Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 104

Vergleich und Bewertung von Methoden und Tools für den Entwurf und die Realisierung von REST APIs

Marcus Eisele

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Dipl.-Inf. Florian Haupt
Beginn am:	11. Mai 2016
Beendet am:	10. November 2016
CR-Nummer:	D.2.2

Kurzfassung

REST-Schnittstellen haben sich die letzten Jahre in der Softwareindustrie etabliert. Abhängig von der eingesetzten Technologie gibt es viele Möglichkeiten eine REST-Schnittstelle zu entwerfen und umzusetzen. Für die Unterstützung des Entwurfs und der Realisierung von REST-Schnittstellen existiert ein modellgetriebener Ansatz mit akademischem Hintergrund. Neben diesem akademischen, modellgetriebenen Ansatz existieren weitere Ansätze basierend auf Beschreibungssprachen wie Swagger oder RAML, die bei Entwurf und Realisierung unterstützen. Diese Arbeit vergleicht den eben beschriebenen akademischen Ansatz mit zwei Ansätzen, welche beide jeweils eine der eben genannten Beschreibungssprachen nutzen. Der auf Swagger-basierende Ansatz wird durch eine bestehende Softwareentwicklung eines Industriepartners repräsentiert.

Der akademische modellgetriebene Ansatz und seine Werkzeuge werden mit den beiden anderen Ansätzen und deren Werkzeuge hinsichtlich ihrer Brauchbarkeit zum Entwurf und zur Umsetzung von REST-Schnittstellen in einem Industrieunternehmen untersucht. Dieser Vergleich der Entwurfs- und Realisierungsmethoden von REST-Schnittstellen wird exemplarisch an einem agil entwickelten Dienst zur Abfrage von Sonderzielen durchgeführt.

Der Vergleich betrachtet die verschiedenen Arbeitsweisen der vorgestellten Ansätze, die dabei enstehenden Artefakte und Modelle sowie den von ihnen erzeugten Quellcode. Für die Durchführung des Vergleichs der verschiedenen Ansätze wurden Nachbauten des Dienstes des Industriepartners für den akademischen Ansatz und den Ansatz unter Verwendung von RAML erstellt. Diese Nachbauten dienen als Grundlage für die Betrachtung des Entwicklungsprozesses, die Befragung der Entwickler sowie die Durchführung einer statischen Codeanalyse.

Die Befragung zeigt, dass die befragten Entwickler des Industriepartners die ihnen neu vorgestellten Ansätze im Allgemeinen nicht als bereit und lohnenswert für den Entwurf und die Umsetzung von REST-Schnittstellen in der Praxis erachteten. Die beteiligten Entwickler waren sich aber einig, dass modellgetriebenen Ansätze attraktive Möglichkeiten bieten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Dieses Kapitel dient der Einführung in diese Arbeit. Es besteht aus der Motivation (Abschnitt 1.1), welche den Grund für diese Arbeit beschreibt, gefolgt von einem Abschnitt der auf das Ziel der Arbeit (Abschnitt 1.2), inklusive der Aufgabenbeschreibung und der Abgrenzung, eingeht. Darauf folgt ein Abschnitt (Abschnitt 1.3) der sich mit dem methodischen Vorgehen der Durchführung der Arbeit beschäftigt. Abgeschlossen wird dieses Kapitel durch einen Überblick (Abschnitt 1.4) über den Aufbau dieser Arbeit.

1.1 Motivation

In den letzten Jahren hat sich REpresentational State Transfer (REST) als Architekturstil für Webservices etabliert und wird heute weitreichend eingesetzt. Viele REST-Schnittstellen besitzen nicht die von Roy Fielding in seiner Dissertation [Fie00] geforderten Eigenschaften. Resultat dieser fehlenden Eigenschaften der REST-Schnittstellen sind Systeme, welche die Vorteile des REST-Stils nicht vollständig ausschöpfen und dadurch viele der erhofften Eigenschaften vermissen lassen.

Neben der Missachtung des Architekturstils gibt es bei dem Entwurf und der Umsetzung von REST-Webservices oft ähnliche und sich wiederholende Arbeitsabläufe. Designentscheidungen für REST-Schnittstellen, wie das Definieren von Ressourcen, werden oftmals anhand formloser Entwürfe an Whiteboards oder mit Hilfe anderer Medien entworfene Skizzen, getroffen. Diese Skizzen dienen zum einen als Grundlage der späteren Implementierung, oft aber auch als Teil der Dokumentation. Die Umsetzung ist dabei, bei ausreichender Vollständigkeit der Skizzen, relativ trivial und bietet einen kleinen Handlungs- und Entscheidungsfreiraum. Unvollständige Entwürfe führen dabei oftmals zu Unterschieden zwischen gewolltem Verhalten der Schnittstelle und tatsächlicher Umsetzung. Um Missachtungen des Architekturstils beim Entwurf und Fehler während der Umsetzung zu vermeiden wäre es von Vorteil, wenn diese Entwürfe bereits ein definiertes Format besäßen, welches alle Unklarheiten bereits im Voraus beseitigt und die zu implementierenden Schnittstellen vollständig und ohne Mehrdeutigkeiten spezifiziert.

Neben der Einhaltung des Architekturstils ist die Aufgabe eines Entwicklers natürlich auch die Implementierung der entworfenen REST-Schnittstelle. Diese Implementierung ist, falls der Entwickler mit den eingesetzten Technologien bereits vertraut ist, eine oftmals sehr

repetitive Tätigkeit. Nach der Fertigstellung der Implementierung hat der Entwickler oftmals auch die Aufgabe das im Entwurf entstandene Dokument, als Teil der Dokumentation, auf aktuellem Stand zu halten. Sind also Änderungen an der Implementierung nötig, so müssen diese in das bereits bestehende Dokument übernommen werden. Durch den weitverbreiteten Einsatz von agilen und iterativen Methoden zur Softwareentwicklung werden diese Änderungen in vielen Projekten immer häufiger.

Mittlerweile gibt es bereits einige Methoden und Werkzeuge aus dem akademischen und industriellen Umfeld, welche sich mit der Modellierung und späteren Erstellung von REST-Schnittstellen beschäftigen. Die Wahl zwischen diesen Methode und den entsprechenden Werkzeugen ist nicht einfach und es gibt bisher keinen bewährten Standard.

1.2 Ziel

Das konkrete Ziel dieser Arbeit ist es unterschiedliche Methoden für den Entwurf und die Realisierung von REST-Schnittstellen zu untersuchen und zu vergleichen. Bei den unterschiedlichen Methoden sollen neben den reinen Entwurfs- und Implementierungsvorgängen auch die Auswirkungen auf bestehende Entwicklungsprozesse untersucht werden.

Neben einer unabhängigen Untersuchung soll der Vergleich der Methoden auch anhand einer Microservice-Implementierung eines Industriepartners durchgeführt werden. Diese Microservice-Implementierung soll dazu mit im Zuge dieser Arbeit erstellten Nachbauten unter Einsatz der unterschiedlichen Methoden verglichen werden.

Der Vergleich der Nachbauten und der bereitgestellten Implementierung soll eine Einschätzung über die realistischen Einsatzmöglichkeiten der verschiedenen Methoden für den Entwurf und die Realisierung von REST-Schnittstellen geben. Neben der Einschätzung kann nach dem Vergleich auch eine Qualitätsaussage über die zur Verfügung gestellte REST-Schnittstelle des Industriepartners gegeben werden.

Abgrenzung

Dieser Abschnitt grenzt das Thema der vorliegenden Arbeit ein und zeigt Themen, welche den Umfang dieser Arbeit übersteigen.

Vollständige Implementierung

Der vom Industriepartner zur Verfügung gestellte Microservice erfüllt neben den funktionalen Anforderungen auch sehr viele nicht funktionale Anforderungen. Zu diesen nicht

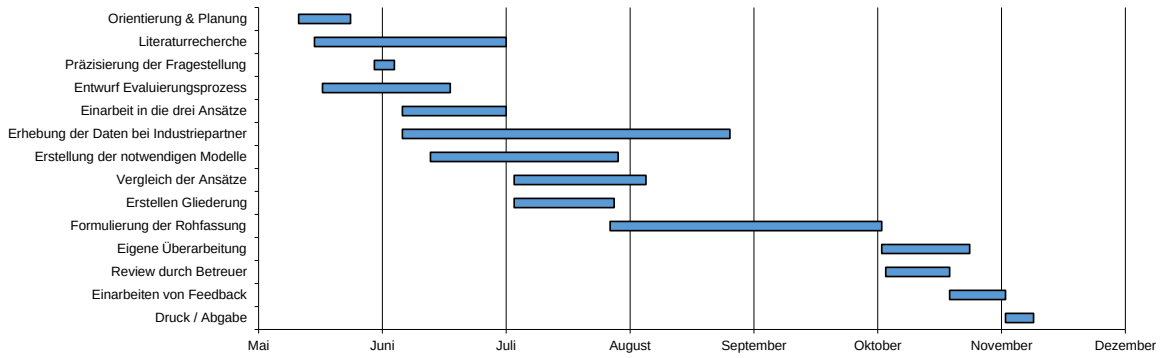

Abbildung 1.1: Gantt-Diagramm: Durchführung der Arbeit

funktionalen Anforderungen gehören besondere Protokollierungseinstellungen, Qualitätsanforderungen und zusätzliche Funktionen für beispielsweise das spätere Monitoring während des Produktivbetriebs.

Der Fokus dieser Arbeit liegt auf der REST-Schnittstelle des Dienstes, deshalb erfüllen die erstellten Nachbauten lediglich den funktionalen Umfang der vom Industriepartner bereit gestellten Implementierung. Sonstige Anforderungen sind eher kosmetischer Natur, werden aber, sofern sie mit niedrigem Aufwand umsetzbar sind, umgesetzt.

Protokoll von REST-Schnittstellen

Fielding weißt in seiner Dissertation mehrmals darauf hin, dass prinzipiell der REST-Architekturstil unabhängig von eingesetzten Protokollen ist [Fie00]. In der Praxis findet man aufgrund der vorhanden Infrastruktur, wie HTTP-Server, HTTP-Bibliotheken und HTTP-Clients aber fast ausschließlich REST-Schnittstellen auf Basis von HTTP. Diese Arbeit wird sich daher ausschließlich mit der Erstellung von REST-Schnittstellen auf Basis von HTTP und anderen Technologien des Internets beschränken.

1.3 Vorgehen

Die Arbeit wurde vom 11.05.2016 bis zum 10.11.2016 durchgeführt. Abbildung 1.1 zeigt eine genaue Übersicht über den Verlauf der Arbeit. Das Projekt lässt sich grob in drei Phasen einteilen: Einarbeitung, Umsetzung und Ergebnis.

Einarbeitung

Haupttätigkeit der Einarbeitung war es das Projekt zu organisieren und die Literaturrecherche durchzuführen. Die Literaturrecherche beschäftigte sich hauptsächlich mit dem Identifizieren von Methoden zum Vergleich unterschiedlicher REST-Schnittstellen und Entwicklungsmethodiken. Die Ergebnisse der Literaturrecherche finden sich zum einem im Vergleich der verschiedenen Ansätze und zum anderem in Kapitel 3 bei den verwandten Arbeiten wieder.

Umsetzung

Die Phase der Umsetzung bestand aus dem Erstellen der Nachbauten, der Evaluierung des Entwicklungsprozesses und dem Vergleich der Methoden. Der Vergleich der Methoden geschah parallel durch mehrere Aktivitäten. Eine der Aktivitäten war die Befragung der Entwickler durch einen speziell dafür angefertigten Fragebogen. Weitere Aktivitäten waren das Zusammenfassen der eigenen Erfahrungen mit den verschiedenen Ansätzen, sowie eine Untersuchung der verschiedenen Ansätze mittels statischer Codeanalyse. Dem allen ging noch eine Zeit der Nachforschungen voraus um eine solide Grundlage auf Basis der gesichteten Literatur zu haben. Auf genauere Details zur Umsetzung wird in Kapitel 5 eingegangen.

Ergebnis

Die letzte Phase des Projekts beinhaltet zum einem die Evaluierung der in der Umsetzung entstandenen Nachbauten, sowie das Verfassen der schriftlichen Ausarbeitung. Zusätzlich wurde dem Industriepartner in dieser Phase auch Rückmeldung über die Qualität der zur Verfügung gestellten REST-Schnittstelle gegeben.

1.4 Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut: Das zweite Kapitel behandelt notwendige Grundlagen um die Arbeit zu verstehen. Kapitel 3 gibt einen Überblick über die verwandten Arbeiten in den verschiedenen betroffenen Themengebieten dieser Arbeit. Das vierte Kapitel beschreibt den Industriepartner, welcher die Schnittstelle zum Vergleich zur Verfügung stellt, sowie das Projektumfeld in welchem die Masterarbeit stattfindet. Kapitel 5 und Kapitel 6 beschäftigen sich mit Methoden und Tools für den Entwurf bzw. für die Realisierung. Der Vergleich der vorgestellten Ansätze wird in Kapitel 7 durchgeführt. In Kapitel 8 wird die Arbeit zusammengefasst wiedergegeben und durch einen Ausblick abgeschlossen.

2 Grundlagen

Dieses Kapitel behandelt Themen, die wichtig für das Verständnis der vorliegenden Arbeit sind. Es soll die Grundlagen für die weiteren Kapitel legen und dem Leser mit Hilfe von Informationen aus Fachliteratur und von den offiziellen Webseiten der eingesetzten Werkzeuge und Methodiken die notwendigen Grundlagen vermitteln.

Das Kapitel teilt sich in mehrere Abschnitte auf, wobei jeder speziell auf ein Thema eingeht. In Abschnitt 2.1 wird auf das Thema Modellierung eingegangen. Im Anschluss daran werden in Abschnitt 2.2 die Grundlagen für Programmierschnittstellen erläutert. Abschnitt 2.3 führt den REST-Architekturstil ein und erklärt diesen. Das als Basis für den Vergleich des Scrum-Prozesses dienende Referenz-Scrum wird in Abschnitt 2.4 vorgestellt.

Die restlichen Abschnitte gehen auf die weiteren Themen der Arbeit ein und erläutern Microservice-Architektur (in Abschnitt 2.6), modellgetriebene Softwareentwicklung (in Abschnitt 2.7) und die für die Arbeit wichtigen modellbasierten Ansätze für REST-Schnittstellen (Abschnitt 2.8).

2.1 Modellierung

Das Wort *"Modell"* ist nicht eindeutig definiert und so auch schwer einzugrenzen. Einige gängige Definitionen beschreiben wichtige Eigenschaften welche einem Modell zugeschrieben werden. Als "Eine in einer klar definierten Sprache geschriebene Beschreibung eines (einem Teil eines) Systems. Äquivalent zu einer Spezifikation." beschreiben Kleppe und andere ihre Definition eines Modells (übersetzt aus [KWB03]). Eine weitere Definition stammt vom *Architecture Board ORMSC*, welches ein Modell als eine Repräsentation eines Teils der Funktion, der Struktur und/oder des Verhaltens eines Systems beschreibt [MM+01]. Miller und andere beschreiben in ihrem *MDA Guide* ein Modell als eine Beschreibung oder Spezifikation des Systems und seiner Umgebung für einen bestimmten Zweck [MM+03].

Wenn man diese drei Definitionen zusammennimmt vermitteln sie ein ganz gutes Bild, was man sich unter einem Modell vorstellen kann. Modelle beschreiben also ein System oder einen Teil davon und nutzen eine fest definierte Sprache um Funktion, Struktur und/oder Verhalten zu beschreiben. Weitere Eigenschaften, welche ein Modell besitzen muss werden von Selic in "The Pragmatics of Model-Driven Development" [Sel03] genannt:

Abstrakt: Ein Modell ist immer eine reduzierte Darstellung des dargestellten Systems. Durch das Entfernen oder Verstecken von für die Ansicht irrelevanten Details wird das Wesentliche sichtbarer. In den immer funktionsfähigeren Softwaresystemen der heutigen Zeit ist Abstraktion der einzige Weg um mit der aus der gestiegenen Funktionalität resultierenden Komplexität umzugehen.

Verständlich: Es reicht nicht nur Details zu abstrahieren, die verbleibenden Details des Modells müssen auch in einer verständlichen Form vorliegen (z.B. einer Notation).

Genau: Ein Modell muss eine der Realität entsprechende Widerspiegelung der darzustellenden Eigenschaften sein.

Prognostisch: Ein Modell muss einsetzbar sein um korrekte Aussagen über die Eigenschaften von Interesse zu treffen. Ein Modell kann für unterschiedliche Aussagen unterschiedlich gut geeignet sein, Selic [Sel03] nennt hier als Beispiel ein mathematisches Modell einer Brücke im Vergleich zu einem Modell gebaut aus Holz - das eine kann gut für die Berechnung der Tragkraft genutzt werden, das andere ist dafür eher schlecht geeignet, ist dafür aber gut geeignet um das Aussehen zu beurteilen.

Nicht zu aufwendig: Das Modell muss signifikant billiger herzustellen und zu analysieren sein als das zu modellierende System.

Ein Modell hat nicht den Anspruch, dass das repräsentierte System tatsächlich existiert. Das beschriebene System kann rein theoretischer Natur sein, oder auch noch nicht existieren. Bei diesen noch nicht existierenden oder auch theoretischen Modellen spricht man von *präskriptiven*, vorschreibenden, Modellen. Weit bekannte Beispiele dafür sind Baupläne, welche vor dem Bau eines Hauses entstehen oder auch eine Aufbauanleitung für ein Möbelstück. Wird ein Modell auf Basis eines bereits existierenden Systems erstellt, so spricht man von einem *deskriptiven*, also beschreibenden, Modell. Eine Karte ist zum Beispiel ein gutes Beispiel für ein eindeutig deskriptives Modell einer Stadt.

Oft ist die Trennung zwischen diesen beiden Modellarten aber nicht möglich, da für eine genaue Bestimmung immer die Entstehungsgeschichte bekannt sein muss. Wenn man beispielsweise einen Kabelplan für ein Stockwerk findet, so weiß man nicht ob es sich dabei um ein präskriptives Modell oder ein deskriptives Modell handelt. Wurde er zur zur Planung der Verkabelung erstellt, so wäre er ein präskriptives Modell, wenn er aber als Dokumentation der bereits verlegten Kabel erstellt wurde, so handelt es sich um ein eindeutig deskriptives Modell. Wie man sieht ist eine Aussage ohne die Herkunftsgeschichte zu kennen nicht immer möglich.

2.2 Programmierschnittstellen

Das Wort Schnittstelle ist in der Informatik sehr vieldeutig. Es gibt viele unterschiedliche Themen welche mit diesem Sammelbegriff in Verbindung gebracht werden. Neben Hardwareschnittstellen, Benutzerschnittstellen und Schnittstellen in der objektorientierten Programmierung gibt es noch viele weitere Arten von Schnittstellen. Sie alle haben gemeinsam, dass sie den Zugriff auf Ressourcen abstrahieren und die spezifischere Implementierung gegenüber dem Nutzer der Schnittstelle verstecken. Im Englischen wird für Schnittstellen das Wort *Interface* benutzt, wir wollen in der Arbeit auf eine besondere Art der Schnittstellen eingehen, auf die sogenannten Application Programming Interfaces (APIs). Sie beschreiben eine Menge von Methodendefinitionen, Protokollen und Werkzeuge welche zusammen genutzt werden können um auf Programme und Anwendungen zuzugreifen.

Wenn ein Programm Anwendern oder andere Programmen seine Funktionalität zur Verfügung stellt, so wird es meist als Dienst (engl. *service*) bezeichnet. Um die Funktionalität zur Verfügung zu stellen wird eine Schnittstelle zwischen dem anbietendem Dienst und dem Nutzer des Dienstes benötigt.

Bei der Entwicklung von Schnittstellen gibt es in der Regel einen Kontrakt, welcher verbindlich die Schnittstelle beschreibt. Ein Schnittstellenanbieter verpflichtet sich seine Schnittstelle entsprechend des Kontrakts implementiert zu haben. Mit der Schnittstellendefinition bzw. Beschreibung kann ein Nutzer eines Dienstes sich so sicher sein, dass der Zugriff auf die Schnittstelle wie im Kontrakt beschrieben abläuft.

Bei der Erstellung der beschriebenen Kontrakte und Schnittstellen gibt es zwei Arten von Ansätzen. Der erste Ansatz wird *top-down*-Ansatz genannt. Er beschreibt die Entwicklungsrichtung ausgehend vom Abstraktionslevel, von oben nach unten bedeutet hier dementsprechend, dass zuerst das abstraktere - also der Kontrakt - entwickelt wird. Bei einer *top-down*-Entwicklung wird also zuerst der Kontrakt geschrieben, bevor mit der eigentlichen Entwicklung begonnen wird. Dieser Ansatz kommt vor allem bei vertraglich gesicherten Entwicklungen, unabhängigen Entwicklungen von Server- und Nutzeranwendungen und Ablösung von bereits existierenden Schnittstellen zum Einsatz.

Beim zweiten Ansatz spricht man von einer *bottom-up*-Entwicklung. Hier wird die Implementierung durchgeführt und während der Entwicklung werden Funktionen der Anwendung für andere Anwender offen gelegt indem eine Schnittstelle bereitgestellt wird. Ein gängiger Ansatz ist hierbei, dass der während der Implementierung entstehende Code genutzt wird um einen Kontrakt für die Nutzer der Schnittstelle zu generieren. Oftmals gibt es bei diesem Ansatz die Möglichkeit den Kontrakt durch Verwendung von Annotationen im Quellcode genauer zu machen.

REST-Schnittstellen bilden einen wichtigen Kern der Arbeit. Sie gehören zu den Webschnittstellen auf Basis des HTTP-Protokolls. Webschnittstellen haben gemeinsam, dass sie einen oder mehrere öffentlich verfügbare Endpunkte besitzen. Sie übertragen in den meisten

Fällen ihre Informationen mittels Extensible Markup Language (XML) oder JavaScript Object Notation (JSON). Weitere bekannte HTTP-Schnittstellenarten sind SOAP, XML-RPC und viele weitere. Auf den REST-Architekturstil wird im nächsten Abschnitt noch genauer eingegangen werden.

2.3 REST

Bei REST handelt es sich um einen Architekturstil für verteilte Hypermedia Systeme. Er wurde von Roy Fielding in seiner Dissertation mit dem Titel "Architectural Styles and the Design of Network-based Software Architectures" definiert [Fie00]. Die Definition umfasst folgende Menge von architektonischen Regeln (*constraints*), welche, wenn als Ganzes eingehalten, positive Eigenschaften für die zu entwickelnde Anwendung mit sich bringen.

Eine dieser Regeln ist der *Client-Server*-Stil, er soll genutzt werden um eine Trennung von Belangen (*Separation of concerns*) zu erreichen. Der Client-Server-Stil ist ein Konzept in verteilten Anwendungen für die Aufgabenverteilung innerhalb eines Netzwerks. Dabei existieren die Rollen *Server*, welcher Dienste oder Ressourcen anbietet, und *Client*, welcher die angebotenen Dienste oder Ressourcen nutzt.

Die Regel der *Zustandslosigkeit* (statelessness) bedeutet, dass der Zustand einer Anwendung vollständig auf Seite des Clients gehalten werden muss und, sofern notwendig, bei jeder Anfrage mit übertragen werden muss. Durch diese Regel ist ein Client nun nicht mehr an einen Server gebunden, da jede Anfrage mit den beinhalteten Daten verarbeitet werden kann. Die Zustandslosigkeit führt zu den positiven Eigenschaften Sichtbarkeit, Zuverlässigkeit und Skalierbarkeit. Was man unter diesen Eigenschaften versteht und wie diese genau durch die Zustandslosigkeit erreicht werden ist in den folgenden Absätzen beschrieben.

Unter Sichtbarkeit versteht man die Möglichkeit die Kommunikation zwischen Servern und Clienten zu überwachen und zu vermitteln. Diese Eigenschaft wird später in diesem Kapitel nochmals aufgegriffen, wenn es um die einheitliche Schnittstelle geht. Die Zustandslosigkeit ermöglicht es bei Fehlern in der Kommunikation lediglich den fehlgeschlagen oder betroffenen Aufruf selbst betrachten zu müssen. Der Aufruf beinhaltet alle zur Analyse notwendigen Informationen.

Die eben beschriebene Eigenschaft, dass eine Nachricht alle notwendigen Information enthält verbessert auch die Zuverlässigkeit des gesamten Systems bei teilweisen Ausfällen. Beim Betrieb mit mehreren Servern ist es so möglich, im Falle eines Serverausfalls, die selbe Anfrage an gleichwertigen anderen Server (z.B. einen gespiegelten Server) zu stellen. Aufgrund der vollständigen Informationen in der Anfrage, kann der Server diese dann entgegennehmen und beantworten.

Diese Vermittlung kann auch gezielt zur Skalierbarkeit von Anwendungen beitragen. Serveranwendungen welche den Zustand auf dem Server speichern benötigen mehr Speicher, da sie Informationen über mehrere Anfragen hinweg speichern müssen. Diese sogenannten Sessions können bei einer zustandslosen Kommunikation schneller wieder freigegeben werden.

Eine weitere Regel ist die *Cache*-Regel, welche der Implementierung auferlegt, dass für alle Antworten explizit oder implizit definiert sein muss ob diese zwischenspeichert werden können. Diese Einschränkung führt dazu, dass Anfragen bereits auf ihrem Weg durch das Netzwerk aus einem Zwischenspeicher beantwortet werden können. Eine Anwendung wird durch diese Einschränkung also effizienter, besser skalierend und die durch den Anwender wahrgenommene Latenz sinkt, da viele Anfragen erst gar nicht an den eigentlichen Server gestellt werden müssen.

Der Architekturstil fordert zusätzliche eine *einheitliche Schnittstelle* zwischen den einzelnen Komponenten. Die Umsetzung dieser Einschränkung führt zu einer Vereinfachung der Architektur des Gesamtsystems und einer Verbesserung der Sichtbarkeit von Interaktionen. REST ist durch vier Schnittstellen Einschränkungen definiert:

Identifikation von Ressourcen: In Anfragen werden einzelne Ressourcen adressiert, dies geschieht in webbasierten REST-Schnittstellen durch den Einsatz von Uniform Resource Identifiers (URIs). Konzeptionell sind dabei Ressourcen und Darstellungen strikt von einander getrennt. Eine Ressource kann beispielsweise eine Darstellung in XML, JSON oder Hypertext Markup Language (HTML) anbieten.

Selbstbeschreibende Nachrichten: Jede Nachricht enthält genug Informationen um herauszufinden wie sie verarbeitet werden kann. Die Wahl des passenden Parsers für die Antwort kann beispielsweise durch die Angabe eines *Content-type*-Header angegeben werden.

Manipulation von Ressourcen durch Darstellungen (representations): Wenn ein Client eine Darstellung einer Ressource inklusive der Metadaten hat, so kann er mit diesen Informationen diese Ressource ändern und löschen.

Hypermedia: Ein Client macht Zustandsübergänge nur über Aktionen welche dynamisch durch Antworten, z.B. in Form von Hypermedia, vom Server identifizert wurden. Außer den fixen Einstiegspunkten trifft der Client keine Annahmen über verfügbare Aktionen irgendwelcher Ressourcen, außer der in den bisherigen Antworten des Server enthaltenen. Dieses Prinzip wird auch als “Hypermedia as the Engine of Application State (HATEOAS)” bezeichnet.

Die *Layered System*-Einschränkung zwingt dazu, dass Komponenten lediglich mit ihren angrenzenden Systemschichten interagieren können und keine weiteren Systeme außerhalb davon kennen. Sogenannte Intermediäre Systeme (*Intermediaries*) dienen so zur Limiterung der Komplexität von Systemen und zur Kapslung der Komponenten. Darüberhinaus

ermöglichen sie verbesserte Skalierbarkeit, beispielsweise mit dem Einsatz von Lastverteilungssystemen (*Load-Balancers*).

Allamaraju fasst in seinem Buch "RESTful Web Services Cookbook" [All10] treffend zusammen, dass REST ein Architekturstil für vernetzte Anwendungen ist, welcher die oben genannten Einschränkungen nutzt und zusammen mit dem HTTP-Protokoll und der Infrastruktur des Internets eine attraktive Möglichkeit darstellt Dienste zu implementieren.

Umsetzung der REST-Prinzipien

Bei der Umsetzung einer HTTP-REST-Schnittstelle wird die Anforderung an einen einheitlichen Zugriff mittels einer einheitlichen Schnittstelle durch die Verwendung des HTTP Protokolls erfüllt. HTTP bietet dabei für jede Ressource wohldefinierten Methoden. Die meist genutzten Methoden sind: *POST*, *GET*, *PUT*, *PATCH* und *DELETE*. Die weiteren Methoden *OPTIONS* und *HEAD* sind für die Beschreibung von REST-Schnittstellen weniger relevant, weil sie eher im Hintergrund genutzt werden um Anforderungen wie Caching umzusetzen.

Um Ressourcen zu verwenden werden die benötigten Operationen auf die Methoden des Protokolls übersetzt. Dabei müssen immer auch die Eigenheiten des unterliegenden Protokolls beachtet werden. Betrachtet man die Methodendefinition der HTTP-Protokollspezifikation [FGM+99] genauer, so fällt auf dass diese den oben genannten Methoden unterschiedliche Eigenschaften zuweist.

Die zwei wichtigsten Eigenschaften sind dabei *Sicherheit* und *Idempotenz* der Methoden. Sichere Methoden stellen den Anspruch, dass ein Aufruf ihrer keine Änderungen an der Ressource zur Folge hat. Dieses Verhalten ist Grundvoraussetzung für viele unterliegende Protokollvorteile wie z.B. für das Caching von Ressourcen. Zu den sicheren Methoden zählen *GET*, *HEAD* und *OPTIONS*. An dieser Stelle sei angemerkt, dass dies nicht vom Protokoll erzwungen werden kann. Ein Dienst kann schlecht implementiert sein, so dass sichere Operationen dennoch Nebeneffekte erzielen. Dieses Verhalten kann aber im Zusammenspiel mit anderen Komponenten zu Problemen führen, wenn diese von einer korrekten Umsetzung des HTTP-Standards ausgehen und sich auf die Sicherheit der Methoden verlassen. Die andere Eigenschaft ist die Idempotenz. Idempotente Methoden sind Methoden, welche mehrmals aufgerufen werden können und dennoch dasselbe Ergebnis zur Folge haben. Zu den idempotenten Methoden zählen *OPTIONS*, *GET*, *HEAD*, *PUT*, *DELETE* und *PATCH*. Abbildung Tabelle 2.1 zeigt eine Übersicht über die Sicherheit und Idempotenz der HTTP-Methoden.

Basierend auf den Gegebenheiten des HTTP-Protokolls ist die folgende Verwendung der Methoden gegeben. Der RFC2616 [FGM+99] gibt die Anweisung nach der Identifizierung und dem Entwurf von Ressourcen die *GET*-Methode zu nutzen um eine Repräsentation der Ressource anzufordern. Die *PUT*-Methode wird genutzt um Änderungen an einer Ressource

Tabelle 2.1: Idempotenz und Sicherheit der HTTP-Methoden nach Allamaraju [All10]

HTTP-Methode	Idempotenz	Sicherheit
OPTIONS	ja	ja
GET	ja	ja
HEAD	ja	ja
PUT	ja	nein
DELETE	ja	nein
POST	nein	nein
PATCH	nein	nein

Tabelle 2.2: Umsetzung des CRUD-Musters mittels REST

CRUD Operation	HTTP Methode	Anmerkung
CREATE	POST	
READ	GET	
UPDATE	PUT	Implementierung muss idempotent sein!
DELETE	DELETE	

vorzunehmen. Um potentiell nicht idempotente und unsichere Operationen auszuführen soll die *POST*-Methode genutzt werden. Darüberhinaus wird die Verwendung von passenden HTTP-Headern, um Anfrage und Antwort zu beschreiben, definiert.

Das "RESTful Web Services Cookbook" [All10] liefert ein Beispiel für eine mögliche korrekte Verwendung des HTTP-Protokolls. Das Beispiel ist eine Umsetzung des gängigen Musters "Erstellen", "Lesen", "Ändern""Löschen" (zu Englisch: "create", "read", "update" and "delete" - *CRUD*) auf die Methoden des HTTP-Protokolls. Eine mögliche Umsetzung ist in Abbildung Tabelle 2.2 dargestellt. Wie man erkennen kann wird die abstrakte Operation "Lesen" mittels der *GET*-Methode, die "Erstellen" Operation mittels der *POST*-Methode, die "Ändern" Operation mittels der *PUT*-Methode und die "Löschen"-Operation mittels der *DELETE*-Methode. Durch diese allgemein anerkannte Mapping können viele Schnittstellen bereits weitgehend ohne die Verwendung einer zusätzlichen Dokumentation benutzt werden.

Über diese Art der Umsetzung von REST-Diensten sind sich die meisten Entwickler und Experten einig, bei anderen Themen haben sich aber zwei Lager gebildet. Auf der einen Seite befinden sich die Puristen, welche ihre REST-Webdienste streng nach den Vorgaben von Roy Fielding erstellen. Im Kontrast dazu gibt es einige Pragmatiker, welche diese Prinzipien nicht voll umsetzen oder bewusst aufweichen um ihre Schnittstellen praktischer zu erstellen. Durch diesen Konflikt kommt es dazu dass viele, wenn nicht sogar die meisten, als REST-API betitelten Schnittstellen die Prinzipien von REST nicht beherzigen oder nicht voll umsetzen.

Fielding beschreibt diesen Missstand in seinem Blogbeitrag “REST APIs must be hypertext-driven” [Fie08], in dem er anklagt wie oft vor allem die Anforderung von HATEOAS verletzt wird und dennoch die jeweiligen Schnittstellen als REST-Schnittstellen bezeichnet werden. Er fordert dazu auf, dass diese Schnittstellen doch ein anderes Buzzword als REST für ihre Bezeichnung nutzen sollen. In der Praxis wird dieser Bitte nicht nachgegangen - deshalb wird auch in dieser Arbeit der aufgeweichte Begriff der REST-Schnittstellen als Maßstab genommen.

