
**HTWK Leipzig**

Fakultät für Informatik, Mathematik & Naturwissenschaften

---

# Bachelor-Thesis

## Generierung und Design einer Client-Bibliothek für einen RESTful Web Service am Beispiel der Spreadshirt-API

Author:

**Andreas Linz**

10INB-T

admin@klingt.net

Leipzig, 5. September 2013

---

Gutachter:

Dr. rer. nat. Johannes Waldmann

HTWK Leipzig – Fakultät für Informatik, Mathematik & Naturwissenschaften

waldmann@imn.htwk-leipzig.de

HTWK Leipzig, F-IMN, Postfach 301166, 04251 Leipzig

Jens Hadlich

Spreadshirt HQ, Gießerstraße 27, 04229 Leipzig

jns@spreadshirt.net

Diese Seite wurde mit Absicht leer gelassen.

**Andreas Linz**
Nibelungenring 52
04279 Leipzig
admin@klingt.net
www.klingt.net

***Generierung und Design einer Client-Bibliothek für einen RESTful Web Service am Beispiel der Spreadshirt-API***
Bachelor Thesis, HTWK-Leipzig, 5. September 2013

made with XeTeX, LaTeX and BibTeX.

# Selbständigkeitserklärung

Ich erkläre hiermit, dass ich diese Bachelor-Thesis selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.
Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

.....
**Andreas Linz**
Leipzig, 5. September 2013

# Danksagungen

# Abstract


## Schlüsselwörter

Codegenerierung, RESTful Web Service, Modellierung, Client-Bibliothek, Spreadshirt-API, Polyglot

# Lizenz




[1] http://creativecommons.org/licenses/by-sa/3.0/deed.de

# Inhaltsverzeichnis

**BibTeX Eintrag** **K**

# 1 Einführung

»Essentially, all models are wrong, but some are useful.«

*Empirical Model-Building and Response Surfaces. p. 424*
***George E. P. Box, Norman R. Draper** (1987)*

Das Ziel dieser Arbeit ist die Erstellung eines Codegenerators, der aus der abstrakten Beschreibung der Spreadshirt-API eine Client-Bibliothek erstellt.

Der Generator soll eine flexible Wahl der Zielsprache bieten, wobei mit »Zielsprache« im folgenden die Programmiersprache der erzeugten Bibliothek gemeint ist. Für das Bibliotheksdesign ist eine DSL (Domain-Specific Language) zu realisieren, mit dem Ziel die Nutzung der API zu vereinfachen.

Als Programmiersprache für den Generator wird *Java* verwendet, *PHP* ist die Zielsprache der Bibliothek. Eine gute Lesbarkeit, hohe Testabdeckung und größtmögliche Typsicherheit, soweit *PHP* dies zulässt, sind Erfolgskriterien für die zu generierende Bibliothek.

Abbildung 1.1 stellt den schematischen Aufbau des gewünschten Generators dar.

Spreadshirt ist eines der führenden Unternehmen für personalisierte Kleidung und zählt zu den *Social Commerce*-Unternehmen. Dieser Begriff beschreibt Handelsunternehmen bei denen die aktive Beteiligung und die persönliche Beziehung, sowie Kommunikation der Kunden untereinander, im Vordergrund

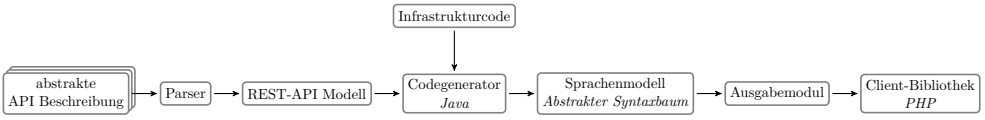


**Abbildung 1.1:** Aufbau des Generatorsystems


**Abbildung 1.2:** Spreadshirt Logo

stehen. Spreadshirt hat Standorte in Europa und Nordamerika, der Hauptsitz ist in Leipzig.

wird Nutzern eine Online-Plattform geboten um Kleidungsstücke selber zu gestalten oder zu kaufen, oder auch um eigene Designs, als Motiv oder in Form von Produkten, zum Verkauf anzubieten. Es wird jedem Nutzer ermöglicht einen eigenen Shop auf der Plattform zu eröffnen und ihn auf der eigenen Internetseite einzubinden. Derzeit gibt es rund 400.000 Spreadshirt-Shops mit ca. 33.000.000 Produkten. Für die Spreadshirt-API können Kunden eigene Anwendungen schreiben, bspw. *zufallsshirt.de* [Pas13] oder *soundslikecotton.com* [Ött13]. Neben dem Endkunden- bedient Spreadshirt auch das Großkundengeschäft als Anbieter von Druckleistungen.

Die *sprd.net AG* zu der auch der Leipziger Hauptsitz gehört beschäftigt derzeit (Juni 2013) 178 Mitarbeiter, davon 29 in der IT.

Die zwei wichtigsten Konstanten in der Anwendungsentwicklung sind laut [Her03] folgende:

- Die Zeit eines Programmierers ist kostbar
- Programmierer mögen keine langweiligen und repetitiven Aufgaben

Codegenerierung greift bei beiden Punkten an und kann zu einer Steigerung der *Produktivität* führen, die durch herkömmliches schreiben von Code nicht zu erreichen wäre.

Änderungen können an zentraler Stelle vorgenommen und durch die Generierung automatisch in den Code übertragen werden, was mit verbesserter *Wartbarkeit* einhergeht. Die gewonnenen Freiräume kann der Entwickler nutzen um sich mit den Grundlegenden Herausforderungen und Problemen seiner Software zu beschäftigen.

Durch die Festlegung eines Schemas für Variablennamen und Funktionssignaturen, wird eine hohe *Konsistenz*, über die gesamte Codebasis hinweg, erreicht. Diese Einheitlichkeit vereinfacht auch die Nutzung des Generats[1], da beispielsweise nicht mit Überraschungen bei den verwendeten Bezeichnern zu rechnen ist.

Als Eingabe für den Generator dient ein *abstraktes Modell* des betreffenden Geschäftsbereiches. Die Erstellung eines solchen Modells vertieft das Verständnis des Entwicklers für das Geschäftsfeld und gibt gleichzeitig Spezialisten aus dem Fachbereich die Möglichkeit Fragestellungen anhand dieses Modells zu formulieren.

Um die immer kürzer werdenden Entwicklungszyklen einhalten zu können, kann durch Codegenerierung die nötige Effizienzsteigerung geleistet werden.

[1] Ergebnis des Codegenerierungsvorgangs

# 2 Web Services

»The purpose of computing is insight, not numbers.«

*Numerical Methods for Scientists and Engineers. Preface*
***Richard Hamming** (1962)*

In diesem Kapitel werden die Grundlagen zu HTTP, Dokumentbeschreibungssprachen, Webanwendungsbeschreibungsformaten und REST erläutert, welche für das Verständnis der Arbeit wichtig sind. Neben XML und JSON werden auch Schemabeschreibungssprachen wie XSD und RelaxNG behandelt. Das Ende bildet die Einführung in REST und WADL.

## 2.1 HTTP

> **Definition 1** (HTTP)**.** *Das* Hypertext Transfer Protocol (HTTP) *ist allgemeines und* zustandsloses *Protokoll, zur Übertragung von Daten über ein Netzwerk, was durch Erweiterung seiner Anfragemethoden, Statuscodes und Header für viele unterschiedliche Anwendungen verwendet werden kann ([Fie+99, Abstract]).*

HTTP arbeitet auf der Anwendungsschicht[1] und ist somit unabhängig von dem zum Datentransport eingesetzten Protokoll. Über eindeutige URIs werden HTTP-Ressourcen angesprochen. Dabei sendet ein *Client* eine Anfrage (*request*) und erhält daraufhin vom Server eine Antwort (*response*). Anfrage und Antwort stellt dabei eine HTTP-Nachricht dar, die aus den zwei Elementen *Header* und *Body* besteht. Letzterer trägt die Nutzdaten und kann, je nach verwendeter HTTP-Methode, auch leer sein.

[1] OSI-Modell Level 7

*RFC 2616, Hypertext Transfer Protocol – HTTP/1.1* ([Fie+99]) definiert einige HTTP-Methoden, wobei die gebräuchlichsten die folgenden sind:

- GET
- PUT
- POST
- DELETE

Eine Nachricht kann je nach verwendeter HTTP-Methode auch nur aus einem Header bestehen.

### 2.1.1 Header

Ein Header einer HTTP-Nachricht besteht aus einer *Request Line* (erste Zeile des Headers) und einer Menge von Schlüssel-Wert Paaren. Listing 2.1 zeigt einen Beispiel Header für die GET Anfrage auf die Spreadshirt-API Ressource: `http://api.spreadshirt.net/api/v1/locales`.

```
1 GET ❶ /api/v1/locales ❷ HTTP/1.1 ❸
2 User-Agent: curl/7.29.0 ❹
3 Host: api.spreadshirt.net ❺
4 Accept: */* ❻
```

**Listing 2.1:** HTTP-Header von GET Request auf Spreadshirt-API Ressource `http://api.spreadshirt.net/api/v1/locales`

❶ Angabe der HTTP-Methode
❷ Ressource
❸ verwendete HTTP-Version
❹ *User-Agent*, Angabe zum Client-System das die Anfrage versendet
❺ *Host*, Server der die Anfrage erhält und der die Ressource ❷ verwaltet
❻ Angabe von *Content-Types* die der Client als Antwort akzeptiert, in diesem Fall eine *Wildcard*, also alle Typen sind als Antwort erlaubt

Nachfolgend die *Response* auf den soeben beschrieben *Request*.

```
1 HTTP/1.1 200 OK ❶
2 Expires: Tue, 20 Aug 2013 19:05:25 GMT
```

```
3  Content-Language: en-gb
4  Content-Type: application/xml;charset=UTF-8 ❷
5  X-Cache-Lookup: MISS from fish07:80
6  X-Server-Name: mem1
7  True-Client-IP: 88.79.226.66
8  Date: Tue, 20 Aug 2013 07:20:25 GMT
9  Content-Length: 1659
10 Connection: keep-alive
```

**Listing 2.2:** HTTP-Header von GET Response aus Spreadshirt-API Ressource `http://api.spreadshirt.net/api/v1/locales`

❶ *Response Line*, Angabe der HTTP-Version am Anfang und danach der HTTP-Statuscode mit Kurzbeschreibung

❷ Angabe des Content-Types des Bodys der Nachricht

Welche Einträge der Header einer HTTP-Nachricht letztendlich enhtält, ist abhängig von der Implementierung des Clients oder Servers und es können auch jederzeit eigene Einträge, die nicht in der HTTP-Spezifikation enthalten sind, hinzugefügt werden.

