



HTWK Leipzig

Fakultät für Informatik, Mathematik & Naturwissenschaften

Bachelorarbeit

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, 1. Oktober 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*
Bachelorarbeit, HTWK Leipzig, 1. Oktober 2013

made with X_YTeX and BiBTeX.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich diese Bachelorarbeit 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, 1. Oktober 2013

Danksagungen

Ein besonderer Dank gilt Dr. rer. nat. Johannes Waldmann für die Anregungen und Ratschläge sowie das gezeigte Interesse am Thema.

Ebenso möchte ich mich bei Jens Hadlich bedanken, der für Fragen jederzeit ein offenes Ohr hatte und mir half den »roten Faden« bei der Strukturierung der Arbeit nicht zu verlieren. Außerdem gilt Spreadshirt ein großer Dank, da sie mir eine wunderbare Arbeitsumgebung sowie nette und hilfreiche Kollegen zur Verfügung gestellt haben.

Ohne Elisa Jentsch würde sicherlich der ein oder andere Rechtschreibfehler den Lesefluss trüben, deshalb vielen Dank für das Korrekturlesen.

Abstract

German

Die vorliegende Arbeit beschreibt den Entwurf eines Codegenerators mit austauschbarer Zielsprache, der aus der abstrakten Beschreibung der Spreadshirt-API eine Client-Bibliothek generieren soll. Die Implementierung des Codegenerators erfolgt in Java, die Zielsprache der Bibliothek ist PHP.

Es werden die Grundlagen von WebServices, Codegeneratoren und Dokumentbeschreibungssprachen erläutert und darauf aufbauend Datenmodelle erstellt welche die Beschreibung der Spreadshirt-API für den Generator enthalten. Außerdem wird das erstellte Sprachenmodell betrachtet, welches die Konstrukte der zu erzeugenden Zielsprache kapselt. Aufbau und Ablauf eines Codegeneratorsystems werden ebenfalls beschrieben.

Die durch den Codegenerator erstellte Client-Bibliothek wird anhand eines Anwendungsbeispiels evaluiert.

English

The present thesis describes the design of a codegenerator with exchangeable target-language, that generates a client-library from the abstract description of the Spreadshirt-API. The implementation of the codegenerator is made in Java, the target-language of the client-library is PHP.

The basics of web services, codegenerators and document-description-languages will be explained and based on that datamodels will be created, that contains the description of Spreadshirt-API, for the generator. Furthermore the created language-model will be examined, which encapsulates the constructs of the

target-language that will be generated. Structure and process of a codegenerator system will be described, as well.

The client-library that was created by the codegenerator will be evaluated on the basis of a usage example.

Keywords

Codegeneration, RESTful Web Service, Modeling, Client-Library, Spreadshirt-API, Polyglot

Lizenz

Die vorliegende Bachelorarbeit **Generierung und Design einer Client-Bibliothek für einen RESTful Web Service am Beispiel der Spreadshirt-API** ist unter Creative Commons CC-BY-SA ¹ lizenziert.



¹<http://creativecommons.org/licenses/by-sa/3.0/deed.de>

Inhaltsverzeichnis

1. Einführung	2
1.1. Anforderungen an die Client-Bibliothek	4
1.2. Typographische Konventionen dieser Bachelorarbeit	4
2. Web Services	6
2.1. Adressierung	6
2.2. HTTP	8
2.2.1. Methoden	8
2.2.2. Header	9
2.2.3. Body	10
2.3. Dokumentbeschreibungsformate	11
2.3.1. XML	11
2.3.2. JSON	12
2.4. XML Schemabeschreibungssprachen (XML Schema)	13
2.4.1. XML Schema Description (XSD)	14
2.5. RESTful Web Service	17
2.5.1. Elemente von REST	17
2.5.2. REST-Prinzipien	21
2.6. WADL	23
3. Codegenerierung	28
3.1. Codegeneratoren	28
3.1.1. Aufgaben eines Generators	29
3.1.2. Vorteile für den Entwickler	29
3.1.3. Generatorformen	30
3.1.4. Optimierung durch den Generator	31
3.2. Datenmodell	32
3.2.1. Abstract Syntax Tree (AST)	32
3.3. Objektorientierte Sprachen	33
3.3.1. Elemente	34
3.3.2. Typsystem	36
3.3.3. PHP	36

4. Generatorsystem für die Spreadshirt-API	39
4.1. Konkrete Datenmodelle	39
4.1.1. REST-Modell	39
4.1.2. Schema-Modell	42
4.1.3. Applikationsmodell	44
4.1.4. Sprachenmodell	44
4.2. Codegenerator	48
4.2.1. Language Factory	48
4.2.2. Ausgabemodul	49
4.2.3. Generatorablauf	49
4.3. Client-Bibliothek	49
4.3.1. Datenklassen	51
4.3.2. Ressourcenklassen	53
4.3.3. De-/Serialisierer	55
4.3.4. Statische Klassen	55
5. Evaluierung	57
5.1. PHP-Zielsprachenmodell	57
5.1.1. Implementierung einer Zielsprache	57
5.1.2. Codestyle/Lesbarkeit	59
5.2. Typsicherheit	60
5.3. Generierung von Tests für die erzeugte Bibliothek	60
5.4. Metriken	60
5.5. Leistungsbewertung	62
5.6. Nutzbarkeit	63
6. Schlussbetrachtung	65
6.1. Ausblick	65
6.2. Fazit	66
A. Implementierung	A
A.1. XML-Parser	A
Glossar	C
Abbildungsverzeichnis	F
Tabellenverzeichnis	G
Listings	H

Definitionsverzeichnis	J
Literaturverzeichnis	K
BIB_T_EX Eintrag	N

1. Einführung

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

[BD87, S. 424]
Box und 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 Be-

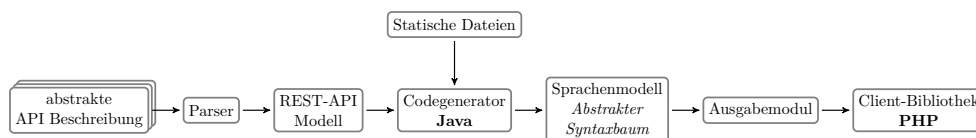


Abbildung 1.1.: Aufbau des Generatorsystems



Abbildung 1.2.: Spreadshirt Logo

ziehung sowie Kommunikation der Kunden untereinander im Vordergrund stehen. Spreadshirt hat Standorte in Europa und Nordamerika, der Hauptsitz ist in Leipzig.

Dem Nutzer wird eine Online-Plattform geboten um Kleidungsstücke selber zu gestalten oder zu kaufen, aber auch um eigene Designs, als Motiv oder in Form von Produkten, zum Verkauf anzubieten. Zusätzlich wird jedem Nutzer ermöglicht einen eigenen Shop auf der Plattform zu eröffnen und diesen auf der eigenen Internetseite einzubinden. Derzeit gibt es rund 400.000 Spreadshirt-Shops mit ca. 33 Millionen Produkten. Für die Spreadshirt-API können Kunden eigene Anwendungen schreiben, beispielsweise *www.zufallsshirt.de* [Pas13] oder *www.soundlikecotton.com* [Ött13]. Spreadshirt bedient neben dem Endkunden auch das Großkundengeschäft als Anbieter von Druckleistungen.

Die *sprd.net AG*, zu der auch der Leipziger Hauptsitz gehört, beschäftigt derzeit 450 Mitarbeiter, davon 50 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* und erhöhter Effizienz 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, da beispielsweise nicht mit Überraschungen bei den verwendeten Bezeichnern zu rechnen ist.

Zusätzlich zu dem bereits genannten allgemeinen Nutzen einer Codegenerierungslösung, entstehen für Spreadshirt noch die folgenden Vorteile:

- Vereinheitlichung bestehender Implementierungen in Form der generierten Bibliothek
- Kapselung der Authentifizierung durch Integration in Client-Bibliothek (Abschnitt 4.3.4.2)
- Erleichterung der [API](#)-Nutzung für externe Entwickler

1.1. Anforderungen an die Client-Bibliothek

- Austauschbarkeit der Zielsprache
- Einfache Bedienbarkeit der Bibliothek
- gute Lesbarkeit des erzeugten Codes
- größtmögliche Typsicherheit
- hohe Testabdeckung der Bibliothek
- vollständige Generierung aller Methoden aus der [API](#)-Beschreibung

1.2. Typographische Konventionen dieser Bachelorarbeit

❶ – ❿:

Für Verweise auf Elemente in Listings werden diese Symbole verwendet

Klassennamen:

Namen von Klassen aus Listings oder Diagrammen werden im Text **fett** hervorgehoben



Schlüsselwörter:

Schlüsselwörter werden im Text *kursiv* hervorgehoben

Quelltext:

Zur Darstellung von Quelltext wird eine **Konstantschrift** verwendet



2. Web Services

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

*[HH73, Vorwort]
Hamming und Hamming (1973)*

In diesem Kapitel werden die Grundlagen zu HTTP, Dokumentbeschreibungssprachen, Beschreibungsformate für Webanwendungen und [REST](#) erläutert, welche für das Verständnis der Arbeit wichtig sind. Neben [XML](#) und [JSON](#) wird auch die Schemabeschreibungssprache [XSD](#) behandelt. Das Ende bildet die Einführung in [REST](#) und [WADL](#).

2.1. Adressierung

Für den Zugriff auf Dienste eines Web Services werden Adressen benötigt. Adressen im Internet sind meist nach einem bestimmten Schema aufgebaut, üblicherweise sind dies [URIs](#). Um Unklarheiten bei der Verwendung der Begriffe [URI](#), [URL](#) und [URN](#) im Verlauf der Arbeit zu vermeiden, werden Sie an dieser Stelle definiert.

Definition 1 ([URI](#)). Ein »Uniform Resource Identifier« ([URI](#)) ist eine kompakte Zeichenkette zur Identifizierung einer abstrakten oder physischen Ressource. ... Eine Ressource ist alles was identifizierbar ist, beispielsweise elektronische Dokumente, Bilder, Dienste und Sammlungen von Ressourcen. (eigene Übersetzung von [\[Ber+98\]](#)).



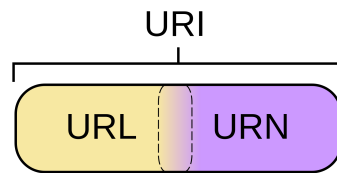


Abbildung 2.1.: Diagramm zur Veranschaulichung der Teilmengenbeziehung zwischen den Adressierungsarten, Quelle [Wik13b]

Definition 2 (URL). Der Begriff »Uniform Resource Locator« (URL) bezieht sich auf eine Teilmenge von URIs. URLs identifizieren Ressourcen über den Zugriffsmechanismus, anstelle des Namens oder anderer Attribute der Ressource. (eigene Übersetzung von [Ber+98]).

Definition 3 (URN). Eine Teilmenge der URIs, die sogenannten »Uniform Resource Names« (URNs), sind global eindeutige und beständige Bezeichner für Ressourcen. Sie müssen verfügbar bleiben auch wenn die bezeichnete Ressource nicht mehr erreichbar oder vorhanden ist. ... Der Unterschied zu einer URL besteht darin, dass ihr primärer Zweck in der dauerhaften Auszeichnung einer Ressource mit einem Bezeichner besteht. (eigene Übersetzung von [Ber+98]).

Beispiel URLs:

- `http://www.spreadshirt.net/`
- `mailto:admin@klingt.net`

Beispiel URNs:

- `urn:spreadshirt:product:2648`
- `urn:isbn:9780131392809`

2.2. HTTP

Definition 4 (HTTP). *Das Hypertext Transfer Protocol (HTTP) ist ein 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 des OSI-Modell 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.

2.2.1. Methoden

»RFC 2616, Hypertext Transfer Protocol – HTTP/1.1« ([Fie+99]) definiert einige HTTP-Methoden, wobei die am meisten verwendeten die folgenden sind:

- GET (Aufruf einer Webseite im Browser → GET-Request an Server)
- PUT
- POST (Übermittlung von Formulardaten eines Client an Server)
- DELETE
- OPTIONS

Alle bis auf POST und OPTIONS sind *idempotent* ([Fie+99] Kapitel 9), d.h. eine Komposition (Ausführung der Methoden hintereinander) der Methode führt zu demselben Ergebnis wie ein einzelner Aufruf.

Methodennamen sind üblicherweise Verben die die auszuführende Aktion beschreiben, deswegen werden HTTP-Methoden auch »verbs« genannt. Die Definition von eigenen Methoden ist möglich.



2.2.2. 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*, der Server welcher 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 beschriebenen *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; ❷
5 charset=UTF-8 ❸
6 X-Cache-Lookup: MISS from fish07:80
7 X-Server-Name: mem1
8 True-Client-IP: 88.79.226.66
9 Date: Tue, 20 Aug 2013 07:20:25 GMT
10 Content-Length: 1659
11 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 (hier [XML](#))
- ❸ Kodierung der Nachricht

Welche Einträge der Header einer [HTTP](#)-Nachricht letztendlich enthält, 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.