2.4 Scrum

Agile Vorgehensmodelle sind geschichtlich aus Problemen der klassischen Projektdurchführung im Softwareumfeld entstanden. Lange Zeit wurden Softwareprojekte ähnlich zu anderen Ingenieursprojekten mit einer langen initialen Planungsphase und anschließender Umsetzungsphase durchgeführt. Beispielhafte Projekte dafür sind der Bau eines Schiffes, die Konstruktion einer Brücke oder der Bau eines Hauses. Dieses lineare nicht inkrementelle Vorgehensmodell wird im Allgemeinen auch Wasserfallmodell genannt. Ursprünglich wurde diese Art von Vorgehensmodell, wenn auch nicht unter dem Namen “Wasserfallmodell”, das erste Mal von Royce in seiner Arbeit “Managing the Development of large Software Systems” [Roy70] vorgestellt.

Der Name “Wasserfallmodell” kommt von der linearen Natur und der Tatsache, dass die Phasenergebnisse als Basis für die jeweils nächsttiefere Phase dienen. Ähnlich wie bei einem Wasserfall ist kein (Informations-)Fluss entgegen der eigentlichen Richtung möglich. Abbildung 2.1 zeigt die einzelnen Phasen des Wasserfallmodells. Projekte bestehen in diesem Modell aus den klar definierten Phasen: Systemanalyse, Softwarespezifikation, Architekturentwurf, Feinentwurf und Codierung, Integration und Test, Installation und Abnahme und zuletzt der Betrieb und Wartung. Jede Phase wird hierbei von einem sogenannten Meilenstein beendet, wessen Kriterien und Ergebnisdokumente für die Abnahme der Phase erfüllt oder erstellt sein müssen. Es eignet sich besonders für Projekte, bei denen bereits in der Planungsphase sehr präzise Anforderungen, Leistungen und Abläufe beschrieben werden können.

In der Softwareentwicklung kam es bei linearen nicht-agilen Projekten immer wieder zu Problemen: In einer Studie [Kom12] mit 457 Befragten, davon 375 aus agilen Projekten und 82 aus Projekten mit klassischem Projektmanagement, hat Komus zeigen können, dass die Anwender klassischer, nicht agiler, Methoden den Erfolg ihre Projekte signifikant schlechter einschätzen.

Scheinbar lässt sich die Durchführung eines Softwareprojektes im Vergleich zu klassischen Ingenieurstätigkeiten weniger vorrausschauend planen. In der Literatur wird Software oftmals als *“ausführbares Wissen”* beschrieben. Hier sieht Armour auch eins der Probleme

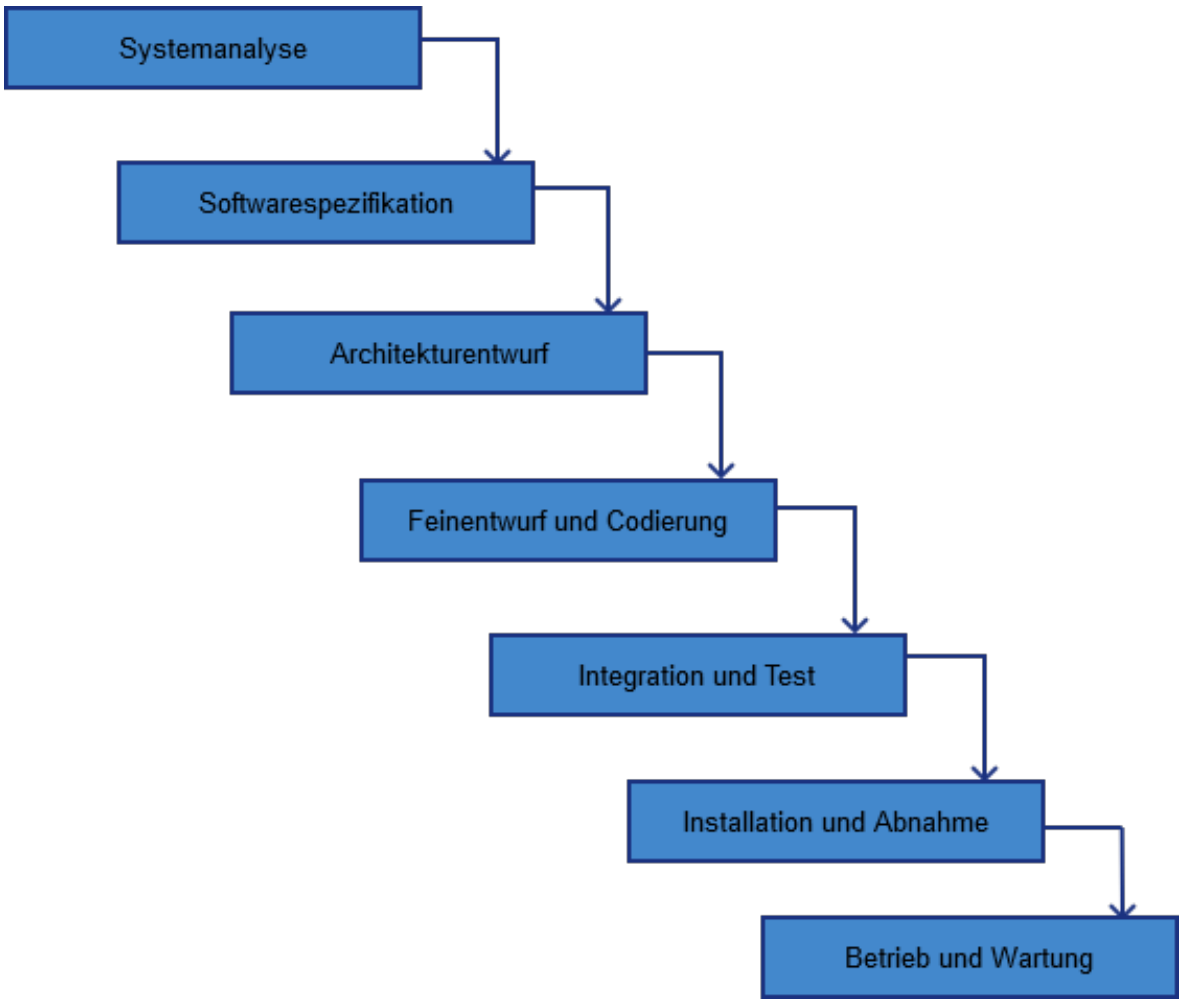

Abbildung 2.1: Das spätere Wasserfallmodell nach Royce [Roy70]

bei der Anwendung des klassischen Projektmanagements auf Softwareprojekten. Er weist in seinem Buch “The Laws of Software Process” [Arm03] daraufhin, dass eine Entwicklung von solch “ausführbarem Wissen” vor allem eine auf Entdeckung basierte Tätigkeit ist. Solche Entdeckungs- und Forschungsaktivitäten lassen sich nicht genauso planen wie sich wiederholende Aufgaben oder Routineaufgaben. Seine Empfehlung ist deshalb rigoros alle definierbaren Tätigkeiten durchzuplanen sich aber bewusst zu sein, dass dies nicht für alle Tätigkeiten in einem Softwareprojekt möglich ist. Deshalb hält er es für notwendig gezielt Möglichkeiten einzuplanen auf diese Unsicherheiten zu reagieren.

Agile Vorgehensmodelle haben ihren Ursprung in den 90er Jahren. Die Verwendung des Wortes agil für die Bezeichnung der Art der Softwareentwicklung wurde auf einer Zusammenkunft in Utah geprägt, das Treffen auf dem auch das bekannte “Agile Manifesto” [BBV+01] entstanden ist. Es fasst bis heute die unterliegenden Prinzipien von agiler Soft-

wareentwicklung zusammen. Die Teilnehmer dieser Versammlung hielten fest, dass obwohl sie alle folgenden Werte für wichtig erachten dennoch einige für wichtiger empfinden. Sie empfinden:

Individuen und Interaktionen wichtiger als Prozesse und Werkzeuge,

Funktionierende Software wichtiger als umfassende Dokumentation,

Zusammenarbeit mit dem Kunden wichtiger als Vertragsverhandlung,

Reagieren auf Veränderung wichtiger als das Befolgen eines Plans.

Eins der bekanntesten agilen Vorgehensmodelle ist Scrum. Wie alle agilen Vorgehensmodelle, so verfolgt auch Scrum diese eben genannten Prinzipien. Scrum hat seine Wurzeln in der Softwareentwicklung und wird deshalb dort sehr häufig eingesetzt - ist es ist aber prinzipiell in jeder Projektart einsetzbar. In der Softwareentwicklung ist Scrum eins der am häufigst eingesetzten agilen Vorgehensmodelle.

Bei Scrum handelt es sich um ein iteratives und inkrementelles Vorgehensmodell. Das bedeutet, dass bei Scrum zum einem durch immer wiederkehrendes Einfließen von Feedback das Produkt verbessert wird und zum anderem das Produkt schrittweise entwickelt wird. Eine Iteration in Scrum nennt sich *Sprint*, die Länge eines Sprints ist in der Regel zwischen zwei und vier Wochen. Jeder dieser Sprints muss aber ein funktionsfähiges, wenn auch funktionsarmes, Produkt zur Folge haben.

Sutherland und Schwaber erklären auf ihrer *Scrum Guides Webseite*[1] wie sie sich genau das Vorgehensmodell vorstellen - dieser Guide kann wohl als Definition von Scrum in seiner reinsten Form angesehen werden. Schwaber hat in einem Interview gesagt, dass Scrum viele Schwächen und Unzulänglichkeiten in den Produkt- und Systementwicklungspraktiken von Unternehmen aufzeigt, und es ihnen ermöglicht diese auszumerzen Seiner Ansicht nach passiert dieses Ausmerzen in der Praxis selten, viel zu oft modifizieren Unternehmen Scrum um diese Schwächen und Unzulänglichkeiten aufzunehmen [Jai].

Es gibt aber auch Unternehmen, die nach einer erfolgreichen Einführung von Scrum entdecken, dass Scrum nicht zu 100%ig auf ihre Bedürfnisse passt. Sie modifizieren Scrum dann um ihre Abläufe zu optimieren. Grund für solche geplanten Modifikationen können beispielsweise bestehende Hierarchien oder bestehende Arbeitsabläufe sein. Wie Diebold et al. erkannt haben, ist es schwer zu sagen, wann und ob man verschiedene Aspekte von Scrum anpassen sollte [DOWZ15]. Um die Vergleichbarkeit zu sichern und die Ergebnisse dieser Arbeit möglichst allgemein anwendbar zu machen, wird ein Teil dieser Arbeit eine kurze Betrachtung des beim Industriepartner praktizierten Scrums sein. Diese genaue Betrachtung findet man in Kapitel 4. Folgend wird das Referenzvorgehensmodell nach Sutherland und Schwaber [Jee], welches Grundlage für den späteren Vergleich ist, kurz beschrieben.

[1] Scrum Guides: http://scrumguides.org/

Scrum basiert auf Empirie, was bedeutet, dass Wissen aus Erfahrung gewonnen wird. Bereits Bekanntes dient dabei als Basis für Entscheidungen. Scrum nutzt einen iterative, inkrementellen Ansatz um Sicherheit in Vorhersagen zukünftiger Termine und Ergebnisse zu optimieren. Die drei wichtigsten Säulen von Scrum sind Transparenz, Überprüfung und Anpassung. Scrum versteht unter Transparenz, dass regelmäßig Fortschritt und Hindernisse sichtbar festgehalten werden. Die Überprüfung bezieht sich nicht nur auf das gefertigte Produkt, sondern auch auf den Enstehungsprozess welcher bei Scrum auch regelmäßig beurteilt werden soll. Die Anpassung ist der agile Anteil in Scrum: Statt einer einmaligen Festlegung aller Anforderungen, Vorgehen und Pläne werden diese kontinuierlich detailliert und angepasst. Große Aufgaben werden so gezielt in kleiner Schritte zerlegt.

Scrum kennt drei Rollen, welche das *Scrum Team* ausmachen: Den *Scrum Master (SM)*, den *Product Owner (PO)* und das *Entwicklungsteam*. Ein Scrum Team ist selbstorganisierend und interdisziplinär, die Teammitglieder entscheiden also zusammen selbst wie sie ihre Arbeit erledigen und verfügen über alle Kompetenzen um diese zu erledigen.

Die einzelnen Rollen haben sehr unterschiedliche Aufgaben. Der *SM* sorgt im Wesentlichen dafür, dass die Regeln von Scrum eingehalten werden. Er ist für das Verständnis und die Durchführung von Scrum im Team verantwortlich. Der *PO* ist eine einzelne Person und definiert die Aufgaben des Entwicklungsteams. Er ist außerdem für die Arbeit des Entwicklungsteams verantwortlich. Seine Aufgabe ist die Wertmaximierung des Produkts und die Priorisierung und Definition der zu erledigenden Aufgaben verantwortlich. *Das Entwicklungsteam* besteht aus Entwicklern - einen anderen Titel gibt es in Scrum nicht. Es gibt auch keine weitere Unterteilung zwischen verschiedenen Aufgabenbereichen. Das Entwicklungsteam ist für die Umsetzung der vom PO definierten Aufgaben verantwortlich, dabei darf es selbst entscheiden auf welche Weise es die Aufgaben umsetzen will. Neben Rollen gibt in Scrum folgende Artefakte: *Product Backlog*, *Sprint Backlog* und das *Inkrement*. Das *Product Backlog* ist eine Liste, welche alle möglichen Features, Funktionalitäten, Verbesserungen und Fehlerbehebungen für zukünftige Releases beinhaltet. Es ist ein nie vollständiges, dynamisches Dokument und kann deshalb vom PO jederzeit ergänzt, umsortiert und bereinigt werden. Product Backlog Einträge sind nach Priorität sortiert und enthalten zusätzlich eine Beschreibung, eine Schätzung über den Aufwand und einen Wert. Das *Sprint Backlog* ist das Equivalent für die Dauer des Sprints. Es enthält eine Teilmenge der Product Backlog Einträge, welche im Sprint umgesetzt werden. Es ist eine Prognose des Entwicklungsteams, was sie in dem jeweiligem Sprint leisten wollen - also welche Funktionalität das nächste Inkrement beinhalten wird.

Das *Inkrement* ist das Ergebnis eines Sprints und setzt sich aus den Teileinträgen der fertiggestellten Product Backlog-Einträge zusammen. Das Inkrement muss am Ende eines Sprintes einen “Done”-Zustand erreicht haben. Es muss also in einem verwendbaren Zustand sein und die vorher definierten Abnahmekriterien müssen erfüllt worden sein. Es muss auch auslieferbar sein, selbst wenn der PO eine Auslieferung noch gar nicht plant.

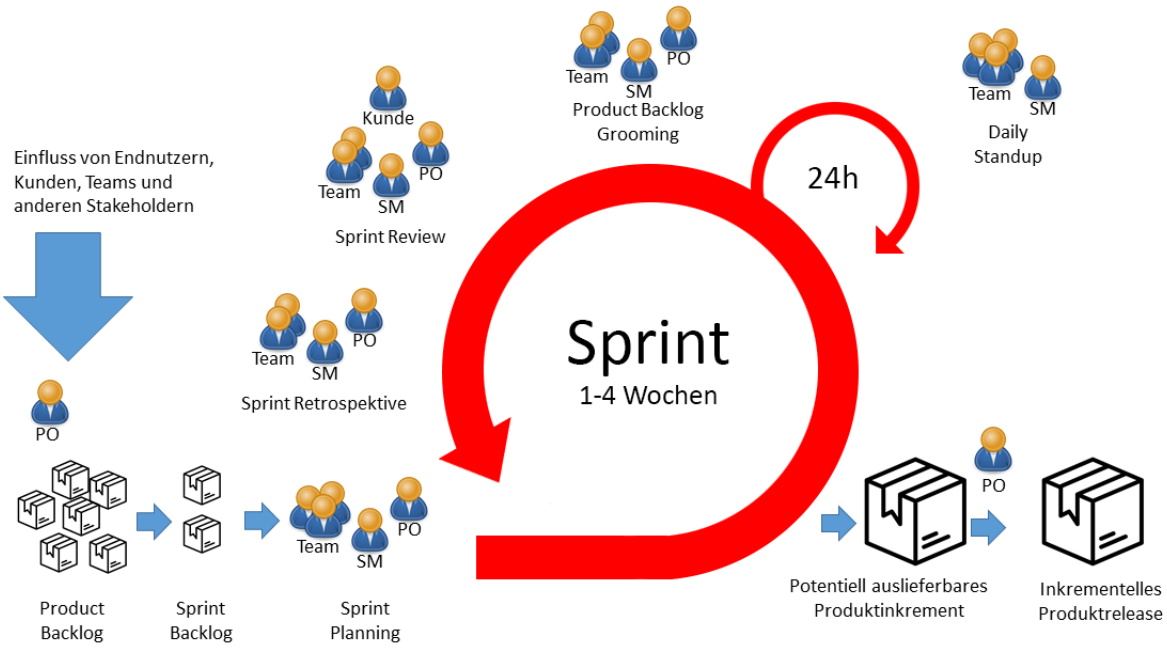

Abbildung 2.2: Ablauf Referenz-Scrumprozess nach Scrum Inc. [Scr]

Abbildung 2.2 zeigt den Ablauf eines typischen Scrumprozesses. Die Abbildung soll genutzt werden um die Ereignisse im Scrumprozess zu erklären. Das Vorgehensmodell ist ein sich wiederholender Prozess, welcher auf dem Product Backlog beruht. Der Product Owner ist wie bereits beschrieben für den Inhalt und die Priorisierung des Product Backlogs verantwortlich (siehe Punkt 1). Grundlage für Einträge in das Product Backlog können Einflüsse verschiedener beteiligter des Projektes sein, wie zum Beispiel Endanwender, Kunden, das Team selbst oder andere Personen.

In den nächsten Schritten entsteht das Sprint Backlog (siehe Punkt 2) im Sprint Planning (Schritt 3). Das Team entscheidet im Sprint Planning zusammen mit dem Product Owner, was nächsten Sprint umgesetzt werden wird. Dann beginnt der Sprint, welcher 1-4 Wochen dauert. Während des Sprints gibt es jeden Tag das Daily Standup (siehe Punkt 4). Es ist Teil des täglichen Arbeitens und dient dem Team sich auszutauschen. Nach dem Daily Standup sollten im ganzen Team die gestrigen und tagesaktuellen Tätigkeiten und Hindernisse aller Entwickler bekannt sein. Teil des Sprints sind auch die Punkte 5-7, Product Backlog Grooming, das Sprint Review und die Sprint Retrospective. Bei dem Product Backlog Grooming handelt es sich um eine Aktivität bei der PO Feedback zum Product Backlog einholen kann um die Product Backlog Einträge bereits vor dem nächsten Sprint Planning ausreichend zu zerlegen. Dies ermöglicht, dass die Product Backlog Einträge definiert genug sind um in den nächsten Sprint mit aufgenommen zu werden. Das Sprint Review wird genutzt um festzustellen welche Product Backlog Items im Sprint umgesetzt worden sind. Neben der

reinen Abnahme durch den PO wird hier auch die Auswirkung des Sprints auf den Release Plan diskutiert. Die letzte Aktivität ist die Sprint Retrospective, welche der langfristigen Verbesserung der Prozesse dient. In der Sprint Retrospective findet das Team gemeinsam Erfolge und Misserfolge des Sprints und arbeitet durch Disskusionen mögliche Verbesserungen heraus. Die Verbesserungen sollen dann in das Team einfließen, indem während des Meetings für jede Verbesserung ein Verantwortlicher bestimmt wird, der die Umsetzung der Verbesserung vorantreibt und überwacht. Das Ergebnis des Sprints ist ein potentiell auslieferbares Produktinkrement (siehe Punkt 8), welches der PO, falls er will, ausrollen kann (siehe Punkt 9).

2.5 Statische Codeanalyse

Oftmals werden Programme durch verschiedene Verfahren getestet und validiert. In der Praxis sind dabei in der Entwicklung die Modultests (sog. *Unit-Tests*) die Art der am häufigst ausgeführten Tests. Bei geeigneter Testabdeckung stellen sie korrekte Ausgaben der getesteten Module sicher und schützen vor dem versehentlichen Einfügen von neuen Fehlern. Neu eingeführte Logik kann aber auch zu verminderter Wartbarkeit führen und so zukünftige Änderungen erschweren. Im Projektumfeld hat sich für dieses Phänomen der Begriff technische Schuld (*technical debt*) eingebürgert. Sie beschreibt in welchem Umfang Arbeit an der bisherigen Implementierung notwendig ist um sie von den angesammelten Mängeln zu befreien. Während der Entwicklung einer Software muss oft zwischen Entwicklungsgeschwindigkeit und technischer Schuld abgewogen werden. Ein weiterer häufig genutzter Begriff in diesem Zusammenhang sind die *"Code Smells"*. Code Smells sind dabei ein von Kent Beck geprägter und von Martin Fowler in seinem Buch "Refactoring: Improving the Design of Existing Code" [Fow09] bekannt gemachter Begriff, den Fowler dort als "sichtbare Symptome im Quellcode, welche auf tiefere Probleme hinweisen" beschreibt.

Ein Mittel um einen Überblick über die technische Schuld eines Projektes zu erhalten ist die statische Codeanalyse. Im Unterschied zu dynamischen Analysen, welche das zu testende Programm während der Ausführung untersuchen, untersuchen statischer Codeanalysen lediglich den vorliegenden Programmquellcode. Der Begriff statische Codeanalyse wird weitläufig für werkzeugunterstützte Tätigkeiten verwendet, auch wenn man Tätigkeiten wie das Code Review ebenso in diese Kategorie einordnen könnte. Statische Analyseverfahren werden in der Softwarentwicklung häufig eingesetzt um häufige Fehler zu erkennen und zu beseitigen.

Zheng und andere haben in ihrer Arbeit *"On the value of static analysis for fault detection in software"* [ZWN+06] anhand eines Beispiels aufgezeigt, dass eine statische Codeanalyse nicht alle Fehler finden kann und findet und darüber hinaus auch viele sogenannte *false positives*, also Treffer welche eigentlich keine sind, finden. Sie konnten jedoch zeigen, dass statische Codeanalyse einen guten Beitrag zum Auffinden und Identifizieren von Fehlern leisten und

Ergebnisse der Analyse ein guter Indikator - auch zum Vergleich verschiedener Module - sind. Aus diesem Grund wird diese Arbeit statische Codeanalysen einsetzen um eine weitere, zusätzlich zum Feedback der Entwickler, Möglichkeit zum Vergleich der unterschiedlichen Lösungen zu haben.

2.6 Microservice-Architektur

Sam Newman beschreibt Microservices als kleine, miteinander kommunizierende, autonome Dienste [New15]. Microservices sollen laut ihm klein sein und sich darauf beschränken nur eine Funktion gut umzusetzen (*'Focused on Doing One Thing Well'*). Auf die Frage nach dem tatsächlichen Umfang bleibt er vage. Er schreibt allerdings, dass Entwickler generell ein gutes Gefühl für die Antwort auf die Frage ob ein Dienst zu groß ist - er rät daher einen Dienst nach Möglichkeit solange zu verfeinern und einzugrenzen, bis dieses Gefühl verschwindet. Eine weitere gefühlgetriebene Entscheidungsmöglichkeit kommt von Jon Eaves von RealEstate.com.au[2], welcher einen Microservice als einen Dienst beschreibt, welcher in zwei Wochen neu geschrieben werden kann. Bei Microservices spricht man also wenn man vom Umfang spricht meist von der fachlichen Funktionalität.

Fowler und Lewis haben eine noch genauer Definition des Begriffes "Microservice". Sie definieren einen Microservice kurzgefasst als einen Ansatz um eine einzelne Anwendung als eine Menge von kleinen Diensten zu implementieren, welche alle in eigenen Prozessen laufen und mittels leichtgewichtigen Mechanismen (meistens über HTTP Schnittstellen) miteinander kommunzieren. Diese kleinen Dienste sind unabhängig von einander von vollautomatischen Deploymentwerkzeugen ausrollbar. Die Dienste benötigen ein Minimum an zentraler Verwaltung und können sehr unterschiedliche Technologien einsetzen (z.B. unterschiedliche Programmiersprachen oder verschiedene Speichertechnologien) [FL].

Die Microservice-Architektur bietet einige allgemein anerkannte [Bad; Gol; Kum; LF; New15; Ste; Wol] Vor- und Nachteile wenn man sie mit der klassischen Architektur groß gewachsener Systeme vergleicht. Ein Microservice ist vom Umfang her, wie der Name sagt, eher klein und kann daher, im Vergleich zu einem größeren System, mit geringerem Aufwand ersetzt werden. Die kleine Größe bietet aber außerdem noch den Vorteil, dass der Dienst von neuen Entwicklern schneller zu verstehen ist und generell kleinere Teams eingesetzt werden können. Das Einsetzen von kleineren Teams kann ggf. den Kommunikationsaufwand verringern.

Durch die strikte Trennung der Funktionalitäten in mehrere Dienste wird auch verhindert, dass sich mit der Zeit Beziehungen zwischen Klassen und Funktionalitäten einschleichen. Da die Kommunikation zwischen den Diensten nur über die bereitgestellten Schnittstellen geschieht kann ohne bewusste Schnittstellenanpassung eine solche Änderung gar nicht erst

[2]RealEstate: http://RealEstate.com.au

durchgeführt werden. Langfristig ist es deshalb so einfacher eine nachhaltige Architektur aufrecht zu erhalten. Ein weiterer Vorteil der Aufteilung ist, dass jeder Dienst theoretisch mit einem anderem Technologie-Stack umgesetzt werden könnte. Durch diese Freiheit in der Wahl der Technologie, kann man für jede Funktionalität das richtige Werkzeug nutzen.

Für den Betrieb bieten die Microservices auch den Vorteil, dass eine Microservice-Architektur robuster sein kann als eine Architektur mit einem großen Dienst. Fehler und Abstürze in einer Komponente wirken sich nicht unbedingt auf alle anderen Dienste aus. Beispielsweise betrifft ein Ausfall einer Login-Komponente nicht unbedingt bereits eingeloggte Nutzer. Diese könnten in diesem einem Fall die eigentliche Funktion der Dienste weiter nutzen.

Neben den genannten Vorteilen gibt es auch einige Nachteile. Bei Microservices versteckt sich die Komplexität in der Verbindung zwischen den Diensten anstatt in den Diensten selber, sie verschwindet also nicht. Außerdem ist das Deployen und Testen von verteilten Anwendungen in der Regel schwieriger als bei einer einzelnen Anwendung, beispielsweise ist es einfacher eine WAR-Datei zu installieren als eine Microservice-Architektur aufzusetzen und zu starten. Eine weitere Schwierigkeit ist, dass durch die benötigten Aufrufe anderer Dienste zusätzliche Latenzen bei der Bearbeitung von Anfragen entstehen können.

Die Microservice-Architektur scheint sehr vielversprechend zu sein. In den letzten Jahren gab es einige interessante und sehr erfolgreiche Umsetzungen, wie zum Beispiel die Microservice-Architekturen von Netflix[3], Spotify[4] oder Amazon[5].

2.7 Modellgetriebene Softwareentwicklung

Modellgetriebene Softwareentwicklung (MDSD) ist ein Ansatz in der Softwareentwicklung um automatisch, auf Grundlage von formalen Modellen, lauffähige Software zu erzeugen. Stahl und andere haben in ihrem Buch “Model-Driven Software Development” [SVC06] die modellgetriebene Softwareentwicklung als einen Ansatz beschrieben der die Modelle nicht nur als Dokumentation ansieht. Im Gegensatz zur klassichen Programmierung stellen sie Modelle auf eine Ebene mit dem Quellcode, da ihre Implementierung generiert wird. Im Buch wird außerdem die modellgetriebene Softwareentwicklung als ein Werkzeug mit erheblichem Potential und Vorteilen beschrieben. Sie gehen sehr ausführlich auf die Grundlagen, wie verschiedene Konzepte und verschiedenen Klassen der MDSD ein. Im Anschluss beschreiben sie dort ausführlich domänenspezifische Modellierungssprachen und zeigen wie man diese erstellt, verwendet und aus ihnen Code erzeugen kann. An dieser Stelle gehen sie auch auf Modeltransformationstechniken ein und wie die Entwickler mit den entstehenden

[3]Netflix: https://www.netflix.com/
[4]Spotify: https://www.spotify.com/
[5]Amazon: https://www.amazon.com

Artefakten in der Versionierung und beim Testen umgehen können. Sie widmen außerdem ein ganzes Kapitel dem MDSD aus Sicht des Managements und beschreiben verschiedene Strategien um den MDSD-Ansatzes in einem Unternehmen oder in einem Projekt erfolgreich einzuführen.

Als Vorteile der MDSD nennen sie eine, durch Automatisierung gewonnene, Erhöhung der Geschwindigkeit in der Entwicklung. Außerdem führt der Einsatz von automatisierten Transformationen und formal-definierten Modellierungssprachen zu einer erhöhten Softwarequalität. Nach der Einführung von MDSD lassen sich die erstellten Architekturen, Modelle und Transformationen für weitere Projekte und Vorhaben einsetzen und fordern damit einen höheren Grad an Wiederverwendbarkeit und Wiederverwendung. Durch die Einführung einer abstrakteren Sicht lassen sich komplexe Systeme besser beherrschen, oft ermöglichen abstraktere Modelle auch das Beheben von Fehlern mehrerer Module an einer zentralen Stelle. Durch all diese Vorteile und Möglichkeiten ist MDSD eine produktive Umgebung und ein Bindeglied zwischen den Feldern: Technologie, Ingenieurskunst und Management.

Neben dem Begriff MDSD muss aber an dieser Stelle noch ein weiterer eingeführt werden. Waddington und Lardieri prägen den Begriff *"Model-Centric Software Development"* [WL06]. Sie beschreiben diesen Ansatz als noch weitreichender als MDSD, da er im Gegensatz zu MDSD nicht Artefakte aus Modellen erstellt, sondern gezielt in jeder Phase eines Projektes domänenspezifische Sprachen einsetzt um automatisch Teile der Implementierung zu erzeugen. Der Vorteil dieses Ansatzes ist es, dass die Modelle und die tatsächliche Implementierung nicht auseinanderlaufen, sprich Konflikte zwischen Modell und Implementierung enstehen, können. Zusätzlich müssen die erzeugten Artefakte auch nicht in die Versionierung eingepflegt werden und sind immer aktuell. Im Zuge einer ausführlichen Literaturrecherche wurde klar, dass viele Autoren, wenn sie MDSD anwenden, ähnliche Ziele verfolgen. Die Anwender von MDSD versuchen in der Praxis, aufgrund der gerade erwähnten Vorteil, auch ihre Modelle als Grundlage für generierte Implementierungsartefakte zu nutzen. Somit ist der Übergang zwischen diesen Begriffen sehr fließend. Diese Arbeit verfolgt wenn sie von MDSD spricht auch den Ansatz aus Modellen Teile der Implementierung automatisch zu generieren.

2.8 Modellbasierte Ansätze für REST-Schnittstellen

Für viele verschiedene Einsatzgebiete gibt es domänenspezifische Modellierungssprachen. Dies ist für die Erstellung und Beschreibung von REST-Schnittstellen nicht anders. Dieser Abschnitt soll auf die Sprachen *Swagger* und *RAML* eingehen und darüber hinaus noch den akademischen Ansatz, welcher in dieser Arbeit ebenso evaluiert wird, vorstellen.

Listing 2.1 Beispiel für Swagger-Definition

```
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting to the user!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name of the user to greet.
      responses:
        200:
          description: Returns the greeting.
          schema:
            type: string
        400:
          description: Invalid characters in "user" were provided.
```

2.8.1 Swagger

Bei Swagger handelt es sich um eine Beschreibungssprache für REST-Schnittstellen. Für Swagger gibt es mehrere Formate, Swagger kann mittels JSON und YAML[6] definiert werden.

Swagger verfolgt einen deskriptiven Ansatz für die Beschreibung von REST-Schnittstellen. Eine bestehende Swagger-Definition ermöglicht es zum einen Clients auf die beschriebene Schnittstellen zuzugreifen ohne deren exakte Implementierung zu kennen. Der Zugriff auf die Schnittstelle wird für viele Programmiersprachen dahingehend unterstützt, dass auf Basis von den Swagger-Definitionen passender Client- und Servercode generiert werden kann. Neben der Codegenerierung werden Entwickler durch Schnittstellendokumentationen unterstützt, welche auf Basis der Swagger-Beschreibungen generiert werden können. Swagger ist für Menschen und maschinell lesbar. Ein Beispiel für eine Swaggerdefinition ist in Listing 2.1 abgebildet.

[6] YAML - vereinfachte Auszeichnungssprache: http://www.yaml.org

2.8.2 RAML

Bei der Restful Api Modeling Language (RAML)[7] handelt es sich um eine auf YAML basierende Spezifikationssprache für das Modellieren von REST-Schnittstellen.

RAML entstand aufgrund der Unzufriedenheit mit Swagger hinsichtlich der Möglichkeiten im Schnittstellenentwurf. Uri Sarid (Mulesoft[8]) hat in einem Interview beschrieben, dass seiner Ansicht nach Swagger zwar gut für die Dokumentation einer bereits implementieren Schnittstelle nutzbar ist, aber für den Entwurf einer zu implementierenden Schnittstelle zu wortreich und zu unübersichtlich sei. Er ist der Meinung, dass fehlende Wiederverwendbarkeit und der fehlende Einsatz von Mustern (Patterns) Swagger für den *Design First*-Ansatz unpraktisch machen [Cag].

