Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Konzept und Implementierung einer DSL für die Generierung von RESTful-Webservices

Bachelorarbeit

im Studiengang Informatik

von

Thorsten Kerber

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Matthew Smith
Betreuer: M. Sc. Tristan Wehrmaker

Hannover, 13. August 2011

Zusammenfassung

Die serverseitige Implementierung eines RESTful Webservices gestaltet sich oft schwer, da für die Entwicklung mit REST eine umfangreiche Architektur realisiert werden muss. Aus diesem Grund wäre es vorteilhaft diesen Teil generieren zu lassen, um so die Implementierung zu vereinfachen.

In dieser Arbeit wird eine domänenspezifische Sprache für die Generierung von RESTful Webservices entwickelt. Durch diese ist es möglich alle erforderlichen Informationen für eine REST Anwendung einzugeben. Mit dem speziell entwickelten Generator werden diese Informationen verarbeitet und der serverseitige Code generiert.

Abstract

The server-side implementation of a RESTful Web services often proves difficult because the development of REST requires a comprehensive architecture. For this reason, it would be profitable to generate this part automatically to make the implementation easier.

In this work, a domain-specific language for the generation of RESTful web services is developed. By this it is possible to enter all necessary information for a REST application. The specially designed generator processes this information and generates the server-side code.

Danksagung

Ich danke allen, die mich beim Erstellen dieser Arbeit unterstützt haben.
Besonderer Dank gebührt Herrn Tristan Wehrmaker, für die hervorragende Betreuung.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Der Architekturstil Representational State Transfer (REST) gerät immer mehr in den Fokus in der Entwicklung von Webservices, da er sich zu einer Alternative des Webservices entwickelt hat. REST beruht auf der Arbeit mit identifizierbaren Ressourcen und ihren unterschiedlichen Repräsentationsformaten. Zudem bietet REST eine Schnittstelle mit einheitlichen Operationen, wodurch ein Client zu jeder Zeit weiß wie ein Service genutzt wird.

Das Verständnis und die Verwendung von REST Services gelten im Allgemeinen als einfach. Trotzdem setzen sie eine umfangreiche Architektur für die serverseitige Implementierung voraus, um allen Ansprüchen eines RESTful Webservices zu genügen.

Aus diesem Grund wäre es wünschenswert die Anforderungen des REST-Ansatzes durch eine domänenspezifische Sprache (DSL) auszudrücken. Mit dieser und anderen Mitteln der modellgetriebenen Softwareentwicklung kann schließlich der serverseitige Code generiert werden.

1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist die Entwicklung einer domänenspezifischen Sprache (DSL) für die Domäne der RESTful Services und die Entwicklung eines Generators, der den serverseitigen Code dieses Service generiert.

Mittels der entwickelten DSL werden alle benötigten Daten für eine Anwendung angegeben. Der Generator verarbeitet diese und generiert schließlich den serverseitigen Code eines RESTful Webservices. Dies führt zur einer Erleichterung bei der Implementierung, da der Generator den Bau der umfangreichen Architektur von REST übernimmt und somit alle benötigten Klassen generiert.

1.3 Struktur dieser Arbeit

Diese Arbeit besteht aus insgesamt sechs Kapiteln. Nachdem in Kapitel 1 die Problemstellung und Zielsetzung dieser Arbeit vorgestellt wurde, werden in Kapitel 2 die Grundlagen zur Bearbeitung des Themas erläutert. Abschnitt 2.1.1 und 2.1.3 sollen dabei den Unterschied des traditionellen Webservice gegenüber des RESTful Webservice deutlich machen. Über die

häufig verwendeten Datenaustauschformate wird in Abschnitt 2.2 gesprochen und am Ende des Kapitels auf die Modellgetriebene Softwareentwicklung eingegangen.

In Kapitel 3 geht es um den Kern dieser Arbeit, das Konzept zur Entwicklung einer domänenspezifischen Sprache. Dabei wird unter anderem auf die Analyse der Referenzimplementierung eingegangen und ein Einblick in die Struktur der DSL gegeben. Im letzten Abschnitt wird anhand eines Beispiels gezeigt, wie die entwickelte DSL zu nutzen ist.

Kapitel 4 behandelt den praktischen Teil dieser Arbeit, die Implementierung. Zuerst werden die wichtigsten Punkte zur Nutzung von Xtext und Jersey abgehandelt und schließlich wird über die Vorgehensweise bei der Implementierung gesprochen.

In Kapitel 5 werden Beispiele gezeigt, wie das in den vorigen Kapiteln vorgestellte Konzept und deren Implementierung in der Praxis aussieht.

Den Schluss dieser Arbeit bildet Kapitel 6 mit einer Zusammenfassung, einem Fazit und einem Ausblick.

2. Grundlagen

Nachdem die Aufgabe und die Motivation dieser Arbeit erläutert wurden, wird in diesem Kapitel auf die verwendeten Techniken und Grundlagen eingegangen. Zunächst wird erklärt, was ein Webservice ist und wo dieser eingesetzt wird. Danach folgt die Beschreibung der Kommunikation in HTTP und die Grundlagen des Architekturstils REST. Der darauffolgende Abschnitt, über die häufig verwendeten Datenaustauschformate, soll einen Einblick über die Struktur der gängigen Repräsentationsformate bieten. Am Ende dieses Kapitels geht es um die modellgetriebene Softwareentwicklung.

2.1 Webservices

Ein Webservice ist ein Dienst, der über ein Netzwerk erreichbar ist und zudem über zugängliche Schnittstellen zu seinen Anwendungsfunktionen verfügt. Diese können mit Standardprotokollen wie z. B. HTTP aufgerufen werden. Webservices werden sowohl in Unternehmen für die Realisierung von Geschäftsprozessen, als auch in der täglichen Nutzung des Internets verwendet. [STK02]

2.1.1 Traditionelle SOAP-basierte Webservices

Das W3C definiert einen Webservice wie folgt:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.* [HB02]

Der Zugriff auf Webservices, nach obiger Definition, wird durch das Protokoll SOAP gesteuert. Auf der Basis von XML wird ein Protokoll auf der Grundlage eines entfernten Prozeduraufrufs, dem Remote Procedure Call (RPC), realisiert.

Über die Schnittstellen des Webservices werden die einzelnen Dienste gekapselt. Die Definition der Schnittstellen erfolgt durch die WSDL (Web Services Description Language).

Webservices sind sich selbstbeschreibende und eigenständig agierende Software-Komponenten, die sich über UDDI (Universal Description, Discovery and Integration)

untereinander aufrufen können. UDDI dient dabei als ein übergeordneter Verzeichnisdienst zur Veröffentlichung von Webservices. [STK02]

2.1.2 Kommunikation in HTTP

Das Hypertext Transfer Protocol, kurz HTTP, ist ein Anfrage- und Antwort-Protokoll, in dem Client und Server über HTTP-Nachrichten kommunizieren.

Ein Client sendet eine Anfrage (HTTP-Request) an den Server. Dieser arbeitet diese ab und schickt eine Antwort (HTTP-Response) mit den geforderten Informationen zurück. [F99]

Folgendes Beispiel zeigt den Aufbau einer HTTP-Request-Nachricht:

```
GET /customers/1234 HTTP/1.1
Host: www.example.org
Accept: application/xml, text/html
Accept-Language: en
```

Listing 1: HTTP-Request

GET ist eine HTTP-Methode, die bestimmt, wie der Server den Request verarbeitet. Der Pfad `/customers/1234` gibt dabei die Adresse der Ressource auf dem Server unter Host an, auf den die Methode angewendet werden soll.

Im Accept-Feld wird dem Server mitgeteilt, welche Formate man als Antwort bevorzugt. In diesem Fall sind es XML und HTML, wobei XML laut Reihenfolge die Priorität hat.

Im sogenannten Body der Anfrage können zusätzlich Informationen als Parameter für den Request enthalten sein. In diesem Fall ist der Body leer, da für die GET Befehl keine weiteren Parameter benötigt werden. In Abschnitt 2.1.2 werden die weiteren HTTP-Methoden genauer erläutert.

Eine entsprechende HTTP-Response liegt im folgenden Text vor:

```
HTTP/1.1 200 OK
Date : Thu, 14 Apr 2011 12:44:48 GMT
Content-Type: application/xml

<customer>
<id>1234</id>
…
</customer>
```

Listing 2: HTTP-Response

Der Statuscode 200 OK gibt an, dass eine Anfrage erfolgreich bearbeitet wurde.
Andere wichtige Response-Codes sind z. B. 201 Created, 301 Moved Permanently, 400 Bad Request, 404 Not Found und 500 Internal Server Error.

Mithilfe des Content-Type-Header wird angegeben, welcher Repräsentationstyp gewählt wurde. In diesem Fall ist es XML. Dahinter folgt der Body mit den geforderten Informationen des Requests.

An dieser Stelle sei zu betonen, dass es eine Vielzahl anderer Header und Response-Codes gibt, die hier nicht weiter erwähnt werden. In der Spezifikation können diese nachgeschlagen werden. [F99]

2.1.3 Representational State Transfer (REST)

Representational **S**tate **T**ransfer (REST) ist ein von Roy Fielding definierter Architekturstil für verteilte Hypermedia Systeme, der im Bereich der Webservices zunehmend an Bedeutung gewinnt. REST orientiert sich hierbei an den grundlegenden Prinzipien des World Wide Web. [S11]

Zu diesen Prinzipien gehören: [F00]

- identifizierbare Ressourcen,
- unterschiedliche Repräsentationen,
- einheitliche Schnittstellen,
- Zustandslosigkeit,
- Verwendung von Hypermedia

Im Folgenden werden diese Prinzipien genauer erläutert.

Identifizierbare Ressourcen:

Die Ressource gehört zu den wichtigsten Teilen in REST.

> *Eine Ressource, alles, was aus Sicht des Architekten verdient, eindeutig identifiziert zu werden.* [T09a]

Sie kann in Form eines HTML-Dokuments, eines Bildes, eines temporären Dienstes wie z. B. eine Wettervorhersage sein. Eine Ressource muss jedoch nicht immer ein einzelnes Objekt repräsentieren und kann somit auch als eine Sammlung von anderen Elementen betrachtet werden. [T09b]

Um Ressourcen zu erreichen und Verbindungen zwischen ihnen zu erstellen, wird eine spezifische Adresse benötigt, der Uniform Resource Identifier (URI). URIs sind der Namensgebungs- und Adressierungsmechanismus des Webs und garantieren, dass eine Ressource eindeutig identifiziert wird. [BFM05]

Für eine bessere Strukturierung hat Tilkov die Ressource in unterschiedliche Ressourcenkategorien eingeteilt. An dieser Stelle werden nur die Kategorien beschrieben, die auch in der Arbeit von Bedeutung sind. [T09b]

- *Primärressourcen* sind z. B. ein einzelner Kunde oder Konto in einem Banksystem. Sie können zum Beispiel die persistenten Entitäten eines klassischen Anwendungsentwurfs repräsentieren.

- *Listenressourcen* sind eine zugehörige Liste zu den Primärressourcen. Auch sie müssen eindeutig identifiziert werden, da sie ebenfalls eine Ressource sind. Im Beispiel des Banksystems wäre eine Listenressource eine Liste sämtlicher Konten eines Kunden.

- *Subressourcen* sind Primärressourcen, die eine Verbindung zu einer anderen übergeordneten Ressource haben. Ein Beispiel dafür ist das Geld auf einem einzelnen Konto eines Kunden.

- *Filterressourcen* sind eine Art Listenressource, die jedoch nicht sämtliche Primärressourcen einer Kategorie enthalten müssen. Eine Liste mit allen Kunden mit den Anfangsbuchstaben "A", wäre eine Filterressource.

Unterschiedliche Repräsentationen:

Repräsentationen sind Darstellungen einer Ressource in einem definierten Format. Jede Ressource kann mehrere Repräsentationen haben. Das Aushandeln des Formats, welches vom Server geliefert wird, erfolgt durch die Anfrage des Client in einem Accept-Header. Dieses dynamische Aushandeln wird als Content Negotiation bezeichnet. Dabei gilt, je allgemeingültiger das Format der Repräsentation, desto mehr Clients können es benutzen.

Der Accept-Header gibt an, welche Art von Repräsentation er als Antwort akzeptiert. Im vorigen Abschnitt 2.1.1 wurde bereits genauer erläutert, wie eine solche Anfrage des Clients bzw. Antwort des Servers gestaltet wird. [T09b]

Einheitliche Schnittstelle:

Jede Ressource besitzt eine einheitliche Schnittstelle, in der sie die folgenden Standardmethoden bereitstellt: [RR07]

GET fragt eine Repräsentation einer Ressource ab.

POST	fügt eine neue Ressource zu einer Listenressource hinzu.
PUT	eine Ressource wird geändert, wenn ihre URI bekannt ist. Falls sie noch nicht existiert, wird eine neue Ressource erstellt.
DELETE	löscht eine angegebene Ressource.
HEAD	fragt Metadaten zu einer Ressource ab. Entspricht einer GET nur ohne Response-Body.
OPTIONS	gibt die Methoden und Eigenschaften zurück, die unterstützt werden.

In der HTTP-Spezifikation werden den Methoden zusätzlich noch zwei weitere Eigenschaften zugeordnet. Zum einen die Eigenschaft Sicherheit. Sie bedeutet, dass egal wie oft eine Anfrage gesendet wurde, keine Veränderungen am Server hervorrufen kann. Für den Server wäre es so, als wäre die Anfrage nur einmal oder keinmal getätigt worden.

Die zweite Eigenschaft ist die Idempotenz. Sie besagt, dass das Senden von Informationen durch mehrmalige Aufrufe einer Methode, nur beim ersten Mal eine Änderung einer Ressource bewirkt. Die restlichen Aufrufe bewirken somit keine Zustandsveränderung.

Die am häufigsten verwendete Methode ist GET, da sie z. B. beim Aufrufen von Webseiten im Browser gefordert ist. In der HTTP-Spezifikation ist sie als sicher definiert, weil auch mehrmaliges Ausführen einer GET-Methode keine Zustandsänderung am Server bewirkt.
Dies führt dazu, dass die Methode GET auch idempotent ist. D.h., dass ein mehrmaliges Aufrufen den gleichen Effekt hat wie ein einmaliges.

PUT aktualisiert eine Ressource, falls sie vorhanden ist oder legt eine Neue an. Da sie vorhandene Ressourcen aktualisiert und somit nicht jedes Mal neue Instanzen derselben Ressource definiert, ist PUT idempotent.

DELETE ist ebenfalls idempotent, da ein mehrmaliges Ausführen nur beim ersten Mal eine Ressource entfernt und die weiteren Aufrufe keine weiteren Auswirkungen haben. Die Operation POST ist weder sicher noch idempotent, was dazu führt, dass sie auf zwei Arten genutzt wird. Zum einen kann sie neue Ressourcen anlegen und zum anderen wird sie benutzt, wenn keine der anderen Standardmethoden für eine Aktion passt.

HEAD und OPTIONS sind sicher und idempotent, da sie wie GET keine Veränderungen durch ihren Aufruf bewirken und nur Daten liefern. [T09b], [F99]

Zustandslosigkeit:

Beim Architekturstil REST ist die Kommunikation zwischen Client und Server zustandslos. Jede Anfrage ist völlig unabhängig zu den Vorigen. Ein Client muss alle notwendigen Informationen an den Server senden, damit dieser die Anforderung versteht, ohne jeglichen Kontext zu haben. Die Sitzung wird somit nur clientseitig gehalten. [T09b]
Dies führt zur Verbesserung der folgenden Punkte: [T09b]

- Sichtbarkeit, da ein Überwachungssystem nicht kontrollieren muss, ob alle zusammengehörenden Teilanfragen vorhanden sind,

- Zuverlässigkeit, da es die Wiederherstellung von Teilausfällen erleichtert und

- Skalierbarkeit, da eingehende Anfragen auf unterschiedliche Maschinen verteilt werden können. Jeder Request ist eine geschlossene Anfrage.

Verwendung von Hypermedia:

Hypermedia beruht auf dem Konzept von Verknüpfungen (Links), welche vor allem aus HTML bekannt sind. Das folgende Beispiel soll zeigen, wie Links in XML eingebunden werden können und die Vorstellung erleichtern, wie eine Applikation diesen Links folgen kann, um an weitere Informationen zu kommen: [T09b]

```
<order href='http://example.org/customers/1234'>
     <amount>23</amount>
     <product ref='http://example.com/products/4554' />
     <customer ref='http://example.org/customers/1234' />
     <link rel='cancel'  ref='./cancellations' />
</order>
```

Listing 3: Beispiel für die Verwendung von Links

Würde man dem Link des product-Tag folgen, könnte man z. B. eine genauere Beschreibung des Produkts erhalten.

Der Vorteil des Hypermedia-Ansatzes ist es, dass Verknüpfungen anwendungsübergreifend funktionieren, was ausschließlich am globalen Namensschema liegt. Dies macht es möglich alle Ressourcen, aus denen das Web besteht, miteinander zu verknüpfen.

Ein weiterer Vorteil von Hypermedia ist die Möglichkeit den Applikationszustand zu steuern. Dies nennt Fielding *Hypermedia as the Engine of Application State*. [FF00]

Der Server kann dem Client über Links mitteilen, welche Aktionen als nächstes ausführbar sind. Ein Beispiel: Der Client kann seine Bestellung stornieren und somit den Status seiner

Bestellung ändern. Außerdem kann der Server den Zustand der Bestellung ändern, indem er das Tag link (s. Zeile 5) entfernt und somit keine Stornierung anbietet.

Die Abschnitte 2.1.1 und 2.1.3 sollen folgende Unterschiede deutlich machen:

- Webservices unterstützen das Konzept der verteilten Funktionsaufrufe zwischen Systemen, wobei REST sein Konzept auf den Zugriff und die Manipulation von Ressourcen in Hypermedia-Systemen auslegt.

- Webservices besitzen unterschiedliche Schnittstellen, die durch eine eigene Sprache definiert sind. REST besitzt hingegen eine einheitliche Schnittstelle, die immer gleich implementiert werden kann.
- Traditionelle Webservices mit SOAP verwenden HTTP nur als Transport-Protokoll, in dem sie Nachrichten immer mit der Methode POST verschicken, und nicht als Anwendungsprotokoll, was es eigentlich ist.

- der Informationsaustausch basiert bei Webservices auf XML. Hingegen ist REST nicht von bestimmten Mediaformate abhängig.

2.2 Häufig verwendete Datenaustauschformate

2.2.1 Extensible Markup Language (XML)

Die Extensible Markup Language, kurz XML, wurde von einer XML-Arbeitsgruppe des W3C entwickelt, welche ursprünglich als das SGML Editorial Review Board bekannt war. XML ist eine Auszeichnungssprache zur Erstellung von hierarchisch strukturierten Daten in Textformat und desweiteren erweiterbar, plattformunabhängig und maschinenlesbar. Durch strukturelle und inhaltliche Einschränkungen mit Schemasprachen wie DTD und XML-Schema kann zudem eine anwendungsspezifische Sprache definiert werden. [BPSMY08], [J09]

Ein XML-Dokument besteht aus einem Prolog, verschiedenen Elementen und optional aus einem Epilog. Ein Element kann Kindelemente oder Geschwisterelemente haben. Jedes Element kann wiederum Attribute enthalten.

Das XML-Dokument zeigt, dass alle Elemente durch Tags dargestellt und Attribute in das Tag eines Elements geschrieben werden:

```
<?xml version="1.0"?>
<cdCollection>
     <cd title="Schiffsverkehr" type="album">
```

```
          <song artist="Herbert Grönemeyer"
               title="Schiffsverkehr" />
               ...
     </cd>
     <cd title="Good News" type="album">
          <song artist="Lena" title="Taken by a Stranger" />
                    ...
          </cd>
</cdCollection>
```

Listing 4: Aufbau eines XML-Dokuments

Das Beispiel zeigt `cdCollection` als Wurzelelement. Dieses enthält zwei Elemente `cd`. `cd` hat die Attribute `title` und `type`. Zudem jeweils ein Kindelement `song`, welches mehrmals vorkommen kann.

Wohlgeformtheit:

Ein XML- Dokument ist wohlgeformt, wenn es die Regeln von XML korrekt einhält. Regeln für syntaktische Korrektheit: [BPSMY08]

- es gibt nur ein Wurzelelement
- jedes Element hat ein Anfangs- und ein dazugehöriges Endtag
- die einzelnen Tags müssen korrekt geschachtelt werden, z. B.:

```
     <cdCollection>
          <cd title="Schiffsverkehr" type="album">
          </cd>
     </cdCollection>
```

Listing 5: Korrekte Verschachtelung in XML

falsch wäre:

```
     <cdCollection>
          <cd title="Schiffsverkehr" type="album">
          </cdCollection>
     </cd>
```

Listing 6: Falsche Verschachtelung in XML

- Attribute eines Elements haben eine einzigartige Bezeichnung
- Elemente unterliegen einer Namensrestriktion

Gültigkeit:

Ein XML- Dokument ist gültig, wenn es

1. wohlgeformt ist,

2. es sich an die Semantik einer Grammatik aus einem DTD oder XML-Schema hält. [BPSMY08]

2.2.2 JavaScript Object Notation (JSON)

JSON ist wie XML ein Datenaustauschformat, das für Menschen einfach zu lesen und zu schreiben ist. Es basiert auf einer Teilmenge der Programmiersprache JavaScript, wird aber in fast jeder Programmiersprache unterstützt. JSON bedient sich der Kurzschreibweisen für Objekte und Arrays. Dadurch ermöglicht es eine sehr platzsparende Speicherung und Übertragung strukturierter Daten. [J09]

Folgendes Beispiel macht die einfache Syntax von JSON deutlich:

```
{ cdCollection: {
           cd: {
                title: "Schiffsverkehr" ,
                type: "album",
                song: {
                     artist: "Herbert Grönemeyer",
                     title: "Schiffsverkehr"
                }
                ...
           }
           cd: {
                title: "Good News",
                type: "album",
                song:{
                     artist: "Lena",
                     title: "Taken by a Stranger"
                }
                ...
           }
           ...
     }
}
```

Listing 7: Aufbau eines JSON-Dokuments

Beim Vergleich der beiden Austauschformate XML und JSON wird deutlich, dass JSON durch die Vernachlässigung der Tags ein geringeres Datenvolumen bei der Übertragung von Nachrichten hat. Außerdem kann JSON direkt in JavaScript verarbeitet werden.

Im Vergleich zu XML gibt es in JSON aber keine Möglichkeit, Namensräume zu definieren und ein Dokument anhand seines Schemas zu validieren. Desweiteren ist XML eine vollwertige Markup-Sprache und bietet somit deutlich mehr Möglichkeiten.

Als Fazit lässt sich sagen, dass JSON für Anwendungen, die sehr kurze Nachrichten übertragen oder die mit einem Webservice auf einem fremden Server kommunizieren möchten, die richtige Wahl ist. In anderen Fällen ist XML eine gleichwertige Alternative.

2.3 Modellgetriebene Softwareentwicklung

Modellgetriebene Softwareentwicklung (Model Driven Software Development, kurz MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [SV05]

In diesem Abschnitt werden die Ziele und Grundlagen, sowie die Verwendung der Technik von MDSD erläutert.

2.3.1 Ziele und Grundlagen

Im Folgenden werden die Ziele und Grundlagen von MDSD beschrieben: [SV05]

- Verbesserung der Softwarequalität, da Systeme eine einheitliche, getestete und dokumentierte Architektur umsetzen
- Wiederverwendung von Architekturen, Modellierungssprachen und Generatoren zur Herstellung mehrerer Softwaresysteme
- Steigerung der Entwicklungseffizienz durch Generierung großer Anteile

Bei MDSD geht es um das Prinzip Code für eine Anwendung zu generieren, die ähnliche bis gleiche Eigenschaften und Anforderungen einer bereits bekannten Domäne enthält. So wird der sich wiederholende Code generiert, anstatt ihn jedes Mal manuell zu implementieren.

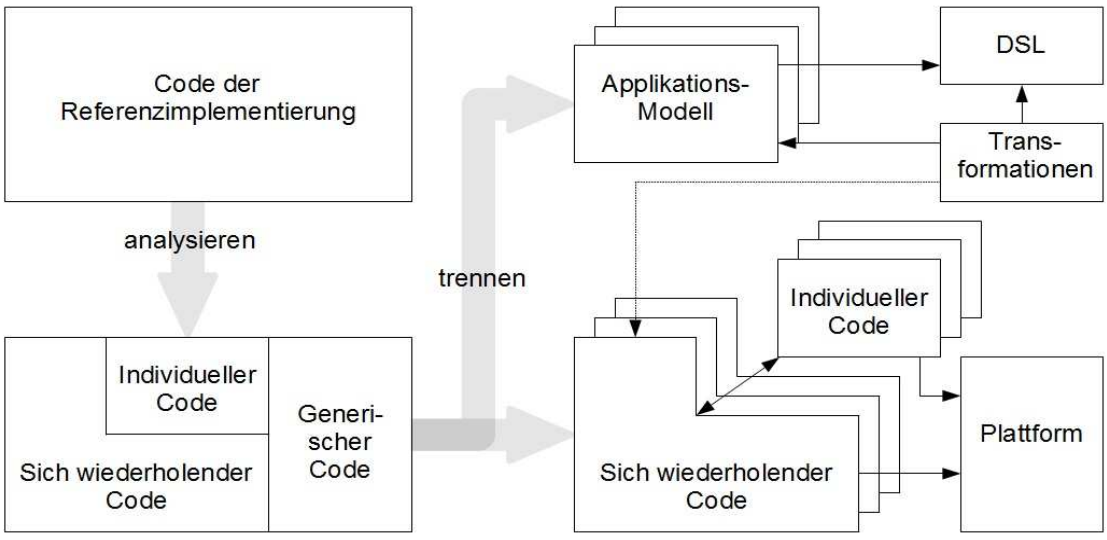

Abbildung 1: Grundidee modellgetriebener Softwareentwicklung nach [SV05] aus [H10]

Die Abbildung 1 zeigt die Herangehensweise bei der modellgetriebenen Softwareentwicklung. Zunächst wird eine Referenzimplementierung analysiert. Ihr Code wird hierbei in drei Teile geteilt. Der für alle Anwendungen identische Code ist der generische Code. Der Teil, der bei allen Domänen ähnlich, aber nicht gleich ist, ist der sich wiederholende Code. Zuletzt gibt es noch den individuellen Code, der bei allen Programmen unterschiedlich ist. Er muss von einem Entwickler manuell implementiert werden. Diese Codeteile werden voneinander getrennt betrachtet. Wobei der generische Code vernachlässigt werden kann, da dieser später generiert wird.

Der sich wiederholende Code soll aus formalen Modellen gewonnen werden. Um dies zu erreichen, benötigt man eine Transformation, die aus einer domänenspezifischen Sprache (Domain Specific Language, DSL) hergeleitet wird. Genauer gesagt ist eine DSL eine Art Programmiersprache für eine bestimmte Domäne. Sie besteht aus einem Metamodell mit dazugehöriger abstrakter Syntax und statischer Semantik sowie der konkreten Syntax. [SV05]

2.3.2 Implementierung

In der Praxis hat es sich bewährt generierten und nicht-generierten Code immer in separaten Dateien und Verzeichnissen zu halten, da der generierte Code ein Wegwerfprodukt ist und nicht modifiziert wird. In einer Zielarchitektur wird entschieden in wie fern der generierte und nicht-generierte Code kombiniert werden. [SV05]

Die einfachste Methode der Implementierung ist die Erstellung von geschützten Bereichen im generierten Code, in die der Entwickler den manuellen Code einfügt. Weitere Generatorläufe können damit keine Änderung am manuellen Code vornehmen.