2.2.2.1. Authorization Request Header

Die [HTTP](#)-Spezifikation [[Fie+99](#), Abschnitt 14] sieht ein Feld für Autorisierungsinformationen im Header vor. Der Feld ist folgendermaßen aufgebaut: **Authorization: credentials**. Der Aufbau des Berechtigungsnachweises (engl. »credentials«) ist nicht näher spezifiziert und kann vom Web Service Betreiber selbst festgelegt werden. Die Spreadshirt-API definiert einen eigenen Autorisierungs-Header (Abschnitt [4.3.4.2](#)).

2.2.3. Body

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <locales
3     xmlns:xlink="http://www.w3.org/1999/xlink"
4     xmlns="http://api.spreadshirt.net"
5     xlink:href="http://api.spreadshirt.net/api/v1/locales"
6     offset="0" limit="50" count="16"
7     sortField="default" sortOrder="default">
8   <locale
9       xlink:href="http://api.spreadshirt.net/api/v1/locales/de_DE"
10      id="de_DE"/>
11   ...
12   <locale
13       xlink:href="http://api.spreadshirt.net/api/v1/locales/nL_BE"
14       id="nL_BE"/>
15 </locales>
```

Listing 2.3: [HTTP](#)-Body der Response aus der GET-Methode auf der Spreadshirt-API-Ressource <http://api.spreadshirt.net/api/v1/locales>



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.

2.3. Dokumentbeschreibungsformate

In diesem Abschnitt werden die beiden von der Spreadshirt-API verwendeten Dokumentbeschreibungsformate, XML und JSON, behandelt. Außerdem wird die Schemabeschreibungssprache XML Schema Description (XSD) eingeführt.

2.3.1. XML

Definition 5 (XML). Die »Extensible Markup Language«, kurz XML, ist eine Auszeichnungssprache (»Markup Language«), die eine Menge von Regeln beschreibt um Dokumente in einem mensch- und maschinenlesbaren Format zu kodieren [W3C08].

Obwohl das Design von XML auf Dokumente ausgerichtet ist, wird es häufig für die Darstellung von beliebigen Daten benutzt [Wik13c], 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 Wurzelement ❷. Die Elemente können Attribute enthalten und werden, wenn sie kein leeres Element sind ❸, von einem schließenden Tag in der gleichen Stufe abgeschlossen ❹. Nicht leere Zeichenketten als Kindelement sind ebenfalls erlaubt ❺.

Mit Hilfe von *Schemabeschreibungssprachen* (Abschnitt 2.4) 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.

2.3.2. JSON

Definition 6 (JSON). »JavaScript Object Notation«, kurz JSON, ist ein leichtgewichtiges, textbasiertes und sprachunabhängiges 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 (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 Ty-



primitiv	strukturiert
Zeichenketten	Objekte
Ganz-, Fließkommazahlen	Arrays
Booleans	
null	

Tabelle 2.1.: JSON Datentypen

pen 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
7   }, ❶
8   "resources": [ ❷
9     ...
10    {
11      "mediaType": "png",
12      "type": "preview",
13      "href": "http://image.spreadshirt.com/images/15513946"
14    },
15    ...
16  ], ❷
17  "created": "30.03.2013_12:37:54",
18  ...
19 }
```

Listing 2.5: Die gekürzte Antwort der API-Ressource als Beispiel für eine JSON-Datei

2.4. 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* gegen-



über einem Schema bezeichnet, falls die Elemente und Attribute dieses Dokumentes die Bedingungen des Schemas erfüllen [Mur+05]. Neben XSD existieren noch weitere Schemasprachen. Diese sind für die Arbeit unwichtig und werden daher nicht behandelt.

2.4.1. XML Schema Description (XSD)

XML Schema Description ist ein stark erweiterte Nachfolger der DTD (Document Type Definition), derzeit spezifiziert 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 [Kapitel 3.2 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

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?> ❶
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://api.spreadshirt.net" version="1.0" elementFormDefault="qualified"> ❷
3   <xs:import namespace="http://www.w3.org/1999/xlink" schemaLocation="xlink.xsd"/> ❸
4   ...
```

Listing 2.6: Beginn der XSD-Datei für die Spreadshirt-API

Eine XSD-Datei beginnt wie jede XML-Datei mit der XML-Deklaration ❶.

Das Wurzelement der Schemadefinition zeigt ❷. Das Attribut `xmlns:xs="http://www.w3.org/2001/XMLSchema"` führt den Namespace-Prefix *xs* ein und gibt außerdem an, dass die Elemente und vordefinierten Datentypen (Ab-



bildung 2.2) aus dem Namensraum *http://www.w3.org/2001/XMLSchema* verwendet werden. Durch das Attribut *targetNamespace* wird der Namensraum der Elemente festgelegt, die in dieser Schemadefinition definiert werden. *Version* gibt die XSD-Version an. Der Wert des Attributs *elementFormDefault* gibt an, ob Elemente des Schemas den *targetNamespace* explizit angeben müssen (»qualified«) oder ob dies implizit geschieht (»unqualified«), die Angabe ist optional.

Externe Schemadefinitionen lassen sich unter Angabe des Namensraumes und einer URI zu der XSD-Datei einbinden ❸.

XML-Schema Description erlaubt die Definition von simplen Typen (»SimpleType«) und Typen mit strukturiertem Inhalt (»ComplexType«).

```
1 <xs:simpleType name="unit">
2   <xs:restriction base="xs:string"> ❶
3     <xs:enumeration value="mm"/> ❷
4     <xs:enumeration value="px"/> ❷
5   </xs:restriction>
6 </xs:simpleType>
```

Listing 2.7: Beispiel für einen SimpleType namens »unit« der Spreadshirt-API

SimpleType-Definitionen dienen zur Beschreibung einfacher Typen wie *Enumeratoren*, oder *Listen* für Daten eines primitiven Typs. Ein Beispiel für die Definition eines Enumerators durch einen SimpleType zeigt Listing 2.7. Der Basisdatentyp des Enumerators wird dabei durch die Angabe des Attributs *base* ❶ festgelegt. Zuordnung von Werten zu dem Enumerator zeigt ❷.

Durch einen SimpleType definierte Listen sind durch Leerzeichen separierte Strings, sie werden meist für den Wert eines Attributes einer XML-Datei verwendet.

```
1 <xs:simpleType name=colors>
2   <xs:list itemType="xs:string"/>
3 </xs:simpleType>
```

Listing 2.8: Beispiel für einen Listentyp definiert durch einen SimpleType

```
1 <test>red green blue</test>
```



Listing 2.9: Beispielinstanz für Typ aus Listing 2.8

Die Definition eines strukturierten Typs zeigt Listing 2.10.

```
1 <xs:complexType name="abstractList" abstract="true"> ❶
2   <xs:sequence> ❷
3     <xs:element minOccurs="0" ❸ name="facets"> ❹
4       <xs:complexType> ❺
5         <xs:sequence>
6           <xs:element xmlns:tns="http://api.spreadshirt.net"
7             minOccurs="0" maxOccurs="unbounded" ref="tns:facet" ❻
8             />
9         </xs:sequence>
10      </xs:complexType>
11    </xs:element>
12  </xs:sequence>
13  <xs:attribute xmlns:xlink="http://www.w3.org/1999/xlink" ref="xlink:href"/>
14    ❼
15  <xs:attribute type="xs:Long" name="offset"/> ❽
16  <xs:attribute type="xs:string" name="query"/>
17  ...
18 </xs:complexType>
```

Listing 2.10: Beispiel für eine Schemabeschreibung mit XSD anhand des »abstractList«-Typs der Spreadshirt-API

Das *ComplexType*-Tag ❶ umschließt die Definition des strukturierten Typs. XML-Schema Description erlaubt das Definieren von abstrakten Typen, nur Ableitungen davon dürfen als Instanzen in einem Dokument auftreten. Abgeleitete Typen dürfen dabei den abstrakten Typ *erweitern* oder *einschränken* (»derivation by extension/restriction«).

Mit *Reihenfolgeindikatoren* ❷ kann die Ordnung von Elementen festgelegt werden. Elemente unterhalb eines *Sequence*-Tags dürfen nur in der Abfolge auftreten, in der sie definiert worden sind. Das *All*-Tag ❹ hingegen erlaubt das Vorkommen ohne festgelegte Reihenfolge. Der Reihenfolgeindikator *Choice* erlaubt nur eines der Elemente, die unterhalb dieses Tags vorkommen.

Durch die optionale Angabe von *Häufigkeitsindikatoren* ❸ kann festgelegt werden wie oft ein Element an der definierten Stelle vorkommen darf. Entfällt



dies, entspricht der Wert von *minOccurs* "1" und *maxOccurs* "1", das heißt, das Element darf genau einmal an dieser Stelle vorkommen.

Elemente einer **XML**-Datei werden durch das gleichnamige *Element* ④ im **XSD** definiert. Ein Element benötigt die Angabe eines Namens und Typs. Die Angabe des Typs kann dabei als Referenz auf die Typdefinition ⑥ oder als Definition unterhalb des Element-Tags erfolgen ⑤.

Attribute eines **XML**-Tags werden durch das *Attribute*-Element definiert. Dies geschieht durch Angabe von Name und Typ ⑦ oder durch eine Referenz auf eine Attributdefinition ⑥.

Referenzen haben die Form *Namensraumbezeichner:Elementname*. Wobei mit *Elementname* jedes Element der Schemabeschreibung gemeint ist, welches ein *name*-Attribut besitzt. Der konkrete Namensraum eines solchen Bezeichners wird vorher mit der Angabe eines Attributes in dieser Form eingeführt:

$$\underbrace{\text{xmlns}}_{\textcircled{1}} : \underbrace{\text{tns}}_{\textcircled{2}} = \underbrace{"\text{http://api.spreadshirt.net}"}_{\textcircled{3}}$$

- ① **XML**-Namespace
- ② Namensraumbezeichner
- ③ Konkreter Namensraum

2.5. RESTful Web Service

Representational State Transfer (deutsch: »Gegenständlicher Zustandstransfer«) ist ein Softwarearchitekturstil für Webanwendungen, welcher von Roy Fielding — einer der Hauptautoren der **HTTP**-Spezifikation — 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.

2.5.1. Elemente von REST

Im folgenden werden die Grundbausteine einer **REST**-Anwendung erläutert. Abschnitt 2.5.1.4 beschreibt die Komponenten, die an einer Aktion auf einer



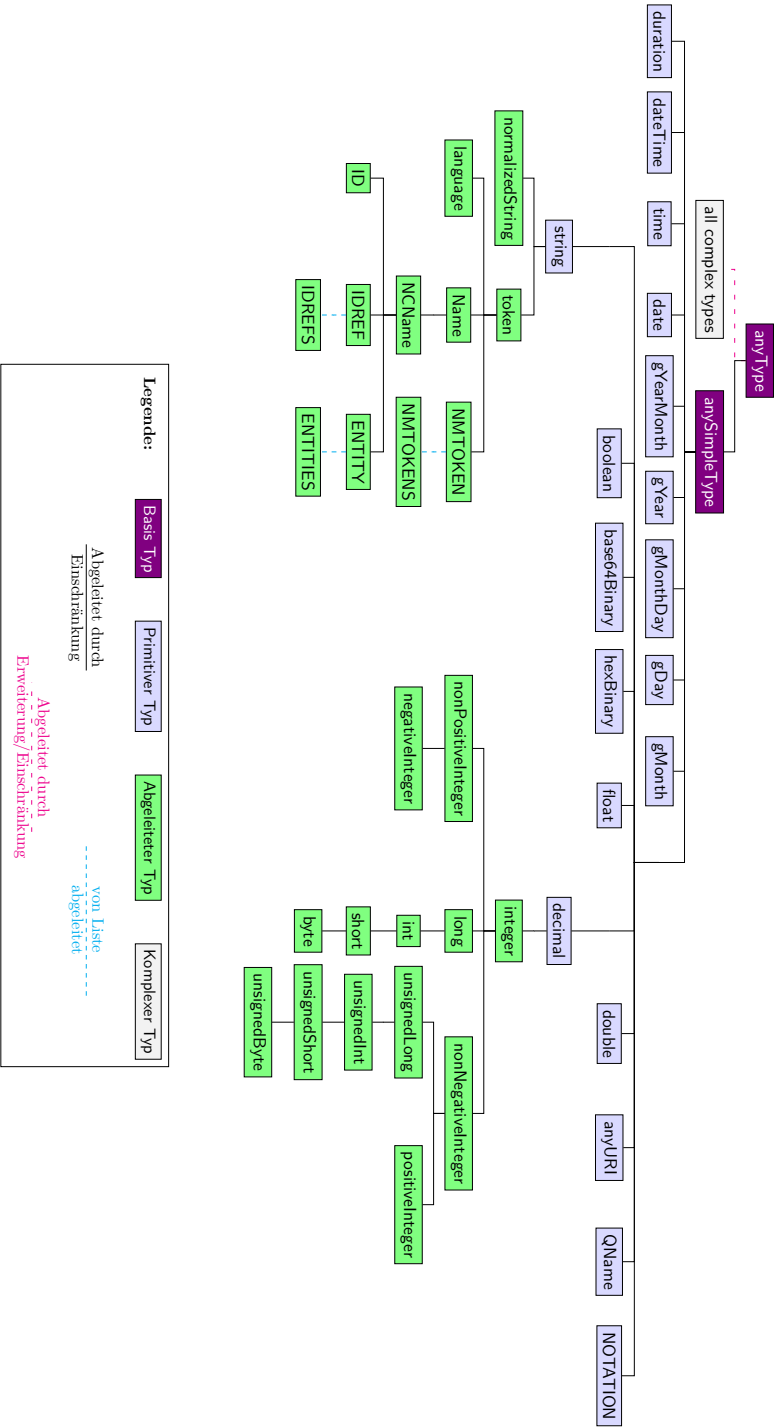


Abbildung 2.2.: vordefinierte XSD Datentypen nach [W3C12] Kapitel 3

Ressource beteiligt sind. Dieser Abschnitt basiert auf Kapitel 5.2 von [Fie00, S. 86 ff.].

2.5.1.1. Ressource

Eine Ressource stellt die wichtigste Abstraktion von Information im [REST](#)-Modell dar. Fielding definiert eine *resource* 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 ein Ressourcenbezeichner, meist eine [URI](#) (Abschnitt 2.5.2.1), immer dieselbe Ressource, nicht aber deren Wert oder Zustand.

Ein Beispiel für eine Ressource in der Spreadshirt [API](#) `/users/{userid}`, wobei `{userid}` die Identifikationsnummer 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`.

2.5.1.2. 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, Abschnitt 2.5.2.4. 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 (Abschnitt 2.2.2), beispielsweise 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 2.11: Beispiel zu Metainformationen für REST-Repräsentation aus WADL-Datei der Spreadshirt-API

Ein Beispiel für eine solche Angabe von Metainformationen ist in Listing 2.11 zu finden, ❶ zeigt dies in Form einer *Typangabe* und ❷ eines *mediaType*-Attributes.

2.5.1.3. 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:

- Anfrage-Kontrolldaten
- Ressourcenidentifizierung (Ressourcenbezeichner)

(optional) Repräsentation der Ressource

Ausgangsparameter:

- Antwort-Kontrolldaten

(optional) Metainformationen

(optional) Repräsentation der Ressource

2.5.1.4. Komponenten

Ursprungsserver:

Serverseitiger Konnektor, der die angeforderten Ressourcen verwaltet. Er



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](#)]

ist die einzige Quelle für Repräsentationen sowie der endgültige Empfänger von Änderungsanfragen an seine Ressourcen (Abschnitt [2.5.2.1](#))

Proxy:

Zwischenkomponente, die explizit vom Client verwendet werden kann, aus Sicherheits-, Performance- oder Kompatibilitätsgründen

Gateway:

dient als Schnittstelle zwischen Client- und Servernetzwerk und 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:

clientseitiger *Konnektor*, der die Anfrage an die [API](#) startet und einziger Empfänger der Antwort ist. In den meisten Fällen ist dies einfach ein *Webbrowser*

2.5.2. REST-Prinzipien

Die fünf grundlegenden [REST](#)-Prinzipien nach [[Til09](#), 11 ff.] lauten:

- Ressourcen mit eindeutiger Identifikation
- Verknüpfungen / Hypermedia
- Standardmethoden (Abschnitt [2.5.2.3](#))
- unterschiedliche Repräsentationen



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

Abbildung 2.3.: Beispiel-URI, um den Artikel 42 aus dem Warenkorb 84 anzusprechen

- statuslose Kommunikation

Nachfolgend werden diese Prinzipien im Detail erläutert.

2.5.2.1. 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 (Beispiel Abbildung 2.3).

2.5.2.2. 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.

2.5.2.3. Standardmethoden

Durch die Nutzung von *Standardmethoden* ist abgesichert, dass 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. Bei der Implementierung eines [RESTful](#) Web Service ist darauf zu achten, dass das serverseitige Verhalten dem der HTTP-Methode entspricht (beispielsweise Idempotenz).

2.5.2.4. Repräsentationen von Ressourcen

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



dadurch mittels *Query-Parameter* oder als Information im [HTTP](#)-Header (Abschnitt 2.2.2) 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 zum Beispiel `application/json`.

2.5.2.5. 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 Sitzungsstatus 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

In Summe wird dadurch eine verbesserte Skalierbarkeit erreicht.

Das Mitsenden von Statusinformationen bei jeder Anfrage wirkt sich, durch eine erhöhte Netzwerklast, nachteilig aus.

2.6. 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 (eigene Übersetzung):

Menge von Ressourcen:

Analog einer Sitemap, also die Übersicht aller verfügbaren Ressourcen



Beziehungen zwischen Ressourcen:

die kausale und referentielle Verknüpfung zwischen Ressourcen

Methoden die auf Ressourcen angewendet werden können:

die von der jeweiligen Ressource unterstützten [[HTTP](#)]-Methoden, deren Ein- und Ausgabe, sowie die unterstützten Formate

Ressourcenrepräsentationen:

die unterstützten [MIME](#)-Typen und verwendeten Datenschemata (Abschnitt [2.4.1](#))