RAML zielt auf die Unterstützung während allen Phasen der Entwicklung ab. Auf der offizellen Webseite wird explizit für die Unterstützung beim Entwurf, bei der Implementierung, beim Testen, beim Dokumentieren, und beim Teilen der Spezifikation geworben.

Es gibt zwei unterschiedliche Versionen von RAML: Version 0.8 und Version 1.0. Die meisten Werkzeuge für RAML 1.0 sind auch mit der älteren Version 0.8 kompatibel. Durch diese Abwärtskompatibilität gibt es für RAML 0.8 tendenziell mehr Werkzeuge als für die Version 1.0. Wenn es um die Beispiele und Tutorials geht, sieht es ähnlich aus. Aus diesen Gründen wurde für den Vergleich die Version 0.8 genauer betrachtet.

Während der Arbeit stellte sich heraus, dass der Einsatz von RAML 0.8 nicht nur Vorteile brachte - RAML 1.0 hat einige neue Sprachelemente, welche die Arbeit mit der Sprache komfortabler machen. Der Umfang dieser Nachteile wird im Abschnitt 5.2 genauer betrachtet. Ein Beispiel für eine RAML-Definition in der Version 0.8 ist in Listing 2.2 zu sehen.

Akademisches Werkzeug

Die modellgetriebene Entwicklung von REST-Schnittstellen ist in der akademischen Welt bisher ein recht junges Feld. Die meisten der Veröffentlichungen sind 2009 oder später erschienen. Als eine der verwandten Arbeiten ist sicher das von Haupt und anderen veröffentlichte Paper *"A model-driven approach for REST compliant services"* [HKLS14] zu sehen. Sie beschreiben einen Ansatz zur modellgetriebenen Entwicklung von REST-Schnittstellen. Dieser Ansatz umfasst neben mehreren Metamodellen für den Entwurf und die Realisierung von REST-Schnittstellen, Diskussion über den Einsatz der Metamodelle und einem dazugehörigen Beispiel auch eine prototypische Implementierung. Die prototypische Implementierung umfasst einen grafischen Editor sowie eine vollständige Werkzeugkette um aus dem Modell

[7] Restful Api Modeling Language: http://raml.org/
[8] Mulesoft: https://www.mulesoft.com/

Listing 2.2 Beispiel für RAML-Definition in Version 0.8

```
#%RAML 0.8
title: Amazon simple storage API
version: 1
baseUri: https://{destinationBucket}.s3.amazonaws.com
/:
  post:
    description: The POST operation adds an object to a specified bucket using HTML
        forms.
    body:
      application/x-www-form-urlencoded:
        formParameters:
          AWSAccessKeyId:
            description: The AWS Access Key ID of the owner of the bucket who grants an
                Anonymous user access for a request that satisfies the set of constraints
                in the Policy.
            type: string
          acl:
            description: Specifies an Amazon S3 access control list. If an invalid access
                control list is specified, an error is generated.
            type: string
          file:
            - type: string
              description: Text content. The text content must be the last field in the
                  form.
            - type: file
              description: File to upload. The file must be the last field in the form.
```

ein lauffähiges Javaprojekt zu generieren. Das von Haupt und anderen beschriebene Konzept besteht aus mehreren Abstraktionsebenen. Die Hierarchie der Modelle ist in Abbildung 2.3 dargestellt. In der Abbildung sieht man das Domänenmodell, das zusammengesetzte Ressourcenmodell (*"Composite Resource Model"*), das atomare Ressourcenmodell, ein URL-Modell, die Dienstbeschreibungen, das JAX-RS-Anwendungsmodell und den Java Code. Das Domänenmodell beschreibt dabei ein von REST unabhängiges fachliches Modell, welches auf einem zur Anwendungsdomäne passenden Metamodell basiert. Das Domänenmodell wird auf das zusammengesetzte Ressourcenmodell oder das atomare Ressourcenmodell abgebildet. Den Kern des Konzepts ist das atomare Ressourcenmodell, es erlaubt das Modellieren einer Anwendung hinsichtlich ihrer Schnittstellen, ihrer Ressourcen und den Beziehungen zwischen den Ressourcen. Das zusammengesetzte Ressourcenmodell fasst mehrere Ressourcen des atomaren Ressourcenmodells zusammen um dessen Komplexität zu reduzieren. Dieses atomare Ressourcenmodel kann zum einen in verschiedene Dienstbeschreibungen transformiert werden, wie beispielsweise Web Application Description Language (WADL) oder Swagger. Es dient aber auch zusammen mit dem URL-Modell als Grundlage für das JAX-RS-Modell. Dabei definiert das URL-Modell unter welchen URIs die einzelnen Ressourcen des atomaren Ressourcenmodells erreichbar sind. Die letzte Modellart in dem Schaubild

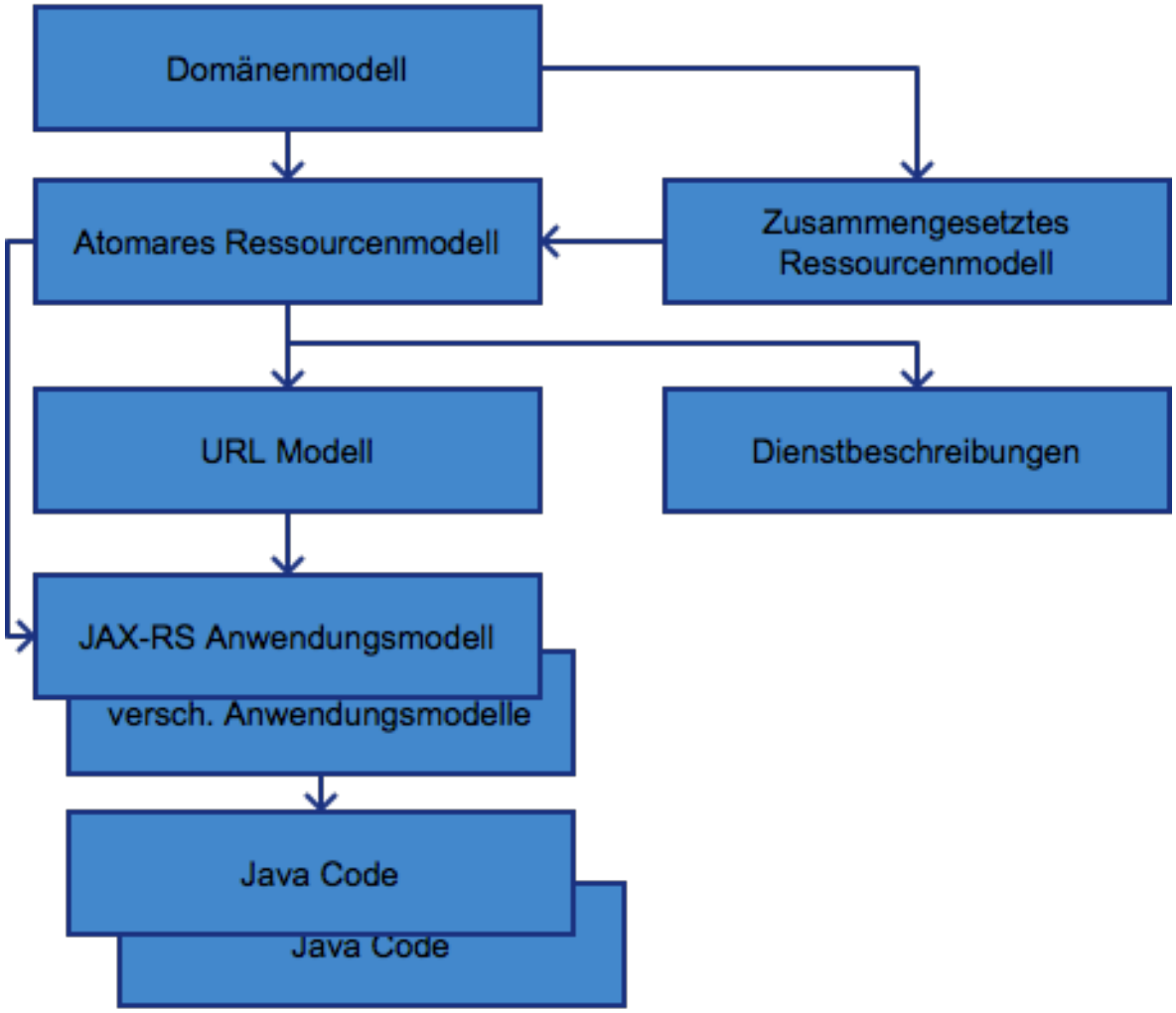

Abbildung 2.3: Metamodelle des akademischen Ansatzes nach Haupt et al. [HKLS14]

sind die Anwendungsmodelle, im Beispiel hier ein JAX-RS-Modell. Die Anwendungsmodelle setzten die Ressourcen des atomaren Ressourcenmodells um und macht sie unter den im URL-Modell definierten URIs erreichbar.

Im Zuge ihrer Arbeit haben Haupt und andere auch eine prototypische Implementierung ihres Ansatzes erstellt, er ist eine der evaluierten Methoden dieser Arbeit. Auf genauere Details dieses Prototypen wird im Hauptteil dieser Arbeit, in Abschnitt 5.3, eingegangen.

Das akademische Werkzeug unterstützt bei der Umsetzung des HATEOAS-Ansatzes. Dies wird durch die strikte Trennung des atomaren Ressourcenmodells und des URL-Modells erreicht. Durch die Umsetztung von HATEOAS ermöglicht der akademische Ansatzu auch erweiterte Interaktionsmuster wie das von Haupt und anderen in “A conversation based approach for modeling REST APIs” [HLP15] vorgestellte konverstationsbasierende Interaktionsmuster oder das in “Service Composition for REST” [HFK+14] vorgestellte Konzept zur Service-Komposition von REST-Diensten.

3 Verwandte Arbeiten

Dieses Kapitel widmet sich der Beschreibung der verwandten Arbeiten. Dabei wird zum einen auf die Themengebiete der modellgetriebenen Erstellung von REST-Diensten (Abschnitt 3.1) und zum anderen auf andere Arbeiten, welche mehrere Werkzeuge bzw. Entwicklungsansätze miteinander vergleichen (Abschnitt 3.2), eingegangen.

3.1 Modellgetriebene Erstellung von REST-Diensten

Das Thema dieser Arbeit ist eine Evaluierung von mehreren modellgetriebenen Ansätzen für Entwurf und Realisierung von REST-Schnittstellen. Es wurden bereits ähnliche Arbeiten wie diese durchgeführt. Beispielsweise hat Robert Wideberg [Wid15] ähnlich zu dieser Arbeit eine Fallstudie verschiedener Spezifikationsformate und HATEOAS in einem IT Unternehmen durchgeführt. Er kam bei dem Vergleich zwischen Swagger, RAML und API Blueprint[1] zu der Erkenntnis, dass alle Formate nicht die Anforderungen seines Vergleichs perfekt umsetzen, aber Swagger und RAML die wohl am ehesten geeigneten Sprachen für den produktiven Einsatz sind. Diese Arbeit fokussiert sich dabei sehr auf die Umsetzung von HATEOAS, hält sich aber im Kontrast zu der vorliegenden Arbeit sehr zurück wenn es darum geht die konkrete umgesetzte Schnittstelle, sowie den Prozess in der Firma zu beschreiben. Eine weitere ähnliche Arbeit verfasste Tomás Procházka mit seiner Masterarbeit *"Model-Driven Development of REST APIs"* [Pro15] in der er die Möglichkeit zur automatischen Erzeugung von REST-Schnittstellen untersuchte. Im Unterschied zu dieser Arbeit untersuchte er zwar ebenso RAML, Swagger und API Blueprint aber er baute auf Basis der drei Technologien einen Codegenerator. Der Codegenerator unterstützt neben der Generierung des Layouts ebenso die Anbindung an eine Datenbank und zusätzlich grundlegenden Verhaltensmuster. Für die Generation setzte er dabei lediglich JavaScript-Projekte und JavaScript-Frameworks ein, so dass seine Ergebnisse leider nicht Teil dieser Arbeit sein konnten. Die Entscheidung gegen Java als Technologie fiel bei ihm bewusst, weil er der Meinung ist, dass faktisch JavaScript Java als Technologie ablösen wird.

[1] API Blueprint: https://apiblueprint.org/

3.2 Vergleich von Werkzeugen und Entwicklungsansätzen

Dieses Arbeit vergleicht mehrere Werkzeuge und Entwicklungsansätze miteinander. Vor Bearbeitung dieser Arbeit wurden daher Arbeiten mit einem methodisch ähnlichem Schwerpunkt gesucht. Auffällig war, dass Arbeiten mit solchen Vergleichen in vielen unterschiedlichen Umfeldern stattfinden. Vergleiche von Werkzeugen und Methoden werden z.B. in der Erkennung von Codeduplikaten (Klonerkennung) [BKA+07; RCK09], beim Vergleich von Softwaretest Strategien [BS87; KL95; Mye78] durchgeführt. Die meisten dieser Arbeiten sind allerdings reine Werkzeugvergleiche.

Eines der Umfelder, welches sich mit Arbeiten dieser Art beschäftigt, ist die Klonerkennung. Beispiele für solche Arbeiten sind z.B. die Arbeiten von Roy, Cordy und Koschke [RCK09] und Bellon et al. [BKA+07]. Roy, Cordy und Koschke [RCK09] vergleichen in ihrer Arbeit "Comparison and evaluation of code clone detection techniques and tools" verschiedene Ansätze zur Klonerkennung. Sie gehen dabei systematisch vor, indem sie zuerst die verschiedenen Ansätze kategorisieren und im Anschluss daran Vergleichskriterien inklusive konkreter Vergleichsattribute definieren. Letztendlich geht es in einem späteren Kapitel dann um den Vergleich der Performance der verschiedenen Ansätze. Bellon et al. [BKA+07] gehen dabei ähnlich vor. Sie kategorisieren ebenso die verschiedenen Ansätze und definieren im Anschluss daran Metriken, anhand von denen sie die verschiedenen Ansätze vergleichen wollen. Um die Metriken zu erheben definieren sie ebenso einen Benchmark.

Ein weiteres Umfeld ist der Vergleich von verschiedenen Softwareteststrategien. Eine Arbeit, die bei der Literaturrecherche aufgefallen ist, ist die Arbeit von Basili und Selby [BS87] "Comparing the Effectiveness of Software Testing Strategies". Sie ist dahingehend interessant, da sie neben rein werkzeugbasierten Ansätzen auch den Ansatz "Code Reading" untersucht, welcher eine rein menschliche Tätigkeit ist. Die Ansätze sind dennoch sehr gut vergleichbar, da sie für ihren Vergleich nur ergebnisorientierte Metriken genutzt haben. Zum Beispiel nutzen sie die Anzahl der gefundenen Fehler, welche auch für das Durchlesen des Quellcodes anwendbar ist. Diese Arbeit zeigt, dass es bei sehr unterschiedlichen Methoden wichtig ist sinnvolle gemeinsame Metriken zu finden.

4 Projekt beim Industriepartner

Dieses Kapitel beschreibt den aktuellen Zustand des Projektes des Industriepartners. Dieses Kapitel soll dem Leser sowohl einen Überblick über den untersuchten Dienst, als auch einen Einblick in die agile Arbeitsweise des Unternehmens, sowie in die technische Details der Implementierung geben. Der Leser soll nach Lesen des Kapitels die Möglichkeit haben, das vorliegende Projekt mit anderen Projekten zu vergleichen und sich selbst ein Bild über die Anwendbarkeit dieser Arbeit zu machen. Im Abschnitt 4.1 wird hierfür das in der Firma praktizierten Scrums mit dem in Abschnitt 2.4 beschriebenen Referenzscrum verglichen. Außerdem wird in Abschnitt 4.2 der untersuchte Service genauer beschrieben und auf seine Eigenschaften eingegangen. Den Abschluss des Kapitels bildet Abschnitt 4.3 und beschreibt in welchem Maße bereits jetzt MDSD-Techniken zum Einsatz kommen.

4.1 Der Industriepartner

Beim Industriepartner handelt es sich um ein Tochterunternehmen eines renommierten Automobilherstellers, das ausschließlich für Kunden innerhalb des Konzerns arbeitet. Das Tochterunternehmen nimmt dabei die Rolle des konzerninternen IT-Dienstleisters ein. Es ordnet sich in die Kategorie *"Mittelständisches Unternehmen mit zwischen 501 bis 2000 Mitarbeitern"* ein und hat mehrere Standorte in Deutschland, Indien, Malaysia und China.

Das Unternehmen beschäftigt sich mit den Geschäftsfeldern *"Car IT and Mobility"*, *"Analytics"*, *"Security"*, *"Shared Services"*, *"Innovation"* und *"IT Retail"*. Diese Arbeit wurde im Bereich der *"Car IT and Mobility"*, in einem Team mit einer Teamgröße von 7 Mitglieder durchgeführt. Das Team beschäftigte sich während der Arbeit mit der Entwicklung mehrerer Java Backend Dienste. Die Dienste dienen als Datenquelle für unterschiedliche Clients für verschiedene Plattformen, welche von anderen Teams des Tochterunternehmens entwickelt werden.

Im Rahmen der Arbeit konnten unterschiedlichste Eindrücke über den Entwicklungsprozess gewonnen werden. Dies geschah zum einem durch regelmäßige Teilnahme an den Scrum-Meetings des Teams und außerdem durch gezielte Rückfragen verschiedener beteiligten Personen. Zur Durchführung dieser Arbeit ermöglichte der Industriepartner den Zugriff auf den Quellcode, sowie auf alle relevanten Kommunikationsplattformen. Darüber hinaus wurde auch der Zugriff auf die Schnittstellendatenbank des Unternehmens ermöglicht, welche alle Definitionen der internen Schnittstellen verwaltet.

Tabelle 4.1: Abweichungen von Referenzscrum

	Abweichung
Scrum Ereignisse:	
Daily Scrum	keine
Sprint Planning	keine
Sprint Review	- Entwicklungsteam stellt nicht wie gefordert dar was gut und schlecht lief - PO stellt nicht aktuellen Stand vor (inkl. neuen Fertigstellungstermin)
Sprint Retro	keine
Sprint	keine
Personen:	
Entwicklungsteam	keine
SM	Kein Coaching der Organisation, die neuen SMs nehmen an Schulung teil.
PO	Keine einzelne Person (interner / externer PO)
	Personalunion von interner PO und SM
Artefakte:	
Product Backlog	keine
Sprint Backlog	keine
Inkrement	keine

SCRUM

Wie bereits in Abschnitt 2.4 beschrieben folgt nun, um die Vergleichbarkeit der Arbeit zu sichern, in diesem Abschnitt nun ein Vergleich des tatsächlich praktizierten Scrums mit dem bereits beschriebenen Referenzscrum. Zur Durchführung des Vergleichs wurden alle im Scrum Guide [Jee] beschrieben Rollen, Aktivitäten, Artefakte und Vorgehen zusammengefasst und untersucht. Eine Visualisierung der Ergebnisse der Gegenüberstellung stellt Tabelle 4.1 dar.

In einem ersten Schritt konnten allgemeine Punkte, wie beispielsweise die *Timebox des Daily Scrums* oder die Dauer der unterschiedlichen Aktivitäten pro Sprint im Kalender gesichtet und validiert werden. Im Anschluss daran stand der *Scrum Master* für die Klärung der noch offenen Punkte und für weitere Rückfragen zur Verfügung.

Das praktizierte Scrum hat eine Sprintlänge von 14 Tagen, welche bisher nur in einem Fall auf Grund von externen Abhängigkeiten und extremen Planungsunsicherheiten auf 7 Tage reduziert wurde. Die Sprintlänge erfüllt so den Anspruch, dass die Sprints alle gleich lang sind und kürzer als einen Monat andauern. Die sonstigen Regeln für einen Sprint sind genauso wie im Scrumguide vorgeschrieben. Es dürfen während eines Sprints weder Änderungen vorgenommen werden, welche das Sprintziel gefährden, noch der Qualitätsanspruch

geschmälert werden. Wie vorgesehen ist es allerdings möglich, bei neuen Erkenntnissen, zwischen PO und Entwicklungsteam den Anforderungsumfang neu zu verhandeln.

Wenn man die Scrum Ereignisse betrachtet, so sind *Daily Scrum*, *Sprint Planning* und die *Sprint-Retrospective* nahezu unverändert zum Referenzprozess. Das *Daily Scrum* ist auf 15 Minuten begrenzt und findet jeden Tag zur selben Uhrzeit statt. Die Entwickler berichten, zum einem was sie gestern erreicht haben um zum Sprintziel beizutragen und was sie heute erledigen wollen um den Sprint weiter voranzubringen. Außerdem spricht jeder Entwickler hier Hindernisse ("Impediments") an auf die er bei seiner Tätigkeit gestoßen ist und die ihn nun behindern. Zusätzlich zum Referenzscrum wird hier noch geklärt ob es weiteren Abstimmungsbedarf gibt. Weitere Diskussion geschehen korrekterweise nach dem Daily Scrum.

Das *Sprint Planning* nimmt in etwa drei Stunden pro Sprint in Anspruch und ist so innerhalb der vorgeschlagenen Timebox von maximal 4h (Scrumguide-Vorschlag ist hier 8h bezogen auf einen 1 Monat langen Sprint). Hier wird, wie vorgesehen, im Team entschieden was Teil des nächsten Produkt Inkrements sein wird und wie das Ziel erreicht werden soll. Das Sprintplanning umfasst seitens des Entwicklerteams die Definition des Ziels des Sprints und eine Prognose über den zukünftigen Funktionsumfang des Produktes. Lediglich das vom Scrumguide geforderte Ausarbeiten der Umsetzung wird im Anschluss an das eigentliche Sprintplanning durchgeführt.

Die *Sprint Retrospective* wird genau wie im Scrumguide beschrieben umgesetzt. Die angedachten Prüfungen des vergangen Sprints in Bezug auf beteiligte Menschen, Beziehungen, Prozesse und Werkzeuge werden durchgeführt und mögliche Verbesserungen werden identifiziert und in eine Reihenfolge gebracht. Als Ergebnis der Retrospective werden Maßnahmen vereinbart um die Probleme in Zukunft zu verbessern.

Das *Sprint Review* ist das einzige Scrum Event, was wirklich abweicht. Es hält zwar die geplante Timebox von 2h ein, hat aber im Vergleich zum Referenzscrum doch einen anderen Inhalt. Während im Referenzscrum hier noch viele Elemente wie eine Darstellung der negativen und positiven Punkte geschieht, hat der vorliegende Scrumprozess diese Elemente lediglich in der Retrospective. Themen wie die Planung des nächsten Sprints und die Präsentation des aktuellen Product Backlogs Seitens des POs sind Teil eigener Aktivitäten. Das Product Backlog wird nur in der *Refinement*-Aktivität aktualisiert. Stattdessen nutzt das Team diesen Termin um Risikomanagement zu betreiben.

Wenn man die Rollen des Referenzscrums betrachtet, so findet man diese auch im praktiziertem Scrumprozess wieder. Dabei sind die Rollen des Entwicklungsteams, des Scrum Masters und des Product Owners nahezu 1:1 übernommen worden. Das Entwicklungsteam besteht aus Entwicklern und ist selbstorganisierend und interdisziplinär. Es liefert iterativ und inkrementell. Eine Besonderheit des vorliegenden Scrumprozesses ist, dass die Rolle des PO doppelt besetzt ist. Einer der POs ist aufgrund des Arbeitsmodells zwischen Tochter- und Muttergesellschaft ein Mitarbeiter des Mutterunternehmens. Zusätzlich dazu ist er auch

nicht immer für das Entwicklerteam direkt verfügbar, deshalb gibt es im Team noch einen internen PO, welcher dem externen PO einige Aufgaben abnimmt und ihn unterstützt. Der interne PO kümmert sich zusätzlich auch um alle organisatorischen und formalen Anforderungen, dazu gehört zum Beispiel auch das Beantragen von Zugriffen und Freigaben auf unterschiedliche Systeme, sowie speziell in diesem Projekt die Kommunikation mit den unterschiedlichen Datendienstleistern.

Die Aufgaben des POs ist vor allem die Pflege des Product Backlogs, er sorgt dafür, dass Einträge klar formuliert und nach Priorität sortiert sind. Außerdem ist er dafür verantwortlich den Nutzen der Arbeit des Entwicklungsteams zu maximieren. Das erreicht er indem er sicherstellt, dass zum einem das Product Backlog sichtbar ist - sprich es transparent ist und es klar ist, woran das Scrum Team als nächstes arbeiten wird - und dass das Entwicklungsteam die Einträge des Product Backlogs im erforderlichen Maß versteht.

Der SM ist für die Einhaltung des Scrumprozesses verantwortlich. Er vermittelt Techniken für eine effektive und effiziente Verwaltung des Product Backlogs und vermittelt dabei ein richtiges Verständniss von Agilität und ihrer Anwendung. Die Rolle im praktizierten Scrum hat alle Aufgaben des Referenzscrums bis auf einen Aufgabenbereich: Die Dienste des Scrum Masters an die gesamte Organisation wird nicht von ihm verantwortet. In der Organisation geschieht das durch die Schulung von zukünftigen SMs durch interne Schulungen anstatt bereits bestehende SMs damit zu beauftragen.

Zusammenfassend findet man eine Scrumimplementierung welche sich sehr nahe am Referenzscrum orientiert. Es gibt kleinere aber bewusste Abweichungen, wobei die auffälligste mit Sicherheit die doppelte Besetzung der PO-Rolle ist. Die kleinen Abweichungen sollten einem Vergleich des Scrum-Prozesses mit anderen, nach dem Referenzscrum umgesetzten, Scrum-Prozessen nicht im Wege stehen.

Erstellung und Umsetzung von REST-Schnittstellen in Scrum

Während der Betrachtung des Prozesses ist aufgefallen, dass ein mögliches Design der REST-Schnittstellen prinzipiell an mehreren Stellen im Scrumprozess stattfinden kann. Die im Folgenden beschriebenen zwei Zeitpunkte der möglichen Schnittstellenentwicklungen sind in Abbildung 4.1 visualisiert. Auf der einen Seite gibt es Entwicklungen, bei denen die Schnittstellen bereits vor Beginn der eigentlichen Implementierung entworfen und spezifiziert werden (Zeitpunkt 1). Diesen Fall kennt man am ehesten aus Projekten zur Ablösung einer bestehenden Implementierung inklusive Schnittstelle oder aus klassischen Projekten, welche mit Pflichten- und Lastenheft arbeiten. Auf der anderen Seite befinden sich Entwicklungen, welche bewusst agil gehalten werden und den Entwurf und die Umsetzung der Schnittstelle während der inkrementellen Implementierung durchführen (Zeitpunkt 2). Selbst dabei gibt es noch unterschiedliche Ansätze, je nachdem ob die Entwickler wie bereits in den Grundlagen (Abschnitt 2.2) beschrieben einen Top-Down- oder Bottom-Up-Ansatz

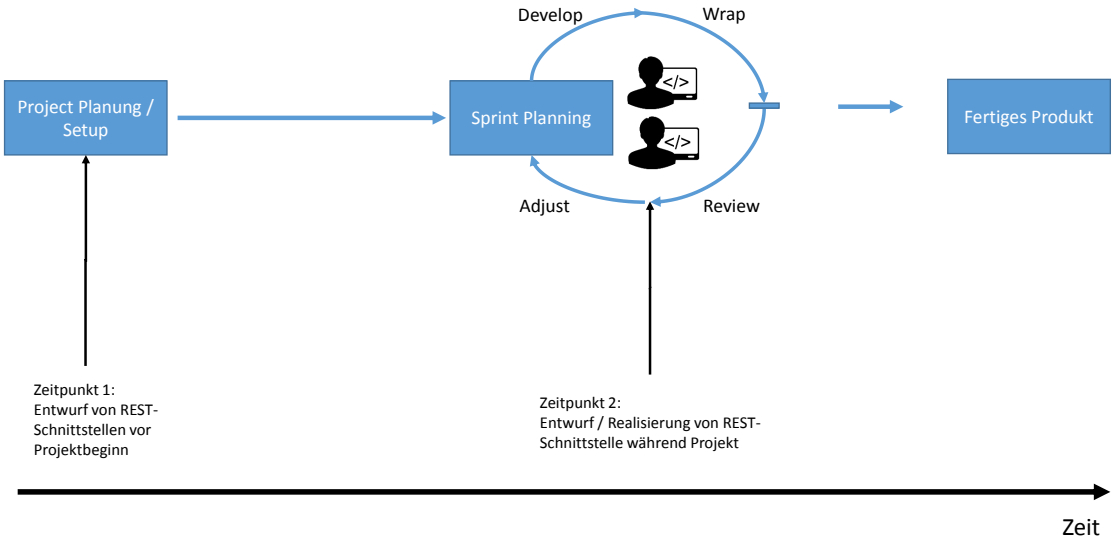

Abbildung 4.1: Unterschiedliche Zeitpunkte des Entwurfs von REST-APIs

wählen. Je nach gewählten Ansatz ist die Geschwindigkeit der Umsetung unterschiedlich. Wir konnten drei unterschiedliche Arten von Schnittstellenentwicklungsstilen identifizieren: *Formale Schnittstelle*, *inkrementelle Entwicklung der Schnittstelle* und einen *Hybridansatz*, welcher beide Ansätze kombiniert. In Abbildung 4.2 werden die Unterschiede im Grad der Fertigstellung der Schnittstellendefinition über die Zeit der verschiedenen Ansätze visualisiert. Die Abbildung verdeutlicht, dass bei dem Ansatz 'Formale Schnittstelle' der Großteil der Spezifikationsarbeit am Anfang des Projektes stattfindet. Bei der "inkrementellen Entwicklung der Schnittstelle" verläuft diese Entwicklung stetig. Der Hybrid-Ansatz kombiniert beide Ansätze und hat deshalb auch ihre Eigenschaften. Hier wird ein großer Teil zuerst spezifiziert aber dann später inkrementell weiterentwickelt.

Bei der Arbeit beim Industriepartner ist deutlich geworden, dass die Entwickler aller Teams gerne einen Hybrid-Ansatz wählen würden. Sie würden gerne zu Beginn eine Schnittstellendefinition besitzen, welche als Grundlage zur Arbeit und zur Diskussion zwischen den Teams dient. Die Schnittstelle würde im optimalen Fall dann dennoch uneingeschränkt stetig weiterentwickelt werden können. In der Praxis sieht es aber etwas anders aus. Bei der Neuentwicklung einer Schnittstelle wird in einem relativ frühen Sprint eine erste Version der Schnittstelle implementiert. Aus dieser ersten Version kann eine Schnittstellendefinition erzeugt werden, welche dann als Grundlage zur Diskussion und Kommunikation dient. Die weitere Entwicklung der Schnittstelle wird dann durch das Anpassen der Schnittstelle im Quellcode durchgeführt. Andere Teams nutzen die aktuellen Schnittstellendefintionen um mit den erstellten REST-Schnittstellen zu arbeiten. Dieser Prozess hat sich über längere Zeit entwickelt, da ein Entwickler in diesem Prozess zum einen keine Schnittstellendefinition von

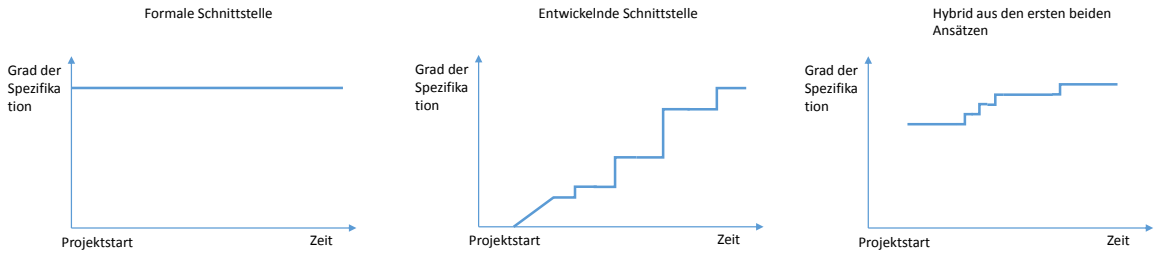

Abbildung 4.2: Entwicklung des Fertigstellungsgrades von REST-APIs in unterschiedlichen Projekttypen

Hand schreiben muss und zum anderen es so einfacher ist die erstellte Schnittstellendefinition auf aktuellem Stand zu halten.

Werkzeugunterstützung

Um die Teams bei der Durchführung der Projekte zu unterstützen wird auf das Tooling der Firma Atlassian zurückgegriffen. Für die allgemeine Kollaboration, genauer gesagt das Festhalten von Entscheidungen, Sammeln von Informationen und Bereitstellen von Guidelines, kommt Atlassian Confluence[1] zum Einsatz. Die Versionierung und Historie des Codes wird durch Atlassian Bitbucket Server[2] (ehemalig Stash) abgedeckt. Das speziell auf Scrum zugeschnittene Tool Atlassian Jira[3] kommt für die ganze Verwaltung des Prozesses zum Einsatz. Hier werden Sprints verwaltet, die Product Backlogs gepflegt und der Fortschritt der einzelnen Aufgabenpakete (Issues) aktualisiert und verfolgt. Als Entwicklungsumgebung wird im ganzen Team IntelliJ IDEA[4] von JetBrains eingesetzt. Doe Kommunikation im Team geschieht über E-Mail, die Kommunationswerkzeuge Slack[5] und Skype for Business[6].

4.2 Der Service

Der zentrale Punkt dieser Arbeit ist der Vergleich mehrerer Methodiken und der dazugehörigen Werkzeuge und Modelle. Der Vergleich wird anhand eines Beispielservices durchgeführt. Bei dem Beispielservice handelt es sich um eine bestehende Dienstimplementierung des

[1] Atlassian Confluence: https://de.atlassian.com/software/confluence
[2] Atlassian Bitbucket Server: https://de.atlassian.com/software/bitbucket/server
[3] Atlassian Jira: https://de.atlassian.com/software/jira
[4] IntelliJ IDEA: https://www.jetbrains.com/idea/
[5] Slack: http://slack.com
[6] Skype for Business: https://www.skype.com/de/business/skype-for-business/

Industriepartners. Der bestehende Beispielservice wird für den Vergleich in dieser Arbeit also mehrmals realisiert. Dieser Abschnitt soll erklären, um was für einen Dienst es sich dabei handelt.