Für eine solche Integration empfehlen sich die Fähigkeiten von UML-Werkzeugen. Sie bieten dabei die Möglichkeit aus Modelldaten Klassenrümpfe zu generieren, in die dann der manuelle Code implementiert werden kann.

Dieses Vorgehen hat jedoch folgende Nachteile: [SV05]

- Der Generator ist komplexer, weil er die Verwaltung, Erkennung und Erhaltung der geschützten Bereiche bewältigen muss
- Es ist nicht immer möglich den Inhalt der geschützten Bereiche zu erhalten. In der Praxis geht immer wieder Code verloren
- Die Trennung von generiertem und manuell erstelltem Code wird aufgeweicht, da beide in derselben Datei/Klasse stehen

Der letzte Punkt ist am problematischsten, da im generierten Code gearbeitet wird, was sich negativ auf das Verständnis und die Versionsverwaltung auswirkt.

3. Konzeption

In den vorigen Kapiteln wurden die Grundlagen erläutert, die für die Konzeption einer DSL für die Domäne der RESTful Webservices wissenswert sind. In diesem Kapitel wird die Analyse der Referenzimplementierung ConTexter beschrieben und die entstandenen Ergebnisse herausgearbeitet. Am Ende des Kapitels wird auf die Sprachmittel der entwickelten DSL eingegangen und anschließend ein Beispiel gezeigt.

3.1 Analyse und Ergebnisse

Für einen besseren Überblick von der Referenzimplementierung wurde zunächst ein Modell angefertigt.

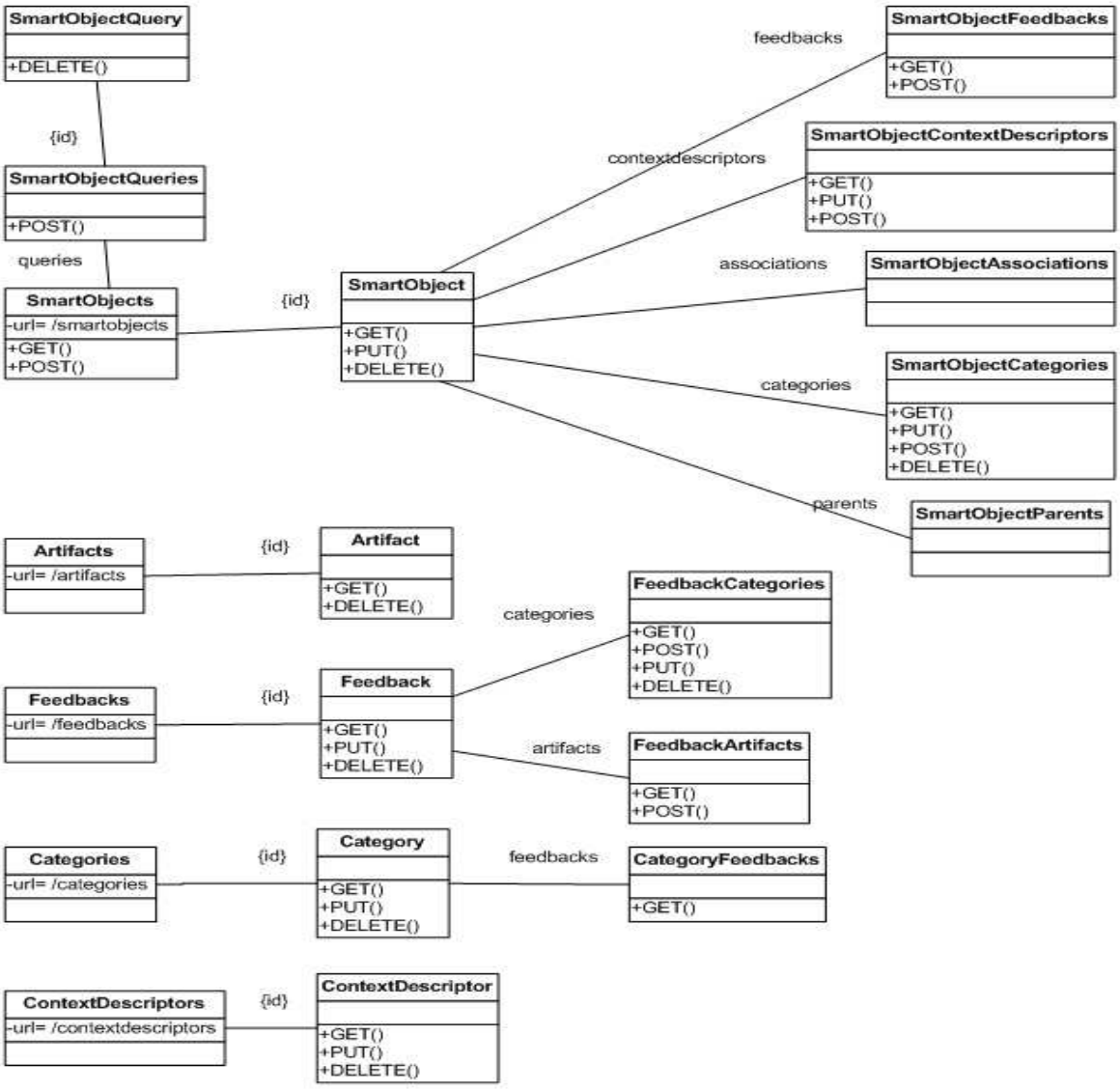

Abbildung 2: Modellierung der Referenzimplementierung

Das Modell zeigt die einzelnen Klassen des ConTexter. Die Verbindungen zwischen den Klassen stehen für den Pfad um zu einer anderen Klasse zu navigieren. Desweiteren zeigt das Modell die einzelnen Methoden, die von einer Klasse unterstützt werden.

Zu Beginn wurden der Aufbau und die Struktur der einzelnen Klassen untersucht. Durch das Einarbeiten in die REST-Domäne und der gegliederten Packages des ConTexter wurde deutlich, dass es sich bei den Klassen um sogenannte *Ressourcen* und die dazugehörigen *Repräsentationen* handelt.

Die Ressourcenklassen besitzen die *Standardmethoden* GET, PUT, POST und DELETE, wobei nicht alle Methoden in einer Klasse implementiert werden müssen.
Durch die Annotation @Produces und @Consumes kann für eine Methode ein Ausgabe-bzw. Eingabeformat (Mime-Type) angegeben werden.
Als Rückgabewert wird immer eine Response geliefert.

In Kapitel 2.1.1 wurden die unterschiedlichen Ressourcenkategorien bereits beschrieben. Diese wurden in der Referenzimplementierung wiedererkannt und bei genauerer Beobachtung einige Unterschiede im Aufbau dieser Klassen gefunden:

- *Listenressourcen* besitzen als einzige Ressourcenart eine Annotation mit einer URL und einen Pfad, um zu den einzelnen Primärressourcen der Liste zu gelangen.

- *Primärressourcen* verfügen über einen Pfad zu den untergeordneten Ressourcen, den Subressourcen.

- *Subressourcen* haben den gleichen Aufbau wie eine Listenressource jedoch ohne die Annotation Pfad.

Zusätzlich enthalten die Ressourcenklassen noch einige Variablen, die im Rahmen dieser Arbeit nicht weiter erläutert werden.

Bei der Analyse der Repräsentationsklassen wurde festgestellt, dass Variablen über Annotationen als XML-Elemente markiert werden können, sodass sie sich bei der Ausgabe in den gewohnten Tags befinden.

Im Verlauf dieser Aufgabe wurde der ConTexter überarbeitet, um die spätere Generierung zu vereinfachen. Dies hatte einige Veränderungen zur Folge.

Die unterschiedlichen Ressourcenkategorien wurden in Collectionressource, Primärressource und Filterressource gegliedert.

Die Collectionressource enthält zur URL eine weitere Annotation. Durch diese wird angegeben, welche Klasse einer Primärressource die Collectionressource enthält.
Außerdem wurde der Pfad zu den einzelnen Primärressourcen einer Collection in eine Oberklasse ausgegliedert, sodass diese nicht neu generiert werden müssen. Zudem wurden die einzelnen Variablen innerhalb der Klassen ausgegliedert.

Die Primärressource erbt ebenfalls von einer Oberklasse und hat eine Annotation. Diese zeigt, zu welcher Collectionressourceklasse eine Primärressource gehört. Desweiteren wird über einen Pfad zu einer Filterressource navigiert.

Die Filterressource hat, wie in der vorigen Version des ConTexter, einen ähnlichen Aufbau wie die Collectionressource. Sie verfügt über die Annotation, die auf eine Primärressourcenklasse verweist und erbt ebenfalls von einer Oberklasse.

Die Analyse des ConTexter zeigte, dass sich ein RESTful Webservice aus mehreren Komponenten zusammensetzt:

- Entitäten repräsentieren mit Attributen den Zustand einer Ressource, wodurch keine weiteren Attribute innerhalb einer Ressource benötigt werden.

- Ressourcen sind das zentrale Objekt in der REST-Domäne. Wie in 2.1.1 beschrieben, gibt es zur besseren Strukturierung unterschiedliche Ressourcen-Typen bzw. Kategorien, die auch hier Anwendung finden.

- Repräsentationen werden je nach Ausgabeformat unterschiedlich implementiert.

- Ressourcen haben Methoden. Diese können, wenn nötig, ein Eingabe- und Ausgabeformat enthalten und haben als Rückgabewert eine *Response*.

