Dynamische Abbildung von Java Objekthierarchien auf REST Schnittstellen

Robert Fehrmann
Matrikelnummer: 4297886
robert.fehrmann@fu-berlin.de

Betreuer:
Prof. Dr. Ina Schieferdecker
Dr. Edzard Höfig

Berlin, 26.03.2012

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

26.03.2012

Robert Fehrmann

Inhaltsverzeichnis

1 Einleitung **1**

2 Theoretischer Teil **3**

2.1 Grundlagen Representational State Transfer 3

2.1.1 Bedingungen einer Software im REST-Architekturstil . 3

2.1.2 Stärken von RESTful Web Services 4

2.1.3 REST-Terminologie 5

2.1.4 Java API for RESTful Web Services (JAX-RS) 5

2.2 Identifizierung von Ressourcen mittels URIs 7

2.2.1 Grundlagen Uniform Resource Identifier 7

2.2.2 Komponenten eines URI 7

2.2.3 Hierarchischer Aufbau eines URI 8

2.2.4 Encoding und Reservierte Zeichen 8

2.2.5 HTTP als REST Kommunikationsprotokoll 9

2.2.6 Identifizierung von Ressourcen in Java Hierarchien mit URIs 10

2.2.7 Ressourcen der Domäne 11

2.2.8 Hierarchisierung von Java Objekten 11

2.2.9 Eindeutige Kennung für Ressourcen 12

2.2.10 Java Typen von Ressourcen 14

2.2.11 Besondere URIs im Zusammenhang mit Collections und Arrays 15

2.3 Abbildung von HTTP-Methoden auf Java Operationen 17

2.3.1 Abgebildete Java Operation 18

2.3.2 Übermittlung von Repräsentationen im Nachrichtenkörper 22

2.3.3 Abbildung von Java Operationen auf HTTP Methoden 28

3 Praktischer Teil **31**

3.1 Struktur und Aufgabenteilung des Frameworks 32

3.2 Java Reflection als Mittel zur Introspektion von Objekten . . . 34

3.2.1 Traversierung des Objektgraphens 34

3.2.2 Ermittlung von Java Typen 37
3.3 Selektionsmechanismus zur Auswahl eines Repräsentationsformats 37
3.3.1 Implementierung des Selektionsmechanismus 37

4 Fazit und Ausblick 39

Kapitel 1

Einleitung

Die Autoren B. Lahres und G. Raýman schreiben: „Eine Klasse beschreibt die gemeinsamen Eigenschaften und Operationen einer Menge von gleichartigen Objekten. Die Objekte, die zu einer Klasse gehören, werden als Exemplar dieser Klasse bezeichnet.“ [1] Java ist eine objektorientierte Sprache und ihre Klassen, Objekte und Exemplare verhalten sich wie in der Definition beschrieben zueinander. Ist ein Softwaresystem in Java entwickelt, liegen alle Informationen, unabhängig davon, wie sie aggregiert und dargestellt werden, in den Exemplaren von Klassen vor. Dabei können diese in Relation mit anderen Exemplaren stehen. Dies ist der Fall, wenn ein Exemplar die Eigenschaft eines anderen ist. Durch diese Verbindung können tiefgehende Hierarchien entstehen.

Mit dem Java Core Reflection API, ist es möglich, zur Laufzeit, die Relationen eines Exemplar zu anderen Exemplaren zu ermitteln, die Eigenschaften zu entnehmen oder Operationen aufzuführen. Sind die Exemplare von vorhandenen Relationen ermittelt, können diese wiederum als Ausgangspunkt für weitere Introspektionen genutzt werden. Somit bietet der Reflection API die Möglichkeit der Informationsentnahme aus Exemplaren und der Traversierung durch eine Hierarchie von Exemplaren.

Ziel dieser Arbeit ist der Entwurf und die prototypische Implementierung eines Java Frameworks um beliebige Java Objektgraphen per REST Prinzip im Netz verfügbar zu machen. Dieses System soll den Java Reflection API für die serverseitige Traversierung durch die Java Objekthierarchien einsetzen. Wie Ressourcen, Java Objekte einschließlich ihrer Felder und Methoden adressiert werden, wird durch ein Schema, welches von REST URIs auf Java Objekthierarchien abbildet, spezifiziert. Ebenfalls wird definiert, wie HTTP Methoden auf Java Operationen abgebildet werden. Anschließen wird ein Selektionsmechanismus vorgestellt, der die in Java gehaltenen Daten auf REST-typische Repräsentationsformate abbildet.

Durch die Freigabe der Objekte im Netz, sollen die von den Objekten gehaltenen Daten verfügbar gemacht werden. Ziel ist es nicht, ein Remote Procedure Call (RPC) [2] System, wie Remote Method Invokation (RMI) [3] zu entwickeln.

Die Existenz einer allgemeinen Schnittstelle, die aufbauend auf dem Reflection API, den Zugriff auf Java Objekthierarchien ermöglicht, würde zwei Gruppen von Nutzer einen Vorteil bringen. Zum einen könnten Ressourcen, auf einfachen Weg angeboten werden. Ressourcenanbieter könnten Daten durch Java Objekte, ohne die Definition einer neuen Schnittstelle, bereitstellen. Zum anderen würden Konsumenten von einer Vielzahl von Formaten, in denen die Daten geliefert werden können, profitieren.

Aufbauen auf einer Recherche, wird eine Einführung in den von Roy Fielding entwickelten REST Architekturstil gegeben. Anschließend wird belegt, wieso keine Implementierung des Java API für RESTful Web Services genutzt werden kann. Danach wird eine Abbildung von URIs auf Java Objekte, einschließlich ihrer Felder und Methoden erstellt. Folgend wird erläutert wie die Abbildung von den bei REST eingesetzten Hypertext Transfer Protocol (HTTP) Methoden auf Operationen der Java Programmiersprache, wie Methodenaufrufe und Auslesen oder Schreiben eines Feldes erfolgt. Nachdem alle Grundlagen erläutert wurden, werden Ausschnitte aus der parallel zur Arbeit erstellten Software vorgestellt. Diese Vorstellung beinhaltet eine Erläuterung der Aufgabenstruktur des Frameworks, die Implementierung eines Selektionsmechanismusses zur Auswahl von Repräsentationsformaten und eine Beschreibung, wie Reflection genutzt wurde, um die Traversierung durch den Objektgraphen zu realisieren.

Kapitel 2

Theoretischer Teil

2.1 Grundlagen Representational State Transfer

Aufbauend auf eine Evaluationen existierender Netzarchitekturen und -technologien, entwarf Roy Thomas Fielding in seiner Dissertation den Representational State Transfer (REST). REST stellt keine Architektur dar, sondern eine Sammlung von Anforderungen, welche an eine Software, die im Web agiert, gestellt werden [4].

2.1.1 Bedingungen einer Software im REST-Architekturstil

Als erste Bedingungen an den REST-Architekturstil werden die Bedingungen an eine Client-Server Architektur übernommen. Dieser Architekturstil ist für netzbasierte Systeme erstellt worden. Ein Server bietet Dienste über eine Schnittstelle an. Ein Client kann diese Dienste anfordern. Client und Server können unabhängig voneinander agieren. Der Server kann seine Dienste mehreren Clients anbieten. Für gewöhnlich terminiert ein Server nicht. Ziel ist eine *separation of concerns* (Trennung der Belange) zu erreichen.

Als nächste Bedingung wird die zustandslose Kommunikation zwischen Client und Server gefordert. Dadurch erhöht sich die Skalierbarkeit, weil keine Daten über eine Verbindung auf dem Server gespeichert werden. Nachteilig ist jedoch, dass sich die Last der Netzwerkverbindung erhöht, da ein Request alle Informationen enthalten muss, die ein Server zur Verarbeitung benötigt. Vorteilhaft dabei ist die Transparenz eines Requests. Es genügt einen einzelnen Request zu betrachten, um die Anfrage zu verstehen.

Um der steigenden Netzwerklast entgegen zu wirken, verlangt der REST-Architekturstil, dass Daten eines Response des Servers, als Antwort auf ein

Request des Client, clientseitig zwischengespeichert (gecached) werden. Dazu müssen explizit Angaben gemacht werden, ob Daten zwischengespeichert werden können.

Als wichtigste Bedingung an einen REST-Architekturstil nennt Roy Thomas Fielding ein Uniform Interface. Ein REST-Interface muss designed sein, um effizient hohen Hypermedia-Datentransfer zu verarbeiten. Dabei muss das Interface auf die Gegebenheiten des Webs eingehen. Vier Bedingungen werden an ein Uniform Interface gestellt: Ressourcenidentifikation, Manipulation von Ressourcen durch Repräsentationen, selbstbeschreibende Nachrichten und „hypermedia as the engine of application state“.

Die letzte Bedingung ist, dass Schichten eingeführt werden. Dabei bietet eine Schicht Dienste der nächsthöheren Schicht an und nutzt die Dienste der Schicht nächsttieferen Schicht. Durch diese Hierarchie wird die Kopplung zwischen Komponenten reduziert und die sichtbare Komplexität für Komponenten minimiert. Nachteilig sind der Mehraufwand und die Latenz der Datenverarbeitung zu nennen, die jedoch durch geteilte Caches minimiert werden können.

2.1.2 Stärken von RESTful Web Services

Durch die Bedingungen, die ein Softwaresystem im REST Architekturstil erfüllen muss, ergeben sich für ein solches System laut Fielding folgenden Vorteile [5]::

- Es wird mit den HTTP Methoden eine generisches Schnittstelle generiert, mit der auf Ressourcen gearbeitet werden kann.
- RESTful Web Services skalieren sehr gut. GET Anfragen auf Ressourcen können gecached werden.
- REST baut auf Web Standards, wie das Hypertext Transfer Protokoll (HTTP) und Uniform Resource Identifier (URI) auf.
- Die Kommunikation erfolgt zustandslos. Alle nötigen Daten befinden sich in einer Anfrage des Clients.

Im Gegensatz dazu, stehen SOAP Web Services (Simple Object Access Protocol). Mit dieser, in der Industrie sehr angenommenen Technologie werden Daten zwischen Softwaresystemen durch Remote Procedure Calls (RPC) ausgetauscht. Welche Methoden der Web Services anbietet, wird mit der Web Services Description Language (WSDL) beschrieben. Dadurch entstehen statische Interfacestrukturen, nach denen sich ein Nutzer richten muss.

2.1.3 REST-Terminologie

Ressoucen

Ressourcen sind eine Abstraktion aller Informationen, die man benennen kann. Dabei stellt eine Ressource, die Abbildung auf einen Menge von Entitäten dar. Es ist ebenfalls möglich, dass eine Ressource auf eine leere Entitäten-Menge verweist. Dies ist eine wichtige Eigenschaft, da so bereits auf das Konzept einer Ressource verwiesen werden kann, bevor diese existiert. Durch die allgemeine Definition einer Ressource ist es möglich für eine Ressource viele verschiedene Informationsquellen, -formate zu nutzen ohne diese zu unterscheiden.

Resourcen Identifier

Resource Indentifier sind für die Kommunikation zwischen Komponenten wichtig. Sie ermöglichen eine Identifizierung einer Ressource. Die Manipulation von Daten erfolgt über Interfaces, die mittels der Resource Identifier genutzt werden können.

Repräsentation von Ressourcen

Repräsentationen stellen ein Kommunikationsmittel zwischen Komponenten dar. In Repräsentationen wird der aktuelle oder der gewünschte Zustand einer Ressource übermittelt. Eine Repräsentation besteht aus Daten und Metadaten. Das Datenformat einer Repräsentation wird durch den Media Type vorgegeben.

Kontrolldaten

Kontrolldaten sind Informationen, die dem Empfänger einer Nachricht mitteilen, welche Aktionen ausgeführt werden sollen. Durch die Kontrolldaten kann ersichtlich werden, ob die Repräsentation den derzeitigen oder den zukünftigen Status einer Ressource darstellt.

2.1.4 Java API for RESTful Web Services (JAX-RS)

Mit der Dissertation von Roy Thomas Fielding entstand auch der Bedarf nach einer Spezifikation für RESTful Web Services in Java. Der Java Community Process stellte die Spezifikationsanfrage 311 mit dem Name Java API for RESTful Web Services (JAX-RS) [6] ein. Die Referenzimplementierung Jersey [7] liegt in der Version 1.9 vor. Diese implementiert JAX-RS 1.1 ab der Version 1.12. Das Projekt wird unter der CDDL 1.1 and GPL

2 with CPE Lizenz [8] angeboten. Neben der vollen Implementierung der JAX-RS Spezifikation, legt Jersey einen Schwerpunkt auf die Bereitstellung eines APIs, um Jersey REST Anwendungen zu erweitern. RESTEasy [9] ist ein weiterer Vertreter für Java REST Frameworks. Es ist ein JBoss Projekt, welches sowohl für die Erstellung von Services, als auch für die Erstellung von Client Anwendungen genutzt werden kann. Das Framework läuft in einem *Servlet-Container* [10], der beispielsweise mit jedem *JBoss Application Server* [11] ausgeliefert wird. Auch ist die Einrichtung eines Containers unabhängig vom JBoss Server, durch die Nutzung von eines *Tomcat* [10] Servers, möglich. Die volle Implementierung von JAX-RS ist zertifiziert. Zu den Stärken des Projekts gehört die Vielzahl unterstützten Formate, wie zum Beispiel: XML, JSON, Atom. Desweiteren wird das Zwischenspeichern (*cachen*) von von Serverantworten unterstützt. Dafür wird zum einen auf eine automatische Erstellung von *ETAGs* [12] und Revalidierung des Zwischenspeichers zurückgegriffen. Als weiteren Vorteil bietet RESTEasy die Unterstüzung von *GZIP*-Komprimierung [13] von Inhalten in Anfragen und Antworten. Schlussendlich macht die guten Einbindungsmöglichkeit anderer Frameworks (bspw. Spring oder EJB), sowie die enge Verbindung zum *JBoss Application Server* das REST Framework, RESTEasy zu einem Favoriten in der Industrie. Neben diesen beiden Frameworks, die ihre Stärken im Umfang des Angebots haben, existiert RESTlet [14]. Dieses Framework besitzt einen kleinen Kern an Funktionalität, der durch *Plugins* erweitert werden kann. Nutzbar ist RESTlet sowohl für Client, als auch Server Anwendungen. Alle REST Konzepte, wie Ressourcen, Rerpäsentationen etc., sowie die gängigsten Formate werden unterstützt. Auch dieses Framework bietet die Möglichkeit der Einbindung in andere Frameworks. Für die Entwicklung des System zur Freigabe von Java Objekthierarchien eignet sich die Spezifikation und die daraus folgenden Implementierungen nicht. In der Spezifikation wird das Java Sprachfeature Annotation für Implementierungen vorgeschlagen, um dem Entwickler einen einfache Programmierweise zu ermöglichen [6]. Diesem Vorschlag folgen alle Implementierungen. Jede Ressource, die angeboten werden soll, muss mit Annotationen bzgl. URI und Zugriffsmethode versehen werden. Für das System zur Freigabe von Objekthierarchien ist dies nicht praktikabel. Die Schnittstelle wird dynamisch generiert und der URI wird erst auf Anfrage hin ausgewertet. Dies stellt einen neuen Ansatz dar, der eine eigene Implementierung eines RESTful Services nötig macht.

2.2 Identifizierung von Ressourcen mittels URIs

2.2.1 Grundlagen Uniform Resource Identifier

Der Uniform Resource Identifier (URI) [15] wurde durch den RFC 3986 spezifiziert. Er ist eine Zeichensequenz, die eine physische oder abstrakte Ressource identifiziert. URIs können durch die drei namensgebenden Begriffe charakterisiert werden: Uniformität (Uniform), Ressource und Kennung (Identifier). Durch Uniformität ist es möglich immer wieder neue Kennungen zur Identifizierung von Ressourcen einzuführen. Dies ist auch möglich, wenn Ressourcen innerhalb eines Kontextes bereits identifiziert werden. Außerdem ist es durch Uniformität möglich, Aussagen über Semantik einer Kennung zu treffen, da die Syntax bei jeder Kennung eingehalten werden muss. Wiederverwendung von Kennung in verschiedenen Kontexten und die Einführung von neuen Kennungen, ohne alte zu stören sind weitere Vorteile der Uniformität. Ein URI ist nicht darauf begrenzt immer die selbe Art von Dingen zu identifizieren. Aus diesem Grund wird der Begriff Ressource genutzt. Eine Ressource kann alles sein, das irgendwie durch einen URI identifiziert werden kann. Die Kennung sind alle Informationen, die nötig sind, um eine Ressource, die es zu finden gilt, von allen anderen zu unterscheiden.

2.2.2 Komponenten eines URI

Für eine ausführlichere Beschreibung der URI-Komponenten, sei auf den RFC 3986 [15] verwiesen. Hier wird nun eine Einführung in den hierarchischen Path und die Query in Bezug auf die Aufgabe in der Arbeit gegeben.

scheme://host:port/path?query#fragment

Abbildung 2.1: Allgemeine Struktur eines URI [16]

- Path (Pfad)
  Der Pfad enthält Daten zur Identifikation von Ressourcen. Liegen diese in hierarchischer Form vor, so bilden in der Hierarchie weiter oben stehende Daten, Segmente eines Pfades, die weiter vorn stehen. Einzelne Segmente eines Pfades werden durch „/“ getrennt. Auch innerhalb dieser Arbeit wird diese Form genutzt, um einen Teil der Informationen zur Identifizierung bereitzustellen. Ressourcen können innerhalb des Pfades ein Segment bilden, wobei eine konkrete Ausprägung einer

Ressource das nächste Segment bildet. Der Pfad wird schließlich durch das erste „?“ terminiert und die Query, falls vorhanden, beginnt [15].

- Query
  Die Query dient zur Angabe von Informationen in nicht-hierarchischer Form. Dabei liegen die Informationen meist in der Form „key=value“ vor. Die Query beginnt mit einem „?“ und endet mit dem ersten „#“ [15].

2.2.3 Hierarchischer Aufbau eines URI

Ein URI ist hierarchisch aufgebaut. Je weiter links ein Segment steht, desto spezieller ist es. Zur Trennung von Komponenten existieren reservierte Zeichen. Durch die uniforme Darstellung von Hierarchien, ist es möglich, URIs schemenunabhängig einzusetzen [15].

2.2.4 Encoding und Reservierte Zeichen

Ein URI besteht aus einer Zeichenfolge, die sich aus einer begrenzten Mengen von Zahlen, Buchstaben und einigen graphischen Zeichen zusammensetzt. Um die oben beschriebene Hierarchisierung zu ermöglichen existieren Zeichen mit Begrenzungsfunktion. Diese Zeichen sind *reservierte Zeichen*, da sie durch die generelle URI Syntax, die Syntex eines Schemas oder der Syntex einer Implementierung zur Dereferenzierung von URIs genutzt werden. Treten Daten von URI Komponenten in Konflikt mit diesen Zeichen, so müssen sie kodiert werden. Hierfür wird „Percent-Encoding“ (PE) genutzt. Dabei werden die betreffenden Zeichen, die normalerweise als Oktett von Bits übertragen werden, zu einen Triplett aus drei Zeichen übertragen, wobei das erste Zeichen ein Prozent ist und die letzten beiden, das Zeichen in Hexerdezimalform sind. Die nachfolgende Tabelle zeigt das PE für alle reservierten Zeichen [15].

Typ	!	#	$	&	’	(	)	*	+
PE	%21	%23	%24	%26	%27	%28	%29	%2A	%2B
Typ	,	/	:	;	=	?	@	[	]
PE	%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	%5D

Tabelle 2.1: Percent-Encoding reservierter Zeichen

2.2.5 HTTP als REST Kommunikationsprotokoll

Das Hypertext Transfer Protocol (HTTP) [12] ist ein Protokoll für verteilte, gemeinschaftlich genutzte Hypermedia Systeme. Es befindet sich in der Anwendungsschicht des ISO/OSI Modells. HTTP ist das zentrale Kommunikationsprotokoll im World Wide Web [17]. Es ermöglicht das Senden von Anfragen an einen Server und das Empfangen von Antworten von einem Server. Das Format der Daten, die in einer Nachricht übermitteln werden, wird durch den *MIME type* festgelegt [18]. Der *MIME type* wird von Client und Server ausgehandelt. Durch diese Eigenschaft, ist HTTP nicht auf das Senden und Empfangen von *hypertext* beschränkt, sondern kann vielfältige Aufgaben übernehmen [12]. Die Kommunikation zwischen Client und Server ist zustandslos. Jede Nachricht enthält somit alle Informationen, um vom Server oder Client verstanden und bearbeitet zu werden.

HTTP Methoden

Eigenschaften von HTTP Methoden

- Sicherheit
  Eine wichtige Eigenschaft einer HTTP-Methode ist die *safe*-Eigenschaft. Durch diese Eigenschaft werden die HTTP-Methoden in sichere und unsichere Methoden geteilt. Sicher ist eine Methode, wenn sie außer dem Erhalten der Ressource keine weiteren Effekte zur Folge hat. Dies schließt die Methoden GET und HEAD ein [12].

- Idempotenz
  Diese Eigenschaft besagt, dass eine wiederholte Ausführung des gleichen Request immer das gleiche Ergebnis liefert. Die Methoden GET, HEAD, PUT, DELETE, OPTIONS und TRACE sollten idempotent implementiert werden [12].

GET Mit der GET Methode ist es möglich eine Ressource zu erhalten, die durch den URI der Anfrage identifiziert wurde. Vom Server wird die Ressource immer in Form einer Entität gesendet. Durch das Einfügen von Bedingungen, ändert sich die GET Methode zu einer *bedingten GET* Methode. Damit geht eine Semantikänderung einher. Bedingungen beziehen sich beispielsweise auf das letzte Änderungsdatum oder auf die Frage ob von den angebotenen Ressourcen eine zur Anfrage passt. [12]

POST Die POST Anfrage wird an den Server mit dem Ziel der Datenübermittlung gesendet. Diese befinden sich innerhalb des *Entity Bodies* der

Anfrage. Damit der Server die Entität dem Body im richtigen Format entnehmen kann, muss im Header der Anfrage der *Content Type* gesetzt werden. Eine POST Anfrage muss nicht, kann aber in einer neuen Ressource resultieren. Diese wird dem *Entity Body* entnommen. Wird eine neue Ressource angelegt, sollte dies mit 201 Created dem Anfragenden in der Antwort bestätigt werden. Außerdem sollte der *Location Header* mit dem URI gesetzt werden, unter dem die neue Ressource adressierbar ist.

PUT Die PUT Methode ähnelt der POST Methode. Sie wird zum Speichern einer Ressource, die sich im *Entity Body* der Anfrage befindet, genutzt. Während bei der POST Methode der URI genutzt wird, um eine Ressource identifiziert, welche durch die Entität im Body ersetzt wird, ist bei der PUT Methode der URI der Speicherort einer neuen Ressource. Diese wird wie bei POST dem *Entity Body* entnommen. Außerdem sind die Reaktionen des Servers strikter definiert. So **muss** der Server beim Erstellen einer Ressource, dem Anfragenden 201 zurückmelden. Auch dürfen keine *Content Header* ignoriert werden. Unterstützt der Server einen *Content Header* nicht, so muss dieser 502 (Not Implemented) als Status melden. [12]

DELETE Durch ein DELETE wird der Server aufgefordert eine Ressource an angegebener URI zu löschen. Es ist darauf zu achten, dass sichergestellt werden kann, ob die Aktion erfolgreich ausgeführt wurde. Erst die Antwort des Servers sollte eine Erfolgsmeldung, wie 200 (OK) enthalten. [12]

HEAD Die HEAD Methode liefert alle Metadaten, die auch im Header der Antwort auf eine GET Anfrage stehen würden. Dabei wird vom Server kein *Entity Body* der Antwort angehängt. [12]

OPTIONS Mit Hilfe einer OPTIONS Anfrage kann ein Client in Erfahrung bringen, welche Methoden der Server für eine Ressource anbietet. [12]

2.2.6 Identifizierung von Ressourcen in Java Hierarchien mit URIs

Als ersten Schritt, für die Umsetzung eines RESTful Webservices gilt es, alle Ressourcen zu identifizieren. Allgemein ist es hierfür notwendig, die Anwendungsdomäne zu untersuchen. In dieser Arbeit, bildet die Programmiersprache Java die Domäne. Konkreter, sind es die Objekte, einschließlich ihrer Felder und Methoden die für die Umsetzung des Frameworks relevant sind.

Im folgenden wird die Domäne analysiert und die Ressourcentypen extrahieren.

2.2.7 Ressourcen der Domäne

Für die Identifizierung von Ressourcen gibt es das allgemeine Vorgehen, die Anwendungsfälle der Software auf Entitäten hin zu untersuchen, für die es vorstellbar ist, sie „anzulegen“, „lesen“, „aktualisieren“ oder „löschen“ zu können [19]. Begründet ist dies in den vier Hauptmethoden POST, GET, PUT und DELETE des HTTP Protokolls, die von RESTful Web Services genutzt werden und auf die später ausgiebig eingegangen wird. In Java ist es möglich, Objekte zu erstellen und zu vernichten, deren Werte abzufragen oder zu aktualisieren. Diese vier Eigenschaften lassen sich auf die HTTP-Methoden POST, DELETE, GET und PUT abbilden. Somit bildet ein Objekt die erste Ressource. Mit dieser Ressource kann auf weitere geschlossen werden. Objekte besitzen Felder und Methoden. In der Domäne sind dies ebenfalls Objekte, jedoch macht es Sinn, ihnen eigene Ressourcentypen zuzuordnen. Felder können zu einer Feld-Ressource abstrahiert werden. Feld-Ressourcen können mit den gleichen HTTP-Methoden, wie Objekt-Ressourcen angefragt werden. Der Unterschied zwischen den beiden Ressourcen liegt in der Abhängigkeit voneinander. Feld-Ressourcen sind abhängig von Objekt-Ressourcen. Andersherum ist dies nicht der Fall. Die Abhängigkeit bezieht sich auf die Existenz des Java-Objektes. Für eine Objekt-Ressource gilt immer, dass sie alleinstehend existieren kann. Der Ressource liegt ein Objekt zugrunde, welches per URI angefragt werden kann. Dies gilt für eine Feld-Ressource nicht. Einer Feld-Ressource liegt zwar ebenfalls ein Objekt zugrunde, jedoch ist der Lebenszyklus, durch die Existenz eines anderen Objektes begrenzt. Wird dieses übergeordnete Objekt entfernt, kann die Feld-Ressource nicht weiter existieren. Die geschilderte Abhängigkeit besteht ebenfalls für Java-Methoden. Sie können nicht ohne einen Kontext existieren. Der Kontext ist hier ebenfalls die Existenz eines Java-Objekts, auf dem die Methode aufgerufen wird.

2.2.8 Hierarchisierung von Java Objekten

Ein Java Objekt einer Klasse, setzt sich aus seinen Eigenschaften (Feldern) und seinen Operationen (Methoden) zusammen. Hierdurch entsteht eine Hierarchie zwischen dem Objekt und seinen Feldern und Methoden. Wenn die Felder eines Objektes ebenfalls Felder und Methoden besitzen, setzt sich dort die Hierarchie fort.

- Klassifizierung von Hierarchiestufen
  - Hierarchiestufe 0:
    Innerhalb dieser Hierarchiestufe sind die Objekte von keinem anderen Objekt abhängig. Nur Objekt-Ressourcen stehen in der Hierarchie auf Stufe 0.
  - Hierarchiestufe n:
    Um ein Objekt in der Hierarchiestufe n zu erreichen, müssen n Objekte „durchlaufen“ werden. Dabei ist das n-te Objekt das Feld oder die Methode des n-1-ten Objektes. Nur Feld-Ressourcen oder Methoden-Ressourcen stehen in der Hierarchie auf Stufe n.

Abbildung 2.2 zeigt ein Beispiel, wie eine solche Hierarchie zwischen Java Objekten aussehen kann.

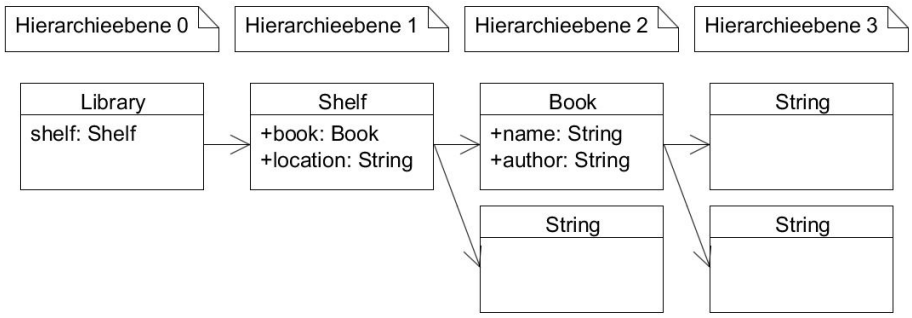

Abbildung 2.2: Darstellung der Hierarchie zwischen Objekten und ihren Feldern in einem UML Klassendiagramm

Das Ausgangsobjekt ist die Bibliothek. Sie ist in der Beispielhierarchie auf der Stufe null vertreten. Die Bibliothek besitzt ein Feld: Regal. Um das Regal auf Hierarchiestufe eins zu erreichen, muss zuvor das Bibliotheksobjekt durchlaufen werden. Diese Vorgehensweise setzt sich weiter fort. Um den Namen des Buches zu erhalten, müssen zunächst Bibliothek, Bücherregal und Buch durchlaufen werden.

2.2.9 Eindeutige Kennung für Ressourcen

Jede Ressource muss eine Kennung besitzen, die sie eindeutig von anderen Ressourcen der gleichen Hierarchiestufe unterscheidet. Dazu werden folgende Regelungen getroffen:

- Feld-Ressource
  Als Kennung für eine Feld-Ressource wird der Name der Feldes genutzt. In der Programmiersprache Java muss dieser innerhalb des Sichtbarkeitsbereiches eindeutig sein.

- Methoden-Ressource
  Für die eindeutige Identifizierung einer Methode, reicht der Name der Methode nicht aus. Daher wird die komplette Signatur, bestehend aus dem Namen der Methode und den Typen der Parameter, zur Identifikation genommen. Innerhalb des Sichtbarkeitsbereiches muss diese ebenfalls eindeutig sein.

- Objekt-Ressource
  Die Objekt-Ressource bildet die Ausnahme bei der Identifizierung von Ressourcen. Sie wird über eine vorher definierte Zeichenfolge identifiziert. Diese bildet genau auf ein Exemplar ab.

URI für Ressourcen

Die Hierarchie zwischen Java Objekten und ihren Felder und Methoden lässt sich mit Hilfe der URI Syntax darstellen. Zur Trennung der Hierarchiestufen wird ein „/“ eingesetzt. Die Namensgebung der Ressourcen wird wie in 2.2.9 beschrieben. Durch die getroffenen Bedingungen an Syntax und Namensgebung lassen sich für alle Ressourcen URIs bilden. Die Spezifikation in Abbildung 2.3 enthält den Aufbau eines Pfades einer URI. Für Spezifikation wird die Augmented Backus-Naur Form (ABNF) [20] genutzt. Ihr ist zu entneh-

```
path          = or-id "/" n-hierarchy
n-hierarchy   = empty  | fr-id "/" n-hierarchy | mr-id "/" n-hierarchy
or-id         = *pchar ; valid identifier for object resource
fr-id         = *pchar ; valid identifier for field resource
mr-id         = *pchar ; valid identifier for method resource
pchar         = unreserved / pct-encoded ; only unreserved or percent encoded chars
empty         = 0<pchar>
```

Abbildung 2.3: Allgemeiner Aufbau des Pfades zur Ressourcenidentifizierung

men, dass jeder gültige Pfad mit einem Identifier für eine Objekt-Ressource (or-id) beginnt. Getrennt durch einen Slash, können tiefere Hierarchiestufen (n-hierarchy) erreicht werden. Dies sind die Felder-Ressourcen (fr) oder Methoden-Ressourcen (mr), die durch ihre Identifier (id) identifiziert werden. Dabei ist der Identifier eine Zeichenkette (*pchar) von nicht-reservierten

(unreserved) oder prozent-kodierten (pct-encoded) Zeichen (chars). Kommt nach einem Slash kein weiterer Identifier, so ist der Pfad bis einschließlich des letzten gültigen Identifier, der Pfad, der eine Ressource identifiziert.

In Abbildung 2.4 wird anhand des Bibliotheksbeispiels (Abb. 2.2), die Bildung des Pfades des URIs einer Feld-Ressource gezeigt. Um den Namen des Buches zu bekommen, wird die Hierarchie Schritt für Schritt (step) durchlaufen.

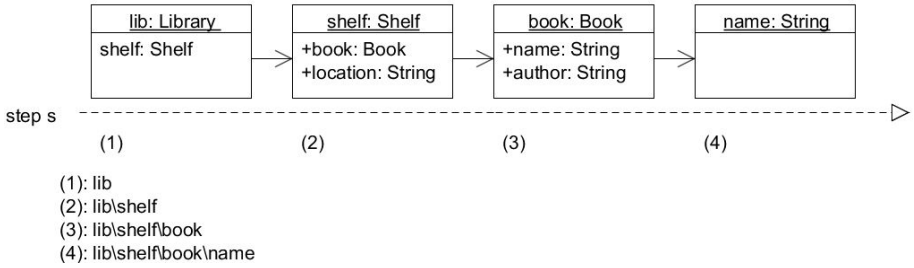

Abbildung 2.4: Aufbau eines Pfades eines URIs für die Identifizierung einer Feld-Ressource

2.2.10 Java Typen von Ressourcen

Jeder URI, der zu einer Ressource führt, bildet auf ein Java Objekt ab. Innerhalb dieser Arbeit werden die Objekte in zwei Kategorien unterteilt. Die Unterteilung erfolgt nach den Typen der Objekte. Zum einen in die *atomaren* und zum anderen in *nicht-atomare* Typen. Die Unterteilung ist in der Darstellbarkeit der Objekte begründet. Für Typen, wie Strings, Integers, Float etc. bilden die Werte gleichzeitig eine Repräsentation. Daher ist es möglich, diese Typen beispielsweise aus Teilen eines URI zu extrahieren. Für komplexe Objekte, vor allem solche, die nicht Teil des Java Software Development Kit [21] sind ist dies nicht ohne weiteres möglich.

Klasse	String	Integer	Float	Boolean
Beispielrepräsentation	'Carl'	100	1.1f	true

Tabelle 2.2: Auflistung atomarer Typen mit Beispielrepräsentanten

2.2.11 Besondere URIs im Zusammenhang mit Collections und Arrays

Das Hauptziel eines URIs ist die eindeutige Lokalisierung einer Ressource. Jedoch gibt es auch sekundäre Ziele, die nicht vernachlässigt werden sollten. Vor allem auf die Lesbarkeit eines URIs ist dabei Wert zu legen. Im Zusammenhang mit Java Collections und Arrays ergeben sich daher Möglichkeiten, die den URI vereinfachen.

- Collections
  Eine Collection ist ein Java Objekt, dass eine Gruppe von Objekten repräsentiert. Mit dem Java Collection Framework wird es ermöglicht, Collections zu repräsentieren und zu manipulieren, unabhängig von den Details der Repräsentation. Für jeden der insgesamt 14 verschiedenen Collection Typen wurde ein Interface definiert, welches das Collection Interface erweitern.

  Da Collections elementare Datentypen in Java sind, bietet es sich an, den URI, für den Zugriff auf deren Elemente zu vereinfachen. Im folgenden werden für drei häufig genutzte Collectionarten, die Gestaltungsmöglichkeit eines URIs erweitert. Dabei wird zunächst anhand des List-Collectiontype's ausführlich gezeigt, wie ein URI ohne und danach mit Erweiterung, aussehen kann. Danach werden weitere Beispiele in tabellarisch aufgelistet. Hinweise, auf welche Elementtypen sich die Erweiterungen beziehen, werden jeweils mit angegeben.

  - List
    Eine Liste ist eine Collection, deren Elemente einer Ordnung unterliegen. Für Elemente wird ein Index geführt.

    * Element an Index i erhalten (Abb. 2.5).
      Schnittstellenmethode: get(int index)

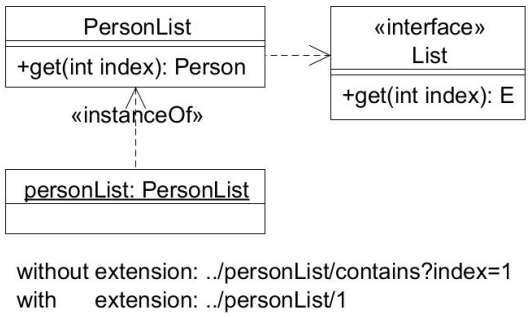

Abbildung 2.5: Vereinfachtes Erhalten eines Elements an Index 1

– Map
  Eine Map ist eine Collection, die *key-value*-Paare aufnimmt. Schlüssel dürfen nicht mehrfach vorkommen. Tbl. 2.3

Methode	get(K key)
'Objektname'	map
Elementtype	K:String
URI	../map/get?key='Carl'
HTTP-Methode	GET
Entity	leer

Methode	put(K key, V value)
'Objektname'	map
Elementtype	K:String, V:String
URI	../map/put
HTTP-Methode	PUT
Entity	Repräsentation für Wert

Methode	put(K key, V value)
'Objektname'	map
Elementtype	K:String, V:String
URI	../map/put
HTTP-Methode	POST
Entity	Repräsentation für Wert

Tabelle 2.3: URI Vereinfachungsmöglichkeiten für Ressourcen vom Typ Map

– Set
  Ein Set ist eine Collection, die jedes Element nur einmal enthält.

Methode	contains(E element)
'Objektname'	set
Elementtype	E:String
URI	../set/contains?element='Carl'
vereinfachter URI	../set/Carl
HTTP-Methode	GET
Entity	leer

Tabelle 2.4: URI Vereinfachungsmöglichkeiten für Ressourcen vom Typ Set

- Arrays
  In Java sind Arrays Objekte, die dynamisch erzeugt werden und denen Variablen vom Typ Objekt zugewiesen werden können. Die Werte des Arrays können über einen Index abgerufen werden. Das Arrayobjekt selbst, besitzt keine Methoden für das Setzen oder Auslesen von Werten, an einem bestimmten Index. Dafür wird die Klasse Array aus dem java.lang.reflect Paket genommen. Egal ob mit oder ohne Vereinfachung, bleibt dem Nutzer verborgen, dass ein Arrayobjekt keine *get*- oder *set*-Methode besitzt.

Methode	get(int index)	set(Object o, int index)
'Objektname'	array	array
Elementtype	String	String
URI	../array/get?index=1	../array/get?index=1& value='Carl'
vereinfachter URI	../array/1	../array/1
HTTP-Methode	GET	POST
Entity	leer	Repräsentation für Wert

Tabelle 2.5: URI Vereinfachungsmöglichkeiten für Ressourcen vom Typ Array

2.3 Abbildung von HTTP-Methoden auf Java Operationen

In diesem Kapitel wird gezeigt, wie ein URI genutzt werden kann, um mit HTTP-Methoden Ressourcen anzufragen. Dabei wird geklärt, inwieweit die Methoden zu den Ressourcen passen und welche Informationen durch eine Anfrage bereitgestellt werden.

2.3.1 Abgebildete Java Operation

Dieser Abschnitt enthält eine Beschreibung relevanter Operationen der Java Programmiersprache.

- Felder auslesen und setzen
  Jedes Java Objekt kann Felder besitzen. Diese können ebenfalls Objekte sein, die referenziert werden können. In Java erfolgt der Zugriff auf ein Feld mit Hilfe des Punktoperators. Dieser hat die Form *class.field*. Die Identifizierung des Feldes erfolgt durch dessen Bezeichner. Ein Aufruf der Form *class.field*, liefert die Referenz zum Objekt des Feldes. Diese Referenz kann manipuliert werden und auf andere Objekte verweisen. Dies geschieht durch Zuweisung einer Referenz zum Feld in der Form: *class.field = object*. Für Felder, die vom Typ Array sind, ergibt sich die Möglichkeit, die Referenz auf ihre Elemente, durch einen Index zu erhalten. Der Zugriff folgt der Form: *array[index]*. Folgend ist eine Auflistung mit allen Möglichkeiten zur Manipulation eines Feldes zu finden. Die exakte Abbildung findet sich in 2.3.3.
  - Auslesen eines Feldes mit GET
  - Setzen eines Feldes mit POST
  - Löschen einer Objektreferenz mit DELETE
  - Erstellen von Objekten mit PUT
- Methodenaufrufe
  Der Methodenaufruf erfolgt ebenfalls mit Hilfe des Punktoperators, in der Form *class.method*. Für die Ausführung einer Methode, werden zusätzlich zum Bezeichner der Methode, die Parameter, wie sie in der Signatur der Methode definiert werden, benötigt. Dies beinhaltet die richtige Anzahl und den richtigen Typ für jeden Parameter. Terminiert eine Methode erfolgreich, bietet sie einen Rückgabewert an. Dieser ist entweder *void* oder die Referenz auf ein Objekt. Innerhalb dieser Arbeit erfolgt eine Unterteilung der Methoden in zwei Klassen. Die erste Klasse beinhaltet alle Methoden, die nur das *Abfragen* eines Feldes zur Aufgabe haben. Diese werden als *getter-Methoden* bezeichnet. Alle anderen Methoden fallen in die zweite Klasse von Methoden. Diese können Seiteneffekte (Veränderungen an der Datenbasis) hervorrufen. Die folgende Auflistung der Methodenaufrufe, wird durch das in 2.3.3 erstellte Schema spezifiziert.
  - Methodenaufruf mit GET
  - Methodenaufruf mit POST

Nachrichten als Mittel des Ressourcenzugriffs

Es existieren zwei Arten von HTTP Nachrichten: Anfragen (*Request* eines Clients) und Antworten (*Response* des Servers). Beide Nachrichtentypen folgen dem Schema aus Abb. 2.6, welches aus der HTTP Spezifikation entnommen ist [12].

```
generic-message =      start-line
                       *(message-header CRLF)
                       CRLF
                       [ message-body ]
start-line = Request-Line | Status-Line
```

Abbildung 2.6: Schema einer HTTP Nachricht

Somit besteht eine Nachricht mindestens aus der *Request-Line* bzw. der *Status-Line*, den *message-header* und einer Leerzeile. Die *message-header* sind Felder, welche die Metadaten der Nachricht enthalten. *Message-header* werden in vier Kategorien: *general, request, response, entity*, eingeteilt. Der Leerzeile folgend, kann optional ein Nachrichtenkörper hinzugefügt werden [12].

HTTP Request Wie in Abb. 2.6 gezeigt, ist die *Request-Line* (Abb. 2.7) ein Bestandteil einer Clientanfrage. Sie enthält die genutzte HTTP Methode, einen URI, der eine Ressource identifiziert und die Version des HTTP Protokolls. Die Bestandteile werden voneinander mit Leerzeichen (*carriage return* - CR) getrennt. Abgeschlossen wird die *Request-Line* mit einem CR und *line feed* (LF).

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Abbildung 2.7: Schema der HTTP *Request-Line*

Die *header* einer Anfrage bestehen aus den allgemeinen *header*, den *header*, die sich die Anfrage beziehen und den *header*, die sich auf den Nachrichtenkörper beziehen. Das Schema einer HTTP Anfrage sieht somit folgendermaßen aus:

Request Header

- Accept
  Mit diesem Header Feld, kann der Client innerhalb einer Anfrage an den

```
Request =       Request-Line
                *(( general-header
                | request-header
                | entity-header ) CRLF)
                CRLF
                [ message-body ]
```

Abbildung 2.8: Schema eines HTTP *Requests*

Server eine Menge von *Media Types* [18] bestimmen, die er annehmen kann [12].

- Accept-Charset
  Dieser Header kann genutzt werden um Angaben zu machen, welchen Zeichensatz der Client vom Server akzeptiert [12].

- Accept-Encoding
  In diesem Header Feld bestimmt der Client eine Kodierung der Antwort des Servers. Unterstützt der Server diese Kodierung nicht, so sendet dieser den *error code* 406 (Not Acceptable) [12].

HTTP Response Als Reaktion auf eine HTTP Anfrage, sendet der Server eine Antwortnachricht. Der Aufbau der Nachricht folgt einem, zum *request*-Schema ähnlichen Schema. Anstelle der *request-line* steht die *status-line* und anstatt von *request-header* enthält das Schema *response-header* (Abb 2.9). Aus der *status-line* kann der Empfänger der Nachricht, die HTTP Version,

```
Response =      Status-Line
                *(( general-header
                | response-header
                | entity-header ) CRLF)
                CRLF
                [ message-body ]
```

Abbildung 2.9: Schema eines HTTP *Response*

den Status Code und die textuelle Beschreibung des Status entnehmen.

Response Status Codes Jede Antwort eines Server besitzt eine *Status Line*. Diese enthält den *Status Code* für die Anfrage. Ein *Status Code* ist eine dreistellige Dezimalzahl. *Status Codes* werden in fünf Kategorien unterteilt. Die Zugehörigkeit eines *Status Codes* zu einer Kategorie, wird durch die erste

Ziffer festgelegt. Es ist nicht zwingend notwendig, dass ein Server alle *Stauts Codes* versteht, aber er muss er mit jede Kategorie umgehen können.

- *Informational* - 1xx

- *Successful* - 2xx
  Status Codes dieser Klasse zeigen an, dass die Anfrage erfolgreich entgegengenommen, verstanden und akzeptiert wurde.

- *Redirection* - 3xx
  Die Kategorie beinhaltet alle Status Codes, die weitere Interaktionen des Client's benötigen, damit die Anfrage abgeschlossen werden kann.

- *Client Error* - 4xx
  In dieser Kategorie werden Status Codes verwaltet, die gesendet werden, falls Syntaxprobleme bei der Anfrage auftreten.

- *Server Error* - 5xx
  Status Codes, die gesendet werden, wenn der Server bemerkt, dass die Anfrage nicht korrekt bearbeitet werden kann.

HTTP Entity Sowohl Anfragen, als auch Antworten können im Nachrichtenkörper Nutzlast transportieren. Diese befindet sich innerhalb des *entity-body*. Dieser kann ebenfalls durch *header* genauer beschrieben werden. Client und Server können für die *entity*, die gleichen *header* setzen.

Entity Header

- Content-Encoding
  Durch dieses Feld kann angegeben werden, dass der *entity body* durch zusätzliche Kodierung modifiziert wurde. Da es möglich ist mehrere Kodierungen anzugeben, ist bei der Dekodierung auf die Reihenfolge zu achten. Die Reihenfolge wird durch die Reihenfolge der Anwendung der Kodierungsverfahren bestimmt. Kann der Server den *entity body* nicht dekodieren, so soll dieser 415 (Unsupported Media Type) dem Client melden.

- Content-Length
  Dieses Feld kann genutzt werden um die Länge Größe der Daten im *entity body* anzugeben. Dabei sind nur Werte größer oder gleich null zulässig.

- Content-Location
  Das Content-Location Header Feld kann auf zwei verschiedene Art und Weisen genutzt werden. Beide haben gemein, dass durch den URI, der in dem Header Feld angegeben wird, eine Ressource lokalisiert werden kann. Der erste Einsatzort ist innerhalb einer Client Anfrage. Wird während einer Client Anfrage eine Entität im *entity body* mitgeschickt und diese besitzt bereits einen URI, so kann durch das Content-Location Header Feld der URI angegeben werden. An der zweiten Verwendungsmöglichkeit ist der Server beteiligt. Besitzt eine Ressource mehrere Entitäten und diese besitzen separate URIs, so kann der Server diese dem Client mit dem Content-Location Header Feld mitteilen. In späteren Verlauf, kann ein Client diese dann direkt ansprechen.

- Content-Type
  Da es möglich ist, dass ein Client eine Anfrage an den Server stellt, die eine Auswahl an *Media Types* enthält, die er entgegennimmt, muss der Server, den *Media Types* seiner Antwort festlegen. Hierfür kann das Content-Type Header Feld genutzt werden.

2.3.2 Übermittlung von Repräsentationen im Nachrichtenkörper

Die Nutzung von Ressourcen bietet einige bereits geschilderte Vorteile. Jedoch sind Ressourcen keine Daten, die von einem Client zum Server oder umgekehrt geschickt werden können. Sie sind abstrakt. Daher wird diese Aufgabe durch die Repräsentationen einer Ressource übernommen. Sie stellen den aktuellen Zustand einer Ressource dar und sind somit konkrete und reale Informationen zu einer Ressource. Für eine Ressource können beliebig viele Repräsentationen existieren. So kann ein Server Repräsentationen einer Ressource in Form eines XML Dokument, einer Web Seite oder eines durch Kommata getrennten Textes anbieten [22] [19].

Auswahl der passenden Repräsentation

Die Möglichkeit der Existenz mehrerer Repräsentationen für eine Ressource, macht es nötig, einen Auswahlmechanismus für Repräsentationen anzubieten. Es existieren verschiedene Wege, dies zu ermöglichen. Im folgenden werden zwei vorgestellt.

Auswahl durch Ressourcengranularität Durch eine Erhöhung der Granularität von URIs ist es möglich, Repräsentationen zu unterscheiden. Dabei

existiert zu jeder Ressource maximal eine Repräsentation. Ein Beispiel dafür ist die Repräsentation einer Webseite in verschiedenen Sprachen (Abb. 2.10). [22]

```
http://www.example.com/releases/104.en

http://www.example.com/releases/104.es
```

Abbildung 2.10: Auswahl von Repräsentationen anhand des URI

Auswahl mit Hilfe des HTTP Protokolls Das HTTP Protokoll bietet ebenfalls Optionen, die zur Auswahl der passenden Repräsentation einer Ressource genutzt werden können. Diese werden durch *header* innerhalb einer Nachricht repräsentiert und betreffen vorrangig das Format der Repräsentation. Ein Client kann in einer Anfrage Werte für den *accept header* setzen. Die Reihenfolge der angegebenen Formate bestimmt die Priorität. Der Server arbeitet die Liste der Reihenfolge nach ab. Kann ein Server, eine Ressource im angeforderten Format liefern, so macht er dies in Form einer Antwort. Dabei enthält der Nachrichtenkörper die Repräsentation. Damit dem Client ersichtlich wird, welche Repräsentation er erhält, wird diese Information in der Antwort des Servers, innerhalb des *content-type header* festgelegt.

Repräsentation von Java Objekten

Innerhalb dieser Arbeit sind Java Objekte die Ressourcen. Repräsentationen beschreiben diese Objekte und werden von Client und Server genutzt, um Daten auszutauschen. Dabei unterscheiden sich die Anforderungen an Repräsentationen zwischen Client und Server. Der Client muss durch eine Repräsentation dem Server alle Informationen bereitstellen, sodass dieser die Werte von Feldern setzen kann, er alle Parameter für einen Methodenaufruf hat und er neue Objekte instanziieren kann. Repräsentationen, die vom Server an den Client geschickt werden, liefern Strukturinformationen zu Objekten, die weiteres Traversieren durch den Objektgraph ermöglichen oder konkrete Werte eines Objektes liefern. Im folgenden werden die Repräsentationen eingehender untersucht und an Beispielen erläutert.

Repräsentationen in Anfragen Mit den Informationen, die ein Server den Repräsentationen einer Anfrage entnimmt, kann dieser Werte von Feldern setzen und Parameter für Methodenaufrufe oder Objektinstanzen er-

stellen. Der Anspruch an eine Repräsentation ist es dabei nicht, dem Server, „fertige“ Objekte zur Verfügung zu stellen, wie es durch Serialisierung [23] möglich ist. Vielmehr soll der Server die übergebenen Werte interpretieren, um daraus neue Objekte zu erstellen oder auf bereits bestehende Objekte zuzugreifen. Dafür ist es notwendig, die Anforderungen an die Repräsentation vorzugeben.

- Struktur der Repräsentation
  Für die Erstellung eines Java Objektes, oder den Aufruf einer Methode, ist die Reihenfolge der Parameter wichtig. Die Struktur der Repräsentation muss die Reihenfolge der Werte beachten, da eine falsche Reihenfolge, die Erstellung eines falschen Objektes oder den Aufruf einer falschen Methode bewirken kann. Aufgrund dieser Strukturbedingung bieten sich Listen an. Da ein Parameter ebenfalls ein nichtatomaren Typ sein kann, ist dieser ebenfalls als Liste darzustellen.

- Format der Repräsentation
  Nicht jedes Format (*MIME type*) ist für die beschriebene Repräsentation geeignet. Das Format muss eine Syntax für Listen besitzen. Zusätzlich muss das Format ein Element, für *hyperlinks* besitzen oder die Möglichkeit, eines zu erstellen. Dieses Element wird benötigt, um auf eine bestehende Ressource zu verweisen.

Diese Anforderungen erfüllend, verfügt das Framework über eine Referenzrepräsentation im JSON Format [24], für die Übergabe von Parameter, von einem Client zu einem Server. Die Repräsentation besteht dabei aus JSON Objekten. JSON Objekte sind unsortierte Listen von *name-value*-Paaren [24]. Grundsätzlich sind alle Werte vom Typ *string*. Die Interpretation übernimmt der Server. Der Bezeichner (*name*) ist frei wählbar, mit Ausnahme von „res“ und „uri“. Diese Bezeichner besitzen eine besondere Bedeutung beim Verweis auf existierende Ressourcen.

Abbildung 2.11 enthält Beispiele für die Repräsentationen von keinen, einen und mehreren Parametern. Die Übermittlung von keinem Parameter kann durch ein leeres JSON Objekt dargestellt werden. Ebenfalls ist es möglich, den Nachrichtenkörper leer zu lassen. Für die Übermittlung von einem Parameter wird ein neues JSON Objekt angelegt. Dieses enthält ein *key-value*-Paar. Da die Parameter nach Reihenfolge bestimmt werden, ist der Bezeichner (key) nicht relevant. Der Wert (value) innerhalb des Paares, enthält den Wert für den Parameter. Werden mehrere Parameter benötigt, beispielsweise für einen Methodenaufruf, so werden mehrere *key-value*-Paare im JSON Objekt deklariert.

```
#no parameter
{}

#one parameter
{
    "name" : "carl"
}

#multiple parameter
{
    "name" : "carl",
    "age" : "20"
}
```

Abbildung 2.11: drei getrennte Repräsentationen für die Übermittlung von keinen, einen und mehreren Parameter

Mit Hilfe von *links* kann innerhalb einer JSON Repräsentation auf eine bestehende Ressource verwiesen werden. Das Beispiel in Abb. 2.12 zeigt eine JSON Repräsentation, mit der drei Parameter übergeben werden. Die ersten beiden Parameter werden wie oben beschrieben deklariert. Der dritte Parameter verweist auf eine Ressource. Dies wird durch Nutzung von „res“ als Bezeichner im *key-value*-Paar deutlich gemacht. Der Wert des Paares enthält anstelle eines String-Wertes ein neues JSON Objekt. Dieses enthält einen Titel der Ressource und einen URI.

```
{
    "name" : "carl",
    "age" : "20",
    "res" : {
                "title" : "friends"
                "uri" : "http://example.com/person/carl/friends"
            }
}
```

Abbildung 2.12: Drei Parameter (”carl”, ”20”, ”friends”) in einer Repräsentation; friends verweist auf eine bereits bestehende Ressource

Repräsentationen in Antworten Ebenso wie in den Repräsentationen einer Clientanfrage, werden in den Repräsentationen einer Serverantwort Daten transportiert. Im Gegensatz zu einer Anfragerepräsentation, die Daten für einen **neuen** Status einer Ressource übermittelt oder im Fall einer Methoden-Ressource Daten für Parameter liefert, wird in eine Serverrepräsentation der **aktuelle** Status einer Ressource versendet. Der Status einer Ressource gliedert sich in zwei Informationstypen. Zum einen, in Informationen über die aktuellen Werte einer Ressource, zum Zeitpunkt der Antwort und zum anderen strukturelle Informationen. Die Wertinformationen entsprechen den Java Objekten. Strukturelle Informationen, werden aus den öffentlichen Feldern und Methoden der Objekte gewonnen.

Für eine Ressource, können mehrere Repräsentationen existieren. Der Anzahl von Repräsentationen für Ressourcen, ist nur durch die Anzahl der verschiedenen Formate (*MIME Type*) begrenzt. Jedoch ergibt die Erstellung einer Repräsentation einer Ressource nicht für jedes dieser Formate einen Sinn. Daher werden im folgenden Leitlinien zur Erstellung von Repräsentationen vorgestellt. Diese haben keinen bindenden Charakter, da die Vorteile, die ein Client, durch die Auswahl eines bestimmten Repräsentationsformates bzw. einer Repräsentationsform hat, nicht exakt benannt werden können.

- Repräsentationsformat, mit *link*-Element
  Es existieren mehrere Anwendungsfälle, in denen es nötig ist, auf eine URI zu verweisen. Aus diesem Grund ist angeraten, ein Format für eine Repräsentation zu wählen, welches ein *link*-Element besitzt bzw. die Erstellung eines solchen erlaubt.

- maschinen- und menschenlesbares Format
  Einige Formate können durch Mensch und Maschine gleichermaßen gut bearbeitet werden.

Im folgenden werden Repräsentationen, anhand eines Beispiels (Abb. 2.13), für die Übermittlung von Wert- und Strukturinformationen von Java Objekten vorgestellt. Diese haben das XHTML-Format, welches die oben aufgeführten Richtlinien erfüllt.

- Sturkturrepräsentation von Objekten
  Um eine Strukturrepräsentation, einer Ressource vom Server zu erhalten, wird dem URI ein **class**-Token hinzugefügt. Dieser Token ist reserviert und kann nur im Zusammenhang mit einer Anfrage einer Strukturrepräsentation genutzt werden. Bei Verlinkungen zu anderen Ressourcen, innerhalb einer Stukturrepräsentation,wird ebenfalls der

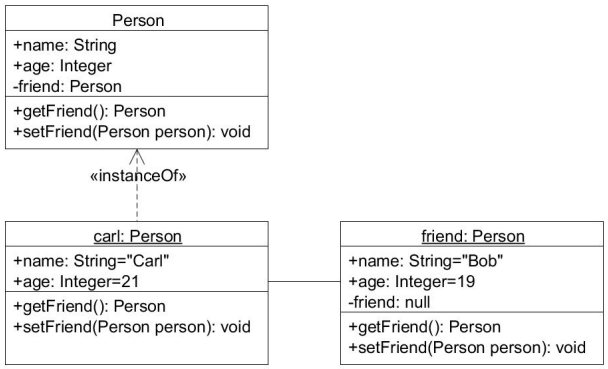

Abbildung 2.13: Objektgrundlage für Repräsentation 2

class-Token genutzt. Dies ist sinnvoll, um nicht nur die Anfrage derer Werte zu erhalten, sondern ebenso weiterführende Traversierung zu ermöglichen. Zusätzlich zu den Verlinkungen auf Felder und Methoden der Ressource, werden Beziehungen zu anderen Ressourcen angezeigt, die ebenfalls über den mitgelieferten *link* angefragt werden können. Für verlinkte Methoden-Ressourcen, ergibt sich dabei eine Besonderheit.

Durch die Möglichkeit über Verlinkungen zu anderen Ressourcen zu gelangen, wird ein zentraler Aspekt, Hypermedia as the Engine of Application State (HATEOAS), eines RESTful Services erfüllt. Dieses Prinzip schreibt die Möglichkeit des Zustandswechsels durch Verlinkungen vor.

- Wertrepräsentation von Objekten
  Mit Hilfe einer Wertrepräsentation wird der aktuelle Zustand einer Ressource vom Client zum Server übermittelt. Das Repräsentationsformat entspricht einem *MIME-Typ*, den der Server aus den akzeptierten Formaten des Client anbieten kann.

  Innerhalb einer Wertrepräsentation stehen die Werte von Ressourcen im Vordergrund. Jedoch ist es angebracht, eine Verlinkung auf die Struktur der Ressource anzubieten. Hierfür existieren sowohl Vor-, als auch Nachteile und die Sinnhaftigkeit, hängt vom Anwender der Repräsentation ab. Wird die Repräsentation maschinell verarbeitet und ausgewertet, bringen zusätzliche Verlinkungen nur einen Vorteil, wenn die Maschine darauf eingehen kann, anderenfalls sind sie überflüssig. Wird die Repräsentation dagegen von einem Browser angefragt, der von einem Menschen bedient wird, ergibt ein Verweis auf die Struktur den Vorteil, dass der Anwender darauf klicken kann, anstelle eine

self

1. self

public fields

1. name
2. age

public methods

safe methods

1. getFriend

unsafe methods

1. setFriend

related ressources

Abbildung 2.14: Darstellung der XHTML Repräsentation aus Abb. 2 im Browser

erneute Eingabe in die Adresszeile zu tätigen.

2.3.3 Abbildung von Java Operationen auf HTTP Methoden

Nachdem gezeigt wurde, wie Ressourcen adressiert werden 2.2.9, welche Interaktionsmöglichkeiten Java, für Objekte 2.3.1, inklusive ihrer Felder und Methoden bereit hält und wie die Repräsentationen für die Datenübermittlung zwischen Client und Server aussieht, wird darauf aufbauend, nun ein Schema zur Abbildung auf HTTP Methoden folgen.

Auslesen eines Feldes mit GET

Das Lesen eines Feldes erfolgt mit Hilfe der GET Methode von HTTP. Die GET Anfrage hat keine Änderung der Datenbasis zur Folge. Wurde die Anfrage erfolgreich ausgeführt, so erhält der Client die Antwort des Servers, welche die Repräsentation der Ressource im *entity body* enthält. Das Format der Repräsentation wird nach dem, in Abschnitt 2.3.2 vorgestellten Verfah-

ren ermittelt. Abbildung 2.15 zeigt ein Beispiel einer solchen Anfrage mit möglicher Serverantwort.

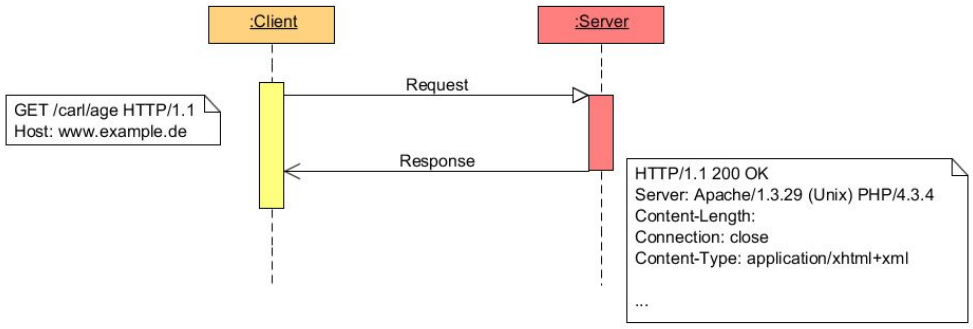

Abbildung 2.15: GET Anfrage auf ein Feld; Repräsentation im XHTML Format

Durch *Modifier* können Felder als *private* markiert sein. Besitzt das Feld eine öffentliche *getter* Methode, so kann das Feld mit dem URI adressiert werden, als wäre es öffentlich, also in der Form *objectname/fieldname*.

Setzen eines Feldes mit POST

Für das Setzen von Werten eines Feldes wird die POST Methode genutzt. Der Wert, der dem Feld zugeordnet werden soll, wird innerhalb des *entity body's*, wie in Abschnitt 2.3.2 beschrieben, übertragen.

Löschen einer Objektreferenz mit DELETE

Das Löschen von Ressourcen muss differenziert betrachtet werden. Es ist nicht möglich ein Java Feld zur Laufzeit zu Löschen. Jedoch kann der Wert eines Feldes auf *null* gesetzt werden, was dem Löschen einer Objektreferenz entspricht. Das Anfragen der Ressource wäre nach dem Löschen unter gleichem URI immer noch möglich, jedoch mit dem Unterschied, dass die Repräsentation nicht mehr den gleichen Wert liefert. Für Ressourcen, die auf Methoden abbilden, ist die DELETE Methode nie sinnvoll. Ist die Ressource, die gelöscht werden soll, Teil einer Collection, so kann diese, mit DELETE vollständig entfernt werden. Das heißt, dass der URI, der die gelöschte Ressource identifiziert hat, nach dem Löschen keine oder eine andere Ressource identifiziert. Der URI kann somit auch ungültig sein.

Erstellen von Objekte mit PUT

Bereits mit der POST Methode werden neue Objekte erstellt. Die Erstellung bezieht sich dabei jedoch auf einen bereits existieren URI. Im Gegenzug dazu, bietet innerhalb des Frameworks, die PUT Methode, die Möglichkeit, ein Objekt an einen URI zu speichern, der möglicherweise bisher nicht existiert. Dies ist notwendig, wenn Collections Werte hinzugefügt werden sollen. Wird bereits mit dem URI der Anfrage eine Ressource identifiziert, so wird diese Ressource durch die Werte innerhalb des *entity body's* ersetzt.

Methodenaufruf mit GET

Die GET Methode ist als eine sichere Methode spezifiziert 2.2.5. Daher ist es nicht möglich, beliebige Java Methoden mit GET anzufragen, da diese seiten-effektbehaftet sein können. Jedoch existieren Methoden, *getter*-Methoden, die diese Eigenschaft nicht haben und daher der Aufruf mit GET legitim ist. *Getter*-Methoden ermöglichen die Abfrage von Werten privater Felder eines Objektes. Der Rückgabewert der *getter*-Methode entspricht somit dem Wert des zugeordneten Feldes.

Methodenaufruf mit POST

Alle Java Methoden, die keine *getter*-Methoden sind, können Seiteneffekte verursachen. Daher werden Anfragen auf unsichere Methoden mit HTTP POST gestellt. Dabei enthält der *entity-body* die Parameter, die für die Ausführung der Methode benötigt werden.

Die Antwort des Servers enthält eine Repräsentation des *return*-Wertes der Methode. Ist der *return*-Wert vom Typ *void*, so ist die Repräsentation leer.

Kapitel 3

Praktischer Teil

Aufbauend auf den theoretischen Grundlagen dieser Arbeit, wurde ein System zur Freigabe von Java Objekten entwickelt. Der Zugriff erfolgt durch eine dynamische REST Schnittstelle. Die Ressourcen, Java Objekte einschließlich ihrer öffentlichen Felder und Methoden werden innerhalb des Systems durch einen URI identifiziert. Jeder *token* innerhalb des Pfades bildet dabei auf ein Java Objekt ab. Dies kann zum einen das Feld eines Objektes oder das Ergebnis einer Methodenausführung sein. Die Namensgebung der *token* ergibt sich aus den Namen der Felder und Methoden und weicht nur in Ausnahmefällen davon ab. Für die Kommunikation zwischen Client und Server, wird das HTTP Protokoll genutzt. Mit Hilfe der HTTP Methoden und URIs ist es möglich, Ressourcen anzufragen. War die Anfrage erfolgreich, kann es zur Übertragung des aktuellen Status der Ressource kommen. Die Ressource wird in Form einer Repräsentation übertragen. Ergänzend zu der Methode und dem URI, kann der Client der Anfrage weitere Informationen, in Form von *header* hinzufügen. Zentrale Bedeutung kommt dabei dem *accept-header* der Anfrage zu. Mit ihm kann der Client eine Auswahl von Repräsentationsformaten (*MIME Types*) wählen, die er akzeptiert. Der Server ist für die Auswahl des passenden Repräsentationsformat zuständig. Kann er ein passendes Format anbieten, so übermittelt er dies im *entity-body* unter Angabe des Formates im *content-type-header*. Das Repräsentationsformate übermittelt zwei Arten von Informationen: Struktur- und Wertinformationen. Repräsentationen werden mit Verlinkungen zu anderen Ressourcen versehen. Dies führt, in einigen Anwendungsfällen, zu einer Erhöhung der Benutzbarkeit. Vor allem haben die Verlinkungen jedoch die Aufgabe, einen Zustandswechsel herbeizuführen.

Nach dieser Zusammenfassung, ergeben sich drei Bereiche der Umsetzung des Frameworks, die im folgenden näher betrachtet werden. Dabei wird für jeden dieser Bereiche die Implementierung bzw. das Design vorgestellt.

3.1 Struktur und Aufgabenteilung des Frameworks

Das Framework gliedert sich in drei Aufgabenbereiche. Diese sind:

- Schnittstelle für Clients zum Dienst
  Das Framework bietet eine Webschnittstelle an. Diese wird durch ein Servlet [25] realisiert. Ein Servlet ist Java basierte Webkomponente, die durch einen Servlet-Container verwaltet wird und dynamisch erstellte Inhalte anbietet. Der Servlet-Container ist Teil eines Web oder Applikation Servers, der Netzwerkdienste anbietet mit denen Anfragen und Antworten gesendet werden können. Jeder Servlet-Container muss HTTP als Übertragungsprotokoll anbieten. Das Servlet innerhalb des Frameworks hat die Aufgabe, Anfragen von Clients entgegen zu nehmen und Antworten auf Anfragen zu verschicken. Anfragen werden nach den HTTP Methoden getrennt behandelt. Für diese Trennung hält das Servlet je eine Java Methode für jede HTTP Methode bereit. Mit Zuweisung der Anfrage, zu einer Java Methode, welche die Anfrage weiter bearbeitet, wird der erste Schritt der Interpretation der Anfrage gemacht. Innerhalb der Java Methode wird der zweite Schritt zum Evaluationsprozess getätigt. Die Anfrage wird dem *Evaluator* übergeben. Ergibt die Evaluation Fehler, so wird dies dem Client, durch eine Serverantwort, mit passender Status Zeile, mitgeteilt. War die Anfrage erfolgreich, so enthält das Ergebnis der Evaluation eine Ressource. Diese Ressource wird dem *ContentSelector* übergeben. Gibt es eine Übereinstimmung zwischen den akzeptierten Repräsentationsformaten der Anfrage und den aktuell angebotenen Repräsentationsformaten des Servers, erhält das Servlet die dazugehörige Repräsentation. Diese Repräsentation wird zusammen mit dem Status der Anfrage, vom Servlet in eine HTTP Antwort verpackt und dem Client gesendet. Der Status für die Status Zeile wird im Evaluationsprozess bestimmt.

- Evaluation der Anfrage
  Der *Evaluator* übernimmt die Evaluation der Anfrage. Die Evaluation der Anfrage erfolgt rekursiv. Der Abschnitt 3.2.1 beschreibt ausführlich, wie mit Reflection, der URI einer Anfrage ausgewertet wird. Bei der Auswertung kann es zu Fehlern kommen, die in Java durch Ausnahmen (engl. Exceptions) abgefangen und bearbeitet werden. Für das Framework sind vor allem Fehler, wie:

  - Zugriff auf ein Feld oder eine Methode verweigert

  - Fehlerhafte Zuweisung von Parameter zu einem Methodenaufruf
  - Fehlerhafte Zuweisung von Werten zu einem Feld
  - Angefragtes Feld oder angefragte Methode existiert nicht

  relevant. In der Serverantwort müssen Fehler auf passende HTTP *Error Codes* abgebildet werden. Neben den Fehlern, existieren Ereignisse, die der Server, dem Client ebenfalls mitteilen sollte. Zu dieser Kategorie gehören beispielsweise:

  - Erfolgreiche Anfrage
  - Erfolgreiches Erstellen eines Objektes

  Ereignisse dieser Art, müssen im Ergebnis der Evaluation berücksichtigt und dem Servlet mitgeteilt werden, damit dieses die richtige Status Zeile für eine Antwort erstellen kann.

- Repräsentation einer Ressource auswählen und erstellen
  Der dritte Aufgabenbereich ist die Auswahl und Erstellung einer Repräsentation einer Ressource. Abschnitt 3.3 erläutert detailliert das Vorgehen zur Ermittlung der Repräsentation.

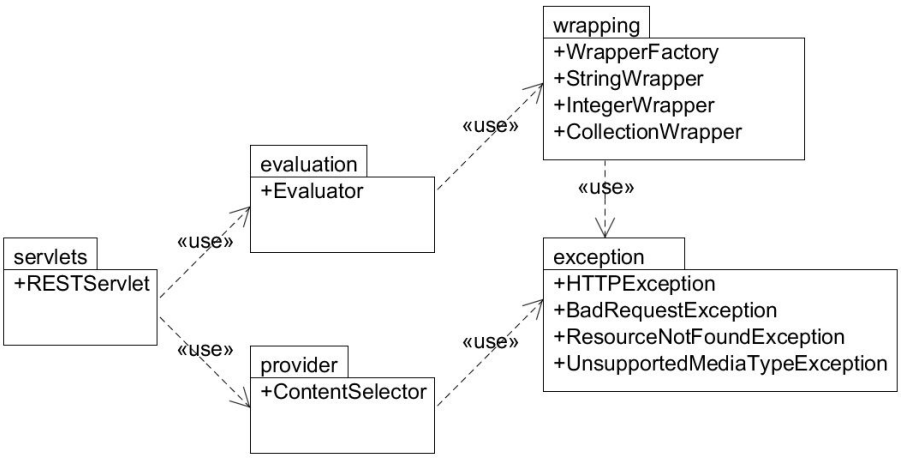

Abbildung 3.1: Darstellung der wesentlichen Struktur des Frameworks in einem *UML Package Diagram*

3.2 Java Reflection als Mittel zur Introspektion von Objekten

Java Reflection ermöglicht es, Informationen zu Feldern, Methoden und Konstruktoren von Java Objekten zur Laufzeit abzurufen.

Aus mehreren Gründen, muss innerhalb des Frameworks, auf das Reflection Sprachfeature zurückgegriffen werden. Wie bereits in 2.1.4 beschrieben, richten sich Implementierungen für RESTful Web Services nach der Spezifikation des JSP. Diese sieht Annotationen für Deklaration von Ressourcen vor. Dies steht im Gegensatz zu der Anforderung an das Framework, eine dynamische REST Schnittstelle anzubieten. Um eine Traversierung durch Objektgraphen zu ermöglichen, wird der URI so gestaltet, dass er für Java Reflection geeignet ist. Wie das Traversieren unter Nutzung von Reflection ermöglicht wird, ist in Abschnitt 3.2.1 beschrieben. Ein weiterer Grund für die Nutzung von Reflection ist, dass keine Informationen über die Java Typen der Parameter innerhalb des URIs bzw. einer übergebenen Repräsentation mitgesendet werden. Diese müssen ebenfalls parallel zur Auswertung des URIs ermittelt werden. Beschrieben wird das Vorgehen in Abschnitt 3.2.2

3.2.1 Traversierung des Objektgraphens

Für die Traversierung durch einen Objektgraphen ist es nötig, ein Einstiegsobjekt festzulegen. Dieses wird in einer Java HashMap, mit einem String-Bezeichner als Schlüssel, gespeichert (Abb. 3.2). Der Bezeichner bildet den Wert, der innerhalb des URIs für das Objekt genutzt wird. Beginnt ein URI mit einem gültigen Startobjekt, d.h. es existiert ein Bezeichner mit gleichem Wert, wie der erste Token des Pfades, des URIs, so beginnt die Introspektion. Das Einstiegsobjekt bildet den Kontext für den nächsten Token, der ausgewertet wird.

Auswertung von Folgetoken nach gültigem Einstiegsobjekt:

Die Auswertung aller Token, die nach dem Einstiegstoken kommen, folgt der folgenden Rekursion:

- kein weiterer Token vorhanden
  Existiert kein weiterer Token, so bricht die rekursive Auswertung ab. Das Ergebnis der Anfrage, ist das Objekt des aktuellen Kontextes.

- weiterer Token vorhanden

```
1   /**
     * binded objects
3    */
    public HashMap<String, Object> bindings = new HashMap<>();

    /**
7    * this method binds an object to an identifier. after binding, the object
     * can be accessed by the identifier within an uri.
9    *
     * @param id
11   * identifier for object.
     * @param entry
13   *    entry object for object traversal
     */
15  public void bind(String id, Object entry) {
      bindings.put(id, entry);
17  }
```

Abbildung 3.2: Implementierung für das Binden eines Einstiegsobjektes

- der Token identifiziert ein Feld oder eine Methode im aktuellen Kontext.
  Das Objekt der Ressource wird mit Reflection (3.2.1) ermittelt und bildet den neuen Kontext für weitere Rekursionsschritte. Der nächste Rekursionsschritt wird mit neuem Kontext und den verbleibenden Token gestartet.
- der Token besitzt eine besondere Bedeutung (bspw. **class**-Token)
  Die Aufgabe des Tokens mit besonderer Bedeutung wird ausgeführt. Die Rekursion bricht ab.
- kein gültiger Token
  Die Auswertung des Tokens ergibt einen Fehler. Es wird ein entsprechender Fehlercode gesendet und die Rekursion bricht ab.

Manipulation von Feldern durch Reflection: Felder eines Objektes werden durch ihren einzigartigen Bezeichner identifiziert. Mit diesem Bezeichner, kann ein Feld im Kontext eines Objektes durch Reflection angefragt werden (Abb. 3.3). Der Reflection API bietet für die Ermittlung eines Feldes die Methode *getDeclaredField* an (Abb.3.3 Zeile 4). Das Erhalten von Werten eines Feldes, wird mit der Methode *get* ermöglicht, wobei das Objekt (*parent*-Objekt), dem das Feld angehört, den einzigen Parameter bildet (Abb.3.3 Zeile 5). Der Methode zum Setzen, *set*, wird zusätzlich zum *parent*-Objekt, ein weiterer Parameter, welcher den neuen Wert des Feldes darstellt,

übergeben (Abb.3.3 Zeile 7).

```
1   Person bob = new Person("Bob", 19, null);
    Person carl = new Person("Carl", 20, bob);

    Field nameField = carl.getClass().getDeclaredField("name");
5   String name = (String) nameField.get(carl); // name.equals("Carl")

7   nameField.set(carl, "Carl Larc");
    String newName = (String) nameField.get(carl); // name.equals("Carl Larc")
```

Abbildung 3.3: Java Code zur Manipulation von Feldern per Reflection

Manipulation von Methoden durch Reflection Die eindeutige Identifizierung einer Methode, erfolgt über die Signatur der Methode. Sie besteht aus dem Namen und den Java Typen der Parameter der Methode. Das kommentierte Beispiel in Abbildung 3.4 zeigt den Einsatz von Reflection zum Ermitteln und Ausführen von Methoden. Zu beachten ist *getMethod*, welches den Namen der Methode und ein Array von Klassen entgegennimmt um die Methode zu identifizieren. Ist eine Methode erfolgreich identifiziert worden, so kann sie mit *invoke*, unter Angabe des Objektes, von der die Methode ausgeführt werden soll und eines Arrays mit Objekten, welche die Parameter darstellen ,ausgeführt werden.

```
    Person bob = new Person("Bob", 19, null);
2   Person carl = new Person("Carl", 20, bob);
    //get method
4   Method getFriendMethod = carl.getClass().getMethod("getFriend", new Class<?>[]{});
    //invoke method
6   Person carlsFriend = (Person) getFriendMethod.invoke(carl, new Object[]{});
    System.out.println(carlsFriend.name); // Bob
8   Person alice = new Person("Alice", 18, null);
    Method setFriendMethod = carl.getClass().getMethod("setFriend", new Class<?>[]{Person.class
        });
10  setFriendMethod.invoke(carl, new Object[]{alice});
    System.out.println(carlsFriend.name); // Alice
```

Abbildung 3.4: Java Code zur Ausführung von Methoden per Reflection

3.2.2 Ermittlung von Java Typen

Parameter werden als Repräsentationen zwischen Client und Server ausgetauscht. Für den Server ist diese textuelle Form nicht brauchbar. Er operiert auf und mit Objekten. Aus diesem Grund müssen aus der Repräsentation, Objekte extrahiert werden. Dies erfolgt in mehreren Schritten und muss für *atomare* und komplexe Parameter differenziert betrachtet werden.

Für die Interpretation von *atomaren* Typen wird die Java Expression Language Bibliothek [26] genutzt. Die Bibliothek wird genutzt, um aus Expressions, Java Objekte zu erstellen. Dies gelingt für alle *atomaren* Typen out-of-the-box.

Komplexe Parameter werden, wie in 2.3.2 beschrieben, so weit zerlegt, dass für die Initialisierung nur noch auf *atomare* Typen zurückgegriffen wird. Diese können dann zu Objekten aufgelöst werden. Sind alle *atomaren* Objekte aufgelöst, so kann der Konstruktor des komplexen Objektes mit den *atomaren* Objekten aufgerufen werden (siehe Anhang 1).

3.3 Selektionsmechanismus zur Auswahl eines Repräsentationsformats

Innerhalb einer Anfrage, kann der Client eine Auswahl von Formaten angeben, in der er eine Repräsentation innerhalb der Antwort des Servers entgegennimmt. Der Server entscheidet aufgrund dieser Angabe und den Formaten, die er anbieten kann, welches Format sich am besten für eine Repräsentation in der Antwort eignet.

3.3.1 Implementierung des Selektionsmechanismus

Wurde eine Ressource erfolgreich angefragt, wird das Ergebnis in ein Wrapperobjekt gespeichert. Zusätzlich zum Speichern, haben Wrapperobjekte die Aufgabe, das Ergebnis der Anfrage, in eine Repräsentation zu überführen. Dies erfolgt über *provider*-Methoden. Welche Repräsentation ein Wrapper anbietet, wird aus den konkreten ContentProvider Schnittstellen, die er implementiert, ersichtlich. Implementiert ein Wrapper den XHTMLContentProvider, so besitzt der Wrapper die Methode *provideXHTMLContent()*, die eine XHTML Repräsentation in als Byte-Array anbietet (Abb. 3.5). Die Schnittstelle besitzt ein Feld, welches das angebotene Format hält. Jeder Wrapper kann beliebig viele ContentProvider Schnittstellen implementieren und damit sein Angebot an Repräsentationsformaten erweitern. Mit einem Wrapper und den aktzeptierten Formaten für Repräsentationen kann die

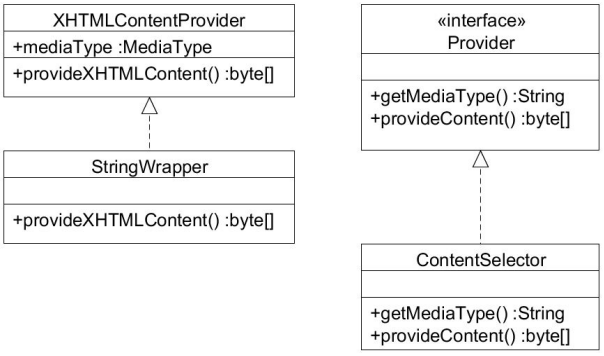

Abbildung 3.5: Klassendiagramm in UML für ContentSelector und den XHTMLContentProvider

Auswahl eines Formates erfolgen. Hierfür wird ein neues ContentSelector Objekt mit einem Wrapper und den akzeptierten Formaten instanziiert. Bei der Initialisierung, werden vom Wrapper alle ContentProvider Schnittstellen extrahiert und gespeichert. Der ContentSelector besitzt eine öffentliche Methode, *provideContent()* (Abb. 3.5). Wird diese ausgeführt, so gleicht der ContentSelector die akzeptierten Formate mit den angegebenen Formaten aus den extrahierten Schnittstellen ab. Kommt es zu einer Übereinstimmung, wird die Methode der Schnittstelle ermittelt und im Wrapper ausgeführt. Das Ergebnis ist der *return* der *provideContent* Methode des ContentSelectors. Wird keine Übereinstimmung erzielt, so kann die Ressource nicht in einem der akzeptierten Formate geliefert werden. Dies führt zu einer Exception, die dem Client als Antwort des Server mit Status Code 415 präsentiert wird.

Kapitel 4

Fazit und Ausblick

Das Ziel dieser Arbeit war, ein System zur Freigabe von Java Objekten per dynamischer REST Schnittstelle zu erstellen. Dafür wurde aufbauend auf eine Literaturrecherche, in das Thema RESTful Web Services eingeführt. Eine Analyse der Spezifikation für Java RESTful Web Services und der bestehenden Referenzimplementierung hat ergeben, dass eine eigene Implementierung für das Softwaresystem notwendig war. Daher wurde sowohl ein Schema zur Abbildung von URIs auf Java Ressourcen, als auch eines für die Abbildung von HTTP Methoden auf Java Operation erstellt. Diese Schemata nutzend, wurde das System implementiert.

Durch die Nutzung des Systems ergeben sich viele positive Aspekte. Dem Anbieter von Ressourcen, bleibt das Deklarieren von Ressourcen, die sie über eine Web Schnittstelle anbieten wollen erspart. Stattdessen werden Objekte freigegeben, die ein Nutzer selbst nach den gewünschten Daten „durchstöbern“ kann. So entsteht für den Anbieter ein sehr geringer Aufwand für das Bereitstellen von Ressourcen. Nicht für alle, aber für sehr viele Java Typen können Ressourcen angeboten werden. So ist die Auswahl nicht auf Strings, Intergers und Booleans beschränkt, sondern erstreckt sich auch beispielsweise auf Images, Collections. Zu dieser Vielzahl an unterschiedlichen Typen von Objekten, die als Ressourcen angeboten werden können, existiert die Möglichkeit, zur Auswahl des Formates der Repräsentation. Wird ein geeignetes Format gewählt, beispielsweise XML, so ist auch eine maschinelle Verarbeitung der Repräsentationen möglich. Dies stellt eine große Flexibilität dar, welche die Anzahl der möglichen Konsumenten eines Services erhöht. Diese wird ebenfalls durch die von REST genutzten etablierten Standard erhöht. Viele Programmiersprachen bieten Bibliotheken für die Erstellung von HTTP Anfragen und Auswertung von HTTP Antworten bereit. Mit Hilfe dieser Bibliotheken können auch Java-fremde Programmiersprachen auf die, durch den Service bereitgestellten Daten zugreifen.

Jedoch ergeben sich auch nicht zu vernachlässigende Nachteile des Frameworks. Vor allem die Übermittlung von komplexen Parametern erweist sich als schwierig. Mit GET Anfragen ist es nicht möglich, komplexe Parameter für die Objektidentifizierung zu übergeben. Die Anfrage mit anderen Methoden bietet keine Option, da jede HTTP Methode feste Aufgaben hat. Somit bleibt beispielsweise der Zugriff auf Maps, die komplexe Objekte als Schlüssel haben, verwehrt. Theoretisch möglich, muss aber aus Übersichtgründen die Ausführung von mehreren Methoden, die keine *getter*-Methoden sind, verweigert werden. Anderenfalls würde ein URI für einen Menschen schwer schreib- und lesbar. Es müsste ein Mechanismus erstellt werden, der die Trennung der Parameter nach Methodenzugehörigkeit ermöglicht. Auch ist die Nutzung im Browser teilweise problematisch. Sollen *links* komfortabel genutzt werden können, so muss mit Scriptsprachen, wie Java Script nachgeholfen werden. Ein Beispiel, wo die Hilfe von Scriptsprachen notwendig ist, stellt das Absenden einer HTML *form* per *link*-klick dar. Scriptsprachen können jedoch nicht überall eingesetzt werden. Striktes XHTML verbietet den Einsatz von Scriptsprachen, da diese die Dokumentenstruktur neu arrangieren können. Ein weiterer Nachteil an HTML Repräsentation ist die fehlende Unterstützung für PUT und DELETE. Diese können nur über einen *workaround* genutzt werden. PUT und DELETE Anfragen werden dabei durch einen Parameter innerhalb einer POST Anfrage gekennzeichnet. Ist der Parameter gesetzt, so wird keine POST Anfrage ausgeführt, sondern eine dem Parameter entsprechende. Somit wird die klare Struktur der Anfragen umgangen. Auch die Implementierung des Servlets musste auf diesen Umstand hin angepasst werden. Der größte Nachteil des Frameworks bildet der Verlust der Datenkapselung [1]. Nutzern auf der Clientseite werden nicht nur Nutzdaten übermittelt. Um das Framework bedienen zu können, muss der Nutzer Kenntnis über die Struktur der Objekte und deren Beziehungen zueinander haben, bzw. er kann sie mit Hilfe von Strukturanfragen erhalten. Bei kleinen Objekthierarchien noch überschaubar, kann es vor allem bei Freigabe von größeren Systemen zum Verlust der Übersicht über freigegebene Daten kommen.

Betrachtet man das Framework, wie zu Beginn dieser Arbeit verdeutlicht, nicht als Ersatz für RPC bzw. für Web Services, die RPC nutzen, so bietet dieses Framework einen Mehrwert für all diejenigen, die Java Ressourcen ohne großen Aufwand im Web verfügbar machen wollen.

Mit der ersten Implementierung ist die Arbeit am Framework noch nicht abgeschlossen. Es wurde gezeigt, dass die Implementierung eines solchen Systems möglich ist, jedoch nicht, wie praxistauglich diese ist. Dafür wäre die Untersuchung der nichtfunktionalen Anforderungen des Systems nötig. Diese sollte vor allem Fragen über die Benutzbarkeit, Effizienz, Sicherheit und

Skalierbarkeit klären

Sicherheit und Zugriffsbeschränkungen wurden innerhalb dieser Arbeit nicht betrachtet. Eine Untersuchung dieser Themenfelder, würde dem Anbieter von Ressourcen helfen, ein Rechtesystem für den Zugriff auf Ressourcen einzurichten. Dies ist vor allem interessant, wenn es um den Zugriff auf Dateien geht, die der Server als Ressourcen anbietet.

Auch bietet HTTP, als Kommunikationprotokoll zwischen Client und Server noch ungenutztes Potenzial. So kann beispielsweise durch Cachen von Ressourcen, die Datenübertragung verringert und somit die Zugriffszeiten gesteigert werden.

Als ein Gegenstück zu dem Produkt dieser Arbeit, könnte eine Java Applikation entworfen und implementiert werden, die auf das Konsumieren der angebotenen Daten spezialisiert ist. Währenddessen könnte geprüft werden, ob eine Anpassung der Repräsentationen, bei dieser Client Server Konstellation, die Probleme beim Übermitteln von komplexen Parameter verringert werden können.

Anhänge

```
/**
 * this method creates a new instance of a class. it considers given
 * parameters. if all given parameter matches to a constructor of class, the
 * method will creates a new instance of the class.
 *
 * @param clazz the class for the new object
 * @param params an array of parameter
 * @return returns a new instance of class or null
 */
public static Object newInstanceByClassAndParams(Class<?> clazz,
    Object[] params) {
  Constructor<?>[] constructors = clazz.getConstructors();
  for (Constructor<?> constructor : constructors) {
    Class<?>[] constructorParams = constructor.getParameterTypes();
    for (int i = 0; i < constructorParams.length; i++) {
      if (params[i] == null
          || constructorParams[i].isAssignableFrom(params[i].getClass())) {
        if (i == constructorParams.length - 1) {
          try {
            return constructor.newInstance(params);
          } catch (Exception e){
            ...
          }
        }
      }
    }
  }
  return null;
}
```

Abbildung 1: Beispielimplementierung einer Methode zum Erstellen von Objekten durch Konstuktoraufruf

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>server-response</title>
  </head>
  <body>
    <h1>self</h1>
      <ol>
        <form id="field1" method="get" action="http://localhost:8080/rest/setFriend">
          <input type="hidden" name="accept" value="application/xml">
          <li><a href="http://localhost:8080/rest/carl/friend">self</a></li>
        </form>
      </ol>
    <h1>public fields</h1>
      <ol>
        <form id="public_field1" method="get" action="http://localhost:8080/rest/carl/name/class
            ">
          <input type="hidden" name="accept" value="application/xml">
          <li><a href="#" onclick="javascript:document.forms['public_field1'].submit()">name</a
              ></li>
        </form>
        <form id="public_field2" method="get" action="http://localhost:8080/rest/carl/age/class">
          <input type="hidden" name="accept" value="application/xml">
          <li><a href="#" onclick="javascript:document.forms['public_field2'].submit()">age</a>
              </li>
        </form>
      </ol>
    <h1>public methods</h1>
      <h2>safe methods</h2>
        <ol>
          <form id="method2" method="get" action="http://localhost:8080/rest/setFriend">
            <input type="hidden" name="accept" value="application/xml">
            <li><a href="#" onclick="javascript:document.forms['method2'].submit()">getFriend<
                /a></li>
          </form>
        </ol>
      <h2>unsafe methods</h2>
        <ol>
          <li>
            <form id="method1" method="post" action="http://localhost:8080/rest/setFriend">
              <p>
                <input type="hidden" name="GigID" value="1"/>
                <input type="hidden" name="accept" value="application/xml">
                <a href="#" onclick="javascript:document.forms['method1'].submit()">setFriend</
                    a>
              </p>
            </form>
          </li>
        </ol>
    <h1>related ressources</h1>
  </body>
</html>
```

Abbildung 2: Representation einer Serverantwort mit Strukturinformationen einer Ressource nach Abb. 2.13

Literaturverzeichnis

[1] B. Lahres and G. Raýman, *Objektorientierte Programmierung - Das umfassende Handbuch*. Galileo Press, 2 ed., 2009. ISBN: 978-3-8362-1401-8.

[2] R. Thurlow, "RPC: Remote Procedure Call Protocol Specification Version 2," tech. rep.

[3] "Remote Method Invocation (RMI)." http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html [online; date: 26.03.2012]. Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

[4] J. Sandoval, *RESTful Java Web Services*. Packt Publishing, 1 ed., Nov 2009. ISBN: 978-1-847196-46-0.

[5] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures* . PhD thesis, University of California, Irvine, 2000.

[6] "Java Specification Request 311." http://jcp.org/en/jsr/detail?id=311 [online; date: 26.03.2012]. This JSR requesting an API for providing support for RESTful(Representational State Transfer) Web Services in the Java Platform.

[7] "Jersey Project Site." http://jersey.java.net/ [online; date: 26.03.2012]. Jersey is the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services.

[8] "CDDL 1.1 and GPL 2 with CPE Lizenz." http://glassfish.java.net/public/CDDL+GPL_1_1.html. [online; date: 26.03.2012].

[9] "RESTEasy." http://www.jboss.org/resteasy. [online; date: 26.03.2012].

[10] "Tomcat 7." http://tomcat.apache.org/tomcat-7.0-doc/index.html, [online; date: 26.03.2012]. Apache Tomcat is an open source software implementation of the Java Servlet and Java Server Pages technologies.

[11] "JBoss Application Server." http://www.jboss.org/jbossas. [online; date: 26.03.2012].

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leachl, and T. Berners-Lee, "Hypertext Transfer Protocol," RFC 2616, Jun 1999.

[13] P. Deutsch, "GZIP file format specification version 4.3," RFC 1952, May 1996.

[14] "RESTlet." http://wiki.restlet.org/docs_2.0/2-restlet.html. [online; date: 26.03.2012].

[15] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, jan 2005.

[16] B. Burke, *RESTful Java with JAX-RS*. O'Reilly, Nov 2009. ISBN: 978-0-596-15804-0.

[17] C. Wong, *HTTP Pocket Reference* . O'Reilly, 1 ed., May 2000. ISBN: 1-56592-862-8.

[18] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," RFC 2046, Nov 1996.

[19] S. Allamaraju, *RESTful Web Services Cookbook*. O'Reilly Media Inc., 1 ed., March 2010. ISBN: 978-0-596-80168-7.

[20] E. D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC.

[21] "Java Development Kit (JDK)." http://www.oracle.com/technetwork/java/javase/downloads/index.html [online; date: 26.03.2012]. To develop Java applications and applets, you need the JDK (Java Development Kit).

[22] L. Richardson and S. Ruby, *RESTful Web Services* . O'Reilly Media Inc., 1 ed., May 2007. ISBN: 0-596-52926-0.

[23] "Java Serialization Specification." http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html [online; date: 26.03.2012].

[24] "JSON Project Site." http://www.json.org/ [online; date: 26.03.2012].

[25] "Java Servlet Technology." http://www.oracle.com/technetwork/java/index-jsp-135475.html [online; date: 26.03.2012]. Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems.

[26] "Java Expression Language - JEXL." http://commons.apache.org/jexl/ [online; date: 26.03.2012]. JEXL is a library intended to facilitate the implementation of dynamic and scripting features in applications and frameworks written in Java.