```

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                 </response>
37             </resource>
38             ...
39         </resources>
40     </application>

```

Listing 2.12: 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>` Ele-



ments 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* weglassen.

Abbildung 2.4 zeigt die baumartige Struktur einer [WADL](#)-Datei. Die Kanten enthalten Informationen über Kardinalität oder die Form des Elements (Definition/Referenz). Innerhalb der Elemente sind deren Attribute aufgelistet, oder ...falls diese schon aufgelistet wurden.

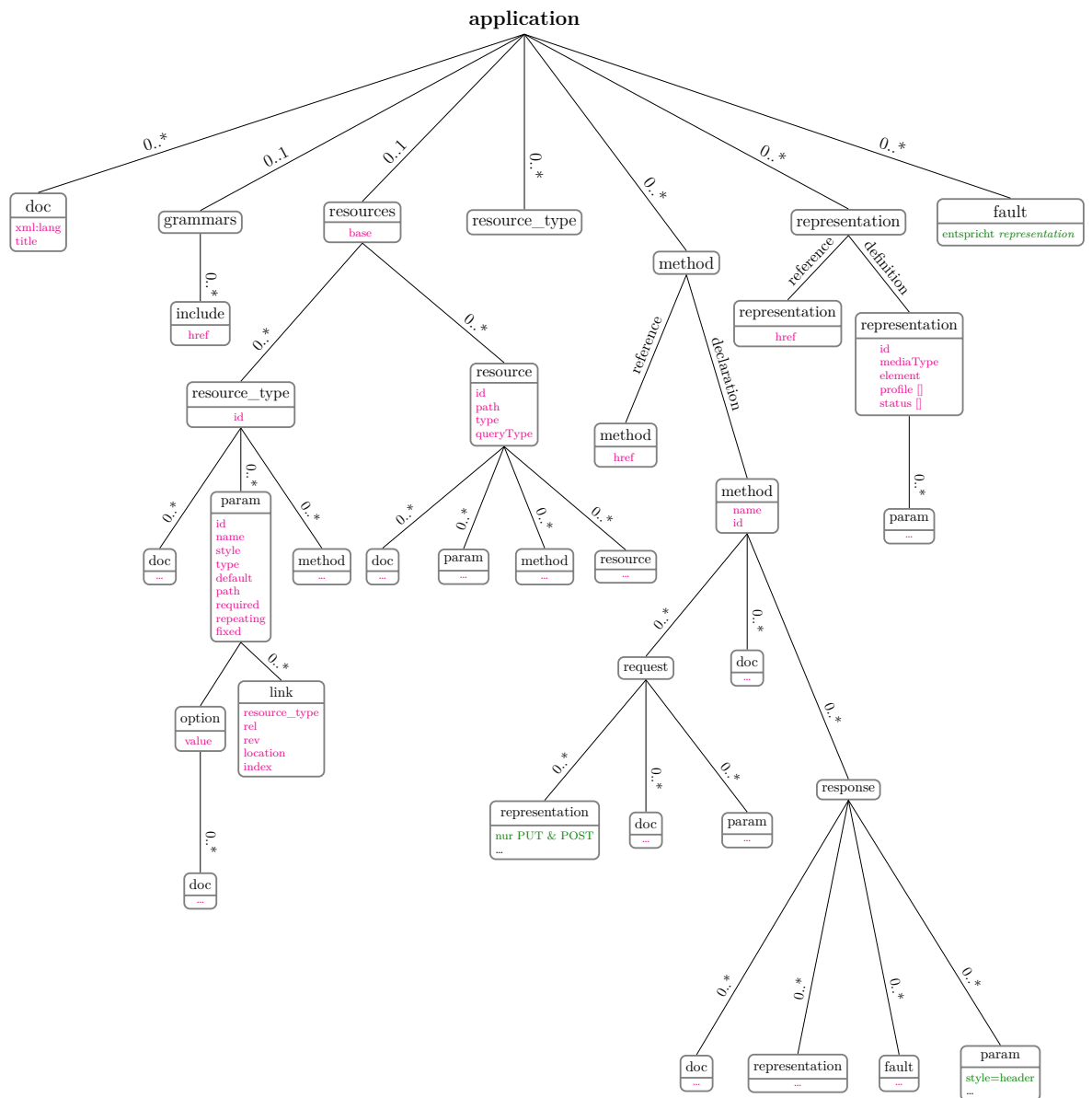


Abbildung 2.4.: Struktur einer WADL-Datei, nach Kapitel 2 [Had06]

3. Codegenerierung

»All problems in computer science can be solved by another level of indirection.«

[WO08, Kapitel 17]
Wilson und Oram (2008)

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.

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 7 (Codegenerator). *Ein Codegenerator ist ein Programm, welches aus einer höbersprachigen Spezifikation (m. a. W.: auf einem höheren Abstraktionslevel als die zur Implementierung verwendete Programmiersprache), 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 beispielsweise [WADL](#) inklusive seiner verwendeten Schemata die Spezifikationsprache 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, beispielsweise bei einem Compiler in einen *Abstrakten Syntaxbaum* (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-

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 und nachfolgender Regenerierung korrigiert werden und bedürfen somit keiner manuellen Korrektur aller generierten Klassen.

Die Erstellung eines Generatorsystems geht in der Regel 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 (»Tier«, zu deutsch: Stufe):

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 die Quelldatei übernommen, sondern meist direkt zu dem Compiler oder Interpreter weitergeleitet.



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-Generators entspricht der des Partial-Class Generators, mit der Ausnahme, dass ein vollständiger Code erzeugt wird, der keiner Nacharbeit bedarf.

n-Tier Generator:

Ein n -Tier Generator erzeugt neben dem eigentlichen Quellcode noch beliebige andere Informationen, beispielsweise 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*).



3.2. Datenmodell

Das Datenmodell enthält die Informationen der Spezifikation und dient als Eingabe für den Generator. Es ist somit die *Basis der Codegenerierung*. Westertinen u. a. definieren den Begriff in [Wes+01] folgendermaßen¹:

Definition 8 (Datenmodell). *Ein Datenmodell ist im Grunde die Darstellung eines Informationsmodells unter Berücksichtigung einer Menge von Mechanismen für die Darstellung, Organisation, 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) oder 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 (Abschnitt 3.1.1) beispielsweise 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.

3.2.1. Abstract Syntax Tree (AST)

Der Generator erzeugt aus den Elementen des Sprachenmodells (Abschnitt 4.1.4) einen *Abstract Syntax Tree*. Die Bestandteile der generierten Klassen beschreiben einen Abstrakten Syntaxbaum.

¹eigene Übersetzung

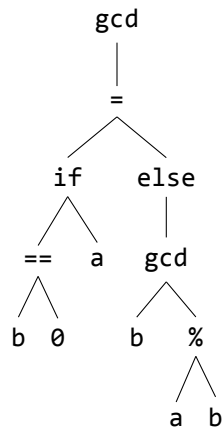


Abbildung 3.1.: Beispiel [Abstract Syntax Tree](#) für den rekursiven euklidischen Algorithmus

Eine anschauliche Definition eines [Abstract Syntax Tree](#) bietet [Aho+06, S. 69] (eigene Übersetzung):

Definition 9 (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 [Abstract Syntax Tree](#) keine Formatierungsspezifische Syntax (beispielsweise Klammern).

3.3. Objektorientierte Sprachen

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



Im Gegensatz zu *prozeduralen Sprachen* (z.B. C), in denen ein Programm eine Liste von Funktionen ist, wird es in objektorientierten Sprachen aus der Interaktion von *Objekten* gebildet.

Objektorientierte Sprachen stellen eine Teilmenge der imperativen Sprachen dar. Programme aus einer imperativen Sprache bestehen dabei aus einer Folge von *Anweisungen* (»Statements«). Anweisungen sind Befehle, formuliert in der Syntax der Programmiersprache, beispielsweise Zuweisungen, Unterprogrammaufrufe oder Schleifen. *Ausdrücke* (»Expressions«) unterscheiden sich zu Anweisungen, indem sie nebenwirkungsfrei sind und nach der Auswertung einen Wert zurückliefern. Viele Programmiersprachen vermischen beide Konstrukte. Ein Beispiel dafür ist der Inkrement-Operator (`++`). Er inkrementiert den Wert einer Variable und liefert ihn zurück, ist also nicht nebenwirkungsfrei. Ein Operator ist ein Operationssymbol der Programmiersprache mit einer bestimmten Stelligkeit und Notation (Post-, Prä-, Infix). Listing 3.1 zeigt den Aufbau eines Ausdrucks, dargestellt durch eine EBNF.

