mit sich und vermindert zudem das Potential Namespaces zu verstecken.

Für die Modellierung des HTTP Response Headers habe ich mich für das Venezianische-Jalousien-Prinzip entschieden. Da das XML-Schema für die Anwender der Erweiterung zusätzlich als Dokumentation dienen soll, werden allerdings nur die Typen global deklariert, die auch in anderen Headertypen wiederverwendet werden.

3.3 General-, Request-, Response- und Entity-Header

Wie bereits in Kapitel 2 beschrieben, unterteilen sich die HTTP Header in General-, Request-, Response- und Entity-Header. Diese Unterteilung wird allerdings nicht im HTTP Protokoll eingebunden, sondern wird lediglich zur Gliederung im RFC beschrieben. Daher stellt sich die Frage, ob diese Unterteilung im XML-Schema berücksichtigt werden soll, oder nicht.

Für die Unterteilung gibt es zwei Alternativen:

- **In Form von Elementen**

Für jeden Header-Typ existiert ein Element, das die entsprechenden Headerfelder als Kindelemente hat.

- **In Form von einem Attribut**

Jedes Headerfeld hat ein Attribut, das den Header-Typ angibt.

Ich habe mich entschieden, die Unterteilung der Header in Form von Attributen umzusetzen. Da in XML-Schema ein Attribut als optional definiert werden kann, liegt die Entscheidung über die Verwendung letztendlich beim XML-Instanzdokument-Autor.

3.4 Datentypen

HTTP spezifiziert mit Ausnahme der Request-Header 28 Headerfelder. Davon sind neun General-Header, neun Response-Header und zehn Entity-Header. Zusätzlich kann, wie bereits in Kapitel 2 beschrieben, jede Gruppe durch so genannte Extension-Header erweitert werden.

Ein Teil der HTTP Header kann direkt auf die vordefinierten Datentypen aus XML-Schema abgebildet werden. In anderen Fällen, in denen der Inhalt aus mehreren Teilen besteht, müssen eigene Typen definiert werden. Im Folgenden werden einige Header erläutert, die im Rahmen der Studienarbeit erstellt wurden.

3.4.1 Einfache Headertypen

Acht der 28 Headerfelder bestehen aus Inhalt der direkt auf die einfachen von XML-Schema vordefinierten Datentypen abgebildet werden kann. Davon ist ein Feld das `Location` Response-Headerfeld. Es wird verwendet, um den Empfänger zu einer anderen Adresse als der angeforderten URI zu verweisen, um die Anfrage zu vervollständigen oder eine neue Ressource zu identifizieren. Der Inhalt des Feldes besteht aus einer einzigen absoluten URI. (vgl. [Fielding u. a., 1999])

Der komplexe Header-Typ `LocationHeader` erweitert daher den simplen vordefinierten Typ `xsd:anyURI` lediglich um das `type`-Attribut. Das Attribut `type` wird von allen Headern verwendet, um die Zugehörigkeit zu einer der drei Header-Gruppen (General-, Response- und Entity-Header) anzugeben.

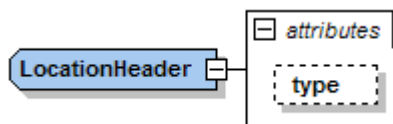


Abb 1: Der komplexe Typ `LocationHeader`

Quelle: Eigene

```
<xsd:complexType name="LocationHeader">
  <xsd:simpleContent>
    <xsd:extension base="xsd:anyURI">
      <xsd:attribute name="type" type="http:type"
                    fixed="response" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

3.4.2 Einfache Headertypen mit Datum

Headerfelder mit einem Datum als Inhalt, haben eine Sonderstellung bei den einfachen Headertypen, da das Format `HTTP-date` nicht direkt auf den von XML-Schema vordefinierten Datentyp `dateTime` abgebildet werden kann. Anstatt

dessen muss der Inhalt dieser Felder von der XSLT-Erweiterung umgewandelt werden. Ein solches Headerfeld ist z.B. das General-Headerfeld `Date`, welches das Datum und die Uhrzeit repräsentiert, zu der die Nachricht erstellt wurde.

Der Inhaltstyp `HTTP-date` erlaubt nach [Fielding u. a., 1999] drei verschiedene Formate:

```
Sun, 06 Nov 1994 08:49:37 GMT ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov 6 08:49:37 1994 ; ANSI C's asctime() format
```

Der W3C XML-Schema Datentyp `dateTime` folgt nach [Biron und Malhotra, 2004] im Unterschied dazu dem Datenformat der Spezifikation ISO 8601. Dies besteht aus Zeichen in der Form:

```
'-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)?
(zzzzzz)?
```

"yyyy" steht dabei für das Jahr, "mm" für den Monat und "dd" für den Tag. Der Buchstabe "T" dient als Trennzeichen zwischen Datum und Zeit. Stunden, Minuten und Sekunden werden durch "hh", "mm" und "ss" repräsentiert. Sekunden können, durch einen Punkt getrennt, um eine beliebige Anzahl von Nachkommastellen erweitert werden. Wahlweise kann anschließend eine Zeitzone definiert werden.

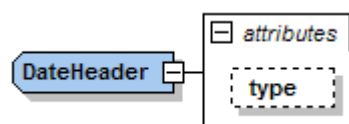


Abb 2: Der komplexe Typ `DateHeader`

Quelle: Eigene