- Datentypen wie z. B. *String* und *Integer* werden benötigt um Attribute zu typisieren.

3.2 Ableitung von Sprachmitteln für die DSL

Für die Entwicklung der DSL bietet es sich zunächst an eine graphische Darstellung zu wählen, anstatt direkt mit der textuellen Beschreibung zu beginnen. Daher wurde zunächst ein Meta-Modell entwickelt.

Für die Realisierung des Meta-Modells gibt es zwei Möglichkeiten. Zum einen das "Top-Down"-Prinzip, bei dem ein Meta-Modell entwickelt und anschließend geprüft wird, ob diese

das gewünschte System ausreichend abdeckt. Zum anderen gibt es das "Bottom-Up"-Prinzip. Bei diesem Verfahren wird das Zielsystem zunächst genau spezifiziert, um dann ein Meta-Modell abzuleiten. [T98]
Da im Rahmen dieser Arbeit die Referenzimplementierung ConTexter zur Untersuchung stand, wurde das letztere Prinzip angewandt.

Folgende Beschreibungen beziehen sich auf das Meta-Modell in Abbildung 3. Zentrales Objekt ist die Klasse *RestWebService*. Innerhalb dieser Klasse werden zunächst Elemente der Klassen *Resource* und *Entity* definiert. Zudem muss ein Name für den Webservice angegeben werden.

In der Klasse *Entity* werden die Attribute, die den Zustand einer Ressource widerspiegeln, sowie der Name der Entität angegeben. Ein Attribut wird von der Klasse *Property* repräsentiert und enthält zum Namen des Attributes, einen Typ und eine optionale Repräsentationseigenschaft. Außerdem können innerhalb der Klasse Entity Repräsentationen für Filter- und Primärressourcen definiert werden.

Die Klasse *Representation* enthält eine Liste von Namen der Attribute, die in einer Repräsentation dargestellt werden sollen.

Ressourcen werden von der Klasse *Resource* vertreten. Resource dient dabei als eine Oberklasse für die Ressourcenkategorien der Klasse *PrimaryResource* und der Klasse *CollectionResource*. Beide Klassen haben einen Namen, der dem eines vorher erstellten Entity entsprechen muss. Außerdem ist es möglich für jede Ressource Methoden zu definieren.

Wichtigster Bestandteil des Modells ist die Hierarchie zwischen den einzelnen Ressourcen. Dabei kann eine Collectionressource oder Listenressource Primärressourcen enthalten. Diese können wiederum Filterressourcen definieren, die dann auch wieder Primärressourcen enthalten, usw.

Die Klasse *Method* hat einen Namen und eine HTTP-Methode aus der Enumeration *Methods*. Zusätzlich lassen sich Eingabe- und Ausgabeformate über Mime-Types angeben.

Die Datenaustauschformate lassen sich über die Klasse *MimeType* definieren und werden von einem Namen als String oder einem der vordefinierten Typen XML,JSON oder beliebig repräsentiert.

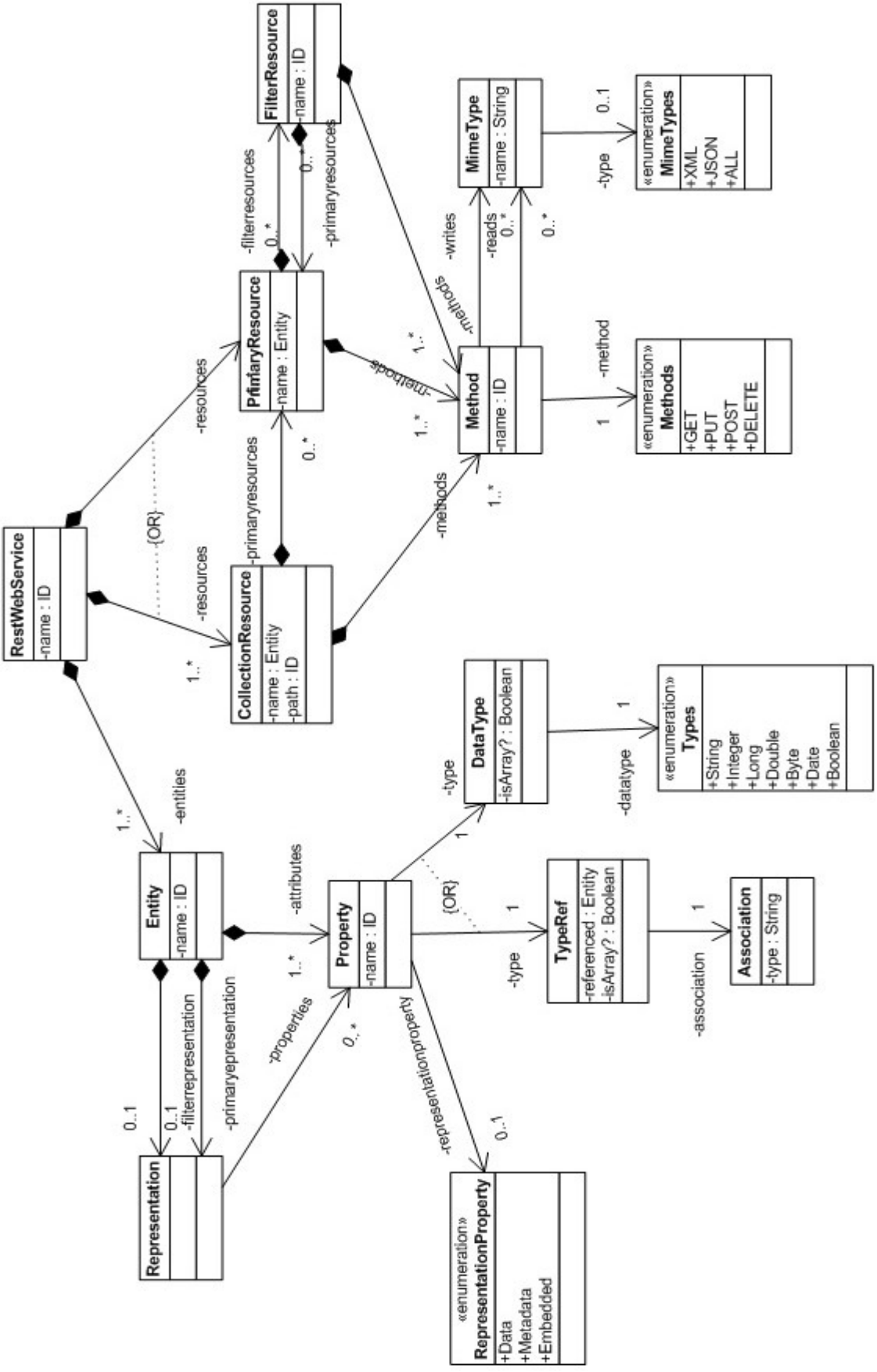

Abbildung 3: Das entwickelte Meta-Modell

Nachdem die abstrakte Syntax beschrieben wurde, folgt nun die konkrete Syntax. Diese soll mithilfe von Xtext umgesetzt werden. Dabei können, durch die Nutzung von Anführungszeichen konkrete Syntaxelemente zur DSL hinzugefügt werden.

- Die Benutzung von Klammern unterstützt zudem eine Art Blockbildung, wie sie bereits aus anderen Programmiersprachen bekannt ist.

- Die Verwendung von Semikola für die Trennung von Statements oder Anweisungen erhöht die Übersichtlichkeit. Desweiteren soll dies, erfahrenen Programmierern in Java, eine gewisse Vertrautheit im Umgang mit der neuen Sprache bringen.

Der Codeausschnitt zeigt wie ein RESTful Webservice mit der entwickelten DSL definiert wird.

```
 1   restwebservice <ID> {
 2         package <ID>;
 3
 4         entity <ID>{
 5               <Association> <isArray?> <Type> <ID> as
 6                     <RepresentationProperty>;
 7               primaryrepresentation <Property>
 8               filterrepresentation <Property>
 9         }
10         collectionresource for <Entity>{
11               path /<ID>;
12               methods get writes <MimeType>, post reads
13                     <MimeType> writes <MimeType>;
14
15               primaryresource for <Entity>{
16                     methods get writes <MimeType>, put reads
17                           <MimeType> writes <MimeType>, delete;
18
19                     filterresource filtered by <ID>{
20                           methods get,post;
21                     }
22                     ...
23               }
24               ...
25         }
26         ...
27   }
```

Listing 8: Entwicklung mit der DSL

Die hervorgehobenen Wörter sind die einzelnen Elemente der abstrakten Syntax und repräsentieren die Schlüsselwörter. Sie symbolisieren was gerade modelliert wird. In Zeile 1 beginnt die Definition eines RESTful Webservices mit dem Schlüsselwort *restwebservice* gefolgt von einem Namen mit dem Typ ID. ID ist eine von Xtext angelegte Regel, die die Struktur einer Zeichenkette bestimmt. Daher kann ein Name optional mit einem Zirkumflex ^

beginnen und aus einer Zeichenkette von Groß- und Kleinbuchstaben, sowie einen Unterstrich bestehen. Wenn in der folgenden Beschreibung kein anderer Typ erwähnt wird, ist der Typ ID gemeint.
Außerdem macht die Zeile 1 die in der konkreten Syntax erwähnte Blockbildung ersichtlich. Durch eine geöffnete, geschweifte Klammer beginnt der Block und in Zeile 27 wird dieser wieder geschlossen. Dieses Prinzip der Blockbildung wird auch bei der Definition der Entität, der Collectionressource, der Primärressource und der Filterressource angewendet.