```
Ausdruck ::= Wert
           | Bezeichner
           | [ Operator ] Ausdruck
           | Ausdruck [ Operator ]
           | Ausdruck Operator Ausdruck
```

Listing 3.1: Aufbau eines Ausdrucks einer imperativen Programmiersprache als EBNF

[] optional, | Auswahl, ::= Definition.

3.3.1. Elemente

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

Objekte:

Sind eine elementare logische Einheit, kapselt Variablen und *Methoden* (Kapselung) und schützt private Daten vor äußerem Zugriff (Data-Hiding). Sie 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 objektorientierten Sprachen bieten Möglichkeiten der Vererbung, d.h., dass 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.

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 beispielsweise 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. Typsystem

Den Begriff »Typsystem« definiert Markus Voelter in [Mar13, S. 253] so (eigene Übersetzung):

Definition 10 (Typsystem). *Ein Typsystem ordnet den Programmelementen Typen zu und prüft die Konformität dieser Typen nach bestimmten vordefinierten Regeln.*

Der *Typ* ist eine Eigenschaft eines Programmkonstruktes, ein solches Konstrukt ist z.B. eine Konstante, Variable, Methode.

Es wird zwischen zwei grundlegenden Formen von Typsystemen unterschieden, *statisch* und *dynamisch*. Das Unterscheidungskriterium ist der Zeitpunkt der Typprüfung. Dynamische Typsysteme prüfen erst während der Laufzeit des Programms, bei statischen Typsystemen hingegen übernimmt der *Compiler* diese Aufgabe. Ein statisches Typsystem erfordert in den meisten Fällen die explizite Angabe des Typs durch den Programmierer. Programmiersprachen mit statischen Typsystemen, welche *Typinferenz* bieten, können oft anhand des Wertes eines Konstruktes seinen Typ erkennen und ersparen in diesen Fällen dem Programmierer dessen explizite Angabe.

3.3.3. 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 statisch typisierten Sprachen wie *Java* muss PHP zur Ausführung nicht kompiliert werden. PHP ist *dynamisch typisiert* und wird von einem Interpreter — dem



namensgebenden *Hypertext Preprocessor* — ausgeführt. Es werden seit Version 5.0 mehrere Programmierparadigmen unterstützt, neben imperativer- auch objektorientierte Programmierung. Version 5.3 fügte die Unterstützung von funktionaler Programmierung durch die Verwendung von [Closures](#) hinzu.

- ❶ Die Start- und Endtags eines PHP-Files, wobei letzteres optional ist. Deren Funktion ist die Abgrenzung vom umliegenden Markup, beispielsweise 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 beispielsweise String oder Integer entspricht.
- ❹ Statische Methoden, können ohne Instanz der umgebenden Klasse aufgerufen werden.