Fachliche Sicht

Fachlich gesehen bietet der Dienst die Möglichkeit anhand einer geographischen Position verschiedene Sonderziele und Informationen über diese Sonderziele abzufragen.

Ein Sonderziel hat eine Menge an zugehörigen Informationen. Diese Informationen lassen sich in zwei Klassen einteilen: statische und nicht-statische Informationen. Statische Informationen sind Informatioen, welche sich nicht oder kaum ändern, beispielsweise ein Identifikator (ID), Position des Sonderziels oder auch die Telefonnummer der Verwaltung. Nicht-statische Daten hingegen sind oftmals Daten die sich in unterschiedlichen, unregelmäßigen und oft auch unvorhersehbaren Intervallen aktualisieren oder auf Basis von anderen nicht-statischen Daten berechnet werden, wie zum Beispiel aktuelle Besucherzahlen oder Angaben über eine geschätzte Wartezeit. Für diesen Dienst ist anzumerken, dass das Datenmodell relativ groß ist. Das Ergebnis einer Anfrage an den Dienst ist ein JSON-Array von gefunden Sonderzielen, welche wiederum selbst über 150 Name-Wert-Paare besitzen kann.

Ein Benutzer kann den Suchbereich auf zwei unterschiedliche Arten einschränken: Zum einem kann eine geographische Position und ein Radius, um die Position herum, angegeben werden und zum anderem können zwei geographische Positionen angegeben werden, welche dann die obere linke und untere rechte Ecke des Suchbereichs darstellen.

Technische Sicht

Aus technischer Sicht lässt sich der Dienst wie folgt beschreiben: Bei dem Dienst handelt es sich um eine *Spring Boot*[7]-Anwendung. Der Dienst nutzt neben der Basisfunktionalität auch erweiterte Funktionen des eingesetzten Frameworks und hat deshalb einige Abhängigkeiten, welche bei einem Wechsel zu einem anderem Framework aufgelöst werden müssten. Diese Tatsache muss berücksichtig werden, falls der Dienst mit Hilfe eines anderen Frameworks umgesetzt werden soll.

Funktional bündelt der Dienst die Daten mehrere Datendienstleister für Sonderziele und aggregiert diese in einem gemeinsamen Format, welches dem Nutzer dann bereitgestellt wird. Der Service ist Teil einer groß angelegten Microservice-Architektur, er hat jedoch recht wenige Abhängigkeiten zu anderen Diensten der Architektur.

[7]Spring Boot: http://projects.spring.io/spring-boot/

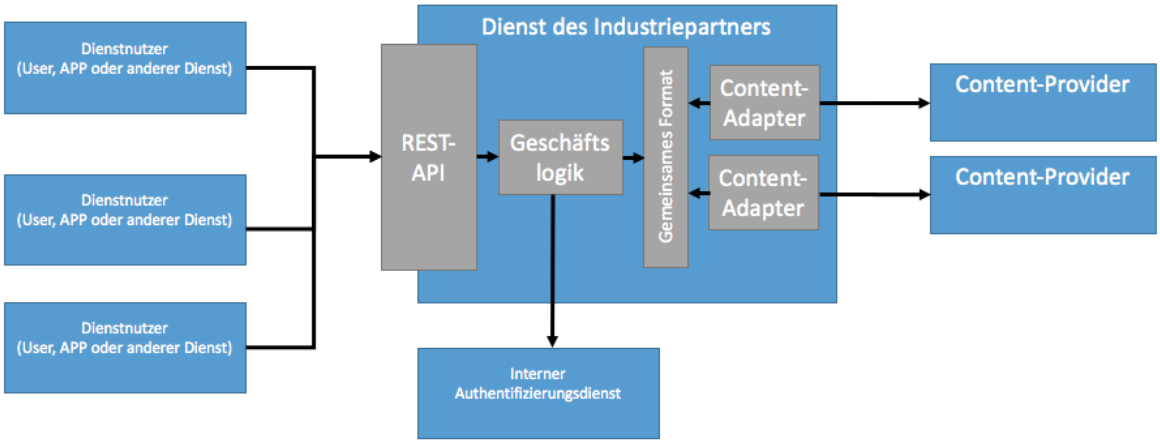

Abbildung 4.3: Logischer Aufbau: Dienst des Industriepartners

In Abbildung 4.3 sieht man mit welchen anderen Diensten ein Datenaustausch stattfindet. Der Dienst benötigt Informationen anderer Dienste um zum einem die Berechtigungen der anfragenden Nutzer zu Prüfen und zum anderem greift er auf die Datenanbieter zu um die angefragten Informationen bereitzustellen.

Um die endgültigen Artefakte zu erzeugen wird Gradle[8] als Buildtool eingesetzt. Die Automatisierung des Builds ermöglicht einige weitere Automatismen, bei der Entwicklung werden damit, zusätzlich neben dem Einsatz von statischen Codeanalysen und automatisierten Unittests, auch Maßnahmen hinsichtlich *Continous Integration (CI)* und *Continous Deployment (CD)* umgesetzt.

Die REST-Schnittstelle

Die REST-Schnittstelle des Dienstes ist aufgrund der Microservice-Architektur sehr überschaubar und besteht hauptsächlich aus zwei Ressourcen. Die Ressourcen sind Teil eines gemeinsamen Pfades (*"/information/v1/"*). Die Teilstücke des Pfades haben keine Funktion, falls man sie dennoch aufruft antwortet der Dienst mit einer HTTP 404 Nachricht welche noch zusätzliche Details wie den aufgerufenen Pfad und einen Zeitstempel enthält.

Die beiden Suchfunktionen sind auf die zwei Ressourcen verteilt. Die Suche in einem durch ein Viereck definiertem Suchbereich ist in der Ressource *"/information/v1/pois"* verfügbar, um die Suche mittels zentralem Punkt und Radius durchzuführen muss der Pfad *"/information/v1/pois/radius"* aufgerufen werden.

[8] Gradle: https://gradle.org/

Listing 4.1 Beispielhafte HTTP-Anfrage an '/information/v1/pois' für einen Bereich in der Stuttgarter Innenstadt

```
GET /information/v1/pois?lat_tl=48.784269&long_tl=9.164285\tabularnewline
     &lat_br=48.765490&long_br=9.200162
Host: meine-rest-api.de
```

Listing 4.2 Beispielhafte HTTP-Anfrage an '/information/v1/pois/radius' für einen 3 km großen Bereich in der Stuttgarter Innenstadt

```
GET /information/v1/pois/radius?lat=48.784269&long=9.164285&radius=3000
Host: meine-rest-api.de
```

Die Ressource *"/information/v1/pois"* besitzt entsprechend ihrer Funktion zur Suche in einem quadratischem Suchbereich die Parameter zur Angabe der Geokoordinaten für die obere linke und untere rechte Ecke *lat_tl*, *long_tl*, *lat_br* und *long_br* (wobie *tl* für *top left* und *br* für *bottom right* steht). Eine beispielhafte HTTP-Anfrage für die Ressource ist in 4.1 dargestellt.

Die Ressource *"information/v1/pois/radius"* hat dementsprechend die Queryparameter *lat*, *long* und *radius* um eine Geokoordinate sowie den Radius anzugeben (siehe Listing 4.2). Beide Ressourcen haben die optionalen Parameter *maxresults* und *offset* um Funktionen zur Paginierung anzubieten.

Die beiden eben genannten Ressourcen geben beide dasselbe Datenobjekt zurück, da sie sich lediglich um die Art der Suche unterscheiden. Eine exemplarische Antwort des Dienstes ist in Listing 4.3 dargestellt.

Die Schnittstelle erwartet zusätzlich zu den Parametern auch noch einen speziellen Anfrage-Header (*Request-Headers*), welcher zur Authentifizierung bei dem . Diese sind aber einigen technischen und fachlichen Anforderungen geschuldet und stehen deshalb nicht im Fokus dieser Arbeit. Eine grafische Übersicht der Ressourcen und den Methoden befindet sich in Abbildung 4.4.

Die beiden Teile des Pfades *"/information"* und *"/v1"* enthalten keine Geschäftslogik und geben jeweils eine HTTP 404 Antwort zurück. Der Pfad dient lediglich als Einstiegspunkt für den Dienst und um diesen von anderen Diensten zu differenzieren. Wenn in den Ressourcen ein Fehler auftritt, so werden die Statusmeldungen 400 *BAD REQUEST*, im Falle eines Fehler im Aufrufs, und der Status 500 *INTERNAL SERVER ERROR*, im Falle eines Fehler im Aufrufs, genutzt um den Nutzer der Schnittstelle auf einen Fehler hinzuweisen. Dabei wird ein sogenanntes Error-Objekt eingesetzt um die Fehler noch genauer zu beschreiben. Ein Beispiel für ein solches Error-Objekt im JSON-Format stellt Listing 4.4 dar.

Listing 4.3 Beispielhafte HTTP-Antwort des Dienstes

```
HTTP/1.1 200 OK
Date: Mon, 18 Jul 2016 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Mon, 18 Jul 20016 19:15:56 GMT
Content-Length: 2840
Content-Type: application/json
Connection: Closed

{
  "moreItems": true,
  "items": [
    {
      ...
    },
    ...
  ]
}
```

Listing 4.4 Beispiel für Error-Objekt im JSON-Format

```
{
  ''errors'': [
    {
       ''errorCode'': 1234,
       ''errorMessage'': ''Dies ist ein Beispielfehler der halt mal passiert!''}
    }
  ]
}
```

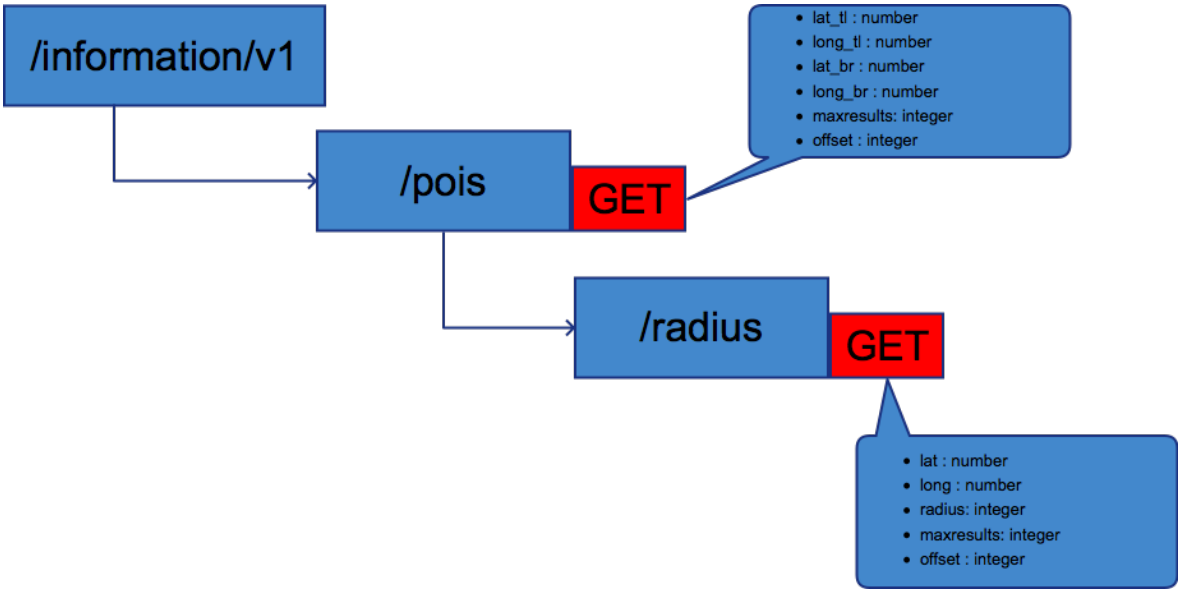

Abbildung 4.4: Layout der REST-Schnittstelle

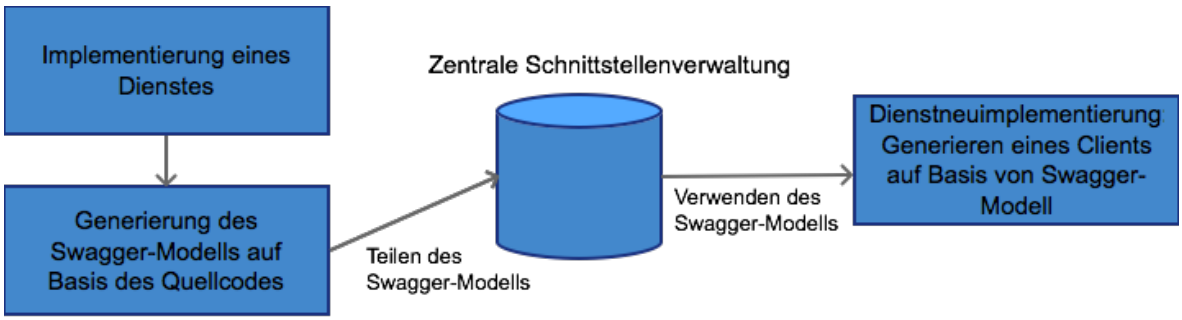

Abbildung 4.5: Nutzung von Swagger beim Industriepartner

4.3 Nutzung von Beschreibungssprachen für REST APIs

In der Organisation wird bereits die Codegenerierung von Swagger eingesetzt. Allerdings wird diese nur für die Erzeugung von REST-Clients auf Basis des Modells genutzt. Bei genauer Betrachtung fällt auf, dass dieser Einsatz von Swagger nicht modellgetrieben ist. Im Moment ist es zwar so, dass alle implementierten REST-Dienste eine Swagger-Dokumentation anbieten müssen. Die Erstellung der Swagger Definition geschieht allerdings nicht vor der Implementierung, sondern wird auf Basis des Quellcodes und zusätzlicher Annotationen erzeugt. Mit dieser Art der Verwendung ist das Modell letztendlich ein Produkt des Codes - bei einer modellgetriebenen Variante würde der Code Produkt des Modells sein.

Die generierten Swagger Definitionen werden zentral verwaltet und sind jedem Entwickler über einen internen Webserver zugänglich. Sie dienen der Organisation als Hilfe beim Zugriff auf die entsprechenden Dienste. Wenn bei einer neuen Implementierung ein anderer Dienst angesprochen werden muss, wird der Client für den entfernten Dienst aus der jeweiligen Swagger Definition unter Einsatz der Codegenerierung des Swagger Editors[9] erzeugt. Dieser Ablauf ist in Abbildung 4.5 skizziert.

Der beschriebene Dienst wurde selbst auch durch den Einsatz von Swagger dokumentiert. Wie genau dabei vorgegangen wurde und welche Annotationen zum Einsatz kamen wird im nächsten Kapitel genauer beschrieben werden.

[9]Swagger Editor: https://github.com/swagger-api/swagger-editor

5 Methoden und Tools für den Entwurf von REST-APIs

Dieses Kapitel schildert, wie die drei unterschiedlichen Methodiken (Tools des IST-Zustandes, RAML, und das akademische Tooling) genutzt werden können um den Entwurf von REST-APIs durchzuführen bzw. zu unterstützen. Bei den Methoden und Tools des IST-Zustands wird zusätzlich detailliert auf die genaue Erstellung der Schnittstellendokumentation mittels Swagger, welche als Modell für die Clientgenerierung dient, eingegangen. Das nächste Kapitel beschäftigt sich dann mit der Umsetzung und Implementierung der einzelnen Entwürfe. Ein Hindernis für das Verständnis dieses Kapitel kann die Tatsache sein, dass es sich bei der Art der Erstellung des IST-Zustands technisch gesehen um einen Bottom-Up-Ansatz handelt. Im Ist-Zustand wird für die Schnittstelle ein deskriptives Swagger-Modell auf Basis des Codes erstellt. Die beiden anderen Ansätze und deren Tools verfolgen hingegen den Top-Down-Ansatz, welcher jeweils ein präskriptives Modell als Resultat hat. Allerdings ist diese Abgrenzung nicht so hart wie hier zuerst dargestellt und angenommen. Der Prozess im Ist-Zustand hat doch auch Ähnlichkeiten zum Top-Down-Ansatz, da die Entwickler sich vor Beginn der Umsetzung natürlich auch ihre Gedanken zur zukünftigen Schnittstelle machen. Diese Überlegungen und Gedanken spiegeln sich dann aber letztendlich doch nur indirekt im Code wider, welcher dann typisch für einem Bottom-Up-Ansatz als Basis für das deskriptive Modell dient.

5.1 Methoden und Tools des IST-Zustands

Wie bereits in Kapitel 4 beschrieben sind die beim IST-Zustand eingesetzten Tools eher codezentrisch, dennoch werden auch hier Entwurfstätigkeiten durchgeführt. Anders als bei den anderen Ansätzen macht der Entwickler sich vor der Implementierung der REST-Schnittstelle zwar auch Überlegungen und Gedanken, aber er erstellt kein verbindliches Modell über das Design der Schnittstelle. Der Fokus in diesem Kapitel liegt also auf der Erstellung von zwei Arten von Modellen: Einmal die des unverbindlichen präskriptiven Modells des Entwicklers, welches für ihn als Anhaltspunkt für die Entwicklung dient und später verworfen wird, und die Erstellung des deskriptiven Swagger-Modells, welches später für die Clientgenerierung und Dokumentation der Schnittstelle verwendet wird.

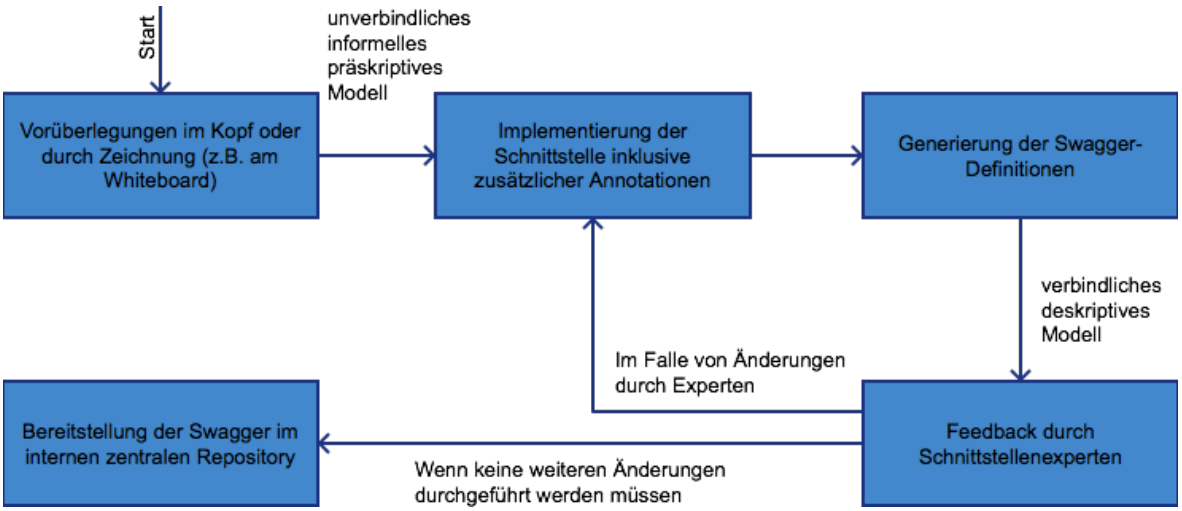

Abbildung 5.1: Workflow: Entwurf und Realisierung von REST-Schnittstellen beim Industriepartner

Der Prozess zum Entwurf und der Realisierung von REST-Schnittstellen beim Industriepartner ist grob in Abbildung 5.1 dargestellt. In der Grafik ist nicht ersichtlich, dass es sich bei der Implementierung um eine iterative Tätigkeit handelt. Als Teil dieser Tätigkeit werden dann die Schritte “Generierung der Swagger-Definiton” und “Feedback durch Schnittstellenexperten” durchlaufen.

Der initiale Designprozess, also die Erstellung des unverbindlichen deskriptiven Modells, ist abhängig von der Anzahl an Ressourcen in der zu entwickeltenden Schnittstelle. Viele Entwickler gaben an, dass sie den Entwurf von Schnittstellen mit nur einer Ressource gar nicht festhalten, sondern diese sobald sie ihre Überlegungen abgeschlossen haben direkt umsetzen. Als Basis für die Diskussion mit anderen Entwicklern dient dann die generierte Schnittstellendokumentation. Bei etwas größeren Schnittstellen werden Entwürfe am Whiteboard oder auf Papier angefertigt, welche aber kein striktes formales Format haben. Sie bestehen laut Aussage der Entwickler meistens aus hierarchischen Vierecken, welche ein in sich selbst einheitliches aber frei gewähltes Farbschema besitzen können. Ein Beispiel für eine nachgezeichnete Skizze in einem freien Format ist Abbildung 5.2. Hier sieht man wie zum einem das Layout der Schnittstelle, sowie das kleine Datenmodell festgehalten wurde. Auf die Frage wie eine Skizze eines größeren Datenmodells aussieht gaben die Entwickler an, dass sie für größere Datenmodelle keine Skizze erstellen, da diese Information bei größeren Schnittstellen sicherlich an anderer Stelle ausreichend dokumentiert ist. In so einem Fall wird hier nur auf die entsprechende Stelle referenziert. Bei der Erstellung werden hier während des Entwurfs gewisse Konventionen getroffen. Eine solche Konvention ist beispielsweise hier das unterstreichen der Pflichtfelder im Datenmodell.

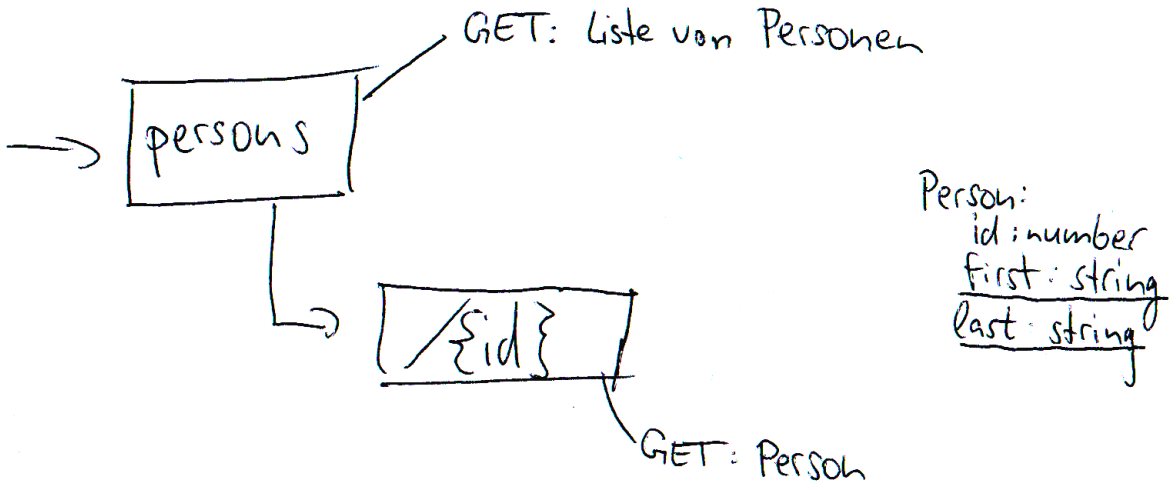

Abbildung 5.2: Nachzeichnung: Whiteboard Entwurf der Personenschnittstelle im freien Format

Dieser Entwurf dient, zusammen mit Wissen über definierte Best Practices und Guidelines, als Grundlage für die Implementierung der REST-Schnittstelle, er ist unverbindlich und dient bestenfalls zur Diskussion mit teaminternen Kollegen.

Zeitlich gesehen kommt nun im Anschluss an den groben Entwurf die Erstellung eines Prototypes der Schnittstelle mittels SpringBoot. Dies ist nicht and die Implementierung der Logik gebunden, diese kann zeitlich auch später angefangen und fertiggestellt werden. Dieser Prototy besteht zu diesem Zeitpunkt lediglich aus den Klassen und den für die Generierung des Modelles notwendigen Methodenköpfen. Der vorläufige Prototyp wird meistens noch um Beispieldaten aus Textdateien ergänzt, welche von der Schnittstelle exemplarisch zurückgegeben werden.

Die so frühzeitig generierten Swagger-Modelle können bereits zur Validierung und Diskussion an unternehmensinterne Experten gegeben werden. Zukünftige Nutzer der Schnittstelle können ebenso mit der Definition ihre Clients erzeugen und anhand der eventuell vorhandenen Beispieldaten die Schnittstelle bereits in ihre Anwendungen integrieren.

Um solch ein deskriptives Swagger-Modell auf Basis des Quellcode zu erzeugen bietet das Swagger-Framework unterschiedliche Möglichkeiten den Quellcode mit Annotationen zu versehen. Diese Annotationen werden später zusammen mit dem Quellcode interpretiert um eine Schnittstellendefinition zu erzeugen. Alle Swagger-Annotationen können auf der entsprechenden Wiki-Seite[1] des Swagger-Repositories im Detail nachgelesen werden, der

[1]Swagger-Core Annotations: https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X

folgende Auszug davon soll als grober Überblick dienen um die Techniken zur Erstellung des Modells besser einschätzen zu können.

Die Annotationen von Swagger lassen sich ihrer Verwendung nach in drei Kategorien einordnen: Zur Deklaration von Ressourcen, zur Deklaration von Operationen und die Annotationen zur Deklaration des Datenmodells. Die Swagger-Annotationen bilden zusammen mit den Spring-Annotationen die Grundlage für den bottom-up-Ansatz von Swagger. Die folgende Aufzählung fasst die wichtigsten Spring-[2] und Swagger-Annotationen[3] zusammen:

@**API:** Swagger-Annotation - Definiert den Namen und eine Beschreibung der Schnittstelle.

@**API-Operation:** Swagger-Annotation - liefert Beschreibung für eine Operation auf einer Ressource.

@**API-Responses:** Swagger-Annotation - liefert zusammen mit den Kindelementen *API-Response* Auskunft über die möglichen HTTP-Codes der Antworten.

@**ApiImplicitParams:** Swagger-Annotation - dient dazu Parameter zu definieren, welche nicht als Spring Parameter (s. RequestParam) definiert werden können.

@**ApiParam:** Swagger-Annotation - kann genutzt werden um dem API-Nutzer zusätzliche Informationen zu einem Parameter zu liefern.

@**RequestMapping:** Spring-Annotation - wird genutzt um einkommende Anfragen an spezielle Klassen oder Methoden zu verweisen.

@**ResponseStatus:** Spring-Annotation - setzt den HTTP-Statuscode, sowie den Grund (bzw. *reason*), der HTTP-Antwort an den Client.

@**RequestParam:** Spring-Annotation - gibt an, dass ein Methodenparameter an einen Parameter der Webanfragen gebunden werden soll.

Das Swagger-Framework bietet noch eine weitere Komponente namens Swagger-UI[4] welche dazu genutzt werden kann um die Swagger-Spezifikation im Browser grafisch und übersichtlich darzustellen. Neben der Verwendung als Dokumentation bietet Swagger-UI auch die Möglichkeit direkt Aufrufe gegen eine Implementierung des Dienstes zu testen. Diese Art von Dokumentation ist besonders bei Entwicklern beliebt, welche die Schnittstelle für ihre Anwendung nutzen wollen, da sie hier alle Operationen und Paramter gesammelt an einer Stelle finden können und diese gleich an dieser Stelle, ohne weitere Tools, direkt ausprobieren können.

[2] Spring Dokumentation: http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/
[3] Swagger Dokumentation: http://docs.swagger.io/swagger-core/apidocs/
[4] Swagger UI: http://swagger.io/swagger-ui/

Neben den Schnittstellenoperationen muss auch das Datenformat spezifiziert werden. Die meisten der zur Zeit beim Industriepartner entwickelten Schnittstellen benutzen nur JSON als Datenformat. In der Entwurfsphase werden die zurückgegebenen Daten in Form von beispielhaften JSON-Antworten beschrieben. Dabei wird sehr genau darauf geachtet, dass die gewählten Beispiele vollständig, konsistent und wohlüberlegt sind. Wohlüberlegt bedeutet hierbei, dass die gewählten Beispieldaten auch das richtige Format beinhalten - was besonders bei kombinierten Datentypen wie Zahlungsbeträgen (Währung + Betrag) und Datumsangaben (z.B. ISO 8601) relevant ist. Wie so ein Beispiel für eine fiktive Datendefinition einer Liste von Personen aussehen kann ist in der 5.1 gezeigt, welche eine Liste bestehenden aus Daten zu drei Personen darstellt. Sie enthalten neben Information zu der ID, dem Vornamen, Nachnamen auch zusätzlich Details zu der Rolle des Benutzers und des Erstelldatums. Für die Spezifikation von JSON-Daten existieren Standards wie beispielsweise "JSON Schema"[5], welche hier aber bewusst nicht zum Einsatz kommen.

Die Umsetzung des in diesem Abschnitt aufgezeigten Schnittstellenentwurfs inklusive der Datendefinition wird im Kapitel 6, das sich mit der Realisierung von REST-Schnittstellen beschäftigt, gezeigt werden.

5.2 Restful Api Modeling Language

Dieser Abschnitt bezieht sich auf die konkreten, von RAML zur Verfügung gestellten Möglichkeiten, Prozesse und verwendeten Tools zum Entwerfen von REST-Schnittstellen. Grundlegend wurde RAML schon in Unterabschnitt 2.8.2 beschrieben.

In dieser Arbeit wurden mehrere Werkzeuge für die Arbeit mit RAML evaluiert. Zum einen wurde ein Blick auf den *Mulesoft Api Designer*[6] geworfen. Er kann mittels Node Package Manager (npm)[7], welcher Teil des NodeJS Toolings ist, einfach installiert werden und direkt aus dem Terminal gestartet werden. Der Api Designer läuft nach dem Starten im Browser. Er ist sehr übersichtlich: er bietet neben einem Fenster zur Bearbeitung der RAML-Definition auch eine Swagger-UI ähnliche Ansicht, in der die bisherige Definition der Schnittstelle betrachtet werden kann. Ein besonderes Feature ist die Möglichkeit einen Mock-Service zu aktivieren, welcher die bereits spezifizierten Teile, durch das Zurückliefern der Beispiel auf den entsprechenden Pfaden, zur Verfügung stellt.

Einziger Wermutstropfen ist der nötige Aufwand für das Importieren, Exportieren und Speichern von Definitionen. Alle diese Operationen geschehen über Operationen im Menü ohne Hotkeys, beim Speichern muss immer eine explizite Datei angegeben werden. Eine

[5] JSON Schema http://json-schema.org/
[6] Api Designer: https://github.com/mulesoft/api-designer
[7] npm: https://www.npmjs.com/

Listing 5.1 Beispiel für JSON-Format: Liste von Personen

```
{
  "persons": [
    {
      "id" : 1,
      "first" : "Max",
      "last" : "Mustermann",
      "details" :
      {
        "created" : "2016-08-23T12:12+00:00",
        "role" : "admin"
      }
    },
    {
      "id" : 2,
      "first" : "Erika",
      "last" : "Mustermann"
      "details" :
      {
        "created" : "2016-08-25T13:37+00:00",
        "role" : "group-owner"
      }
    },
    {
      "id" : 3,
      "first" : "Marcus",
      "last" : "Mustermann"
      "details" :
      {
        "created" : "2016-10-23T16:32+00:00",
        "role" : "user"
      }
    }
  ]
}
```

schnelle Möglichkeit eine bearbeitet Definition zu speichern besteht also nicht und wurde bei der Verwendung sehr vermisst.

Das zweite betrachtete Werkzeug war die *Api Workbench*[8] welche als Package für den Editor Atom[9] installiert werden kann. Die Api Workbench konnte im Unterschied zum Api Designer durch ihre Integration in das lokale Dateisystem sowie durch, die von Atom bereitgestellten, Features des Editors überzeugen. Beide Werkzeuge boten eine gute Unterstützung beim Schreiben der Spezifikation in Form von automatischer Vervollständigung.

[8] Api Workbench: http://apiworkbench.com/
[9] Atom: https://atom.io/

RAML verfolgt konsequent den Top-Down-Ansatz um REST-Schnittstellen umzusetzen. Aus diesem Grund ist es auch nicht verwunderlich, dass die Sprachfeatures von RAML über die ledigliche deskriptive Beschreibung hinausgehen. RAML ermöglicht an verschiedenen Stellen einen hohen Grad an Wiederverwendung und setzt somit das *DRY-Prinzip (Don't repeat yourself)* um. Die Definition von sogenannten *"resource types"* erlaubt es gemeinsames Verhalten mehrerer Ressourcen nur einmal zu definieren und dann wiederzuverwenden.

Ein Beispiel für die Umsetzung eines "collection/collection-item" Musters ist in Listing 5.2 aufgezeigt. Die exemplarische Verwendung ist in 5.3 aufgezeigt. In ihr kann man sehen, wie kompakt Definitionen von gängigen Mustern aussehen können. In 5.2 kann man sehen wie zwei unterschiedliche *"resourceTypes"* (`collection` und `collection-item`) definiert werden und anschließend in 5.3 wiederverwendet werden.