### 2.1.2 Body

Der Body enthält die eigentlichen Nutzdaten. Deren Format wird mit dem *Content-Type* Eintrag des Headers angegeben. Listing 2.3 zeigt den Body der *response* von listing 2.2 in XML Format (siehe Abschnitt 2.2.1).

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <locales xmlns:xlink="http://www.w3.org/1999/xlink" xmlns="http://api.
       spreadshirt.net" xlink:href="http://api.spreadshirt.net/api/v1/
       locales" offset="0" limit="50" count="16" sortField="default"
       sortOrder="default">
3      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/de_DE"
           id="de_DE"/>
4      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/fr_FR"
           id="fr_FR"/>
5      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/en_GB"
           id="en_GB"/>
6      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/nl_NL"
           id="nl_NL"/>
7      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/it_IT"
           id="it_IT"/>
```

```
8       <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/no_NO"
            id="no_NO"/>
9       <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/se_SE"
            id="se_SE"/>
10      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/es_ES"
            id="es_ES"/>
11      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/en_EU"
            id="en_EU"/>
12      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/en_IE"
            id="en_IE"/>
13      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/dk_DK"
            id="dk_DK"/>
14      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/pl_PL"
            id="pl_PL"/>
15      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/fi_FI"
            id="fi_FI"/>
16      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/de_AT"
            id="de_AT"/>
17      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/fr_BE"
            id="fr_BE"/>
18      <locale xlink:href="http://api.spreadshirt.net/api/v1/locales/nl_BE"
            id="nl_BE"/>
19  </locales>
```

**Listing 2.3:** HTTP-Body von GET Response aus Spreadshirt-API Ressource `http://api.spreadshirt.net/api/v1/locales`

## 2.2 Dokumentbeschreibungsformate

In diesem Abschnitt werden die Dokumentbeschreibungsformate XML und JSON, der Spreadshirt-API behandelt. Außerdem werden die Schemabeschreibungssprachen *XML Schema Description* und *RelaxNG* eingeführt.

### 2.2.1 XML

> **Definition 2** (XML). *Die* Extensible Markup Language*, kurz XML, ist eine Auszeichnungssprache (»Markup Language«) die eine Menge von Regeln beschreibt um Dokumente in einem mensch- und maschinen lesbaren Format zu kodieren [W3C08].*

Obwohl das Design von XML auf Dokumente ausgerichtet ist, wird es häufig für die Darstellung von beliebigen Daten benutzt [Wik13], z.B. um diese für die Übertragung zu serialisieren.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?> ❶
2  <design xmlns:xlink="http://www.w3.org/1999/xLink"
3          xmlns="http://api.spreadshirt.net"
4          ...>❷
5      <name>tape_recorder</name>
6      ...
7      <size unit="px">
8          <width>3340.0</width>
9          <height>3243.0</height>
10     </size>
11     <colors/>❸
12     ...
13     <created>
14         2013-03-30T12:37:54Z ❹
15     </created>
16     <modified>2013-04-02T11:13:02Z</modified>
17 </design>❺
```

**Listing 2.4:** Die gekürzte Antwort der API-Ressource *users/userid/designs/designID* als Beispiel für eine XML-Datei

Eine valide XML-Datei beginnt mit der *XML-Deklaration* ❶, diese enthält Angaben über die verwendete XML-Spezifikation und die Kodierung der Datei. Im Gegensatz zu gewöhnlichen Tags, wird dieses mit `<?` und mit `?>` beendet. Danach folgen beliebig viele baumartig geschachtelte *Elemente* mit einem Wurzelelement ❷. Die Elemente können Attribute enthalten und werden, wenn sie kein leeres Element sind ❸, von einem schließenden Tag in der gleichen Stufe abgeschlossen ❺. Nicht leere Zeichenketten als Kindelement sind ebenfalls erlaubt ❹.

Mit Hilfe von *Schemabeschreibungssprachen* (siehe Abschnitt 2.3) kann der Inhalt und die Struktur eines Dokumentes festgelegt und gegen diese validiert werden. Der Begriff *XML Schema* ist mehrdeutig und wird oft auch für eine konkrete Beschreibungssprache, die »XML Schema Definition«, kurz XSD, verwendet.

| primitiv | strukturiert |
|---|---|
| Zeichenketten | Objekte |
| Ganz- und Fließkommazahlen | Arrays |
| Booleans | |
| null | |

**Tabelle 2.1:** JSON Datentypen

### 2.2.2 JSON

> **Definition 3** (JSON)**.** Javascript Object Notation, *kurz* JSON, *ist ein leichtgewichtiges, textbasiertes und sprachunbhängiges Datenaustauschformat. Es ist von JavaScript abgeleitet und definiert eine kleine Menge von Formatierungsregeln für die transportable Darstellung (Serialisierung) von strukturierten Daten (nach [Cro06]).*

Im Gegensatz zu XML ist JSON weit weniger mächtig, es gibt z.B. keine Unterstützung für Namensräume und es wird nur eine geringe Menge an Datentypen unterstützt (siehe Tabelle 2.1). Durch seine einfache Struktur wird aber ein deutlich geringerer »syntaktischer Overhead« erzeugt. Mit *JSON Schema* [GZ13] ist es möglich eine Dokumentstruktur vorzugeben und gegen diese zu validieren.

Objekte werden in JSON von geschweiften- ❶, Arrays hingegen von eckigen Klammern begrenzt ❷. Objekte enthalten *key-value-pairs* (Schlüssel-Wert-Paare) ❸. Schlüssel sind immer Zeichenketten, die Werte dürfen von allen Typen aus Tabelle 2.1 sein und beliebig tief geschachtelt werden.

```
1 {
2     "name": "tape_recorder", ❸
3     "description": "",
4     "user": { ❶
5         "id": "1956580",
6         "href": "http://api.spreadshirt.net/api/v1/users/1956580"
```

```
7       }, ❶
8       "resources": [ ❷
9       ...
10          {
11              "mediaType": "png",
12              "type": "preview",
13              "href": "http://image.spreadshirt.net/image-server/v1/designs
                    /15513946"
14          },
15          ...
16      ], ❷
17      "created": "30.03.2013␣12:37:54",
18      ...
19  }
```

**Listing 2.5:** Die gekürzte Antwort der API-Ressource *users/userid/designs/designID* als Beispiel für eine JSON-Datei

## 2.3 XML Schemabeschreibungssprachen (XML Schema)

*XML Schema* bezeichnet XML-basierte Sprachen mit denen sich Elemente, Attribute und Aufbau einer Menge von XML-Dokumenten — die dem Schema entsprechen — beschreiben lassen. Ein XML-Dokument wird als *valid* gegenüber einem Schema bezeichnet, falls die Elemente und Attribute dieses Dokumentes die Bedingungen des Schemas erfüllen [Mur+05]. Neben XSD und RelaxNG (siehe Abschnitt 2.3.2) existieren noch weitere Schemasprachen, die hier aber aufgrund ihrer geringen Relevanz nicht behandelt werden. Die beiden hier behandelten Schemasprachen bieten den Vorteil selbst XML-Dokumente zu sein, somit können sie durch herkömmliche XML-Tools bearbeitet werden.

### 2.3.1 XML Schema Description (XSD)

*XML Schema Description* ist ein stark erweiterte Nachfolger der *DTD* (Document Type Definition), derzeit spezifiert in Version 1.1 [W3C12]. Die Syntax von *XSD* ist XML, damit ist die Schemabeschreibung ebenfalls ein gültiges XML-Dokument. Als Dateiendung wird üblicherweise `.xsd` verwendet.

Die Hauptmerkmale von XSD sind nach [Mur+05] , die folgenden:

- Komplexe Typen (strukturierter Inhalt)

- anonyme Typen (besitzen kein `type`-Attribut)
- Modellgruppen
- Ableitung durch Erweiterung oder Einschränkung (»derivation by extension/restriction«)
- Definition von abstrakten Typen
- Integritätsbedingungen (»integrity constraints«):
  *unique*, *keys* und *keyref*, dies entspricht den *unique*-, *primary*- und *foreign*-keys aus dem Bereich der Datenbanken

Die XSD Spezifikation enthält bereits eine Menge vordefinierter Datentypen, dargestellt in Abbildung 2.1.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    version="1.0"
    targetNamespace="myNamespace"
    elementFormDefault="qualified">
    <xsd:complexType name="product">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="price" type="xsd:decimal"/>
            <xsd:element name="description"
                ref="myNamespace:description"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="description">
        <xsd:all>
            <xsd:element name="title" type="xsd:string">
            <xsd:element name="content" type="xsd:string">
        </xsd:all>
    </xsd:complexType>
</xsd:schema>
```

**Listing 2.6:** Minimalbeispiel für eine Schemabeschreibung mit XSD

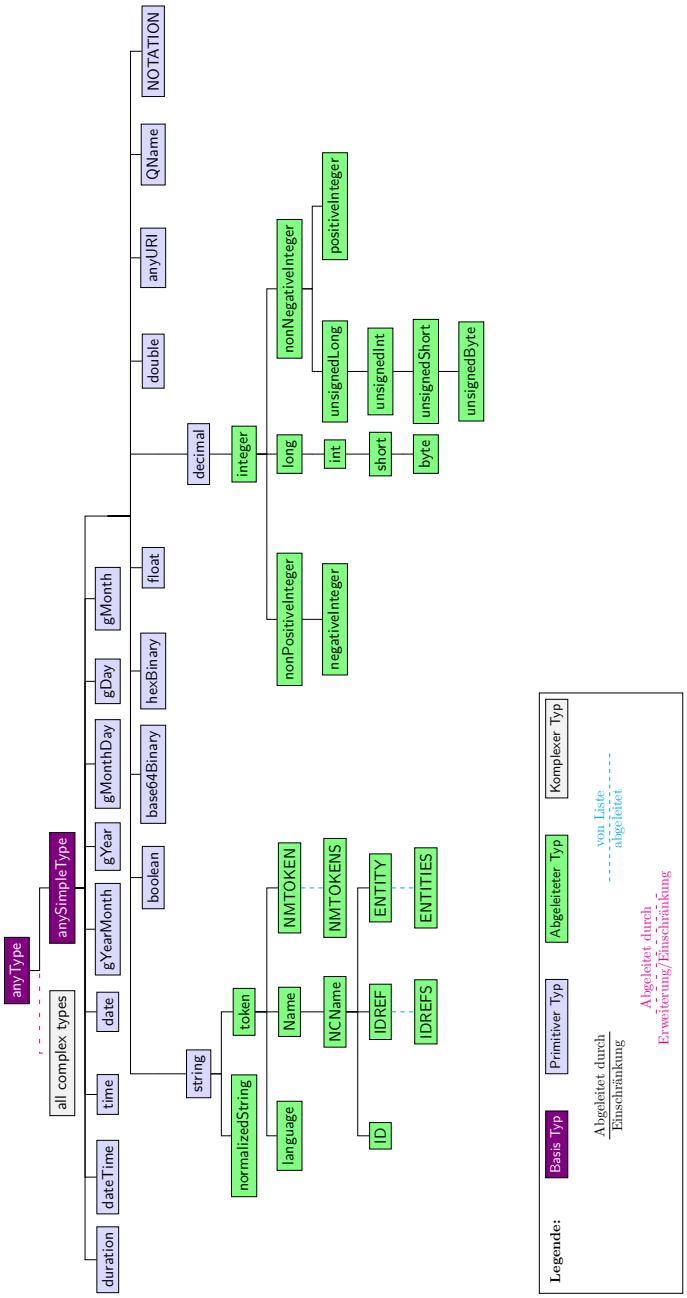


**Abbildung 2.1:** vordefinierte XSD Datentypen nach [W3C12] Kapitel 3

### 2.3.2 RelaxNG

Ebenso wie XSD (siehe Abschnitt 2.3.1) ist die *Regular Language Description for XML New Generation* eine XML-Schemasprache zur Definition der Struktur von XML-Dokumenten. Schemas werden in *RelaxNG* durch XML-Syntax oder eine eigene, kompaktere nicht-XML Syntax formuliert. Ebenso wie bei *XML Schema* werden Namespaces unterstützt. RelaxNG Schemabeschreibungen verwenden meist `.rng` als Dateiendung.

Unterschiede zu XML Schema:

- Unterstützung von ungeordneten Inhalten
- kompaktere nicht-XML Syntax
- *nichtdeterministisches* oder auch *mehrdeutiges* Inhaltsmodell [Vli04, Kapitel 16]

```
<?xml version="1.0" encoding="utf-8"?>
<grammar
    xmlns="http://relaxng.org/ns/structure/1.0"
    ns="myNamespace">
    <start>
        <ref name="product"/>
    </start>
    <define name="product">
        <oneOrMore>
            <element name="name"/>
                <text/>
            </element>
            <element name="price"/>
                <text/>
            </element>
            <ref name="description"/>
        </oneOrMore>
    </define>
    <define name="description">
        <oneOrMore>
            <element name="title">
                <text/>
            </element>
            <element name="content">
                <text/>
            </element>
```

```
27          </oneOrMore>
28      </define>
29  </grammar>
```

**Listing 2.7:** Minimalbeispiel für eine Schemadefinition in RelaxNG

## 2.4 RESTful Web Service

*Representational State Transfer* (deutsch: »Gegenständlicher Zustandstransfer«) ist ein Softwarearchitekturstil für Webanwendungen, welcher von Roy Fielding[2] in seiner Dissertation aus dem Jahre 2000 beschrieben wurde [Kapitel 5 Fie00, 95 ff.].

Als RESTful bezeichnet man dabei eine Webanwendung die den Prinzipien von REST entspricht.

### 2.4.1 Elemente von REST

Im folgenden werden die Grundbausteine einer REST-Anwendung erläutert. Abschnitt 2.4.1.4 beschreibt die Komponenten, die an einer Aktion auf einer *Ressource* beteiligt sind. Dieser Abschnitt basiert auf Kapitel 5.2 von [Fie00, S. 86 ff.].

#### 2.4.1.1 Ressource

Eine Ressource stellt die wichtigste Abstraktion von Information im REST-Modell dar. Fielding definiert eine *ressource* wie folgt:

> *»Any information that can be named can be a resource: a document or image, [...]. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.«*

Eine Ressource kann somit alle Konzepte abbilden die sich über einen Bezeichner referenzieren lassen. Dies können konkrete Dokumente, aber auch Dienste oder sogar Sammlungen von Ressourcen sein. Außerdem identifiziert

[2] Roy Thomas Fielding, geboren 1965, ist einer der Hauptautoren der HTTP-Spezifikation

ein Ressourcenbezeichner, meist eine *URI* (siehe Abschnitt 2.4.2.1), immer dieselbe Ressource, nicht aber deren Wert oder Zustand.

Ein Beispiel für eine Ressource in der Spreadshirt API `/users/{userid}`, wobei `{userid}` die ID eines konkreten Nutzers bezeichnet. Diese Ressource enthält dabei eine Menge von Elementen, ein Beispiel für eine Ressource die nur ein einziges Element enthält ist `/serverTime`.

#### 2.4.1.2 Repräsentation

Eine Repräsentation (*representation*) stellt den aktuellen oder den gewünschten Zustand einer Ressource dar. Das Format der Repräsentation ist dabei unabhängig von dem der Ressource, siehe Abschnitt 2.4.2.4. Aktionen mit Komponenten einer REST-API werden durch den Austausch von solchen Repräsentationen durchgeführt.

Im Allgemeinen wird unter einer Repräsentation nur eine Folge von Bytes verstanden, inklusive *Metainformationen* welche den Inhalt der Bytefolge klassifiziert, sowie *Kontrolldaten* die die gewünschte Aktion oder die Bedeutung der Anfrage beschreiben. Letztere sind meist HTTP-Header-Felder (siehe Abschnitt 2.1.1), bspw. um das Cachingverhalten zu ändern.

```
1     ...
2     <response>
3         <representation xmlns:sns="http://api.spreadshirt.net",
4             element="sns:productTypes" status="200", ❶
5             mediaType="application/xml"> ❷
6             <doc title="Success"/>
7         </representation>
8         ...
9     </response>
```

**Listing 2.8:** Beispiel zu Metainformationen für REST-Repräsentation aus WADL-Datei der Spreadshirt-API

Ein Beispiel für eine solche Angabe von Metainformationen ist in Listing 2.8 zu finden, ❶ zeigt dies in Form einer *Typangabe* und ❷ eines *mediaType*-Attributes.

| Konnektor | Beispiel |
|---|---|
| client | libwww |
| server | libwww, Apache HTTP-Server API |
| cache | browser cache, Akamai |
| resolver | bind |
| tunnel | SOCKS |

**Tabelle 2.2:** Beispiele für Konnektoren nach [Fie00]

#### 2.4.1.3 Konnektoren

Konnektoren stellen eine Schnittstelle für die Kommunikation mit Komponenten der REST-Webanwendung dar. Aktionen auf Ressourcen und der Austausch von Repräsentationen finden über diese Schnittstellen statt. Der Konnektor bildet die Parameter der Schnittstelle auf das gewünschte Protokoll ab.

Eingangsparameter[3]:

- Anfrage-Kontrolldaten
- Ressourcenidentifizierung (Ressourcenbezeichner)
- · Repräsentation der Ressource

Ausgangsparameter:

- Antwort-Kontrolldaten
- · Metainformationen
- · Repräsentation der Ressource

Tabelle 2.2 listet Beispiele für Konnektoren auf.

#### 2.4.1.4 Komponenten

***Ursprungsserver:***

Serverseitiger Konnektor der die angeforderten Ressourcen verwaltet. Er

[3] · optionaler Parameter

ist die einzige Quelle für Repräsentationen, sowie der endgültige Empfänger von Änderungsanfragen an seine Ressourcen. (siehe Abschnitt 2.4.2.1).

***Proxy:***
Zwischenkomponente die explizit vom Client verwendet kann, aus Sicherheits-, Performance- oder Kompatibilitätsgründen.

***Gateway:***
Dient als Schnittstelle zwischen Client- und Servernetzwerk, kann zusätzlich aus den gleichen Gründen wie der Proxy verwendet werden. Konträr zum Proxy kann der Client aber nicht entscheiden ob er einen Gateway nutzen möchte.

***User Client:***
Ein clientseitiger *Konnektor*, der die Anfrage an die API startet und einziger Empfänger der Antwort ist. In den meisten Fällen ist dies einfach ein *Webbrowser*.

### 2.4.2 REST-Prinzipien

Die fünf Grundlegenden REST-Prinzipien, nach [Til09, 11 ff.]:

- Ressourcen mit eindeutiger Indentifikation
- Verknüpfungen / Hypermedia
- Standardmethoden (siehe Abschnitt 2.4.2.3)
- unterschiedliche Repräsentationen
- statuslose Kommunikation

#### 2.4.2.1 Eindeutige Identifikation

Um eine *eindeutige Identifikation* zu erreichen, wird jeder Ressource eine URI vergeben. Dadurch ist es möglich zu jeder verfügbaren Ressource einen Link zu setzen. Nachfolgend eine Beispiel-URI, um den Artikel 42 aus dem Warenkorb

84 anzusprechen:

$$\underbrace{http://api.spreadshirt.net/api/v1/}_{Basis-URL}\underbrace{\overbrace{baskets}^{Warenkorb}/84/\overbrace{item}^{Artikel}/42}_{Ressource}$$

#### 2.4.2.2 Hypermedia

Innerhalb einer Ressource kann auf weitere verlinkt werden (*Hypermedia*). Als Nebeneffekt der eindeutigen Identifikation durch URIs sind diese auch außerhalb des Kontextes der aktuellen Anwendung gültig.

#### 2.4.2.3 Standardmethoden

Durch die Nutzung von *Standardmethoden* ist abgesichert das eine Anwendung mit den Ressourcen arbeiten kann, vorausgesetzt sie unterstützt diese.

REST ist nicht auf HTTP beschränkt, praktisch alle REST-APIs nutzen aber dieses Protokoll. HTTP umfasst dabei folgende Methoden[4]:

- GET
- PUT
- POST
- DELETE
- HEAD
- OPTIONS

Alle bis auf *POST* und *OPTIONS* sind *idempotent* ([Fie+99] Kapitel 9), d.h. eine hintereinander Ausführung der Methode führt zu demselben Ergebnis wie ein einzelner Aufruf. Dies bedeutet das sich ein *RESTful Web Service* serverseitig ebenso verhalten muss.

#### 2.4.2.4 Repräsentationen von Ressourcen

Die Repräsentation sollte unabhängig von der Ressource sein, um die Darstellung gegebenenfalls für den Client anzupassen. Die Client-Anwendung kann

[4] Kapitel 9 des HTTP 1.1 RFC2616 beschreibt diese inklusive *TRACE* und *CONNECT* umfassend [Fie+99]

dadurch mittels *Query-Parameter* oder als Information im *HTTP-Header* (siehe Abschnitt 2.1.1) das gewünschte Format angeben und erhält die entsprechend formatierte Antwort. Anhand des *Content-Type* Feldes aus dem HTTP-Header kann der Client das Format der Antwort überprüfen, für JSON lautet dies bspw. `application/json`.

#### 2.4.2.5 Statuslose Kommunikation

Es soll kein Sitzungsstatus (*session-state*) vom Server gespeichert werden, d.h. jede Anfrage des Client muss alle Informationen enthalten, die nötig sind um diese serverseitig verarbeiten zu können. Der Sitzungstatus wird dabei vollständig vom Client gehalten. Diese Restriktion führt zu einigen Vorteilen:

- Verringerung der Kopplung zwischen Client und Server
- zwei aufeinanderfolgende Anfragen können von unterschiedlichen Serverinstanzen beantwortet werden

↪ verbesserte Skalierbarkeit

Daraus resultiert eine erhöhte Netzwerklast, da die Statusinformationen bei jeder Anfrage mitgesendet werden müssen.

## 2.5 WADL

Die *Web Application Description Language* (kurz WADL) ist eine maschinenlesbare Beschreibung einer HTTP-basierten Webanwendung, einschließlich einer Menge von *XML Schematas* [Had06]. Die aktuelle Revision ist vom 31. Aug. 2009 [Had09a], im weiteren beziehe ich mich aber auf die in der Spreadshirt-API verwendeten Version, datiert am 9. November 2006. Die Unterschiede zwischen beiden Revisionen können unter [Had09b] nachvollzogen werden.

Die Beschreibung eines Webservices durch WADL besteht nach [Had06] im groben aus den folgenden vier Bestandteilen:

***Set of resources:***

Analog einer Sitemap, die Übersicht aller verfügbaren Ressourcen.

***Relationships between resources:***

Die kausale und referentielle Verknüpfung zwischen Ressourcen.

***Methods that can be applied to each resource:***

Die von der jeweiligen Ressource unterstützten [HTTP]-Methoden, deren Ein- und Ausgabe, sowie die unterstützten Formate.

***Resource representation formats:***

Die unterstützten MIME-Typen und verwendeten Datenschemas (Abschnitt 2.3.1).

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?> ❶
2  <application xmlns="http://research.sun.com/wadl/2006/10"> ❷
3      <grammars> ❸
4          <include href="http://api.spreadshirt.net/api/v1/metaData/api.xsd
               ">
5              <doc>Catalog XML Schema.</doc>
6          </include>
7          ...
8      </grammars>
9      <resources base="http://api.spreadshirt.net/api/v1/"> ❹
10         <resource path="users/{userId}"> ❺
11             <doc>Return user data.</doc>
12             <method name="GET"> ❻
13                 <doc>...</doc>
14                 <request> ❼
15                     <param
16                         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
17                         name="mediaType"
18                         style="query"
19                         type="xsd:string">
20                     <doc>...</doc>
21                     </param>
22                     ...
23                 </request>
24                 <response> ❽
25                     <representation
26                         xmlns:sns="http://api.spreadshirt.net"
27                         element="sns:user"
28                         status="200"
29                         mediaType="application/xml">
30                     <doc title="Success"/>
31                     </representation>
```

```
32                      <fault status="500" mediaType="text/plain">
33                          <doc title="Internal␣Server␣Error"/>
34                      </fault>
35                      ...
36                      }
37                  ...
```

**Listing 2.9:** Beispielaufbau einer WADL-Datei anhand der Spreadshirt-API Beschreibung

Die Datei beginnt mit der Angabe der XML-Deklaration ❶. Die Attribute des Wurzelknotens `<application>` enthalten *namespace* Definitionen, u. a. auch den der verwendeten WADL-Spezifikation ❷. Innerhalb des `<grammars>` Elements werden die benutzten *XML Schemas* angegeben ❸. Um die Ressourcen der Webanwendung ansprechen zu können wird noch die Angabe der Basisadresse benötigt ❹. Innerhalb des `<resources>` Elements findet sich die Beschreibung der einzelnen Ressourcen. Diese sind gekennzeichnet, durch eine zur Basisadresse relativen URI ❺. In `{…}` eingeschlossene Teile einer URI, werden durch den Wert des gleichnamigen *request* Parameters ersetzt um die URI zu bilden (generative URIs). Im Folgenden werden die von der Ressource unterstützten HTTP-Methoden beschrieben ❻, deren Anfrageparameter `<request>` ❼, sowie die möglichen Ausgaben der jeweiligen Methode `<response>` ❽.

Die Dokumentations-Tags `<doc>` sind für alle XML-Elemente optional. Um das Listing nicht unnötig zu verlängern wurden die schließenden *Tags* weggelassen.

Abbildung 2.2 zeigt die Struktur einer WADL-Datei.

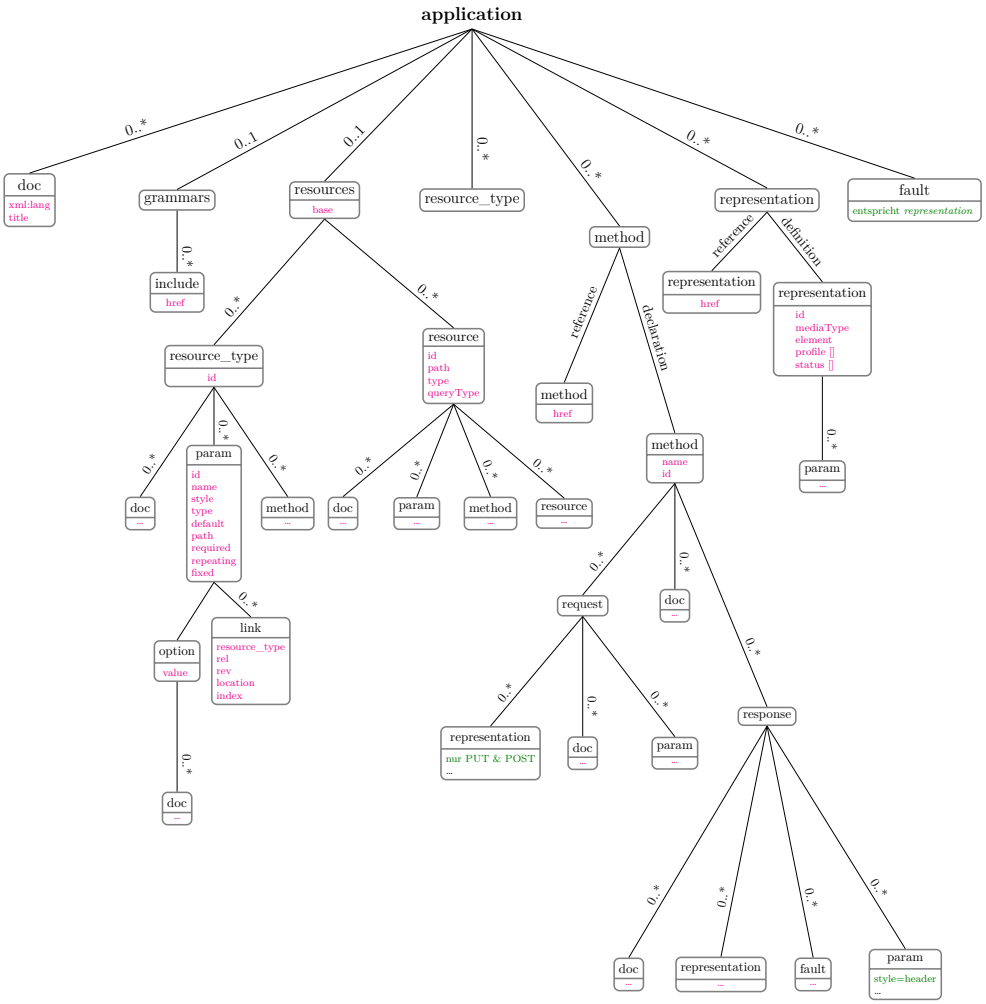


**Abbildung 2.2:** Struktur einer WADL-Datei, nach Kapitel 2 [Had06]

# 3 Codegenerierung

»Any problem in computer science can be solved with another level of indirection.«

*Turing Award Lecture. February 17, 1993.*
***David Wheeler** (1993)*

Im folgenden Kapitel werden grundlegende Begriffe im Zusammenhang mit Codegeneratoren und Datenmodellen definiert, zusätzlich wird eine Übersicht über gebräuchliche Generatorformen, dessen Aufgaben und Arten der Optimierung durch den Generator, gegeben. Abschnitt 3.2 …

## 3.1 Codegeneratoren

Der Begriff »Generator« ist sehr allgemein und wird für verschiedene Technologien verwendet, wie *Compiler*, *Präprozessoren*, *Metafunktionen* (Template-Metaprogramming in C++), *Codetransformatoren* und natürlich *Codegeneratoren*. *Generator* und *Codegenerator* werden in diesem Kapitel synonym verwendet.

> **Definition 4** (Codegenerator)**.** *Ein* Codegenerator *ist ein Programm, welches aus einer höhersprachigen Spezifikation*[1] *einer Software oder eines Teilaspektes, die Implementierung erzeugt (nach [CE00]).*

Folglich ist der Generator die Schnittstelle zwischen dem *Modell-* und *Implementationsraum*. Der Modellraum beinhaltet das *domänenspezifische Modell*. Dieses Modell wird durch die höhersprachige Spezifikation in einer *Systemspezifikationssprache* beschrieben. In Bezug auf einen RESTful Webservice ist

bspw. WADL inklusive seiner verwendeten Schemata[2] die Spezifikationssprache und eine WADL-Datei mit den konkreten Spezifikationen demzufolge das domänenspezifische Modell.

Der Informationsgehalt der Spezifikation ist ausschlaggebend für den Grad der zu erreichenden Automatisierung.

### 3.1.1 Aufgaben eines Generators

optional) Analyse der Spezifikation

1. Validieren der Spezifikation
2. Spezifikation durch Vorgabewerte vervollständigen
3. Optimierungen vornehmen
4. Implementierung erzeugen

Je nach Form der Spezifikation, muss diese durch einen Analyse-Schritt (*parsing*) in die interne Darstellung des Generators überführt werden, bspw. bei einem Compiler in einen *Abstrakten Syntaxbaum* (siehe Abschnitt 3.2.1).

### 3.1.2 Vorteile für den Entwickler

Bei der Nutzung eines Codegenerierungsansatzes ergeben sich nach [Her03, S. 15] folgende Vorteile für den Entwickler:

***Qualität:***
Bugfixes und Verbesserungen werden durch das Generatorsystem in die gesamte Codebasis propagiert.

***Konsistenz:***
Durch ein vorgegebenes Schema für die Schnittstellen- und Variablenbezeichner wird eine hohe Einheitlichkeit erreicht.

***zentrale Wissensbasis:***
Das domänenspezifische Wissen wird in dem Meta- oder auch domänenspezifischen Modell gebündelt, das dem Generator als Eingabe dient. Än-

[2] siehe Abschnitt 2.2

derungen am Modell werden durch den Generator in die gesamte Codebasis eingepflegt.

***signifikantere Designentscheidungen:***
Aufgrund des verringerten Implementierungsaufwandes kann der Entwickler mehr Zeit für das Design seiner Architektur, API etc. verwenden. Designfehlentscheidungen können durch Änderungen am Modell oder auch Templates[3] korrigiert werden und bedürfen somit keiner manuellen Korrektur aller generierten Klassen.

Die Erstellung eines Generatorsystems geht mit einem nicht unerheblichen Aufwand einher, dieser sollte in Relation zum Umfang des zu erzeugenden Codes gesehen werden. Ist der Umfang des Erzeugnisses zu gering, kann der Aufwand zur Entwicklung einer Generatorlösung kontraproduktiv sein.

### 3.1.3 Generatorformen

Die folgende Tabelle klassifiziert einige Generatorformen nach der Menge des erzeugten Codes:

| teilweise | vollständig | mehrfach |
|---|---|---|
| Inline-Code Expander | Tier-Generator | n-Tier Generator |
| Mixed-Code Generator | | |
| Partial-Class Generator | | |

**Tabelle 3.1:** Generatoren Klassifikation nach Generierungsmenge

Herrington beschreibt in [Kapitel 4 Her03] die Formen aus Tabelle 3.1 so:

***Inline-Code Expander:***
Ein Inline-Code Expander nimmt Quellcode als Eingabe und ersetzt spezielle Mark-Up Sequenzen durch seine Ausgabe. Die Änderungen werden hierbei nicht in das Quellfile übernommen sondern meist direkt zu dem Compiler oder Interpreter weitergeleitet.

[3] Falls der Generator seinen Code über einen Templateansatz erzeugt.

***Mixed-Code Generator:***

Der Mixed-Code Generator arbeitet grundsätzlich wie der Inline-Code Expander, seine Änderungen werden aber in die Quelle zurückgeschrieben.

***Partial-Class Generator:***

Partial-Class Generatoren erzeugen aus einer abstrakten Beschreibung und Templates einen Satz von Klassen, diese bilden aber nicht das vollständige Programm sondern werden durch manuell erzeugten Code vervollständigt.

***Tier-Generator:***

Die Arbeitsweise des Stufen- oder Tier[4]-Generators entspricht der des Partial-Class Generators, mit der Ausnahme das vollständiger Code erzeugt wird, der keiner Nacharbeit bedarf.[5]

***$n$-Tier Generator:***

Ein $n$-Tier Generator erzeugt neben dem eigentlichen Quellcode noch beliebige andere Informationen, bspw. eine Dokumentation oder Testfälle.

Die Entwicklung einer »Full-Domain Language« stellt die oberste Stufe der Generatorformen dar. Eine solche Sprache ist Turing-vollständig und speziell auf die Problemdomäne ausgerichtet.

### 3.1.4 Optimierung durch den Generator

Die Effektivität von Optimierungen steigt mit dem Abstraktionslevel, deshalb ist es ratsam diese vom Generator durchführen zu lassen. Im Gegensatz zum Compiler, der viele dieser Optimierungen auch selbst durchführt, besitzt der Generator domänenspezifisches »Wissen« und kann teilweise ohne Abhängigkeiten der Zielsprache optimieren (*domain-specific optimization*).

Czarnecki und Eisenecker beschreiben in [CE00] Optimierungen die vom Generator durchgeführt werden können, einen um *Parallelisierung* erweiterten Auszug bietet die nachfolgende Liste:

[4] Tier, zu deutsch »Stufe«

[5] Der im Laufe dieser Arbeit entwickelte Generator entspricht diesem Schema.

***Inlining:***
Ein Symbol durch seine Definition ersetzen oder anstelle eines Funktionsaufrufes, die Deklaration der Funktion selbst einfügen.

***Constant folding:***
Auswertung von Ausdrücken deren Operanden während der *compile time* bekannt sind.

***Data caching:***
Anstatt mehrfach denselben Ausdruck auszuwerten, das Ergebnis einmal berechnen und darauf an anderer Stelle referenzieren.

***Loop fusion:***
Zusammenführen von Schleifen, die über den gleichen Bereich iterieren und deren Schleifenkörper unabhängig voneinander ist.

***Loop unrolling:***
Die Schleife durch $n$-mal deren Inhalt ersetzen, wobei $n$ die Anzahl der Iterationen ist.

***Code motion:***
Invariante[6] Codebereiche aus dem Schleifenkörper herausnehmen.

***Dead-code elimination:***
Entfernen von ungenutzten Variablen und unerreichbaren Codebereichen.

***Partial evaluation/Specialisation:***
*Partial evaluation* oder auch *Specialisation* meinen das Erzeugen von spezialisierten Funktionen. Diese implementieren statische Eingaben mit dem Ziel einer kleineren Funktionssignatur.

***Parallelization:***
Analyse der Datenabhängigkeit durch den Generator und eventuelles parallelisieren voneinander unabhängiger Bereiche.

[6] unveränderliche

## 3.2 Datenmodell

Das Datenmodell enthält die Informationen der Spezifikation und dient als Eingabe für den Generator, es ist somit die *Basis der Codegenerierung.* Westerinen u. a. definieren den Begriff in [Wes+01] folgenderweise[7]:

> **Definition 5** (Datenmodell)**.** *Ein Datenmodell ist im Grunde die Darstellung eines Informationsmodells unter Berücksichtigung einer Menge von Mechanismen für die Darstellung, Organisierung, Speicherung und Bearbeitung von Daten. Das Modell besteht aus einer Sammlung von …*
>
> - *Datenstrukturen, wie Listen, Tabellen, Relationen etc.*
> - *Operationen die auf die Strukturen angewendet werden können, wie Abfrage, Aktualisierung, …*
> - *Integritätsbedingungen die gültige Zustände (Menge von Werten) odder Zustandsänderungen (Operationen auf Werten) definieren.*

Bei dieser Definition wird der Begriff *Informationsmodell* genutzt, er beschreibt die Informationen die im Datenmodell abgebildet werden sollen ohne Berücksichtigung softwaretechnischer Aspekte. Das Informationsmodell stellt somit die »natürlichen Daten« dar.

Bei einem Codegenerator entspricht das Datenmodell der internen Darstellung der Spezifikation. Neben den direkt in der Spezifikation enthaltenen Informationen kann der Generator im Analyseschritt (siehe Abschnitt 3.1.1) bspw. Datenabhängigkeiten erkennen und diese zur Optimierung nutzen oder das interne Datenmodell damit anreichern. Das *erkennen von Semantik* im Eingabemodell ist aber nicht auf Datenabhängigkeiten beschränkt sondern kann auf beliebige Beziehungen ausgeweitet werden.

Wie man in Abbildung 2.2 erkennen kann, entspricht die Struktur einer WADL-Datei einem Baum. Aus diesem Grund eignet sich eine Baumstruktur für das Datenmodell des Generators besonders gut. Um die Zielsprache (siehe Abschnitt 3.3) im internen Datenmodell abbilden zu können, wird ein Abstract Syntax Tree als Datenstruktur gewählt.

[7] eigene Übersetzung

### 3.2.1 Abstract Syntax Tree (AST)

Eine anschauliche Definition enthält [Aho+06, S. 69] (eigene Übersetzung) :

> **Definition 6** (Abstract Syntax Tree – Aho u. a.)**.** *Ein Abstrakter Syntaxbaum ist die Darstellung eines Ausdrucks, wo jeder Knoten einen* Operator *und dessen Kindknoten die* Operanden *repräsentieren. Im Allgemeinen kann für jedes Programmierkonstrukt ein Operator erzeugt werden, dessen semantisch bedeutsamen Komponenten dann als Operanden gehandhabt werden.*

Er ist das Endprodukt eines Parsingschrittes des Quelltextes, im Gegensatz zum *konkreten Syntaxbaum* (auch *Parse Tree*) enthält der *AST* keine Formatierungsspezifische Syntax (bspw. Klammern).

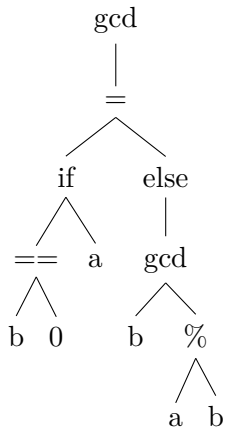


**Abbildung 3.1:** Beispiel AST für den rekursiven euklidischen Algorithmus

## 3.3 Objektorientierte Sprachen

Ziel des Generators ist die Erzeugung von Code in einer Objektorientierten[8] Sprache. Aus diesem Grund werden die elementaren Konzepte solcher Sprachen in diesem Abschnitt näher erläutert, sowie die Besonderheiten der Generatorzielsprache (PHP) beschrieben.

[8] nachfolgend nur noch OO

Im Gegensatz zu *Prozeduralen Sprachen*[9], in denen ein Programm eine Liste von Funktionen ist, wird dieses im OO-Programmierparadigma aus der Interaktion von *Objekten* gebildet.

### 3.3.1 Elemente

Die Beschreibung der Elemente einer OO-Sprache basiert auf [PK12].

***Objekte:***

Elementare logische Einheit, kapselt Variablen und *Methoden* (Kapselung) und schützt private Daten vor äußerem Zugriff (Data-Hiding). Der Zugriff auf die Elemente des Objektes wird über *Access Modifier* geregelt. Bilden einen Namensraum und schützen davor das Änderungen an privaten Daten sich auf andere Objekte auswirken.

***Klassen:***

Klassen beschreiben die Variablen und Methoden für Objekte die aus ihnen erzeugt werden. Ein Objekt ist eine *Instanz* einer Klasse. Eine Klasse kann ein aus ihr erzeugtes Objekt mit bestimmten Vorgabewerten initialisieren. Objekte einer Klasse werden erzeugt oder auch instanziiert durch den Aufruf der Konstruktormethode der Klasse. Die meisten OO-Sprachen bieten Möglichkeiten der Vererbung, d.h. das Klassen gewisse Eigenschaften und Methoden von einer Klasse »erben« können. Weiterhin können Klassen auch abstrakt sein, also die in ihnen enthaltenen Klassen und Methoden sind nur Bezeichner aber besitzen keine Definition. Diese müssen dann von den klassen definiert werden die diese Abstrakten Klassen *implementieren*.

***Methoden:***

Methoden sind die *Funktionen* des Objektes, sie beschreiben sein *Verhalten*.

***Felder:***

Felder enthalten die Daten des Objektes. Ihr Inhalt repräsentiert den *Zustand* des Objektes.

[9] Zu den Prozeduralen Sprachen zählt bspw. C und Pascal

***Access Modifier:***

Access Modifier regeln den Zugriff auf die Elemente eines Objektes, die gebräuchlichsten sind hierbei `public, private` und `protected`, durch deren Verwendung wird die Kapselung von Daten erreicht. Welche Arten der Zugriffskontrolle letztendlich vorhanden sind ist abhängig von der verwendeten Programmiersprache.

***Namensräume:***

Namensräume erlauben die Verwendung von gleichen Bezeichnern in unterschiedlichen Namensräumen. Wie im Punkt Objekte erwähnt, bilden diese bspw. einen eigenen Namensraum. Der Zugriff auf ein Element eines Objektes erfolgt über seinen Namensraum, will man auf das Element `bar` des Objektes `foo` zugreifen, geschieht dies z.B. in PHP folgendermaßen: `foo->bar`.

### 3.3.2 PHP

PHP ist eine General Purpose Language die aber vorwiegend auf die Entwicklung von serverseitigen Webapplikationen ausgerichtet ist. PHP Skripte können in HTML-Dateien eingebettet werden, welche der Server bei einer Client-Anfrage verarbeitet, die PHP Elemente durch deren Ausgabe ersetzt und dem Client zurücksendet. Die Sprache gehört somit zu den *Server-Side Scripting Languages*. Die Verwendung ist aber nicht auf diesen Bereich beschränkt, denn PHP Anweisungen müssen nicht in HTML eingebettet sein sondern können auch unabhängig davon, als eigene Datei, ausgeführt werden. Im Gegensatz zu *Java* ist PHP nicht statisch typisiert und muss zur Ausführung auch nicht kompiliert werden. PHP ist *dynamisch typisiert* und wird von einem Interpreter — dem namensgebenden *Hypertext Preprocessor* — ausgeführt. Es werden mehrere Programmierparadigmen unterstützt, seit Version 5.0 neben Imperativer- auch Objektorientierte Programmierung. Version 5.3 fügte die Unterstützung von Closures hinzu.

```
1  <?php ❶
2     require_once('Dto.php'); ❷
3     require_once('OperationDTO.php'); ❷
```

```
4
5     class BatchDTO
6     {
7        private $operations = array(); // operationDTO
8
9        function __construct(operationDTO ❸ $operations)
10       {
11          $this->operations = $operations;
12       }
13
14       public function getOperations()
15       {
16          return $operations = $this->operations;
17       }
18
19       public function setOperations(operationDTO $operations)
20       {
21          $this->operations = $operations;
22       }
23
24       public function toJSON()
25       {
26          $json = json_decode(/* BatchDTO */ $this);
27          return $json;
28       }
29
30       ...
31
32       public static ❹ function fromXML(SimpleXMLElement $xml)
33       {
34          $operations = OperationDTO::fromXML(/* SimpleXMLElement */ $xml
                ->operations);
35          $BatchDTO =  new BatchDTO(/* operationDTO */ $operations);
36          return $BatchDTO;
37       }
38
39       ...
40    }
41
42 ?> ❶
```

**Listing 3.1:** Durch den Generator erzeugte BatchDTO Datenklasse der Spreadshirt-API als Beispiel für eine PHP-Datei

❶ Start- und Endtags eines PHPFiles, wobei letzteres optional ist. Deren Funktion ist die Abgrenzung vom umliegenden Markup, bspw. wenn der PHP-Code in eine HTML Datei eingebettet ist.

❷ PHP unterstützt das importieren von Quellcodefiles anhand verschiedener Befehle, in diesem Fall `require_once`.

❸ Nur in der Argumentliste einer Methodendefinition sind Typangaben erlaubt, solange der Typ kein primitiver ist, d.h. den sprachinternen primitiven Datentypen wie bspw. String oder Integer entspricht.

❹ Statische Methoden, können ohne Instanz der umgebenden Klasse aufgerufen werden, sind ebenfalls ünterstützt.

# 4 Codegenerierung für die Spreadshirt-API

## 4.1 Konkretes Datenmodell

Thema dieses Abschnittes ist die Struktur und die verwendeten Entwurfsmuster in den Datenmodellen, die die Ein- bzw. Ausgabedaten des Generators beinhalten.

### 4.1.1 REST-Modell

Zuerst muss die abstrakte Beschreibung der Spreadshirt-API von der XML-Form, bestehend aus einem *WADL* (Abschnitt 2.5) und einem oder mehreren Schemabeschreibungen (Abschnitt 2.2), in ein für den Generator verarbeitbares Format überführt werden.

Die durch die WADL-Datei beschriebene Baumstruktur muss in ein Datenmodell bestehend aus Klassen und Objekten transformiert werden. Um effektiv mit der XML Darstellung arbeiten zu können wird diese zuerst mit einem Parser in ein DOM-Modell überführt (Abschnitt 6.1) welches im Arbeitsspeicher gehalten wird und damit einen schnellen Zugriff für nachfolgende Operationen darauf erlaubt. In einem nächsten Schritt wird das DOM-Modell, welches noch viele XML-Dokument spezifische Informationen enthält, auf die wesentlichen API beschreibenden Merkmale reduziert. Im Gegensatz zu der in Abbildung 2.2 veranschaulichten Webanwendungsbeschreibung werden Referenzen durch deren Definition im Modell ersetzt.

Wurzelelement des Modells ist **Application**, über dieses Element kann auf alle *Ressourcen* zugegriffen werden. Es enthält außerdem den Basisbezeichner der API bspw. `http://api.spreadshirt.net/api/v1/`.

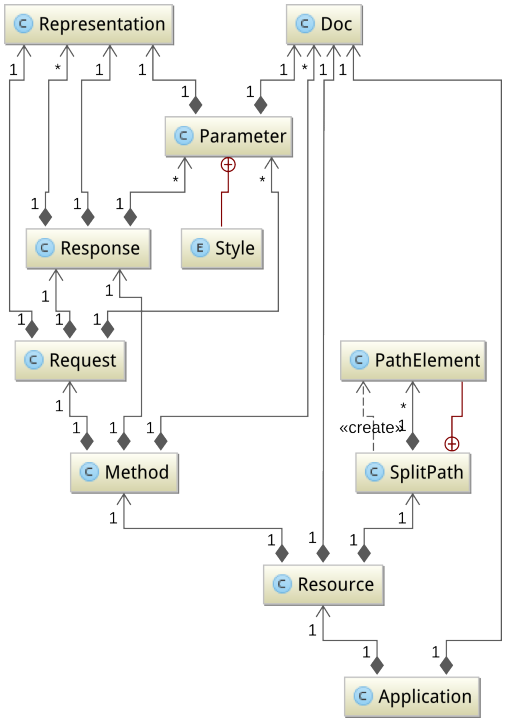


**Abbildung 4.1:** UML-Diagramm des REST-Modells

Eine **Ressource** enthält eine Menge von *Methoden*, sowie einen Ressourcenbezeichner, diese sind relativ zum Basisbezeichner des Wurzelelements. Die Ressourcenbezeichner können *Template-Parameter* enthalten, diese werden bei einer Anfrage durch einen konkreten Wert ersetzt. Beispielweise enthält der Bezeichner für die Ressource eines bestimmten Users den Template-Parameter {userid}, vollständiger Ressourcenbezeichner `users/{userid}`. Ressourcenbezeichner werden durch **SplitPath** repräsentiert.

Jede Methode enthält ein **Request** und ein *Response* Element, welche die nötigen Informationen für eine Anfrage auf bzw. den Aufbau der Antwort von, einer Ressource enthalten.

**Request** enthält eine Liste von Query-Parametern sowie ein *Representation* und *Response* Element.

**Parameter** enthält Angaben zum *Style*, Typ, Vorgabewert und ob dessen Angabe »required«, also notwendig ist. Die Angabe des Typs ist eine Referenz auf einen Typ aus einer XML-Schemabeschreibung. Der *Style* gibt an wie der Parameter übermittelt wird, als Teil der Query `…?mediaType=xml`, *Key-Value Pair* des HTTP-Header oder als *Template-Parameter* des Ressourcenbezeichners.

Das **Response** Element enthält eine Menge von *Representations*, sowie eine Liste von Parametern. Die *Representations* enthalten die Beschreibung der Daten die bei einer erfolgreichen Anfrage an die Ressource zurückgesendet werden, sowie die der Fehlermeldung die der Client anderenfalls erhält. Zwischen einer Fehlermeldung und einer Antwort auf eine erfolgreiche Anfrage kann anhand des Wertes des HTTP-Statuscodes unterschieden werden. Erfolgreiche Antworten liefern meist einen Statuscode 200 **OK** oder 201 **Created** zurück. Die Response Parameter geben meist Einträge im HTTP-Header an, welche für den Client nützliche Informationen enthalten. Legt der Client z.B. via POST auf der Ressource `sessions` eine neue API-Session an, enthält das Feld `Location` des HTTP-Headers der Antwort, die URL auf die Ressource der angelegten Session.

**Representation** dient zur Beschreibung der Daten welche entweder zur API gesendet oder von dieser empfangen werden, sie besteht aus einer Angabe des *media-type*, des HTTP-Statuscodes und eine Referenz auf die Definition des Datentyps. Die *Representation* eines Request einer PUT- oder POST-Methode charakterisiert zum Beispiel den Aufbau der Daten welche der Ressource übermittelt werden, üblicherweise im HTTP-Body. Die Charakterisierung erfolgt dabei in Form einer Referenz auf einen Typ aus einer Schemabeschreibung sowie der Angabe des *media-type*. Die *Representation* der PUT-Methode auf Ressource `users/{userId}/designs/{designId}` enthält den media-type `application/xml` und eine Referenz auf den Typ `sns:design`.

Referenzen auf Typdeklaration aus einer Schemabeschreibung werden nachfolgend im Modell durch die konkrete Deklaration des Typs aus der XML-Schemabeschreibung ersetzt, siehe Abschnitt 4.1.3. **Doc** Element enthalten nur einen Titel und eine Kurzbeschreibung des zugehörigen Elements. Aus dieser *Dokumentation* wird durch den Generator in ein Kommentar erzeugt.

### 4.1.2 Schema-Modell

### 4.1.3 Applikationsmodell

### 4.1.4 Zielsprachenmodell

### 4.1.5 PHP-Zielsprachenmodell

### 4.1.6 Language Factory

## 4.2 Generatorsystem

Ablauf des Generators und Diagramm

### 4.2.1 Datenklassen

Zielsprachenabhängige Repräsentation

### 4.2.2 Ressourcenklassen

Nur Zielsprachenabhängig

### 4.2.3 Serialisierer

Transportorientierte Repräsentation

### 4.2.4 Deserialisierer

### 4.2.5 Ausgabemodul

## 4.3 API-Design

Im Folgenden Abschnitt werden Kriterien und Designentscheidungen für die zu erzeugende Bibliothek behandelt.

### 4.3.1 Kriterien

### 4.3.2 Sessions

### 4.3.3 Parameterobjekte

### 4.3.4 Entwurfsmuster

#### 4.3.4.1 Expression Builder

#### 4.3.4.2 Fluent Interface

# 5 Evaluation

## 5.1 Nutzbarkeit

## 5.2 Leistungsbewertung

## 5.3 Codemetriken

# 6 Implementierung

In diesem Kapitel werden die verwendeten Softwarebibliotheken kurz erläutert und die Gründe für deren Auswahl dargelegt. Außerdem wird die konkrete Implementierung des Datenmodells hier vorgestellt. Der Generator ist in *Java* implementiert worden.

## 6.1 XML-Parser

Um mit der abstrakten Beschreibung der Spreadshirt-API arbeiten zu können, muss diese zuerst in das interne Datenmodell überführt werden. Diese liegt in XML-basierter Form vor, welche in Kapitel 2 näher beschrieben wurde. Folglich wird ein XML-Parser für die Verarbeitung der Beschreibungsformate benötigt.

Die *Java API for XML Processing* kurz *JAXP* abstrahiert die Parserschnittstelle von der eigentlichen Implementierung. JAXP ist dabei keine einzelne API sondern es beschreibt Schnittstellen für folgende vier XML-Parser Modelle:

***DOM:***
: *Document Object Model*-Parser überführen das XML-Dokument in ein baumartiges Objektmodell, welches vollständig im Arbeitsspeicher liegt.

***SAX:***
: *Simple API for XML* basierte, sogenannte Push-Parser verarbeiten das XML-Dokument seriell und eventbasiert. Ein Event ist hierbei bspw. ein öffnendes oder schließendes XML-Element.

***StAX:***
: *Streaming API for XML* basierte, sogenannte Pull-Parser arbeiten ebenso wie bei *SAX* seriell und eventbasiert, können aber im Gegensatz dazu die Erzeugung von Events selber steuern.

***TrAX:***

*Transformation API for XML* bietet eine Schnittstelle mit der sich XML-Dokumente durch *Extensible Stylesheet Language Transformations (XSLT)* in Java transformieren lassen.

Tabelle 6.1 enthält eine Übersicht zu den Parsing-Konzepten, ausgenommen *TrAX* da diese API vorwiegend für die Modifikation von XML-Dateien gedacht ist.

Bei dem zu entwickelnden Codegenerator sind der Speicherverbrauch und die verwendete CPU-Zeit kein Teil der *nichtfunktionalen Anforderungen*, somit fiel die Entscheidung auf einen DOM-Parser. Dieser lässt sich durch das komplett im Speicher gehaltene Objektmodell mit geringem Aufwand verwenden. Durch JAXP ist die Implementierung transparent und es wird die im *JDK* enthaltene Standart DOM-Parser Implementierung verwendet.

| | **DOM** | **SAX** | **StAX** |
|---|---|---|---|
| **API-Typ** | In-Memory Tree | push-streaming | pull-streaming |
| **Speicherverbrauch** | hoch | gering | < DOM |
| **Prozessorlast** | hoch | gering | gering |
| **Elementzugriff** | beliebig | seriell | seriell |
| **Nutzerfreundlichkeit** | niedrig | hoch | mittel |
| **XML schreiben** | ja | nein | ja |

**Tabelle 6.1:** Übersicht über die verschiedenen XML-Parsing Konzepte in JAXP

## 6.2 Generator Schnittstelle

# 7 Zusammenfassung

## 7.1 Fazit

## 7.2 Ausblick

# Glossar

***Abstract Syntax Tree:***
Ein *Abstrakter Syntaxbaum* ist die Baumdarstellung einer abstrakten Syntaktischen Struktur von Quellcode einer Programmiersprache. Jeder Knoten des Baumes kennzeichnet ein Konstrukt des Quellcodes. Der *AST* stellt für gewöhnlich nicht alle Details des Quelltextes dar, bspw. formatierende Element wie etwa Klammern werden häufig weggelassen. . 30, A, B

***API:***
*Application Programming Interface* (deutsch: »Schnittstelle zur Anwendungsprogrammierung«) spezifiziert wie Softwarekomponenten über diese Schnittstelle miteinander interagieren können . 3, A

***Closure:***
Eine *Closure* ist eine Funktion welche die besondere Eigenschaft besitzt auf Variablen aus ihrem Entstehungskontext zugreifen zu können. Die Funktion wird meist in einer Variablen gespeichert um den Zugriff darauf zu sichern. . 33, A

***DSL:***
*Domain Specific Language* (deutsch: »Domänenspezifische Sprache«) ist eine Programmiersprache die nur auf eine bestimmte Domäne oder auch Problembereich optimiert ist. . 3, A, B

***DTD:***
*Document Type Definition*, manchmal auch *Data Type Definition*, ist eine Menge von Angaben die einen Dokumenttyp beschreiben. Es werden konkret Element- und Attributtypen, Entitäten und deren Struktur beschrieben. Die bekanntesten Schemasprachen für XML-Dokumente sind XSD und RelaxNG. A, *siehe* XSD

***General Purpose Language:***
Eine *General Purpose Language* bezeichnet eine Programmiersprache welche für den Einsatz in den verschiedensten Anwendungsbereichen verwen-

det kann, im Gegensatz zur einer DSL, welche nur auf einen speziellen Bereich beschränkt ist. . 33, A, B

***JSON:***
*JavaScript Object Notation* ist ein Mensch- und Maschinenlesbares Format zu Codierung und Austausch von Daten. Bietet im Gegensatz zu XML keine Erweiterbarkeit und Unterstützung für Namesräume, ist aber kompakter und einfacher zu parsen. 6, 9, 21, A, *siehe* XML

***MDA:***
*Model Driven Architecture* ist ein modell-getriebener Softwareentwicklungsansatz. Das zu modellierende System wird hierbei durch ein plattformunabhängiges Modell beschrieben mittels einer DSL beschrieben. Dieses Modell wird dann durch einen Generator in ein plattformspezifisches Modell, meist in einer General Purpose Language übersetzt . A

***MDE:***
*Model Driven Engineering.* A, *siehe* MDSD

***MDSD:***
*Model Driven Software Development*, auch *Model Driven Engineering* ist eine Softwareentwicklungsmethode welche ihren Fokus auf das erzeugen und nutzen von Domänen-Modellen, anstelle der algorithmischen Konzepte, legt . A

***Metaprogramming:***
beschreibt das erstellen von Programmen welche sich selbst, oder andere Programme, modifizieren oder die einen Teil des Kompilierungsschrittes übernehmen (bspw. der C-Präprozessor) . A

***MIME:***
*Multipurpose Internet Mail Extensions* dienen zu Deklaration von Inhalten (Typ des Inhalts) in verschiedenen Internetprotokollen. . 22, A

***Parsing:***
Parsing oder auch *Syntaxanalyse* erzeugt aus einer Zeichenkette einen Abstract Syntax Tree aufgrund der Regeln einer Grammatik. Der Aufbau der Zeichenkette wird also nach den Regeln der Grammatik analysiert und die einzelnen Elemente in einen Abstract Syntax Tree überführt. . A

***Polyglot:***
*mehrsprachig* . A

***RelaxNG:***
*Regular Language Description for XML New Generation* ist ebenso wie *XSD* eine Schemabeschreibungssprache, bietet aber zwei Syntaxformen, eine XML basierte und eine kompaktere eigene Syntax. 6, A, *siehe* XSD

***REST:***
*Representational State Transfer* (deutsch: »Gegenständlicher Zustandstransfer«) ist ein Softwarearchitekturstil für Webanwendungen, welcher von Roy Fielding in seiner Dissertation [1] beschrieben wurde. Die Daten liegen dabei in eindeutig addressierbaren *resources* vor. Die Interaktion basiert auf dem Austausch von *representations* – also ein Dokument was den aktuellen oder gewünschten Zustand einer resource beschreibt. Beispiel-URL für das Item *84* aus dem Warenkorb *42*:
`http://api.spreadshirt.net/api/v1/baskets/84/item/42` . 6, 16, 20, A

***RESTful:***
Als *RESTful* bezeichnet man einen Webservice der den Prinzipien von REST entspricht. 16, A, *siehe* REST

***Template-Engine:***
Eine *Template-Engine* ersetzt markierte Bereiche in einer Template-Datei (i. Allg. Textdateien) nach vorgegebenen Regeln . A

***URI:***
*Unified Resource Identifier* ist ein Folge von Zeichen, die einen Name oder eine Web-Ressource identifiziert.. 6, 19, 20, 23, A

***URL:***
*Unified Resource Locator* sind eine Untermenge der *URIs*. Der Unterschied besteht in der expliziten Angabe des Zugrissmechanismus und des Ortes (»Location«) durch *URLs*, bspw. `http` oder `ftp`. A, *siehe* URI

***WADL:***
*Web Application Description Language* ist eine maschinenlesbare Beschreibung einer HTTP-basierten Webanwendung. 6, 21, 26, A, *siehe* XML

[1] `http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`

***XML:***

*Extensible Markup Language* (deutsch: »erweiterbare Auszeichnungssprache«) ist ein Mensch- und Maschinenlesbares Format für Codierung und Austausch von Daten, spezifiziert vom W3C [2] . 6, 8, 9, A

***XSD:***

*XML Schema Description*, auch nur *XML Schema*, ist eine Schemabeschreibungssprache und enthält Regeln für den Aufbau und zum Validieren einer XML-Datei. Die Beschreibung ist selbst wieder eine gültige XML-Datei. 6, 10, A, *siehe* XML

[2] `http://www.w3.org/TR/REC-xml`

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# Definitionsverzeichnis

# Literaturverzeichnis


[Aho+06] Alfred V. Aho u. a. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[CE00] K. Czarnecki und U. Eisenecker. *Generative programming: methods, tools, and applications*. Addison Wesley, 2000. ISBN: 9780201309775. URL: http://books.google.de/books?id=cCZXYQ6Pau4C.

[Cro06] Douglas Crockford. *RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)*. Techn. Ber. Juli 2006. URL: http://tools.ietf.org/html/rfc4627.

[Fie00] Roy Thomas Fielding. »Architectural styles and the design of network-based software architectures«. AAI9980887. Diss. 2000. ISBN: 0-599-87118-0.

[Fie+99] R. Fielding u. a. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[GZ13] F. Galiegue und K. Zyp. *JSON Schema: core definitions and terminology*. 31. Jan. 2013. URL: http://tools.ietf.org/html/draft-zyp-json-schema-04.

[Had06] Sun Microsystems Inc. Hadley Marc J. *Web Application Description Language (WADL)*. abgerufen am 21.06.2013. 9. Nov. 2006. URL: https://wadl.java.net/wadl20061109.pdf.

[Had09a] Marc Hadley. *Web Application Description Language – Changes since November 2006 Publication*. 31. Aug. 2009. URL: http://www.w3.org/Submission/wadl.

[Had09b] Marc Hadley. *Web Application Description Language – Changes since November 2006 Publication*. 31. Aug. 2009. URL: http://www.w3.org/Submission/wadl/#x3-41000D.1.

[Her03] J. Herrington. *Code Generation in Action*. In Action Series. Manning, 2003. ISBN: 9781930110977. URL: http://books.google.de/books?id=VHVC8WnSgbYC.

[Mur+05] Makoto Murata u. a. »Taxonomy of XML schema languages using formal language theory«. In: *ACM Trans. Internet Technol.* 5.4 (Nov. 2005), S. 660–704. ISSN: 1533-5399. DOI: 10.1145/1111627.1111631. URL: http://doi.acm.org/10.1145/1111627.1111631.

[Ött13] Tim Ötting. *soundslikecotton.com*. 20. Juni 2013. URL: http://www.soundslikecotton.com/.

[Pas13] Kathrin Passig. *zufallsshirt.de*. 20. Juli 2013. URL: http://zufallsshirt.de/.

[PK12] Franz Puntigam und Andreas Krall. *Objektorientierte Programmiertechniken*. 2012.

[Til09] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. Heidelberg: dpunkt, 2009. ISBN: 978-3-89864-583-6.

[Vli04] Eric van der Vlist. *RELAX NG - a simpler schema language for XML*. O'Reilly, 2004, S. I–XVIII, 1–486. ISBN: 978-0-596-00421-7.

[W3C08] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 26. Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/ (besucht am 25. 06. 2013).

[W3C12] W3C. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 5. Apr. 2012. URL: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (besucht am 30. 06. 2013).

[Wes+01] A. Westerinen u. a. *RFC 3198, Terminology for Policy-Based Management*. Nov. 2001. URL: http://tools.ietf.org/html/rfc3198.

[Wik13] Wikipedia. *XML — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-June-2013]. 2013. URL: \url{http://en.wikipedia.org/w/index.php?title=XML&oldid=561587115} (besucht am 26. 06. 2013).

# BibTeX Eintrag

```
@phdthesis{AndreasLinz2013,
        type = {Bachelorthesis}
        author = {Linz, Andreas},
        year = {2013},
        month = {9},
        day = {5},
        timestamp = {201395},
        title = {Generierung und Design einer Client-Bibliothek für einen
            RESTful Web Service am Beispiel der Spreadshirt-API},
        school = {HTWK-Leipzig},
        pdf = {http://www.klingt.net/bachelor/thesis/thesis.pdf}
        %ToDo: PUT IN THE URL
}
```