```
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->
35                 operations);
36             $BatchDTO = new BatchDTO(/* operationDTO */ $operations);
37             return $BatchDTO;
38         }
39
40         ...
41     }
42     ?> ❶
```

Listing 3.2: Durch den Generator erzeugte BatchDTO Datenklasse der Spreadshirt-API als Beispiel für eine PHP-Datei



4. Generatorsystem für die Spreadshirt-API

»An abstraction is one thing that represents several real things equally well.«

[Dij07, zitiert von David Lorge Parnas]
Dijkstra (2007)

Nachdem in den vorangegangenen Kapiteln allgemeine Grundlagen über Webservices (Kapitel 2) und Codegenerierung (Kapitel 3) behandelt wurden, werden in diesem Kapitel die konkreten Datenmodelle, der Generator und der Aufbau der generierten Bibliothek für die Spreadshirt-API beschrieben.

4.1. Konkrete Datenmodelle

Neben den Datenmodellen, welche die Informationen beinhalten aus denen der Generator ausführbaren Quellcode erzeugt, wird im folgenden Abschnitt das im Sprachenmodell verwendete Kompositum-Muster erläutert.

4.1.1. REST-Modell

Zuerst muss die abstrakte Beschreibung der Spreadshirt-API von der XML-Form, bestehend aus einem WADL und einem oder mehreren Schemabeschreibungen, 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



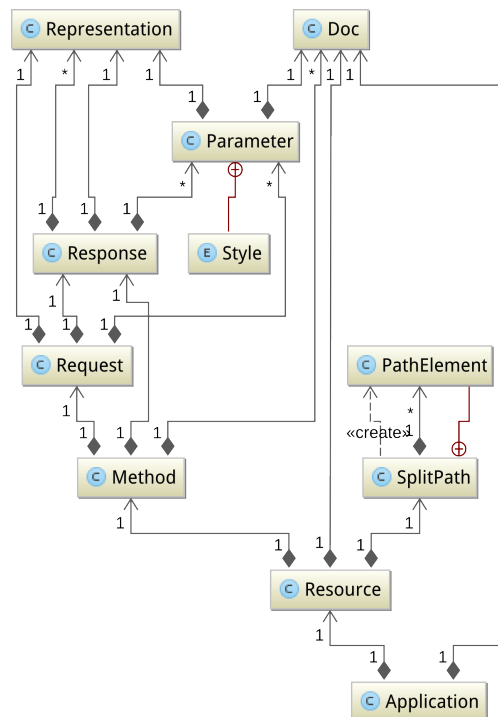


Abbildung 4.1.: UML Klassendiagramm des REST-Modells

(Anhang A.1) in ein *Document Object Model* (kurz DOM) überführt, welches im Arbeitsspeicher gehalten wird und damit einen schnellen Zugriff für nachfolgende Operationen darauf erlaubt. In einem nächsten Schritt wird das DOM, welches noch viele XML spezifische Informationen enthält, auf die wesentlichen API beschreibenden Merkmale reduziert. Im Gegensatz zu der in Abbildung 2.4 veranschaulichten Webanwendungsbeschreibung werden Referenzen durch deren Definition im Modell ersetzt. Die Klassennamen des Datenmodells orientieren sich an den WADL Elementnamen.

Wurzelelement des Modells (Abbildung 4.1) ist die Klasse **Application**, sie enthält *Ressource*-Objekte und den Basisbezeichner der API beispielsweise `http://api.spreadshirt.net/api/v1/`.

Eine **Ressource**-Klasse enthält eine Menge von *Method*-Objekten sowie einen Ressourcenbezeichner. Dieser ist relativ zum Basisbezeichner des Wurzelelements. Die Ressourcenbezeichner können *Template-Parameter* enthalten.



Diese werden bei einer Anfrage durch einen konkreten Wert ersetzt. Beispielsweise enthält der Bezeichner für die Ressource eines bestimmten Users den Template-Parameter `{userid}`, vollständiger Ressourcenbezeichner `user-s/{userid}`. Ressourcenbezeichner werden durch die Klasse **SplitPath** repräsentiert.

Jede **Method**-Klasse enthält ein *Request* und ein *Response* Objekt. Sie enthalten die nötigen Informationen für den Aufruf der Methode, beziehungsweise über den Aufbau der Antwortnachricht.

Eine **Request**-Klasse enthält eine Liste von Query-Parametern sowie ein *Representation* und *Response* Objekt.

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.

Die Klasse **Response** enthält eine Liste mit *Representation*-Objekten und Parameter-Objekten. Die Objekte vom Typ *Representation* enthalten die Beschreibung der Daten, die bei einer erfolgreichen Anfrage an die Ressource zurückgesendet werden sowie die der Fehlermeldung, welche der Client andernfalls erhält. Zwischen einer Fehlermeldung und einer erfolgreichen Anfrage kann anhand des Werts des [HTTP](#)-Statuscodes unterschieden werden. Erfolgreiche Anfragen liefern in der Antwort meist einen Statuscode 200 **OK** oder 201 **Created** zurück, abhängig von der Anfragemethode. Die Response Parameter geben 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, so enthält das Feld `Location` des [HTTP](#)-Headers der Serverantwort eine [URL](#) auf die Ressource der angelegten Session.

Die **Representation**-Klasse 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. Das *Representation*-Objekt des Request einer PUT- oder POST-Methode charakterisiert zum Beispiel den Aufbau der Da-



ten, 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*. Beispielsweise enthält das *Representation*-Objekt der PUT-Methode auf Ressource `users/{userId}/designs/{designId}` 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, Abschnitt [4.1.3](#).

Ein Objekt der **Doc**-Klasse enthält einen Titel und eine Kurzbeschreibung des zugehörigen Elements. Der Generator erzeugt daraus Quellcodekommentare für die Dokumentation der Bibliothek.

4.1.2. Schema-Modell

Wurzel des Schemadatenmodells ist die Klasse **Schema**. Ein Schema kann Objekte vom Typ *Complex*- und *SimpleType* sowie *Attribute* und *Element* enthalten.

[XSD](#)-Dateien erlauben das Importieren anderer Schemadefinitionen, die Klasse **Import** ermöglicht dies im Schemamodell. Sie besitzt ein Objekt des zu importierenden Schemas sowie eine [URI](#) auf die zugehörige [XSD](#)-Datei.

Primitive Schematypen werden durch die Klasse *SimpleType* abgebildet. Objekte dieser Klasse enthalten eine Kennzeichnung der Art des SimpleType (Enumerator, Liste, einfacher Wert) und bei Enumeratoren zusätzlich die einzelnen Enumeratorwerte sowie die Angabe des Basisdatentyps.

Die **ComplexType**-Klasse repräsentiert die gleichnamigen strukturierten Typen aus der Schemabeschreibung. Ein ComplexType kann Attribute, Elemente, Elementsequenzen und strukturierten Inhalt (*ComplexContent*) enthalten.

ComplexContent kann die gleichen Objekte wie *ComplexType* enthalten, sowie einen Basistyp der erweitert oder eingeschränkt wird (»derivation by extension/restriction«).



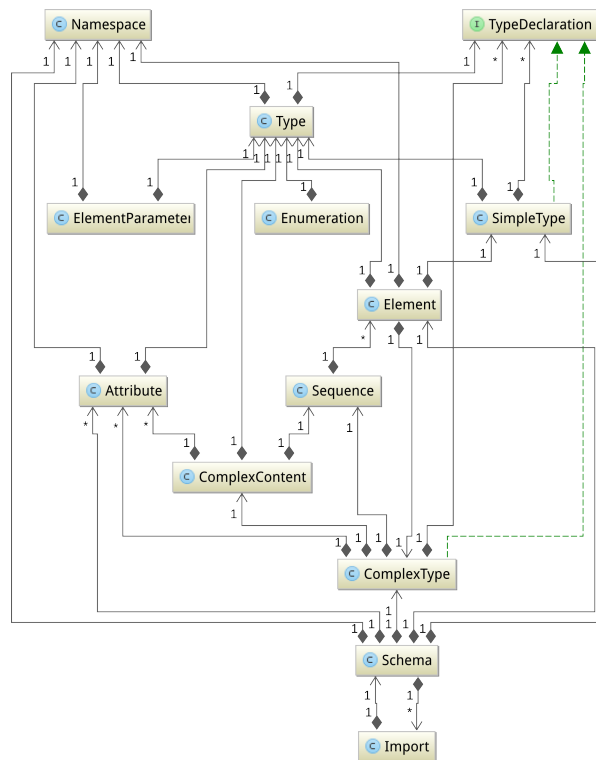


Abbildung 4.2.: UML Klassendiagramm des Schemadatenmodells

Attribute werden durch die gleichnamige Klasse **Attribute** gekapselt, sie besitzen einen Attributnamen sowie eine Definition ihres Typs.

Elementsequenzen werden durch die **Sequence**-Klasse repräsentiert. Sie enthält einen Reihenfolgeindikator und die Elemente der Sequenz.

Objekte der Klasse **Element** besitzen einen Bezeichner sowie einen Complex- oder SimpleType und optional eine Angabe der Auftrittshäufigkeit. Die Klasse *ElementParameter* dient nur zur Kapselung der Daten, welche an den Konstruktor der Elementklasse gegeben werden.

Durch die Klasse **Namespace** werden der Namensraumbezeichner und der konkrete Namensraum eines Typs aus dem Schema gekapselt.



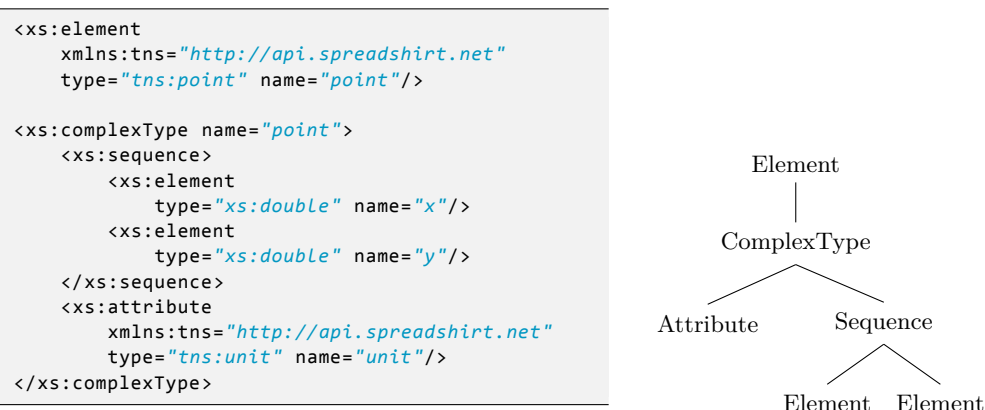


Abbildung 4.3.: Datentyp Point mit Gegenüberstellung im Schemamodell

4.1.3. Applikationsmodell

Das Applikationsmodell ist die Gesamtheit des **REST**- und Schemamodells. Referenzen auf Typenbeschreibungen im **REST**-Modell werden durch deren Definition im Schemamodell ersetzt. Dieses gemeinsame Modell dient dem Generator als Eingabequelle.

4.1.4. Sprachenmodell

Um die gewünschte Austauschbarkeit der Zielsprache zu gewährleisten wurde ein abstraktes Sprachenmodell entworfen welches die Konstrukte einer dateibasierten objektorientierten Programmiersprache (Abschnitt 3.3) abbildet. Die gewünschte Zielsprache muss dabei die Klassen und Methoden des Modells implementieren sowie eine *Language Factory* (Abschnitt 4.2.1) bereitstellen um vom Generator genutzt werden zu können. Um Semantik und Syntax der Zielsprache im Modell zu trennen — abgesehen von Symbolen und Schlüsselwörtern — wird die Syntax in der Klasse *LanguageVisitor* (Abschnitt 4.2.1.1) gekapselt. Das Sprachenmodell kapselt somit die Semantik der Zielsprache.

Alle Interfaces des Modells (Abbildung 4.4) erweitern das Interface **Visitor**. Somit ist sichergestellt, dass alle Klassen, die diese Schnittstellen implementieren, eine *accept*-Methode für den *LanguageVisitor* bereitstellen.



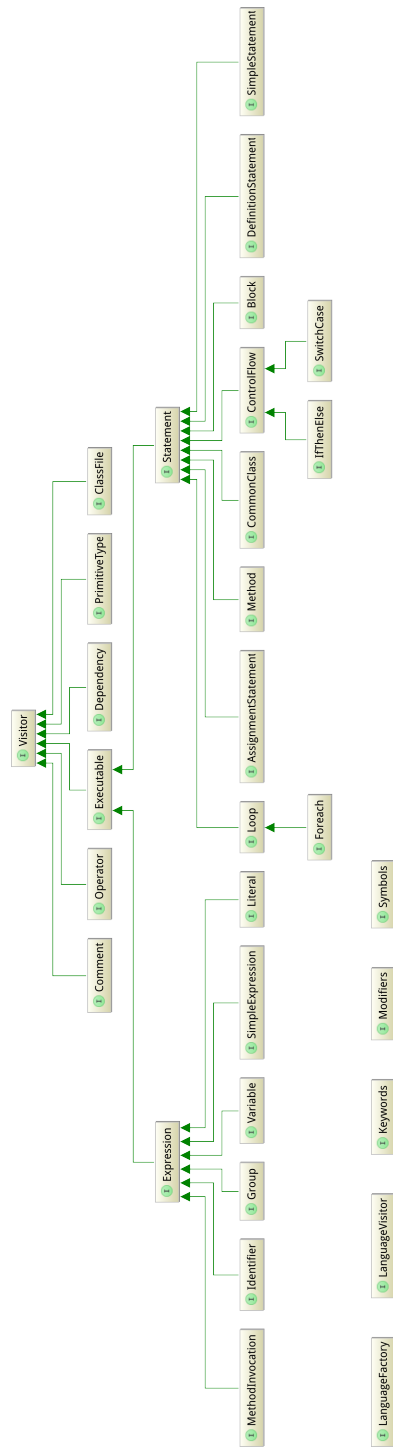


Abbildung 4.4.: UML Klassendiagramm des Zielsprachenmodells

Basis des Modells ist das Interface **ClassFile**. Es abstrahiert eine Klassendatei mit den Eigenschaften:

- Dateiname
- Namensraum
- Liste von Abhängigkeiten (*Dependency*-Klasse)
- Klassendefinition

Die Liste von Abhängigkeiten der zu generierenden Klassen muss vorher aus dem Eingabemodell ermittelt werden. Dies geschieht durch Analyse der in den Elementdefinitionen des Schemamodells enthaltenen Typen.

Dependency enthält das Schlüsselwort oder Methodenaufruf zum Import einer Quellcodedatei. In PHP werden solche Dateien bspw. so importiert: `require_once("foo.php");`.

Executable implementieren alle Elemente der Zielsprache die »ausführbar« sind. Das Modell unterscheidet dabei zwischen *Ausdruck* und *Anweisung* (Abschnitt 3.3).

Das Interface **CommonClass** dient der Implementierung einer Klassendefinition. Da das Interface selbst *Statement* erweitert, kann eine Klasse weitere Klassendefinitionen beinhalten. Eine Klassendefinition besteht dabei aus einem Klassename, Modifiers und aus einer Menge von Statements:

- *DefinitionStatement* zur Einführung von lokalen Variablen.
- *Method* zur Definition von Methoden.

Das **Modifier**-Interface deklariert Methoden um die Schlüsselwörter für *Sichtbarkeitsmodifikatoren* (»Access Modifier«) und »Non Access Modifiers« wie **static** oder **final** zu erhalten.

Durch das **Method**-Interface kann eine Methodendefinition implementiert werden. Eine Methode beinhaltet dabei:

- Modifier
- Methodenname
- Rückgabetyt
- Liste von Parametern (Parameter können dabei alle Klassen sein die *Expression* implementieren)



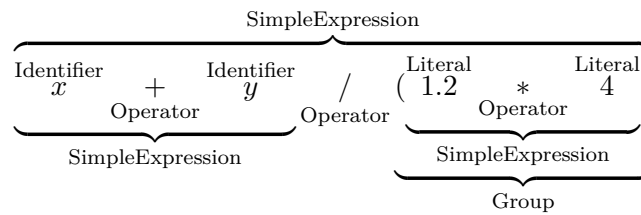


Abbildung 4.5.: Beispiel für den Aufbau einer »Expression« im Sprachenmodell

- *Block*

Block kapselt eine Menge von Statements.

Operatoren der Zielsprache müssen das Interface **Operator** implementieren, ein Operator ist durch seine Arität (Stelligkeit), Notation und sein Symbol gekennzeichnet. Zum Beispiel ist der Dereferenzierungsoperator in PHP *zweistellig*, *Infix* notiert und durch das Symbol \rightarrow gekennzeichnet.

Keywords und **Symbols** dienen zur Kapselung der Schlüsselwörter und Symbole einer Sprache. **Keywords** enthält Methoden zur Abfrage typischer Schlüsselwörter wie `class`, `import`, `new` oder `this`. Sprachspezifische Symbole wie *Verkettungs*- und *Scope*-Operatoren oder Präfixe für Variablennamen können über Methoden der Klasse **Symbols** vom Generator abgefragt werden.

4.1.4.1. Kompositum-Pattern im Sprachenmodell

Zweck des Composite- oder auch Kompositum-Patterns ist die Gleichbehandlung von Einzelelementen und Elementgruppierungen in einer verschachtelten Struktur (z. B. Baum), sodass aus Sicht des Clients keine explizite Unterscheidung notwendig ist (nach [ES13, S. 102]).

Anwendung fand das Pattern an zwei Stellen im Modell, erstens in den Klassen welche **Expression** und zweitens, welche **Statement** implementieren. Ein Beispiel für den Aufbau eines Ausdrucks durch Klassen des Interface *Expression* aus dem Sprachenmodell zeigt Abbildung 4.5. Von *Statement* abgeleitete Klassen können ebenso eine Baumstruktur bilden. *Block* kann beispielsweise selbst wieder Codeblöcke — also Statements vom Typ *Block* — enthalten.



4.2. Codegenerator

Nachdem in Abschnitt 4.1 die Datenmodelle des Generators betrachtet wurden, widmet sich dieser Abschnitt nur dem Aufbau des Codegenerators und den dort verwendeten Entwurfsmustern, *Factory* (Abschnitt 4.2.1) und *Visitor* (Abschnitt 4.2.1.1).

Die Aufgabe des Generators ist die Transformierung des Applikationsmodells in das Modell der Zielsprache. Als Form wurde die des Tier-Generators gewählt, da vollständiger Code vom Generator erzeugt werden soll der keiner Nacharbeit mehr Bedarf.

4.2.1. Language Factory

Das *Factory*-Pattern behandelt das Problem Familien von Objekten erzeugen zu wollen ohne die konkreten Klassen zu spezifizieren, sondern nur Interfaces festzulegen [ES13, S. 26]. Um eine Zielsprachenunabhängigkeit zu erreichen, wird dem Generator bei der Erzeugung eine Factory übergeben die das Interface *Language Factory* des Sprachenmodells implementiert.

Der Generator erzeugt Sprachelemente nur über diese Factory. Ein Aufruf einer Factorymethode gibt ein Sprachelement der Zielsprache zurück, der Generator kennt aber nur den Interface-Typ. Für ihn ist die konkrete Implementierung somit transparent. Die *Language Factory* bildet damit die Schnittstelle zwischen dem Generator und der Implementierung des Zielsprachenmodells.

4.2.1.1. Language Visitor

Eilebrecht und Starke definieren den Verwendungszweck des Patterns in [ES13, S. 60] so:

»Dieses Pattern ermöglicht es, neue Operationen auf den Elementen einer Struktur zu definieren, ohne die Elemente selbst anzupassen.«

Die Aufgabe des *Language Visitor* im Generator ist die Transformation des Sprachenmodells in eine Zeichenketten-Darstellung. Wie in Abschnitt 4.1.4 schon erwähnt, enthält die Klasse, die das *LanguageVisitor*-Interface implementiert, Regeln für eine syntaktisch korrekte Ausgabe des Sprachenmodells.



Zusätzlich können in den `LanguageVisitor` »code conventions« implementiert werden, beispielsweise Einrückungstiefen, Zeilenlängen etc.

4.2.2. Ausgabemodul

Das Ausgabemodul beinhaltet Methoden zur Speicherung der Zeichenketten-darstellung aus dem *Language Visitor*. Üblicherweise ist dies die Speicherung in Dateiform, es ist aber ebenso die Ausgabe auf `stdout` oder die Speicherung in einer Datenbank möglich.

4.2.3. Generatorablauf

Die Abbildung 4.6 stellt den Prozess der Bibliotheksgenerierung in Diagramm-form dar.

Zu Beginn steht die Analyse der [API](#)- und Schema-Beschreibung sowie die Überführung in die entsprechenden Datenmodelle. In einem nachgelagerten Schritt werden diese beiden Datenmodelle zu dem *Applikationsmodell* zusammengeführt. Der Generator erhält dieses Modell und eine *LanguageFactory* als Eingabe. Daraus wird das *Zielsprachenmodell* generiert, welches den zu erzeugenden Code in Form eines [Abstract Syntax Tree](#) enthält. Aus diesem Modell erzeugt das Ausgabemodul mit Hilfe des *LanguageVisitor* die Bibliotheksdateien. Der *LanguageVisitor* enthält die syntaktischen Informationen, um aus dem [Abstract Syntax Tree](#) gültigen Code zu erzeugen.

4.3. Client-Bibliothek

Die generierte Client-Bibliothek lässt sich in 2 verschiedene Arten von Klassen gliedern:

1. die Elemente und Typen aus der [XML](#)-Schemabeschreibung repräsentieren (*Datenklassen*);
2. die [API](#)-Ressourcen und deren Methoden abbilden (*Ressourcenklassen*).

Zusätzlich wurden für Aufgaben die keiner explizite Generierung bedürfen, wie [HTTP](#)-Methodenaufrufe, manuell Klassen mit statischen Methoden erstellt.



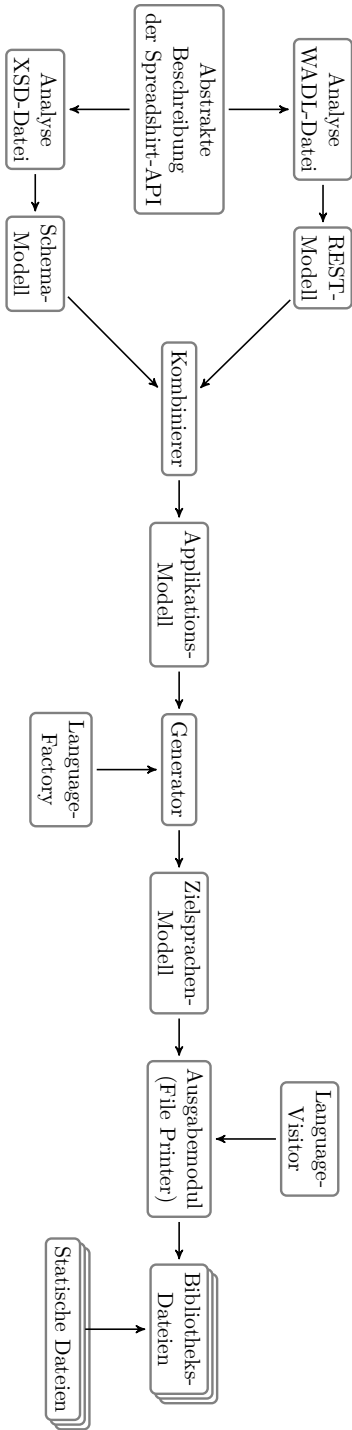


Abbildung 4.6.: Ablaufdiagramm des Generators

Dem Generator werden die Klassen als Abhängigkeiten im Eingabemodell bekannt gegeben und an entsprechender Stelle durch ihn als *Import*-Anweisungen in der Bibliothek eingefügt. Keine explizite Generierung wird benötigt wenn der zu erzeugende Code keine oder sehr wenig variable Bestandteile enthält.

4.3.1. Datenklassen

Die Datenklassen sind die zielsprachenabhängige Repräsentation der Elemente und Typen aus der [XML](#)-Schemabeschreibung.

Der Name der Klasse entspricht dabei der Bezeichnung des Typs oder des Elements. Die Variablen einer solchen Klasse sind die Attribute und Elemente aus der Schemabeschreibung des Typs. PHP bietet keine Enumeratoren, deshalb werden die einzelnen Enum-Werte als statische Variablen vom Typ **string** generiert. Für alle Variablen werden außerdem Getter- und Setter-Methoden durch den Codegenerator erzeugt.

Konstruktoren zur Erzeugung von Objekten aus den Datenklassen werden ebenfalls vom Generator erstellt. Dabei werden die *Häufigkeitsindikatoren* aus der Schemabeschreibung berücksichtigt. Bei einem **minOccurs**-Wert größer eins wird das Element zu den Konstruktor-Parametern hinzugefügt. Somit ist sichergestellt, dass notwendigen Angaben auch Werte zugeordnet werden.

Methoden zur De-/Serialisierung (Abschnitt [4.3.3](#)) in eine der beiden von der Spreadshirt-API unterstützten Dokumentbeschreibungsformate ([JSON](#), [XML](#)) sind ebenfalls Bestandteil einer Datenklasse.

Listing [4.1](#) zeigt einen gekürzten Ausschnitt der generierten Datenklasse zum Element **Point** aus der Schemabeschreibung der Spreadshirt-API.