Zeile 2 zeigt die Packagedeklaration, die durch das Wort *package* eingeleitet wird und einen zugehörigen Namen vom Typ ID hat. Dieser ist entweder in Form einer Zeichenkette oder einer Aneinanderreihung von Zeichenketten, verbunden durch einen Punkt, wie z. B. de.unihannover festgelegt. Ein Semikolon beendet die Packagedeklarierung.

Das Erstellen einer Entität wird durch ***entity*** eingeleitet und fordert einen Namen sowie ein oder mehrere Attribute. Zusätzlich können nach den Schlüsselwörtern *primaryrepresentation* oder *filterrepresentation* in Zeile 6 und 7 die Namen von vorher definierten Attributen angegeben werden, damit diese als Element einer Repräsentation generiert werden. Bei der Eingabe der Attribute muss die Reihenfolge nicht beachtet werden.

Wie in Zeile 5 zu erkennen, besteht ein Attribut aus einer optionalen Assoziation, einer Mengenangabe, einem Typ, einem Namen, einer optionalen Repräsentationseigenschaft und einem Semikolon, welches den Abschluss einer Attributdeklarierung symbolisiert.

Die Assoziation gibt das Verhältnis zwischen Entitäten an und findet ausschließlich Verwendung, wenn der Typ eines Attributes eine referenzierte Entität ist.
Dabei kann angegeben werden, ob eine Entität andere Entitäten enthält, *one has* oder ob mehrere Entitäten andere Entitäten enthalten, *many have*.

Über die optionale Mengenangabe wird angegeben, ob es sich um eine einzelne Variable oder eine Liste handelt. Für eine Darstellung in Listenform muss bei *isArray?* ein *list* stehen. Ebenfalls in Zeile 5 zu finden.

Der Typ eines Attributes kann ein *String, Integer, Long, Double, Date, Boolean* oder der *Name* einer zuvor deklarierten Entität sein.

Die optionale Eingabe der Repräsentationseigenschaften wird durch das Schlüsselwort **as** eingeleitet. Anschließend wird eine Annotation aus *data, metadata und embedded* angefügt. Diese bestimmen, welche Annotation ein Attribut in einer Repräsentation zugeordnet wird.

Die Ressource kann auf zwei Arten definiert werden. Zum einen als CollectionResource und zum anderen als PrimaryResource.

Das Erzeugen einer CollectionResource beginnt mit den Worten *collectionresource for* und einem Namen, der ausschließlich den Namen einer Entität haben muss.
Nach einer offenen geschweiften Klammer folgt der Befehl *path/* in Zeile 11, um einen Pfad für die spätere Navigation zur Ressource anzugeben. Außerdem können Methoden und Primärressourcen definiert werden. Für den Abschluss der CollectionResource wird eine geschlossene geschweifte Klammer verwendet.

Die Deklarierung einer Methode erfolgt durch das Wort **methods** und einer darauffolgenden Operation *get,put,post* oder *delete*. Über die Befehle *read* und *write* und anschließenden MimeTypes ist es zudem möglich Eingabe- und Ausgabeformate der Methode anzugeben. Die einzelnen Formate und Operationen werden durch Komma voneinander getrennt und hintereinander geschrieben. Ein Semikolon schließt die Aufzählung ab.

Ein *MimeType* ist entweder ein *String* oder ein vordefinierter Wert wie *xml, json oder* *.

Die PrimaryResource hat eine ähnliche Struktur wie die CollectionResource.
Es fehlt lediglich der Pfad, der hier nicht benötigt wird und anstatt der Primärressourcen können Filterressourcen initialisiert werden.

Die Deklaration einer FilterResource startet mit *filterresource filtered by* und einem Namen. Wie bei CollectionResource und PrimaryResource, werden die weiteren Elemente zwischen geschweiften Klammern definiert. Diese können Methoden und Primärressourcen sein.

Für einen Einblick, wie eine textuelle DSL mit Xtext aussehen kann, befindet sich der komplette Quellcode auf der beigelegten CD.

3.3 Beispiel To-Do-Liste

Auf den vorigen Seiten wurde über die theoretischen Grundlagen und das Konzept zur Implementierung einer DSL gesprochen. In diesem Abschnitt wird anhand des Beispiels einer To-Do-Liste gezeigt, wie die erstellte DSL genutzt wird.

```
restwebservice ToDoList{
      package de.example;

      entity Entry{
            String text as data;
            Boolean status as metadata;
            primaryrepresentation status,text;
      }
      collectionresource for Entry{
            path /entries;
            methods get,post;
            primaryresource for{
                  methods get,put,delete;
            }

      }
}
```

Listing 9: Entwicklung einer ToDo-Liste

Der RESTful Webservice trägt den Namen `ToDoList` und legt als Packagedeklaration den Titel `de.example` fest.

Danach wird die Entität `Entry` angelegt. In ihr sind die Attribute String `text` und Boolean `done`. In dem Attribut `text` wird eine Aufgabe eingetragen. In Attribut `done` wird gespeichert, ob eine Aufgabe bereits getätigt wurde. Beide Attribute sollen in der Primärrepresentation zu sehen sein.

Um zur Collectionressource von `Entry` zu navigieren wurde der Pfad `entries` angegeben. Mit den Methoden `get` erhält man sämtliche Einträge der Collection und mit `post` werden neue Einträge in die Collection integriert.

Die Primärressource enthält die Methoden `get` zum Wiedergeben des Inhalts, `put` für die Aktualisierung des Inhalts und `delete` um einen Eintrag zu löschen.

4. Implementierung

In diesem Kapitel geht es um die Implementierung der DSL und des Generators. Zunächst werden ein paar Grundlagen für die Entwicklung mit Xtext und REST-Framework Jersey erläutert. Um die weitere Beschreibung verständlicher zu gestalten, werden anschließend Begriffe und Befehle für die Programmierung der Templates erläutert. Danach wird gezeigt wie die Verwendung der DSL zu generierten Code führt.

4.1 Xtext

Für die Implementierung einer DSL legt Xtext eine Reihe von Regeln fest. Eine von diesen Regeln ist die sogenannte Parser Rule. Sie ist für die Definition der abstrakten Syntax verantwortlich. Der folgende Quellcode zeigt eine einfache Parser Rule: [E00]

```
1    State :
2         'state' name=ID
3         ('actions' '{' (actions+=[Command])+ '}')?
4         (transitions+=Transition)*
5         'end'
6    ;
```

Listing 10: Parser Rule State

In der Regel **State** werden sogenannte Features definiert. Features können als Rule Calls, Cross References oder Terminal Rules auftreten. Rule Calls sind Aufrufe anderer Regeln. Zu sehen ist dies in Zeile 4, bei dem Aufruf einer anderen Parser Rule **Transition**.

Ein weiteres Feature wird in Zeile 3 gezeigt, die Cross Reference. Sie wird in eckigen Klammern angegeben und referenziert eine vorhandene Instanz einer Parser Rule. Der hauptsächliche Unterschied der zuletzt genannten Regeln ist, dass bei einem Rule Call eine neue Instanz angelegt und bei einer Cross Reference eine Instanz referenziert wird.

Terminal Rules sind vordefinierte Regeln, die zur Beschreibung von abgeschlossenen, atomaren Regeln benutzt. In dieser Arbeit wird die Terminal Rule ID verwendet. Ihre Definition wurde bereits erläutert.

Desweiteren wurden in dieser Arbeit noch zwei spezielle Parser Rules verwendet, die Data Type Rules und die Enum Rules. Die Data Type Rule wird bei der Deklarierung von Namespacing verwendet. Dies könnte wie folgt aussehen und ermöglicht die Angabe eines Namensraum nach dem Prinzip <text>.<text>. Das Codebeispiel zeigt eine Data Type Rule: [E00]

```
QualifiedName:
      ID  ('.' ID)*
;
```

Listing 11: Data Type Rule QualifiedName

Enum Rules werden bei Aufzählungen verwendet und können, wie eine Parser Rule, einem Typ zugewiesen werden. Folgendes Beispiel zeigt eine Enum Rule: [E00]

```
enum Visibility:
PUBLIC='public' | PRIVATE='private' | PROTECTED='protected';
```

Listing 12: Enum Rule Visibility

Bei der Verwendung von Enum Rules muss beachtet werden, dass jedes Element einen textuellen Wert besitzt.

Damit die genannten Regeln untereinander verknüpft werden können, wird die sogenannte EBNF (Erweiterte Backaus-Naur-Form) verwendet. [E00]

Operator	**Bedeutung**
Keine Angabe	Genau einmal vorkommend
?	Keinmal oder einmal vorkommend
*	Keinmal oder beliebig oft
+	Einmal oder beliebig oft
|	Zeichen für eine Alternative

Tabelle 1: Verwendete EBNF-Ausdrücke

Innerhalb der Features werden die Regeln jedoch nicht durch die reine EBNF ausgedrückt, sondern von Xtext definierten Zuweisungsoperatoren.

Wie am Anfang zu sehen ist, werden im Quellcode die beiden Zuweisungsoperatoren = und += benutzt. Der erste ist eine einfache Zuweisung und bedeutet, dass ein Feature genau einen Wert erhält. Der zweite bedeutet, dass ein Feature mehrere Werte darstellt, wie z. B. eine Liste.