Um einen möglichst hohen Grad an Wiederverwendung zu ermöglichen können die Beschreibungen aus 5.2 abhängig von dem Namen der später implementierenden Ressource gemacht werden. Dabei ist es möglich die variablen Teile mittels `!singularize` und `!pluralize` anzupassen. In den *resourceTypes* werden die HTTP-Methoden, der HTTP-Status der Antworten sowie die jeweiligen Nachrichtenbodies definiert. Für die Nachrichtenbodies können auch explizite Beispiele definiert werden. Die definierten ResourceTypes können im Anschluss für mehrere Resourcen verwendet werden. 5.3 zeigt wie dies genau funktioniert. Bei der Definition einer Ressource wird einfach der vorher definierte ResourceType als type angegeben. Die "/songs"-Ressource nutzt hier beispielsweise den "collection"-ResourceType und die "/songId"-Ressource nutzt den "collection-item"-ResourceType

In Kombination mit den in RAML möglichen *"Includes"*, und der daraus resultierenden Möglichkeit solche Blöcke auch auszulagern, können viele REST-Schnittstellen mit sehr wenig Aufwand sehr detailliert spezifiziert werden.

Eine andere Form der Wiederverwendbarkeit bieten die sogenannten *"Traits"*, sie können, ähnlich wie man es von abstrakten Klassen aus der Programmierung kennt, genutzt werden um gemeinsames Verhalten abzubilden. Besonders ist hierbei, dass man einer Operation mehrere Traits zuweisen kann. Dieses Sprachfeature ist vor allem sehr sinnvoll einsetzbar, wenn man viele Suchressourcen hat, welche Filterung und Paginierung unterstützen sollen.

Das Erstellen des Schnittstellenlayouts ist aber nur ein Teil der Aufgabe. Zusätzlich dazu muss auch das Datenmodell definiert werden. In RAML gibt es generell zwei Möglichkeiten um dies zu bewerkstelligen. Eine Möglichkeit ist es die einzelnen fachlichen Objekte mittels JSON-Schema zu definieren. Wer mit JSON-Schema bereits gearbeitet hat wird so schnell sein Datenmodell beschrieben haben - das Einbinden der Schemas in die RAML-Spezifikation ist allerdings leider nicht ganz so einfach. An vielen Stellen ist unklar ob eine Referenz nun auf das JSON-Schema geschieht oder ein Schema der RAML-Definition referenziert wird. Ein weiteres Manko ist die Tatsache, dass es bei der Erstellung dieser Arbeit nicht gelungen ist die, in die RAML-Definition eingebundenen, JSON-Schemas korrekt zu validieren und gleichzeitig das Modell generierbar zu halten. Grund hierfür ist, dass der Editor der API

Listing 5.2 Beispiel für Definition des Collection/Collection-Item Musters in RAML

```
resourceTypes:
  - collection:
      description: Collection of available <<resourcePathName>> in Jukebox.
      get:
        description: Get a list of <<resourcePathName>>.
        responses:
          200:
            body:
              application/json:
      post:
        description: |
          Add a new <<resourcePathName|!singularize>> to Jukebox.
        queryParameters:
          access_token:
            description: "The access token provided by the authentication application"
            example: AABBCCDD
            required: true
            type: string
        body:
          application/json:
            schema: <<resourcePathName|!singularize>>
        responses:
          200:
            body:
              application/json:
                example: |
                    { "message": "The <<resourcePathName|!singularize>> has been properly
                        entered" }
  - collection-item:
      description: Entity representing a <<resourcePathName|!singularize>>
      get:
        description: |
          Get the <<resourcePathName|!singularize>>
          with <<resourcePathName|!singularize>>Id =
          {<<resourcePathName|!singularize>>Id}
        responses:
          200:
            body:
              application/json:
          404:
            body:
              application/json:
                example: |
                  {"message": "<<resourcePathName|!singularize>> not found" }
```

Listing 5.3 Beispiel für die Verwendung des Collection/Collection-Item Muster in RAML

```
/songs:
 type:
  collection:
   exampleCollection: !include jukebox-include-songs.sample
   exampleItem: !include jukebox-include-song-new.sample
 /{songId}:
  type:
   collection-item:
    exampleItem: !include jukebox-include-song-retrieve.sample
```

Workbench mit Importen von Schemas wohl anders umgeht, wie der eingesetzte Java-Codegenerator. Somit ist die Möglichkeit die definierten Beispiele gegen das entsprechende JSON-Schema zu validieren nicht gegeben.

Die zweite Möglichkeit zur Definition von Datenmodellen ist der Einsatz von sogenannten *types*. Type-Definitionen sind nicht im JSON-Schema-Format, sie sind in YAML geschrieben und dabei einfacher als JSON-Schema gehalten. Prinzipiell ist die Anwendung von Type-Definitionen zu empfehlen, da sie einige sehr praktische Möglichkeiten bieten. Zum Beispiel vereinfacht der Einsatz von Type-Definitionen das Erstellen von Collection-Ressourcen ungemein. Wenn man Abbildung 5.4, welche eine Array-Definition mittels JSON-Schema darstellt, mit 5.5 vergleicht so kann man erkennen, dass die Arraydefinition unter Einsatz von Type-Definitionen um einiges kürzer, übersichtlicher und einfacher ist. Die Verwendung von JSON-Schema in RAML ist schlicht weg verwirrend, da hier JSON-Schema-Entitäten und RAML-Entitäten gemischt werden - beispielweise ist die Referenz in `persons` auf `person` eine Referenz aus dem JSON-Schema auf die RAML-Entität `person`.

Trotz der Nachteile kam in dieser Arbeit dennoch die JSON-Schema-Variante zum Einsatz, da zum Zeitpunkt der Toolauswahl - und auch bis zum Ende der Arbeit - kein funktionaler Java-Codegenerator mit Unterstützung für RAML 1.0 verfügbar war und die Type-Definitionen eine Sprachfeature der Version 1.0 sind.

5.3 Akademischer Ansatz

Das Tooling rund um den akademischen Ansatz ist als Eclipse-Plugin verfügbar. Neben einer Möglichkeit aus den erstellten Definitionen ein Maven-Dropwizard-Projekt zu erstellen, beinhaltet es auch die Editoren für die unterschiedlichen Modelle des Ansatzes.

Alle Modelle des Toolings nutzen XML als Auszeichnungssprache und sind so gut zu versionieren und auch von Hand bearbeitbar. Herzstück der Definition ist das sogenannte *RestResourceModel*. Das Herzstück der RestRessource sind die Ressourcendefinitionen. Sie definieren die Ressourcen selbst, sowie deren angebotene HTTP-Methoden (GET/PUT/POST/DELETE).

Listing 5.4 Beispiel für die Verwendung von JSON-Schema in RAML

```
schemas:
 - person: |
    {
      "$schema": "http://json-schema.org/draft-04/schema#",
      "type": "object",
      "properties": {
         "id": { "type":"number" },
         "first" : { "type":"string" },
         "last" : { "type":"string" }
      },
      "required": ["first", "last"]
    }
 - persons: |
      {
         "$schema": "http://json-schema.org/draft-04/schema#",
         "type": "array",
         "items": { "$ref": "person" }
      }
```

Listing 5.5 Beispiel für die Verwendung der Types-Definitionen in RAML 1.0

```
types:
 person:
   type: object
   properties:
     id:
       type: number
     first:
       type: string
       required: true
     last:
       type: string
       required: true
 persons:
   type: array
   items: person
```

Für die HTTP-Methoden werden hierbei direkt die Parameter, sowie deren Datentypen, und das Schema, das die Struktur der Antworten der Ressource definiert, spezifiziert. Zusätzlich zu den Ressourcedefinitionen beinhaltet die RestRessource aber auch noch Navigiationseinträge, sogenannte Connections, welche Beziehungen zwischen den einzelnen Ressourcen beschreiben. Diese werden genutzt um das HATEOAS-Prinzip umzusetzen. Das akademische Tooling ist der einzige Ansatz, der den Einsatz von HATEOAS aktiv unterstützt und fördert. Mit den anderen Tools ist der Einsatz von HATEOAS zwar auch möglich fordert aber sehr viele zusätzliche Konventionen wie die Einführung von speziellen Feldern für die Verlinkung von Ressourcen.

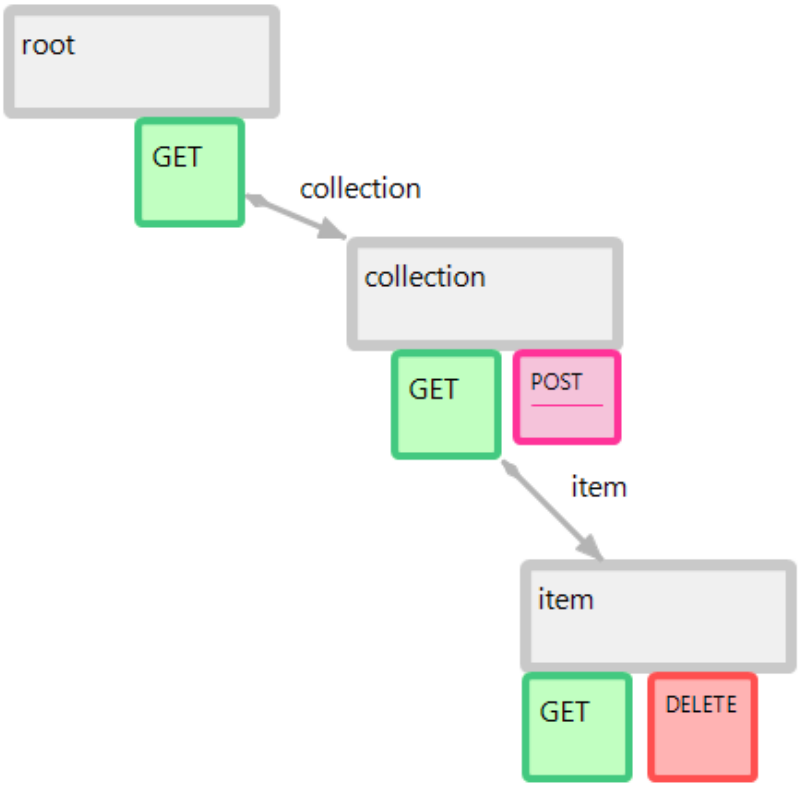

Abbildung 5.3: Beispiel für RestRessource-Diagramm des akademischen Ansatzes

Zusätzlich zur RestRessource gibt es noch das RestRessourceDiagram, welches zusammen mit dem mitgeliefertem Editor eine grafische Möglichkeit zur Betrachtung und Bearbeitung der REST-Schnitststellendefinition liefert. Eine solche Darstellung kann zum Beispiel in Abbildung 5.3 betrachtet werden. Diese Darstellung stellt die Schnittstelle als Graph dar und kann genutzt werden um sich einen schnellen Überblick über Eigenschaften der Schnittstelle zu verschaffen, welche aus anderen Arten der Definition nicht direkt ersichtlich sind. Zum Beispiel kann der Breite und Tiefe sowie der Verlinkung zwischen den Ressourcen schnell erkannt werden, wieviel verschiedene Ressourcen es gibt und wie mit ihnen interagiert wird. Typische Ressourcenarten wie das Collection-Ressourcen, welche eine Liste von unterliegenden Item-Ressourcen bündeln, können schon nach wenigen Anwendungen schnell erkannt werden, da sie ein auffälliges Layout haben. Es können schnell Ressourcen angelegt werden und mit Hilfe von Navigationen miteinander verknüpft werden.

Die Erstellung des Datenmodells geschieht hier im Editor der RestRessource. Hier bietet das Tooling an beliebig verschachtelte Strukturen zu erstellen und diese den entsprechenden Methoden der Ressourcen zuzuweisen. Auf Basis der RestRessource können im Anschluss an die Erstellung des Modells Deployment-Model, JaxRS-Modell und Maven-Modell generiert werden. Diese Modelle können dann genutzt werden um mit dem Generator ein lauffähiges

auf Dropwizard basierendes Mavenprojekt zu generieren. Das erstellte Projekt enthält dann alle notwendigen Datenmodelle und Ressourcedefinitionen um die Implementierung des REST-Dienstes zu beginnen. Um für die späteren Nutzer der Schnittstelle eine Dokumentation anzubieten ist es möglich die Definition in ein Swagger-Modell zu transformieren. Die Anwender können dann, wenn sie wollen, das von Swagger gewöhnte Tooling nutzen.

5.4 Erstellung der Modell-Artefakte

Die beiden Modelle für den Vergleich wurden auf Basis der bestehenden Dienst-Implementierung erstellt. Die Intention dahinter ist es, den Entwicklern die Möglichkeit zu geben ihre, händisch und mit viel Aufwand verbunden, entwickelte Schnittstelle mit den generierten zu vergleichen. Neben der Erstellung der Modelle mussten auch passende Java-Projekte erstellt werden um den lauffähigen Code auf Basis der Modelle zu generieren.

Akademischer Ansatz

Das Tooling des akademischen Ansatzes kommt als Eclipse Plugin inklusive einer Installationsanleitung daher. Nach der Installation des Plugins konnte direkt mit dem Einstieg in die Arbeit mit dem Tooling gestartet werden. Grundlage für den Start der Verwendung des akademischen Ansatzes war die kurze Einführung, welche Teil der Installationsanleitung ist. In dieser Einführung wird kurz erklärt wie die Erzeugung der unterschiedlichen Modelle und eines lauffähigen Modells funktioniert. Zur Nutzung der Editoren ist allerdings keine Anleitung beigelegt. Das hatte zur Folge, dass die Editoren explorativ erlernt werden mussten - was sich aber nicht als all zu schwer herausstellte.

Nach einer kurzen Einarbeitungszeit, welche sich vor allem mit der Erlernung der Bedienung beschäftigte, konnte ziemlich schnell eine kleine beispielhafte REST-Schnittstelle modelliert werden. Die ganze Modellierung spielte sich in der RestRessource ab, alle anderen Modelle wurden daraus generiert und mussten nicht weiter angepasst werden.

Für die Erstellung des Modells für den Vergleich der Ansätze musste jedoch noch mehr getan werden. Das grobe Layout, welches in diesem Schritt erstellt wurde ist im Screenshot in Abbildung 5.4 ersichtlich. Es enthält für jeden Pfadteil der Referenzschnittstelle eine Ressource (*information*, *v1*, *poi* und *radius*), sowie eine WurzelRessource (*root*). Jede Ressource bietet eine GET-Operation welche die Ressourcen, entsprechend der Hierarchie, mit der entsprechenden KindRessource verknüpft. Die Anordnung der Knoten, Operationen und Verbindungen ist durch den Benutzer des Modellierungstools zu definieren. Das Modellierungstool bietet hinsichtlich der Anordnung der Elemente ein automatisches Layout, welches aber nicht immer zufriedenstellende Ergebnisse liefert.

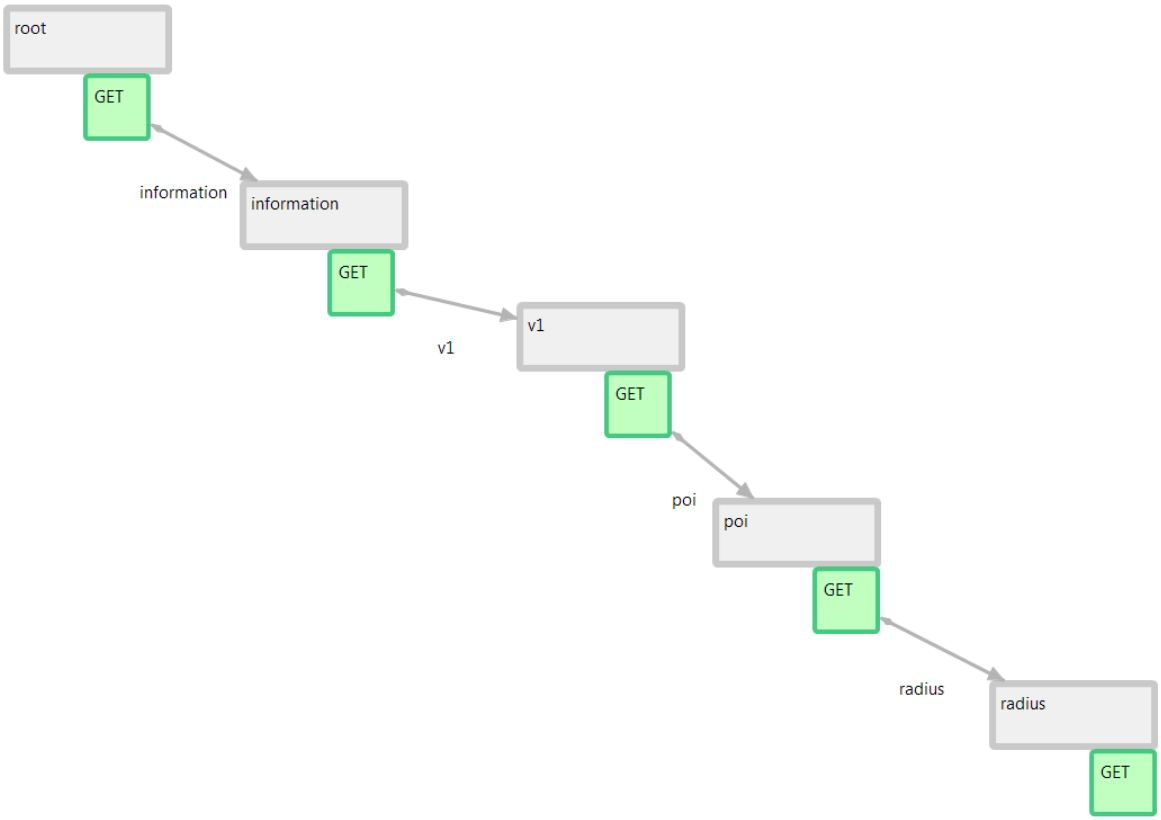

Abbildung 5.4: Screenshot: Grafische Ansicht des Akademischen Ansatzes

Zur Definition der Parameter der einzelnen Methoden musste in die Ansicht der RestRessource gewechselt werden, da diese Möglichkeit in der jetzigen Version des Werkzeuges noch nicht verfügbar ist. Hier konnten die Parameter dann direkt über das Kontextmenü der GET-Operationen erstellt werden. Neben der Angabe des Namens und des Types des Parameters muss hier noch angegeben werden ob der Parameter ein Pflichtparameter ist. In Abbildung 5.5 sieht man einen Auszug des erstellten Modells, welcher die Parameter der *poi*-Ressource darstellt. Die Abbildung zeigt außerdem den für die Paramtererstellung relevanten Teil des Kontextmenüs.

Nach der Erstellung der Ressourcen und deren Operationen musste für die Operationen noch ein Datenschema erzeugt werden, welches die genaue Form des Ergebnisses der Operationen definiert. In diesem Fall teilen sich die beiden wichtigen Ressourcen *poi* und *radius* ein und dasselbe Schema. Der akademische Ansatz unterscheidet bei den möglichen Typen innerhalb eines Schemas zwischen *Simple Types*, *Array Types* und *Object Types*. Die *Simple Types* sind recht einfach erklärt, sie sind einfache Datentypen und können einen der Typen `BOOLEAN`, `STRING`, `INTEGER` oder `NUMBER` annehmen. Bei dem *Array Types* handelt es sich um eine Möglichkeit ein Array von einem bestimmten Typ zu erstellen. Der letzte Typ sind die *Object Types*, welche eine Komposition mehrere Typen erlaubt. Dieser Typ wird

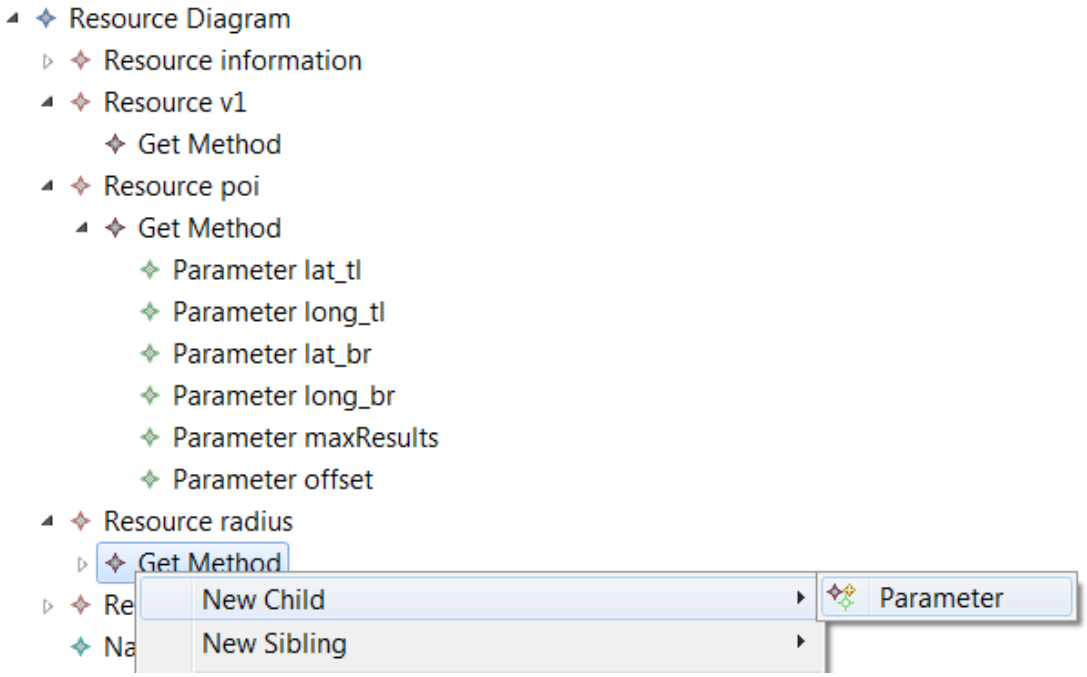

Abbildung 5.5: Screenshot: Darstellung der Parameter beim akademischen Ansatz

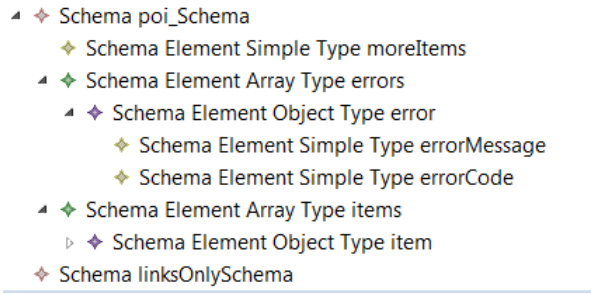

Abbildung 5.6: Screenshot: Schemadarstellung beim akademischen Ansatz

besonders für fachliche Domänenobjekte eingesetzt. Alle Typen können nahezu beliebig geschachtelt werden, so dass sehr komplexe Zusammenhalte abgebildet werden können. Um zum Beispiel eine Liste von Domänenklassen zurückzugeben, kann ein *Array Type* definiert werden, welcher wiederrum einen *Object Type* mit den entsprechenden *Simple Types* enthält. Abbildung 5.6 zeigt repräsentativ einen Auszug des erstellten Schemas. Der *Array Type* errors ist ein Beispiel für die eben beschriebene Schachtelung mehrere Schematypen. Dieses Art der Schemadefinition ermöglicht ähnliche Beschreibungen wie die Nutzung von JSON-Schema zur Definition der Domänenklassen. Allerdings muss man hier anmerken, dass die Erstellung im Vergleich zum JSON-Schema beachtlich einfacher, schneller und fehlerfreier geschieht.

RAML

Grundlage für die Erstellung des RAML-Modells waren die beiden Tutorials[10,11] auf der offiziellen RAML-Webseite. Sie vermitteln ein gutes Gefühl für die Funktionen der Sprache und zeigen, was notwendig ist um ein erstes generierbares Modell zu erstellen. Für weitere Sprachfeatures wurde die Spezifikation der eingesetzten RAML Version 0.8[12] als Nachschlagewerk zu Rate gezogen. Das RAML-Modell wurde auf Basis des Quellcodes erstellt, das entstandene REST-Layout deckt sich daher mit dem bereits vorgestellten aus Abbildung 4.4. Ein relativ hoher Aufwand musste getrieben werden um die gesamte Dokumentation aus den verschiedenen Swagger Annotationen in der Implementierung des Industriepartners in die RAML-Definition zu übertragen.

Die Erstellung des Datenmodell geschah auf Basis der JSON-Antworten des vom Industriepartner implementierten Dienstes. Eine erste Grundversion wurde mit der Webanwendung "JSONSchema"[13] erstellt. Die Anwendung ermöglicht ein konkretes JSON einzufügen und auf Basis der vorhandenen Feldern ein grobes Schema zu erstellen. Die Generierung kann an vielen Stellen nur vermuten ob ein Feld optional ist, deshalb wurde das Schema um die Spezifikation von benötigten Feldern erweitert. Als letzten Schritt wurde das Schema nochmals von Hand überprüft, dabei wurden Felder, welche laut Spezifikation vorhanden sein müssen aber nicht in den Testdaten vorhanden waren, ergänzt um so ein vollständiges Schema zu erhalten.

Das in diesem Kapitel erstellte RAML-Modell ist die Grundlage für die im nächsten Kapitel beschriebene Erstellung eines lauffähigen RAML-Java-Projektes.

[10]RAML 100 Tutorial: http://raml.org/developers/raml-100-tutorial#step-introduction
[11]RAML 200 Tutorial: http://raml.org/developers/raml-200-tutorial#step-introduction
[12]RAML 0.8 Spezifikation: https://github.com/raml-org/raml-spec/blob/master/versions/raml-08/raml-08.md
[13]JSONSchema: http://jsonschema.net/

6 Methoden und Tools für die Realisierung von REST-APIs

Dieses Kapitel beschäftigt sich nach der Betrachtung der Methoden und Tools für den Entwurf nun mit der Betrachtung von Methoden und Tools für die Realisierung von REST-APIs. Hierbei wird auf die Methoden und Tools des IST-Zustandes (Abschnitt 6.1), des Ansatzes unter Verwendung von RAML (Abschnitt 6.2) und des akademischen Ansatzes (Abschnitt 6.3) eingegangen werden.

6.1 Methoden und Tools des IST-Zustands

Bei den vom Industriepartner bereitgestellten Java-Projekten handelte es sich um SpringBoot-Anwendungen, welche mit Hilfe von Swagger-Annotationen dokumentiert wurden. SpringBoot ist ein auf Spring-basierendes Framework. Bei der Implementierung des Entwicklers kam also kein JAX-RS zum Einsatz, sondern generell die Annotationen des Spring-Frameworks. Durch das eingesetzte Framework SpringBoot sowie die Abhängigkeit SpringFox[1] sind die Swagger-Dokumentationen während der Laufzeit des Dienstes über das REST-Interface erreichbar und so immer aktuell. Das zusätzliche Einbinden von Swagger-UI stellt ein grafisch ansprechendes und übersichtliches Webinterface zum Betrachten der Swagger Dokumentation zur Verfügung. Swagger UI ermöglicht es außerdem Testaufrufe an die Schnittstelle abzusetzen.

Der Industriepartner implementiert die RestRessourcen in dedizierten RessourceKlassen. Klassen welche in SpringBoot für die Verarbeitung von REST-Anfragen verantwortlich sind müssen mit der *RestController*-Annotation versehen werden. Nur damit werden die Pfadinformationen aus ihr extrahiert und zur Laufzeit zur Verfügung gestellt. In den mit @*RestController* annotierten Klassen werden die einzelne Methoden nun mit @*RequestMapping*-Annotationen versehen. Die @*RequestMapping*-Annotationen spezifizieren den Pfad unter dem die Methode bereitgestellt wird, sowie die zugehörige HTTP-Methode und den produzierten Mediatype. Eine beispielhafte Definition eines solchen Mappings ist in in Listing 6.1 dargestellt. Die Klasse besitzt eine Methode, welche durch

[1] SpringFox: https://springfox.github.io/springfox/

Listing 6.1 Beispielhafte Ressource-Klasse in SpringBoot

```
@RestController
public class RestaurantsResource {
   @RequestMapping( value = "pointsofinterest/restaurants", method = RequestMethod.GET,
       produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
   public RestaurantsResponse getRestaurants(...){...};
}
```

die @*RequestMapping*-Annotation immer aufgerufen wird, wenn eine GET-Anfrage an den definierten Pfad ("`pointsofinterest/restaurants`") gesendet wird. Außerdem wurde hier spezifiziert, dass die GET-Anfragen welche bei dieser Methode landen mit einer JSON-Antwort beantwortet werden.

Um die Paramter der REST-Methoden zu definieren kommt in SpringBoot die @*RequestParam*-Annotation zum Einsatz, sie wird genutzt um den späteren Paramternamen zu definieren. Eine Datentypangabe ist nicht notwendig, da das Framework die bereits vorhandene Information über den Typ des Methodenparameters nutzt. SpringBoot unterstützt zusätzlich die Verwendung der `Javax.validation.constraints`[2], welche genutzt werden können um zusätzlich zum abgeleiteten Typ und dem Namen auch noch Informationen über den Wertebereich der Methodenvariable zu definieren. Die Definition des Wertebereichs ermöglicht eine Validierung zur Laufzeit. Eine Verletzung des gültigen Fehlerbereichs kann so auf eine passende Fehlermeldung abgebildet werden. Die in Listing 6.1 dargestellte Klasse kann nun so erweitert werden, dass die Methode drei Parameter erhält. Für unsere Beispiel könnte man sich vorstellen, dass eine Restaurant anhand seiner Geokoordinaten und eines Radius gesucht werden kann. Wenn man diese Paramter sinnvoll eingrenzt sieht die Definition der Methode wie in Listing 6.2 aus. Hier sind Parameter `lat` (Breitengrad) und `long` (Längengrad) auf die Bereiche von -90 bis 90.0 bzw. auf -180 und 180 begrenzt. Der Radius ist hierbei auf 200000 limitiert.

Neben der Hauptaufgabe der Implementierung der REST-Schnittstelle müssen zusätzlich dazu noch Datenmodelle für die Ergebnisse der Anfrage implementiert werden. Der Stand im Projekt ist, dass diese Klassen von Hand geschrieben werden. Die Felder und Methoden dieser Klassen werden dann zusätzlich noch um eventuell sinnvolle und notwendige Swagger-Annotationen zu Dokumentationszwecken erweitert. Die Serializierung der Objekte dieser Klassen erfolgt dann unter Einsatz der in SpringBoot integrierten Jackson-Bibliothek[3]. Beim Einsatz von Jackson sind weitgehend keinerlei gesonderte Annotationen notwendig um Objekte erfolgreich zu serializieren und zu deserializieren. Daher ist das Erstellen dieser Klassen weitgehend "Fleißarbeit" und muss sorgsam und korrekt durchgeführt werden. Um

[2]Javax.validation.constraints: http://docs.oracle.com/javaee/6/api/javax/validation/constraints/package-summary.html

[3]Jackson-Bibliothek: https://github.com/FasterXML/jackson

Listing 6.2 Beispielhafte Methode mit Parametern in SpringBoot

```
@RequestMapping( value = "pointsofinterest/restaurants", method = RequestMethod.GET,
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
public RestaurantsResponse getRestaurants(
   @RequestParam("lat")
   @DecimalMin(value = "-90.0", message = "latitude must be at least -90.0")
   @DecimalMax(value = "90.0", message = "latitude can only be up to 90.0")
         double latitude,
   @RequestParam("long")
   @DecimalMin(value = "-180.0", message = "longitude must be at least -180.0")
   @DecimalMax(value = "180.0", message = "longitude can only be up to 180.0")
         double longitude,
   @RequestParam("radius")
   @Min(value = 1, message = "radius must be at least 1")
   @Max(value = 200000, message = "radius can only be up to 200000 meters")
         int radius
      ){...};
}
```

Listing 6.3 Beispielhafte Datenklasse

```
public class Restaurant {
   private String name;
   private double latitude;
   private double longitude;
   private String type;

   public void setName(String name){
      this.name = name;
   }

   public void getName(){
      return name;
   }

   ... andere Getter und Setter ...
}
```

beim Beispiel des Restaurants aus den Listings 6.1 und 6.2 zu bleiben, zeigt Listing 6.3 eine exemplarische Modelklasse für ein Restaurant an.

Die Implementierung des Datenmodells macht bei dem vorliegenden Dienst einen großen Umfang, sowohl von Zeit und Codezeilen, aus. Es gibt zwei Gründe hierfür. Der Dienst liefert eine Menge Informationen über die einzelnen Sonderziele, dementsprechend groß ist dadurch natürlich auch das dazugehörige Modell. Der andere Grund ist, dass der Dienst selbst ein Microservice ist und so die Anzahl der Ressourcen (hier effektiv zwei - “/poi” und “/radius”) nicht sehr groß ist.

Zusammenfassend kann man sagen, dass der Dienst effektiv und sorgfältig nach gängigen Best-Practices entwickelt wurde und auch für die Nutzer der Schnittstelle, dank der bereitgestellten Swagger-Dokumentation, leicht zugänglich ist.

6.2 Restful Api Modeling Language

Nachdem das RAML-Modell aus dem vorherigen Kapitel fertiggestellt war musste nun ein passendes Java-Projekt dafür erzeugt werden.

Der dafür eingesetzte Generator ist Teil des *RAML for JAX-RS*[4] Projektes. Das Projekt beinhaltet Tools um mit JAX-RS und RAML zu arbeiten, dabei gibt es Tooling für beide Richtungen, also sowohl für die Richtung von JAX-RS zu RAML als auch anders herum. Diese Arbeit beschäftigt sich lediglich mit dem Tooling zum Erzeugen von JAX-RS Code auf Basis von RAML-Modellen. Neben den Tools enthält das Repository auch noch einge Beispiele zur Einbindung der Modelle in bestehende Projekte oder zum Aufsetzen neuer Projekte unter Einsatz des Toolings. Neben einer CLI-Version gibt es auch Plugins für Gradle und Maven, so dass das der Generator prinzipiell in jedes Build eingebaut werden kann.