```
1  <?php
2      require_once('Unit.php');
3
4      class Point
5      {
6          private $unit; // unit
7          private $y; // double
8          private $x; // double
9
10         function __construct(
11             /* double */ $y,
```



```
12         /* double */ $x
13     )
14     {
15         $this->y = $y;
16         $this->x = $x;
17     }
18
19     public function setUnit(
20         /* unit */ $unit
21     )
22     {
23         $this->unit = $unit;
24     }
25
26     public function toJSON()
27     {
28         $json = json_decode(/* Point */ $this);
29         return $json;
30     }
31
32     public function toXML()
33     {
34         $xml = new SimpleXMLElement(/* Point */ '<Login xmlns="http://api.
35             spreadshirt.net"/>');
36         $xml->addChild(/* string */ 'unit', /* unit */ $this->unit);
37         $xml->addChild(/* string */ 'y', /* double */ $this->y);
38         $xml->addChild(/* string */ 'x', /* double */ $this->x);
39         return $xml->asXML();
40     }
41
42     public static function fromXML(
43         /* SimpleXMLElement */ $xml
44     )
45     {
46         $unit = Unit::fromXML(/* SimpleXMLElement */ $xml->unit);
47         $y = $xml->y;
48         $x = $xml->x;
49         $Point = new Point(/* double */ $y, /* double */ $x);
50         $Point->setUnit(/* unit */ $unit);
51         return $Point;
52     }
53     ...
54
55     public function getX()
56     {
57         return $x = $this->x;
```

```
58     }  
59 }  
60 ?>
```

Listing 4.1: Point-Klasse als (gekürztes) Beispiel für eine generierte Datenklasse

4.3.2. Ressourcenklassen

Ressourcenklassen sind die zielsprachenabhängige Abbildung der Ressourcenbeschreibungen aus WADL-Datei der Spreadshirt-API.

Eine Ressourcenklasse beinhaltet:

- ❶ ein Feld, welches die Basis-URL der API beinhaltet;
- ❷ ein Feld in welches die komplette URL, inklusive der ersetzten Template-Parameter (Abschnitt 4.1.1), der Ressource erhält;
- ❸ eine Menge von Feldern, die jeweils einem Template-Parameter zugeordnet sind;
- ❹ einen Konstruktor, dessen Argumente den Template-Parametern entsprechen und der aus diesen und der Basis-URL die Ressourcen-URL erstellt;
- ❺ Abbildungen der Methoden aus der Ressourcenbeschreibung. Methodenparameter, die zur Authentifizierung an der API notwendig sind, werden durch einen Parameter der Klasse APIUSER (Abschnitt 4.3.4.2) substituiert ❻.

Listing 4.2 beinhaltet die generierte Klasse zur Ressource `users/userId/-products` der Spreadshirt-API.



```

1  <?php
2      require_once('Static/methods.php');
3      require_once('Static/apiUser.php');
4
5      /* Create or list products for user. */
6      class UsersUserIdProducts
7      {
8          private $baseUrl = 'http://192.168.13.10:8080/api/v1/'; // string ❶
9          public $userId; // string ❸
10         private $resourceUrl = ''; // string ❷
11
12         /* */
13         public function POST( ❺
14             /* array */ $parameters,
15             /* ApiUser */ $apiUser,
16             /* ProductDTO */ $productDTO
17         )
18         {
19             $auth = $apiUser->getAuthenticationHeader(/* string */ 'POST',/*
20                 string */ $this->resourceUrl);
21             return Methods::post(/* string */ $this->resourceUrl,/* string */
22                 $auth,/* array */ $parameters,/* ProductDTO */ $productDTO);
23         }
24
25         /* Sample Url is: http://... */
26         public function GET( ❺
27             /* array */ $parameters,
28             /* ApiUser */ $apiUser ❻
29         )
30         {
31             $auth = $apiUser->getAuthenticationHeader(/* string */ 'GET',/*
32                 string */ $this->resourceUrl);
33             return Methods::get(/* string */ $this->resourceUrl,/* string */ $auth
34                 ,/* array */ $parameters);
35         }
36
37         function __construct( ❹
38             /* string */ $userId
39         )
40         {
41             $this->userId = $userId;
42             $this->resourceUrl = $this->baseUrl . 'users' . '/' . $userId . '
43                 products';
44         }
45     }
46 }
47 ?>

```

Listing 4.2: Klasse zur Ressource users/userId/products als Beispiel für eine Ressourcenklasse

4.3.3. De-/Serialisierer

Um *Ressourcen-Repräsentationen* (Abschnitt 2.5.1.2) mit der Spreadshirt-API transportunabhängig austauschen zu können, müssen die strukturierten Datenklassen *serialisiert* werden. In umgekehrter Richtung müssen *Repräsentation* aus der API deserialisiert — also wieder in eine Datenklasse transformiert — werden.

Die Datenklassen-Methoden zur Serialisierung und Deserialisierung besitzen einheitliche Bezeichner, nach dem Schema `toXML`, `toJSON`, respektive `fromXML`, `fromJSON`. Die Deserialisierer-Methoden sind *statisch* um das unnötige Instanzieren einer Datenklasse zu vermeiden, nur um ihre Klassendarstellung aus der serialisierten Form zu erhalten.

Beispiele für beide Arten finden sich in Listing 4.1.

4.3.4. Statische Klassen

Statische Klassen bedeutet in dem Codegenerierungskontext, dass diese *manuell* erstellt wurden. Die statischen Klassen enthalten Code der von anderen Klassen gemeinsam genutzt wird und keine variablen Bestandteile enthält.

Prinzipiell werden zwei Probleme durch statische Klassen vermindert:

1. unnötige Vergrößerung des generierten Codes
2. Vermeidung des Aufwands zur Implementierung der statischen Inhalte in das Eingabemodell des Generators

Die generierte Client-Bibliothek beinhaltet dabei zwei dieser Klassen:

1. zur Kommunikation mit der API über HTTP-Methoden
2. zur Kapselung von Authentifizierungsinformationen

Den Import beider Klassen zeigt Listing 4.2 in Zeile 2 und 3.

4.3.4.1. Nutzung der HTTP-Methoden

Um die generierten Ressourcenklassen nicht unnötig zu vergrößern wurde der *einheitliche* Vorgang zum Aufruf der HTTP-Methoden in eine manuell erstellte Klasse ausgelagert.



Listing 4.2 zeigt in Zeile 20 und 30 den Aufruf zweier solcher Methoden aus einer Ressourcenklasse.

4.3.4.2. API Authentifizierung

In der Spreadshirt-API sind geschützte und ungeschützte Ressourcen vorhanden.

Das zur Authentifizierung eines API-Nutzers verwendete Protokoll *SprdAuth* basiert auf »HTTPs Authorization Request Header« sowie dem »WWW-Authenticate Response Header« [BB13].

Die Spreadshirt-API unterstützt die Übergabe der nötigen Autorisierungsparameter als Teil der URI-Query oder in Form des *Authorization-Header*. Die erzeugte Client-Bibliothek beschränkt sich auf die Nutzung des *Authorization-Headers*, dieser besitzt folgenden Aufbau:

```
1 Authorization: SprdAuth apiKey="<apikey>", ❶
2               data="<method><url><time>",
3               sig="SHA1(<method><url><time><secret>❷)",
4               sessionId="<sessionId>" ❸
```

Listing 4.3: Aufbau des Spreadshirt Authentication Header

Die Klasse **ApiUser** kapselt die Daten, die zur Autorisierung an der API nötig sind und stellt eine Methode bereit, die dem Nutzer das Erstellen des Authorization-Headers erspart. Der Konstruktor der ApiUser-Klasse erwartet dabei die Angabe der folgenden Parameter:

Alle Methoden, die eine Autorisation benötigen erhalten vom Codegenerator einen Parameter vom Typ *ApiUser* wie ❹ in Listing 4.2 zeigt.

- *UserID*, die Identifikationsnummer des Spreadshirt-Nutzers
- *ApiKey* und *Secret* (❶, ❷), diese Informationen erhält der Nutzer, wenn er sich als API-User bei Spreadshirt registriert
- *SessionID* (❸), die SessionID ist in der Response der POST-Methode auf Ressource **sessions** enthalten.



5. Evaluierung

»There are two ways to write error-free programs;
only the third one works«

[Per82]
Perlis (1982)

In diesem Kapitel soll der Generator und die erzeugte Bibliothek gegenüber den Anforderungen evaluiert werden. Dies geschieht für die Nutzbarkeit (Abschnitt 5.6) anhand eines einfachen Anwendungsbeispiels.

5.1. PHP-Zielsprachenmodell

Das PHP-Zielsprachemodell ist die Implementierung der Schnittstellen die durch das Sprachenmodell vorgegeben sind. Eine Gegenüberstellung eines Ausschnittes einer generierten Klassendatei mit ihrer Repräsentation im [Abstract Syntax Tree](#) der aus Elementen des Sprachenmodells gebildet wird, zeigen Abbildung 5.1 und Listing 5.1.

5.1.1. Implementierung einer Zielsprache

Um eine neue Zielsprache für den Generator zu implementieren ist es notwendig alle Interfaces des Sprachenmodells (Abschnitt 4.1.4) zu implementieren. Methoden zum Erhalt von Schlüsselwörter, Sichtbarkeitsmodifikatoren und Symbole sind in den Interfaces *Keywords*, *Modifiers* und *Symbols* deklariert. Die Syntax der Zielsprache, sowie der verwendete Code-Style werden durch die Implementierung des Interfaces *LanguageVisitor* festgelegt.



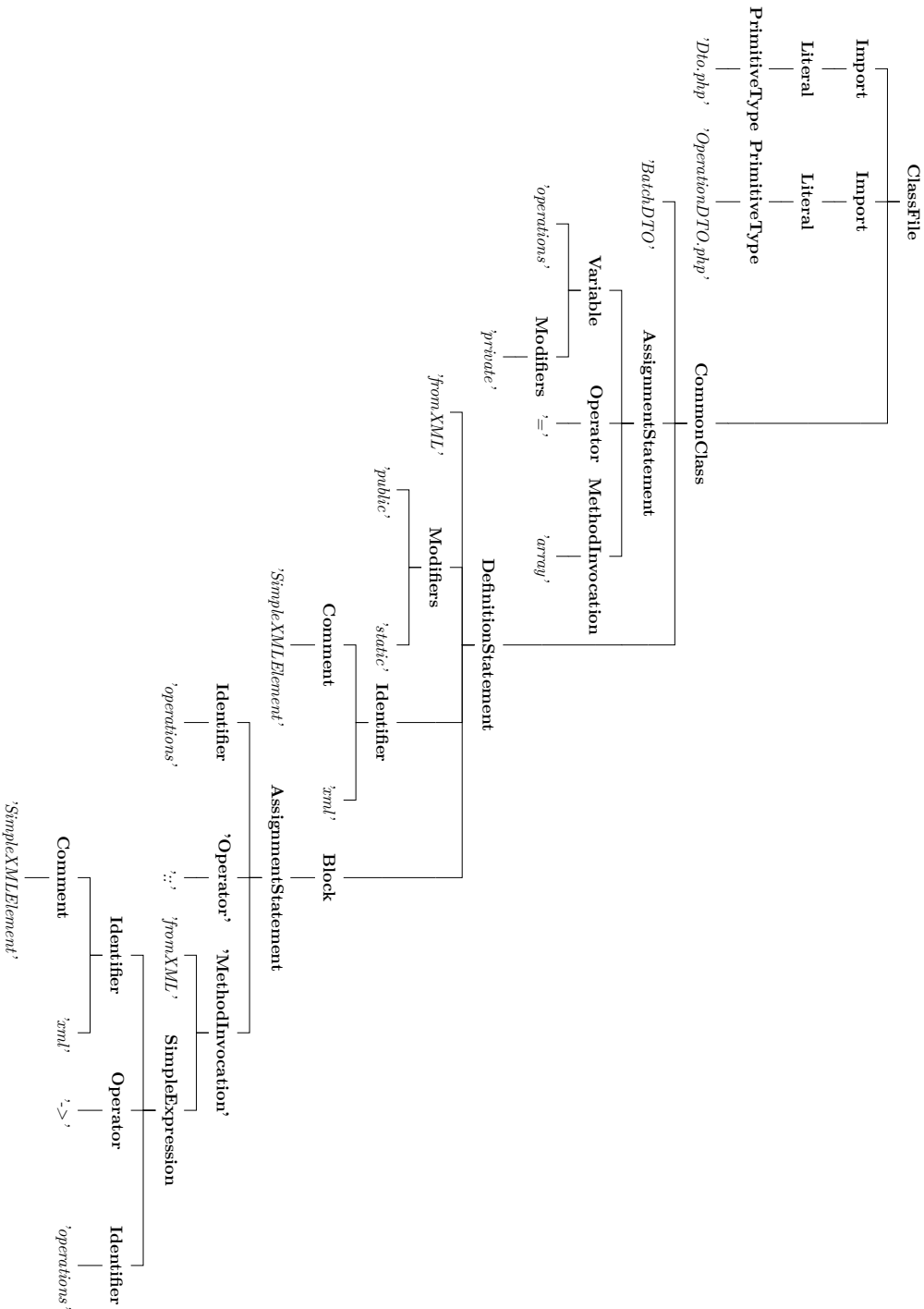


Abbildung 5.1.: Darstellung von BatchDTO aus Listing 5.1 im Sprachmodell [Klasse, Zeichenkette]

Neben den Interfaces des Sprachenmodells müssen auch die statischen Klassen in der Zielsprache implementiert werden. Die Erzeugung der Sprachelemente findet durch die *LanguageFactory* statt, die dem Generator übergeben wird.