Für ein besseres Verständnis über Xtext empfiehlt sich die Dokumentation. [E00]

Xtext bietet, neben der Entwicklung einer DSL, die Möglichkeit einen Generator zu implementieren. Dabei erstellt Xtext ein Projekt auf der Basis der Modeling Workflow Engine (MWE2) und verwendet dabei die Komponenten von Xpand, Xtend, Check und einem MWE2-Workflow. Durch sie entsteht die Transformation eines Modells in z. B. Quellcode.

Der Workflow enthält zwei Instanzen. Zum einen den Reader, der die in Xtext angelegten Sprachen initialisiert und für das Laden der Modelle verantwortlich ist.

Zum anderen den Generator, der ein Xpand-Template aufruft und das Zielverzeichnis für die Generierungen bestimmt. Xpand ist eine statisch typisierte Templatesprache mit speziellen Eigenschaften für die Generierung, darunter die Transformation von Modellen. Xtext ist eine funktionale Sprache, die genutzt wird um während der Laufzeit Änderungen an einem Modell vorzunehmen. Dies wird in sogenannten Extensions Dateien definiert. Die Sprache Check wird für die Überprüfung der Validität eines Modells genutzt und prüft unter anderem die Definiton von Modellelementen, um gegebenenfalls eine Transformation abzubrechen.

Für die Implementierung der Templates wurden hauptsächlich die Befehle in Tabelle 2 genutzt. Jeder Befehl, ausgenommen EXPAND, muss auch wieder geschlossen werden. Beispielsweise wird der Anfang eines Templates mit «DEFINE ... » definiert und mit «ENDDEFINE» geschlossen.

Befehl	**Bedeutung**
«DEFINE *definitionName* FOR *Type*»	Definition eines Templates. Sie besteht aus einem Namen, einer Parameterliste (optional) und einen *Type*, für den die Definition gilt.
«EXTENSION *name*»	Deklariert die Verwendung einer Extension. Die in der Extension definierten Funktionen können im Template genutzt werden.
«EXPAND *defintion* FOR/EACH *Type*»	Expandiert einen *DEFINE*-Block. Der Type gibt an, für welche(s) Elemente(e)die Definition aufgerufen wird. Für Collections muss *FOREACH* verwendet werden und bei einwertigen Ergebnistypen *FOR*.
«IF *condition*»	Expandiert den Inhalt Bedingungsabhängig.
«FOREACH *elements* AS *e*»	Expandiert den Inhalt des Blocks für jedes Element von *elements*. Dabei wird das aktuelle Element an eine Variable *e* zugewiesen.
«FILE *fileName*»	Leitet die Ausgabe eines Templates in eine Datei.
«REM *comment*»	Kommentare

Tabelle 2: Funktionen für die Templatesprache Xpand [E00]

4.2 Jersey

Die generierten Anwendungen sollen auf dem REST-Framework Jersey beruhen. Jersey ist die Referenzimplementierung des JAX-RS-Standards. Dieser definiert die Java API für RESTful Webservices.

Die Ziele von JAX-RS sind: [T09a]

- REST-Konformität,
- Einfachheit bei der Entwicklung,
- Ausnutzen moderner Sprachmittel (insbesondere Annotations) und
- der Zugriff auf alle Konstrukte von HTTP.

Der interessanteste Teil für die Generierung sind die Annotationen, die zur Pfadangabe, Ausgabeformate oder Parameter verwendet werden. Im Folgenden werden die Annotationen erläutert, die bei der Generierung verwendet wurden:

Annotation	Beschreibung
Consumes	bestimmt eine Liste von Medientypen, die gelesen werden können
Produces	bestimmt eine Liste von Medientypen, die geschrieben werden können
GET,PUT,POST,DELETE	legen fest, dass eine Methode HTTP-Requests je nach Annotation verarbeiten können
Path	gibt den relativen Pfad einer Ressource an. Als Klassenannotation definiert es eine Wurzel-Ressource(CollectionResource) und als Methodenannotation identifiziert es eine Sub-Ressource(FilterResource)
FormParam	verbindet den Wert einer Formularvariable mit einem Parameter einer Methode
QueryParam	verbindet den Wert eines Query-Parameter mit einem Parameter einer Methode

Tabelle 3: Verwendete Annotation mit Bedeutung nach [HS09]

4.3 Entwicklung des Generators

Zu Beginn der Implementierungsphase wurde festgestellt, dass eine gute Übersicht in der Generierung bewahrt werden muss. Um dies zu erreichen, wurden die verschiedenen Bestandteile und sich wiederholenden Ausdrücke in unterschiedliche Templates gegliedert.

Der Aspekt der Übersicht wurde auch bei den generierten Dateien berücksichtigt. Mit dem Befehl *FILE* ,aus der Tabelle 2, ist es nicht nur möglich den Titel einer Klasse zu bestimmen, sondern diese Datei in ein bestimmtes Package generieren zu lassen. So befinden sich alle Entitäten in einem Package .ejb.entities, alle Repräsentationen in .representations und alle Ressourcen in .resources. Außerdem werden alle Ressourcen in ein separates Package gegliedert, entsprechend ihres Ressourcennamens.

Für die Quellcodegenerierung wird sich an die Konventionen der Referenzimplementierung gehalten. Klassennamen beginnen z. B. mit Großbuchstaben und Attribute der Entitäten mit Kleinbuchstaben.

Als Grundlage für die einzelnen Templates wurden zunächst Quellcodeteile direkt an die entsprechenden Stellen der Templates positioniert und anschließend die Befehle
aus Tabelle 2 verwendet, um den restlichen Code aus den Modellen zu generieren.

Die Generierung beginnt mit der Template-Datei *Main*. In ihr werden die weiteren Aufrufe der Modellelemente getätigt. Eines dieser Templates ist das Template *Entity*, in dem die einzelnen Entitäten mit den entsprechenden Attributen entstehen. An dieser Stelle wird nicht nur überprüft, ob ein Attribut eine Liste ist, sondern auch welche Annotation es für die Assoziation erhält. Die Tabelle 4 zeigt die vier unterschiedlichen Fälle, die dabei zu beachten sind. Außerdem werden für jedes Attribut getter- und setter- Methoden konstruiert.

isArray?	**Association**	**Annotation**
list	one has	@OneToMany
list	many have	@ManyToMany
	one has	@OneToOne
	many have	@ManyToOne

Tabelle 4: Berechnung der Annotation

Als nächstes werden die Templates der *PrimaryRepresentation* und der *Collection-SubRepresentation* aufgerufen. Innerhalb dieser beiden Dateien wird die Auslagerung sich wiederholender Ausdrücke deutlich, da sie neben den manuell implementierten Codeteilen nur einen Befehl zum Aufruf eines weiteren Templates haben. Dieses generiert die einzelnen Attribute einer Repräsentation und deren zugehörige Annotation.

Danach folgt die Generierung der einzelnen Collectionressourcen im Template *CollectionResource*. Zunächst entstand die Idee sämtliche Methoden der Ressourcen in ein einzelnes Template auszugliedern, um so Codewiederholungen zu vermeiden. Dies wurde aber schnell wieder verworfen, da sich die Methoden nur auf dem ersten Blick ähneln. Schließlich wurde nur die Auswertung der Eingabe- und Ausgabeformate der Methoden in

das Template *Mediatypes* ausgelagert. In CollectionResource wird zudem der Aufruf der Templates *CollectionRepresentation* und *PrimaryResource* gemacht. Da Collection-Representation fast nur aus manuell implementierten Teilen besteht, bedarf es hier keiner weiteren Erklärung.

Das Template PrimaryResource hat, wie CollectionResource, eine Implementierung der HTTP-Methoden und besitzt zusätzlich die getter-Methoden zu den Filterressourcen. Danach wird das Template *FilterResource* aufgerufen, welches nicht weiter erklärt werden muss, da es den vorigen Templates im Aufbau ähnelt.

Der folgende Quellcode zeigt den Inhalt des fertigen Templates *Entity***.** Jedes Template beginnt mit den beiden ersten Zeilen, die dafür sorgen, dass die Elemente der DSL und eine Extensions-Datei genutzt werden können. Außerdem zeigt er die angesprochenen Quellcodefragmente und den zuvor erwähnten Befehl File und die Angabe für die Namens- und Packagedeklarierung.

```
«IMPORT de::unihannover::restdsl::restDsl»
«EXTENSION templates::Extensions»

«DEFINE main(PackageDeclaration p) FOR Entity -»
«FILE (applicationPackageName(p,"ejb.entities")+
     "."+name).replaceAll("\\.", "/") +".java"-»

package «applicationPackageName(p,"ejb.entities")»;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.*;
import «applicationPackageName(p,"server.ejb.facades.")»
     «name»Facade;
import «applicationPackageName(p,"restlib.jpa.Facade")»;
import «applicationPackageName(p,"restlib.jpa.Model")»;

@Entity
@Facade(«name»Facade.class)
public class «name» extends Model implements Serializable {

     private static final long serialVersionUID = 1L;

     «EXPAND propertyName FOREACH properties-»
     «EXPAND GetterSetter FOREACH properties-»
}
«ENDFILE-»
«ENDDEFINE»
«DEFINE propertyName FOR DataProperty-»
     «FOREACH type.eContainer().eContents().typeSelect(TypeRef) AS
          e-»
     «IF e.isArray-»
```

```
      «switch(e.association.type){
            case "one has": "@OneToMany"
            case "many have" : "@ManyToMany"
            default: ""
      }»
      private List<«e.referenced.name»> «name»;
      «ELSE-»
      «switch(e.association.type){
            case "one has": "@OneToOne"
            case "many have" : "@ManyToOne"
            default: ""
      }»
      private «e.referenced.name» «name»;
      «ENDIF-»
      «ENDFOREACH-»
      ...
      «ENDFOREACH»
«ENDDEFINE»
   ...
«ENDDEFINE»
```

Listing 13: Auszug aus dem Template Entity

Für die spätere Verwendung der entwickelten DSL und des Generators müssen diese in den Buildprozess eines Projektes integriert werden. Als Buildmanagement Tool wird das Programm Maven verwendet.

5. Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse und Erkenntnisse der Bachelorarbeit zusammengefasst und ein Ausblick auf mögliche Erweiterungen der entwickelten DSL gegeben.

5.1 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung einer domänenspezifischen Sprache für die Domäne REST. Anschließend sollte mit den Techniken aus der modellgetriebenen Softwareentwicklung und der erstellten DSL ein Generator implementiert werden, der als Ergebnis RESTful Webservices liefert.

In Kapitel 2 wurden die Grundlagen zur Bearbeitung der Bachelorarbeit erklärt. Zunächst ging es um die Definition und Eigenschaften eines traditionellen Webservices. Danach wurde auf die Kommunikation in HTTP eingegangen und der Aufbau eines Request und einer Response gezeigt. Im Anschluss wurden die Prinzipien des Representational State Transfer erläutert und ein Vergleich zwischen Webservices und RESTful Webservices gemacht. Da es bei dem Architekturstil REST um unterschiedliche Datenaustauschformate ging, wurden die Grundlagen der häufig verwendeten Formate XML und JSON, näher geschildert. Das Ende des Kapitels behandelte den Abschnitt der modellgetriebenen Softwareentwicklung. In diesem ging es um die Ziele und Grundlagen, sowie die Verwendung der Techniken bei MDSD.

Das 3. Kapitel beschäftigte sich mit dem Kern dieser Arbeit. Zu Beginn ging es um die Analyse der Referenzimplementierung ConTexter. Aus den Ergebnissen der Analyse und dem erlangten Wissen aus der Domäne entstand das Konzept der DSL. Im weiteren Verlauf des Kapitels wurde die abstrakte Syntax der DSL mit einem Meta-Modell vorgestellt. Anhand eines Codeausschnitts vom Grundgerüst der DSL und eines kleinen Beispiels wurde die Verwendung der entwickelten Sprache erklärt.

Nachdem die neue domänenspezifische Sprache feststand, ging es in Kapitel 4 um die Implementierung eines Generators. Am Anfang des Kapitels wurden die Grundlagen für die Entwicklung mit Xtext gezeigt. Da die generierten Anwendungen auf dem REST-Framework Jersey beruhen sollten, musste an dieser Stelle noch ein Teil des JAX-RS Standards erläutert werden. Dabei wurden die, in den generierten Dateien, verwendeten Annotation präsentiert.
Der letzte Abschnitt schilderte die Entwicklung der Templates.

5.1.1 Fazit

Der Representational State Transfer ist ein sehr interessanter Architekturstil um Webservices zu entwickeln. Der Aspekt von Hypermedia hat dabei besonders fasziniert, da es dadurch möglich ist, jede adressierbare Ressource in einer Anwendung zu verknüpfen und somit einen riesigen Zugriff auf Informationen im Web hat.

Während des Studiums wird auf die Grundideen der modellgetriebenen Softwareentwicklung eingegangen und häufig die Abbildung 1 präsentiert. Auf dem ersten Blick hat man noch keine wirkliche Vorstellung darüber, welcher Aufwand in der Umsetzung dieser Techniken besteht. Dies wurde bei der Erstellung dieser Bachelorarbeit ersichtlich. Aus diesem Grund sollte vorher überprüft werden, ob sich die Verwendung von MDSD lohnt. Die Entwicklung von ein bis zwei Anwendungen für eine Domäne könnten durch andere Softwareentwicklungsmethoden schneller realisiert werden.

Der Entwurf und die Implementierung einer domänenspezifischen Sprache erwiesen sich als anspruchsvoller als zuvor gedacht. Zum einen muss darauf geachtet werden, dass die Elemente einer Sprache auch domänenspezifisch sind und gegebenenfalls auf spezielle Teile verzichtet werden muss. Zum anderen wurde die Abhängigkeit zwischen DSL und Generator deutlich, da Veränderungen an der Sprache zu Veränderungen an dem Generator führten und umgekehrt. Oft stellte man während der Implementierung fest, dass bestimmte Sprachelemente nicht passen oder fehlen.

Am Ende lässt sich sagen, dass die Entwicklung mit der erstellten DSL einfach gestaltet wurde, leicht zu bedienen und zu erlernen ist. Außerdem wird durch die entwickelte Sprache der Aufwand bei der Realisierung eines RESTful Webservices minimiert.

5.2 Ausblick

Am Ende der Arbeit entstanden zwei mögliche Punkte, in wie fern man die entwickelte Sprache erweitern könnte. Der erste Punkt ist die Integration einer Dokumentation. In dieser könnten die einzelnen Elemente eines Webservices genauer beschrieben und gleichzeitig als Kommentierungen für die Codegenerierung genutzt werden. Beispielsweise könnte in der Dokumentation angegeben werden, welche Methoden eine Ressource unterstützt oder welche Subressourcen sie besitzt.
Dabei wäre es vorteilhaft, wenn die Dokumentation unterschiedliche Formate unterstützt, damit der Client bei der Beschaffung von Informationen über den Webservice mehrere Optionen hat.

Der zweite Punkt ist die Versionierung. Mit einer Versionierung der Modellelemente könnte man die Möglichkeit schaffen, verschiedene Versionen eines Webservices bereitzustellen, ohne dass ein komplett neues Modell entworfen werden muss. Ein wichtiger Punkt ist die Berücksichtigung der Kompatibilität zwischen den einzelnen Versionen. Dies könnte durch Vererbung der einzelnen Versionen berücksichtigt werden, da neue Elemente hinzugefügt und bereits vorhandene Elemente direkt übernommen werde.

Neben diesen zwei Erweiterungen, könnte man die entwickelte DSL für die Generierung von Anwendungen mit anderen REST-Frameworks benutzen. Dafür müssten lediglich neue Templates für die entsprechende Plattform entwickelt werden.

Literaturverzeichnis

[BFM05] T. Berners-Lee, R. Fielding, and L. Masinter. (2005) *Uniform Resource Identifier (URI): Generic Syntax.* Link: http://www.ietf.org/rfc/rfc3986.txt

[BPSMY08] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. (2008) *Extensible Markup Language (XML) 1.0 (Fifth Edition).* Link: http://www.w3.org/TR/2008/REC-xml-20081126/

[E00] *Eclipse Helios (3.6) Documentation* (2000) Link: http://help.eclipse.org/helios/index.jsp

[F00] R. Fielding. (2000) *Architectural Styles and the Design of Network-based Software Architectures.* Link: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[F09] R. Fielding et al. (1999) *Hypertext Transfer Protocol -- HTTP/1.1*. Link: http://www.ietf.org/rfc/rfc2616.txt

[H10] T.Hüther. (2010) Eine domänenspezifische Sprache zur Transformation von Anwendungszuständen in Webapplikationen. Link: http://static.se.uni-hannover.de/f/pub/File/pdfpapers/Huether2010.pdf

[HB02] H. Haas and A. Brown. (2002) *Web Services Glossary.* Link: http://www.w3.org/TR/ws-gloss/

[HS09] M. Hadley and P. Sandoz. (2009) *JAX-RS: JAVA API for RESTful Web Services.* Link: http://jsr311.java.net/nonav/releases/1.1/spec/spec.html

[K09] K. Jäger, *Ajax in der Praxis*. Heidelberg: Springer, 2009.

[RR07] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol,CA, USA: O'Reilly, 2007.

[S11] S. Schreier. (2011) *Modeling RESTful applications.* Link: http://ws-rest.org/2011/proc/a4-schreier.pdf

[STK02] J. Snell, D. Tidwell, and P. Kulchenko, *Programming Web services with SOAP*. Sebastopol,CA, USA: O'Reilly, 2002.

[SV05] T. Stahl and M. Völter, *Modellgetriebene Softwareentwicklung*. Heidelberg: Dpunkt, 2005.

[T09a] S. Tilkov. (2009) *RESTful Web Services mit Java.* Link: http://www.innoq.com/files/JM_1.09_tilkov_jax-RS.pdf

[T09b] S. Tilkov, *REST und HTTP: Einsatz der Architektur des für Integrationsszenarien*. Heidelberg: Dpunkt, 2009.

[T98] J. Terpenny. (1998) *Blending Top-Down and Bottom-Up Approaches in Ceptual Design.* Link: http://wwwrp.me.vt.edu/bohn/projects/terpenny/IERC98_Janis_Terpenny.pdf

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang

A. Inhalt der CD

Auf der beigelegten CD zu dieser Arbeit, befindet sich:

- jeweils eine Version dieser Ausarbeitung im PDF- und DOC-Format

- der Quellcode der Implementierung als Eclipse Xtext Projekt

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, 13.August 2011

Thorsten Kerber