```

<xsd:complexType name="DateHeader">
  <xsd:simpleContent>
    <xsd:extension base="xsd:dateTime">
      <xsd:attribute name="type" type="http:type"
                    fixed="general" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

```

3.4.3 Komplexe Headertypen

Von den 28 Headerfeldern bestehen die 20 Weiteren aus Inhalt, der nicht direkt auf die vordefinierten Datentypen aus XML-Schema abgebildet werden kann. Ein Beispiel ist das `Server` Response-Headerfeld. Es enthält Informationen über die Software, welche von dem Server verwendet wird, der die Anfrage bearbeitet. Das Feld kann in beliebiger Reihenfolge mehrere Produkt-Tokens und Kommentare enthalten, um den Server sowie signifikante Produkte zu identifizieren. (vgl. [Fielding u. a., 1999])

Da beide Elemente (`product` und `comment`) auch in anderen Headerfeldern verwendet werden, sind diese global definiert und werden über eine Referenz eingebunden.

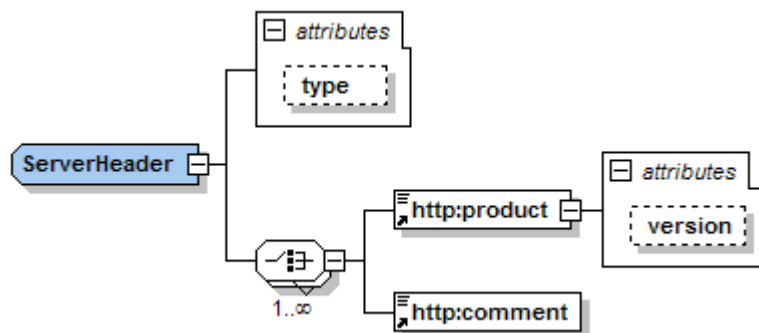


Abb 3: Der komplexe Typ ServerHeader

Quelle: Eigene

```

<xsd:complexType name="ServerHeader">
  <xsd:choice maxOccurs="unbounded">
    <xsd:element ref="http:product" />
    <xsd:element ref="http:comment" />
  </xsd:choice>
  <xsd:attribute name="type" type="http:type"
                fixed="response" />
</xsd:complexType>

```

3.4.4 Komplexe Headertypen mit Einschränkungen

Vier der 20 komplexen Headertypen schränken den Inhalt der Headerfelder noch weiter über das Element `xsd:restriction` ein. Dazu gehört z.B. das Response-Headerfeld `Proxy-Authenticate`, das aus einer oder mehreren `challenges` besteht, welche das Authentifizierungsschema und die Parameter angeben, die für den Proxy der Anfrage-URI gelten. (vgl. [Fielding u. a., 1999])

Der Header-Typ `ProxyAuthenticateHeader` enthält daher ein oder mehrere `challenge` Elemente, welche jeweils aus dem Attribut `auth-scheme` und einem oder mehreren `auth-param` Elementen bestehen. Da der Inhalt des Attributs `auth-scheme` auf die Werte „Digest“ und „Basic“ beschränkt ist, werden diese Werte mit Hilfe des Elements `xsd:restriction` in Form einer `xsd:enumeration` vom Schema fest vorgegeben.

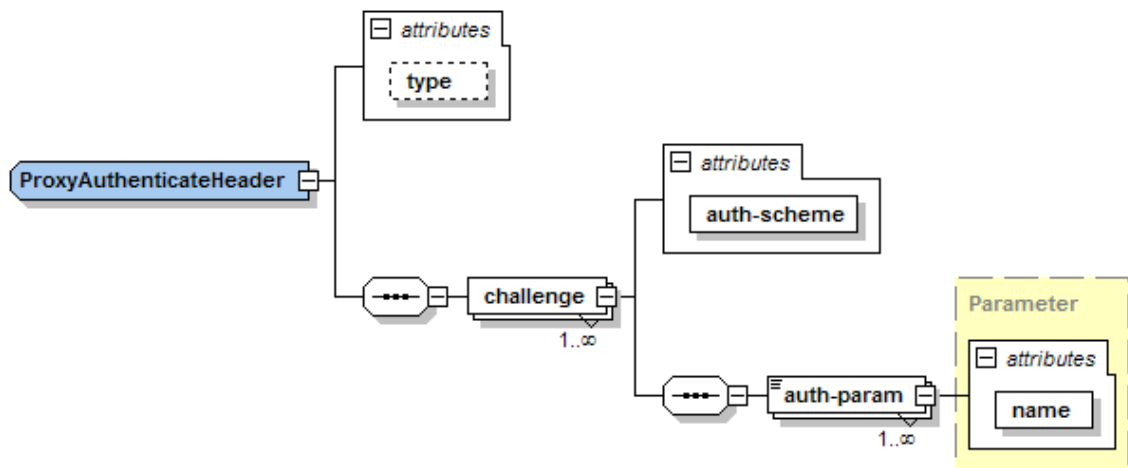


Abb 4: Der komplexe Typ ProxyAuthenticateHeader

Quelle: Eigene

```

<xsd:complexType name="ProxyAuthenticateHeader">
  <xsd:sequence>
    <xsd:element name="challenge" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="auth-param" type="http:Parameter"
            maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="auth-scheme" use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="Basic" />
              <xsd:enumeration value="Digest" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="type" type="http:type"
    fixed="response" />
</xsd:complexType>

```

3.4.5 Headertypen mit abweichender Grammatik

Insgesamt drei der 28 Headertypen halten sich nicht direkt an die vorgegebene Grammatik der HTTP Spezifikation. So z.B. das Entity-Headerfeld `Content-Range`, das zusammen mit einem Teilbereich des Entity-Body versandt wird, um festzulegen an welcher Stelle im Entity-Body der Teilbereich eingefügt werden soll. (vgl. [Fielding u. a., 1999])

Der Header-Typ `ContentRangeHeader` erlaubt als Inhalt ausschließlich die Elemente `byte-range-resp-spec` und `instance-length`. Diese dürfen einmal oder gar nicht auftreten. Das Element `byte-range-resp-spec` wird nach [Fielding u. a., 1999] wie folgt definiert:

```
byte-range-resp-spec = (first-byte-pos "-" last-byte-pos)
| "*"

```

Ein Stern deutet darauf hin, dass der angefragte Teilbereich nicht vorhanden ist. Im Schema wird dies im Gegensatz dazu durch Fehlen des `byte-range-resp-spec` Elements dargestellt. Ist der angefragte Teilbereich vorhanden, enthält das Element die beiden Elemente `first-byte-pos` und `last-byte-pos`, um den Teilbereich des Entity-Body zu spezifizieren.

Um auf diesen Unterschied hinzuweisen, enthält die Definition eines Headertyps mit abweichender Grammatik das Element `xsd:annotation`, worin der Unterschied innerhalb des Elements `xsd:documentation` dokumentiert wird.

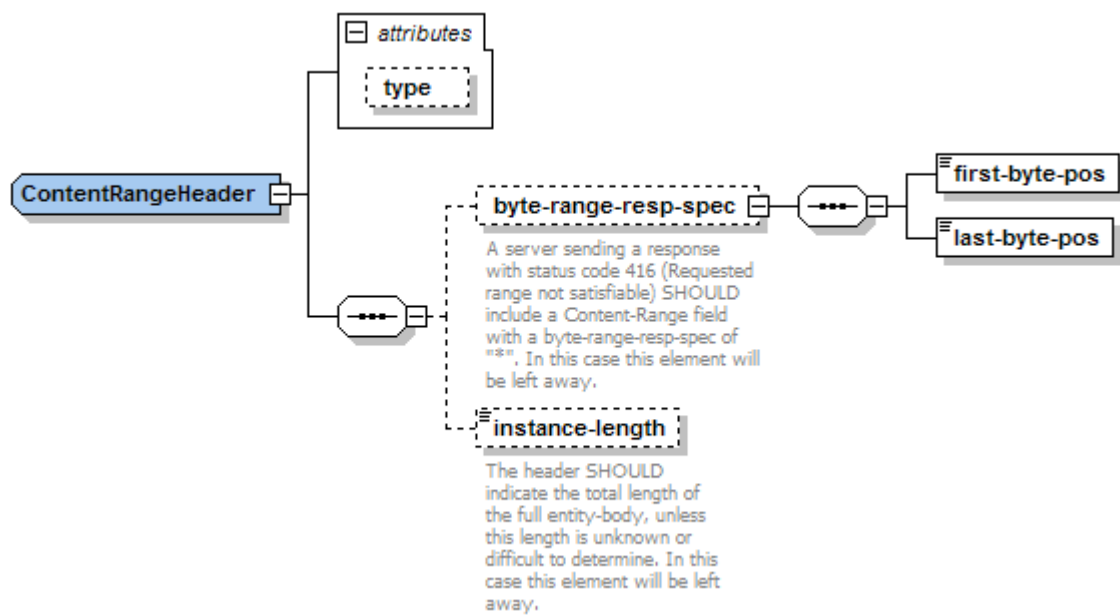


Abb 5: Der komplexe Typ ContentRangeHeader

Quelle: Eigene

```

<xsd:complexType name="ContentRangeHeader">
  <xsd:sequence>
    <xsd:element name="byte-range-resp-spec" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>A server ...</xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="first-byte-pos"
            type="xsd:positiveInteger" />
          <xsd:element name="last-byte-pos"
            type="xsd:positiveInteger" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="instance-length"
      type="xsd:positiveInteger" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>The header ...</xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

```

```

    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="type" type="http:type" fixed="entity" />
</xsd:complexType>

```

3.5 Ergebnis

Nach dem alle Fragen geklärt und die einzelnen Headerfelder modelliert wurden, steht das erste Teilergebnis fest. Das XML-Schema für den HTTP Response Header. Zur besseren Übersicht wurden, nach Vorbild der HTTP Spezifikation, einerseits die einzelnen Headertypen in die Gruppen GeneralHeaders, ResponseHeaders und EntityHeaders aufgeteilt. Andererseits wurden auch die Elemente http-version, status-code und reason-phrase innerhalb der StatusLine gruppiert. Ein Schema-valides XML-Instanzdokument besteht nach dieser Modellierung mindestens aus den Elementen der StatusLine. Alle weiteren Teile sind optional.

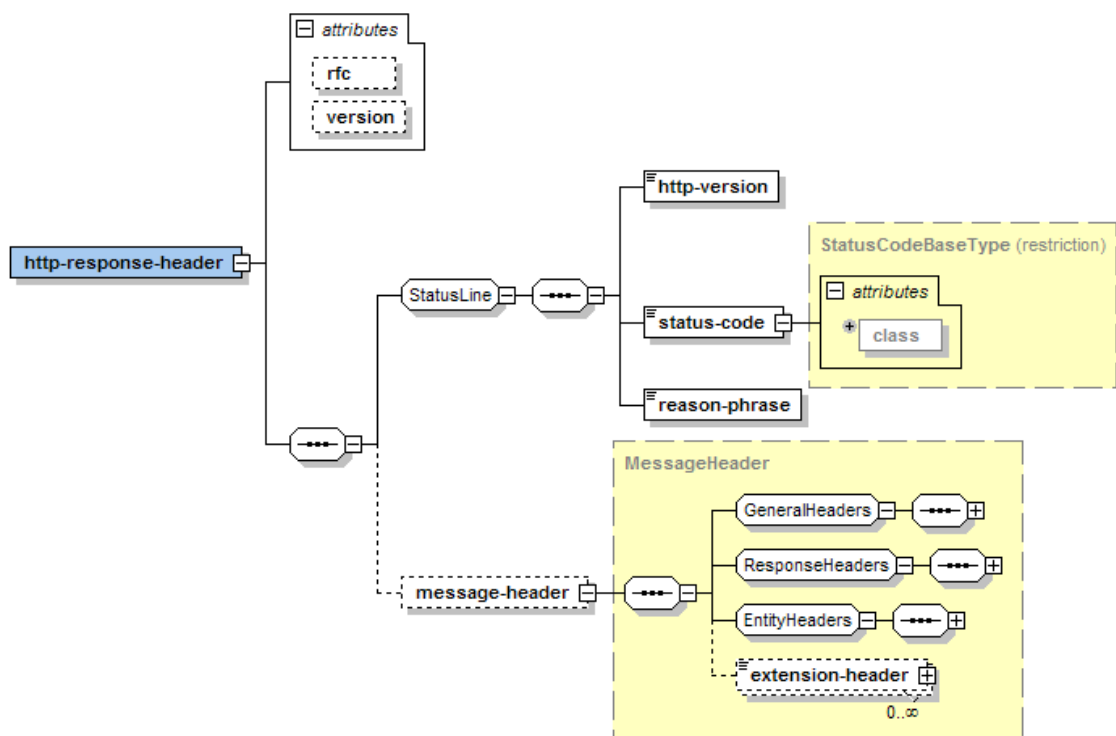


Abb 6: Das XML-Schema für den HTTP Response Header

Quelle: Eigene

4 Entwicklung der XSLT-Erweiterung

In Kapitel 3 wurde das XML-Schema für den HTTP Response entwickelt und dient im folgenden als Grundlage für die Entwicklung der XSLT-Erweiterung. Hierzu müssen die einzelnen Response Header geparkt und ein Schema-valides XML-Dokument erzeugt werden.

4.1 HTTP-Zugriff

Bei der Recherche nach Zugriffsmöglichkeiten auf Ressourcen via HTTP aus Java, wird schnell der Jakarta Commons HttpClient aus dem Apache Jakarta Projekt gefunden. Dieser bietet deutlich mehr Möglichkeiten als das Java Paket java.net, wie z.B. persistente Verbindungen durch KeepAlive in HTTP/1.0 und persistence in HTTP/1.1. (vgl. [Becke u. a., 2006])

Da eine Anforderung des Projektes allerdings die Vermeidung von Abhängigkeiten ist und HttpURLConnection die nötigen Funktionen bereits bietet, verwendete ich für das Projekt das Java Standardpaket java.net.

4.2 XML erzeugen

Für die Erstellung des XML-Instanzdokuments gibt es verschiedene Möglichkeiten. Im Folgenden werden davon drei genauer beschrieben.

4.2.1 StringBuffer

Die technologisch einfachste dieser Möglichkeiten besteht darin, die Headerfelder in einer XML-Baumstruktur innerhalb eines StringBuffers zu konkatenieren.

Diese Methode ist allerdings auch die fehleranfälligste Variante, da keine Validierung vorgenommen wird und komplexe Verschachtelungen schnell unübersichtlich werden.

4.2.2 Document Object Model

Eine weitere Methode bietet das Document Object Model (DOM) Interface, das in vielen Script- und Programmiersprachen wie z.B. Java, PHP oder ECMAScript implementiert ist. (vgl. [Le Hégaret u. a., 2005])

Ein DOM-Dokument besteht aus einem Baummodell, das komplett aus dem Interface `Node` aufgebaut wird. Abgeleitet von diesem Interface, bietet das DOM zahlreiche XML-spezifische Interfaces wie `Element`, `Document`, `Attr` und `Text`. In einem typischen XML-Instanzdokument könnte die Struktur wie folgt aussehen:

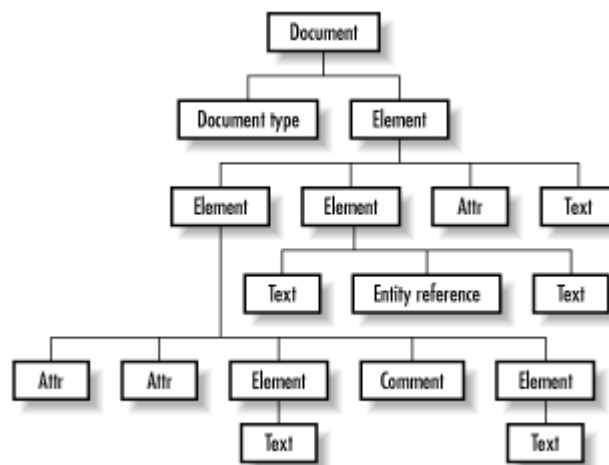


Abb 7: Typische Struktur eines XML-Instanzdokuments

Quelle: Java & XML, 2nd Edition

Mit Hilfe der Navigationsmethoden des `Node` Interface, wie z.B. `getParent()` und `getChildren()`, kann das DOM-Dokument durchlaufen werden. (vgl. [McLaughlin, 2001])

Um einen DOM-Baum in Java aufzubauen, wird eine Klasse benötigt, die das DOM-Interface implementiert. Sun bietet hierzu mit der Java API for XML Processing (JAXP) einen so genannten „Pluggability Layer“, der über seine herstellerneutrale Fabrik-Klasse `DocumentBuilderFactory` die implementierende Klasse zur Verfügung stellt. (vgl. [Armstrong u. a., 2005])

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory .newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();
```

Die einzelnen Nodes werden über entsprechende create-Methoden der Klasse Document erzeugt und jeweils dem Elternelement als Kind angehängt.

```
Element root = document.createElement("root");
Element node = document.createElement("node");
root.appendChild(node);
document.appendChild(root);
```

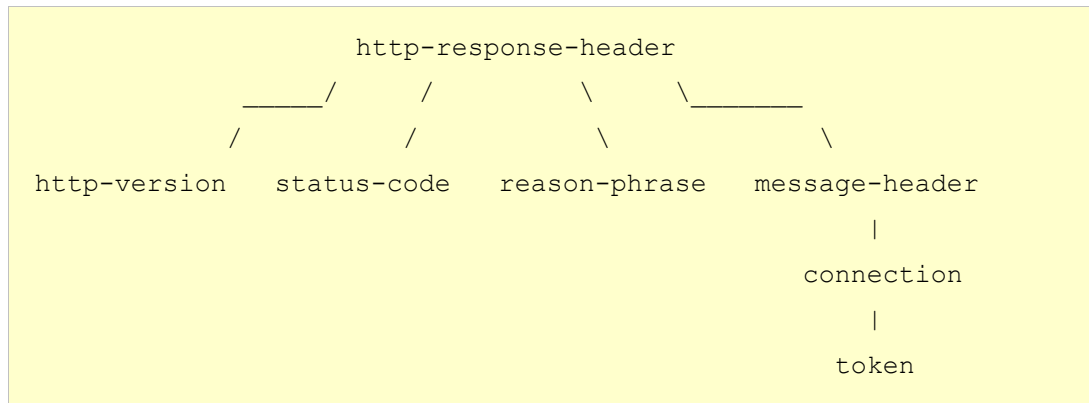
4.2.3 XML Data Binding

Nach [Bourret, 2006] ermöglicht XML Data Binding Daten aus XML-Dokumenten an Objekte zu binden. Dadurch wird den in der Regel datenorientierten Anwendungen ermöglicht, serialisierte XML-Daten in einer natürlicheren Art und Weise als über das DOM zu bearbeiten. Betrachten wir hierzu folgendes Beispiel-Dokument:

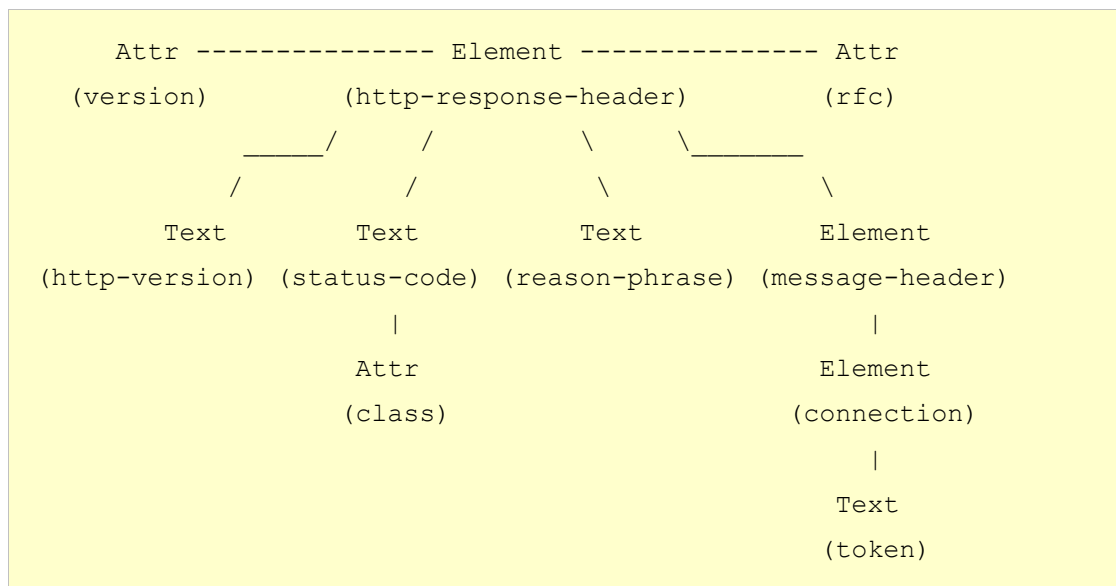
```
<http-response-header>
  <http-version>HTTP/1.1</http-version>
  <status-code class="Informational">101</status-code>
  <reason-phrase>String</reason-phrase>

  <message-header>
    <connection>
      <token>String</token>
    </connection>
  </message-header>
</http-response-header>
```

Dies könnte mit den Klassen `HttpResponseHeader`, `HttpVersion`, `StatusCode`, `ReasonPhrase`, `MessageHeader`, `Connection` und `Token` verknüpft werden, was zur folgenden Baumdarstellung führen würde:



DOM modelliert im Gegensatz dazu das Dokument selbst und würde einen Baum in folgender Form erzeugen:



Für die Entwicklung der XSLT-Erweiterung ist die zweite Variante sichtlich einfacher, da so direkt mit den Objekten, wie z.B. `HttpResponseHeader`, gearbeitet werden kann. Genau dafür wird XML Data Binding eingesetzt. Es bietet die Möglichkeit ein XML-Schema, in Form einer DTD oder eines XML-Schema Dokuments, auf ein Objekt-Schema abzubilden.

Anhand der Abbildung können anschließend Objekte aus XML-Dokumenten erstellt werden (unmarshalling) oder Objekte als XML serialisiert werden (marshalling).

Java Architecture for XML Binding (JAXB)

JAXB ist ein XML Data Binding Produkt in Java, das seit der Version 6 ein fester Bestandteil der Java Standard Edition ist (vgl. [Sun Microsystems]). Eine JAXB Implementierung besteht nach [Eric Jendrock u. a., 2006] aus den folgenden Architektur-Komponenten:

- Einem **Schema Compiler**, der ein Schema an ein Set von Schemata abgeleiteten Programmklassen bindet.
- Einen **Schema Generator**, der ein Set von existieren Programmelementen auf ein abgeleitetes Schema abbildet.
- Und einem **Binding Runtime Framework**, das unmarschalling und marshalling Operationen zur Verfügung stellt, um auf XML-Inhalt zuzugreifen, diesen zu bearbeiten und zu validieren.

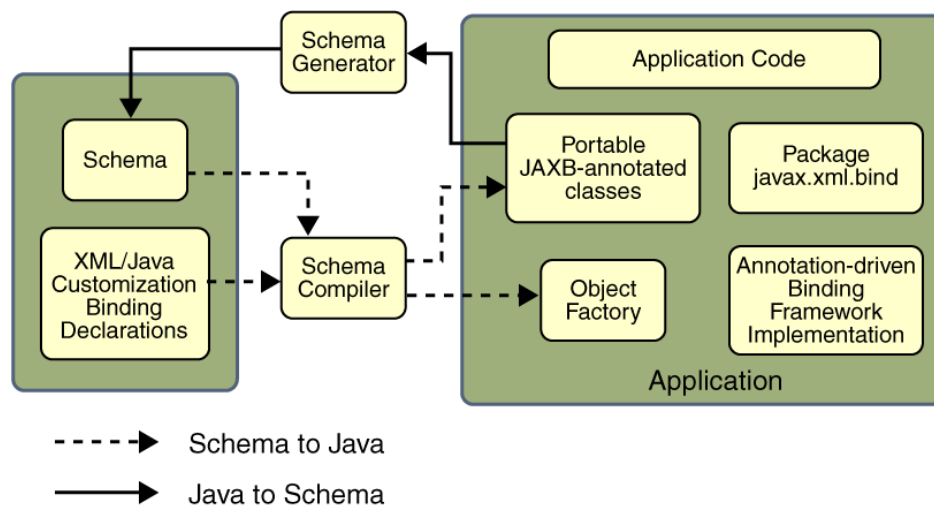


Abb 8: Die Komponenten aus denen eine JAXB Implementierung besteht

Quelle: <http://java.sun.com/javaee/5/docs/tutorial/doc/JAXB2.html>

4.3 Architektur

Die XSLT-Erweiterung besteht aus vier Klassen, sowie aus den von JAXB generierten Klassen und einigen Unit-Tests. Die Funktionsweise und Paket-Struktur wird im Folgenden genauer beschrieben.

4.3.1 Archive

httpresponseheader.jar

In dem Jar-Archiv `httpresponseheader.jar` befinden sich die vier Klassen `Client`, `HttpResponse`, `HTTPHeaderCommons` und `HttpObjectFactory` im Verzeichnis „`java/http`“ sowie das XML-Schema im Ordner „`schema`“.

- **Client**

Baut die HTTP Verbindung zum URL auf, erzeugt das XML-Dokument und liefert dieses an den XSLT-Prozessor als `org.w3c.dom.Node`.

- **HttpResponse**

Repräsentiert die Antwort eines HTTP Servers.

- **HTTPHeaderCommons**

Bietet statische Methoden, um den Inhalt der Headerfelder gemäß der erweiterten Backus-Naur Form des RFC 2616 [Fielding u. a., 1999] zu parsen und in geeignete Objekte umzuwandeln.

- **HttpObjectFactory**

Erweitert die `ObjectFactory` von JAXB. Die einzelnen `create`-Methoden für die Headerfelder parsen den übergebenen String und erzeugen daraus das Headerfeld sowie den entsprechenden Inhalt.

com.linkwerk.namespaces._2007_01_08.http.response.jar

In diesem Archiv befinden sich die von JAXB generierten Klassen. Diese werden in einem eigenen Archiv verpackt, damit sie leichter austauschbar sind und zudem einfacher von anderen Projekten verwendet werden können. (vgl. [Leisegang, 2007])

4.3.2 Namenskonvention

Bei der Erstellung des XML-Dokuments aus dem HTTP Response Header ist der Name des Headerfelds ein wesentlicher Punkt, da über diesen die Verbindung zu der entsprechenden Klasse hergestellt wird.

Die erste Möglichkeit hierbei ist für jedes Headerfeld eine Abfrage zu schreiben und die Namen der Headerfelder einzeln zu überprüfen. Wird keine der 28 Abfragen erfüllt, handelt es sich um einen Extension-Header.

Eine Alternative hierzu bietet das Reflection API, in der Form wie es auch bei Java Beans zum Einsatz kommt. Dort wird es eingesetzt, „um die Datenelemente der Bean-Klassen (in diesem Kontext meist Properties genannt) zu ermitteln und deren Werte auszulesen bzw. neu zu setzen. Diese Properties sind zwar spezifisch für alle Bean-Klassen, jedoch folgen die Bean-Properties einem Muster: Zum Einstellen einer Property gibt es eine `setXXX()`-Methode und zum Auslesen eine `getXXX`-Methode, wobei XXX für den Namen der Property steht.“ (vgl. [Middendorf u. a., 2002]).

Dieses Prinzip kann auch für die Headerfelder der HTTP-Response genutzt werden, insofern sich alle Header an ein bestimmtes Muster bzw. an eine bestimmte Namenskonvention halten:

1. Jedes Headerfeld hat einen eigenen Typ.
2. Der Name des Headerfeld-Datentyps folgt dem Muster `XXXHeader`, wobei `XXX` für den Namen des HTTP Headerfeldes gemäß RFC 2616 [Fielding u. a., 1999] in Pascal-Case-Schreibweise (vgl. [Microsoft Corporation]) steht, aus dem zusätzlich alle Minus-Zeichen entfernt wurden.

Dieser Weg ist deutlich besser. Er ist offen für Erweiterungen und folgt dem DRY-Prinzip [Hunt und Thomas, 2000], wodurch an dieser Stelle die Wiederholung von 28 fast identischen Abfragen vermieden werden kann. Allerdings birgt dieser Weg nach [Leisegang, 2007] auch einige Schwierigkeiten: So ist z.B. der Aufruf von Methoden über das Reflection-API deutlich langsamer und der Einsatz eines Obfuscation-Postcompilers nicht möglich. Beides sind jedoch Punkte die

vernachlässigt werden können, da es sich zum einen um eine überschaubare Anzahl an aufgerufenen Methoden handelt und zum anderen kein Obfuscation-Postcompiler eingesetzt wird.

Um die Namen der HTTP Headerfelder in den Namen der entsprechenden Klasse zu wandeln wird die Methode `headerKeyToClassName()` eingesetzt. Diese wandelt das erste Zeichen, sowie jedes Zeichen nach einem Bindestrich in einen Großbuchstaben. Die Bindestriche werden entfernt. Alle anderen Zeichen werden in Kleinbuchstaben gewandelt.

```
public static String headerKeyToClassName(String headerKey) {
    StringBuffer className = new StringBuffer();
    char[] headerKeyArray = headerKey.toCharArray();

    for(int j=0; j<headerKeyArray.length; j++) {
        // Das erste Zeichen und alle Zeichen nach einem Minus
        // Zeichen werden in Großbuchstaben umgewandelt
        if(j==0 || headerKeyArray[j-1] == '-') {
            className.append(
                String.valueOf(headerKeyArray[j]).toUpperCase()
            );
        }
        // Minus Zeichen werden entfernt. Alle anderen Zeichen
        // werden in Kleinbuchstaben umgewandelt
        else if(headerKeyArray[j] != '-') {
            className.append(
                String.valueOf(headerKeyArray[j]).toLowerCase()
            );
        }
    }
    return className.toString();
}
```

4.4 Header Elemente

4.4.1 Header erzeugen

Wie gerade beschrieben folgen alle Header-Klassen der Namenskonvention. So kann die Klasse `HttpObjectFactory`, die von der generierten Klasse `ObjectFactory` abgeleitet wird, jeden Header über die entsprechende `createXXXHeader()` Methode erzeugen.

Existiert für einen HTTP Header keine passende Header-Klasse, wirft die Methode `getMethod()` eine `NoSuchMethodException`. In diesem Fall handelt es sich um einen Extension-Header, der anstelle der bekannten Header im `catch`-Block erzeugt wird.

```
private static void addHeaderField(ObjectFactory factory,
    MessageHeader messageHeader, String headerKey,
    String headerValue) {
    try {
        String headerClassName = headerKeyToClassName(headerKey);
        Method createHeader = factory.getClass().getMethod(
            "create"+headerClassName+"Header",
            new Class[] {" ".getClass()});
        );
        Class headerClass = createHeader.getReturnType();
        Object header = createHeader.invoke(factory, headerValue);
        messageHeader.getClass().getMethod(
            "set"+headerClassName, new Class[] {headerClass}
        );
        setHeader.invoke(messageHeader, new Object[] {header});
    }
    catch (NoSuchMethodException e) {
        ExtensionHeader element = factory.createExtensionHeader();
        element.setName(headerKey);
        element.setValue(headerValue);
        messageHeader.getExtensionHeader().add(element);
    }
}
```

4.4.2 HttpHeadersCommons

Die Klasse `HttpHeaderCommons` bietet eine Reihe von statischen Methoden an, um den Inhalt der Headerfelder, gemäß der erweiterten Backus-Naur Form des RFC 2616 [Fielding u. a., 1999], zu parsen und in geeignete Objekte zu wandeln.