Grundlage für das erstellte Java-Projekt war ein Beispiel des verwendeten RAML-JAX-RS Generators[5]. Das vorliegende Beispiel ist ein Maven-Projekt, was ein von Mulesoft[6] bereitgestelltes Maven Plugin nutzt um die Codegenerierung in den Build zu integrieren. Zu Beginn der Durchführung der Arbeit war das Aufsetzen des Maven-Projekts etwas aufwendiger, da das entsprechende Maven-Plugin nicht in dem entsprechenden Maven-Repository vorhanden war bzw. entfernt wurde. Ein Weg das Projekt dennoch erfolgreich zu erstellen bestand darin das Maven-Plugin manuell in das lokale Maven-Repository zu installieren und es im Anschluss dann in der *pom.xml* zu referenzieren. Zum Zeitpunkt des Schreibens dieser Arbeit (genau: 05.10.2016) befindet sich das aktuelle Plugin mittlerweile aber wieder zumindest in den Mulesoft Repositories - in MVN Central befindet sich allerdings immer noch nur die ältere Version 1.3.4 statt der aktuellen Version 1.3.5.

Nach erfolgreicher Überarbeitung des Projektes, was vor allem darin bestand Beispielimplementierungen zu entfernen und das neue Modell einzubinden, konnte die REST-Schnittstelle generiert werden. Es ist angenehm aufgefallen, dass der generierte Code nicht in den Codestand des Projektes übergeht sondern lediglich jedes Mal im von Maven dafür vorgesehen Buildschritt *generate sources* erzeugt wird. Der generierte Code ist so nicht Teil der Versionierung, Änderungen am Modell haben also so nur Auswirkungen auf die eventuelle Logik

[4] RAML for JAX-RS: https://github.com/mulesoft/raml-for-jax-rs
[5] Beispiel Jersey RAML-to-JAXRS Projekt: https://github.com/mulesoft/raml-for-jax-rs/tree/master/raml-to-jaxrs/examples/jersey-example
[6] Mulesoft: https://www.mulesoft.com/

und nicht auf sowieso generierte Artefakte. Dies führt zu einer (im Vergleich zu anderen Lösungen) sauberen Historie. Einziger Nachteil ist, dass ein Entwickler in seiner Entwicklungsumgebung nach einem Säubern (*Clean*) nicht vergessen darf die REST-Schnittstelle erneut zu generieren, da sonst das Programm nicht ausführbar ist.

Nachdem die Schnittstelle erzeugt worden ist musste nun die eigentliche Logik aus der Implementierung des Industriepartners hinzugefügt werden. Dafür konnten glücklicherweise große Teile der Implementierung wiederverwendet werden. Die Logik zum Verarbeiten der JSON-Antworten der verschiedenen Content-Provider musste lediglich dahingehend angepasst werden, dass sie in einer eigenen JSON-Parserinstanz läuft. Die Spring Boot Implementierung des Industriepartners nutzt die Möglichkeit mittels *RestTemplate* (ein Teil des Spring-Frameworks[7]) und der Injektion des passenden *Deserializers* die Antwort der Datenanbieter bereits beim Empfangen in das passenden Format zu überführen. Bei der Anpassung des Parsers wurde auch das Mapping überarbeitet, damit die empfangenen Daten korrekt auf die nun generierten Domänenklassen passen.

Zuletzt musste die Schnittstelle noch ausimplementiert werden, damit alle REST-Operationen die gerade beschriebe Logik aufrufen. Dies war letztendlich recht einfach durchzuführen, es muss lediglich ein Interface ausimplementiert werden. Dabei hat jede HTTP-Operation des REST-Interfaces eine eigene Methode welche die interne Logik aufrufen muss. Da der vorher erstellte Parser genau die Datenmodelle der REST-Schnittstelle liefert bestand die Logik letztenendlich darin einfach das Ergebnis des angepassten Parsers zurückzugeben.

6.3 Akademischer Ansatz

Nachdem das Modell in Form einer *RestRessource* mit zugehörigem *RestRessourceDiagram* mit Hilfe des akademischen Ansatzes fertiggestellt war, konnte nun daraus ein Java-Projekt generiert werden. Die Generation der Implementierung erfolgte mit dem integrierten Generator, welcher Teil des Eclipse Plugins ist. Um das Java-Projekt zu generieren musste die *RestRessource* zuerst durch mehrere Modell-zu-Modell-Transformationen umgewandelt werden. Im ersten Schritt wurde aus der RestResource ein *DeploymentModell* generiert. Für diesen einfachen Anwendungsfall musste nichts angepasst werden, allerdings ermöglicht die händische Anpassung des DeploymentModells einen zusätzlichen Einfluss auf die genauen Pfade zu den Ressourcen. Der nächste Schritt war die Umwandlung des DeploymentModells zu einem *JAX-RS-PSM Modells*. Das JAX-RS-PSM Modell hat Einfluss auf die späteren Eigenschaften des generierten Projektes wie den Projektnamen, den verwendeten Paketnamen und die definierte Projektversion. Neben diesen Eigenschaften können im JAX-RS-PSM auch die genauen Klassennamen für die einzelnen Ressourcen definiert werden. Der vorletzte

[7] Spring-Framework: https://spring.io/

Schritt ist eine Transformation des eben erstellten JAX-RS-PSM-Modells in ein Maven-Modell. Das Maven-Modell gibt Informationen über das zukünfige Maven-Artefakt an. Hier können detaillierte Eigenschaften des zukünftigen Maven-Artefakts definiert werden wie beispielsweise die *groupID*.

Auf Basis des Maven-Modells und den anderen Modellen kann ein lauffähiges Dropwizard-Projekt, welches Maven als Buildtool nutzt, erzeugt werden. Die Ausimplementierung konnte ähnlich wie bei dem bereits vorgestellten Ansatz unter Verwendung von RAML durchgeführt werden. Um das generierte Projekt zu nutzen mussten die generierten Java-Interfaces ausimplementiert werden. Die Hauptarbeit war wie bei der Verwendung von RAML die Anpassung des bestehenden Parsers auf das neue generierte Datenmodell.

7 Vergleich der Ansätze

Dieses Kapitel vergleicht die während dieser Arbeit eingesetzten Ansätze und die daraus resultierenden Modelle. Teil dieses Vergleichs sind zum einem die bei der Erstellungen aufgefallenen Eigenschaften der verschiedenen Ansätze und zum anderem wurde eine Befragung der Entwickler des Industriepartners durchgeführt, welche das Ziel hatte deren Meinung in die Evaluierung mit einzubeziehen.

7.1 Best-Practices: Entwurf von REST-Schnittstellen

Bei der Umsetzung von REST-Schnittstellen haben sich einige sogenannte *"Best-Practices"* herausgebildet. Um eine Sammlung von solchen Best-Practices zu erhalten, wurden während der Literaturrecherche einige Quellen von REST-Best-Practices und Richtlinien gesammelt und analysiert. Als Grundlage für den Vergleich gelten die folgenden Quellen:

- *WhiteHouse Api Standards* [The]: Besteht aus Richtlinien und Beispielen für die Schnittstellen des "White House" (USA).
- „Musterlösungen und Best Practices für das Design und die Realisierung von REST-Schnittstellen" [SRD14]: Eine Fachstudie über verschiedene Best-Practices, welche anhand der Schnittstelle der Plattform GitHub erklärt werden.
- *Best Practices for the Design of RESTful Web Services* [GGS+]: Ein Paper welches verschiedene Best-Practices identifiziert, sammelt und kategorisiert.
- *Best Practices for Designing a Pragmatic RESTful API* [Sah]: Große Sammlungen von Best-Practices mit anschaulichen Beispielen.
- *Design Beautiful REST + JSON APIs* [Haz]: Eine Präsentation über die Umsetzung von REST-Schnittstellen. Besonders wird dabei auf Best-Practices eingegangen.
- *10 Best Practices for Better RESTful API* [Jau]: Eine Sammlung von 10 ausführlich beschriebenen Best-Practices für die Realisierung von REST-Schnittstellen.

Tabelle 7.1: Best-Practices in der Literatur

	[The]	[SRD14]	[GGS+]	[Sah]	[Haz]	[Jau]
Allgemeines:						
URL identifiziert Ressource						
Semantische Benamung: Menschlich lesbare URL						
Plural bei Benennung von Ressourcen						
Versionierung:						
Version am Anfang der URL						
Format						
Format in URL (z.bB. /api/v1/magazines.json)						
Einsatz verschiedener Header-Felder für Format						
Fehlerbehandlung:						
Nutzung von geeignetem HTTP Status Code						
Nachricht enthält Nachricht für Entwickler						
Nachricht enthält Nachricht für Endnutzer						
Nachricht enthält internen Fehlercode						
Nachricht enthält Verweis auf Fehlerdokumentation						
Teilantworten:						
Ermöglichen von Teilantworten						
Einsatz von “optional” Feld in URL						
Paginierung:						
Standardwerte, wenn kein Limit gesetzt wurde						
Nutzung von “limit” und “offset” als Parameter						
Caching:						
Einsatz von ETAG oder Last-Modified Header						

Best-Practices müssen im Allgemeinen anerkannt sein. Bei der Sammlung der Best-Practices ist aufgefallen, dass sich viele der Autoren hinsichtlich ihrer Meinungen sogar widersprechen. Auffällig war, dass diese Widersprüche zwischen den Autoren nicht zufällig waren sondern man die Autoren in zwei Gruppen einordnen könnte: Die einen, welche man als REST-Puristen bezeichnen könnte, halten sich strikt an die von Roy Fielding definierten Prinzipien. Die anderen, welche man als Pragmatiker bezeichnen könnte, weichen diese Prinzipien auf um eine, ihrer Meinung, praktischere Umsetzung des REST-Musters zu erhalten.

Die Tabelle 7.1 zeigt das Ergebnis dieser Analyse inklusive vorhandener Widersprüche. Die Tabelle zeigt die genannten Best-Practices in den verschiedenen Dokumenten, dabei steht ein grünes Feld für das Vertreten der Meinung, rot für einen gegensätzige Meinung und gelb für keine Nennung dieser Best-Practices. Die Tabelle ist lediglich eine grobe Zusammenfassung der Best-Practice Analyse, Empfehlungen welche nur in einem der Dokumente vorgekommen

Listing 7.1 Beispielhafte HTTP-Anfrage mit Accept-Header

```
GET /api/resource
Host: meine-rest-api.de
Accept: application/vnd.meine-rest-api+json;version=2
```

sind wurden außen vor gelassen. Zusammenfassend lässt sich sagen, dass sich beide Seiten bei der Verwendung der HTTP Verben, der Benennung von Ressourcen, der Möglichkeit zur Anfrage von Teilantworten, der Fehlerbehandlung und der Lösung der Paginierung einig sind. Die größten Differenzen gab es bei der Umsetzung der Versionierung, bei der es die Möglichkeit zur Nutzung des Headers oder der Versionierung innerhalb der URL gibt. Die REST-Puristen haben hier den Standpunkt, dass es sich, unabhängig von der Version, fachlich jederzeit um dieselbe Resource handelt und sie deshalb unter derselben URL erreichbar sein muss. Sie raten, daher die Version mittels eines versionierten *Content-Types* im *Accept-Header* anzugeben. REST-Pragmatiker hingegen empfehlen die Platzierung einer Version in der URL. Ihre Argumente für dieses Art der Versionierung ist die damit erhaltene Zugänglichkeit der Schnittstelle, sie kann so ohne das Setzen eines Headers aufgerufen werden. Damit ist sie für einen Benutzer direkt im Browser eindeutig aufrufbar und muss nicht mittels eines geeigneten headerfähigen HTTP-Client aufgerufen werden. Das beschleunigt die Entwicklung und befähigt Entwickler schnell Testaufrufe an die Schnittstelle abzusetzen.

Pragmatisch umgesetzte Schnittstellen erkennt man direkt an der Version in der Url (z.B. http://meine-rest-api.de/api/v1/resource), bei einer Umsetzung der Puristen ist dies nicht so direkt ersichtlich, ihre Schnittstelle sieht auf den ersten Blick unversioniert aus (entsprechend http://meine-rest-api.de/api/resource). Bei einer Anfrage an diese Schnittstelle wird sie oftmals, sofern man bei der Anfrage keine Version im *Accept-Header* spezifiert standardmäßig die aktuellste Version zurückgeben. Manche Schnittstellen zwingen die Nutzer auch eine Version anzugeben. Für Clients ist es aus diesem Grund wichtig immer eine Version im Anfrage-Header anzugeben, da sich sonst das Verhalten, bei einem Versionswechsel der Schnittstelle, sehr schlagartig ändern würde und die Clientimplementierung auf einmal nicht mehr funktionieren würde.

Die Anfrage einer speziellen Version könnte dann wie in der Beispielabfrage in 7.1 aussehen. Hier wird der angefragte Typ mittels dem *Accept-Header* gesetzt. In diesem Beispiel gibt er an, dass die Repräsentation der Ressource bitte im *application/vnd.meine-rest-api+json;version=2* Format zurückgegeben werden soll. Der *vnd*-Teil des wird von RFC6838 in Abschnitt 3.2 [FKH13] vorgeschlagen und ist für vendorspezifische Datentypen vorgesehen.

Der Teil im Anschluss (*meine-rest-api+json*) gibt an um was für ein Format es sich genau handelt. Oft sieht man hier, dass der Datentyp mittels eines Plus angehängt wird. Im Anschluss daran wird dann mit einem Semikolon die gewünschte Version spezifiziert.

Ein Ziel dieser Arbeit war es die unterschiedlichen Ansätze auf die Umsetzung der gerade vorgestellten Best-Practices zu untersuchen. Leider konnte diese Untersuchung nicht sinnvoll durchgeführt werden. Grund hierfür ist, dass keiner der Ansätze während des Entwurfs beim Umsetzen der oben genannten Best-Practices unterstützend zur Seite steht. Unabhängig vom verwendeten Ansatz ist der Anwender selbst für die Umsetzung und Einhaltung dieser verantwortlich. Mit jedem der vorgestellten Ansätze lassen sich gute und schlecht konzipierte REST-Schnittstellen entwerfen und umsetzen. Beispielsweise ist es mit allen Ansätzen möglich eine Schnittstelle umzusetzen, welche URLs nutzt um Methoden anstatt Ressourcen zu bennen.

7.2 Evaluierung durch den Autor

Dieser Abschnitt spiegelt primär die Erfahrung und Meinung des Autors bei der Arbeit mit den verschiedenen Werkzeugen und deren Nutzung wider. Alle Aussagen und Informationen dieses Abschnittes beziehen sich auf die gesamte Werkzeugkette der jeweiligen Lösung. Wir vergleichen hier somit alle drei Varianten bestehend aus den verschiedenen Ansätzen zum Entwurf und der Realisierung von REST-Schnittstellen. Beim Ist-Zustand des Industriepartners haben wir somit die Vorüberlegungen der Entwickler, sowie den dabei ggf. entstehenden Schaubildern, und die Implementierung von Hand, welche mit Annotationen versehen wird um das deskriptive Modell zu erzeugen. Die Vorüberlegungen der Entwickler sind meist sehr abstrakt, ungeordnet und oftmals auch nur in den Köpfen der Entwickler vorhanden. Die mit Annotationen versehen Methoden der Implementierung bilden also die Referenz für diesen Vergleich. Die beiden anderen Varianten sind modellgetriebenen Ansätze und bieten damit Werkzeuge für die Entwicklung des Modells, sowie für die spätere Generierung der Schnittstelle.

Neben subjektiven Aussagen enthält dieser Abschnitt auch die Untersuchung und Bewertung der drei Methodiken nach den verschiedenen Kriterien welche im Abschnitt 2.1 der Grundlagen beschrieben wurden. Zur Wiederholung: Die 5 Eigenschaften nach denen wir Modelle bewerten wollen sind der *Abstraktionsgrad*, *Verständlichkeit*, die *Genauigkeit*, die *Fähigkeit zur Voraussage* und der benötigte *Aufwand zur Erstellung des Modells*.

Was in den Grundlagen nicht getan wurde aber für den Vergleich notwendig ist, ist die Überlegung wie die Ansätze genau miteinander verglichen werden sollen. Die 5 Eigenschaften sind schwer in verschiedene Kategorien einzuordnen, wohl aber können die Ansätze hinsichtlich dieser Eigenschaften paarweise miteinander verglichen werden. Beispielsweise ist es schwerer eine genaue Bewertung für die Abstraktion der Ansätze auf einer Punkteskala zu bestimmen, als sich darauf festzulegen, dass der akademische Ansatz um ein vielfaches abstrakter ist als der Ansatz des Industriepartners.

Aufgrund dieser Tatsache wurde der Analytische Hierarchieprozess (AHP), auch bekannt als die Saaty-Methode, eingesetzt um die Ansätze miteinander zu vergleichen. Die paarweisen Vergleiche können bei AHP mit 5 Bewertungen versehen werden. Eine Alternative kann dabei folgende Bewertungen im Vergleich mit einer anderen Alternative annehmen:

Skalenwert 1: gleich groß

Skalenwert 3: etwas größer

Skalenwert 5: deutlich größer

Skalenwert 7: sehr viel größer

Skalenwert 9: absolut dominierend

Da es sich um paarweise Vergleiche handelt besitzt der gegensätzliche Vergleich den Kehrwert. Wenn z.B. ein Vergleich von Alternative A und B mit 5 bewertet wird, so ist der Vergleich von B und A mit 1/5 zu bewerten.

Bei der konkreten Nutzung von AHP wurden die bereits genannten Kategorien als Grundlage genutzt. Die Wichtigkeit der 5 Eigenschaften wurde dabei naiv als gleich eingestuft, jede Kategorie geht also zu 20% in die Bewertung mit ein (bzw. Koeffizient 0,2). Das Ergebnis der gesamten AHP ist dabei in der endgültigen AHP-Matrix Tabelle 7.7 dargestellt, die Erläuterung des Ergebnisses folgt im Anschluss an die Betrachtung der einzelnen Eigenschaften.

Die erste betrachtete Eigenschaft ist der Abstraktionsgrad. Der Ansatz des Industriepartners ist, wenn man wie erwähnt von der mit Annotationen versehenen Implementierung ausgeht, der am wenigsten abstrakte Ansatz. Im Vergleich zu dem Ansatz unter Verwendung von RAML wurde er daher als deutlich weniger abstrakt eingestuft. Der akademische Ansatz ist im Vergleich zu der Implementierung von Hand sehr viel abstrakter. Vergleicht man den Ansatz mittels RAML mit dem akademischen Ansatz, so ist der akademische etwas abstrakter, da er eine grafische Ansicht bietet und noch mehr Details der Implementierung abstrahiert. Die resultierende Matrix ist in Tabelle 7.2 abgebildet.

Die zweite Eigenschaft ist die Verständlichkeit. Hier wurde verglichen wie gut aus den verbleibenden abstrakten Informationen noch Schlüsse über die Schnittstelle möglich sind. RAML demonstriert in dieser Kategorie Stärke als sehr ausdrucksstarke Sprache. Im Vergleich zu sowohl dem Ansatz des Industriepartners als auch dem akademischen Ansatz erreicht es RAML, dass bei sehr hoher Informationsdichte dennoch nahezu alle Informationen der Schnittstelle direkt ersichtlich sind. Dies ist etwas besser als bei dem Ansatz des Industriepartners und deutlich besser als bei dem akademischen Ansatz. Die Lösung des Industriepartner besitzt zwar eine weniger kompakte Ansicht, jedoch ist diese vollständig. Der akademische Ansatz hingegen abstrahiert sehr stark und lässt einige Informationen in der grafischen Ansicht, zumindest in der bei der Untersuchung vorliegenden Version, vermissen. Vergleicht man den akademischen Ansatz mit der Lösung des Industriepartners so wurden diese Ansätze als gleich eingestuft. Sie sind zwar total unterschiedlich, haben aber

Tabelle 7.2: AHP: Abstraktionsgrad

	akademischer Ansatz	RAML	Industriepartner
akademischer Ansatz	1	3	7
RAML	1/3	1	5
Industriepartner	1/7	1/5	1

Tabelle 7.3: AHP: Verständlichkeit

	akademischer Ansatz	RAML	Industriepartner
akademischer Ansatz	1	1/5	1
RAML	5	1	3
Industriepartner	1	1/3	1

beide ihre Stärken und Schwächen. Der akademische Ansatz ist sehr viel übersichtlicher als der des Industriepartners, da man hier nicht die einzelnen Klassen untersuchen muss. Allerdings ist der Ansatz des Industriepartners sehr viel verständlicher, wenn es um die genauen Parameter geht. Die AHP-Matrix für die Verständlichkeit ist in Tabelle 7.3 dargestellt.

Bei der Untersuchung der Genauigkeit der einzelnen Ansätze ist aufgefallen, dass alle Ansätze sehr genau sind. Wenn in einem der Ansätze etwas definiert wird, so spiegelt es auch die Eigenschaften der geplanten REST-Schnittstelle wider. Da alle Ansätze gleich zu einander bewertet wurden enthält jede Zelle der zugehörigen AHP-Matrix eine 1 (vgl. Tabelle 7.4)

Bei der Bewertung über die Möglichkeit Prognosen mit Hilfe des Modells über die zukünftige REST-Schnittstelle abzugeben hat der Ansatz des Industriepartners sehr gut abgeschnitten. Er ist dadurch, dass er zum Teil aus der Implementierung besteht, nahezu identisch mit der späteren Schnittstelle. Bei den beiden anderen Ansätzen wird hierbei die Codegenerierung bewertet. Der akademische Ansatz hat einen sehr hohen Abstraktionsgrad und arbeitet mit einigen Konventionen (z.B. wenn es um die Fehlerbehandlung geht), daher ist er eher schwer einzuschätzen. Außerdem fehlen in der grafischen Ansicht (zumindest in der jetzigen Version) noch einige nützliche Details, wie beispielsweise die Anzeige der Operationen

Tabelle 7.4: AHP: Genauigkeit

	akademischer Ansatz	RAML	Industriepartner
akademischer Ansatz	1	1	1
RAML	1	1	1
Industriepartner	1	1	1

Tabelle 7.5: AHP: Prognose

	akademischer Ansatz	RAML	Industriepartner
akademischer Ansatz	1	1	1/7
RAML	1	1	1/7
Industriepartner	7	7	1

Tabelle 7.6: AHP: Aufwand

	akademischer Ansatz	RAML	Industriepartner
akademischer Ansatz	1	7	5
RAML	1/7	1	1/3
Industriepartner	1/5	3	1

inklusive zugehöriger Parameter. Der Ansatz mittels RAML vermittelt durch die Syntax ein gutes Gefühl, wenn es um die Prognose geht. Allerdings bleibt der Codegenerator hinter den Erwartungen zurück und setzt die definierten Schnittstellen nicht immer vollständig um. Die Bewertung ist daher wie folgt: Der Ansatz des Industriepartners ist sehr viel besser zur Prognose geeignet als die beiden anderen Ansätze. Wenn man den Ansatz mittels RAML mit dem akademischen Ansatz vergleicht so sind diese gleich gut einzuschätzen. RAML gibt zwar ein sicheres Gefühl was die Syntax allein angeht, der akademische Syntax ist aber besser wenn es um die Konsistenz zwischen Modell und wirklich generierten Code geht. Die resultierende AHP-Matrix ist in Tabelle 7.5 dargestellt.

Bei der Eigenschaft des Erstellungsaufwands wird verglichen wie viel Aufwand in das Modell gesteckt werden muss im Vergleich zur Implementierung der Schnittstelle selbst. Bei der Implementierung des Industriepartners handelt es sich letztendlich um mindestens einen Prototypen der Schnittstelle, der Aufwand ist daher nicht ganz unerheblich. Der Aufwand des Ansatzes unter Verwendung von RAML hat auch einen ziemlich hohen Aufwand, was vorallem an der Erstellung der JSON-Schemas liegt. Der akademische Ansatz schneidet hier am besten ab. Er unterstützt den Nutzer durch den grafischen Editor sehr und nimmt auch sehr viel Arbeit bei der Erstellung der Datendefinitionen ab. Der akademische Ansatz ist sehr viel schneller zu erstellen als der Ansatz unter Verwendung von RAML. Im Vergleich zur Lösung des Industriepartners ist er deutlich schneller umzusetzen, da bei dem bestehenden Ansatz des Industriepartners die Datenklassen auch von Hand definiert werden müssen. Vergleicht man den Ansatz des Industriepartners mit dem unter Verwendung von RAML, so ist der des Industriepartners immer noch etwas schneller. Die AHP-Matrix für den Erstellungsaufwand ist in Tabelle 7.6 abgebildet.

Tabelle 7.7: Endgültige AHP-Matrix

	Abstraktion	Verständlichkeit	Genauigkeit	Prognose	Aufwand	**Ergebnis**
akademischer Ansatz	0,649118	0,156182	0,333333	0,111111	0,73065	**0,396079**
RAML	0,278955	0,658644	0,333333	0,111111	0,08096	**0,292601**
Industriepartner	0,071927	0,185174	0,333333	0,777778	0,188394	**0,311321**

Ergebnisse des Analytischen Hierarchieprozesses

Die Bewertung innerhalb der einzelnen Kategorien wurden nun mittels der Berechnung der Eigenvektoren der jeweiligen AHP-Matrix bestimmt. Das Ergebnis dieser Berechnungen ist in der endgültigen AHP-Matrix aufgezeigt Tabelle 7.7. Basierend auf den Punkten der drei Varianten in den 5 Kategorien und der gleichmäßigen Gewichtung der einzelnen Kategorien hat der akademische Ansatz die beste Punktzahl (0,396079) erzielt. Der Ansatz des Industriepartners ist mit einer Bewertung von 0,311321 auf Platz zwei gelandet. Die Methode unter Verwendung von RAML ist das Schlusslicht mit einer Wertung von 0,292601.

Bei genauerer Betrachtung erkennt man, dass diese Wertung sehr stark von der Gewichtung abhängt, da jeder der Ansätze an anderen Stellen Stärken und Schwächen besitzt. Wählt man die Gewichtung anders, so ist schnell einer der anderen Ansätze auf Platz 1. Dennoch ist diese Analyse sehr wertvoll. Mit ihr kann man erkennen wo die einzelnen Ansätze ihre Stärken und Schwächen besitzen. In der eben erwähnten AHP-Matrix Tabelle 7.7 sieht man, dass beispielsweise der akademische Ansatz seine Stärken im Aufwand und der Abstraktion besitzt, wohingegen der Ansatz des Industriepartners eher Stärken in der Möglichkeit zur Prognose ausspielen kann.

7.3 Befragung der Entwickler

Während der gesamten Dauer der Arbeit bestand enger Kontakt zu den Entwicklern des Industriepartners. Sie standen stets helfend zur Seite, wenn es Probleme bei der Nutzung ihrer Implementierung gab und bei allgemeinen Fragen zu ihrem Arbeitsablauf und ihren Tätigkeiten beim Entwurf und der Umsetzung ihrer REST-Schnittstellen. Darüber hinaus konnte auch an vielen der planenden Meetings teilgenommen werden um auch dort einen Eindruck über das Vorgehen und die Prozesse zu gewinnen.

Um die Meinung der Entwickler des Industriepartners in die Arbeit mit einfließen zu lassen wurden während der Durchführung der Arbeit regelmäßig Gespräche durchgeführt. In der Phase der Evaluierung wurden gezielte Einzelinterviews mit einigen der Entwicklern durchgeführt. Ziel dieser Interviews war es zum einem ihre Grundhaltung gegenüber modellgetriebener Softwareentwicklung zu erfahren, aber auch mit Ihnen gemeinsam eine

Einschätzung über die Brauchbarkeit der verschiedenen Ansätze in ihrem jetzigen Projektumfeld zu entwickeln.

Zur Steuerung des Interviews wurde ein Fragebogen erstellt. Bei dem Entwurf des Fragebogens ist etwas Arbeit vorausgegangen um einen größtmöglichen Nutzen aus der limitierten Zeit der Entwickler zu ziehen. Der Fragebogen wurde mit Orientierung an dem Bericht von Daniel W. Turner III [Tur10] und dem Buch "Qualitative Inquiry & Research Design" von John W. Creswell [Cre13] erstellt.

Aufgrund deren Empfehlungen wurden geschlossene Fragen mit offenen Fragen kombiniert. Außerdem empfehlen beide Autoren nicht direkt nach Rangfolgen zu fragen, sondern diese Frage etwas zu verschleiern. In dem entwickelten Fragebogen wurde somit ein ganzzahliges Punktesystem von 0 (schlecht) bis 10 (sehr gut) Punkten für die Ansätze verwendet, wenn auch für die Auswertung lediglich die Reihenfolge der Ansätze von Interesse war. Keiner der Entwickler vergab für mehrere Ansätze die gleiche Punktzahl wie für einen anderen. Der Fragebogen für die Entwickler besteht aus den vier Abschnitten "Allgemein", "Vorführung Modellwerkzeuge", "Vorführung generierter Code / Workflow", "Feedback". Der allgemeine Teil zielt dient dabei als Einstieg in die Befragung und stellt Fragen zur bisherigen Erfahrung des Entwicklers mit modellgetrieben Werkzeugen. Er enthält folgende Fragen mit den dazugehörigen Antwortmöglickeiten:

1. **"Hast du bereits modellgetriebene Werkzeuge bei der Softwareentwicklung genutzt?"**:
   ja / nein
2. **"Für welchen Zweck hast du diese Werkzeuge eingesetzt?"**:
   offene Frage
3. **"Wie würdest du das modellgetriebene Werkzeug im Vergleich zu einer Entwicklung ohne dieses Werkzeug bewerten?"**:
   sehr unterlegen / unterlegen / überlegen / sehr überlegen / neutral (keine Präferenz)
4. **"Was war besser oder schlechter im Vergleich zur Entwicklung ohne das verwendete Werkzeug?"**:
   offene Frage mit Einordnung in positiv und negativ

Der Teil der Befragung zur Vorführung der Modellwerkzeuge zielte direkt auf die Bewertung der drei zu vergleichenden Ansätze ab. Er enthält folgende Fragen:

1. **"Wieviel Punkte würdest du Variante 1 [Anmerkung: Ansatz des Industriepartners] geben (1 schlecht bis 10 perfekt)?"**:
   Antwort von 1-10
2. **"Wieviel Punkte würdest du Variante 2 [Anmerkung: Akademischer Ansatz] geben (1 schlecht bis 10 perfekt)?"**:
   Antwort von 1-10

3. **“Wieviel Punkte würdest du Variante 3 [Anmerkung: Verwendung von RAML] geben (1 schlecht bis 10 perfekt)?”**:
   Antwort von 1-10
4. **“Was sind die Gründe für die einzelnen Bewertungen?”**:
   offene Frage

Ähnlich zur Befragung zu den Modellwerkzeugen wurde auch die Befragung im Abschnitt “Vorführung generierter Code / Workflow” durchgeführt. Dieser Teil enthält die gleiche Fragen wie im vorherigen Teil, diesmal nur bezogen auf die Codegenerierung und den eingesetzten Workflow:

1. **“Wieviel Punkte würdest du Variante 1 [Anmerkung: Ansatz des Industriepartners] geben (1 schlecht bis 10 perfekt)?”**:
   Antwort von 1-10
2. **“Wieviel Punkte würdest du Variante 2 [Anmerkung: Akademischer Ansatz] geben (1 schlecht bis 10 perfekt)?”**:
   Antwort von 1-10
3. **“Wieviel Punkte würdest du Variante 3 [Anmerkung: Verwendung von RAML] geben (1 schlecht bis 10 perfekt)?”**:
   Antwort von 1-10
4. **“Was sind die Gründe für die einzelnen Bewertungen?”**:
   offene Frage

Der Abschluss der Befragung fand im Abschnitt “Feedback” statt. Hier hatte der Entwickler nochmals die Möglichkeit in offenen Fragen seine Eindrücke zu schildern. Die drei vorbereiteten Fragen waren dabei:

1. **“Wo siehst du mögliche Vorteile / Nachteile der drei vorgestellten Werkzeuge?”**:
   offene Frage
2. **“Könntest du dir den Einsatz von Teilen der Werkzeugkette vorstellen?”**:
   offene Frage
3. **“Was müsste ein Modellingwerkzeug können / was für Eigenschaften müsste es haben um dich zu überzeugen es bei der Erstellung von REST-Schnittstellen einzusetzen?”**:
   offene Frage

Ein Teil der Durchführung der Befragung war eine Präsentation der unterschiedlichen Ansätze. Hierfür wurde zusammen mit dem Entwickler für die beiden neuen Ansätze eine einfache REST-Schnittstelle modelliert und generiert. Die während der Befragung mittels RAML und akademischen Ansatzes erzeugten REST-Schnittstellen entsprechen dem Layout

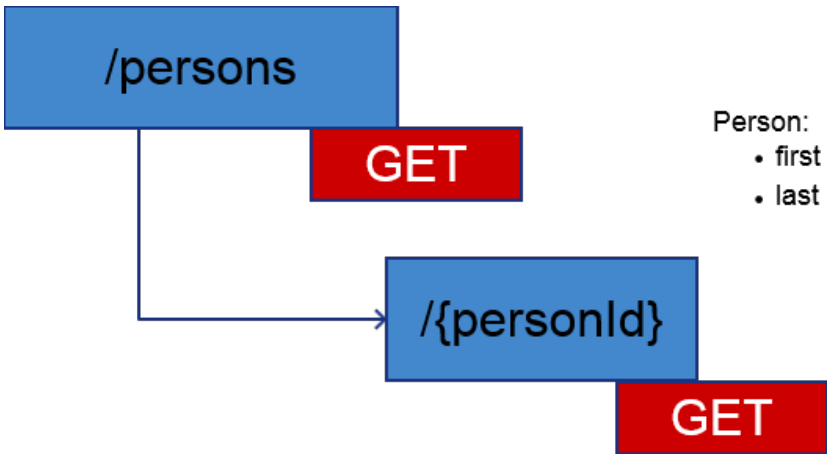

Abbildung 7.1: Layout der Person REST-Schnittstelle

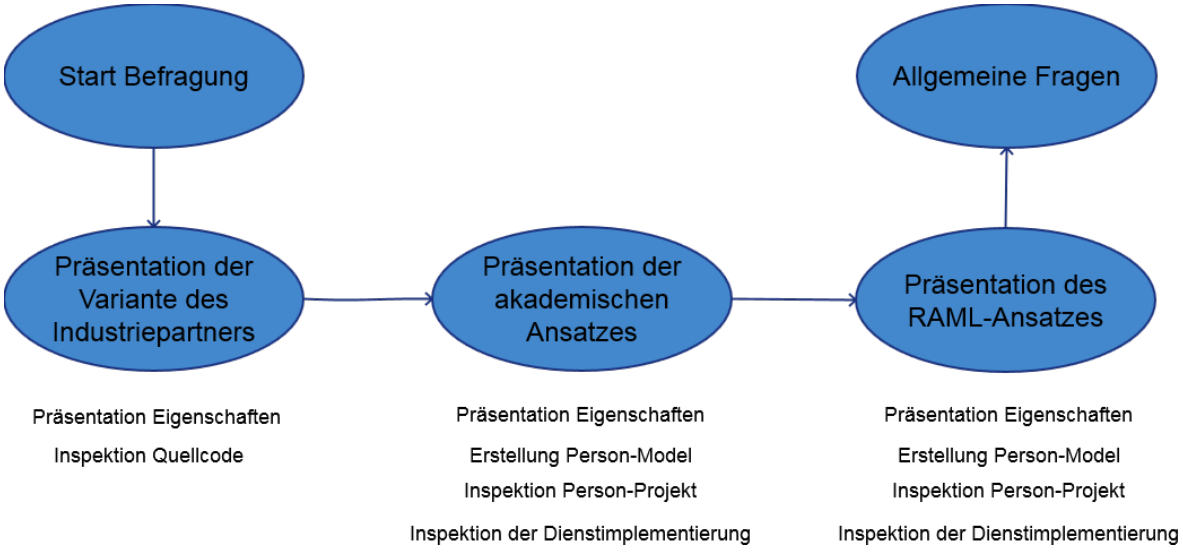

Abbildung 7.2: Ablauf der Entwicklerbefragung

aus Abbildung 7.1 mit den Domänenobjekt Person, bestehend aus Vor- und Nachname ("*first*" und "*last*"). Der Ablauf der Befragung lief wie in Abbildung 7.2 dargestellt ab. Zuerst wurde dem Entwickler die von ihm selbst mitentwickelte Lösung präsentiert um erneut Eigenschaften dieser bewusst zu machen und eine Grundlage für den späteren Vergleich zu schaffen. Die vom Industriepartner entwickelte Variante wurde als eine von Hand implementierte, auf Spring Boot basierende Lösung präsentiert. Besonderer Fokus lag hierbei auf dem bewusst machen der unterschiedlichen verwendeten Annotationen für das Definieren der REST-Schnittstelle und der Domänenklassen.