```
1  <?php
2      require_once('Dto.php');
3      require_once('OperationDTO.php');
4
5      class BatchDTO
6      {
7          private $operations = array(); // operationDTO
8          ...
9          public static function fromXML(
10              /* SimpleXMLElement */ $xml
11          )
12          {
13              $operations = OperationDTO::fromXML(/* SimpleXMLElement */ $xml->
14                  operations)
15          }
16          ...
17      }
18  ?>
```

Listing 5.1: Ausschnitt der generierten Datenklasse BatchDTO

5.1.2. Codestyle/Lesbarkeit

Um den erzeugten Code möglichst lesbar zu gestalten, wurden folgende Konventionen im *LanguageVisitor* implementiert:

- erhöhen der Einrückungstiefe innerhalb jedes Codeblockes
- Zeilenumbruch vor jedem Block-Begrenzer ({, })
- Zeilenumbruch vor jedem Parameter in einer Funktionsdefinition (z.B. Listing 4.2, Zeile 13 und folgende)



5.2. Typsicherheit

PHP bietet durch seine dynamische Typisierung leider keine Typprüfung zur Compilezeit. Da dem Generator die erwarteten Typen bekannt sind, werden die Typnamen an entsprechender Stelle im Code durch ein Block-Kommentar eingefügt (wie u.a. in [listing 4.2](#) und [Listing 4.1](#) zu sehen). Somit ist zumindest eine manuelle Prüfung der zu übergebenden Typen für den Nutzer der Bibliothek möglich.

Durch die Implementierung speziell formatierter Kommentare — beispielsweise `/** @var [typename] */` in IntelliJ IDEA — kann eine Typprüfung durch die IDE ermöglicht werden. Um sich nicht auf eine spezielle IDE festlegen zu müssen, werden derzeit normale Block-Kommentare an dieser Stelle erzeugt.

5.3. Generierung von Tests für die erzeugte Bibliothek

Die Generierung von Tests für die erstellte Bibliothek ist derzeit noch nicht implementiert. Ein Testszenario für die erzeugten Datenklassen wäre die Prüfung auf Gleichheit vor und nach der Serialisierung. Zur Überprüfung der Ressourcenklassen werden erfolgreiche Tests der Datenklassen vorausgesetzt, da die Testdaten aus den Datenklassen erstellt werden. Die erfolgreiche Ausführung der Methoden einer Ressourcenklassen lässt sich über deren Statuscode überprüfen.

Eine andere Vorgehensweise zur Prüfung der erzeugten Bibliothek wäre die manuelle Erstellung von Tests basierend auf typischen Anwendungsfällen, wie dem Anlegen neuer Designs oder dem Erstellen neuer Produkte. Besser wäre dieses Vorgehen zusätzlich zu automatisch erzeugten Tests durchzuführen um eine möglichst hohe Abdeckung zu erreichen.

5.4. Metriken

Die Erzeugung der Client-Bibliothek erfolgt nur bei Änderungen an der [Spreadshirt-API](#), weshalb keine besonderen Anforderungen an die Dauer des Generierungs-

	Dateien	Zeilen	Zeichen
Datenklassen	192	17455	38550
Ressourcenklassen	97	3730	11079
Σ	289	21185	49629

Tabelle 5.1.: Metriken zur Menge des generierten Codes (ohne statische Klassen)

vorganges gestellt wurden. Um aus der *Spreadshirt WADL-Datei* [AG13] die Bibliothek zu generieren, benötigt der Codegenerator:

- 23.1s, falls die API-Beschreibung vom Server generiert werden muss.
- 1.25s, falls die API-Beschreibung gecached ist.

Die Werte wurden über drei Generierungsvorgänge gemittelt und auf folgenden System erfasst:

- Intel® Core™ i5-2520M CPU
- 8GB DDR3 RAM
- 128GB Samsung® 830 SSD
- Linux Kernel 3.8.0-31-generic (x64)
- Oracle® Java™ SE Runtime Environment (build 1.7.0_40-b43)

Der Großteil der Zeit wird für den Verbindungsaufbau mit dem Server und dem anschließenden Transfer der API-Beschreibung verbraucht.

Tabelle 5.1 zeigt den Umfang, der aus der API-Beschreibung mit Stand vom 17. September 2013, erzeugten Bibliothek. Das Kommando zur Ermittlung der Werte aus Tabelle 5.1 lautet:

```
find data/ resources/ -type f -name *.php -exec wc +
```

Jede erzeugte Datei erhält dabei eine Klasse. Die Gesamtheit aller in [AG13] beschriebenen Ressourcen entsprechen der Anzahl der generierten Ressourcenklassen (97). Somit ist die Anforderung erfüllt alle Methoden aus der API-Beschreibung zu generieren.



5.5. Leistungsbewertung

Im folgenden soll ein typischer Arbeitsablauf mit der Client-Bibliothek gezeigt werden. Der Client sollte zum Einstieg ein Objekt der Datenklasse **LoginDTO** anlegen ❶ und seine Login-Informationen eintragen, dies wird benötigt um über die Ressourcenklasse **Sessions** auf der gleichnamigen **API**-Ressource eine neue Sitzung (»Session«) anzulegen. Die *Response* der Session-Ressource enthält eine **URI** auf die Ressource der angelegten Sitzung. Diese **URI** enthält die *SessionID* welche durch ❷ aus der **URI** extrahiert wird.

Nun sind alle nötigen Informationen vorhanden um ein Objekt der Klasse **ApiUser** anzulegen ❸, welches die Authentifizierungsinformationen für geschützte Ressourcen der Spreadshirt-**API** enthält.

Nun kann über die Bibliotheksklassen auch auf geschützte **API**-Ressourcen zugegriffen werden, in diesem Fall werden beispielsweise die Produkte eines bestimmten Users abgefragt. Entgegen den vorherigen Aufrufen von Methoden der Ressourcenklassen wird ein Parameter-Array erzeugt ❹ um den *mediaType* der Response festzulegen. Vor dem Zugriff auf die Ressource muss noch die entsprechende Ressourcenklasse instanziiert werden ❺.




```
1 <?php
2
3 require_once("data/LoginDTO.php");
4 require_once("Static/apiUser.php");
5 require_once("resources/UsersUserId.php");
6 require_once("resources/UsersUserIdProducts.php");
7 require_once("resources/Sessions.php");
8
9 $loginDTO = new LoginDTO(); ❶
10 $loginDTO->setUsername("username");
11 $loginDTO->setPassword("password");
12 $resource = new Sessions();
13 $session = $resource->POST(null, $loginDTO); ❷
14 $sessionId = preg_replace('/.*\\/', "", $session['header']['Location']); ❸
15
16 $apiUser = new ApiUser("userId", "apiKey", "secret", $sessionId); ❹
17
18 $parameters = array("mediaType"=>"json"); ❺
19 $resource = new UsersUserIdProducts($userId); ❻
20 $products = $resource->GET($parameters, $apiUser); ❼
21
22 ?>
```

Listing 5.2: Beispiel für eine Interaktion mit der Spreadshirt-API über die generierte Client-Bibliothek (Authentifizierungsinformationen wurden anonymisiert)

5.6. Nutzbarkeit

Quellcodefiles welche Bibliotheksfunktionalitäten nutzen müssen im Wurzelverzeichnis selbiger liegen. Dies ist damit begründet das der PHP-Interpreter bei Importanweisungen mit relativen Pfaden, den aktuellen Pfad der ausgeführten Datei als Basis nimmt.

Derzeit ist die Bibliothek noch eingeschränkt nutzbar, da die De-/Serialisierung von strukturierten Typen noch nicht fehlerfrei generiert werden. Die Informationen, die nötig sind um Datenklassen verlustfrei zu serialisieren beziehungsweise deren [XML](#)-Repräsentation zu deserialisieren sind im Schema-Modell (Abschnitt [4.1.2](#)) vorhanden, der Algorithmus im Codegenerator zur Erzeugung dieser Methoden muss deshalb überarbeitet werden.



Der Aufruf des Generators erfolgt über die Kommandozeile. Als einziges Argument wird die [URL](#) zur [WADL](#)-Datei erwartet. Beispiel:

```
java api-client-libs.jar http://api.spreadshirt.net/api/v1/metaData/api.wadl
```

Der Ausgabepfad der erzeugten Bibliothek ist relativ zum Aufrufpfad des Generators: `./generated/php`.



6. Schlussbetrachtung

»Der Zauber steckt immer im Detail.«

[Fon+94]
Fontane u. a. (1994)

6.1. Ausblick

Der in dieser Arbeit dokumentierte Codegenerator bietet mehrere Ansatzpunkte für Erweiterungen oder Verbesserungen:

- Generierung eines *Fluent-Interface* Pattern. Dieses Entwurfsmuster wurde von Fowler in [Fow10] beschrieben und basiert auf der Technik des »method-chaining«, also der Hintereinanderausführung von Methoden, wobei jede Methode mit dem Resultat der vorangegangenen arbeitet.
- Die Erzeugung von Parameterobjekten welche die Parametersignatur der jeweiligen Methode repräsentieren, wird verhindert das der Nutzer unerlaubte Parameter an eine Methode übergibt. Dies ist derzeit möglich, da vom Nutzer beliebiger Inhalt die Arrays eingetragen werden kann, die zur Übermittlung der Methodenparameter dienen.
- Implementieren weiterer Sprachenmodelle, beispielsweise zur Generierung einer Java-Bibliothek.
- Erzeugen von Tests durch den Generator um die generierte Bibliothek automatisch prüfen zu können.



6.2. Fazit

Der entwickelte Codegenerator zur Erzeugung einer Client-Bibliothek für die Spreadshirt-API ...

Vorteile die sich daraus ergeben...

Probleme bei der Erstellung...



A. Implementierung

»Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.«

[Hof99, S. 152]
Hofstadter (1999)

Dieses Kapitel begründet die Entscheidung für den gewählten XML-Parser anhand eines Vergleichs verschiedener Parser-Modelle in Java.

A.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: Die »Simple API for XML« basierten sogenannten Push-Parser verarbeiten das XML-Dokument seriell und eventbasiert. Ein Event ist hierbei beispielsweise ein öffnendes oder schließendes Xml-Element.

StAX:

Auf »Streaming API for XML« basierende sogenannte Pull-Parser arbeiten ebenso wie bei SAX seriell und eventbasiert, können aber im Gegensatz dazu die Erzeugung von Events selber steuern.



TrAX:

Die »Transformation [API](#) for [XML](#)« bietet eine Schnittstelle, mit der sich [XML](#)-Dokumente durch *Extensible Stylesheet Language Transformations* (XSLT) in Java transformieren lassen.

Tabelle [A.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 Standard 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 A.1.: Übersicht über die verschiedenen [XML](#)-Parsing Konzepte in JAXP

Glossar

API: *Application Programming Interface* (deutsch: »Schnittstelle zur Anwendungsprogrammierung«) spezifiziert, wie Softwarekomponenten über diese Schnittstelle miteinander interagieren können. [III](#), [IV](#), [2–4](#), [9–11](#), [13](#), [14](#), [19–23](#), [25](#), [30](#), [39–41](#), [49](#), [51](#), [53](#), [55](#), [56](#), [60–62](#), [66](#), [A](#), [B](#), [H](#)

DSL: *Domain Specific Language* (deutsch: »Domänenspezifische Sprache«) ist eine Programmiersprache die nur auf eine bestimmte Domäne oder auch Problembereich optimiert ist. [2](#), [E](#)

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. [14](#), *siehe* [XSD](#)

HTTP:
Das *Hypertext Transfer Protocol* (HTTP) ist ein 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 ([\[Fie+99, Abstract\]](#)). [8–10](#), [17](#), [19](#), [21–24](#), [26](#), [41](#), [42](#), [49](#), [55](#), [56](#), [H](#), [J](#)

JSON:
JavaScript Object Notation, kurz JSON, ist ein leichtgewichtiges, textbasiertes und sprachunabhängiges Datenaustauschformat. Es ist von JavaScript abgeleitet und definiert eine kleine Menge von Formatierungsregeln für die transportable Darstellung (Serialisierung) von strukturierten Daten (nach [\[Cro06\]](#)). [6](#), [11–13](#), [23](#), [51](#), [G](#), [H](#), [J](#), *siehe* [XML](#)

MIME:
Multipurpose Internet Mail Extensions dienen zu Deklaration von Inhalten (Typ des Inhalts) in verschiedenen Internetprotokollen. [24](#)

OSI-Modell:
Das OSI-Modell (»Open Systems Interconnection«) ist ein Konzeptuelles

Modell, dass Funktionalitäten von Netzwerkprotokollen durch die Aufteilung in Abstraktionsschichten beschreibt und vereinheitlicht [Wik13a] .
8

REST:

Representational State Transfer (deutsch: »Gegenständlicher Zustands-transfer«) ist ein Softwarearchitekturstil für Webanwendungen, welcher von Roy Fielding in seiner Dissertation [Fie00] beschrieben wurde. Die Daten liegen dabei in eindeutig adressierbaren *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, 17, 19–22, 40, 44, F, H

RESTful:

Als *RESTful* bezeichnet man einen Webservice der den Prinzipien von REST entspricht. IV, 17, 22, 29, siehe REST

URI: Ein »Uniform Resource Identifier« (URI) ist eine kompakte Zeichenkette zur Identifizierung einer abstrakten oder physischen Ressource. ... Eine Ressource ist alles was identifizierbar ist, beispielsweise elektronische Dokumente, Bilder, Dienste und Sammlungen von Ressourcen. (eigene Übersetzung von [Ber+98]).. D, 6–8, 15, 19, 22, 26, 42, 56, 62, F, J

URL:

Der Begriff »Uniform Resource Locator« (URL) bezieht sich auf eine Teilmenge von URIs. URLs identifizieren Ressourcen über den Zugriffsmechanismus, anstelle des Namens oder anderer Attribute der Ressource. (eigene Übersetzung von [Ber+98]).. D, 6, 7, 41, 53, 64, J, siehe URI

URN:

Eine Teilmenge der URIs, die sogenannten »Uniform Resource Names« (URNs), sind global eindeutige und beständige Bezeichner für Ressourcen. Sie müssen verfügbar bleiben auch wenn die bezeichnete Ressource nicht mehr erreichbar oder vorhanden ist. ... Der Unterschied zu einer URL besteht darin, das ihr primärer Zweck in der dauerhaften Auszeichnung einer Ressource mit einem Bezeichner besteht. (eigene Übersetzung von [Ber+98]).. D, 6, 7, J, siehe URI

WADL:

Web Application Description Language ist eine maschinenlesbare Be-



schreibung einer HTTP-basierten Webanwendung. [6](#), [20](#), [23](#), [25–27](#), [29](#), [39](#), [40](#), [64](#), [F](#), [H](#), *siehe* [XML](#)

XML:

Die *Extensible Markup Language*, kurz XML, ist eine Auszeichnungssprache (»Markup Language«), die eine Menge von Regeln beschreibt um Dokumente in einem mensch- und maschinenlesbaren Format zu kodieren [[W3C08](#)] . [6](#), [10–17](#), [23](#), [25](#), [26](#), [39–42](#), [49](#), [51](#), [63](#), [A](#), [B](#), [G](#), [H](#), [J](#)

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](#), [11](#), [12](#), [14–18](#), [42](#), [F](#), [H](#), *siehe* [XML](#)

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, beispielsweise formatierende Element wie etwa Klammern werden häufig weggelassen . [32](#), [33](#), [49](#), [57](#), [F](#)

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 . [37](#)

General Purpose Language:

Eine *General Purpose Language* bezeichnet eine Programmiersprache welche für den Einsatz in den verschiedensten Anwendungsbereichen verwendet kann, im Gegensatz zu einer [DSL](#), welche nur auf einen speziellen Bereich beschränkt ist . [36](#)

Abbildungsverzeichnis

1.1. Aufbau des Generatorsystems	2
1.2. Spreadshirt Logo	3
2.1. Diagramm zur Veranschaulichung der Teilmengenbeziehung zwischen den Adressierungsarten, Quelle [Wik13b]	7
2.2. vordefinierte XSD Datentypen nach [W3C12] Kapitel 3	18
2.3. Beispiel-URI, um den Artikel 42 aus dem Warenkorb 84 anzusprechen	22
2.4. Struktur einer WADL-Datei, nach Kapitel 2 [Had06]	27
3.1. Beispiel Abstract Syntax Tree für den rekursiven euklidischen Algorithmus	33
4.1. UML Klassendiagramm des REST-Modells	40
4.2. UML Klassendiagramm des Schemadatenmodells	43
4.3. Datentyp Point mit Gegenüberstellung im Schemamodell	44
4.4. UML Klassendiagramm des Zielsprachenmodells	45
4.5. Beispiel für den Aufbau einer »Expression« im Sprachenmodell	47
4.6. Ablaufdiagramm des Generators	50
5.1. Darstellung von BatchDTO aus Listing 5.1 im Sprachenmodell [Klasse, Zeichenkette]	58

Tabellenverzeichnis

2.1. JSON Datentypen	13
2.2. Beispiele für Konnektoren nach [Fie00]	21
3.1. Generatoren Klassifikation nach Generierungsmenge	30
5.1. Metriken zur Menge des generierten Codes (ohne statische Klassen)	61
A.1. Übersicht über die verschiedenen XML-Parsing Konzepte in JAXP	B

Listings

2.1.	HTTP-Header von GET Request auf Spreadshirt-API Ressource <code>http://api.spreadshirt.net/api/v1/locales</code>	9
2.2.	HTTP-Header von GET Response aus Spreadshirt-API Ressource <code>http://api.spreadshirt.net/api/v1/locales</code>	9
2.3.	HTTP-Body der Response aus der GET-Methode auf der Spreadshirt-API-Ressource <code>http://api.spreadshirt.net/api/v1/locales</code>	10
2.4.	Die gekürzte Antwort der API-Ressource <code>users/userid/designs/designID</code> als Beispiel für eine XML-Datei	11
2.5.	Die gekürzte Antwort der API-Ressource <code>users/userid/designs/designID</code> als Beispiel für eine JSON-Datei	13
2.6.	Beginn der XSD-Datei für die Spreadshirt-API	14
2.7.	Beispiel für einen SimpleType namens »unit« der Spreadshirt-API	15
2.8.	Beispiel für einen Listentyp definiert durch einen SimpleType	15
2.9.	Beispielinstanz für Typ aus Listing 2.8	15
2.10.	Beispiel für eine Schemabeschreibung mit XSD anhand des »abstractList«-Typs der Spreadshirt-API	16
2.11.	Beispiel zu Metainformationen für REST-Repräsentation aus WADL-Datei der Spreadshirt-API	20
2.12.	Beispielaufbau einer WADL-Datei anhand der Spreadshirt-API Beschreibung	25
3.1.	Aufbau eines Ausdrucks einer imperativen Programmiersprache als EBNF	34
3.2.	Durch den Generator erzeugte BatchDTO Datenklasse der Spreadshirt-API als Beispiel für eine PHP-Datei	38
4.1.	Point-Klasse als (gekürztes) Beispiel für eine generierte Datenklasse	51
4.2.	Klasse zur Ressource <code>users/userId/products</code> als Beispiel für eine Ressourcenklasse	54
4.3.	Aufbau des Spreadshirt Authentication Header	56
5.1.	Ausschnitt der generierten Datenklasse BatchDTO	59



5.2. Beispiel für eine Interaktion mit der Spreadshirt-API über die generierte Client-Bibliothek (Authentifizierungsinformationen wurden anonymisiert)	63
--	----



Definitionsverzeichnis

1.	Definition (URI)	6
2.	Definition (URL)	7
3.	Definition (URN)	7
4.	Definition (HTTP)	8
5.	Definition (XML)	11
6.	Definition (JSON)	12
7.	Definition (Codegenerator)	28
8.	Definition (Datenmodell)	32
9.	Definition (Abstract Syntax Tree – Aho u. a.)	33
10.	Definition (Typsystem)	36

Literaturverzeichnis

- [AG13] sprd.net AG. *Spreadshirt WADL-Datei*. 30. Sep. 2013. URL: <http://api.spreadshirt.net/api/v1/metaData/api.wadl>.
- [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.
- [BB13] Martin Breest und Karsten Breit. *Spreadshirt-API Security*. 14. Sep. 2013. URL: <http://developer.spreadshirt.net/display/API/Security>.
- [BD87] G.E.P. Box und N.R. Draper. *Empirical model-building and response surfaces*. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley, 1987. ISBN: 9780471810339.
- [Ber+98] Tim Berners-Lee u. a. *Uniform Resource Identifiers (URI): Generic Syntax*. Aug. 1998. URL: <http://www.ietf.org/rfc/rfc2396.txt> (besucht am 30. 09. 2013).
- [CE00] K. Czarnecki und U. Eisenecker. *Generative programming: methods, tools, and applications*. Addison Wesley, 2000. ISBN: 9780201309775.
- [Cro06] Douglas Crockford. *RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)*. IETF. Juli 2006. URL: <http://tools.ietf.org/html/rfc4627>.
- [Dij07] Edsger W. Dijkstra. »Forum«. In: *Commun. ACM* 50.6 (Juni 2007). Hrsg. von Diane Crawford, S. 7–9.
- [ES13] Karl Eilebrecht und Gernot Starke. *Patterns kompakt*. Springer Vieweg, 2013. ISBN: 978-3-642-34717-7.
- [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«. In: (1999). URL: <http://www.ietf.org/rfc/rfc2616.txt>.
- [Fon+94] T. Fontane u. a. *Briefe an Georg Friedlaender*. Insel Taschenbuch. Insel, 1994.



- [Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 9780131392809.
- [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.
- [HH73] R.W. Hamming und R.W. Hamming. *Numerical Methods for Scientists and Engineers*. Dover Books on Mathematics Series. Dover, 1973. ISBN: 9780486652412.
- [Hof99] D.R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books. Basic Books, 1999. ISBN: 9780465026562.
- [Mar13] Christian Dietrich Markus Voelter Sebastian Benz. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0. URL: <http://www.dslbook.org>.
- [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](https://doi.org/10.1145/1111627.1111631). URL: <http://doi.acm.org/10.1145/1111627.1111631>.
- [Ött13] Tim Ötting. *www.soundlikecotton.com*. 20. Juni 2013. URL: <http://www.soundlikecotton.com/>.
- [Pas13] Kathrin Passig. *www.zufallsshirt.de*. 20. Juli 2013. URL: <http://zufallsshirt.de/>.
- [Per82] Alan J. Perlis. »Special Feature: Epigrams on programming«. In: *SIGPLAN Not.* 17.9 (Sep. 1982), S. 7–13. ISSN: 0362-1340.



- [PK12] Franz Puntigam und Andreas Krall. *Objektorientierte Programmier-techniken*. 2012.
- [Til09] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. Heidelberg: dpunkt, 2009. ISBN: 978-3-89864-583-6.
- [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«. In: (Nov. 2001). URL: <http://tools.ietf.org/html/rfc3198>.
- [Wik13a] Wikipedia. *OSI model*. 30. Sep. 2013. URL: http://en.wikipedia.org/w/index.php?title=OSI_model&oldid=574771928.
- [Wik13b] Wikipedia. *Uniform resource name*. 30. Sep. 2013. URL: http://en.wikipedia.org/w/index.php?title=Uniform_resource_name&oldid=563710451.
- [Wik13c] Wikipedia. *XML — Wikipedia, The Free Encyclopedia*. 2013. URL: <http://en.wikipedia.org/w/index.php?title=XML&oldid=561587115> (besucht am 26.06.2013).
- [WO08] G. Wilson und A. Oram. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, 2008. ISBN: 9780596554675.

BIB_TE_X Eintrag

```
@phdthesis{AndreasLinz2013,  
  type = {Bachelorarbeit}  
  author = {Linz, Andreas},  
  year = {2013},  
  month = {10},  
  day = {1},  
  timestamp = {2013101},  
  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  
}
```