So verwenden zum Beispiel zahlreiche Headerfelder eine Liste der Form „#element“. Die ganze Form davon lautet „<n>#<m>element“, wobei *n* die Mindestanzahl und *m* die maximale Anzahl an Elementen in der Liste ist. Einzelne Elemente dieser Liste werden durch ein oder mehrere Kommas und optionalen Whitespace getrennt.

Um eine Liste in dieser Form zu parsen, wird die Methode `commaList()` verwendet, welche wiederum die Methode `list()` aufruft und dieser als `separator` ein Komma übergibt. Als Antwort erhält man die Liste als String-Array. Taucht das Trennzeichen nicht in `list` auf, enthält der Rückgabe-Array ausschließlich den String `list`. Andernfalls wird der String an jedem Trennzeichen gesplittet und die einzelnen Teile anschließend als Array zurückgegeben.

```
public static String[] list(String list, char separator) {
    String[] tokens;
    if(list.indexOf(separator) == -1) {
        tokens = new String[] { list.trim() };
    }
    else {
        tokens = list.split("[ "+separator+" ]");
    }
    return tokens;
}
```

4.5 XML an XSLT liefern

Die Methode `httpResponseHeader()` ist die Schnittstelle zu XSLT. Sie stellt die Verbindung zum übergebenen URL her, parst den HTTP Response und liefert anschließend das Ergebnis an XSLT in Form einer `Node` zurück.

Dafür wird zuerst mit Hilfe der `DocumentBuilderFactory` von JAXP ein leeres `Document`-Objekt erstellt. Schlägt bereits die Erzeugung des Parsers fehl, wird eine `ParserConfigurationException` geworfen. In diesem Fall muss ein `null`-Wert zurückgegeben werden, da es keine andere Möglichkeit gibt ein Objekt der abstrakten Klasse `Node` zu erzeugen. Andernfalls wird die Verbindung zum URL aufgebaut und das Objekt-Modell der HTTP Response in einem Objekt der generierten Klasse `HttpResponseHeader`, dem Wurzelement unseres XML-Instanzdokuments, erzeugt.

Mit dem `Marshaller` des `JAXBContext` für den HTTP Response wird das Objekt-Modell in XML gewandelt und an das leere `Document`-Objekt gehängt. Tritt hierbei kein Fehler im Zusammenhang mit JAXB auf, wird das Wurzelement des Dokuments an XSLT zurückgegeben.

```
public static Node httpResponseHeader(String URL) {
    try {
        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.newDocument();

        try {
            HttpURLConnection con = Client.connect(URL);
            HttpResponse httpResponse = new HttpResponse(con);
            HttpResponseHeader httpResponseHeader =
                createHttpResponseHeader(httpResponse);

            JAXBContext jc = JAXBContext.newInstance(headerPackage);
            Marshaller marshaller = jc.createMarshaller();
            marshaller.setProperty(
```

```

        Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true)
    );

    marshaller.marshal(httpResponseHeader, doc);
} catch (JAXBException e) {
    e.printStackTrace();
}

return doc.getDocumentElement();
} catch (ParserConfigurationException e1) {
    e1.printStackTrace();
    return null;
}
}

```

4.6 Verwendung der Erweiterung

Der XSLT-Standard [Clark, 1999] definiert zwei Arten von Erweiterungen: Erweiterungselemente und Erweiterungsfunktionen. Beide werden exakt wie eingebaute Elemente und Funktionen, z.B. `xsd:template` und `document()`, verwendet. Sie werden allerdings nicht vom XSLT-Prozessor selbst, sondern von außerhalb, in unserem Fall von Java, bearbeitet. (vgl. [Harold, 2002])

Eine Funktion wird in XSLT als Erweiterungsfunktion angesehen, falls ihr Name im Funktionsaufruf-Ausdruck kein `NCName` ist. Trifft dies zu wird der Funktionsname um die Namensraum-Deklarationen des Evaluierungskontexts erweitert.

Für die Verwendung eines Erweiterungselements muss vorab, zusätzlich zum Namensraum, der Erweiterungsname definiert werden. Dazu wird der Namensraum als Erweiterungsname ausgezeichnet, indem er in einem `extension-element-prefixes`-Attribut innerhalb des `xslt:stylesheet`-Elements eines literalen Ergebniselements oder Erweiterungselements deklariert wird. Taucht ein Element in diesem Namensraum auf, wird es als Anweisung an Stelle eines literalen Ergebniselements behandelt. (vgl. [Clark, 1999])

In diesem Fall wird um die XSLT-Erweiterung einzusetzen, zuerst der Namespace `httpresponse` definiert und anschließend als Erweiterungsname­nsraum deklariert:

```
<xslt:stylesheet version="1.0"
  xmlns:xslt="http://www.w3.org/1999/XSL/Transform"
  xmlns:httpresponse="http.Client"
  extension-element-prefixes="httpresponse">
```

So kann die Methode `httpResponseHeader()` z.B. als Erweiterungsfunktion in einem `xslt:apply-templates`-Element verwendet werden:

```
<xslt:apply-templates
  select="httpresponse:httpResponseHeader('http://www.w3.org')"
  mode="http" />
```

5 Fazit und Ausblick

Mit der vorliegenden Arbeit wurde die Entwicklung einer XSLT-Erweiterungsfunktion für die HTTP-HEAD Methode beschrieben, die ebenfalls im Rahmen der Studienarbeit entstand. Dazu wurden vorab die Grundlagen zu HTTP und XML-Schema besprochen und anschließend die Modellierung des XML-Schemas erläutert, das als Basis für die weitere Entwicklung diene.

Da immer mehr Anwendungen XML unterstützen, ist davon auszugehen, dass die Bedeutung von XSLT ebenso stark zunehmen wird und somit auch die Bedeutung der Erweiterungen. Bereits vor einigen Jahren formierte sich eine Gruppe von Entwicklern unter Exslt.org mit dem Ziel eine Standardisierung von XSLT-Erweiterungen zu erreichen.

Auch wenn ich kein Befürworter der Eierlegenden-Wollmilchsau bin, ist es an vielen Stellen sinnvoll unterschiedliche Welten mit einander zu verbinden und die Vorteile beider Welten zu nutzen. Ein gutes Beispiel hierfür ist die im Rahmen dieser Studienarbeit erstellen Erweiterung.

A Literaturverzeichnis

- [Armstrong u. a., 2005] **Eric Armstrong u.a.**, The J2EE 1.4 Tutorial, 2005, Online im Internet, <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXPIntro.html>, Abfrage vom 11.02.2007
- [Becke u. a., 2006] **Michael Becke u. a.**, Jakarta Commons HttpClient, 2006, Online im Internet, <http://jakarta.apache.org/commons/httpclient/>, Abfrage vom 12.01.2007
- [Biron und Malhotra, 2004] **Paul V. Biron, Ashok Malhotra**, XML Schema Part 2: Datatypes Second Edition, W3C, 2004, Online im Internet, <http://www.w3.org/TR/xmlschema-2/>, Abfrage vom 27.12.2006
- [Bourret, 2006] **Ronald Bourret**, XML Data Binding Resources, 2006, Online im Internet, <http://www.rpbourret.com/xml/XMLDataBinding.htm>, Abfrage vom 08.01.2007
- [Clark, 1999] **James Clark**, XSL Transformations (XSLT), W3C, 1999, Online im Internet, <http://www.w3.org/TR/xslt>, Abfrage vom 12.02.2007
- [Clark, 2003] **James Clark**, RELAX NG, 2003, Online im Internet, <http://relaxng.org/>, Abfrage vom 27.12.2006
- [Costello, 2001] **Roger L. Costello**, Global versus Local, 2001, Online im Internet, <http://www.xfront.com/GlobalVersusLocal.pdf>, Abfrage vom 30.12.2006
- [Cowan und Tobin, 2001] **John Cowan, Richard Tobin**, XML Information Set, W3C, 2001, Online im Internet,

- <http://www.w3.org/TR/2001/WD-xml-infoset-20010316/>, Abfrage vom 27.12.2006
- [Crocker, 1982] **David H. Crocker**, RFC 822 - Standard for the format of ARPA internet text messages, IETF, 1982, Online im Internet, <http://tools.ietf.org/html/rfc822>, Abfrage vom 27.12.2006
- [Eric Jendrock u. a., 2006] **Eric Jendrock u. a.**, The Java EE 5 Tutorial, 2006, Online im Internet, <http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorialFront.html>, Abfrage vom 12.02.2007
- [Fallside und Walmsley, 2004] **David C. Fallside, Priscilla Walmsley**, XML Schema Part 0: Primer Second Edition, W3C, 2004, Online im Internet, <http://www.w3.org/TR/xmlschema-0/>, Abfrage vom 27.12.2006
- [Fielding u. a., 1999] **Roy T. Fielding, u. a.**, RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1, IETF, 1999, Online im Internet, <http://tools.ietf.org/html/rfc2616>, Abfrage vom 19.12.2006
- [Gourley und Totty, 2002] **David Gourley, Brian Totty**, HTTP - The Definitive Guide, O'Reilly, 2002
- [Harold und Means, 2002] **Elliote Rusty Harlod, W. Scott Means**, XML in a nutshell, O'Reilly, 2002
- [Harold, 2002] **Elliote Rusty Harold**, Processing XML with Java, 2002, Online im Internet, <http://www.cafeconleche.org/books/xmljava/>, Abfrage vom 12.02.2007
- [Hunt und Thomas, 2000] **Andrew Hunt und David Thomas**, The Pragmatic Programmer, Addison Wesley, 2000
- [Le Hégaret u. a., 2005] **Philippe Le Hégaret u. a.**, Document Object

- Model (DOM), W3C, 2005, Online im Internet, <http://www.w3.org/DOM/>, Abfrage vom 17.01.2007
- [Leisegang, 2007] Angaben von Herrn **Christoph Leisegang**, Geschäftsführer, Linkwerk GmbH, Hamburg, E-Mail vom 12.02.2007
- [Maler, 2002] **Eve Maler**, Schema Design Rules for UBL...and Maybe for You, 2002, Online im Internet, http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-01-02/05-01-02.html, Abfrage vom 31.12.2006
- [McLaughlin, 2001] **Brett McLaughlin**, Java & XML, O'Reilly, 2001
- [Microsoft Corporation] **Microsoft Corporation**, Capitalization Styles, o.J., Online im Internet, <http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconcapitalizationstyles.asp>, Abfrage vom 12.02.2007
- [Middendorf u. a., 2002] **Stefan Middendorf, Reiner Singer, Jörn Heid**, Java - Programmierhandbuch und Referenz für die Java-2-Plattform SE, dpunkt.Verlag, 2002
- [Sperberg-McQueen und Thompson, 2007] **C. M. Sperberg-McQueen, Henry Thompson**, XML Schema, W3C, 2007, Online im Internet, <http://www.w3.org/XML/Schema>, Abfrage vom 20.01.2007
- [Sun Microsystems] **Sun Microsystems**, Java SE 6 Release Notes - Features and Enhancements, o.A., Online im Internet, <http://java.sun.com/javase/6/webnotes/features.html>, Abfrage vom 12.02.2007

[Thompson u. a., 2004]

Henry S. Thompson u .a., XML Schema Part 1: Structures Second Edition, W3C, 2004, Online im Internet, <http://www.w3.org/TR/xmlschema-1/>, Abfrage vom 27.12.2006

[van der Vlist, 2002]

Eric van der Vlist, XML Schema, O'Reilly, 2002

[Wong, 2000]

Clinton Wong, HTTP Pocket Reference: Hypertext Transfer Protocol, O'Reilly, 2000

[Jelliffe, 2001]

Rick Jelliffe, The Schematron, 2001, Online im Internet, <http://xml.ascc.net/resource/schematron/schematron.html>, Abfrage vom 27.12.2006