Danach wurde die akademische Lösung vorgestellt, hierzu wurden zuerst die Eigenschaften dieses Ansatzes aufgelistet und im Anschluss daran wurde, zusammen mit dem Entwickler,

das einfache REST-Layout modelliert und generiert. Der Entwickler hatte dann die Möglichkeit sich die daraus resultierende Schnittstelle genau anzuschauen und auch den generierten Code genau zu untersuchen. Nachdem die Betrachtung des gemeinsam erzeugten Projektes abgeschlossen war, wurde die in Abschnitt 5.4 vorgestellte Vergleichsimplementierung des Dienstes des Industriepartners vorgestellt. Die Präsentation war ähnlich der Erstellung, also wurde zuerst das Modell vorgestellt und im Anschluss daran die dazugehörige Implementierung.

Im Anschluss daran wurde dasselbe mittels RAML umgesetzt. Zuerst die Modellierung und Generierung der einfachen REST-Schnittstelle und im Anschluss daran die Betrachtung der mittels RAML umgesetzte Variante der Dienstimplementierung des Industriepartners.

Präsentation des akademischer Ansatzes

Der akademische Ansatz wurde als ein modellgetriebener Ansatz mit grafischem Editor vorgestellt. Es wurde hier bereits darauf hingewiesen, dass im Moment nur ein Generator für das Dropwizard Framework existiert. Neben dem Hinweis auf das Framework, wurde erwähnt, dass die später gezeigte Implementierung nicht der Qualität der vom Entwickler bereits gekannten übereinstimmt, da in diese viel mehr Zeit investiert wurde. Das repräsentative Beispiel für den akademischen Ansatz sollte lediglich die Arbeit mit der generierten Schnittstelle und den Domänenklassen aufzeigen, weitere Implementierungsdetails wie Authentifizierung und das Einbinden von Logging-Lösungen sollten nicht im Fokus der Arbeit liegen. Nachdem der Ansatz vorgestellt war, wurde zusammen mit dem Entwickler eine einfache REST-Schnittstelle modelliert und generiert. Das Layout dieser REST-Schnittstelle ist das bereits vorgestellte aus Abbildung 7.1. Das Ergebnis dieser Modellierung ist in Abbildung 7.3 ersichtlich, das daraus generierte Projekt wurde mit dem Entwickler genauer inspiziert und er konnte sich einen Eindruck machen was für Auswirkungen die Modellierung auf die daraus generierte Implementierung hat.

Nach diesem kleinen Beispiel wurde dem Entwickler das Modell der Dienst Implementierung gezeigt. Ein Bild dieses Modells ist in Abbildung 7.4 abgebildet. Der Großteil der Arbeit an diesem Modell war allerdings nicht die Erstellung des Layouts der verschiedenen Ressourcen sondern das Schema der Ressourcen, da ein *Sonderziel* viele Informationen enthält. Nachdem der Entwickler sich mit dem Modell vertraut gemacht hatte wurde ihm das dazugehörige mit dem Codegenerator erzeugte und anschließend ausimplementierte Projekt präsentiert. Bei diesem Schritt wurde besonders auf die bereits in Abschnitt 5.4 erwähnten Unterschiede zur ursprünglichen Implementierung, inklusive deren Ursachen, eingegangen. Dem Entwickler wurde nun noch 10 Minuten Zeit gegeben sich die Implementierung selbstständig anzusehen, während den 10 Minuten hatte der Entwickler ausserdem jederzeit die Möglichkeit Fragen zu stellen. Im Anschluss daran wurde der Ansatz unter Verwendung von RAML präsentiert.

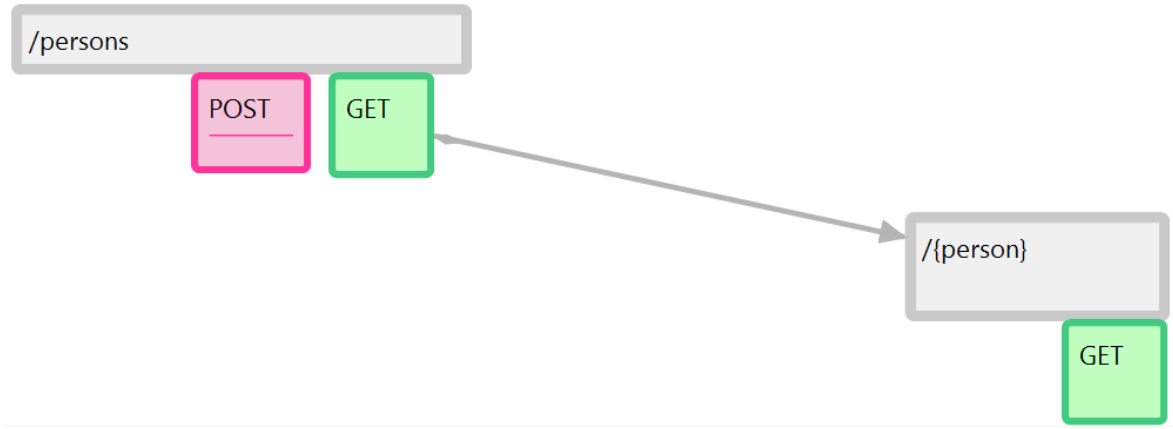

Abbildung 7.3: Akademischer Ansatz: Layout Personenschnittstelle

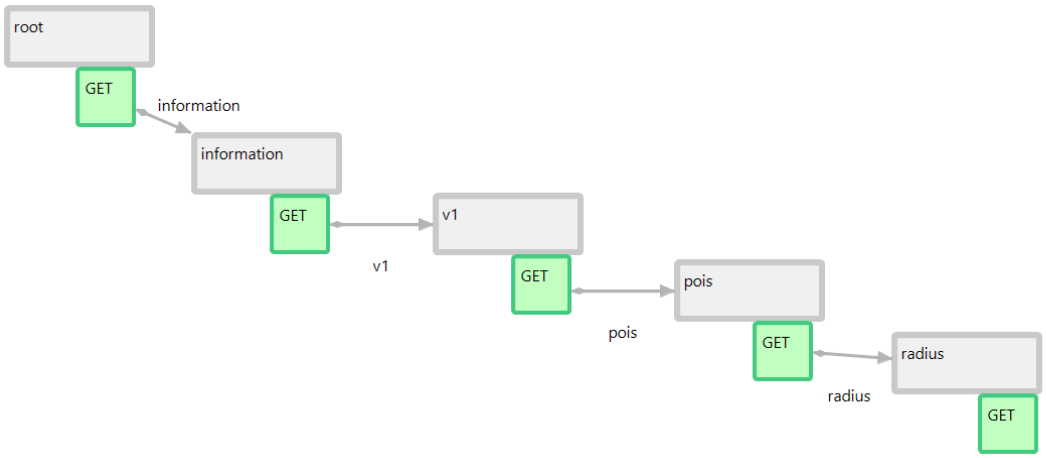

Abbildung 7.4: Layout der Information REST-Schnittstelle

Präsentation des RAML-Ansatzes

RAML wurde ähnlich dem akademischen Ansatz als modellgetriebene Variante vorgestellt. Zusätzlich wurde das RAML-System mit der Spezifikation und unterschiedlichem Tooling vorgestellt. Besonders wurde darauf eingegangen, dass der eingesetzte Generator einer von vielen ist und sich mit der Erzeugung von JAX-RS kompatiblen Schnittstellen beschränkt. Es wurde ebenso darauf hingewiesen, dass die Qualität der Umsetzung des Dienstes des Industriepartners, genauso wie bei dem akademischem Ansatz, nicht mit der des produktiv eingesetzten Dienstes des Industriepartners vergleichbar ist.

Als nächsten Schritt wurde, wie beim akademischem Ansatz, ebenfalls zusammen mit dem Entwickler ein Beispiel passend zum bereits vorgestellten Personen-Layout (siehe

Abbildung 7.1) erstellt. Das Resultat der Modellierung ist aufgrund der Spezifikationssprache ein YAML-Artefakt.

Im Anschluss an die Erstellung des Modells wurde ein, während der Durchführung der Arbeit, vorbereitetes Mavenprojekt präsentiert, was die automatische Generierung eines beliebigen RAML-Modells bereits integriert hat. Im Unterschied zu dem akademischem Ansatz verfolgt der Einsatz des RAML-JAX-RS-Generators das Ziel den generierten Code nicht in die Versionierung einzupflegen. Dem Entwickler wurde dies verdeutlicht und die Funktionsweise der Generierung wurde ihm dabei genau erläutert. Das Generieren der Schnittstelle und des Datenmodells ist in die *generate-sources*-Phase der Builderstellung integriert. Das hat zur Folge, dass die generierten Klassen lediglich im *target*-Verzeichnis des Projektes liegen und so zwar zur Kompilier- und Laufzeit verfügbar sind aber, bei richtiger Konfiguration der Versionierung, nicht Teil des versionierten Quellcodes sind. Der Vorteil bei diesem Ansatz ist, dass man die Änderungen am Modell lediglich dort hat und nicht zusätzlich an vielen Stellen der Generierung. Das Projekt wurde dann genutzt um den Quellcode für das eben erstellte Personen-Modell zu erzeugen. Nun wurde dem Entwickler noch gezeigt, wie er das Projekt ausimplementieren kann um die generierte Schnittstelle zu nutzen. Die generierte Schnittstelle wurde nun noch genau mit dem Entwickler untersucht, um ihm einen guten Einblick zu gewähren.

Nach der Erzeugung der Beispielschnittstelle wurde dazu übergegangen sich das RAML-Projekt, das für die Umsetzung der Schnittstelle des Industriepartners mittels RAML erstellt wurde, zu betrachten (vergleiche Abschnitt 5.4). Hier lag der Fokus ebenso wie bei der Präsentation des akademischen Ansatzes auf der erstellten Schnittstelle, den generiertenen Domänenklassen sowie der Umsetzung der Implementierung unter Verwendung der generierten Klassen. Dem Entwickler wurde ebenfalls die Möglichkeit gegeben sich die mit RAML erstellte Umsetzung 10 Minuten lang, mit Möglichkeit zu Rückfragen, zu betrachten.

Im Anschluss daran wurden weitere Fragen des Entwicklers beantwortet und zusammen mit ihm der Fragebogen bearbeitet.

7.4 Ergebnisse der Befragung

Dieser Abschnitt soll ein einheitliches Bild über die Befragung der Entwickler abgeben. Auf die offenen Fragen über den Grund der Bewertung wird im folgenden Abschnitt *"Analyse und Zusammenfassung"* eingegangen.

Die drei befragten Entwickler waren laut eigener Aussage allgemein gegenüber der modellgetriebenen Entwicklung nicht voreingenommen. Sie haben alle bereits mit Codegenerierung positive wie auch negative Erfahrungen sammeln können. Bei ihren Erfahrungen handelte es sich immer um das Generieren von Domänenklassen aus UML-Modellen oder aus Swagger-Definitionen. Einer der Entwickler gab hier explizit an, dass er bei Swagger-Definitionen

oftmals sich nur die Domänenklassen generieren lässt um sie in einem selbst entwickelten Client zu nutzen.

Alle Entwickler gaben an, dass sie bei der bisherigen Verwendung von modellbasierten Werkzeuge keine Präferenz gegenüber Einsatz oder Nichteinsatz der Werkzeuge hatten. Ein Vorteile den sie dabei erlebt haben, war dass das Modell einen übersichtlichen Überblick über alle Domänenklassen gab. Im weiterem Gespräch erwähnten sie, dass sie das Modell während der weiteren Entwicklung an Änderungen aus dem Quellcode anpassen mussten. Sie nutzen das Modell also zur initialen Generierung und später nur noch als getrenntes Dokumentationsartefakt, welches von Hand auf aktuellem Stand gehalten werden musste. Diesen zusätzlichen Aufwand empfanden sie auch als Nachteil. Betrachtet man diese Verwendung genau so muss man feststellen, dass es sich streng genommen hierbei nicht um modellgetriebene Entwicklung handelt - von den Entwicklern aber als solche wahrgenommen wurde.

Bei der Bewertung der Modellierungstools konnte eine starke Präferenz für das klassiche Vorgehen festgestellt werden. Es erhielt im Durchschnitt 8,3 Punkte (genauer: 9, 8 und 8 Punkte). Auf dem zweiten Platz landete die Modellierung mittels RAML, welche in der Befragung durchschnittlich 6,3 Punkte erhalten hat (7, 6 und 6 Punkte). Das Schlusslicht bildete hier knapp der akademische Ansatz mit durchschnittlich 5,6 Punkten (4, 7, 6). Auffällig war hier, dass einer der Entwickler den akademischen Ansatz mehr Punkte gegeben hat als dem Ansatz mit Verwendung von RAML.

Wenn man die Codeerzeugungen vergleicht, so ist der Unterschied der Ansätze hier noch deutlicher, als bei dem Vergleich der Modellierungswerkzeuge. Hier hatten alle Entwickler die gleiche Reihenfolge gewählt: Auf Platz 1 landete das klassische Vorgehen mit durchschnittlich 9,3 Punkten (9, 9, 10), gefolgt von RAML mit durchschnittlich 3,6 (4, 4, 3) und dem akademischen Ansatz mit durchschnittlich 2,3 (3, 2, 2). Die Platzierungen der verschiedenen Ansätze sowohl für die Modellierungswerkzeuge als auch für den Code und den Workflow sind in Abbildung 7.5 visualisiert.

In der darauffolgenden Frage nach dem Grund gaben die Entwickler an, dass ihnen am bestehenden Ansatz die volle Kontrolle über den Quellcode sehr zu sagt. Er ist technisch unabhängig und leicht zu verstehen. Die beiden anderen Ansätze seien schwerer zu verstehen und unübersichtlicher. Der Grund für das bessere Abschneiden des Ansatzes unter Verwendung von RAML im Vergleich zum akademischen Ansatz war laut ihnen nicht die Qualität des Quellcodes. Letztendlich gefiel ihnen der Workflow besser. Der Ansatz unter Verwendung von RAML ermöglicht es den generierten Code innerhalb des Projektes regelmäßig erneut zu generieren. Dabei schafft er es durch die Trennung von generierten und nicht generierten Code eine saubere Grundlage für die Versionierung zu legen. Als Nachteile der modellgetriebenen Ansätze konnten sie die Bindung an das vom Generator verwendete Framework identifizieren.

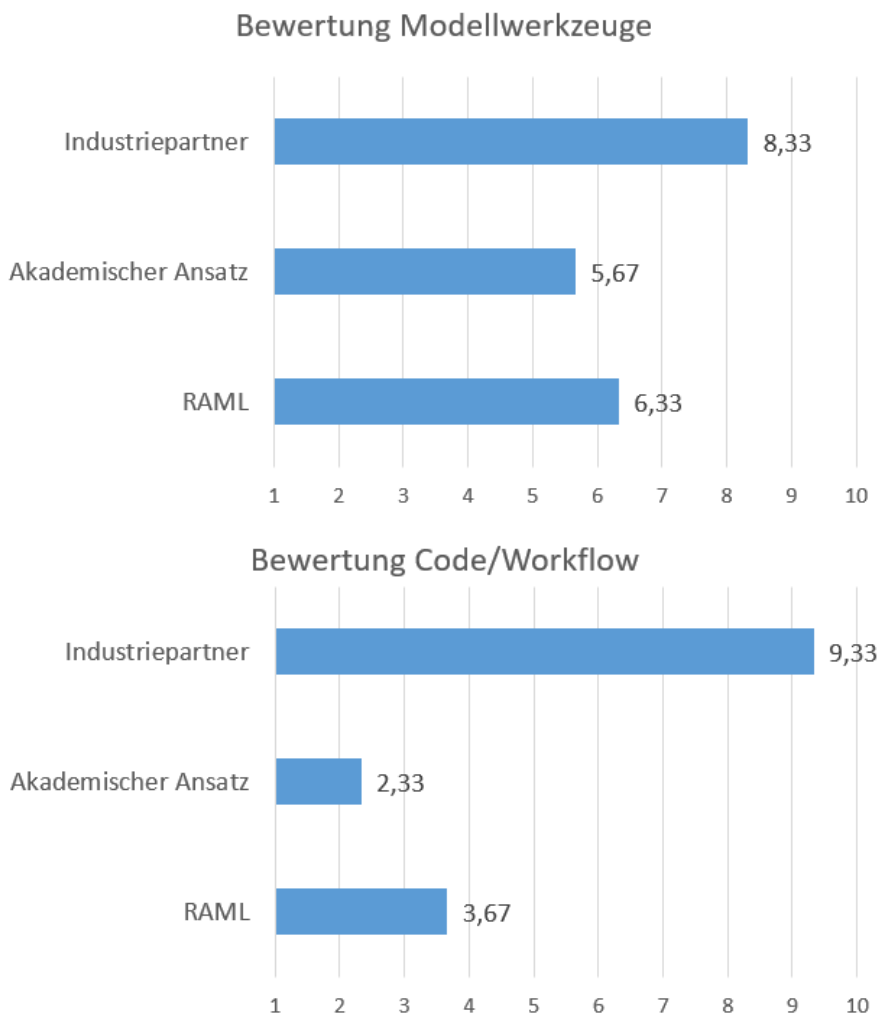

Abbildung 7.5: Auswertung der Punktevergabe der Entwickler

Nach dem Vergleich und der Betrachtung aller Varianten äußerten die Entwickler auf die Frage für welche Einsatzzwecke sie sich Teile der Werkzeuge vorstellen könnten, dass die grafische Ansicht des akademischen Ansatzes einen gewissen Reiz hat, da sie die technischen Informationen gut graphisch visualisiert. Sie schränkten diese Aussage allerdings direkt wieder ein indem sie daraufhin ergänzten, dass in einem Umfeld der Microservices, aufgrund der Einfachheit der Schnittstellen, der Mehrnutzen einer solchen Visualisierung sehr gering ist.

Als mögliche Vorteile der modellgetriebenen Ansätze nannten die Entwickler eine mögliche schnellere Umsetzung einer REST-Schnittstelle als unter Verwendung ihrer bisherigen Werkzeuge. Sie waren allerdings skeptisch gegenüber dem langfristigem Einsatz beider modellgetriebener Ansätze, da ihnen zum einen die Codequalität nicht zusagte und sie der

Meinung waren, dass Änderungen sowohl am Modell als auch am Metamodell zu aufwendig sind. Sie verglichen die Ansätze mit ihrer Lösung eines deskriptiven Swagger-Modells und konnten für sich in der Phase der Weiterentwicklung der Schnittstelle keine wirklichen Vorteile erkennen. Auf die Frage was eine Toolchain für den Entwurf und die Umsetzung von REST-Schnittstellen können müsste antworteten zwei der Entwickler, dass sie ungern die Hoheit über den generierten Code abgeben würden. Auf Frage nach dem Grund nannten sie, dass sich ändernde Anforderungen damit wohl nicht so frei umsetzten lassen würden - oder sie zumindest Bedenken hätten alle Anforderung mit dem generiertem Code umzusetzen.

7.5 Untersuchung mittels statischer Codeanalysewerkzeuge

Um zusätzlich zum subjektiven qualitativen Feedback der Entwickler noch eine zusätzliche objektive Aussage festzuhalten wurden die erzeugten Code-Artefakte mittels statischer Codeanalysen untersucht.

Die Wahl des einzusetzenden Werkzeugs für diese Arbeit ist auf SonarQube[1] gefallen. Grund hierfür ist die Aggregation von vielen unterschiedlichen Analyseverfahren wie die Überprüfung der Einhaltung von Richtlinien (*Checkstyle*), Bad practices und mögliche Programmfehler kombiniert und so einen sehr guten Gesamteindruck über den Zustand einer Software gibt. Neben der Funktionalität überzeugt SonarQube auch durch die Verbreitung in der Industrie, es ist wohl das meist benutzte Werkzeug für die Verwaltung der Codequalität.

In diesem Abschnitt werden zuerst die eingesetzten Metriken und Methodiken genannt, beschrieben und den Grund für deren Einsatz erklärt. Im Anschluss daran werden die Ergebnisse der statischen Codeanalyse gezeigt und detailliert analysiert - hier wird dann auch auf die mögliche Interpretation der einzelnen Metriken und Methoden, sowie auf das präsentierte Gesamtbild eingegangen.

7.5.1 Eingesetzte Metriken und Methodiken

SonarQube bietet einige Möglichkeiten zu Analyse von Softwareprojekten an. Im Detail wurden diese schon in den Grundlagen beschrieben, deshalb folgt hier lediglich kurz die Nennung der verwendeten Methoden und Metriken.

Für die mit den verschiedenen Ansätzen erstellten Projekte wurde jeweils ein, auf die REST-Ressourcen und Modelklassen eingeschränkter, Analyselauf durchgeführt. Grundlage für

[1] SonarQube: http://www.sonarqube.org/

die spätere Auswertung waren die von SonarQube gelieferten Ergebnisse in den Kategorien Zuverlässigkeit (Reliability), Sicherheit (Security), Wartbarkeit (Maintainability), Duplikate (Duplications), Größe (Size) und Komplexität (Complexity). Die eben genannten Metriken sind auf der entsprechenden Seite in der SonarQube Dokumentation [2] genau erläutert. Die Kategorien werden im Testlauf durch die dazu passenden Metriken repräsentiert. Die Zuverlässigkeit wurde im Testlauf durch die Metrik "Bugs" überprüft. Die Bugs-Metrik hat es nicht in die Ergebnistabelle geschafft, da keiner der Ansätze einen Bug aufgewiesen hat. Die Basis für die Kategorie Sicherheit stellt die Anzahl der Schwachstellen dar. Die Wartbarkeit wurde durch die beiden Metriken "Code Smells" und Technische Schuld abgedeckt. Die Anzahl der duplizierten Codestellen ist der Kategorie Duplikate zuzuordnen. Die Größe wurde in diesem Test durch das einfache Zählen der Anzahl der Quellcodezeilen erhoben. Die letzte Kategorie, die Komplexität, wurde direkt übernommen. Diese Metrik zählt die Anzahl der Verzweigungen im Quellcode. Neben den eben genannten Kategorien liefert SonarQube zusätzlich noch weitere Werte, welche nicht für die Analyse genutzt wurden. Zum einen ist das die Kategorie Dokumentation (Documentation), welche den Anteil an Kommentaren im Quellcode aufzeigt und zum anderen die Summe aller Probleme (Issues). Der Dokumentationsgrad ist in der Literatur sehr umstritten. Viele Entwickler vertreten die Meinung, dass Kommentare eher vermieden werden sollten und durch sprechende Methoden und Variablennamen ersetzt werden können. Ein niedriger Dokumentationsgrad kann so also auch ein Indikator für sehr hohe Codequalität sein. Die andere nicht verwendete Kategorie sind die zusammengefassten Probleme (Issues), da sie eigentlich eine Zusammenfassung der bereits vorliegenden Befunde sind und so keine zusätzlichen Informationen bieten.

7.5.2 Resultate der Analyseläufe

Dieser Abschnitt zeigt die Ergebnisse der Analyseläufe, eine Analyse der Ergebnisse wird im darauffolgendem Abschnitt durchgeführt. Die folgende Beschreibung der Ergebnisse ist in der Abbildung 7.6 visuell dargestellt.

Bei den Schwachstellen ist lediglich der Ansatz unter Verwendung von RAML aufgefallen, er weist 3 Schwachstellen auf und ist damit der einzige Ansatz der überhaupt Schwachstellen aufweist. Die Code Smells reichen von den 34 der Implementierung des Industriepartners über 136 bei der generierten Variante unter Verwendung des akademischen Ansatzes bis zu 393 bei dem erzeugtem Quellcode unter Einsatz des RAML-Toolings. Die technische Schuld ist erneut beim RAML-Ansatz am höchsten und beträgt dort drei Tage, gefolgt von der technischen Schuld der Lösung des akademischen Ansatzes mit einem Tag. Am besten hat hier erneut die Lösung des Industriepartners abgeschnitten - die technische Schuld dieser Lösung beträgt zwei Stunden. Betrachtet man den prozentuallen Anteil an dupliziertem Code

[2] SonarQube Dokumentation:
http://docs.sonarqube.org/display/SONAR/Metric+Definitions#MetricDefinitions-Reliability

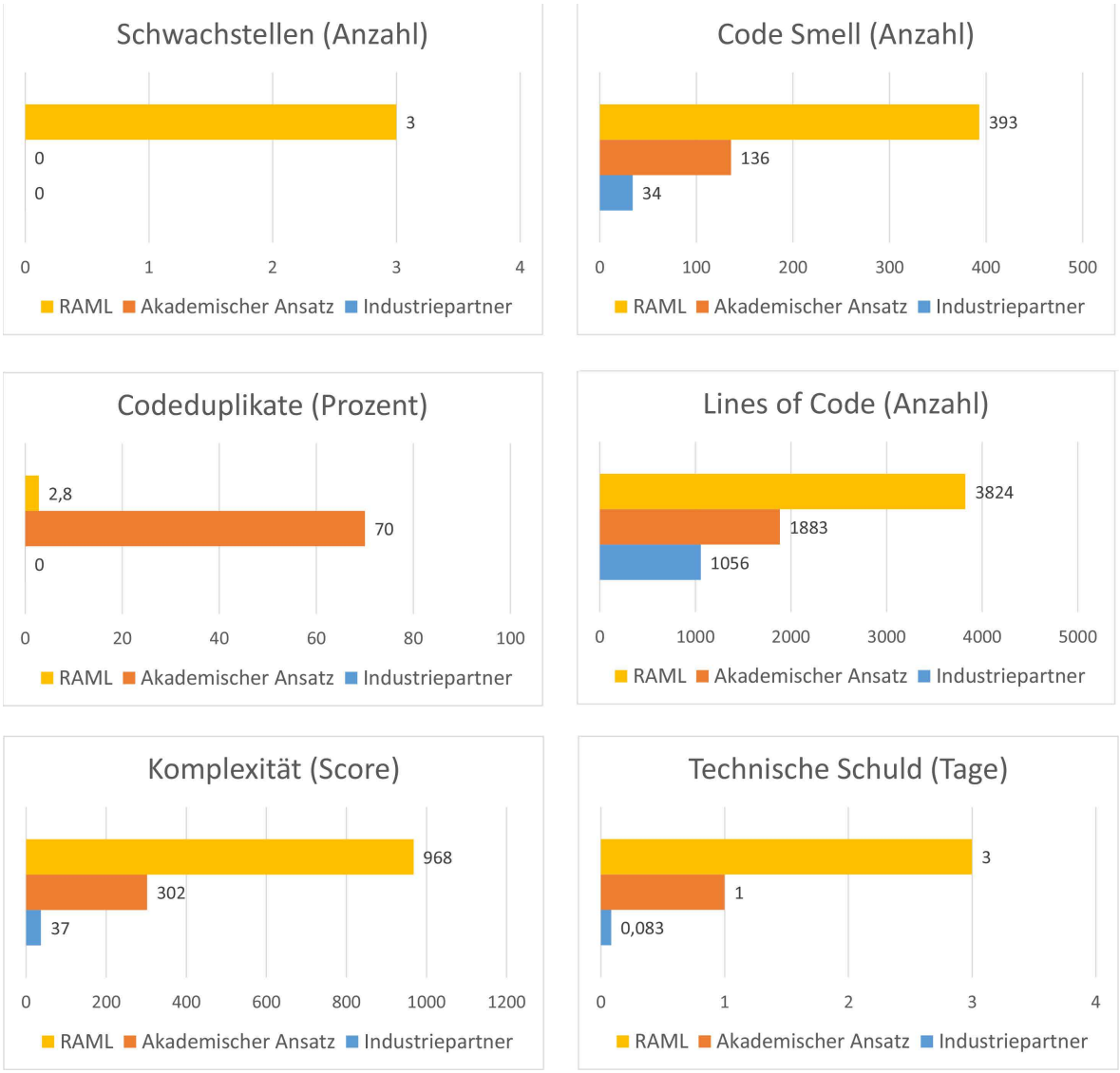

Abbildung 7.6: SonarQube-Ergebnisse

so fällt die Variante unter Einsatz des akademischen Ansatzes aus der Reihe. Wohingegen die Lösung des Industriepartners (0%), sowie die mittels RAML erstellte Lösung (2,8%) frei oder beinahe frei von erkannten Codeduplikaten sind, so hat die mittels des akademischen Ansatzes erstellte Lösung eine sehr hohe Quote (70%). Der Umfang der Lösungen ist sehr unterschiedlich, die kompakteste Variante bildet die Lösung des Industriepartners mit 1056 Zeilen Code, gefolgt von der Lösung mittels des akademischen Ansatzes mit 1883 Zeilen Code, das Schlusslicht bildet hier die mittels RAML erzeugte Variante welche einen Umfang von 3824 Zeilen Code hat. Bei der gemessenen Komplexität ergibt sich ein ähnliches Bild: Hier ist die Lösung mittels RAML die komplexeste mit einer Punktzahl von 968, gefolgt von

der Lösung des akademischen Ansatzes mit 302, die niedrigste Komplexität hat die vom Industriepartner erstellte Lösung mit einer Punktzahl von lediglich 37.

7.5.3 Analyse der Ergebnisse

Bereits bei der Betrachtung der Resultate ist aufgefallen, dass die Lösung mittels RAML die einzige ist welche Schwachstellen aufweist. Schaut man sich das Ergebnis des Analyselaufs genauer an, so fällt auf dass es sich bei den Schwachstellen mehrmals um das Werfen einer generischen Ausnahme (Exception) in den REST-Ressourcen handelt. Scheinbar ist die Fehlerbehandlung im generierten Code des RAML-Codegenerators nicht perfekt und sollte für einen produktiven Einsatz verbessert werden.

Die Anzahl der Code Smells zeigt eine deutliche Abstufung zwischen den einzelnen Lösungen auf. Die vom Industriepartner erzeugte Lösung hat mit Abstand am wenigsten. Bei genauerer Betrachtung des akademischen Ansatzes fällt auf, dass viele der 136 Codesmells von ähnlichem Typ sind. Besonders häufig treten dabei ungenutzte Importe (23-mal), fehlerhafte Variablen- und Klassenbenennung (17-mal), sowie der veraltete Einsatz des Diamond-Operators (41-mal) auf. Allein mit der Behebung dieser recht simplen Änderungen im Generator könnten die Codesmells auf 55 reduziert werden - was schon eher an die vom Industriepartner entwickelte Variante herankommt. Die 393 Codesmells des generierten Code des RAML-Codegenerators sind zum Großteil auf den Einsatz von unnötigen Klammern (244-mal), Duplizierung von Stringliteralen (40-mal) und der nicht korrekten Reihenfolge von Modifikatoren (29-mal) zurückzuführen. Der Generator erweckt den Eindruck, dass er zur Sicherung gegen Syntaxfehler eine Klammer zuviel benutzt. Durch eine Beseitigung der Ursachen dieser Codesmells im Generator könnte für diesen Fall die Anzahl der Codesmells beachtlich reduziert werden - von anfänglich 393 wären dann gerade noch 80 übrig.

Die technische Schuld setzt sich aus den gerade erläuterten Codesmells und Duplikaten zusammen. Sie gibt aber ein genaueres Bild über den Zustand des Codes an, da die einzelnen Codesmells unterschiedlich schwer auszumerzen sind. Die technische Schuld aggregiert somit die geschätzte Zeit, welche ein Entwickler für das Ausmerzen aller Codesmells und Codeduplikate benötigen würde. Sie ist in Kombination mit der Anzahl der Codesmells viel aussagekräftiger als die pure Anzahl an Codesmells alleine. Auch wenn die Anzahl der Codesmells nicht unbedingt im Verhältnis zur technischen Schuld stehen muss, so tut sie es in diesem Fall. Die Implementierung des Industriepartners besitzt eine technische Schuld von zwei Stunden, wohingegen die generierten Lösungen eine höhere technische Schuld aufweisen - die technische Schuld für die Lösung mittels des akademischen Ansatzes beträgt einen Tag und die der mittels RAML erstellten Lösung drei Tage.

Ein Maßstab für den Umfang des Quellcodes ist die Anzahl der Codezeilen. Wie bei den anderen Metriken spiegelt sich hier ein ähnliches Bild wider. Die vom Industriepartner von Hand geschriebene Implementierung ist mit 1056 Codezeilen mit Abstand am kürzesten.

Den zweiten Platz dieser Metrik belegt der erzeugte Code des akademischen Ansatzes mit 1883 Zeilen. Am längsten ist der erzeugte Code des RAML-Codegenerators mit insgesamt 3824 Zeilen. Zur Betrachtung hierfür ist eine weitere Metrik interessant und aufschlussreich: Die Codeduplikate. Man sieht im Schaubild, dass die Codeduplikate für die Implementierung des Industriepartners (0%), sowie für die mittels RAML-Tooling (2,8%) erzeugten Code vernachlässigbar sind. Anders hingegen ist dies beim akademischen Ansatz - hier beträgt die Codeduplikatrate beachtliche 70%. Wenn man eine Erklärung hierfür sucht wird man schnell fündig. Das untersuchte Beispiel benutzt für zwei Ressourcen dasselbe Datenmodell. Bei der händischen Implementierung und der mittels RAML erzeugten Variante gibt es für beide Ressourcen ein gemeinsames Datenmodell, bei dem akademischen Ansatz allerdings nicht. Da in der untersuchten Schnittstelle das Datenmodell im Vergleich zur REST-Schnittstelle relativ groß ist bedeutet dies auch einen erheblichen Anteil an Codeduplikaten. Der akademische Ansatz sollte in Zukunft das Datenmodell außerhalb der Ressourcenklassen generieren. Durch diese Maßnahme wären die unterschiedlichen Ressourcen zu einander kompatibel und man könnte Hilfsklassen, Parser und andere Komponenten für beide Ressourcen nutzen. Ein positiver Nebeneffekt wäre in diesem Beispiel die Reduzierung der Länge des erzeugten Codes um etwa 850 Zeilen. Somit würde die Länge auf in etwa 1030 Zeilen zurückgehen und wäre somit potentiell sogar kürzer als die händisch entwickelt Variante. Um allerdings gerecht zu bleiben muss man gestehen, dass die von Hand geschrieben Variante einige Zeilen nutzt um die Rest-Ressourcen sowie das Datenmodell für die Dokumentation mit Swagger zu annotieren, wenn man diese Annotationen beim akademischen Ansatz einfügen würde, so wäre diese in etwa 50 Zeilen länger als die Implementierung des Industriepartners. Der Ausreißer in dieser Metrik ist wohl der vom RAML-Codegenerator erzeugte Quellcode mit ganzen 3824 Zeilen Quellcode. Dies liegt daran, dass die erzeugten Modellklassen viel größer sind als bei den anderen Ansätzen. Ursache hierfür sind zum einem erzeugte Hilfsfunktionen und generische Getter, welche den Umgang mit den Modellklassen praktischer gestalten, und zum anderem viele Annotationen welche Eigenschaften zur Serialisierung und Deserialisierung explizit angeben und so robuster gegenüber Eigenheiten der verwendeten JSON-Bibliothek sind.

Die verschiedenen Metriken geben ein recht eindeutiges Bild ab. Die statische Codeanalyse hat Schwächen der beiden Generatoren für die modellgetriebenen Ansätze aufgedeckt und die Überlegenheit der vom Industriepartner entwickelten Lösung aufgezeigt. Die Ergebnisse der statischen Codeanalyse decken sich also mit der Meinung der Entwickler in der Befragung. Sie haben darüber hinaus aber auch gezeigt, dass durch gegebenenfalls sehr einfache Anpassungen an den Generatoren die Codequalität des generierten Codes stark gesteigert werden kann.

Gerechterweise kann man nun anmerken, dass die Codequalität in einem modellgetriebenen Ansatz eher sekundär ist, solange der generierte Code die geforderte Funktionalität besitzt. Aber so einfach ist es nicht - man muss bedenken, dass Schwächen wie in diesem Kapitel aufgezeigt auch eine Fehlersuche im Quellcode erschweren. Wenn man einen modellgetriebenen Ansatz einführt, so wird man dazu übergehen Fehler im Generator zu suchen und

auszumerzen. Bei all diesen Tätigkeiten ist eine möglichst gute generierte Implementierung sehr erstrebenswert.

7.6 Analyse und Zusammenfassung

Die Entwickler waren in der Befragung generell von der bestehenden Lösung des Industriepartners überzeugt. Prinzipiell ist dies nicht sehr überraschend, da sie diese Lösung selbst nach ihren speziellen Anforderungen entwickelt haben. Dennoch äußerten sie gerechtfertigte Kritik an den vorgestellten modellgetriebenen Ansätzen. Sie beschrieben, dass die Kontrolle über den letztendlich entstehenden Code für sie von höchster Wichtigkeit ist und sie ungern die Kontrolle über die Definition der Schnittstelle an einen Generator mit ungewissen Regeln abgeben würden. Neben der generellen Kontrolle empfanden sie auch die Bindung an die unterschiedlichen Tools und Frameworks (z.B. Eclipse, Dropwizard und Jersey) für sehr störend. Weitere Tools sind bei der täglichen Arbeit eher störend und die Bindung an ein bestimmtes Framework kann bei der Umsetzung spezieller Anforderungen schnell zu einem Problem werden. Die größten Kritikpunkte waren letztendlich, dass die Entwickler sich bei der Nutzung verunsichert über das Ergebnis der Modellierung fühlten und dass der generierte Code im Anschluss nicht dem entsprach, was sie selbst implementiert hätten. Die Entwickler beschrieben dies als ein fehlendes Vertrauen in das Mapping der Generatoren, für sie war es auch schwer vorherzusagen was für Datentypen in dem später, auf Basis des Modells, generierten Quellcodes genutzt werden. Laut ihrer Aussage war dies vor allem beim akademischen Ansatz der Fall, aber auch bei RAML war diese Ungewissheit vorhanden. Sie meinten, dass RAML eine bessere technische Übersicht über die Schnittstelle gibt als das bei dem akademischen Ansatz der Fall ist, da bei ihm Parameter und Datentypen nicht direkt im Schaubild ersichtlich sind. Für die Erkennung des Layouts einer Schnittstelle sei der akademische Ansatz aber mit am besten geeignet.

Bei der Erstellung des Datenmodells waren die Entwickler recht verhalten und der Meinung, dass sie weder Vor- und Nachteile für die Generierung des Datenmodells sehen. Die Generierung von Hand ist recht schnell, erfordert relativ wenig Denkarbeit und ist deshalb nicht all zu fehlerbehaftet. Der Aufwand ist nicht so hoch und die Tätigkeit nicht so schwer, dass die Entwickler hierbei unbedingt Unterstützung durch ein Werkzeug benötigen.

Bei der Betrachtung der von den Codegeneratoren erzeugten Quellcodes haben die Entwickler auf den ersten Blick Mängel hinsichtlich der Einhaltung der Programmierrichtlinien, beispielsweise bei der Groß- und Kleinschreibung von Variablennamen, entdeckt. Nach diesem ersten negativen Eindruck waren die Entwickler, ihrer Aussage nach, etwas voreingenommen und haben ihre Zweifel in die möglicherweise nicht ganz einwandfreie Übersetzung der Modelle in Quellcode bestätigt gefühlt. Der erste Eindruck der Entwickler ließ sich in

den Ergebnissen der statischen Codeanalyse wieder finden. Die statische Codeanalyse bestätigte, dass die Codegeneratoren komplexeren, längeren und mit höherer technischer Schuld belasteten Code generieren.

Im Allgemeinen sahen die Entwickler wenig Vorteile in der modellgetriebenen Erzeugung einer REST-Schnittstelle im Vergleich zu der bisherigen Methode. Für sie ist es komfortabler eine Schnittstelle genau so zu implementieren, wie sie es sich vorstellen und dabei, oder im Anschluss daran, die bestehenden Methoden mit dokumentierenden Annotationen zu versehen. Ihrer Aussage nach haben sie mit diesem Ansatz das beste beider Welten, eine immer aktuelle Dokumentation und volle Kontrolle über die technische Definition der Schnittstelle selbst. Bei der Wahl des eingesetzten Dokumentationsframeworks haben sie darauf geachtet, dass die Dokumentation sinnvoll in einem Browser anzeigbar ist und zusätzlich schnell die Generierung eines passenden Clients in mehreren Sprachen erlaubt. Swagger als Lösung für die Annotationen der Schnittstelle wird ihrem Anspruch dabei gerecht.

Nichtsdestotrotz muss man festhalten, dass die modellgetriebenen Ansätze auch Vorteile besitzen. Die Entwickler gaben an, dass sie eine grafische Ansicht der Schnittstelle in manchen Situationen für sehr nützlich halten würden. Gerade die Darstellung in einem Graph oder zumindest in einem Baum lasse sehr schnell Rückschlüsse auf das tatsächliche Layout zu, was andere Ansätze nicht ermöglichen. Im Kontrast zu dem vorgegebenen akademischen Ansatz würden sie es aber unterstützen, wenn eine solche Darstellung auf Basis ihres Quellcodes erzeugt werden würde und nicht die Grundlage für weitere Modellierung und spätere Übersetzung in Quellcode bilden würde.

Bei der Durchführung der Arbeit ist aufgefallen, dass die drei Ansätze für verschiedene Einsatzzwecke unterschiedlich gut geeignet sind. Dies wurde durch die Befragung der Entwickler deutlich und durch die Durchführung des AHP bestätigt. Der Ansatz des Industriepartners ist in der Praxis sehr gut, solange man keine sehr große Schnittstelle umsetzen will. Bei einer sehr großen Schnittstelle ist der Ansatz vermutlich weniger geeignet, da es nicht mehr so schnell möglich sein wird eine erste Version der Schnittstellendefinition zu erzeugen. Die Schnittstellendefinition basiert auf einer ersten Implementierung, welche sich bei einer größeren Schnittstelle verzögern könnte.

Die Alleinstellungsmerkmale des akademischen Ansatzes sind sicherlich die Möglichkeiten zur grafischen Modellierung und zur Umsetzung des HATEOAS-Prinzips. Er eignet sich so prinzipiell für besonders große Schnittstellen mit vielen Ressourcen. Im jetzigen Zustand wird man aber vermutlich für die Verwendung in der Praxis einen eigenen Codegenerator zu schreiben bzw. den aktuellen anzupassen.

RAML hat von den Ansätzen die besten Ansätze für Wiederverwendbarkeit und Wiederverwendung. Dieser Ansatz eignet sich von der Art der Spezifikation für fast alle Schnittstellen, da gemeinsam genutztes Verhalten, wie beispielsweise das in der Arbeit vorgestellte Collection/Collection-Item Pattern, auch für kleine Schnittstellen einfach importiert werden

könnte. Was die Codegeneration angeht ist der Zustand ähnlich wie beim akademischen Ansatz - auch hier wird man das bestehende Tooling an die eigenen Bedürfnisse anpassen müssen.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt wie eine bestehende Implementierung eines Industriepartners in einem agilen Entwicklungsprozess mittels modellgetriebener Werkzeuge umgesetzt werden kann. Im Laufe der Arbeit wurde die, auf Spring Boot basierende, bereits bestehende REST-Schnittstelle unter Verwendung zweier unterschiedlicher Werkzeuge und Methoden (RAML und der akademische Ansatz) umgesetzt. Um die Ergebnisse dieser Arbeit auf andere Unternehmen mit anderen Entwicklern anwendbar zu machen wurde zu Beginn der Arbeit ein Vergleich des vorliegenden Scrum-Entwicklungsprozesses mit dem Referenzscrumprozess durchgeführt. Dieser Vergleich zeigte, dass trotz minimaler Abweichungen der Prozess dennoch fast vollständig dem Referenzprozess und dem Gedanken von Scrum entspricht und somit gut mit anderen Scrumprozessen verglichen werden kann.

Neben der Erstellung der REST-Schnittstelle mittels zweier modellgetriebener Methodiken beschäftigte die Arbeit sich desweiteren mit dem Vergleich dieser, dann in Summe drei, Varianten. Während der Erstellung der zwei modellgetriebenen Varianten konnten schon einige Eindrücke, sowie Stärken und Schwächen der unterschiedlichen Varianten gesammelt werden. Die entsprechenden Ergebnisse wurden im Abschnitt 7.2 gesammelt.

Anschließend dazu wurde eine Befragung der Entwickler des Industriepartners durchgeführt, bei der sie die drei Varianten inklusive einiger Beispiele betrachten und bewerten konnten. Die Durchführung und Ergebnisse der Befragung, sowie eine statische Codeanalyse der verschiedenen Codeartefakte, bilden den Rest von Kapitel 7. In diesem Kapitel wurde deutlich, dass der Einsatz von Modellen im Allgemeinen - der Industriepartner benutzt ja selbst Swagger für die Beschreibung der Schnittstelle - sehr sinnvoll ist.

Die Befragung der Entwickler brachte ans Licht, dass die Nutzung von modellgetrieben Werkzeugen für die Erzeugung von REST-Schnittstellen aus Sicht der befragten Entwickler noch nicht zufriedenstellend umgesetzt worden ist. Die Entwickler waren sich zwar einig, dass man sich vor der Implementierung einer Schnittstelle Gedanken über die Umsetzung machen müsse, aber sahen die Erstellung einer genauen Spezifikation als einen zu hohen Aufwand und eine zu hohe Verpflichtung an. In der agilen Softwareentwicklung, bei der in vielen Fällen alle zwei Wochen ein neues Inkrement geliefert wird und während der Entwicklung ständig neue Erkenntnisse gewonnen werden, hat ihrer Meinung nach eine Spezifikation eher eine dokumentierende Aufgabe ohne Anspruch auf zukünftige Einhaltung. Ab eines gewissen Reifegrades, bzw. ab einer gewissen Anzahl an Nutzern, ist eine Berücksichtigung von Kompatibilitäten unabdinglich, ob dies in einem deskriptiven (wie

Swagger) oder einem präskriptiven Modell (einem der anderen beiden Ansätzen) geschieht ist prinzipiell egal, da beide Modellarten den Ist-Zustand darstellen oder widerspiegeln.

Die Arbeit hat gezeigt, dass in dem vorliegendem Fallbeispiel die modellgetriebenen Ansätze für die Entwickler kein zufriedenstellendes Ergebnis lieferten. Sie haben kaum einen Mehrwert in der Arbeit mit den modellgetriebenen Ansätzen gesehen. Zu Beginn der Arbeit wurden die Vorteile der modellgetriebenen Softwareentwicklung genannt. Viele der zu Beginn der Arbeit angesprochenen Vorteile von MDSD konnten bei der Generierung von REST-Schnittstellen nicht recht ausgenutzt werden. Die Gründe hierfür sind vielseitig: Einerseits konnte die erhoffte Steigerung der Produktivität nicht erreicht werden, da die Entwickler sehr erfahren im Umgang mit dem eingesetzten Spring Boot Framework waren. Andererseits ist der Vorteil der erhöhten Codequalität nicht in Erscheinung getreten, da die Generatoren, nach Eindruck der Entwickler und auch nach den Ergebnissen der statischen Codeanalyse, schlechteren Code erzeugen als die Entwickler selbst.

Die Entwickler stellen für ihre Arbeit zwei große Anforderung an den Einsatz ihrer Werkzeuge: Zum einen benötigen sie im Umfeld ihrer Arbeit die Möglichkeit den Konsumenten ihrer Schnittstelle eine Dokumentation inklusive der Möglichkeit zur Erzeugung unterschiedlicher Clients bereit zustellen. Die andere Anforderung ist, dass sie die volle Kontrolle über den Quellcode und die Wahl eines Frameworks benötigen. Dies ist vor allem darauf zurückzuführen, dass zu Beginn des Projektes noch nicht alle Anforderungen an die Schnittstelle bekannt sind. Oft kommt es vor, dass im Laufe des Projektes weitere Anforderungen, wie das Monitoring, Logging, besondere Fehlerbehandlung oder spezielle Authentifizerungsarten, hinzukommen. Im Falle einer generierten REST-Schnittstelle ist es dann oftmals schwierig oder unmöglich diese Anforderungen ohne die Anpassung des Codes (oder des Codegenerators) umzusetzen.

Insgesamt muss man also sagen, dass die beiden präskriptiven Varianten zwar auf die Anforderung einer Beschreibung für die Schnittstelle eingehen aber aufgrund des erzeugten Codes keine gänzlich zufriedenstellenden Ergebnisse liefern. Beim betrachteten Anwendungsfall macht es keinen großen Sinn ein Modell zum Selbstzweck, also ohne die Verwendung von späterer Codegenerierung, einzuführen. Der betrachtete Dienst ist nicht umfangreich genug und rechtfertig solch ein Modell nicht. Ein solches präskriptives Modell selbstständig, ohne den Einsatz von Codegeneratoren umzusetzen, macht hier auch nur bedingt Sinn da diese Art zwangsläufig zu zwei zu pflegenden Artefakten führt. Die dabei entstehenden Artefakte (Modell und der dazugehörige Quellcode) sind getrennt voneinander zu pflegen, was so zu einem doppelten Wartungsaufwand führt. Sollten Änderungen am Quellcode nicht am Modell nachgezogen werden, so entstehen Widersprüche, welche nicht einfach erkennbar sind aber erhebliche Auswirkung für die generierten Clients der REST-Schnittstelle haben.

Der Einsatz von Swagger als deskriptives Modell für die manuell erstellte REST-Schnittstelle ermöglicht eine an den aktuellen Code gebundene Generierung von Clients und einer Dokumentation. Diese Art von Modell hat den Vorteil, dass kaum Diskrepanzen zwischen Modell und eigentlicher Implementierung entstehen können, da die Implementierung Grundlage

des Modells ist. Die händische Umsetzung und anschließende Erzeugung eines Modells hat auch den weiteren Vorteil, dass die Entwickler die volle Kontrolle über den entstehenden Quellcode haben. Diese Kontrolle führt zu mehr Freiheit bei der Umsetzung von Anforderungen und einer höheren Qualität in Form von Konformität des Quellcodes gegenüber selbst bestimmten oder ausgewählten Richtlinien.

Das sind im Wesentlichen auch die Gründe des Industriepartners für die bisherige Entscheidung für dieser Variante. Während dieser Arbeit hat sich diese Wahl im Vergleich mit den anderen beiden modellgetriebene Ansätzen bewährt.

Ausblick

Die Verwendung von modellgetriebenen Ansätzen zum Entwurf und der Realisierung von REST-Schnittstellen konnte die Entwickler des Industriepartners nicht überzeugen. Die Frage nach den Gründen wurde bereits im vorherigen Abschnitt diskutiert. Zusammenfassend kann man sagen, dass die Nutzung eines modellgetriebenen Ansatzes einen ähnlichen Aufwand mit sich bringt, aber ein qualitativ schlechteres Ergebnis zur Folge hat. Dieser Zusammenhang schreckt die Entwickler ab, da sie letztendlich an der Funktion und Qualität ihrer Software gemessen werden und nicht an der Art wie sie Software entwickeln. Außerdem erschweren sie sich, durch die Generierung von Quellcode mit höherer technischer Schuld, ihre zukünftige Arbeit bei der Weiterentwicklung und Wartung.

Wenn man diese Arbeit kritisch betrachtet so wird man sich damit auseinander setzen müssen, dass der Vergleich zwischen den Ansätzen nicht ganz gerecht war. Die Entwickler kannten ihre eigene Implementierung bereits und sind mit der Art der Umsetzung, beispielsweise die Wahl des Frameworks, deshalb sicher sehr zufrieden. Die Entwickler waren außerdem zum einen skeptisch aufgrund der unterschiedlichen Frameworks zwischen den Ansätzen und zum anderen stark abgeschreckt von der Qualität des von den Generatoren erzeugten Quellcodes. Es wäre sicherlich interessant in einer zukünftigen Arbeit zu sehen wie eine Befragung der Entwickler ausfallen würde, wenn die vorgestellten modellgetriebenen Ansätze einen dem Code des Industriepartners sehr ähnlichen Quellcode erzeugen würde. Für die Durchführung einer solchen Befragung im vorliegendem Projekt des Industriepartners müsste für die beiden modellgetriebenen Ansätze jeweils ein SpringBoot-Codegenerator entwickelt werden.

Weiterhin muss festgehalten werden, dass es im Moment keinen Standard zur Beschreibung von REST-Schnittstellen gibt. Bei anderen Schnittstellenformaten wie Beispielsweise SOAP hat sich ein Standard (die Web Service Description Language (WSDL)) etabliert. Vermutlich ist einer der Gründe für das Fehlen eines solchen Standards, nicht etwa das Alter des REST-Architekturstils, sondern eher die fehlende Notwendigkeit. Die Verwendung einer REST Schnittstelle besitzt eine niedrigere Einstiegsschwelle für den Anwender. Viele bei

einer Kommunikation zu klärenden Parameter, wie die Wahl des Protokolls, die Definition der Operationen und Datentypen werden einem durch den HTTP-Standard, sowie gängige Best-Practices, bereits abgenommen. Einige REST-Puristen wie Roy Fielding erklären sogenannte *'out-of-band'* Informationen, welche außerhalb des Primärkanals übermittelt werden - wozu sicher auch Dokumentationen zählen, als problematisch und Indikator für fehlende Umsetzung von HATEOAS. Aus ihrer Sicht ist ein solcher Standard nicht weiter notwendig und deshalb vermeidbar. In der Praxis allerdings sind deskriptive Ansätze, wie Swagger, weitverbreitet. Der REST-Architekturstil könnte von solch einem Standard für die Modellierung stark profitieren. Keiner der vorgestellten Ansätze überprüft die erstellten Modelle auf Umsetzung von allgemein gültigen Best-Practices. Die Herausbildung eines allgemein anerkannter Standards könnte die Entwicklung von Werkzeugen, welche den Entwickler über die ledigliche Erstellung eines Modells hinaus unterstützen, extrem beschleunigen.

Prinzipiell fanden die Entwickler die modellgetriebenen Ansätze sehr interessant aber verbesserungswürdig. Um modellgetriebene Ansätze beim Entwurf und der Umsetzung von REST-Schnittstellen in der Praxis weiterzubringen wird es sicherlich notwendig sein den Entwicklern mehr als nur eine Möglichkeit geben ihre Schnittstellen zu beschreiben. Es muss ein Bewusstsein dafür geschaffen werden, dass MDSD kein Allheilmittel für alle Probleme der Entwickler ist, sondern eine Möglichkeit ihre eigene Produktivität und Leistung zu erhöhen. Diese eben erwähnte Produktivitätssteigerung bekommt man nicht geschenkt. Hierfür muss die ganze Werkzeugkette verbessert werden. Es ist eine unrealistische Erwartung, aus einer Spezifikation ein Artefakt zu generieren, welches genau so aussieht wie man es nach langer Arbeit selber umgesetzt hätte - dieser Anforderung kann man nur gerecht werden, wenn man den Generator selber geschrieben hat oder ihn seinen Ansprüchen entsprechend angepasst hat. Für Entwickler ist es wichtig ihre verwendeten Werkzeuge anpassen zu können, ganz besonders wenn sie Quellcode erzeugen, welcher unverändert später produktiv eingesetzt werden soll. Deshalb sollten Entwickler bei den unterschiedlichen Ansätzen dazu bemächtigt werden ihre Generatoren selbst zu konfigurieren oder zu verbessern um für sich passende Lösungen zu finden. Die modellgetriebene Umsetzung einer REST-Schnittstelle endet eben nicht mit der Erstellung eines Modells - sondern erst nach der Fertigstellung der tatsächlichen REST-Schnittstelle.

Abkürzungsverzeichnis

Abkürzung	Bedeutung	Erstes Vorkommen
AHP	Analytische Hierarchieprozess	79
API	Application Programming Interface	19
CD	Continous Deployment	48
CI	Continous Integration	48
HATEOAS	Hypermedia as the Engine of Application State	21
HTML	Hypertext Markup Language	21
JSON	JavaScript Object Notation	20
MDSD	Modellgetriebene Softwareentwicklung	31
npm	Node Package Manager	57
PO	Product Owner	27
RAML	Restful Api Modeling Language	34
REST	REpresentational State Transfer	13
SM	Scrum Master	27
URI	Uniform Resource Identifier	21
WADL	Web Application Description Language	35
WSDL	Web Service Description Language	101
XML	Extensible Markup Language	20

Literaturverzeichnis

[All10] S. Allamaraju. *Restful web services cookbook: solutions for improving scalability and simplicity*. O'Reilly Media, Inc., 2010 (zitiert auf S. 22, 23).

[Arm03] P. G. Armour. *The Laws of Software Process: A New Model for the Production and Management of Software*. CRC Press, 2003 (zitiert auf S. 25).

[Bad] V. Badola. *Microservices architecture: advantages and drawbacks*. URL: http://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/ (zitiert auf S. 30).

[BBV+01] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries et al. *Manifesto for agile software development*. 2001 (zitiert auf S. 25).

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. „Comparison and evaluation of clone detection tools". In: *IEEE Transactions on Software Engineering* 33.9 (2007), S. 577–591 (zitiert auf S. 40).

[BS87] V. R. Basili, R. W. Selby. „Comparing the effectiveness of software testing strategies". In: *IEEE transactions on software engineering* 12 (1987), S. 1278–1296 (zitiert auf S. 40).

[Cag] S. Caganoff. *Anypoint for APIs: An Interview with Uri Sarid*. URL: https://www.infoq.com/news/2014/02/anypoint-api-sarid (zitiert auf S. 34).

[Cre13] J. W. Creswell. *Qualitative inquiry and research design: Choosing among five approaches*. Sage, 2013 (zitiert auf S. 83).

[DOWZ15] P. Diebold, J.-P. Ostberg, S. Wagner, U. Zendler. „What do practitioners vary in using scrum?" In: *International Conference on Agile Software Development*. Springer. 2015, S. 40–51 (zitiert auf S. 26).

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Rfc 2616, hypertext transfer protocol–http/1.1*. 1999. URL: https://www.w3.org/Protocols/rfc2616/rfc2616.html (zitiert auf S. 22).

[Fie00] R. T. Fielding. „Architectural styles and the design of network-based software architectures". Diss. University of California, Irvine, 2000 (zitiert auf S. 13, 15, 20).

[Fie08] R. T. Fielding. „REST APIs must be hypertext-driven“. In: *Untangled musings of Roy T. Fielding* (2008). URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (zitiert auf S. 24).

[FKH13] N. Freed, J. Klensin, T. Hansen. *Media type specifications and registration procedures*. Techn. Ber. 2013 (zitiert auf S. 77).

[FL] M. Fowler, J. Lewis. *Microservices*. URL: http://www.martinfowler.com/articles/microservices.html (zitiert auf S. 30).

[Fow09] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009 (zitiert auf S. 29).

[GGS+] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, S. Abeck. *Best Practices for the Design of RESTful Web Services* (zitiert auf S. 75, 76).

[Gol] K. Goldsmith. *How Spotify Builds Products (Organization. Architecture, Autonomy, Accountability)*. Spotify. URL: http://de.slideshare.net/kevingoldsmith/how-spotify-builds-products-organization-architecture-autonomy-accountability (zitiert auf S. 30).

[Haz] L. Hazlewood. *Design Beautiful REST + JSON APIs*. Stormpath. URL: http://www.slideshare.net/stormpath/rest-jsonapis (zitiert auf S. 75, 76).

[HFK+14] F. Haupt, M. Fischer, D. Karastoyanova, F. Leymann, K. Vukojevic-Haupt. „Service Composition for REST“. In: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference, EDOC 2014*. IEEE, 2014, S. 110–119. DOI: 10.1109/EDOC.2014.24 (zitiert auf S. 37).

[HKLS14] F. Haupt, D. Karastoyanova, F. Leymann, B. Schroth. „A Model-Driven Approach for REST Compliant Services“. In: *Proceedings of the IEEE International Conference on Web Services (ICWS 2014)*. IEEE, 2014, S. 129–136. DOI: 10.1109/ICWS.2014.30 (zitiert auf S. 34, 36).

[HLP15] F. Haupt, F. Leymann, C. Pautasso. „A conversation based approach for modeling REST APIs“. In: *12th Working IEEE / IFIP Conference on Software Architecture - WICSA 2015*. IEEE Computer Society, 2015 (zitiert auf S. 37).

[Jai] S. Jain. *Interview with Ken Schwaber*. URL: https://web.archive.org/web/20120316064715/http://www.agilecollab.com/interview-with-ken-schwaber (zitiert auf S. 26).

[Jau] S. Jauker. *10 Best Practices for Better RESTful API*. M-Way Solutions. URL: http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/ (zitiert auf S. 75, 76).

[Jee] K. S. Jeef Sutherland. *Scrum Guides*. URL: http://scrumguides.org/ (zitiert auf S. 26, 42).

[KL95] E. Kamsties, C. M. Lott. „An empirical evaluation of three defect-detection techniques“. In: *European Software Engineering Conference*. Springer. 1995, S. 362–383 (zitiert auf S. 40).

[Kom12] A. Komus. „Status Quo Agile“. In: *Studie zur Verbreitung und Nutzen agiler Methoden. Hochschule Koblenz* (2012) (zitiert auf S. 24).

[Kum] S. Kumar. *8 Benefits of Microservices | Digital Transformation*. URL: http://blogs.perficient.com/digitaltransformation/2015/06/01/microservices-and-its-benefits/ (zitiert auf S. 30).

[KWB03] A. G. Kleppe, J. B. Warmer, W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003 (zitiert auf S. 17).

[LF] J. Lewis, M. Fowler. *Microservices*. URL: http://martinfowler.com/articles/microservices.html (zitiert auf S. 30).

[MM+01] J. Miller, J. Mukerji et al. „Model driven architecture (mda)“. In: *Object Management Group, Draft Specification ormsc/2001-07-01* (2001) (zitiert auf S. 17).

[MM+03] J. Miller, J. Mukerji et al. *MDA Guide Version 1.0. 1*. 2003 (zitiert auf S. 17).

[Mye78] G. J. Myers. „A controlled experiment in program testing and code walkthroughs/inspections“. In: *Communications of the ACM* 21.9 (1978), S. 760–768 (zitiert auf S. 40).

[New15] S. Newman. *Building Microservices*. O'Reilly Media, Inc., 2015 (zitiert auf S. 30).

[Pro15] T. Procházka. *Model-Driven Development of REST APIs*. 2015 (zitiert auf S. 39).

[RCK09] C. K. Roy, J. R. Cordy, R. Koschke. „Comparison and evaluation of code clone detection techniques and tools: A qualitative approach“. In: *Science of Computer Programming* 74.7 (2009), S. 470–495 (zitiert auf S. 40).

[Roy70] W. W. Royce. „Managing the development of large software systems“. In: *proceedings of IEEE WESCON*. Bd. 26. 8. Los Angeles. 1970, S. 328–338 (zitiert auf S. 24, 25).

[Sah] V. Sahni. *Best Practices for Designing a Pragmatic RESTful API*. Enchant. URL: http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api (zitiert auf S. 75, 76).

[Scr] Scrum Inc. *The Scrum Framework - Scrum Inc*. Scrum Inc. URL: https://www.scruminc.com/scrum-framework/ (zitiert auf S. 28).

[Sel03] B. Selic. „The pragmatics of model-driven development“. In: *IEEE software* 20.5 (2003), S. 19 (zitiert auf S. 17, 18).

[SRD14] M. Schmid, T. Rohloff, P. Duwe. „Musterlösungen und Best Practices für das Design und die Realisierung von REST-Schnittstellen“. In: (2014) (zitiert auf S. 75, 76).

[Ste] G. Steinacker. *Von Monolithen und Microservices - Informatik Aktuell*. URL: https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html (zitiert auf S. 30).

[SVC06] T. Stahl, M. Voelter, K. Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006 (zitiert auf S. 31).

[The] The White House. *WhiteHouse Api Standards*. The White House. URL: https://github.com/WhiteHouse/api-standards (zitiert auf S. 75, 76).

[Tur10] D. W. Turner III. „Qualitative interview design: A practical guide for novice investigators“. In: *The qualitative report* 15.3 (2010), S. 754 (zitiert auf S. 83).

[Wid15] R. Wideberg. *RESTful Services in an Enterprise Environment: A Comparative Case Study of Specification Formats and HATEOAS*. 2015 (zitiert auf S. 39).

[WL06] D. Waddington, P. Lardieri. „Model-Centric Software Development“. In: *COMPUTER-IEEE COMPUTER SOCIETY* 39.2 (2006), S. 2 (zitiert auf S. 32).

[Wol] E. Wolf. *Microservice-Architekturen nicht nur für agile Projekte - Informatik Aktuell*. URL: https://www.informatik-aktuell.de/entwicklung/methoden/microservice-architekturen-nicht-nur-fuer-agile-projekte.html (zitiert auf S. 30).

[ZWN+06] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, M. A. Vouk. „On the value of static analysis for fault detection in software“. In: *IEEE transactions on software engineering* 32.4 (2006), S. 240–253 (zitiert auf S. 29).

Alle URLs wurden zuletzt am 07. 11. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift