Fachhochschule Münster
Fachbereich Elektrotechnik und Informatik

Modellgetriebener Entwurf einer REST-konformen Softwareschnittstelle am Beispiel einer VPN-Management-Software

Masterarbeit

von

Hendrik Schwartke

vorgelegt im August 2010

Erstprüfer: Prof. Dr. rer. nat. Nikolaus Wulff
Zweitprüfer: Ralf Spenneberg

Danksagung

Ich möchte mich an dieser Stelle bei den Personen bedanken, die mich bei der Erarbeitung der Masterarbeit unterstützt haben. Mein herzlicher Dank gilt Herrn Prof. Dr. rer. nat. Nikolaus Wulff und Herrn Ralf Spenneberg für die Betreuung meiner Arbeit. Durch die freie Zusammenarbeit mit der Firma OpenSource Training Ralf Spenneberg kam ich zu meinem Thema.

Mein ganz besonderer Dank gilt abschließend meinen Eltern, die mir meinen Berufswunsch ermöglicht haben, mir stets helfend zur Seite standen und sich als unbeteiligte Lektoren zur Verfügung stellten.

Kurzbeschreibung

Die Realisierung lose koppelbarer Softwarekomponenten stellt eine der großen Herausforderungen der praktischen Informatik dar. Der Informationsaustausch zwischen verschiedenen Softwaresystemen, die nicht explizit zur Interaktion mit dem jeweiligen Kommunikationspartner entworfen sind, bedarf daher nicht selten einer zeitaufwändigen Modifikation der Software, die insbesondere bei einer großen Anzahl von Kommunikationspartnern, einem dynamischen Interaktionsverhalten oder bei nur einmaligem Informationsaustausch oft nicht sinnvoll ist. Dieser Sachverhalt hemmt insbesondere die unternehmensübergreifende Vernetzung von Softwarediensten und die Kooperation von Softwarekomponenten verschiedener Hersteller. Im Rahmen dieser Arbeit wird untersucht wie durch Kombination eines modellgetriebenen Entwicklungsansatzes und des Architekturstils Representational State Transfer ein lose gekoppeltes Softwaresystem realisiert werden kann. Hierdurch soll dazu beigetragen werden die lose Kopplung von Softwaresystemen zu vereinfachen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Teil I.

Einführung

1. Einleitung

1.1. Problemstellung

Mit der wachsenden Anzahl und der steigenden Komplexität von Aufgaben, die durch Software übernommen werden sollen, ist es oftmals kaum mehr möglich und sinnvoll, Software als ein monolithisches und in sich abgeschlossenes System zu entwerfen. Vielmehr werden die Anforderungen auf modulare Art und Weise mittels unterschiedlicher, miteinander kooperierender Anwendungen realisiert und somit ein verteiltes Softwaresystem geschaffen. Sowohl der Druck, Kosten durch Integration bestehender Anwendungen in das Gesamtsystem zu sparen als auch der Anspruch, die durch Software abgebildeten Prozesse flexibel, schnell und kostengünstig anzupassen, verstärken diese Tendenz.

Besonders aufgrund einer fortschreitenden Globalisierung und Vernetzung gewinnt aber auch die unternehmensübergreifende Kooperation von Software an Bedeutung. Durch diesen Umstand wird zum einen die Anzahl der potentiell miteinander agierenden Softwaresysteme drastisch erhöht. Zum anderen ändert sich oftmals auch die Art einer solchen Kooperation dahingehend, dass diese ad hoc zustande kommt und nur kurz andauert. Um die Kooperation zwischen verschiedenen Softwaresystemen, welche nicht zur Zusammenarbeit mit dem jeweiligen Kommunikationspartner konzipiert sind, zu ermöglichen, bedarf es jedoch klassischerweise einer relativ lang andauernden Planungs- und Implementierungsphase zur Anpassung der beteiligten Anwendungen. Die dadurch entstehenden Kosten rentieren sich i.A. nur, sofern die Kooperation über einen entsprechend langen Zeitraum fortbesteht. Durch den Einsatz klassischer Entwicklungsansätze ist es daher ökonomisch häufig nicht sinnvoll kurzlebige Geschäftsprozesse zu automatisieren. Somit werden viele Vorzüge einer elektronischen Datenverarbeitung nicht genutzt.

Ziel dieser Arbeit ist es daher Konzepte aufzuzeigen und zu kombinieren, mittels derer die Interaktion von Software vereinfacht und damit den geänderten Anforderungen an Software Rechnung getragen werden kann.

1.2. Lösungsansatz

Die dazu verwendeten Konzepte entstammen zum einen dem *World Wide Web* (WWW), welches ein enorm komplexes und skalierbares verteiltes System ist und Informationen und Dienste verschiedener Anbieter lose koppeln kann. Konkret wird zu diesem Zweck der Architekturstil *Representational State Transfer* (REST), welcher durchweg auf den Grundlagen des WWW basiert, aufgegriffen. Ein anderes Grundkonzept, das dieser Arbeit zugrunde liegt, ist Abstraktion, also das Weglassen von Unwesentlichem und das Herausheben des Wesentlichen. Im Rahmen dieser Arbeit wird Abstraktion insbesondere in Form eines *modellgetriebenen Entwicklungsansatzes* genutzt.

Die Kombination dieser beiden Entwicklungsansätze wird im Hinblick auf die beschriebene Anforderungssituation hin untersucht und exemplarisch, anhand der Entwicklung einer lose gekoppelten Softwareschnittstelle einer Software zur Administration von virtuellen privaten Netzwerken (VPN), angewandt.

1.3. OpenVPN

Grundlage der VPN-Management-Software ist die Open Source Software *OpenVPN*, welche es ermöglicht, VPNs zu erstellen. Ein solches VPN erlaubt eine sichere Kommunikation – vergleichbar mit der in einem privaten Netzwerk – über unsichere Netzwerke, wie z.B. dem Internet. OpenVPN besteht hierzu sowohl aus einer Client- als auch aus einer Serversoftware.

Wenngleich es zu diesem Zweck eine Reihe von verschiedenen Softwareprodukten gibt, so hat OpenVPN in den vergangenen Jahren eine große Verbreitung erlangt. Dies ist u.A. wohl auf die, im Vergleich mit vielen anderen Lösungen, einfachere Konfiguration zurückzuführen. Insbesondere im Vergleich zu dem Standard „Internet Protocol Security“ (IPSec) ist die Einrichtung und Wartung eines VPNs mittels OpenVPN oftmals erheblich einfacher.

Dennoch kann der Konfigurationsaufwand eines OpenVPN-Clients oder -Servers in Einsatzszenarien, in denen die zugrundeliegenden Netzwerke untypisch strukturiert sind oder in denen spezielle Anforderungen an das VPN gestellt werden, erheblich sein. Weiterhin umfasst die eigentliche OpenVPN-Software keine Lösung zur zentralen Administration der OpenVPN-Clients. Bei einer größerer Anzahl von Clients, z.B. durch umfangreiche Nutzung von Heimarbeitsplätzen, kann somit ein beträchtlicher Administrationsaufwand entstehen. Die im Zusammenhang mit dieser Arbeit zu erstellende VPN-Management-

Software soll es daher zum einen ermöglichen, auch eine große Anzahl von OpenVPN-Clients einfach und zentral administrieren zu können. Der Fokus dieser Ausarbeitung liegt dabei auf der Entwicklung der Softwareschnittstelle zwischen dem Server und dem Client dieser Management-Software.

1.4. Aufbau der Arbeit

Die vorliegende Arbeit ist in vier Teile gegliedert. Der nach dieser Einführung folgende Teil der Arbeit erläutert die *Grundlagen*, die im Kontext der zu erstellenden VPN-Management-Software zur Realisierung einer lose koppelbaren Softwareschnittstelle dienen. Diese Grundlagen nähern sich der Problemstellung lose gekoppelter Softwaresysteme im Allgemeinen und bestehen zum einen aus dem in Kapitel 2 beschriebenen allgemeinen Betrachtungsweise von Kommunikation informationsverarbeitender Systeme. Darauf aufbauen wird in Kapitel 3 der Architekturstil Representational State Transfer (REST) erläutert, mittels dessen sich Kommunikation von Software im Sinne des in Kapitel 2 aufgeführten Kommunikationsmodells ermöglichen lässt. Ferner wird in Kapitel 4 in die modellgetriebene Softwareentwicklung eingeführt, die im späteren Verlauf zur Beschreibung wesentlicher Schnittstellenmerkmale herangezogen wird.

Im Anschluss hieran behandelt der darauf folgende Teil dieser Arbeit die *Anwendung* dieser Grundlagen zur Erstellung einer Softwareschnittstelle im Kontext der VPN-Management-Software. Diese Schnittstelle basiert dabei zum einen auf einer REST-konformen Middleware, dessen Entwicklung in Kapitel 5 beschrieben wird. Des Weiteren wird zur Spezifikation der semantischen Elemente der Schnittstelle in Kapitel 6 auf einen modellgetriebenen Entwicklungsansatz zurückgegriffen.

Abschließend wird in Kapitel 8 ein *Fazit* gezogen.

Teil II.

Grundlagen

2. Kommunikation

2.1. Einleitung

Im Mittelpunkt dieser Arbeit steht die Kommunikation zwischen Softwaresystemen und die damit einhergehende Kopplung eben dieser. Aus diesem Grund werden im vorliegenden Kapitel die für diese Arbeit wesentlichen Merkmale von Kommunikation, im Kontext interagierender Softwaresysteme betrachtet und in Form eines *Schichtenmodells* strukturiert.

Motiviert ist die Nutzung eines Schichtenmodells zur Vereinfachung der Entwicklung lose gekoppelter Systeme durch andere Referenzmodelle, wie z.B. dem ISO OSI- (s.h. [37]) oder dem TCP/IP-Referenzmodell (s.h. [8]). Diese Schichtenmodelle haben die lose Kopplung von Hardware, welches in vielerlei Hinsicht ein ähnliches Problem wie die lose Kopplung von Software ist, ermöglicht. Im Gegensatz zu anderen Modellen fokussiert das im folgenden Abschnitt vorgestellte Modell aber nicht die technischen Anforderungen zur Übertragung von Daten, sondern geht primär auf die mit der *Verarbeitung* und *Interpretation* von Daten verbundenen Fragestellungen ein und ermöglicht eine Einordnung und Bewertung der Entwicklungsansätze die im Laufe dieser Arbeit erläutert werden.

2.2. Kommunikationsmodell

Die Informationstheorie versteht unter dem Begriff Kommunikation eine auf dem Sender-Empfänger-Modell gestützte Übermittlung von Informationen. Diese werden mittels eines gemeinsamen Zeichenvorrats kodiert und von einem Sender anhand eines Übertragungskanals an einen Empfänger übermittelt. Die übertragenen Daten lassen sich hierbei, angelehnt an [18], bzgl. Kodierung, Syntax und Semantik untersuchen.

Kodierung Die Kodierung beschreibt den zugrundeliegenden Zeichenvorrat mittels dessen ein Datum gebildet ist.

Syntax Die Syntax ist das Regelwerk zur Bildung von Wörtern auf Basis des zugrundeliegenden Zeichenvorrats.

Schicht	Name	Fragestellung	Beispiele
4	Bedeutung (Semantik)	Was *bedeuten* Daten?	XML Linking Language, ATOM
3	Struktur (Syntax)	Wie werden Daten *strukturiert*?	XML, Notation 3, JSON
2	Kodierung	Wie werden Daten *kodiert*?	ASCII, UTF-8, EBCDIC
1	Transfer	Wir wird auf Daten *zugegriffen*?	HTTP, XMPP, SMTP

Tabelle 2.1.: Schichtenmodell von Kommunikation

Semantik Die Semantik ordnet den Wörtern Bedeutung zu.

Diese Merkmale dienen als Basis des in Tabelle 2.1 abgebildeten Schichtenmodells, welches im Folgenden erläutert wird. Die *Transferschicht* regelt, als unterste Schicht, die Art und Weise des Zugriffs auf Daten und deren Übertragung. Sie ermöglicht es somit Daten eines fremden Systems mittels einer wohldefinierten Schnittstelle auszulesen und zu modifizieren. Aufbauend auf dieser Möglichkeit spezifiziert die *Kodierungsschicht* die Menge der Zeichen aus denen sich die Daten zusammensetzen. Die Art und Weise wie diese Zeichen zusammengesetzt und somit Daten strukturiert werden spezifiziert die *Strukturschicht*. Ferner wird den einzelnen Wörtern auf der *Bedeutungsschicht* Semantik zugewiesen.

Optimalerweise sollten die im Rahmen dieses Modells gewählten Verfahren im Sinne eines strikten Schichtenmodells lediglich Implikationen bzgl. der Beschaffenheit niedrigerer Schichten vornehmen. Die damit erzielte Unabhängigkeit niedrigerer von höheren Schichten bedingt die vereinfachte Austauschbarkeit einzelner Spezifikationen und stärkt damit den modularen Aufbau der Kommunikation. Beispielsweise sollte die Art, wie Daten von einem System zu einem anderen übertragen werden, keine Bedingungen an die Struktur der Daten stellen.

2.3. Kopplung

Eng verbunden mit der Kommunikation von Softwaresystemen ist der Aspekte der *Kopplung*, welcher in dieser Arbeit eine zentrale Rolle spielt. Der Begriff Kopplung bezeichnet

hierbei ein Maß für die Verknüpfung von Softwaresystemen und dient zur Qualifizierung von Abhängigkeitsverhältnissen zwischen Systemen. Kopplung dient also der Bestimmung, in wie weit ein Softwaresystem oder eine -komponente von einer anderen abhängig ist. Grob unterscheiden lässt sich zwischen *enger* und *loser Kopplung*. Während einer engen Kopplung eine starke Bindung zwischen Systemen zugrundeliegt und dadurch ein hohes Maß an Spezialisierung bzgl. des verknüpften Systems ermöglicht wird, basiert eine lose Kopplung auf einer möglichst geringen Anzahl von Annahmen, die spezifisch für das verknüpfte System sind. Eine umfassende Analyse von Kopplung bietet [24].

2.4. Zusammenfassung

Ein zentrales Konzept das dieser Arbeit zur Erstellung eines lose gekoppelten Systems zugrundeliegt ist die separierte Umsetzung der verschiedenen Aspekte von Kommunikation. Zu diesem Zweck dient ein Schichtenmodell, das die Kommunikation in vier Ebenen unterteilt. Namentlich sind dies die Transfer-, die Kodierungs-, die Struktur- und die Bedeutungsschicht.

3. Representational State Transfer (REST)

3.1. Einleitung

Eine Anforderung an Softwaresysteme, die insbesondere aufgrund der schnell wachsenden Vernetzung durch das Internet an Bedeutung gewinnt und dessen Umsetzung eine der größten Herausforderungen in der praktischen Informatik ist, ist die lose Kopplung. Wenngleich es verschiedene Architekturen und Architekturstile gibt die sich diesem Problem annehmen, so sticht doch das World Wide Web (WWW) im besonderen Maße hervor. Das WWW ermöglicht es, eine immens große Menge verschiedener Informationen und Dienste auf lose Art und Weise miteinander zu verknüpfen.

Um diese Fähigkeit gezielt auf konkrete Architekturen und Softwaresysteme zu übertragen, stellt sich somit die Frage, was das WWW aus Sicht eines Softwarearchitekten ausmacht. Die Beantwortung dieser Frage hat Fielding seine Dissertation [14] gewidmet und stellt den architektonischen Kern des WWW in Form des Architekturstils *Representational State Transfer* (REST) heraus.

Im Folgenden wird dieser Architekturstil und die verbundenen Konzepte erläutert und in das in Abschnitt 2.2 beschriebene Schichtenmodell eingeordnet. Die Abschnitte 3.2 bis 3.4 orientieren sich dabei bewusst stark an [14] und greifen die für diese Arbeit bedeutenden Aspekte von REST auf. Darüber hinaus wird REST mit anderen Architekturen und Architekturstilen verglichen und von diesen abgegrenzt.

3.2. Definition

Der Begriff Representational State Transfer (REST) bezeichnet einen Softwarearchitekturstil für verteilte Hypermedia-Informationssysteme wie das World Wide Web (WWW) und hat seinen Ursprung in der Dissertation von Fielding (s.h. [14]).

Fielding, der durch seine Mitarbeit an Standards wie dem Hypertext Transfer Protocol (HTTP) (s.h. [13]) und dem Uniform Resource Identifier (URI) (s.h. [5]) das WWW

maßgeblich geprägt und zu seinem Erfolg wesentlich beigetragen hat, stellt in seiner Dissertation die Essenz dieses Erfolgs in Form des Architekturstils REST heraus. Er definiert REST im Rahmen seiner Arbeit wie folgt:

> The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application. [14, S. 86]

Demzufolge beschreibt REST die wesentlichen Eigenschaften der Architekturelemente eines verteilten Hypermedia-Informationssystems [1] mit Fokus auf den Merkmalen und der Interaktion der beteiligten Komponenten. Hierzu arbeitet er die unter architektonischen Gesichtspunkten wesentlichen Elemente und Konzepte des WWW heraus. REST stellt somit in gewissem Sinne eine Rückbesinnung auf die Grundlagen des WWW dar.

3.3. Elemente

3.3.1. Ressource

Ein grundlegendes Element von REST ist die *Ressource*. Der Begriff Ressource bezeichnet hierbei einen eindeutig identifizierbaren, real existierenden oder virtuellen Gegenstand. Die Identifizierung erfolgt mittels Ressourcenbezeichnern[2], welche es ermöglichen, sich auf Ressourcen zu beziehen und Aussagen über diese zu treffen. Dies bedingt, dass die Identität einer Ressource im Gegensatz zu ihrem Zustand unveränderlich ist.[3]

Zu beachten ist ebenfalls, dass einer Ressource nicht zwingend ein maschinenlesbarer Zustand hinterlegt sein muss. [4]

[1] Softwaresysteme oder konkrete Architekturen, die die von REST festgelegten Merkmale besitzen, werden auch als *RESTful* bezeichnet.

[2] REST schreibt nicht vor, wie ein solcher Ressourcenbezeichner aufgebaut ist. Typischerweise werden zur Beschreibung von Ressourcen jedoch URIs verwendet.

[3] Sei beispielsweise durch die URI „http://news.de/artikel/367“ eine Ressource die einen Nachrichtenartikel beinhaltet, identifiziert, so kann der Artikel über die Zeit modifiziert werden, um z.B. Korrekturen oder Aktualisierungen wiederzugeben. Die URI darf aber niemals einen gänzlich anderen Artikel „identifizieren“.

[4] Beispielsweise kann ein Dokument eine Ressource darstellen, welche durch die URI

3.3.2. Repräsentation

Komponenten einer REST-Architektur lesen den Zustand einer Ressource aus oder modifizieren diesen mittels Repräsentationen. Im Kontext dieser Arbeit stellt eine Repräsentation eine Menge von Daten dar, die den Zustand einer Ressource wiedergeben und dessen Form und Zusammensetzung wohldefiniert ist. Während der Zustand einer Ressource sich aus formlosen Informationen zusammensetzt, entspricht eine Repräsentation also einer maschinenlesbaren Darstellung dieses Zustandes. Ordnet man die Repräsentation in das Schichtenmodell aus Abschnitt 2.2 ein, so wird diese durch Spezifikationen der Schichten zwei bis vier definiert.

Die Unterscheidung von Ressource und Repräsentation ermöglicht es, alternative Repräsentationsformen für eine Ressource zur Verfügung zu stellen. Je nach Präferenz oder Fähigkeit kann ein Client daher eine von mehreren möglichen Darstellungsformen auswählen. Diese Option ist im Kontext lose gekoppelter Softwaresysteme von Bedeutung, da die Darstellung von Daten typischerweise eng mit der Verarbeitung und den dafür zugrundeliegenden Technologien zusammenhängt und somit zwischen verschiedenen Systemen variiert.

Eine weitere Betrachtungsweise des Unterschiedes von Ressource und Repräsentation ist die Trennung von Daten in abstrakte und konkrete Syntax. Die abstrakte Syntax beschreibt die Struktur von Daten unabhängig von einer Notation, die benutzt wird, um diese Daten zu repräsentieren. Die Art und Weise der Notation dieser Daten wird durch eine konkrete Syntax spezifiziert. Hierbei wird die abstrakte Syntax durch die Ressource und die konkrete Syntax durch eine Repräsentation dieser Ressource definiert.

An dieser Stelle sei ausdrücklich darauf hingewiesen, dass diese Definition eine Verallgemeinerung der Definition von Fielding ist, welcher Repräsentationen ausschließlich als eine „Sequenz von Bytes“ (vgl. [14, S. X]) betrachtet. Durch den Verzicht, Repräsentationen lediglich als Daten in serialisierter Form zu betrachten, ist es möglich, auch plattformabhängige Darstellungen zum Austausch eines Ressourcenzustandes zu nutzen.

Ein Mehrwert dieser Definition liegt darin, dass auch solche Softwarekomponenten,

„http://bsp.org/artikel.html“ identifiziert wird. Ferner kann aber auch eine Person durch die URI „urn:person:Schwartke,Hendrik“ identifiziert und somit als Ressource verstanden werden. Ein wesentlicher Unterschied zwischen diesen beiden Beispielen ist, dass der letzten Ressource kein maschinenlesbarer Zustand hinterlegt ist. Ihr Ressourcenbezeichner dient somit lediglich als Name und nicht als Adresse zur Ermittlung des Ressourcezustandes. Unabhängig davon ist es jedoch möglich, sich auf diese Ressource zu beziehen, um Aussagen über die entsprechende Person zu machen. Beide URIs identifizieren also Ressourcen im Sinne von REST.

die auf derselben Plattform ausgeführt werden, einfach und REST-konform interagieren können. Der u.U. komplexe Vorgang der Serialisierung und Deserialisierung kann durch den direkten Datenaustausch in Form einer plattformabhängigen Darstellung entfallen. [5]

3.3.3. Konnektor

Zur Übertragung von Repräsentationen ist des Weiteren ein Protokoll notwendig, welches die Schnittstelle der Anwendung (s.h. Abschnitt 3.4.4) über die Systemgrenzen hinweg verfügbar macht und es in diesem Zuge ermöglicht, Repräsentationen zwischen Systemen auszutauschen. Zu diesem Zweck muss eine Abbildung der Funktionen der Schnittstelle auf die Spezifika des ausgewählten Protokolls erstellt werden. Der Teil der Software der diese Abbildung implementiert, wird im Kontext eines REST-konformen Softwaresystems als Konnektor bezeichnet. Ein solcher setzt also die Aspekte um, die mit der in Abschnitt 2.2 beschriebenen Transferschicht verbunden sind.

Durch den Einsatz mehrerer Konnektoren kann der Zugriff auf die Schnittstelle auch mittels verschiedener Protokolle realisiert werden. Dies hat den Vorteil, dass je nach Anforderung der an der Kommunikation teilnehmenden Systeme das zweckmäßigste Protokoll genutzt werden kann. Ein weiterer Vorteil besteht darin, Protokolle, die bereits im Anwendungsumfeld eingesetzt werden, weiterhin nutzen zu können.

3.4. Konzepte

3.4.1. Client-Server-Paradigma

REST folgt dem Client-Server-Paradigma, welches die Rollenverteilung zweier interagierender Softwarekomponenten, dem Client und dem Server, beschreibt. Der Server ist diejenige Komponente, welche Dienste oder Daten anbietet und der Client jene, die diese nutzt und eine Interaktion mit dem Server initiiert. Client und Server können hierbei auf unterschiedlichen Hardwareplattformen installiert sein und über ein Netzwerk kommunizieren. Das Kommunikationsverhalten ist nach [4] dadurch gekennzeichnet, dass der Client eine Anfrage an den Server sendet und dieser darauf eine entsprechende Antwort zurückschickt.

[5] So ist beispielsweise der Austausch eines im Hauptspeicher gehaltenen Objektes zwischen zwei auf der gleichen Programmiersprache basierenden Softwarekomponenten, welche auf derselben Plattform ausgeführt werden, ohne kostspielige Serialisierung bzw. Deserialisierung des Objektes möglich.

Die Interaktion kann sowohl synchron als auch asynchron erfolgen. Es wird also keine Aussage darüber getroffen, ob der Client zwischen Anfragestellung und Antworterhalt blockiert. [6]

3.4.2. Statuslosigkeit

Eine wesentliche Eigenschaft von REST besteht darin, dass die Interaktion zwischen Client und Server als zustandslos für den Server entworfen wird. Der Sitzungszustand, also insbesondere alle Informationen bzgl. der Interaktion die über eine einzelne Anfrage hinausgehen, liegen ausschließlich auf dem Client vor. Aus Sicht des Servers besteht die Interaktion mit dem Client daher aus einer Folge voneinander unabhängigen Anfragen. Somit ist es dem Server möglich, alle für die Beantwortung einer Anfrage notwendigen Systemressourcen nach der Abarbeitung der Anfrage freizugeben.

Da der Server entsprechende Systemressourcen nicht über eine Anfrage hinaus vorhalten und verwalten muss, ergeben sich für diesen u.U. erhebliche Performancevorteile. Des Weiteren ist eine feste Bindung eines Clients an einen Server über die gesamte Sitzungsdauer nicht notwendig. Grundsätzlich ist es möglich, dass mehrere infolge gestellte Anfragen eines Clients von unterschiedlichen Servern beantwortet werden, womit wesentlich zur Skalierbarkeit der Anwendung beigetragen wird. [7]

Ein weiterer Vorteil besteht in der einfacheren Überwachung solcher Serversysteme. Eine Überwachungskomponente ist prinzipiell imstande, jede Anfrage unabhängig von vorangegangenen oder folgenden Anfragen zu beurteilen. Sie kann also ebenfalls statuslos sein.

3.4.3. Zwischenspeicherbarkeit

Eine elementare Eigenschaft des WWW zur effizienten Anfragebeantwortung, welche von REST aufgegriffen wird, ist die Möglichkeit, Repräsentationen zwischenzuspeichern und bei späteren Anfragen wiederzuverwenden. Dieses Vorgehen kann je nach konkreter Umsetzung erhebliche Performancevorteile erzielen, ist jedoch auch mit einer Reihe von

[6] Einige Autoren gehen davon aus, dass sich REST auf ein blockierendes Aufrufverhalten beschränkt, also eine asynchrone Kommunikation nicht ermöglicht. Dies mag darin begründet liegen, dass REST häufig im Zusammenhang mit HTTP, welches ein ausschließlich synchrones Netzwerkprotokoll ist, eingesetzt wird. Grundsätzlich ist es aber auch möglich, eine asynchron agierende Software REST-konform zu gestalten.

[7] Ein weiterer, nicht zu unterschätzender Vorteil ist die u.U. erheblich vereinfachte Möglichkeit, den Server während des Entwicklungsprozesses zu testen, da der Server bzgl. der Interaktion nicht erst in einen speziellen Zustand gebracht werden muss, bevor der Test durchgeführt werden kann.

Problemen behaftet. Es stellt sich insbesondere die Frage, wie und wann der Zwischenspeicher mit den Originaldaten synchronisiert wird. Die zu erzielenden Vorteile und die notwendigen Schritte zur Datensynchronisation hängen dabei wesentlich von der Platzierung des Zwischenspeichers ab.

So kann die Integration eines Zwischenspeichers in den Client zu einer signifikanten Steigerung der Antwortzeiten und damit des Echtzeitverhaltens des Clients führen, da dieser Repräsentationen lokal zwischenspeichert und bei Wiederholung einer entsprechenden Anfrage wieder verwenden kann. Der Nachteil dieses Ansatzes ist die Notwendigkeit, u.U. komplexe Protokolle zur Synchronisation zwischen Client und Server einsetzen zu müssen. [8]

Einen alternativen Ansatz stellt die Integration eines Zwischenspeichers in den Server dar. Die Nutzung eines solchen Zwischenspeichers reduziert weder Datendurchsatz noch Anfragehäufigkeit. Da aber die Abarbeitung einer Anfrage oftmals mit großem Rechenaufwand und erheblicher Speichernutzung verbunden ist, kann eine Wiederverwendung von Antworten einen erheblichen Performancezuwachs bedeuten. Zwar ist auch in diesem Fall eine Synchronisierung des Zwischenspeichers notwendig, dieser Vorgang kann jedoch unabhängig vom Client und für diesen vollständig transparent durchgeführt werden. Somit entfällt die Notwendigkeit, entsprechende Vorgehensweisen clientseitig zu implementieren, was insbesondere einer losen Kopplung der Systeme zugutekommt.

Eine weiterer Möglichkeit, Zwischenspeicher zu verwenden, besteht darin, eine entsprechende Komponente in den Kommunikationsweg zu integrieren. Dies kann sowohl für den Client als auch für den Server transparent geschehen.

Notwendig für die Nutzung eines Zwischenspeichers ist die Festlegung, welche Daten zwischengespeichert werden dürfen. [9] Eine wesentliche Anforderung an die Architektur ist daher die Möglichkeit, eben diese Unterscheidung durchzuführen.

3.4.4. Generische Schnittstelle

Der wohl bedeutendste Unterschied zwischen REST und anderen Architekturstilen ist der Einsatz einer einheitlichen Schnittstelle für alle Funktionalitäten, die ein Server zur

[8] Ein Beispiel für einen solchen Ansatz ist das Hypertext Transfer Protocol (HTTP), welches ausgefeilte Verfahren zur Verwaltung des clientseitigen Zwischenspeichers implementiert.

[9] Beispielsweise ist es i.A. möglich und sinnvoll, das Ergebnis einer komplexen Suchanfrage zwischenzuspeichern und somit bei einer identischen Anfrage wiederzuverwenden, ohne die eigentliche Suche erneut durchzuführen. Die Antwort auf eine eingehende Warenbestellung unter Umgehung der Fachlogik wiederzuverwenden, ist jedoch nicht sinnvoll.

Verfügung stellt. [10] Dies geschieht durch Abstraktion der Funktionen des Servers und hat zum Ziel, eine Schnittstelle zu schaffen, die speziell auf die Bedürfnisse von lose gekoppelten Systemen ausgerichtet ist, um sowohl die Gesamtarchitektur des Softwaresystems als auch die Interaktion zwischen den verteilten Softwarekomponenten zu vereinfachen. Zu diesem Zweck wird eine Reihe von Anforderungen an diese Schnittstelle gestellt. So muss eine solche Schnittstelle zum einen die vier folgenden Merkmale aufweisen:

Identifizierbarkeit Ressourcen müssen eindeutig identifizierbar sein. [11]

Modifizierbarkeit Die Modifikation einer Ressource erfolgt anhand ihrer Repräsentationen.

Selbstbeschreibend Die Bedeutung einer Ressource muss vollständig durch sich selbst, d.h. ohne die Auswertung anderer Ressourcen ermittelbar sein. [12]

Hypermedia-getrieben Hypermedia wird zur Verknüpfung von Ressourcen genutzt und dient als primärer Mechanismus zur Zustandsänderung des Clients. [13]

Ferner muss die Schnittstelle eine Funktion bereitstellen, mittels derer Ressourcen anhand ihres Bezeichners und in Form einer Repräsentation ausgelesen werden können, ohne den Zustand des Servers und insbesondere den der auszulesenden Ressource zu verändern. Eine solche Funktion wird im Zusammenhang mit REST als *sicher* bezeichnet. [14]

Im Zusammenhang mit der eindeutigen Identifizierbarkeit jeder Ressource mittels eines Ressourcenbezeichners ergibt sich die Möglichkeit Ressourcen zu referenzieren und somit Ressourcen untereinander zu verknüpfen. Insbesondere kann der Server dem Client durch

[10] Die wohl am weitesten verbreitete Fehlannahme über REST ist, dass eine allgemein für alle REST-Architekturen „genormte“ Schnittstelle existiert. Insbesondere werden häufig die HTTP-Methoden GET, POST, PUT und DELETE als eben diese allgemeingültige Schnittstelle betrachtet. HTTP ist jedoch lediglich ein mögliches Protokoll zur Anbindung an eine REST-konforme Anwendung. Wenngleich REST einige Anforderungen an diese Schnittstelle stellt, die von allen REST-konformen Architekturen erfüllt werden müssen, so hängt die konkrete Beschaffenheit einer solchen Schnittstelle ganz wesentlich von den Anforderungen an die konkrete Anwendung ab.

[11] Obgleich die Art und Weise wie diese Identifizierung umgesetzt werden soll von REST nicht vorgegeben ist, so werden zu diesem Zweck typischerweise URIs verwendet.

[12] Alternativ kann man die Semantik einer Ressource auch als *kontextfrei* bezeichnen.

[13] Dieses Merkmal wird von Fielding als „hypermedia as the engine of application state“ (HATEOAS) bezeichnet.

[14] Der Begriff *sicher* bezeichnet also die Zusicherung des Servers, dass der Aufruf einer solchen Funktion frei von Seiteneffekten ist. Eine sichere Funktion kann somit jederzeit bedenkenlos aufgerufen werden. Abzugrenzen ist der Begriff insbesondere von dem der sicheren Datenübertragung, z.B. mittels Kryptografie.

solche Verweise weitere Informationen und Dienste, welche im Zusammenhang mit der jeweiligen Ressource zweckmäßig sind, bekannt geben. Auch ist es möglich, solche Verknüpfungen über Systemgrenzen hinweg zu nutzen, womit sich komplexe, lose gekoppelte und skalierbare verteilte Systeme erstellen lassen. [15]

Ein solches System lässt sich im Ganzen oder in Teilen mittels sicherer Funktionen erkunden, indem ausgehend von einer Ressource, die als Einstiegspunkt dient, sukzessive anhand von Referenzen weitere Ressourcen ausgelesen werden. [16]

3.5. Abgrenzung

3.5.1. Remote Procedure Call (RPC)

Einer der ältesten Ansätze zur Entwicklung verteilter Systeme sind *Remote Procedure Calls* (RPC). Das Grundkonzept hinter RPC ist der Aufruf von Prozeduren über Systemgrenzen hinweg. Im Zuge dessen werden Aufrufe entfernter Prozeduren aus Sicht des Entwicklers der Fachlogik möglichst transparent durchgeführt, d.h. ohne dessen Wissen um die Verteilung und über die tatsächliche Lokalität der entsprechenden Implementierung. Dadurch ist es theoretisch möglich, die Verteilung der Prozeduren unabhängig von ihrer Implementierung und Nutzung zu gestalten. Insbesondere ist eine nachträgliche Verteilung der Prozeduren einer Anwendung und die Änderung der Verteilung prinzipiell möglich.

Mit der für den Entwickler transparenten Nutzung von entfernten Prozeduren gehen aber auch eine Reihe von Problemen einher, da sich einige Merkmale der lokalen Nutzung von Prozeduren nicht direkt auf verteilte Systeme überführen lassen. So kann ein Aufruf einer entfernten Prozedur – samt der notwendigen Kodierung und Dekodierung von Parameter- und Rückgabewerten und der eigentlichen Übertragung der Daten über ein Netzwerk um viele Größenordnungen – langsamer als ein lokaler Aufruf sein. Insbesondere sind Prozeduren häufig feingranular, d.h. sie führen anhand einer kleinen Menge von

[15] Dieses Merkmal von REST-konformen Systemen setzt zwar voraus, dass alle beteiligten Systeme kompatible Ressourcenbezeichner nutzen, da zu diesem Zweck ganz überwiegend URIs eingesetzt werden, stellt dies jedoch praktisch keine Einschränkung dar.

[16] Der Nutzen dieses Ansatzes ist im WWW gut erkennbar. Ein Benutzer, welcher eine Webseite geladen hat, kann mittels Verweisen weitere mit der Seite verbundene Informationen auffinden. Auch wird dieser Ansatz zur automatischen Verarbeitung von Webseiten genutzt. So nutzen Webcrawler die Verknüpfung um das Web sukzessive zu durchsuchen und relevante Informationen, z.B. für Suchmaschinen, ausfindig zu machen. Der Nutzen dieses Vorgehens wird allerdings durch eine nicht maschinenauswertbare Darstellung von Informationen, z.B. in natürlich sprachlicher Form, begrenzt.

Eingabedaten einen Arbeitsschritt mit geringem Umfang aus. Zur Realisierung komplexer Anwendungen mittels feingranularer Prozeduren ist daher typischerweise eine große Anzahl an Prozeduraufrufen notwendig. Obwohl dieses Vorgehen durchaus ein sinnvolles Mittel zur Strukturierung nicht verteilter Systeme ist, führt es im Kontext verteilter Systeme wegen der resultierenden hochfrequenten Interaktion der Systeme nicht selten zu einem erheblichen Performanceverlust.

Im Gegensatz hierzu stellen Ressourcen im Sinne von REST grobgranulare Informationseinheiten dar. Diese sind ferner selbstbeschreibend (vgl. Abschnitt 3.4.4), enthalten also alle Informationen zur vollständigen Beschreibung des mit der Ressource dargestellten Sachverhaltes. Hierdurch ist es möglich, grobgranulare Funktionen zu realisieren, welche einen komplexen Arbeitsschritt auf Basis eines einzelnen Aufrufs ausführen und somit typischerweise wesentlich seltener aufgerufen werden müssen. Hierdurch kann die insgesamt zur Datenübertragung benötigte Zeit reduziert und sowohl die Performance als auch die Skalierbarkeit des Gesamtsystems erhöht werden.

Eine weitere Eigenschaft von Prozeduren ist ihre enge Kopplung an den Aufrufer. Diesem muss eine große Menge an Details bzgl. der jeweiligen Prozedur, wie z.B. die Anzahl, Art und Bedeutung der möglichen Parameter, ebenso wie der Datentypen eines Rückgabewertes, bekannt sein. REST nutzt hingegen eine einheitliche Schnittstelle, um die Anzahl der technischen Details, die dem Client zur Interaktion bekannt sein müssen zu minimieren. Des Weiteren wird die Beschaffenheit von Ressourcen i.A. nicht durch einen Datentypen beschrieben. Lädt ein Client eine Repräsentation einer Ressource, so hat dieser, im Gegensatz zu einem RPC-basierten System, a priori kein Wissen über den konkreten Inhalt der Repräsentation. Er reagiert vielmehr im Rahmen der von ihm zu lösenden Aufgabe auf die erhaltenen Daten. Der Zustand des Clients wird also primär durch die vom Server erhaltenen Informationen und die vom Client genutzten Referenzen zu anderen Ressourcen bestimmt (vgl. Abschnitt 3.4.4).

Ein weiteres Problem im Zusammenhang mit entfernten Prozeduraufrufen stellt die Verarbeitung von Zeiger- und Referenzsemantik von Prozedurparametern und -rückgabewerten dar. RPC-basierende Systeme verfügen typischerweise nicht über einen gemeinsamen Speicher, so dass eine solche Übergabeart nicht ohne weiteres umzusetzen ist. Nicht wenige RPC-Implementierungen verzichten daher auf die Möglichkeit, Verweise auf Daten auf einheitliche Weise abbilden zu können und erschweren somit die Verknüpfung von Informationen.

Im Kontext von REST ist dies hingegen, in Form der Referenzierung von Ressourcen, eines der zentralen Konzepte (vgl. Abschnitt 3.2). Durch URIs steht hierzu ein zweck-

mäßiger und standardisierter Ansatz zur Verfügung.[17]

Ebenfalls können in verteilten Systemen Fehlerzustände entstehen, die in nicht verteilten Systemen nicht auftreten, wie z.B. Übertragungsfehler oder Ausfälle einzelner Komponenten. Durch die angestrebte Verteilungstransparenz RPC-basierter Systeme ist der Entwickler nur sehr begrenzt imstande, entsprechende Fehlerquellen zu erkennen und sinnvolle Fehlerbehandlungsroutinen zu implementieren. Daraus resultiert, dass Teilausfälle in RPC-basierten Systemen nicht selten die Funktionstüchtigkeit des Gesamtsystems unterbinden. Wenngleich diese Fehlerarten ebenso in einem REST-konformen System existieren, so bedingt REST jedoch keine Verteilungstransparenz und gibt damit dem Entwickler die Möglichkeit, entfernte Aufrufe als solche zu erkennen und entsprechende Fehlerfälle zu berücksichtigen.

Ein weiteres Problem stellt die mangelnde Interoperabilität vieler RPC-basierter Systeme dar. Diese wird dadurch bedingt, dass die transparente Nutzung von entfernten Prozeduraufrufen durch Middlewareprodukte umgesetzt wird, welche typischerweise keine Anpassung der zugrundeliegenden Protokolle zur Kommunikation und Datenrepräsentation ermöglichen. Somit können nur Softwarekomponenten, welche auf identischer Middleware basieren, direkt interagieren. Durch diesen Umstand wird nicht selten die Menge der für die Implementierung der Fachlogik zur Auswahl stehenden Technologien – wie Programmiersprachen oder Softwareplattformen – stark eingeschränkt.

REST hingegen unterstützt die Nutzung verschiedener Darstellungsformen (vgl. Abschnitt 3.3.2) explizit. Ferner müssen die mit einer systemübergreifenden Kommunikation verbundenen Einschränkungen nicht vor dem Entwickler verborgen werden, so dass dieser sinnvolle Maßnahmen für den Fehlerfall ergreifen kann.

3.5.2. Web Services

Eine weitere, recht populäre Möglichkeit zur Realisierung verteilter Systeme sind *Web Services*. REST wird in der Literatur nicht selten als leichtgewichtige Alternative zu diesen betrachtet. [18] Die dieser Betrachtung zugrundeliegenden Definition des Begriffs Web Service entstammt dem World Wide Web Consortium (W3C), welches für einen Grossteil der im Rahmen von Web Services eingesetzten Protokolle verantwortlich ist.

[17] In diesem Zusammenhang kann die Menge aller Ressourcen auch als Inhalt eines gemeinsamen Speichers, der mittels URIs adressiert wird, interpretiert werden.

[18] Da der Begriff Web Service aber zum Teil mit erheblich abweichender Bedeutung belegt und nicht selten auch der Begriff REST nicht in dem von [14] spezifizierten Sinne interpretiert wird, sind die geführten Argumentationen zum Teil schwer nachvollziehbar und somit schlecht vergleichbar.

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [22]

Die Kommunikation mit einem Web Service geschieht folglich durch auf SOAP (s.h. [19]) basierenden Nachrichten, welche typischerweise mittels HTTP übertragen werden. SOAP kann hierbei auf zwei verschiedene Arten genutzt werden: Zum einen ist es möglich, entfernte Prozeduraufrufe mittels SOAP zu formulieren und zu übertragen. Die damit einhergehenden Probleme sind prinzipiell identisch mit denen anderer RPC-basierter Systeme (vgl. Abschnitt 3.5.1). Zum anderen ist es mittels SOAP möglich, Daten in Form von XML direkt an einen Dienst zu übertragen.

Obgleich diese Art der Datenübertragung Ähnlichkeiten mit der Übermittlung von Repräsentationen im Sinne von REST aufweist, so gibt es doch eine Reihe wesentlicher Unterschiede. Einer dieser Unterschiede ist, dass eine REST-konforme Anwendung eine einheitliche Schnittstelle zum Zugriff auf Daten und zu deren Modifikation bereitstellt. Dem Client wird eine von der konkreten Art der Information unabhängige Schnittstelle und ein zugehöriges Transferprotokoll, im Sinne des in Abschnitt 2.2 erläuterten Kommunikationsmodells, zur Verfügung gestellt. Dies erleichtert den Datenzugriff, da somit keine syntaktischen oder semantischen Eigenschaften der Informationen zum eigentlichen Abruf bekannt sein müssen. Die Nutzung einer nicht einheitlichen Schnittstelle führt bei steigender Anzahl von Diensten auch zu einer steigenden Anzahl von Schnittstellen, die dem Client bekannt gemacht werden müssen. Um den damit verbundenen wiederkehrenden Implementierungsaufwand möglichst gering zu halten, werden die Schnittstellen mittels der Web Service Description Language (WSDL) (s.h. [11]) formal spezifiziert. Somit ist es einem Client theoretisch möglich, eine solche Schnittstellenbeschreibung zur Laufzeit zu laden, auszuwerten und die beschriebenen Funktionen im Anschluss zu nutzen. In der Praxis hat sich ein solches Verfahren zur Anpassung der Software zur Laufzeit aus verschiedenen Gründen aber nicht durchgesetzt. So bedarf es zur Nutzung von Diensten der Erstellung von technischem Quellcode, sogenannten Client-Stubs, basierend auf WSDL-Beschreibungen. Des Weiteren ist es mittels WSDL theoretisch möglich, plattformunabhängig Datenstrukturen und Schnittstellen zu beschreiben, und somit – unabhängig von der zur Implementierung verwendeten Technologie – interoperabele Systeme zu schaffen. In der Praxis stellt sich dies jedoch nicht selten als problematisch heraus.

Da verschiedene Softwarewerkzeuge oftmals unterschiedliche Teilmengen der zugrundeliegenden Spezifikationen implementieren, sind diese trotz der Nutzung von Standards teilweise inkompatibel.

Im Gegensatz dazu ist die Einbindung neuer Dienste in einen REST-konformen Client durch Referenzen auf entsprechende Ressourcen erheblich einfacher. [19]

Ein weiterer Schwachpunkt von Web Services bzgl. der Entwicklung interoperabler Software ist die z.T. nicht spezifikationsgerechte Nutzung von HTTP. So weist die Spezifikation bzgl. der Übertragung von SOAP mittels HTTP einige Fehlkonzeptionen auf. SOAP nutzt, sofern es mithilfe HTTP genutzt wird, i.d.R. die HTTP-Methode `POST`, um Funktionsaufrufe zu übertragen. Die Nutzung dieser Methode in Zusammenhang mit sicheren oder idempotenten Funktionsaufrufen verstößt jedoch gegen die HTTP-Spezifikation und stört damit u.U. Infrastrukturkomponenten, wie z.B. Zwischenspeicher. Diese Fehlkonzeptionen werden von einer Reihe von Autoren (vgl. [35]) als REST-Anti-Pattern bezeichnet. [20]

3.6. Beispiel

Die in diesem Kapitel vorgestellten Grundlagen und dessen Zusammenspiel werden im Folgenden beispielhaft an einem fiktiven REST-konformen Onlineshop verdeutlicht. Zu diesem Zweck wird die Kommunikation zwischen einem Client und den beteiligten Serversystemen aufgeführt und erläutert.

Die Interaktion des Onlineshops lässt sich gemäß des in Abschnitt 2.2 erläuterten Kommunikationsmodells wie folgt beschreiben: Auf der Transferschicht wird HTTP genutzt, das in diesem Beispiel nur auszugsweise und vereinfacht dargestellt wird. Insbesondere werden Verfahren zur Authentifikation des Benutzers nicht aufgeführt. Auf der Kodierungsschicht wird UTF-8 (s.h. [36]) verwendet. Ferner wird auf der Strukturschicht, mit einigen Ausnahmen, XML eingesetzt. Des Weiteren werden zur Beschreibung der Bedeutung von Ressourcen drei verschiedene Sprachspezifikationen verwendet. Zum einen wird die XML Linking Language (s.h. [12]) des W3C genutzt, um Ressourcen untereinander

[19] Hierbei bleibt natürlich unberührt, dass dem Client und dem Server gemeinsame Protokolle bzgl. des in Abschnitt 2.2 erläuterten Kommunikationsmodell zugrundeliegen müssen. Ist dem Client beispielsweise die Bedeutung des Vokabulars, welches der Server nutzt, nicht bekannt, so kann dieser den Dienst nicht in vollem Umfang nutzen.

[20] Die Bezeichnung als REST-Anti-Pattern ist allerdings irreführend, da keine fehlerhafte Anwendung des REST-Architekturstils zugrundeliegt, sondern vielmehr die Verletzung einer Protokollspezifikation, hier HTTP.

zu verknüpfen. Der zugehörige Namensraum ist `http://www.w3.org/1999/xlink`. Darüber hinaus wird eine fiktive Sprache zur Beschreibung von Onlineshops genutzt, die ein spezifisches Vokabular umfasst, um typische Sachverhalte im Zusammenhang mit dem Handel von Waren zu versprachlichen. Die Elemente dieser Sprache sind im Namensraum `urn:businessDescriptionLanguage` enthalten. Schließlich werden auch Sprachelemente eingesetzt, welche spezifisch für den beschriebenen Onlineshop sind. Diese sind im Namensraum `http://shop.de/` zusammengefasst. Die genaue Semantik der für dieses Beispiel bedeutenden Elemente wird jeweils an der dafür sinnvollen Stelle im Laufe des Beispiels erläutert.

In diesem Beispiel ist der erste Schritt des Clients der Aufruf der Ressource, welche den Index des Onlineshops beinhaltet. Es wird davon ausgegangen, dass diesem die URI dieser Ressource, hier `http://shop.de/`, bereits bekannt ist. Mittels der sicheren HTTP-Methode `GET` lädt dieser eine Repräsentation der Ressource und spezifiziert ferner mittels des HTTP-Parameters `Accept`, welche Darstellungsart dieser verarbeiten kann.

```
1 GET http://shop.de/
2 Accept: text/xml
```

Der Server antwortet nun, wie vom Client gewünscht mit einer Repräsentation der Indexressource im Format XML.

```
3  <?xml version="1.0" encoding="UTF-8"?>
4  <index  xml:base="http://shop.de/"
5          xmlns="urn:businessDescriptionLanguage"
6          xmlns:s="http://shop.de/"
7          xmlns:xlink="http://www.w3.org/1999/xlink">
8    <meinKonto xlink:href="/users/323"/>
9    <produktsuche xlink:href="/search"/>
10   <bestellungsAnnahme xlink:href="/order"/>
11   <reklamationsAnnahme
12         xlink:href="http://complaints.shop.de/service"/>
13   <s:neueArtikel xlink:href="/new"/>
14   <s:sonderangebot xlink:href="/products/p5532">
15   <s:sonderangebot xlink:href="/products/p1356">
16   <s:partnerSeite xlink:href="http://otherShop.de/">
17   <s:partnerSeite xlink:href="http://yetanotherShop.de/">
18 </index>
```

Die empfangene Repräsentation nutzt die XML Linking Language, um Ressourcen zu referenzieren. Somit ist es einem Client, dem dieser Standard bekannt ist, möglich, weitere Ressourcen aufzufinden. Hierzu ist er insbesondere auch dann in der Lage, wenn diesem die anderen Sprachelemente nicht bekannt sind. Somit ist es auch einem solchem Client möglich, die Ressourcen, die der Onlineshop zur Verfügung stellt, aufzufinden. Diese Möglichkeit kann z.B. von Suchmaschinen genutzt werden, die die Ressourcen auslesen und für eine spätere Suche aufbereiten. Auch kann ein Client, der z.B. nur gezielt nach Artikeln sucht und dem daher, neben der XML Linking Language, nur die Sprachelemente, die einen Artikel beschreiben bekannt sind, das Onlineangebot zielgerichtet nach solchen durchsuchen.

Darüber hinaus werden Elemente der fiktiven Beschreibungssprache für Onlineshops verwendet, mithilfe derer typische Merkmale des Onlineshops ausgedrückt werden. Beispielweise stellt diese Sprache ein Element mit dem Namen `produktsuche` zur Verfügung, um die für die Produktsuche zuständige Ressource zu identifizieren und ein Element namens `bestellungsAnnahme`, das die Ressource zur Annahme von Bestellungen identifiziert. Ein Client, der mit diesem Standard vertraut ist, kann somit auch ohne das Wissen um Elemente, die spezifisch für den konkreten Onlineshop sind, zumindest einen Teil der bereitgestellten Informationen und Funktionen nutzen.

Ferner werden Merkmale des Onlineshops, die nicht durch andere Spezifikationen abgedeckt sind, mittels eines weiteren Vokabulars ausgedrückt. Ein Beispiel für ein solches Sprachelement ist das Element `sonderangebot`. Ein solches Merkmal kann nur von Clientsystemen, die explizit für diesen Onlineshop entwickelt wurden, interpretiert werden. Dennoch ist es auch einem Client, der diese Voraussetzung nicht erfüllt, möglich, den als Sonderangebot ausgezeichneten Artikel zu finden, da das entsprechende Sprachmerkmal wiederum die XML Linking Language zur Referenzierung einsetzt. Lediglich die Bedeutung, die dem Artikel durch die Auszeichnung als Sonderangebot zugewiesen wird, bleibt einem solchen Client verborgen.

Nun lädt der Client die Ressource mit der URI `http://shop.de/products/p5532`, welche in der Indexressource aufgeführt ist.

```
19 GET http://shop.de/products/p5532
20 Accept: text/xml
```

Hierbei ist dem Client, wie bei allen REST-konformen Operationen, a priori nicht bekannt, wie die angeforderte Ressource konkret beschaffen ist, also ob die Ressource beispielweise einen Artikel oder eine Warenbestellung beschreibt. Zwischen Client und Server

existieren keinerlei Vereinbarungen welcher „Art“ Ressourcen sind. Dies verdeutlicht noch einmal einen wesentlichen Unterschied zwischen REST-konformen und RPC-basierten Systemen, wie sie in Abschnitt 3.5.1 erläutert sind.

Im Zusammenhang mit einer RPC-basierten Anwendung, als auch mit einem typischen Web Service, würde der Client die untenstehenden Informationen beispielsweise durch die Funktion `getArtikel` ermitteln. Die Spezifikation der Serverschnittstelle, in welcher diese Funktion beschrieben wäre, würde u.A. auch den Rückgabetypen der Funktion angeben. Der Client würde also die empfangenen Daten in eine interne Darstellungsform eben dieses Datentyps überführen. Dieses Vorgehen setzt jedoch eine große Menge Wissen des Clients über den Server voraus, wie z.B. über die vom Server bereitgestellten Funktionen oder deren Merkmale. Ein solcher Ansatz würde letztendlich zu einer engen Kopplung der Systeme führen.

Der REST-konforme Client dieses Beispiels macht jedoch keine Annahmen über die Art der Ressourcen, sondern reagiert auf die vom Server übermittelten Repräsentationen. Unter der Annahme, dass es sich bei dem Client um ein System handelt, dessen Aufgabe es ist, automatisch Artikelinformationen zu finden und zu vergleichen, so würde der Client nach Erhalt einer Repräsentation eines Artikels den entsprechenden Algorithmus zur Verarbeitung aufrufen. Im Sinne von REST ist es jedoch auch denkbar, dass der Server in dieser Situation nicht eine Artikelbeschreibung, sondern eine anders geartete Ressource, wie z.B. eine Benachrichtigung, dass der Artikel bereits ausverkauft ist, zurücksendet. In diesem Fall würde der Client entsprechend anders auf die Situation reagieren, z.B. indem er die Antwort des Server nicht weiter auswertet und stattdessen anhand anderer Referenzen auf Ressourcen des Onlineshops die Suche nach Artikeln fortsetzt.

Der Server antwortet nun mit einer entsprechenden Repräsentation der angeforderten Ressource.

```
21 <?xml version="1.0" encoding="UTF-8"?>
22 <artikel xml:base="http://shop.de/"
23         xmlns="urn:businessDescriptionLanguage"
24         xmlns:s="http://shop.de/"
25         xmlns:xlink="http://www.w3.org/1999/xlink">
26   <artikelnr>p5532</artikelnr>
27   <bezeichnung>WD7500BPVT</bezeichnung>
28   <beschreibung>
29         Die 2,5"-Festplatte WD7500BPVT
30         von Western Digital bietet ...
```

```
31    </beschreibung>
32    <abbildung
33          xmlns:xlink="http://www.wdc.com/img/WD7500BPVT_1.jpg"/>
34    <hersteller xmlns:xlink="http://www.wdc.com/aboutUs.xml"/>
35    <preis waehrung="Euro">250</preis>
36    <s:zubehoer
37          xlink:href="/products/p2267"
38          xlink:title="Serial ATA Datenkabel"/>
39    <s:zubehoer
40          xlink:href="/products/p6993"
41          xlink:title="Wechselrahmen"/>
42    <s:datenblatt
43       xmlns:xlink="http://www.wdc.com/products/WD7500BPVT.pdf"/>
44    <s:kapazitaet einheit="GB">750</kapazitaet>
45    <s:zugriffszeit einheit="ms">12</zugriffszeit>
46  </artikel>
```

Die Repräsentation des Artikels enthält zum einen für Artikelbeschreibungen typische Elemente, wie z.B. Artikelnummer, Beschreibung und Preis, welche mittels der fiktiven Beschreibungssprache für Onlineshops spezifiziert werden. Daneben werden aber auch einige speziellere Merkmale des Artikels, welcher in diesem Fall eine Festplatte ist, beschrieben. Beispiele hierfür sind die Kapazität und die Zugriffszeit der Festplatte. Um diese Artikeleigenschaften zu beschreiben, wird auf Sprachelemente zurückgegriffen, die spezifisch für diesen Onlineshop sind. Weitere Beispiele für solche Elemente sind `zubehoer` und `datenblatt`, welche ihrerseits wiederum die XML Linking Language nutzen, um weitere Ressourcen zu referenzieren.

Im Anschluss hieran versucht der Client eine Repräsentation des Datenblattes zu laden, dessen URI er dem Element `datenblatt` entnimmt. Es ist anzumerken, dass diese Ressource nicht vom Onlineshop, sondern vom Festplattenhersteller bereitgestellt wird und somit eine lose Kopplung der Serversysteme vorliegt. Da beide Diensteanbieter dem Client eine ihm bekannte Schnittstelle zum Transfer von Repräsentationen bereitstellen – hier jeweils HTTP – ist diesem eine Interaktion mit beiden Systemen grundsätzlich möglich. Anders ausgedrückt, sind Client und Server bzgl. der Transferschicht kompatibel. Dies verdeutlicht, dass durch die sinnvolle Nutzung von Standards der konkrete Dienst des Festplattenherstellers bei der Entwicklung des Clients nicht berücksichtigt

werden muss, um einen Transfer von Daten, welche dem Client nützlich erscheinen, zu ermöglichen.

Hiermit wird ein wesentlicher Unterschied zu typischen, auf WSDL basierenden Web Services hervorgehoben. Die Interaktion mit einem System, das durch WSDL beschrieben wird, ist typischerweise nur nach Erstellung von Quellcode, der auf der WSDL-Beschreibung basiert, und somit nach erneuter Übersetzung der Anwendung möglich. Eine entsprechende Anpassung zur Laufzeit ist i.d.R. nicht möglich. Eine solche Anpassung der Anwendung bedarf daher sowohl eines Entwicklers, der diese Arbeit durchführt, als auch des Zugriffs auf den Quellcode der Anwendung. Insbesondere eine ad hoc zustandekommende Interaktion wird somit unterbunden.[21] Ebenfalls wird auch der Unterschied in der Art und Weise der Verknüpfung von Informationen REST-konformer Dienste zu Web Services deutlich. Dadurch dass REST auf dem Konzept von Ressourcen (s.h. Abschnitt 3.3.1) aufbaut, welche grundsätzlich adressierbar sind und durch die Existenz einfacher und zweckmäßiger Standards wie z.B. URIs oder der XML Linking Language, lassen sich solche Verknüpfungen auf einfachem Wege realisieren. Web Services, welchen hingegen weder das Konzept von Ressourcen noch ein ähnliches zugrundeliegt, ermöglichen eine solche Verknüpfung auf allgemeinem Wege nicht.

So würde eine solche Verknüpfung durch Web Services typischerweise mittels einer weiteren spezifischen Funktion realisiert, die anhand eines Bezeichners die verknüpften Informationen zur Verfügung stellt. Beispielsweise könnte vom Festplattenhersteller hierzu die Funktion `getDatenblatt` bereitgestellt werden, welche als einzigen Parameter einen Bezeichner des Datenblattes erwartet. Wenngleich dieses Vorgehen im Kontext eines eng gekoppelten Clients u.U. unproblematisch ist, so ist eine lose Kopplung – also eine Nutzung der Dienste ohne dieses spezifische Wissen – auf diesem Wege kaum möglich.

Die konkrete Anfrage zur Ermittlung des Datenblattes sieht wie folgt aus:

```
47 GET http://www.wdc.com/products/WD7500BPVT.pdf
48 Accept: text/xml
```

Der Server antwortet in diesem Fall jedoch mit einer Fehlermeldung.

```
49 415 Unsupported Media Type
```

[21] Diese Einschränkung von Web Services liegt allerdings nicht zwingend vor. So ist es denkbar, eine entsprechende Anpassung der Software auch automatisiert zur Laufzeit, also nach Abruf einer WSDL-Beschreibung, durchzuführen. Dieses Vorgehen ist aufgrund einer Vielzahl von technischen Problemen allerdings zurzeit nicht praxistauglich und es ist fraglich, ob es diesbezüglich zu wesentlichen und praktisch anwendbaren Fortschritten kommen wird.

```
50 Possible types are: application/pdf
```

Die Anfrage kann nicht ausgeführt werden, da die Ressource nicht in Form von XML ausgeliefert werden kann. Da der Client aber das einzige vom Server zur Verfügung gestellte Repräsentationsformat nicht kennt, ist es diesem nicht möglich, die Informationen zu ermitteln. An dieser Stelle liegt also eine Inkompatibilität auf der Strukturschicht, wie in Abschnitt 2.2 erläutert, vor.

Im Anschluss an diesen fehlgeschlagenen Versuch, weitere Details über den Artikel zu ermitteln, versucht der Client nun ähnliche Artikel, die der Onlineshop führt, zu finden. Hierzu erstellt dieser eine Sucheanfrage und übergibt diese an die zuständige Ressource, dessen URI ihm aus der Indexressource durch das Element `produktsuche` bekannt ist. Er formuliert diese Anfrage konkret in der Sprache SPARQL (s.h. [31]) und teilt dies dem Server anhand des HTTP-Parameters `Content-Type` mit. Ferner gibt der Client auch das Format, in dem er die Antwort wünscht, mittels des HTTP-Parameters `Accept` an. Im konkreten Fall sucht der Client nach Artikeln desselben Herstellers mit einer Zugriffszeit, die bei weniger als 15ms liegt.

```
51 POST http://shop.de/search
52 Accept: application/sparql-results+xml
53 Content-Type: application/sparql-query
54
55 PREFIX <urn:businessDescriptionLanguage>
56 PREFIX s:<http://shop.de/>
57 SELECT ?artikel, ?bez, ?kap
58 WHERE {
59         ?artikel
60                 hersteller <http://www.wdc.com/aboutUs.xml>;
61                 bezeichnung ?bez;
62                 s:kapazitaet ?kap.
63         FILTER (s:zugriffszeit < 15)
64 }
```

Der Server führt diese Suche aus und liefert das Sucherergebnis in dem vom Client geforderten Format zurück.

```
65 <?xml version="1.0" encoding="UTF-8"?>
66 <sparql xmlns="http://www.w3.org/2005/sparql-results#">
```

```
<head>
  <variable name="artikel"/>
  <variable name="bez"/>
  <variable name="kap"/>
</head>
<results>
  <result>
    <binding name="artikel">
      http://shop.de/products/p5532
    </binding>
    <binding name="bez">WD7500BPVT</binding>
    <binding name="kap">750</binding>
  </result>
  <result>
    <binding name="artikel">
      http://shop.de/products/p3174
    </binding>
    <binding name="bez">WD1000TG</binding>
    <binding name="kap">100</binding>
  </result>
 </results>
</sparql>
```

Das Resultat enthält, neben dem bereits bekannten Artikel, einen zweiten Artikel auf den die Sucheparameter zutreffen. Anhand der ermittelten URI dieses Artikels ist der Client nun imstande, die Ressource auszulesen, die diesen Artikel beschreibt.

Statt einer weiteren Recherche bestellt der Client nun jedoch den ursprünglich gefundenen Artikel, indem er eine entsprechende Bestellung erzeugt. Die Sprachelemente, aus denen sich die Bestellung zusammensetzt, sind der fiktiven Beschreibungssprache für Onlineshops entnommen. Hier ist insbesondere auch anzumerken, dass der Client den zu bestellenden Artikel in Zeile X REST-konform, d.h. mittels einer URI referenziert. Ebenso spezifiziert dieser die Lieferadresse in Zeile X anhand einer Ressource, die auf einem weiteren System vorliegt und koppelt somit das System des Onlineshops mit dem Dienstleister, der diese Ressource bereitstellt.

Der Server des Onlineshops muss daher zur Verarbeitung der Bestellung den Zustand

der entsprechenden Ressource auslesen und interpretieren. Er nimmt also zu diesem Zweck die Rolle eines Clients ein, der REST-konform mit dem Dienst, der die Lieferadresse bereitstellt, interagiert. Damit dieser Informationsaustausch erfolgreich sein kann, müssen beide Systeme wiederum bzgl. der vier Kommunikationsebenen, die in Abschnitt 2.2 erläutert werden, kompatibel sein. Der Dienst des Onlineshops muss also seinerseits HTTP zum Transfer von Daten einsetzen können. Des Weiteren muss eine für beide verständliche Kodierung, als auch eine gemeinsame Darstellungsform der Daten existieren. Ferner muss dem Dienst, der die Bestellung verarbeitet, die Bedeutung der Sprachelemente der Adressressource, so weit wie zur Verarbeitung der Bestellung notwendig, bekannt sein.

Die Bestellung übermittelt der Client an die in der Indexressource mittels des Elementes `bestellungsAnnahme` gekennzeichnete URI. Er nutzt hierfür die HTTP-Methode `POST`, da es sich bei der Bestellung weder um eine sichere noch um eine idempotente Aktion handelt. Da der Server REST-konform ist, bedarf es keiner Erweiterung der Serverschnittstelle um eine spezifischen Funktion `bestelle`, wie dies bei anderen verteilten Systemen z.T. der Fall wäre.

```
89  POST http://shop.de/order
90  Accept: text/xml
91  Content-Type: text/xml
92
93  <?xml version="1.0" encoding="UTF-8"?>
94  <bestellung xmlns="urn:businessDescriptionLanguage">
95    <lieferadresse xlink:href="http://myprivatesite.de/adresse"/>
96    <artikelliste>
97      <artikel xlink:href="http://shop.de/products/p5532">
98        <preis waehrung="Euro">250</preis>
99        <anzahl>1</anzahl>
100     </artikel>
101   </artikelliste>
102 </bestellung>
```

Der Server nimmt diese Bestellung entgegen und leitet den Client durch seine Antwort zu einer durch die Bestellung neu entstandenen Ressource weiter.

```
103 <?xml version="1.0" encoding="UTF-8"?>
104 <link xml:base="http://shop.de/" xlink:href="/order/o65221"/>
```

Der Client folgt der empfangenen Referenz und ruft die Ressource ab.

```
GET http://shop.de/order/o65221
Accept: text/xml
```

Der Server übermittelt darauf hin die neu entstandene Ressource.

```
<?xml version="1.0" encoding="UTF-8"?>
<bestellung
        xml:base="http://shop.de/"
        xmlns="urn:businessDescriptionLanguage">
  <eingegangenAm>2010-08-07 8:31 AM</eingegangenAm>
  <status>versendet</status>
  <kunde xlink:href="http://shop.de/users/323"/>
  <lieferadresse xlink:href="http://myprivatesite.de/adresse"/>
  <rechnung xlink:href="/bills/r78323"/>
  <artikelliste>
    <artikel xlink:href="/products/p5532">
      <preis waehrung="Euro">250</preis>
      <anzahl>1</anzahl>
    </artikel>
  </artikelliste>
</bestellung>
```

Neben den vom Client gemachten Angaben ist die Ressource, welche die Bestellung darstellt, durch den Bestellvorgang um weitere Informationen angereichert.

Abschließend lässt sich an diesem Beispiel auch erkennen, dass die jeweiligen Implementierungen der Dienste verborgen werden. Auch wird insbesondere aufgrund der Nutzung von etablierten Standards eine Implementierung weiterer Clients vereinfacht. Somit hängt die Komplexität eines Clientsystems primär von der Komplexität der an ihn gestellten Aufgaben ab und nicht von technischen Rahmenbedingungen. Beispielsweise wäre es wenig aufwändig, einen Client mittels einer einfachen Skriptsprache zu implementieren, der eine Artikelsuche anhand festgelegter Merkmale, welche vom Benutzer mit konkreten Werten belegt werden können, durchführt.

3.7. Zusammenfassung

Abschließend lässt sich REST als der Architekturstil, der dem WWW zugrundeliegt, zusammenfassen. In diesem Zuge benennt REST die drei folgenden Elemente als wesentlich:

Ressourcen sind eindeutig identifizierbare reale oder virtuelle Gegenstände.

Repräsentation dienen der Darstellung von Ressourcen mittels einer konkreten Syntax.

Konnektoren ermöglichen entfernten Systemen den Zugriff auf Ressourcen mittels Repräsentationen.

Ferner führt REST eine Reihe von Konzepten auf, die der Interaktion im WWW zugrundeliegen:

- Verteilte Systeme interagieren auf Basis des *Client-Server-Paradigmas*.
- Die Interaktion ist aus Sicht des Server *statuslos*.
- Ressourcen können *zwischengespeichert* werden, um die Performance und Skalierbarkeit des WWW signifikant zu erhöhen.
- Der Zugriff auf alle Ressourcen, die eine Software bereitstellt, erfolgt mittels einer *einheitlichen Schnittstelle*.

REST ermöglicht es somit, die architektonischen Grundlagen des WWW, die wesentlich zu dessen Erfolg beitragen, gezielt auf Softwaresysteme zu überführen.

4. Modellgetriebene Softwareentwicklung

4.1. Einleitung

Kernaufgabe der praktischen Informatik ist die Speicherung und Verarbeitung von Informationen. Aufgrund der kontinuierlich steigenden Komplexität die mit dieser Aufgabe verbunden ist, sind in der Vergangenheit eine Reihe von Verfahren zur Vereinfachung der Softwareentwicklung entstanden. Wenngleich sich diese Verfahren z.T. erheblich voneinander unterscheiden, so kristallisieren sich doch einige vielversprechende und grundlegende Konzepte heraus. Eines dieser Konzepte ist das Abstrahieren durch Modellbildung. Statt alle Details eines Gegenstandes und somit typischerweise eine sehr große Anzahl von Informationen zu betrachten, werden die für die konkrete Zielsetzung relevanten Informationen in Form eines Modells zusammengetragen. Im Anschluss daran können die weiteren Arbeitsschritte nun an dem Modell durchgeführt werden. Ein Beispiel hierfür sind höhere Programmiersprachen, die von Software als Folge von Maschineninstruktionen abstrahieren und es ermöglichen, diese als Modell basierend auf verschiedenen Kontroll- und Datenstrukturen zu spezifizieren.

Dem Prinzip der Abstraktion durch Modellbildung verschreibt sich die *modellgetriebene Softwareentwicklung* im besonderen Maße, um Software prägnant und ausdrucksstark zu beschreiben.

4.2. Motivation

Motiviert ist die modellgetriebene Softwareentwicklung dabei nach [33] primär durch folgende Ziele:

Erstens steht die *Steigerung der Effizienz* der Softwareentwicklung, konkret eine Verkürzung der Entwicklungsdauer, sowie die Vereinfachung der Weiterentwicklung und Wartung der Software im Vordergrund. Eins der größten Hemmnisse bei der Entwicklung und Modifikation von komplexen Softwaresystemen ist, dass an ein solches System eine große Anzahl verschiedener fachlicher wie technischer Anforderungen gestellt werden,

welche letztendlich in stark vermengter Form mittels Quellcode umgesetzt werden. In einem modellgetriebenen Vorgehen werden die einzelnen Teile der zu entwickelnden Software im Hinblick auf die Anforderungen betrachtet und dessen wesentliche Eigenschaften formal modelliert. Dies hat den Vorteil, dass ein solches Modell die Softwaremerkmale, die zur Umsetzung einer oder einiger weniger Anforderungen dienen, zweckmäßig und separiert von anderen Merkmalen abbilden kann. Somit enthält ein solches Modell i.d.R. sehr viel ausdrucksstärkere Elemente als eine höhere Programmiersprache. Darüber hinaus kann auf die Modellierung von Details, die bzgl. der betrachteten Anforderungen unerheblich sind, verzichtet werden. Zur Implementierung notwendige Details können durch Transformationen hinzugefügt werden. Solch eine modulare Beschreibung von fachlichen wie technischen Merkmalen von Software ist mit typischen Programmiersprachen für gewöhnlich nicht umsetzbar. Die im Rahmen eines modellgetriebenen Vorgehens erreichte Trennung unterschiedlicher Aspekte der Software durch Modelle und die ausdrucksstarke Semantik, die mit diesen verbunden ist, bedingt i.A. eine einfachere und damit effizientere Softwareentwicklung.

Zweitens wird eine mögliche *Wiederverwendung* von Teilen der Software vereinfacht. Nicht selten hemmt die starke Bindung von Fachlogik an technische Implementierungsdetails, welche häufig durch höhere Programmiersprachen bedingt ist, die Wiederverwendbarkeit. Somit wird die Wiederverwendung von Fachlogik in einer anderen Software wesentlich erschwert, wenn nicht beide Anwendungen auf der gleichen oder zumindest sehr ähnlichen technischen Implementierung basieren. Im Gegensatz hierzu kann ein Modell eines solchen Softwareaspekts, welches unabhängig von technischen Details ist, grundsätzlich mittels verschiedener Technologien implementiert werden. Die Portierung des modellierten Aspekts bedarf typischerweise nicht der Änderung des Modells selbst, sondern nur einer Anpassung der Transformation, welche diese auf eine konkrete Technologie abbildet. Insbesondere wird somit auch die Änderung der technischen Plattform einer bestehenden Anwendung vereinfacht.

Drittens kann ein modellgetriebener Ansatz wesentlich zur *Qualitätssicherung* der Software beitragen, da durch Transformationen der Prozess von abstrakten Softwaremerkmalen hin zu einer konkreten Implementierung ebenfalls formal spezifiziert wird. Somit kann der Entstehungsprozess einer Implementierung, ausgehend von den maßgebenden Modellen, jederzeit nachverfolgt und überprüft werden. Ferner führt die automatisierte Transformation zu einer großen Homogenität der erzeugten Implementierung, da bei der Transformation gleichartige Modellelemente stets zu gleichartigem Quellcode überführt werden. Daneben besteht die Möglichkeit, unterschiedliche Aspekte des Software-

systems mittels verschiedener Transformationen umzusetzen, deren Ergebnisse schließlich verknüpft werden. Somit wird eine Spezialisierung der Entwickler, welche sich in der Erstellung verschiedener Transformationen ausdrückt, gefördert.

4.3. Definition und Einordnung

Der Begriff der modellgetriebenen Softwareentwicklung (Model Driven Software Development, MDSD) wird in der Literatur nicht einheitlich verwendet. Dennoch gibt es einige Aspekte, die MDSD klar zuzuordnen sind. So definiert [33] MDSD wie folgt:

> Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [33, S. 11]

Die drei hieraus entnehmbaren wesentlichen Aspekte von MDSD sind demnach folgende:

formale Modelle Zentraler Bestandteil von MDSD sind Modelle, also eine auf das Wesentliche reduzierte Darstellung eines Softwaresystems oder Teilen davon. Ein Merkmal dieser Modelle ist ferner, dass sie formal sind, d.h. maschinenles- und auswertbar (vgl. Abschnitt 4.4.2).

lauffähige Software Das letztendliche Ziel von MDSD ist die Erstellung von lauffähiger Software. Der Begriff Software ist hier im weiteren Sinne zu verstehen. So kann dieser auch Dokumentation über die Anwendung und deren Entwicklungsprozess umfassen.

automatisierte Generierung Lauffähige Software wird im Kontext von MDSD durch automatisierte Transformationen der formalen Modelle erstellt. Es bedarf daher einer formalen Transformationsvorschrift, welche automatisiert ausgeführt werden kann (vgl. Abschnitt 4.4.4).

Insbesondere zur Abgrenzung von klassischen Entwicklungsansätzen (vgl. Abschnitt 4.5.1) ist die zitierte Definition aber zu allgemein gefasst. Aus diesem Grund liegt dieser Arbeit eine erweiterte Definition zugrunde:

Modellgetriebene Softwareentwicklung ist ein Oberbegriff für Techniken, welche sich formaler Modelle bedienen, um die wesentlichen Aspekte und Eigenschaften einer Software oder Teile davon auf hohem Abstraktionsniveau möglichst ausdrucksstark und prägnant zu formulieren und diese Modelle mittels automatisierter Transformationen in lauffähige Software zu überführen.

4.4. Konzepte

4.4.1. Abstraktion

Eines der zentralen Konzepte von MDSD ist das Abstrahieren, also die Fokussierung auf die wesentlichen Informationen bei Vernachlässigung der weniger bedeutenden Elemente. MDSD setzt somit eine Entwicklung in der Informatik fort, die in der Vergangenheit zu signifikanten Qualitäts- und Effizienzsteigerungen geführt hat. Als Beispiel ist die Abstraktion von Programmen als Folge von Maschineninstruktionen hin zu einer Menge von Daten- und Kontrollstrukturen in Form höherer Programmiersprachen wie auch die Abstraktion realer Maschinen durch virtuelle oder die objektorientierte Softwareentwicklung anzuführen. Allerdings sind die in der Vergangenheit gemachten Abstraktionsansätze zumeist beschränkt bzgl. ihrer Ausdrucksfähigkeit konkreter Merkmale von Software.

So ist die Erstellung umfangreicher Anwendungen trotz höherer Programmiersprachen nach wie vor ein komplexes Vorhaben. Dies liegt neben der Komplexität, die in der abzubildenden Fachlogik verankert ist, u.A. darin begründet, dass Daten- und Kontrollstrukturen, also die Basiselemente, mit denen Software typischerweise beschrieben wird, wenig spezialisiert und damit i.A. wenig ausdruckstark bzgl. der konkreten Merkmale der zu erstellenden Software sind. Typischerweise muss die Software aus einer großen Anzahl dieser Basiselemente zusammengesetzt werden, was z.T. mit einer ausgeprägten Verteilung der Implementierung eines Softwaremerkmals über verschiedene Codefragmente einhergeht. Ebenfalls ist die klare Trennung der Implementierung verschiedener Merkmale, ebenso wie die Trennung fachlicher und technischer Merkmale oftmals nur mit erheblichem Aufwand oder gar nicht möglich.

Der Grund für dieses Dilemma liegt zu einem großen Teil in dem Vorgang der Abstraktion selbst. Zur Unterscheidung von Wesentlichem und Unwesentlichem ist ein der Abstraktion zugrundeliegender Zweck notwendig. Die Art der Anforderungen, die an Software gestellt werden, und die daraus resultierenden Merkmale sind aber derart vielfältig, dass eine bzgl. ihrer Prägnanz und Ausdruckstärke optimale Programmiersprache für Software im Allgemeinen nicht existiert. Der Ansatz, Sprachen zu entwerfen, um Software im Allgemeinen zu realisieren, wie er u.A. höheren Programmiersprachen zugrundeliegt, ist daher beschränkt. Aus dieser Erkenntnis heraus zielt die modellgetriebene Softwareentwicklung nicht darauf ab, weitere Abstraktionen für Software im Allgemeinen, sondern diese für spezielle Aspekte einer zu erstellenden Software oder Softwarefamilie zu finden. Ein wesentliches Merkmal eines modellgetriebenen Vorgehens besteht also darin, sinnvolle Abstraktionen zu finden, mittels derer die konkrete Software ausdruckstark und

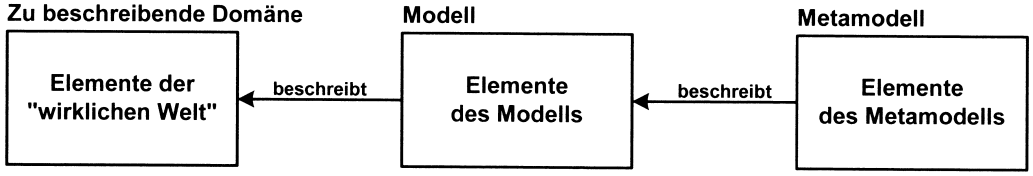

Abbildung 4.1.: Beziehung zwischen Original, Modell und Metamodell ([33, S. 60] entnommen)

prägnant beschrieben werden kann.

4.4.2. Modellierung

Realisiert wird dieser Kerngedanke der modellgetriebenen Softwareentwicklung durch *Modelle* und *Metamodelle*. Ein Modell ist im Kontext von MDSD durch folgende an [32] angelehnte Merkmale gekennzeichnet [1]:

Abbildung Ein Modell ist ein Abbild eines real existierenden oder virtuellen Originals, das selbst wieder ein Modell sein kann.

Abstraktion Nur diejenigen Merkmale des Originals, welche als wesentlich angesehen werden, werden durch das Modell beschrieben.

Pragmatismus Ein Modell dient einem Ziel. Es ist dem Original daher nicht von sich aus zugeordnet, sondern wird zu diesem durch eine Interpretation, die der Umsetzung des Ziels sachdienlich ist, in Beziehung gesetzt.

Formalität Die Strukturen, aus denen ein Modell gebildet ist, und deren Zusammensetzung, also dessen abstrakte Syntax, sind vollständig und eindeutig spezifiziert. Das Modell ist maschinell les- und auswertbar.

Eine weitere wesentliche Eigenschaft eines Modells im Kontext von MDSD ist, dass dessen abstrakte Syntax selbst als Modell vorliegt und damit seinerseits ebenfall maschinenlesbar ist. Ein Modell, das die abstrakte Syntax eines anderen Modells beschreibt, wird im Kontext von MDSD Metamodell genannt. Der Zusammenhang von Original, Modell und Metamodell ist in Abb. 4.1 graphisch skizziert.

[1] Es ist anzumerken, dass die Begriffe *Modell* und *formales Modell* im Folgenden synonym genutzt werden.

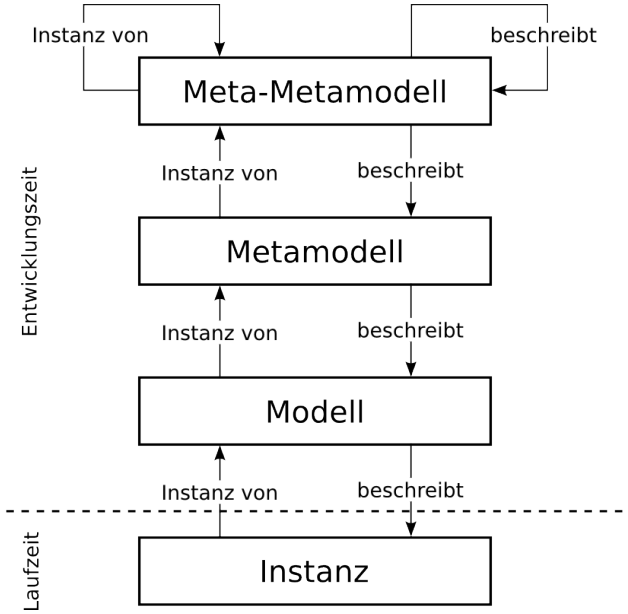

Abbildung 4.2.: Metamodell-Ebenen (angelehnt an [33, S. 62])

Wenngleich die Nutzung von Modellen in der Informatik ein übliches Vorgehen, z.B. in Form höherer Programmiersprachen (vgl. Abschnitt 4.5.1) ist, so werden die spezifizierten Modellelemente und deren Zusammensetzung, sprich die Syntax des Modells, klassischerweise nicht formal spezifiziert. [2] Im Kontext von MDSD ist ein Metamodell, welches definitionsgemäß selbst ein Modell ist, jedoch ebenfalls formal spezifiziert. Der Vorteil dieses Vorgehens ist, dass auch ein Metamodell maschinell eingelesen und verarbeitet werden kann, sofern dessen Metamodell, also das Meta-Metamodell, der Anwendung bekannt ist. Somit können der das Modell verarbeitenden Anwendung alle Eigenschaften des Modells durch ein solches Metamodell zur Laufzeit bekannt gegeben werden.

Obgleich man diese Beziehung zwischen Modellen beliebig oft wiederholen könnte, so beschränkt man sich doch typischerweise auf insgesamt vier Schichten, welche in Abb. 4.2 dargestellt sind: Die Begrenzung auf vier Schichten wird dadurch herbeigeführt, dass das

[2] Beispielsweise wird die Syntax der Programmiersprache Java, also dessen Metamodell, durch [17] mittels natürlicher Sprache beschrieben. So besagt die Spezifikation u.a., dass eine Klasse aus beliebig vielen Attributen besteht, welchen wiederum ein Typ zugeordnet werden muss. Um Werkzeuge zu erstellen, die diese Syntax verarbeiten, muss also eine manuelle Formalisierung dieser Spezifikation, z.B. in Form von Quellcode, durch einen Entwickler ausgeführt werden. Eine nachträgliche Änderung des Metamodells bedingt eine Wiederholung dieses Schrittes und macht damit eine Anpassung des Metamodells sehr aufwändig.

Meta-Metamodell durch sich selbst beschrieben wird und somit sein eigenes Metamodell ist.

4.4.3. Domänenspezifische Sprachen

Zur Speicherung und Verarbeitung eines Modells bedarf es neben der abstrakten Syntax, gegeben durch ein Metamodell, auch einer konkreten Syntax, also einer Notation der Modellelemente. Die Form dieser Notation kann sowohl graphischer als auch textueller Natur sein. Auch sind Mischformen möglich. Die Kombination aus abstrakter und konkreter Syntax wird im Kontext von MDSD als *domänenspezifische Sprache* (Domain Specific Language, DSL) bezeichnet. Eine DSL ist somit eine formale Sprache, dessen abstrakte Syntax durch ein Metamodell im Sinne von MDSD (vgl. Abschnitt 4.4.2) definiert ist.

Eine wesentliche Bedeutung erlangen DSLs in der MDSD insbesondere dadurch, dass die Art und Weise der Darstellung von Modellen wesentlichen Einfluss auf die Lesbarkeit der Modelle für einen Entwickler hat und somit bedeutend zur Modifizierbarkeit und Wartbarkeit einer Anwendung beitragen.

Abschließend ist anzumerken, dass es zu einer abstrakten Syntax durchaus mehrere konkrete Syntaxen geben kann und damit mehrere DSLs auf Basis desselben Modells definiert werden können.

4.4.4. Transformation

Der Begriff *Transformation* bezeichnet im Kontext von MDSD formal spezifizierte Abbildungsvorschriften zwischen Modellen auf der Basis ihrer Metamodelle. Transformationen sind im Zusammenhang mit der modellgetriebenen Softwareentwicklung deshalb so essenziell, da durch sie die Modelle, die eine Software beschreiben, in ein maschinenausführbares Format[3] überführt werden und somit erst durch diesen Schritt Software entsteht.

Eine solche Transformation, die Modelle einliest und die Software oder Teile davon und gegebenenfalls dessen Dokumentation generiert, ist üblicherweise wiederum aus einer Reihe von Transformationen zusammengesetzt, welche nur einzelne Teile der Eingabemodelle oder nur bestimmte Aspekte von ihnen verarbeiten. Durch diese Möglichkeit

[3] Typischerweise handelt es sich bei einem solchen Format um eine höhere Programmiersprache, die dann mithilfe eines gewöhnlichen Compilers weiterverarbeitet und somit letztendlich eine ausführbare Anwendung generiert wird.

der Verschachtelung von Transformationen und der damit verbundenen Separierung verschiedener Aspekte der Softwareentwicklung wird eine modulare Abbildung des Softwareentwicklungsprozesses ermöglicht und somit eine Wiederverwendung von Teilen dieses Prozesses vereinfacht.[4] Des Weiteren lassen sich Transformationen in *Modell-zu-Modell-* und *Modell-zu-Text-Transformationen* unterteilen. Die Erstegenannten erzeugen aus Eingabemodellen Ausgabemodelle, Letztere erzeugen hingegen Text.[5]

4.5. Abgrenzung

4.5.1. Klassische Softwareentwicklung

Wenngleich die Unterschiede zwischen einem modellgetriebenen und einem klassischen Entwicklungsansatz[6] auf den ersten Blick vielfältig erscheinen, so kann man bei genauer Betrachtung eine Reihe von Gemeinsamkeiten feststellen. Insbesondere die Nutzung von formalen Modellen ist beiden gemein. So ist die Basis einer jeden Programmiersprache ein formales Modell, wie es in Abschnitt 4.4.2 definiert ist. Darüber hinaus stellt die automatisierte Übersetzung von Quellcode in Maschineninstruktionen mittels eines Compilers eine Transformation, wie in Abschnitt 4.4.4 erläutert, dar.

Die Unterschiede zu MDSD ergeben sich daher viel mehr aus der Anwendungsart dieser Vorgehensweisen. In MDSD dienen Modelle dazu, Software möglichst ausdrucksstark und prägnant zu beschreiben. Aus diesem Grund sind die Modelle und Metamodelle speziell auf den zu beschreibenden Aspekt zugeschnitten, während in der klassischen

[4] Wenngleich typische Softwarebibliotheken, welche in höheren Programmiersprachen verfasst und im Zusammenhang mit diesen eingesetzt werden, ein ähnliches Ziel verfolgen, so werden von diesen i.d.R. zahlreiche nicht funktionale Anforderungen, wie z.B. die eingesetzte Programmiersprache oder die Softwareplattform und die Verfügbarkeit anderer Bibliotheken, gestellt. Da ein Modell im Sinne von MDSD unabhängig von Transformationsvorschriften spezifiziert wird, ist es möglich unabhängig voneinander verschiedene Transformationen für dasselbe Modell, aber für unterschiedliche Softwareplattformen zu erstellen. Somit ist es möglich, funktionale Aspekte der Software durch Modelle und nicht funktionale Aspekte durch Transformationen zu beschreiben.

[5] Bei einem solchen Text kann es sich jedoch durchaus wiederum um ein Modell handeln, sofern der Text einer konkreten Syntax eines Modells genügt. Dies ist beispielsweise der Fall, wenn Transformationen dazu verwendet werden, Quellcode zu erstellen. Dieser Quellcode liegt zwar in Form einer Folge von einfachen Zeichen, wie z.B. dem ASCII-Zeichensatz (s.h. [10]) vor, folgt aber einer formalen Sprache und kann somit als Modell interpretiert werden. Aus diesem Grund ist eine solche Unterscheidung z.T. irreführend.

[6] Klassische Softwareentwicklung bezeichnet in diesem Zusammenhang die Entwicklung von Software durch die Programmierung mittels höherer Programmiersprachen, der evtl. eine nicht formale Modellierung der Software, z.B. durch Diagramme, vorausgeht.

Softwareentwicklung primär standardisierte Modell, z.B. in Form höherer Programmiersprachen, genutzt werden. Die Transformation der Merkmale der zu erstellenden Software zu Quellcode wird im klassischen Ansatz direkt durch den Entwickler durchgeführt und nicht formal spezifiziert. Dies bedingt u.a., dass ähnliche Merkmale der Software wiederholt durch den Entwickler programmiert, sprich von ihm zu Quellcode transformiert werden müssen. [7]

Ferner bedingt dieser Ansatz typischerweise die Mischung von fachlichen und technischen Aspekten, die ihren Ursprung in der zur Implementierung eingesetzten Technologie hat. Auch wird durch diesen Ansatz keine gleichbleibende Abbildungsweise gleichartiger oder ähnlicher Softwaremerkmale garantiert. Der modellgetriebene Ansatz, zu dessen Umsetzung zuerst die wesentlichen Softwaremerkmale formal beschrieben werden, anschließend Transformationen spezifiziert und letztlich mit deren Hilfe lauffähige Software erstellt wird, ist hingegen wesentlich modularer aufgebaut. So kann strikt zwischen Merkmalen der Software und der Art und Weise der entsprechenden Implementierung unterschieden werden. Dies lässt insbesondere eine einfachere Analyse der verschiedenen Merkmale der Software zu und vereinfacht dadurch Entwicklung und Wartung.

4.5.2. Computer-Aided Software Engineering (CASE)

Ein in den 90er Jahren aufkommender Ansatz, Abstraktion in der praktischen Informatik weiter voranzutreiben, ist das sogenannte Computer-Aided Software Engineering (CASE). Wenngleich der Begriff CASE in der Literatur nicht einheitlich verwendet wird, so analysiert [2] diesen ausführlich und ordnet ihm folgende Bedeutung zu:[8]

> [...] CASE [befaßt sich] mit allen computerunterstützten Hilfsmitteln – CASE-Werkzeugen (tools) genannt – die dazu beitragen, die Software-Produktivität und die Software-Qualität zu verbessern. [2, S. 123]

Wenngleich diese Erklärung wenig spezifisch ist, so wird hierzu anknüpfend insbesondere die Vollständigkeit als ein Merkmal von CASE-Umgebungen angeführt.

[7] Dieser Vorgang wird bei modernen Programmiersprachen z.T. durch Abstraktionsmechanismen, wie z.B. der Vererbung in objektorientierten Sprachen, unterstützt. Diese Vereinfachungen sind aufgrund der generischen Natur von höheren Programmiersprachen aber begrenzt.

[8] Teilweise wird CASE auch im rein wörtlichen Sinne, also als computerunterstützte Softwareentwicklung genutzt und somit lediglich als eine Art der Softwareentwicklung verstanden, die durch Software unterstützt wird. Da moderne Softwareentwicklungsprozesse aber ohne eine solche Unterstützung von Werkzeugen, wie z.B. integrierte Entwicklungsumgebungen, Compilern und Versionverwaltungswerkzeugen praktisch undenkbar sind, ist diese Interpretation letztendlich äquivalent mit dem Begriff der Softwareentwicklung im Allgemeinen und somit wenig sinnvoll.

> Vollständigkeit i.w.S. bezogen auf eine CASE-Umgebung bedeutet, daß alle Aktivitäten aller Personen, die am Software-Erstellungsprozess beteiligt sind, für alle Produktarten optimal durch Hilfsmittel der CASE-Umgebung unterstützt werden. [2, S. 135]

Insbesondere der Versuch, diesen Vollständigkeitsanspruch umzusetzen, führt bei vielen CASE-Werkzeugen zu einer Reihe von Unzulänglichkeiten. So sind viele dieser Werkzeuge enorm komplexe Softwareentwicklungssysteme, die nicht selten eine langwierige Einarbeitung erfordern.

Wesentlich kritischer ist jedoch, dass viele CASE-Werkzeuge in dem Bestreben, alle erdenklichen Belange der Softwareentwicklung abzudecken, eine Reihe von generischen und bzgl. der damit zu erstellenden Software oft wenig sinnvollen Ansätze zur Verfügung stellen. Auch leiden diese Werkzeuge häufig unter der Tatsache, dass sie trotz großer Bemühungen den Vollständigkeitsanspruch nicht umfassend umsetzen. Dies führt nicht selten zu dem Phänomen, dass zwar typische Anforderungen an das zu erstellende Softwaresystem schnell umgesetzt werden können, spezifischere Anforderungen aber u.U. nur durch eine nicht bestimmungsgemäße Nutzung der CASE-Werkzeuge realisiert werden können. Somit kommt es u.U. zu einer wesentlich erschwerten Verständlichkeit der Programmspezifikation und somit auch zu einer erschwerten Modifikation und Wartung der Software. Die angestrebte Steigerung der Entwicklungsgeschwindigkeit und der Softwarequalität durch den Einsatz von CASE-Werkzeugen kann sich somit ins Gegenteil verkehren.

MDSD stellt hingegen nicht Werkzeuge, sondern vielmehr auf die Software zugeschnittene Modelle und Metamodelle in den Mittelpunkt. Ein typisches MDSD-Werkzeug dient daher nicht direkt der eigentlichen Erstellung der Software, sondern der Erstellung von Modellen und Metamodellen, welche ihrerseits diese Software ausdrucksstark und prägnant beschreiben. Im weiteren Sinne dienen MDSD-Werkzeuge auch dazu, an diese Modelle angepasste Werkzeuge zu erstellen.

Zusätzlich bedingt CASE häufig eine ausgeprägte Abhängigkeit zu einem Softwareentwicklungsprodukt und damit zu dessen Anbieter. Im Kontext von MDSD spielen Werkzeuge, z.B. zur Modellierung oder Transformation, zwar ebenfalls eine wichtige Rolle, diese sind aber im Vergleich mit CASE-Umgebungen typischerweise wesentlich loser gekoppelt, da dessen Informationsaustausch primär auf den zu verarbeitenden Modellen basiert. Somit ist es i.A. einfacher, Verarbeitungsschritte auf andere Werkzeuge zu portieren.

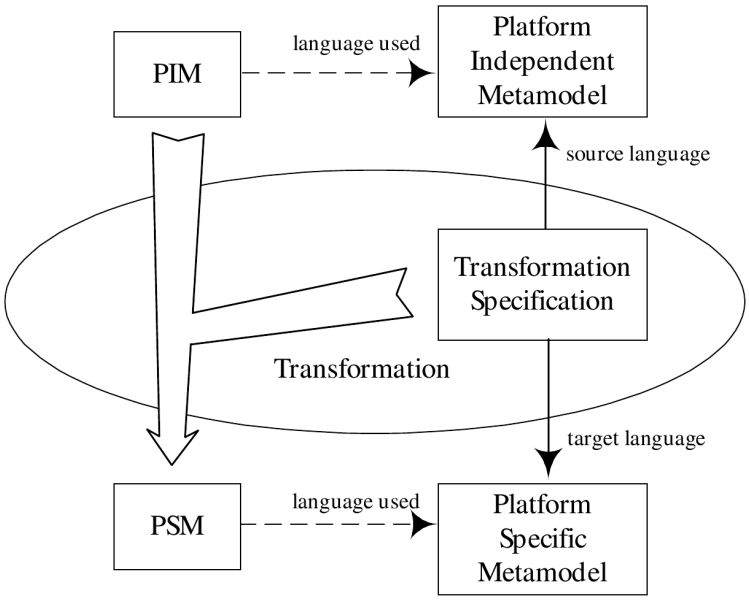

Abbildung 4.3.: Transformation von PIM zu PSM ([23] entnommen)

4.5.3. Model Driven Architecture (MDA)

Eine weiterer Entwicklungsansatz, der häufig im Zusammenhang von MDSD genannt wird, ist die *Model Driven Architecture* (MDA). MDA ist ein von der Object Management Group (OMG) spezifizierter modellgetriebener Entwicklungsansatz (s.h. [23]), der viele der in diesem Kapitel erläuterten Grundlagen aufgreift und konkretisiert. Die OMG hat mit der MDA einen einheitlichen und standardisierten modellgetriebenen Entwicklungsansatz zum Ziel und bindet zu diesem Zweck eine Reihe weiterer Standards, nicht zuletzt die Unified Modeling Language (UML) (s.h. [28]) und Query/View/Transformation (QVT) (s.h. [25]) in diese Bemühungen ein. Somit ist es wenig verwunderlich, dass die MDA sich insbesondere auf die UML zur Beschreibung von Software und die Schaffung entsprechender Werkzeuge fokussiert und die auf UML aufbauende Metaobject Facility (MOF) (s.h. [27]) alternativlos als Metamodell nutzt. Ferner klassifiziert MDA Modelle u.a. als plattform-unabhängig (engl.: platform independent model, PIM) und plattform-spezifisch (engl.: platform specific model, PSM) und nutzt Transformationen, wie in Abb. 4.3 dargestellt, zur Umwandlung. Ein PIM ist ein Modell, welches die Software ohne Berücksichtigung der jeweiligen Plattform beschreibt. Dieses Modell wird mittels Transformationen angereichert und auf eine konkrete Softwareplattform angepasst, so

dass ein PSM entsteht.

Es wird ausdrücklich darauf hingewiesen, dass diese Arbeit nicht auf der Model Driven Architecture basiert und dass die Konzepte des PIM und des PSM, die diese spezifiziert, nicht angewendet werden.

4.6. Zusammenfassung

Zusammenfassend lässt sich die modellgetriebene Softwareentwicklung als weiterentwickelte Anwendung von Abstraktion in der praktischen Informatik beschreiben. Im Unterschied zu vielen bisherigen Ansätzen, wie z.B. höheren Programmiersprachen, wird jedoch nicht versucht, Abstraktionen von Software in ihrer Allgemeinheit zu finden. Stattdessen werden im Rahmen eines modellgetriebenen Entwicklungsprozesses Abstraktionen auf Basis der konkret zu entwickelnden Software und im Hinblick der mit dieser verbundenen Aufgabenstellungen entworfen. Dies erfolgt mittels formaler Modelle, um die automatische Analyse und Weiterverarbeitung der Informationen in Form eines Transformationsprozesses zu ermöglichen. Ein Transformationsprozess basiert zum einen auf einem solchen formalen Modell, also der Beschreibung der Software oder eines Teils davon und einer Transformationsvorschrift. Letztere ist ein formal spezifiziertes Regelwerk, das angibt, wie die Informationen des Modells verarbeitet werden sollen. Grundsätzlich lassen sich zwei Arten von Transformationen unterscheiden. Zum einen sind dies Modell-zu-Modell-Transformationen, welche die Informationen eines Eingabemodells auf ein Ausgabemodell abbilden, welches wiederum als Eingabe einer Transformation dienen kann. Zum anderen sind dies Modell-zu-Text-Transformationen, solche die auf Basis eines Eingabemodells Text generieren. Mit letzten ist es somit insbesondere möglich, Quellcode zu erzeugen. Die modellgetriebene Softwareentwicklung lässt sich somit als konsequente Fortführung der klassischen Softwareentwicklung betrachten.

Teil III.

Anwendung

5. REST-konforme Middleware

5.1. Einleitung

Aus den in Kapitel 3 aufgeführten Merkmalen von REST ergeben sich für eine REST-konforme Software eine Reihe von Anforderungen. In diesem Kapitel wird eine mögliche Umsetzung von Anforderungen, welche sich auf den Transfer und die Darstellung von Daten beziehen, anhand einer REST-konformen Middleware gezeigt. Zweck dieser Middleware ist es, technische Aspekte möglichst separiert von Fachlogik implementieren zu können und somit die Vorteile eines modularen Aufbaus einer Softwareschnittstelle in Form des in Abschnitts 2.2 aufgeführten Schichtenmodells zu erreichen.

Im folgenden Abschnitt werden hierzu die Anforderungen, die im Rahmen dieser Arbeit an die Middleware gestellt werden, erläutert. Im Anschluss daran wird in den Abschnitten 5.3 und 5.4 die Umsetzung dieser Anforderungen mittels eines objektorientierten Entwicklungsansatzes gezeigt. Schließlich wird die Nutzung eines Transferprotokolls, in diesem Fall HTTP, in Abschnitt 5.5 thematisiert.

5.2. Anforderungen

Die übergeordneten Ziele, die mit der Erstellung der Middleware verbunden sind, sind zum einen, möglichst viele Details vor der Fachlogik zu verbergen und somit eine Vereinfachung der Softwareentwicklung und -wartung zu erzielen. Zum anderen soll hierdurch einer wiederkehrenden Implementierung von REST-spezifischen Softwaredetails entgegengewirkt und somit die Qualitätssicherung vereinfacht werden.

Die zu diesem Zweck gestellten Anforderungen an die Middleware werden im Folgenden erläutert.[1]

- Die Middleware soll eine REST-konforme generische Schnittstelle (vgl. Abschnitt 3.4.4)

[1] Einige dieser Anforderungen, wie z.B. die Wahl der Programmiersprache, basieren allerdings weniger auf den in den Kapiteln 3 und 4 vorgestellten Grundlagen, sondern auf pragmatischen Überlegungen bzgl. der zu erstellenden Management-Software.

zur Verfügung stellen, welche möglichst unabhängig von konkreten fachlichen Anforderungen ist.

- Verschiedenartige Repräsentationsformen für Ressourcen (vgl. Abschnitt 3.3.2) sollen einfach implementiert und integriert werden können.
- Die Art und Weise der Verarbeitung und Speicherung von Ressourcen soll durch die Middleware nicht fest vorgegeben werden.
- Es soll eine Abbildung der generischen Schnittstelle auf verschiedene Netzwerkprotokolle mittels Konnektoren möglich sein (vgl. Abschnitt 3.3.3).
- Die auf der Middleware aufbauenden Softwarefragmente sollen im Sinne eines komponentenbasierten Designs möglichst unabhängig voneinander entwickelt und in die Software integriert werden können.
- Die Middleware soll in objektorientierter Form mittels der Programmiersprache Java implementiert werden.
- Die auf der Middleware aufbauenden Komponenten sollen neben der Programmiersprache Java grundsätzlich auch mittels anderer auf der Java Virtual Machine (JVM) basierender Sprachen implementiert werden können.

Die Umsetzung dieser Anforderungen ist im Folgenden in zwei Teile unterteilt. Zuerst wird die der Middleware zugrundeliegende Architektur beschrieben, dann werden einige wesentliche auf dieser Middleware basierenden Komponenten erläutert.

5.3. Einheitliche Schnittstelle

Dem Architekturstil REST folgend sind alle Funktionen, welche vom Server bereitgestellt werden, mittels einer generischen Schnittstelle aufrufbar (vgl. Abschnitt 3.4.4).[2] Die zu diesem Zweck entworfene Schnittstelle besteht aus insgesamt fünf Funktionen, die im Folgenden näher erläutert werden.

load

Beschreibung Die Funktion `load` dient zum laden einer Ressource. Die Funktion ist sicher und idempotent.

Parameter **uri** spezifiziert die zu ladende Ressource.

[2] Die Anbindung an ein Clientsystem mittels eines Konnektors wird in Abschnitt 5.5 erläutert.

Rückgabewert Repräsentation der durch den Parameter `uri` identifizierten Ressource.

save

Beschreibung Mittles der Funktion `save` werden Ressource gespeichert. Existiert die Ressource nicht, wird sie erstellt. Die Funktion ist idempotent.

Parameter **uri** Die uri der zu speichernden Ressource

representation Die Repräsentation der Ressource, die gespeichert werden soll

Rückgabewert Repräsentation der gespeicherten Ressource

create

Beschreibung Mithilfe der Funktion `create` können neue Ressourcen erstellte werden.

Parameter **representation** Die Repräsentation der Ressource die gespeichert werden soll

Rückgabewert Repräsentation der erstellten Ressource

delete

Löschen einer Ressource
Parameter:

Beschreibung Die Funktion `delete` dient zum löschen einer Ressource.

Parameter **uri** Die Uri der zu löschenden Ressource

Rückgabewert –

process

Beschreibung Mittels der Funktion `process` können Repräsentationen verarbeitet werden.

Parameter **representation** Die Repräsentation der Ressource die verarbeitet werden soll.

Rückgabewert Ergebnis der Verarbeitung

Die Methoden `load`, `save`, `create` und `delete` sind insofern generisch, als die aufgeführte Semantik unabhängig von der jeweiligen Art der Ressource ist. Anders verhält es sich mit der Methode `process`. Die Semantik dieser Methode ist abhängig von dem Datentypen der mittels der übergebenen Repräsentation dargestellten Ressource. Somit ist es möglich, typabhängige Funktionalitäten abzubilden. [3]

[3] Beispielsweise könnte die konkrete Semantik eines Aufrufs der Funktion `process`, die mit der Repräsentation einer Warenbestellung parametrisiert ist, dazu führen, die zur Bestellung der aufgeführten Ar-

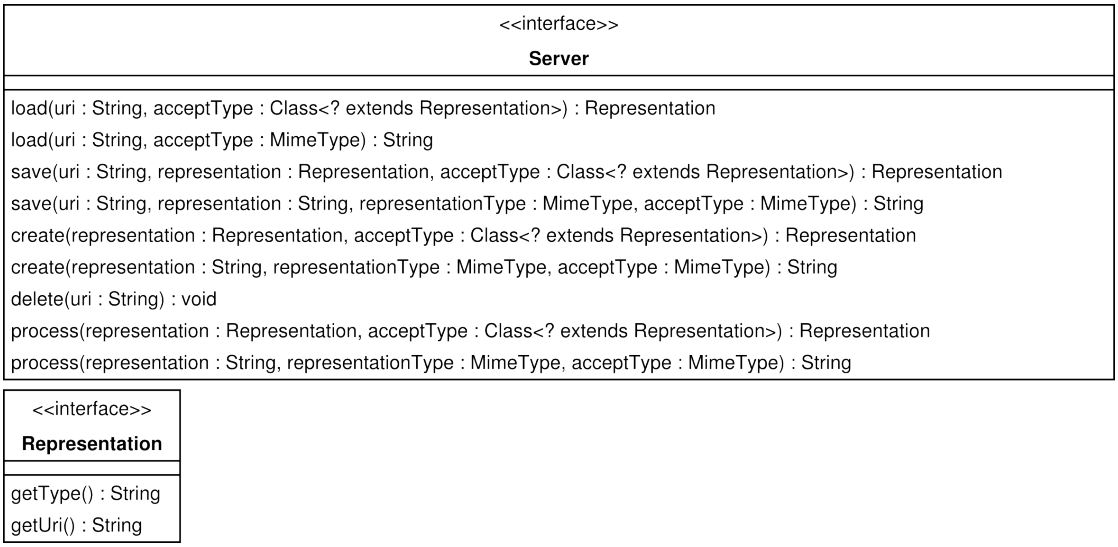

Abbildung 5.1.: Serverschnittstelle

Die Umsetzung der konzeptionellen Schnittstelle in der Programmiersprache Java ist in Abb. 5.1 als UML-Klassendiagram dargestellt. Diese Schnittstelle unterscheidet dich in einigen Punkten von der konzeptionellen Schnittstelle. Dies ist, durch die Anforderung bedingt, verschiedene Repräsentationen für ein und dieselbe Ressource bereitstellen zu können. So existiert neben der Schnittstelle *Server* eine weitere Schnittstelle namens *Representation*. Letztere stellt die von Repräsentationen zu implementierende Schnittstelle dar. Mittels der Funktion `getType` ist es möglich, zu ermittelten, was die Repräsentation beinhaltet. Ferner kann die Ressource, welche die Repräsentation darstellt, mittels der Methode `getUri` herausgefunden werden.

Darüber hinaus beinhaltet die Schnittstelle Server für jede Funktion der konzeptuellen Schnittstelle, mit Ausnahme der Funktion `delete`, jeweils zwei entsprechend benannte Funktionsdeklarationen. Dies hat zum Zweck, sowohl Repräsentationen in eine serialisierte Darstellung als auch in Form eines Java-Objekts übergeben zu können. Des Weiteren kann bei Funktionen, welche eine Repräsentation zurückgeben, mittels des Parameters *acceptType* spezifiziert werden, welche Repräsentationsform hierzu genutzt werden soll. Dies geschieht durch die Angabe der Klasse, welche die Repräsentation implementiert,

tikel notwendigen Schritte durchzuführen. Die Übergabe einer Suchanfrage an diese Funktion könnte hingegen eine entsprechende Suche auslösen und zur Rückgabe einer entsprechenden Ergebnismenge führen.

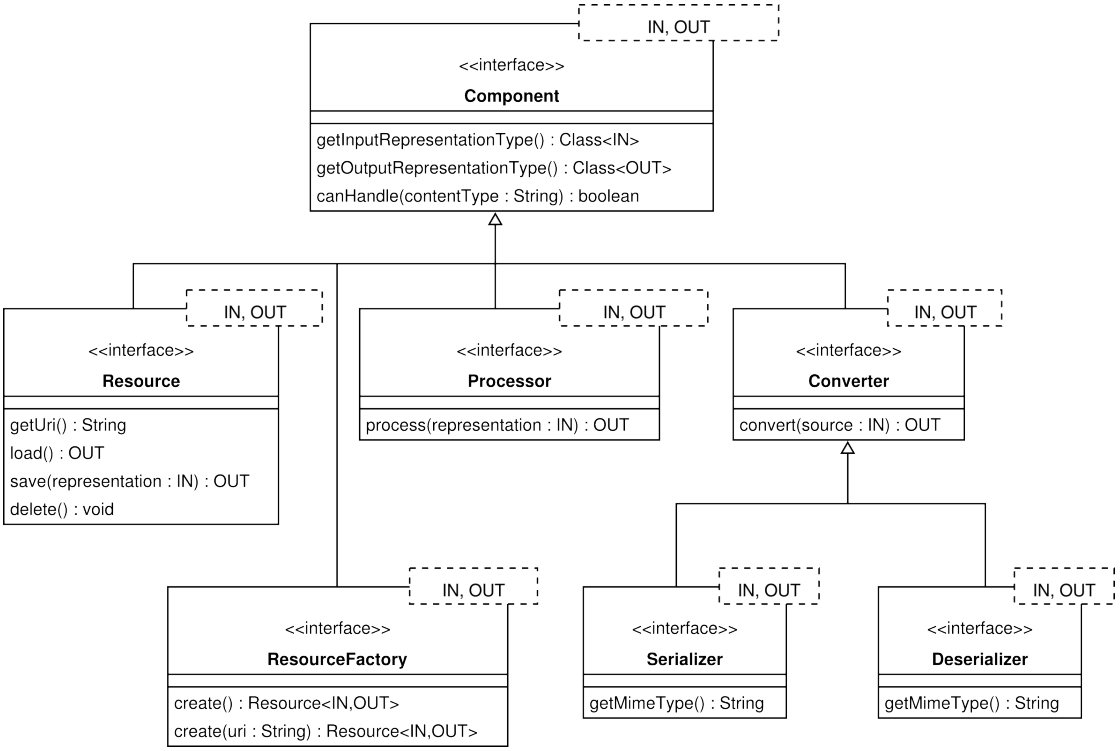

Abbildung 5.2.: Schnittstellen der Komponenten

bei serialisierten Formen alternativ durch einen MimeType (s.h. [15]). [4]

5.4. Objektorientierte Architektur

Um einen modularen und objektorientierten Aufbau einer Software, welche auf dieser Middleware basiert, zu ermöglichen, wird die Menge der Funktionen der Schnittstelle Server sinnvoll unterteilt und in ein objektorientiertes Design, welches in Abb. 5.2 dargestellt ist, überführt. Die Funktionen `load`, `save` und `delete` dienen der direkten Interaktion mit einer, anhand eines übergebenen Ressourcenbezeichners eindeutig bestimmbaren Ressource und werden durch die Java-Schnittstelle *Resource* abgebildet. Ferner wird die Java-Schnittstelle *ResourceFactory* eingeführt, mittels derer eine Funktion zur Erstellung einer neuen Ressource implementiert werden kann. Des Weiteren dient die

[4] Beispielsweise kann mittels der Aufrufe `load("http://bsp.org/Kunde/573", "text/xml")` oder `load("http://bsp.org/Kunde/573", XmlRepresentation.class)` gleichermaßen eine Repräsentation der angegebenen Ressource in einem XML-Format geladen werden.

Java-Schnittstelle *Processor* zur Implementierung der Funktion `process`.

Abweichend vom Fassadenmuster setzt sich die Management-Software jedoch aus mehreren Implementierungen dieser Schnittstellen, an die ein Aufruf delegiert werden kann, zusammen. Der Zusammenhang dieser drei Schnittstellen zur Schnittstelle Server entspricht also weitestgehend dem Fassadenmuster (s.h. [16]). Ein Aufruf einer Funktion der Schnittstelle Server wird durch dessen Implementierung an die entsprechende Funktion einer Implementierung der Schnittstelle Resource, ResourceFactory oder Processor delegiert. Aus diesem Grund muss vor der eigentlichen Abarbeitung der Anfrage entschieden werden, welche dieser Implementierungen genutzt werden soll.

Im Fall der Funktionen `load`, `save` und `delete` geschieht dies, wie in Abb. 5.3 dargestellt, anhand eines Abgleichs des übergebenen Ressourcenbezeichners mit dem der Ressourcenimplementierung zugewiesenen Bezeichner, der anhand der Funktion `getURI` ermittelt werden kann. Zur Bestimmung der zuständigen Implementierung der Schnittstellen ResourceFactory und Processor dient, wie in Abb. 5.4 dargestellt, der Typ der übergebenen Repräsentation.

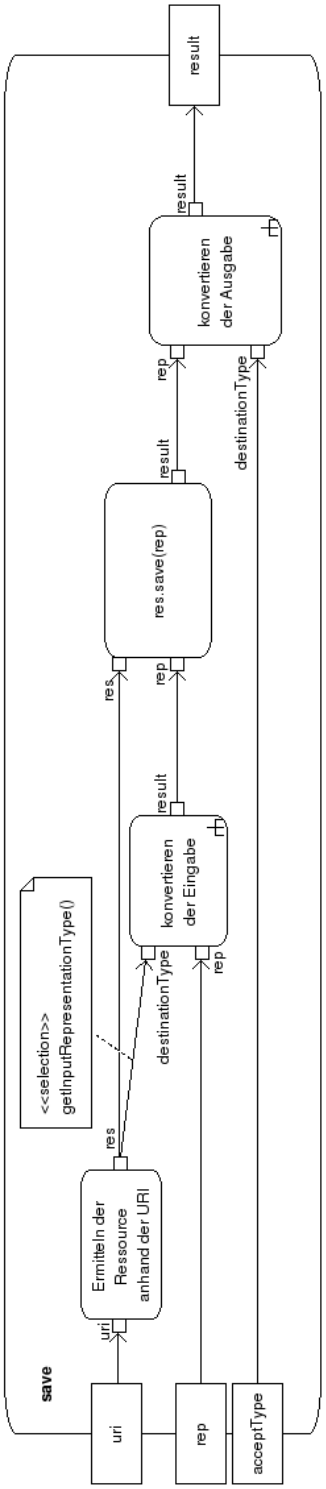

Abbildung 5.3.: Methode „save“

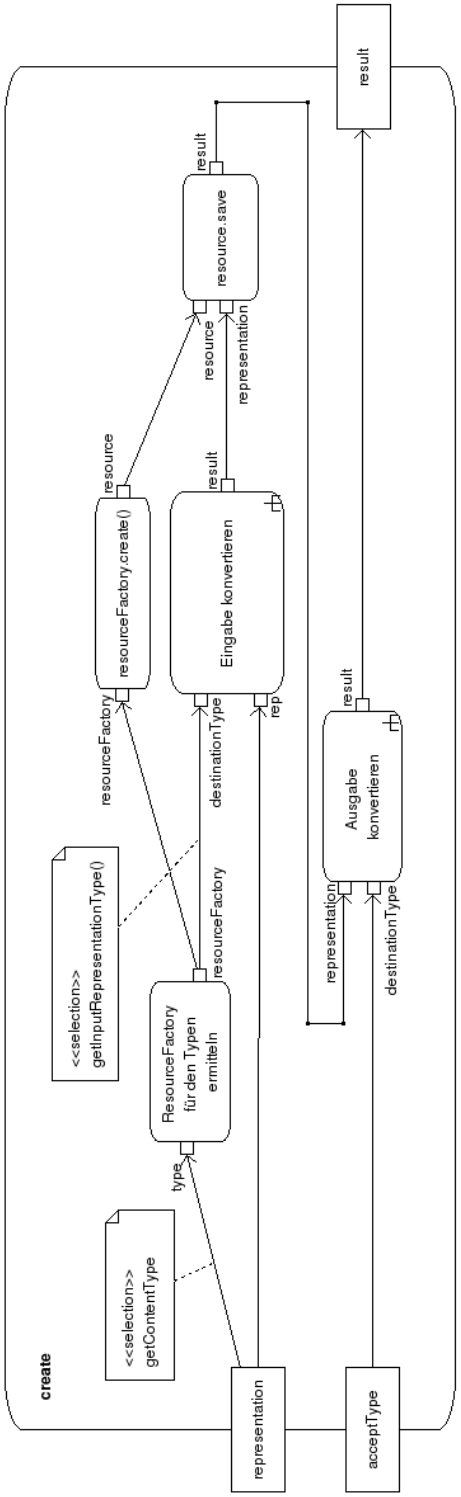

Abbildung 5.4.: Methode „create“

Alle drei Schnittstellen sind Kinder der Schnittstelle *Component*, welche es ermöglicht, die Repräsentationsformen, die die jeweiligen Komponenten als Eingabe verarbeiten können und zur Ausgabe nutzen, zu ermitteln. [5] Darüber hinaus kann zur Laufzeit durch die Funktion `canHandle` überprüft werden, ob die Komponente einen entsprechenden Typ verarbeiten kann. Somit kann insbesondere ermittelt werden, welche Implementierung der Schnittstelle ResourceFactory oder Processor zur Verarbeitung einer Anfrage genutzt werden muss. Durch die Funktion der Schnittstelle *Component* ist es ferner möglich, Inkompatibilitäten festzustellen, die dadurch zustande kommen, dass der Komponente, die für die Verarbeitung einer Anfrage zuständig ist, die genutzte Repräsentationsform nicht bekannt ist.

Auch soll es die Middleware ermöglichen, dem Client verschiedene Repräsentationsformen einer Ressource zur Verfügung zu stellen. Aus diesen Gründen existiert die Möglichkeit mittels Konvertern, eine Repräsentationsform in eine andere zu überführen. Konverter werden mittels der Schnittstelle *Converter* implementiert. Zwei konkrete Ausprägungen dieser Konverter sind Serialisierer und Deserialisierer, mittels derer serialisierte Repräsentationen erstellt bzw. eingelesen werden können. Zur Implementierung dieser speziellen Konverter dienen die Schnittstellen *Serializer* bzw. *Deserializer*.

Die Zusammenführung aller Implementierungen der erläuterten Schnittstelle, im Sinne eines komponentenbasierten Vorgehens, beruhen hierbei auf den Spezifikationen der OSGi.[6]. Diese beschreibt eine auf der Java Virtual Maschine (JVM) basierende und somit hardwareunabhängige Softwareplattform, welche eine Reihe von Funktionen zur Installation und Administration von Softwarekomponenten bereitstellt. Insbesondere beinhaltet diese eine Registrierung zur Bereitstellung von Diensten.

Ein Dienst wird hierbei durch ein gewöhnliches Java-Objekt repräsentiert und mittels einer implementierten Java-Schnittstelle charakterisiert. Dies ermöglicht es, die Softwarekomponenten der Management-Software weitgehend unabhängig von OSGi zu entwerfen und zu implementieren. Lediglich die Registrierung der erstellten Komponenten am OSGi-Framework, welche separiert von der Fachlogik der Komponente realisiert wird, bindet diese an OSGi. Der Kern der in Abschnitt 5 beschriebenen REST-konformen Middleware kann die so registrierten Komponenten zur Laufzeit auffinden und diese so-

[5] Beispielsweise sei eine Implementierung der Schnittstelle Processor gegeben, welche Instanzen der Java-Klasse „Query“ verarbeiten kann und in diesem Zuge Instanzen der Klasse „ResultSet“zurückgibt. Diese Merkmale der Implementierung können zur Laufzeit anhand der Funktionen `getInputRepresentation` und `getOutputRepresentation` ermittelt werden.

[6] Die ursprüngliche Bedeutung von OSGi ist „Open Service Gateway initiative“. Mittlerweile wird das Kürzel aber losgelöst von dieser Bedeutung genutzt.

Funktion	HTTP-Methode	URI
`load`	`GET`	*ressourcenabhängig*
`save`	`PUT`	*ressourcenabhängig*
`delete`	`DELETE`	*ressourcenabhängig*
`create`	`POST`	`/create`
`process`	`POST`	`/process`

Tabelle 5.1.: Abbildung der generischen Schnittstelle auf HTTP

wohl anderen Softwarekomponenten desselben Systems als auch externen Anwendungen mittels eines Konnektors verfügbar machen.

5.5. HTTP-Konnektor

Zur Nutzung der Schnittstelle aus Abb. 5.1 über die Systemgrenzen hinweg wird ein Konnektor (vgl. Abschnitt 3.3.3) spezifiziert. Auf Grundlage der in Abschnitt 5.2 aufgeführten Anforderungen wird dazu HTTP verwendet, da somit die Schnittstelle aus einem Webbrowser heraus angesprochen werden kann. Die Nutzung weiterer Protokolle zu diesem Zweck sind aufgrund des modularen Aufbaus der Middleware aber ebenfalls möglich.

Darüber hinaus bietet sich HTTP aber auch deshalb an, da es auf sehr ähnlichen Konzepten wie REST, beispielsweise der Ressource und deren Repräsentation, basiert. Auch besitzt HTTP zweckmäßige Mechanismen zum Zwischenspeichern von Repräsentationen, wie dies in Abschnitt 3.4.3 beschrieben ist.

Zur Entwicklung einer protokollkonformen Abbildung der zugrundeliegenden Schnittstelle auf HTTP ist insbesondere eine Einteilung der Methoden nach Sicherheit und Idempotenz entscheidend. So besagt die HTTP-Spezifikation, dass sichere Funktionen auf die HTTP-Methode `GET`, idempotente Funktionen hingegen auf die Methode `PUT` abgebildet werden sollen (s.h. [13, S. 52 ff.]). Des Weiteren existiert die Methode `DELETE` zum Löschen von Ressourcen. Funktionsaufrufe, die darüber hinaus weder sicher noch idempotent sind, sollen mittels der Methode `POST` übertragen werden.

Die Abbildung der Funktionen der generischen Schnittstelle auf die durch HTTP definierten Methoden wird in Tabelle 5.1 aufgelistet. Aus den aufgeführten Anforderungen von HTTP lässt sich die Abbildung der Funktionen zu einem großen Teil direkt erschließen. Die Abbildung der Funktionen `create` und `process` bedarf jedoch einer näheren

Betrachtung. Beide Funktionen sind weder sicher noch idempotent, was bedingt, dass nur die HTTP-Methode `POST` zur Übertragung infrage kommt. Die Auswahl einer der beiden Funktionen durch einen HTTP-Aufruf geschieht mittels der in Tabelle 5.1 aufgeführten URI. Somit ist zum einen eine für den Client einfach zu implementierende Anbindung an den Server möglich, und zum anderen ist gewährleistet, dass der Konnektor HTTP-Anfragen eindeutig auf den entsprechenden Funktionsaufruf zurückführen kann.

5.5.1. Datenspeicher

Wenngleich die Art und Weise der Speicherung von Daten durch die Middleware nicht fest vorgegeben wird, so stellt diese zu diesem Zweck doch eine entsprechende Implementierung der Schnittstelle Resource, die auf dem *Resource Description Framework* (RDF) (s.h. [9] und [20])basiert, bereit. [7] RDF umfasst eine Reihe von Standards des World Wide Web Consortiums (W3C) zur formalen Beschreibung von Informationen über Ressourcen und ist wesentlicher Bestandteil der Aktivitäten des W3C rund um das semantische Web (s.h. [7]).

Der Einsatz von RDF im Rahmen dieser Arbeit ist zum einen darin begründet, dass RDF ein einfaches und für Wissensrepräsentationen zweckmäßiges Metamodell nutzt, welches alle Informationen durch eine Menge von dreielementigen Aussagen, sogenannten RDF-Triplen, repräsentiert. Ein solches Tripel besteht aus einer URI, die die Ressource, auf die sich die Aussage bezieht, identifiziert, einem Prädikat, welches eine Eigenschaft der Ressource spezifiziert und einem Eigenschaftswert. [8] Da Eigenschaftswerte neben einfachen literalen Typen, wie z.B. Zeichenketten oder Ganzzahlen, auch URIs umfassen kann auf andere Ressourcen verwiesen werden. Dadurch lassen sich komplexe gerichtete Graphen abbilden.

Somit ist es trotz des einfachen Metamodells von RDF i.d.R. möglich, Modelle, die auf gänzlich anderen Metamodellen basieren, abzubilden. Beispielsweise können so sowohl Informationen, die in einem objektorientierten Modell ihren Ursprung haben, als auch

[7] Die genutzte Open Source Java-Implementierung namens Jena wurde von dem Unternehmen Hewlett Packard erstellt.

[8] Repräsentiere beispielsweise die Ressource http://beispiel.org/Benutzer/HSchwartke einen Benutzer der den Vornamen „Hendrik“ und den Nachnamen „Schwartke“ trägt, so lassen sich diese Eigenschaften durch die beiden folgenden Aussagen, welche in der Notation 3 (s.h. [6]) dargestellt sind, ausdrücken:
<http://beispiel.org/Benutzer/HSchwartke> <urn:beispiel.org:Vorname> „Hendrik“ .
<http://beispiel.org/Benutzer/HSchwartke> <urn:beispiel.org:Nachname> „Schwartke“ .

hierarchisch strukturierte Daten leicht abgebildet werden.

Ein anderer Vorteil von RDF besteht darin, dass durch die *SPARQL Protocol and RDF Query Language* (SPARQL) (s.h. [31]) eine für RDF-Modelle standardisierte und vielfältige Abfragesprache existiert. Somit können komplexe Abfragen auf Basis von Ressourcen einfach und prägnant formuliert werden.

Darüber hinaus gibt es eine Reihe von Repräsentationsformen für RDF-Modelle, die vom W3C standardisiert sind, um einen Informationsaustausch zwischen verschiedenen, auf RDF basierenden Systemen zu vereinfachen. So definiert [3] beispielsweise eine Abbildung von RDF-Modellen auf XML.

Darüber hinaus ermöglicht RDF aufgrund des einfachen und generischen Metamodells einen sehr dynamischen Umgang mit verschiedenartigen Ressourcen. So muss nicht, wie bei typischen relationalen Datenbanken zur Entwicklungszeit ein Datenmodell erstellt werden, von dem dann zur Laufzeit nicht mehr ohne Weiteres abgewichen werden kann. Das Fehlen der Notwendigkeit eines Schemas, dass über die RDF-Tripel hinausgeht, ermöglicht daher eine hohe Flexibilität der Anwendung. [9]

Ein Nachteil von RDF besteht im Umgang mit nicht strukturierten Daten. Beispielsweise ist die Speicherung von Benutzerzertifikaten, die im Rahmen dieser Arbeit lediglich als unstrukturierte binäre Bytefolge angesehen werden, in RDF i.d.R. nicht sinnvoll. Aus diesem Grund wird zur Speicherung solcher binärer Daten eine zweite Implementierung der Schnittstelle Resource und ResourceFactory genutzt, welche Ressourcen direkt im Dateisystem speichert.

5.6. Zusammenfassung

Die erstellte Middleware ermöglicht es, eine REST-konforme Software modular auf einfache Art und Weise zu entwickeln. Zur Umsetzung der mit REST verbundenen Elemente (vgl. Abschnitt 3.3) sind die folgenden Schnittstellen bestimmt:

Resource Die Klasse *Resource* dient zur Implementierung einer Ressource im Sinne von REST (vgl. Abschnitt 3.3.1).

Processor Durch die Klasse *Processor* kann Logik zur Verarbeitung von Repräsentationen spezifiziert werden.

[9] Letzte Eigenschaft ist insbesondere im Kontext des Semantic Webs von besonderer Bedeutung. RDF macht keine „Closed-World-Assumption“, geht also von vornherein nicht von einem in sich geschlossenen System aus, sondern nimmt an, dass Ressourcen und die mit ihnen verbundenen Informationen, ihren Ursprung in verschiedenen verteilten und lose gekoppelten Systemen haben.

Representation Mittels der Klasse *Representation* können Repräsentationen (vgl. Abschnitt 3.3.2 implementiert werden. Die Middleware lässt sich somit einfach um Darstellungsformen erweitern.

Converter Anhand der Klasse *Converter* ist es möglich, Konverter zu erstellen, die verschiedene Repräsentationsformen ineinander umwandeln.

Darüber hinaus wird die Anforderung nach einer einheitlichen Schnittstelle durch die *Server*, dessen Implementierung die Instanzen der oben genannten Schnittstellen verwaltet, umgesetzt. Schließlich ist diese Schnittstelle durch einen Konnektor mittels HTTP aufrufbar.

6. Modellierung semantischer Elemente

6.1. Einleitung

Während im vorhergehenden Kapitel auf die Umsetzung technischer Aspekte einer REST-konformen Anwendung eingegangen wird, wird in diesem Kapitel beschrieben, wie die semantischen Elemente der Schnittstelle, im Sinne des in Abschnitt 2.2 erläuterten Schichtenmodells, spezifiziert werden. Zu diesem Zweck wird auf einen modellgetriebenen Entwicklungsentwurf zurückgegriffen und in Kapitel 6.3.1 ein Metamodell zur Beschreibung von OpenVPN-Parametern erstellt. Ferner wird in Kapitel 6.3.2 gezeigt, wie auf Grundlage dieser Beschreibung und eines entsprechenden Metamodells, Formulare für einen graphischen Client mittels Transformationen generiert werden. Es wird also an diesem Beispiel demonstriert, wie sementische Elemente der Softwareschnittstelle mittels eines Modell beschrieben werden können und wie dieses verwendet werden kann, um darauf aufbauend weitere Softwarekomponenten zu erstellen. Die zu diesem Zweck verwendeten Technologien werden im folgenden Abschnitt erläutert.

6.2. Begriffsbildung

6.2.1. Ecore

Ecore (EMF-Core) ist ein, streng typisierte, objektorientierte Metamodell, welches integraler Bestandteil der Entwicklungsumgebung Eclipse ist. Der primäre Zweck von Ecore ist die Definition von Metamodellen zur Beschreibung von Software.

Um die Erstellung von Metamodellen möglichst einfach zu gestalten, besteht Ecore nur aus einer geringen Anzahl grundlegender Elemente, die zum überwiegenden Anteil der objektorientierten Modellierung struktureller Eigenschaften eines Metamodells dienen. In Abb. 6.1 ist eine Übersicht der wesentlichen Elemente dargestellt.

Im Mittelpunkt von Ecore steht die Klasse *EClass* welche dazu dient, eine Klasse im Sinne von EMF zu spezifizieren. Einer solchen Klasse können mittels einer Instanz der Klasse *EStructuralFeature* strukturelle Eigenschaften – konkret können dies entweder

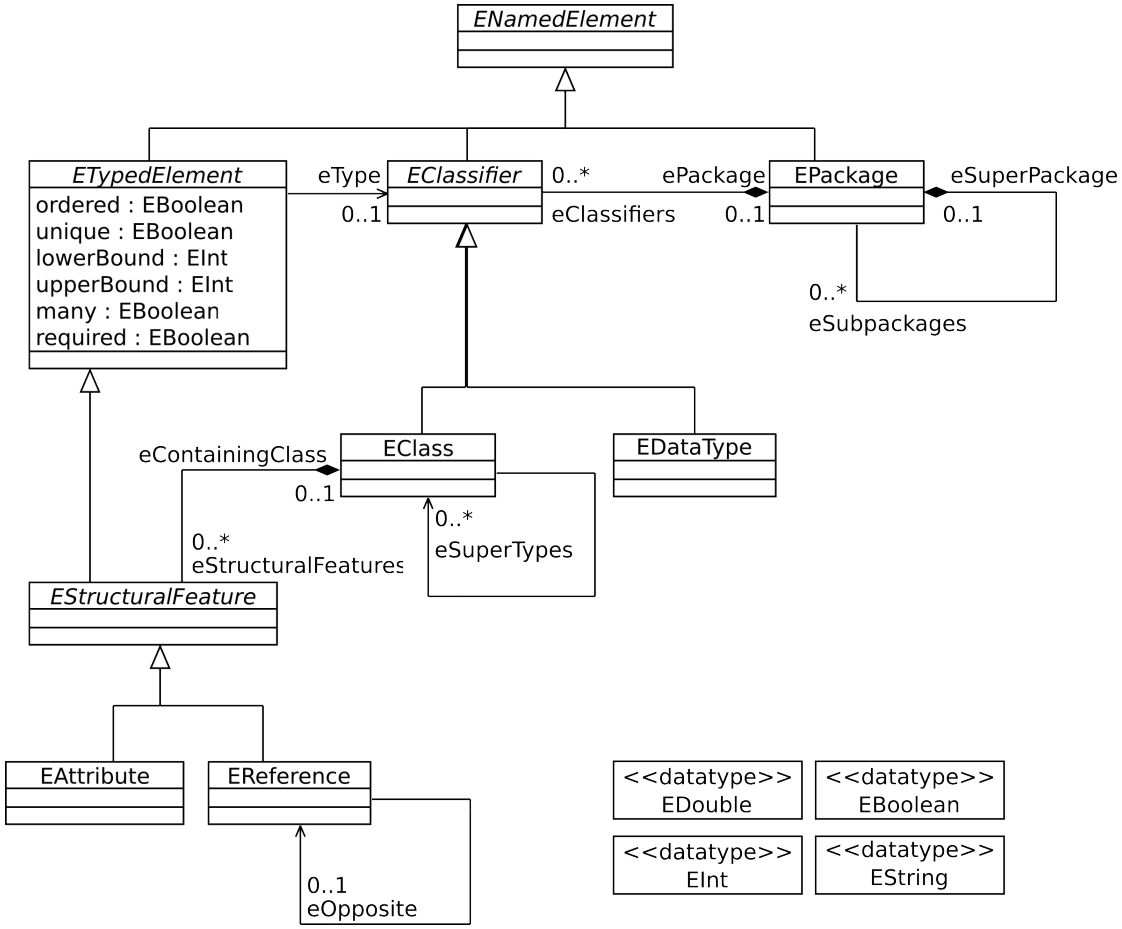

Abbildung 6.1.: Ecore-Metamodell (auszugsweise)

Attribute oder Referenzen sein – zugewiesen werden. Attribute, welche durch die Klasse *EAttribute* beschrieben werden, werden durch einen primitiven Datentypen charakterisiert. Referenzen, welche mittels der Klasse *EReference* spezifiziert werden, verweisen hingegen auf eine andere Instanz der Klasse EClass. Ferner lassen sich die Modellelemente mittels der Klasse *EPackage* auf verschiedene Pakete verteilen und sich somit auch komplexe Modelle sinnvoll strukturieren. Eine vollständige Beschreibung aller Elemente gibt [34] und [1]. Des Weiteren ist Ecore selbstbeschreibend, d.h. alle Elemente des Modells können durch eben diese Modelle beschrieben werden.

6.2.2. xText

Um Modellen eine konkrete Syntax zuweisen zu können, und somit domänenspezifische Sprachen zu definieren, dient xText, das ebenfalls Bestandteil von Eclipse ist. Zur Beschreibung von DSLs nutzt xText eine Art der erweiterten Backus-Normalformen (EBNF) (s.h. [21]) und setzt somit auf einen erprobten Ansatz zur Spezifikation von konkreten Syntaxen.

Im Unterschied zu vielen üblichen Parsergeneratoren endet die Verarbeitung einer mittels xText spezifizierten formalen Sprache aber nicht mit der Erstellung eines Parserbaumes. Stattdessen werden die eingelesenen Strukturen verwendet, um automatisch ein auf dem Ecore-Metamodell basierendes Modell zu generieren, welches kompatibel zu einem auf Ecore basierenden Metamodell ist, das die abstrakte Syntax der Sprache beschreibt. Dieses Metamodell wird entweder separat erstellt oder unmittelbar aus der Beschreibung der DSL abgeleitet. Somit ermöglicht es xText, sowohl bestehenden Metamodellen konkrete Syntaxen zuzuordnen, als auch die Erstellung gänzlich neuer domänenspezifischer Sprachen.

Zu den Werkzeugen die xText anhand einer solchen Syntaxbeschreibung erstellt, gehören neben dem Parser auch ein auf Eclipse basierender umfangreicher und zweckmäßiger Editor, der u.a. eine auf die DSL zugeschnittene Syntaxhervorhebung, Autovervollständigung und die Möglichkeit der automatisierten Formatierung des Quellcodes bietet. Somit kann mit einer mittels xText erstellten DSL ähnlich komfortabel gearbeitet werden, wie dies bei klassischen Programmiersprachen üblich ist.

Da sich der erstellte Parser nahtlos in die Architektur von EMF integriert, kann die DSL ferner von allen mit EMF-kompatiblen Werkzeugen eingelesen werden. Für eine weitergehende Betrachtung von xText sei auf [1] verwiesen.

6.2.3. Query View Transformation (QVT)

Query View Transformation (QVT) ist eine von der Object Management Group (OMG) ausgearbeitete Spezifikation die Technologien zur Abfrage, Filterung und Transformation von Modellen im Kontext von MDSD beschreibt. QVT besteht aus den drei Sprachen QVT Core (QVTC), QVT Relational (QVTR) und QVT Operational (QVTO), welche allesamt als *Modell-zu-Modell-Transformationssprachen* eingesetzt werden können, zu diesem Zweck jedoch verschiedene Ansätze unterstützen. Da weder QVTC noch QVTR im Rahmen dieser Arbeit genutzt werden, werden diese hier nicht näher erläutert und stattdessen auf [25] verwiesen.

QVTO wird im Rahmen dieser Arbeit als Modell-zu-Modell-Transformationssprache genutzt und ist eine imperative, auf der Object Constraint Language (OCL) (s.h. [26]) basierende Sprache. Mittels OCL können Ausdrücke zur Selektion und Abfrage von Modellelementen spezifiziert und in die imperativen Sprachelemente von QVTO eingebettet werden. Zentrales Sprachelement ist hierbei das Mapping-Konstrukt, welches dazu dient, Bestandteile eines Eingabemodels in Elemente eines Ausgabemodells zu überführen. [1] Eine Implementierung von QVTO ist Bestandteil von Eclipse.

6.3. Modellierung

6.3.1. OpenVPN-Konfigurations-Metamodell

Ein zentraler Aspekt der VPN-Management-Software ist die Verarbeitung von OpenVPN-Konfigurationen. Da zur Konfiguration der OpenVPN-Software eine große Anzahl von Parametern zur Verfügung stehen, die Version 2.1 spezifiziert über zweihundert solcher Parameter, die in [29] dokumentiert sind, werden diese im Sinne der modellgetriebenen Softwareentwicklung mittels eines formalen Modells beschrieben. Somit wird gewährleistet, dass trotz der damit verbundenen großen Menge an Informationen, diese übersichtlich, klar strukturiert und unabhängig von anderen Aspekten der Software dargestellt werden können. [2]

[1] Hierbei sei angemerkt, dass das Mapping-Konstrukt tatsächlich wesentlich vielseitiger ist und auch komplexere Abbildungsszenarien unterstützt.

[2] In einem klassischen objektorientierten Ansatz würden die Parameter und die mit ihnen verbundenen Merkmale, evtl. nach vorausgegangener Skizzierung des Entwurfs in Form von Diagrammen, direkt durch Softwareentwickler in Form von Quellcode abgebildet. Dies bedeutet nicht selten eine häufige manuelle Wiederholung ähnlicher Tätigkeiten und führt zu einer Vermengung mit nicht fachlichen Aspekten, wie die Wahl der Programmiersprache oder der Softwareplattform. Insbesondere sind Entscheidungen über die Art und Weise, mittels derer die Anforderungen auf Eigenschaften und Fä-

```
1  mode p2p
2  proto udp
3  port 1194
4  dev tun
5  ifconfig 10.4.0.1 10.4.0.2
6  verb 3
7  secret static.key
8  keepalive 10 60
9  persist-tun
10 persist-key
11 persist-local-ip
12 comp-lzo
```

Listing 6.1: OpenVPN-Konfigurationsbeispiel

Listing 6.1 zeigt beispielhaft eine solche Konfiguration in Form der von OpenVPN vorgegebenen Syntax. Jede Zeile in diesem Beispiel beinhaltet genau eine Parameterspezifikation und beginnt jeweils mit dem Namen des beschriebenen Parameters. Hierauf kann je nach Parameter ein einzelner Wert oder eine Reihe von Werten folgen, wobei im letzteren Fall die Bedeutung eines Wertes durch seine Position in der Folge bestimmt wird. [3]

Zur formalen Beschreibung aller Parameter mittels derer OpenVPN konfiguriert werden kann wird ein Modell auf Basis von Ecore (vgl. Abschnitt 6.2.1) erstellt, das in Form eines UML-Klassendiagramms in Abb 6.2 dargestellt wird. Dessen Elemente werden im Folgenden erläutert:

higkeiten einer gewählten Softwaretechnologie abgebildet werden, im Nachhinein schwer revidierbar, da hiermit oftmals ein erheblicher manuell durchzuführender Arbeitsaufwand einhergehen würde. Die Wartbarkeit wäre auch dadurch erschwert, dass bei Erweiterungen der Parameterbeschreibung um weitere Merkmale womöglich ein Großteil oder gar alle zugrundeliegenden Codefragmente manuell angepasst werden müssten.

[3] Die Zeile 5 des Listings 6.1 spezifiziert z.B. mittels des ersten Parameterwertes „10.4.0.1“ eine IP-Adresse, die sich am lokalen Tunnelende und durch den zweiten Wert „10.4.0.2“ eine IP-Adresse, die sich am entfernten Tunnelende befindet.

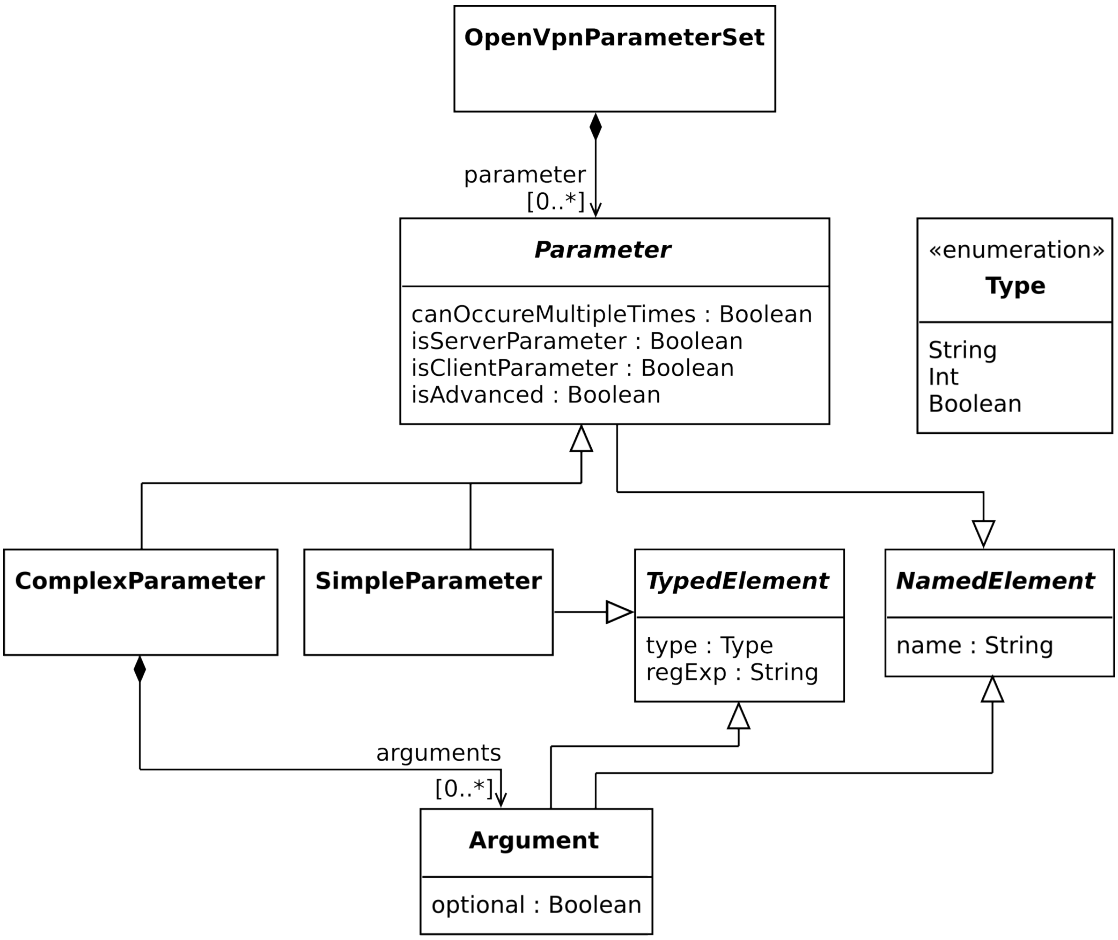

Abbildung 6.2.: OpenVPN-Konfigurations-Metamodell

OpenVpnParameterSet

Beschreibung	Die Klasse OpenVpnParameterSet dient zur Beschreibung von OpenVPN-Parametern und hält alle Parameterbeschreibungen.
Elternklassen	-
Attribute	`parameter` speichert die Menge aller Parameterbeschreibungen

Parameter

Beschreibung	Die Klasse Parameter beschreibt einen OpenVPN-Konfigurationsparameter.
Elternklassen	NamedElement
Attribute	`isServerParameter` gibt an, ob der Parameter zur Konfiguration eines *OpenVPN-Servers* dient. `isClientParameter` gibt an, ob der Parameter zur Konfiguration eines *OpenVPN-Clients* dient. `isAdvanced` gibt an, ob der Parameter nur fortgeschrittenen Administratoren zugänglich gemacht werden soll. `canOccurMultipleTimes` gibt an, ob ein Parameter mehrfach in derselben Konfiguration angegeben werden darf.
Bemerkungen	`isServerParamter` oder `isClientParameter` muss auf wahr gesetzt sein.

SimpleParameter

Beschreibung	Die Klasse SimpleParameter dient zur Beschreibung eines einfach strukturierten, d.h. nur aus einem einzelnen Wert bestehenden Parameters.
Elternklassen	Parameter, TypedElement

ComplexParameter

Beschreibung	Die Klasse ComplexParameter dient zur Spezifizierung eines komplex struktirierten, d.h. aus mehreren Argumenten zusammengesetzten, Parameters.

Elternklassen Parameter

Attribute `arguments` hält die geordnete Menge der Argumente des Parameters

Argument

Beschreibung Die Klasse Argument dient zur Spezifizierung eines einzelnen Argumentes eines komplex strukturierten Parameters.

Elternklassen NamedElement, TypedElement

Attribute `isOptional` spezifiziert, ob ein Argument zwingend angegeben werden muss.

TypedElement

Beschreibung Die Klasse TypedElement dient zur Modellierung von Objekten, welchen ein Typ zugeordnet ist.

Elternklassen -

Attribute `type` gibt den Typ des Elements an.

`regExp` speichert einen regulären Ausdruck, welcher die Menge gültiger Werte über den spezifizierten Typen hiaus einschränkt. Ist kein solcher Ausdruck angegeben, ist der Wertebereich des Elementes ausschließlich durch den Typen spezifiziert.

Bemerkungen Das Attribut regExp wird in dem Fall das der Type Boolean ist ignoriert.

NamedElement

Beschreibung Die Klasse NamedElement dient zur Modellierung von Objekten, welche anhand ihres Namens identifiziert werden können.

Elternklassen -

Attribute `name` speichert den Namen des Objektes

Type

Beschreibung	Type ist ein Aufzählungstyp zur Spezifizierung der Datentypen, die in einer OpenVPN-Konfiguration zur Verfügung stehen. Die erlaubten Werte sind folgende:
String	beliebige Zeichenfolge
Int	Ganzzahl
Boolean	Wahrheitswert, also entweder *wahr* oder *falsch*

Um ein auf dem erläuterten Metamodell basierendes Modell übersichtlich darzustellen und einfach modifizieren zu können, wird mittels xText eine textuelle DSL erstellt, dessen Spezifikation in Listing 6.2 aufgeführt ist und hier anhand eines Auszugs des Konfigurationsmodells, welches in Abbildung 6.3 dargestellt ist, erläutert wird.

In der Abbildung werden die acht Konfigurationsparameter `route`, `log`, `user`, `max-clients`, `username-as-common-name`, `connect-retry`, `connect-retry-max` und `ifconfig` spezifiziert. [4] Jede Parameterspezifikation wird mit dem eindeutigen Namen des Parameters eingeleitet, welchem eine Reihe von Markern folgen kann. Diese entsprechen den boolschen Attributen des Metamodellelements Parameter, welche bei Vorhandensein eines entsprechenden Markers auf wahr, ansonsten auf falsch gesetzt werden. Hierbei sind die Marker *, a, c, s in dieser Reihenfolge den Metamodellattributen `canOccureMultipleTimes`, `isAdvanced`, `isClientParameter` und `isServerParameter` zugeordnet. Handelt es sich bei dem jeweiligen Parameter um einen einfach strukturierten, so folgt dessen Typ und optional ein regulärer Ausdruck.

Handelt es sich hingegen um einen komplex strukturierten Parameter, so werden sukzessive dessen Argumente mit Namen, Typ und optional einem regulären Ausdruck aufgeführt. Optionalen Argumenten folgt ein Fragezeichen hinter dessen Namen. Abgeschlossen wird die Spezifizierung eines Parameters mit einem Semikolon.

Das in Abbildung 6.3 dargestellte Modell beschreibt folglich fünf einfach und zwei komplex strukturierte Parameter.

Des Weiteren dienen alle Parameter, mit Ausnahme des Parameters `max-clients` zur Konfiguration eines OpenVPN-Clients, was durch dem Marker *c* spezifiziert wird. Die Parameter `log`, `max-clients` und `ifconfig` können, erkennbar an dem Marker *s*, auch zur Konfiguration eines OpenVPN-Servers eingesetzt werden. Des Weiteren sind alle Parameter, mit Ausnahme der Parameter `user`, `username-as-common-name` und `route` mittels

[4] Die genaue Bedeutung aller Parameter wird in [29] beschrieben.

```
1  OpenVpnParameterSet:      (parameter+=Parameter)*;
2  SimpleParameter:
3          name=NAME canOccurMultipleTimes?='*'?
4          (
5                    canOccurInConnectionBlock?='b'
6                  | advanced?='a'
7                  | serverParameter?='s'
8                  | clientParameter?='c'
9          )*
10         type=Type regExp=REGEXP? ';'
11 ;
12 ComplexParameter:
13         name=NAME canOccurMultipleTimes?='*'?
14         (
15                   canOccurInConnectionBlock?='b'
16                 | advanced?='a'
17                 | serverParameter?='s'
18                 | clientParameter?='c'
19         )*
20         arguments+=Argument* ';'
21 ;
22 Argument:
23         name=NAME optional?='?'? type=Type?
24         (regExp=REGEXP|enumeration=Enumeration)?
25 ;
26 Enumeration:      '[' value+=NAME (',' value+=NAME)* ']';
27
28 terminal WS:          (' '|'\t'|'\r'|'\n')+;
29 terminal NAME:        ('a'..'z'|'A'..'Z'|'_'|'-'|'0'..'9')*;
30 terminal COMMENT:     '#' !('\n'|'\r')* ('\r'? '\n')?;
31 terminal REGEXP:      '/' !('/')* '/';
```

Listing 6.2: OpenVPN-Parameter-DSL

```
1  log                          a c s   String;
2
3  user                           c     String   /[0-9a-zA-Z]+/;
4  username-as-common-name        c     Boolean;
5
6  connect-retry                a c     Int;
7  connect-retry-max            a c     Int;
8
9  max-clients                  a   s   Int;
10
11 route*                         c
12         networkIP                    String   /([0-9]{3}\.){3}[0-9]{3}/
13         netmask?                     String   /([0-9]{3}\.){3}[0-9]{3}/
14         gateway?                     String   /([0-9]{3}\.){3}[0-9]{3}/
15         metric?                      Int
16 ;
17
18 ifconfig*                      c
19         local                        String   /([0-9]{3}\.){3}[0-9]{3}/
20         remote                       String   /([0-9]{3}\.){3}[0-9]{3}/
21 ;
```

Listing 6.3: OpenVPN-Parameterspezifikation (auszugsweise)

des Markers *a* als „advanced“ gekennzeichnet und sollen somit nur fortgeschrittenen Adminstratoren zugänglich gemacht werden.

6.3.2. Formular-Metamodell

Ein weiterer wesentlicher Aspekt der Management-Software ist ein graphischer Client, mittels dessen alle Funktionen, die der Server bereitstellt, genutzt werden können. Um dies zu ermöglichen, werden Formulare, mittels derer der Administrator auf einfache Art und Weise verschiedene Entitäten erzeugen, einsehen und modifizieren kann, erstellt.

Insbesondere auf Grund eines in der Praxis häufigen Wechsels von Technologien, die die Basis eines solchen Clients bilden, wird ein technologieunabhängiges Modell konzipiert. Hierdurch wird auch die Erstellung alternativer Clients vereinfacht.

Das zur Modellierung von Formularen zugrundeliegende Metamodell ist an Abb. 6.3 in Form eines UML-Klassendiagramms dargestellt und dessen Elemente werden im Folgenden näher erläutert:

Formular

Beschreibung	Die Klasse Formular dient zur Beschreibung eines Formulars.
Elternklassen	Group

Group

Beschreibung	Die Klasse Group dient zur Gruppierung mehrerer graphischer Elemente.
Attribute	`elements` speichert die Elemente, welche durch die Gruppe zusammengefasst werden.
Bemerkungen	Die Gruppierung der enthaltenen Elemente muss in der graphischen Benutzeroberfläche, z.B. durch gemeinsame Umrandung, klar erkennbar sein.

LabeledElement

Beschreibung	Die Klasse LabeledElement dient zur Beschreibung eines graphischen Elementes, dessen Zweck durch eine kurze, prägnante Beschriftung erläutert wird.

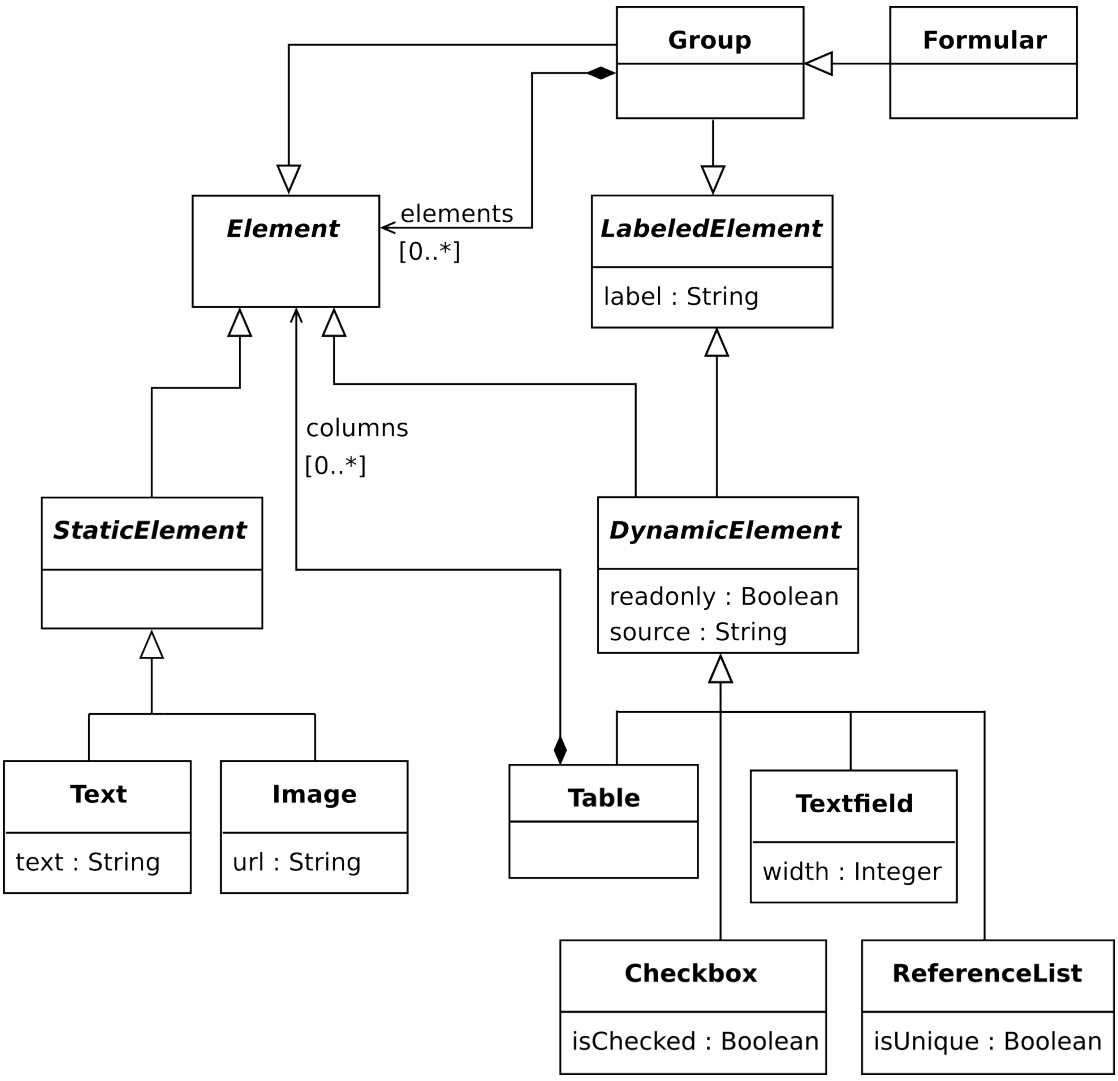

Abbildung 6.3.: Formular-Metamodell

Elternklassen -

Attribute `label` speichert die Beschriftung des Elementes

Element

Beschreibung Die Klasse Element dient zur Beschreibung eines graphischen Elementes.

Elternklassen -

Attribute -

StaticElement

Beschreibung Die Klasse StaticElement dient zur Beschreibung statischer, d.h. nicht modifizierbarer graphischer Elemente.

Elternklassen Element

Text

Beschreibung Die Klasse Text dient zur Beschreibung eines statischen Textes.

Elternklasse StaticElement

Attribute `text` gibt den anzuzeigenden Text an.

Image

Beschreibung Die Klasse Image dient zur Beschreibung einer graphischen Abbildung.

Elternklasse StaticElement

Attribute `url` speichert die URL der darzustellenden Abbildung.

DynamicElement

Beschreibung Die Klasse DynamicElement dient zur Beschreibung von dynamischen graphischen Elementen, also solchen, dessen zugrundeliegender Wert erst zur Laufzeit ermittelt wird und welcher vom Benutzer modifiziert werden kann.

Elternklassen Element

Attribute `readonly` gibt an, ob der dem Element zugrundeliegende Wert modifiziert werden kann. Ist das Attribut auf falsch gesetzt oder nicht angegeben, so kann dieser Wert modifiziert werden, ansonsten ist dieser schreibgeschützt.

`source` gibt einen eindeutigen Namen des zugrundeliegenden Wertes, der durch das Element graphisch dargestellt werden soll, an.

Textfield

Beschreibung Die Klasse Textfield dient zur Beschreibung eines Textfeldes.

Elternklasse DynamicElement

Attribute `width` gibt die Breite des Textfeldes an.

Checkbox

Beschreibung Die Klasse Checkbox dient zur Beschreibung einer Checkbox.

Elternklasse DynamicElement

Attribute `isChecked` gibt an ob die Checkbox markiert werden soll, sofern dem Element kein Wert zugrundeliegt.

ReferenceList

Beschreibung Die Klasse ReferenceList dient zur Beschreibung einer Liste von Verweisen auf andere Elemente.

Elternklasse DynamicElement

Attribute `isUnique` gibt an ob Verweise zu einem Objekt mehrfach in der Liste vorkommen dürfen. Hat das Attribut den Wert true, so ist eine mehrfaches Vorkommen des selben Elementes nicht erlaubt.

Table

Beschreibung Die Klasse Table dient zur Beschreibung einer tabellarischen Auflistung mehrerer Werte.

Elternklasse	DynamicElement
Attribute	`columns` spezifiziert eine geordnete Menge von Elementen die als Spalten der Tabelle dienen.

6.4. Transformation

Im Folgenden wird verdeutlicht wie mithilfe der Transformation aus Listing, die in QVTO verfasst ist, 6.4 auf Grundlage der OpenVPN-Parameterbeschreibung, ein Modell einer graphischen Oberfläche generiert wird. Hierzu werden eine Reihe von Teiltransformationen erstellt. Ausgangspunkt der Transformation ist die Abbildung `toForm` die auf Basis eines Objektes vom Type OpenVpnParameterSet ein Objekt vom Type Formular erzeugt. Diese Abbildungsvorschrift bedient sich der Abbildung `toDynamicElement`, um aus allen OpenVPN-Parametern Objekte vom Type DynamicElement zu erzeugen. Hierzu wird wiederum an eine der Abbildungen `toTextfield`, `toCheckbox` oder `toTable` deligiert, die je nach Typ des OpenVPN-Parameters ein Textfeld, eine Checkbox oder eine Tabelle erstellen.

```
1  //=============================================================
2  // Namensraum der fuer die eindeutige Benennung alle Parameter
3  // genutzt wird.
4  //=============================================================
5  property targetNamespace :=
6    "http://www.opensource-training.de/openvpn/config/";
7
8  //=============================================================
9  // Standardbreite eines Testfeldes.
10 //=============================================================
11 property defaultTextfieldWidth := 50;
12
13 //=============================================================
14 // Abbildung der Menge aller Konfigurationsoptionen auf ein
15 // Formular.
16 //=============================================================
17 mapping OpenVpnParameterSet::toForm() : Formular {
18         label     := "OpenVPN-Konfiguration";
```

```
19          elements := self.parameter->map toDynamicElement();
20  }
21
22  //=============================================================
23  // Abbildung eines Parameters auf ein dynamisches Element.
24  // Hierzu wird entweder an die Abbildung toTable, toCheckbox
25  // oder toTextfield delegiert.
26  //=============================================================
27  mapping Parameter::toDynamicElement() : DynamicElement
28  disjuncts ComplexParameter::toTable,
29            TypedElement::toChechbox,
30            TypedElement::toTextfield {}
31
32  //=============================================================
33  // Abbildung eines Elementes dem ein Typ zugeordnet ist auf
34  // eine Checkbox oder auf ein Textfeld.
35  // Hierzu wird entweder an die Abbildung toCheckbox oder
36  // toTextfield delegiert.
37  //=============================================================
38  mapping TypedElement::toDynamicElement() : DynamicElement
39  disjuncts TypedElement::toChechbox,
40            TypedElement::toTextfield {}
41
42  //=============================================================
43  // Abbildung eines Elementes dem ein Typ zugeordnet ist auf
44  // eine Checkbox.
45  //=============================================================
46  mapping TypedElement::toCheckbox() : Checkbox {
47  when{self.type = Checkbox}
48  {
49      label  := self.name;
50      source := self.getQualifiedName();
51  }
52
53  //=============================================================
```

```
54 // Abbildung eines Elementes dem ein Typ zugeordnet ist auf
55 // ein Textfeld.
56 //=================================================================
57 mapping TypedElement::toTextfield() : Textfield {
58 when{self.type = Textfield}
59 {
60     label  := self.name;
61     source := self.getQualifiedName();
62     width  := defaultTextfieldWidth;
63 }
64
65 //=================================================================
66 // Abbildung eines komplex strukturierten Parameters auf eine
67 // Tabelle.
68 //=================================================================
69 mapping ComplexParameter::toTable() : Table {
70     label   := self.name;
71     source  := self.getQualifiedName();
72     columns := self.arguments->map toDynamicElement();
73 }
74
75
76 //=================================================================
77 // Hilfsfunktion zur Ermittlung eines eindeutigen Namens fuer
78 // ein benanntes Element.
79 //=================================================================
80 query NamedElement::getQualifiedName() : String {
81         return targetNamespace + self.name;
82 }
```

Listing 6.4: Transformation

7. Anwendungsbeispiel

Abschließend soll anhand eines Beispiels die Funktionsweise der Schnittstelle der VPN-Management-Software und das Zusammenwirken mit dem Client verdeutlicht werden. Hierzu wird die Kommunikation zwischen Client und Server, als auch die interne Verarbeitung seitens des Servers erläutert.

Die Interaktion nutzt den in Abschnitt 5.5 erläuterten Konnektor und nutzt somit HTTP auf der Transferschicht. Die HTTP-Spezifika werden in diesem Beispiel aber nur auszugsweise und vereinfacht aufgeführt. Insbesondere werden Verfahren zur Authentifikation des Benutzers vernachlässigt. Ferner wird auf der Strukturschicht, mit einigen Ausnahmen, XML eingesetzt. Des Weiteren werden die in Abschnitt 6.3.1 modellierten Elemente zur Beschreibung von OpenVPN-Konfigurationen genutzt, welche im Namensraum `http://www.opensource-training.de/openvpn/config/` zusammengefasst sind. Neben dieser spezialisierten Beschreibungssprache wird die XML Linking Language (s.h. [12]) des W3C genutzt, um Ressourcen untereinander zu verknüpfen. Der zugehörige Namensraum ist `http://www.w3.org/1999/xlink`.

Nach dem Start des Clients ruft dieser, mittels einer ihm bekannten URI, die Indexressource der VPM-Management-Software auf.

```
1 GET http://server.de/index
2 Accept: text/xml
```

Die Verarbeitung dieser Anfrage ist in Abb. 7.1 in Form eines UML-Sequenzdiagramms dargestellt. Der HTTP-Konnektor empfängt die Anfrage und delegiert den Aufruf an die Funktion `load` der Serverschnittstelle (vgl. Abschnitt 5.3). Die Implementierung dieser Schnittstelle ermittelt nun das Objekt, welches die Ressource mit der angegebenen URI darstellt (vgl. Abschnitt 5.4). Daraufhin wird mittels der Funktion `load` der entsprechenden Ressourcenimplementierung eine Repräsentation der Ressource ausgelesen. Da die Repräsentation der Ressource in diesem Fall als RDF-Graph und nicht, wie vom Client gefordert, serialisiert in Form von XML vorliegt, wird ein entsprechender Konverter genutzt, um eine Umwandlung durchzuführen. Im Anschluss daran wird die serialisierte

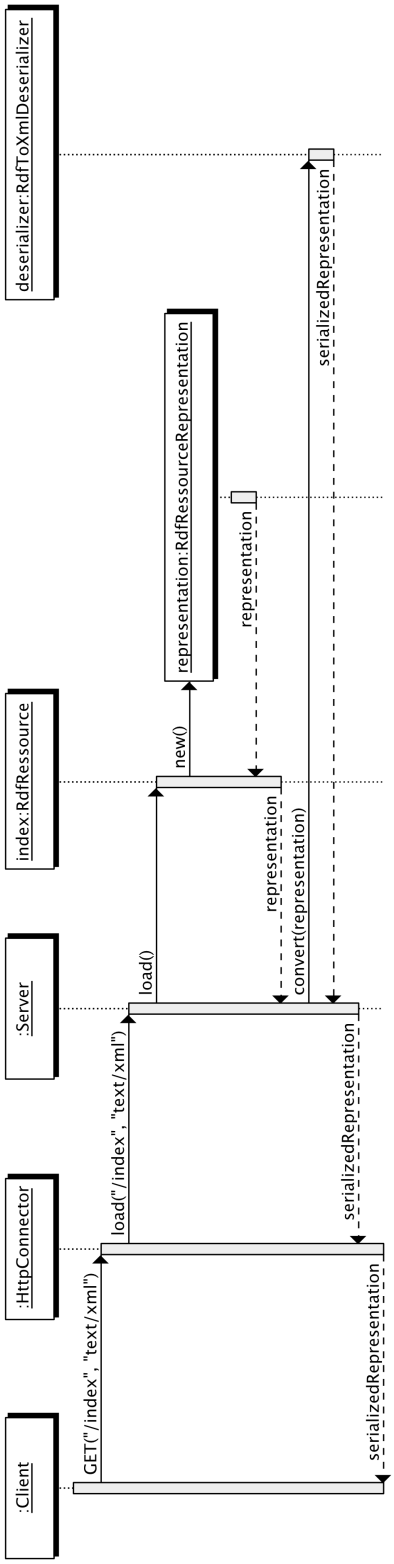

Abbildung 7.1.: Ladevorgang der Indexressource

Repräsentation, welche im Folgenden aufgeführt ist, an den Client übermittelt.

```
<?xml version="1.0" encoding="UTF-8"?>
<index xml:base="http://server.de/"
  xmlns="http://www.opensource-training.de/openvpn/server"
  xmlns:conf="http://www.opensource-training.de/openvpn/config/"
  xmlns:xlink="http://www.w3.org/1999/xlink">
<users>
  <link xlink:title="Hendrik Schwartke"
        xlink:href="res/1f731b0d-47f2-44ab-b4d3-1f9db5f84a0c"/>
  <link xlink:title="Ralf Spenneberg"
        xlink:href="res/b3341d6f-d1c8-4e9f-a7f0-6e24c792abf1"/>
  <link xlink:title="Max Mustermann"
        xlink:href="res/81db8aa7-fcd3-47d5-87d8-767a6d47c02c"/>
</users>
<groups>
  <link xlink:title="Administratoren"
        xlink:href="res/c116ed07-6db2-490f-a835-237185227ec6"/>
  <link xlink:title="Heimarbeiter"
        xlink:href="res/6d10086a-9976-4aab-bd50-c8b38eb13de3"/>
</groups>
<configs>
  <link xlink:title="Standard-Konf."
        xlink:href="res/31d2e52b-f204-40f7-83cd-e2f33d9d02c9"/>
  <link xlink:title="Spezielle Konf. fuer Herrn Mustermann"
        xlink:href="res/4cf21b65-7717-4558-bd12-6a70fe9f2e84"/>
  <link xlink:title="Konf. ohne Routen"
        xlink:href="res/625f51f3-0461-4377-8406-e39d6374c98f"/>
</configs>
<create xlink:href="/create"/>
<search xlink:href="/process"/>
</index>
```

Die Repräsentation enthält eine Auflistung der wesentlichen Ressourcen des Serversystems. Zum einen sind dies die Benutzer, welche Zugriff auf die OpenVPN-Konfigurationen

haben und die Benutzergruppen, zu denen diese zugeordnet sind. [1]

Im Folgenden ruft der Client einer der Open-VPN-Konfigurationen ab.

```
33 GET http://server.de/res/4cf21b65-7717-4558-bd12-6a70fe9f2e84
34 Accept: text/xml
```

Daraufhin ermittelt der Server, analog zum Abruf der Indexressource, die entsprechende Ressource, überführt dessen Repräsentation in eine XML nutzende Repräsentation und gibt diese schießlich zurück:

```
35 <?xml version="1.0" encoding="UTF-8"?>
36 <config xml:base="http://server.de/"
37     xmlns="http://www.opensource-training.de/openvpn/config/"
38     xmlns:xlink="http://www.w3.org/1999/xlink">
39   <name>Spezielle Konf. fuer Herrn Mustermann</name>
40   <user xlink:title="Max Mustermann"
41         xlink:href="res/81db8aa7-fcd3-47d5-87d8-767a6d47c02c"/>
42   <mode>p2p</mode>
43   <proto>udp</proto>
44   <port>1194</port>
45   <ifconfig>
46     <local>10.4.0.1</local>
47     <remote>10.5.0.1</local>
48   </ifconfig>
49   <route>
50     <networkIP>192.168.1.0</networkIP>
51     <netmask>255.255.255.0</netmask>
52     <gateway>192.168.0.99</gateway>
53     <metric>1</metric>
54   </route>
55   <route>
56     <networkIP>172.16.10.0</networkIP>
57     <netmask>255.255.255.0</netmask>
58     <gateway>192.168.0.99</gateway>
```

[1] Auf eine genaue Beschreibung der durch die Serversoftware realisierten Benutzerverwaltung wird verzichtet, da die Art der Umsetzung in allen für diese Arbeit wesentlichen Punkten analog zu den beschriebenen Softwarekomponenten ist.

```
59      <metric>5</metric>
60    </route>
61    <user
62  </config>
```

Alternativ könnte der Client die Ressource auch anhand einer anderen Repräsentationsform abrufen. Beispielsweise wäre denkbar, dass er die Konfiguration direkt an den OpenVPN-Client-Prozess, der die oben aufgeführte XML-basierte Darstellung nicht verarbeiten kann, weitergeben möchte. In diesem Fall würde dieser die Ressource wie folgt abrufen und den Server anweisen, diese im OpenVPN-spezifischen Konfigurationsformat auszugeben:

```
63 GET http://server.de/res/4cf21b65-7717-4558-bd12-6a70fe9f2e84
64 Accept: text/vnd.openvpntechnologies.openvpn.config
```

Die Verarbeitung dieser Anfrage auf dem Server entspricht der oben aufgeführten, mit der Ausnahme, dass ein anderer Konverter zur Erzeugung der entsprechenden Repräsentation genutzt wird.

```
65 mode p2p
66 proto udp
67 port 1194
68 ifconfig 10.4.0.1 10.5.0.1
69 route 192.168.1.0 255.255.255.0 192.168.0.99 1
70 route 172.16.10.0 255.255.255.0 192.168.0.99 5
```

Zu beachten ist jedoch, dass diese Repräsentationsform speziell zur Darstellung einer Konfiguration entworfen und deshalb nicht erweiterbar ist.[2] können in diesem Fall die Eigenschaften `name` und `user` nicht abgebildet werden.

Im Anschluss ordnet der Client dieser OpenVPN-Konfiguration einen weiteren Benutzer zu, indem dieser die vorliegende XML-Repräsentation entsprechend abändert und diese zurück an den Server übermittelt. Da es sich hierbei um eine idempotente Operation handelt verwendet der Client hierzu, getreu der HTTP-Spezifikation die Methode `PUT`.

[2] Diese Darstellungsform ist also in Abhängigkeit zu den Elementen, die dargestellt werden sollen, entworfen. Sie ist somit in gewissem Sinne ein Negativbeispiel, dass in Kapitel 2.2 empfohlenen Vorgehens.

```
71 PUT http://server.de/res/4cf21b65-7717-4558-bd12-6a70fe9f2e84
72 Content-Type: text/xml
73 Accept: text/xml
74
75 <?xml version="1.0" encoding="UTF-8"?>
76 <config xml:base="http://server.de/"
77     xmlns="http://www.opensource-training.de/openvpn/config/"
78     xmlns:xlink="http://www.w3.org/1999/xlink">
79   <name>Spezielle Konf. fuer Herrn Mustermann</name>
80   <user xlink:title="Max Mustermann"
81         xlink:href="res/81db8aa7-fcd3-47d5-87d8-767a6d47c02c"/>
82   <link xlink:href="res/1f731b0d-47f2-44ab-b4d3-1f9db5f84a0c"/>
83   <mode>p2p</mode>
84   <proto>udp</proto>
85   <port>1194</port>
86   <ifconfig>
87     <local>10.4.0.1</local>
88     <remote>10.5.0.1</local>
89   </ifconfig>
90   <route>
91     <networkIP>192.168.1.0</networkIP>
92     <netmask>255.255.255.0</netmask>
93     <gateway>192.168.0.99</gateway>
94     <metric>1</metric>
95   </route>
96   <route>
97     <networkIP>172.16.10.0</networkIP>
98     <netmask>255.255.255.0</netmask>
99     <gateway>192.168.0.99</gateway>
100    <metric>5</metric>
101  </route>
102  <user
103 </config>
```

Nachfolgend wird die Antwort des Server auf diese Anfrage, welche die geänderte Res-

source zurückgibt, in gekürzter Form dargestellt:

```
104 <?xml version="1.0" encoding="UTF-8"?>
105 <config xml:base="http://server.de/"
106     xmlns="http://www.opensource-training.de/openvpn/config/"
107     xmlns:xlink="http://www.w3.org/1999/xlink">
108   <name>Spezielle Konf. fuer Herrn Mustermann</name>
109   <user xlink:title="Max Mustermann"
110         xlink:href="res/81db8aa7-fcd3-47d5-87d8-767a6d47c02c"/>
111   <link xlink:title="Hendrik Schwartke"
112         xlink:href="res/1f731b0d-47f2-44ab-b4d3-1f9db5f84a0c"/>
113   <mode>p2p</mode>
114   <proto>udp</proto>
115
116   ...
117
118 </config>
```

Abschließend ermittelt der Client alle OpenVPN-Konfigurationen, die einen VPN-Tunnel auf Basis des User Datagram Protocols (UDP) (s.h. [30]) spezifizieren. Dazu verfasst dieser eine entsprechende SPARQL-Anfrage und übermittelt diese der entsprechende Ressource, dessen URI diesem durch das Element `search` aus der Indexresource bekannt ist:

```
119 POST http://server.de/process
120 Accept: application/sparql-results+xml
121 Content-Type: application/sparql-query
122
123 PREFIX <http://www.opensource-training.de/openvpn/config/>
124 SELECT ?config WHERE {?config proto "udp".}
```

Der Server ermittelt daraufhin die Implementierung der Schnittstelle Processor, die für SPARQL-Anfragen zuständig ist und übergibt dieser die Anfrage. Das Suchergebnis wird im Anschluss zurückgegeben:

```
125 <?xml version="1.0" encoding="UTF-8"?>
126 <sparql xmlns="http://www.w3.org/2005/sparql-results#">
127   <head>
```

```
128     <variable name="config"/>
129   </head>
130   <results>
131    <result>
132     <binding name="config">
133      http://server.de/res/31d2e52b-f204-40f7-83cd-e2f33d9d02c9
134     </binding>
135    </result>
136    <result>
137     <binding name="config">
138      http://server.de/res/4cf21b65-7717-4558-bd12-6a70fe9f2e84
139     </binding>
140    </result>
141   </results>
142 </sparql>
```

Teil IV.

Abschluss

8. Fazit

Wenngleich die bearbeitete Fragestellung, wie lose gekoppelte Systeme realisiert werden können, in ihrer Breite nicht durch eine einzelne Arbeit beantwortet werden kann, so bietet der gewählte Lösungsansatz, den Architekturstil REST und einen modellgetriebenen Entwicklungsansatz zu kombinieren, doch ein großes Potential zur Lösung dieser Fragestellung.

Insbesondere daran, dass sich durch diesen Ansatz Aspekte, die unweigerlich mit der Kommunikation von Softwaresystemen verbunden sind, wie der Zugriff auf Daten, deren Darstellung und deren Bedeutung, weitestgehend separiert voneinander umsetzten lassen, zeigt einen nicht zu unterschätzenden Mehrwert gegenüber anderen Ansätzen. Zwar mag die Zerlegung in Teilprobleme auf den ersten Blick kontraproduktiv wirken, da sich daraus einige Nachteile gegenüber einer Lösung „aus einem Guss“ ergeben. Beispielsweise kann eine Kommunikation, die nicht auf verschiedenen aufeinander aufbauenden Protokollen basiert, sondern auf einem einzelnen auf die konkrete Anwendungssituation optimierten Protokoll, u.U. wesentliche Performancevorteile mit sich bringen.

Es zeigen sich aber auch eine Reihe von Vorteilen, die die Erstellung lose gekoppelter Systeme wesentlich vereinfachen können. So ist es aufgrund der klaren Strukturierung von Kommunikation auf abstraktem Niveau in Form eines Schichtenmodells möglich, die verschiedenen Anforderungen von Kommunikation weitestgehend unabhängig voneinander zu analysieren und umzusetzen. Somit wird eine Kombination von Lösungsansätzen und die Wiederverwendung deren Implementierung ermöglicht.

Ferner zeigt sich, dass der Architekturstil REST eine solche modulare Strukturierung von Kommunikation mit recht geringem Aufwand ermöglicht. Insbesondere die Tatsache, dass die Interaktion von Softwaresystemen durch REST sehr präzise aber totzdem auf hohem Abstraktionsniveau beschrieben wird, ermöglicht es, REST-konforme Software zu erstellen, die auf individuelle Anforderungen hin zugeschnitten ist, und trotzdem alle mit REST verbundenen Vorteile aufweist.

Des Weitern zeigt sich, dass der Einsatz eines modellgetriebenen Entwicklungsansatzes durchaus mit dem Architekturstil REST harmoniert. Insbesondere zur Modellierung

semantischer Elemente kann ein solcher Ansatz dienen.

Literaturverzeichnis

[1] *EMF Developer Guide - Package org.eclipse.emf.ecore.* `http://help.eclipse.org/galileo/index.jsp`, Abruf: 3. August 2010

[2] Balzert, Helmut (Hrsg.): *CASE - Systeme und Werkzeuge.* 5., vollst. überarb. und aktualisierte Aufl. Mannheim [u.a.] : BI-Wiss.-Verl., 1993 (Angewandte Informatik ; 7). – 815 S. : graph. Darst.. – ISBN 3–411–14685–0

[3] Beckett, Dave: RDF/XML Syntax Specification (Revised) / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

[4] Bengel, Günther: *Verteilte Systeme : Client-server-computing für Studenten und Praktiker.* 2., überarb. und erw. Aufl. Braunschweig [u.a.] : Vieweg, 2002 (Vieweg-Lehrbuch). – XII, 383 S. : Ill.. – ISBN 3–528–15738–0. – 3. Aufl. u.d.T.: Bengel, Günther: Grundkurs verteilte Systeme

[5] Berners-Lee, T. ; Fielding, R. ; Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986 (Standard). `http://www.ietf.org/rfc/rfc3986.txt`. Version: Januar 2005 (Request for Comments)

[6] Berners-Lee, Tim: Notation 3 - An readable language for data on the Web. Version: March 2006. `http://www.w3.org/DesignIssues/Notation3`, Abruf: 3. August 2010. 2006. – Forschungsbericht

[7] Berners-Lee, Tim ; Hendler, James ; Lassila, Ora: The Semantic Web. In: *Scientific American* 284 (2001), Nr. 5, 34-43. `http://www.scientificamerican.com/article.cfm?id=the-semantic-web`. – ISSN 0036–8733

[8] Braden, R.: *Requirements for Internet Hosts - Communication Layers.* RFC 1122

(Standard). `http://www.ietf.org/rfc/rfc1122.txt`. Version: Oktober 1989 (Request for Comments). – Updated by RFCs 1349, 4379, 5884

[9] Carroll, Jeremy J. ; Klyne, Graham: Resource Description Framework (RDF): Concepts and Abstract Syntax / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[10] Cerf, V.G.: *ASCII format for network interchange.* RFC 20. `http://www.ietf.org/rfc/rfc20.txt`. Version: Oktober 1969 (Request for Comments)

[11] Christense, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: Web Services Description Language (WSDL) 1.1 / W3C. Version: March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, Abruf: 3. August 2010. 2001. – W3C Note

[12] DeRose, Steven ; Orchard, David ; Maler, Eve: XML Linking Language (XLink) Version 1.0 / W3C. 2001. – W3C Recommendation. – http://www.w3.org/TR/2001/REC-xlink-20010627/

[13] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). `http://www.ietf.org/rfc/rfc2616.txt`. Version: Juni 1999 (Request for Comments). – Updated by RFCs 2817, 5785

[14] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000. `http://www.ics.uci.edu/ fielding/pubs/dissertation/fielding_dissertation.pdf`, Abruf: 3. August 2010

[15] Freed, N. ; Borenstein, N.: *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* RFC 2045 (Draft Standard). `http://www.ietf.org/rfc/rfc2045.txt`. Version: November 1996 (Request for Comments). – Updated by RFCs 2184, 2231, 5335

[16] Gamma, Erich (Hrsg.): *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software.* 1. Aufl., [Neuaufl.]. München [u.a.] : Addison-Wesley, 2004 (Programmer's choice). – XX, 479 S. : graph. Darst.. – ISBN 3–8273–2199–9. – Design patterns <dt.>

[17] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification, Third Edition*. 3. Amsterdam : Addison-Wesley Longman, 2005. – 688 S. – ISBN 0321246780

[18] Grimm, Rüdiger: *Digitale Kommunikation*. München : Oldenbourg Verlag, 2005

[19] Gudgin, Martin ; Hadley, Marc ; Mendelsohn, Noah ; Lafon, Yves ; Moreau, Jean-Jacques ; Karmarkar, Anish ; Nielsen, Henrik F.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) / W3C. Version: April 2007. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`, Abruf: 3. August 2010. 2007. – W3C Recommendation

[20] Hayes, Patrick: RDF Semantics / W3C. 2004. – W3C Recommendation. – http://www.w3.org/TR/2004/REC-rdf-mt-20040210/

[21] ISO: *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. 1996. – 12 S. `http://www.iso.ch/cate/d26153.html`

[22] McCabe, Francis ; Booth, David ; Ferris, Christopher ; Orchard, David ; Champion, Mike ; Newcomer, Eric ; Haas, Hugo: Web Services Architecture / W3C. 2004. – W3C Note. – http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[23] Miller, J. ; Mukerji, J.: MDA Guide Version 1.0.1 / Object Management Group (OMG). 2003. – Forschungsbericht

[24] Myers, Glenford J.: *Reliable software through composite design*. Petrocelli/Charter, 1975. – ISBN 0884052842

[25] OMG: *MOF QVT Final Adopted Specification*, June 2005. `http://fparreiras/papers/mof_qvt_final.pdf`

[26] OMG: *Object Constraint Language Specification, version 2.0.* : Object Modeling Group, June 2005. (OMG document formal/2006-05-01) . `http://fparreiras/papers/OCLSpec.pdf`

[27] omg: *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`

[28] OMG: OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Version: 2010. `http://www.omg.org/spec/UML/2.3/Infrastructure/PDF`. 2010 (formal/2010-05-03). – Forschungsbericht

[29] OpenVPN Technologies, Inc.: *OpenVPN 2.1*, November 2008. `http://openvpn.net/index.php/open-source/documentation/manuals/69-openvpn-21.html`

[30] Postel, J.: *User Datagram Protocol*. RFC 768 (Standard). `http://www.ietf.org/rfc/rfc768.txt`. Version: August 1980 (Request for Comments)

[31] Prud'hommeaux, Eric ; Seaborne, Andy: SPARQL Query Language for RDF / W3C. 2008. – W3C Recommendation. – http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

[32] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Wien [u.a.] : Springer, 1973. – XV, 494 S. : graph. Darst.. – ISBN 3–211–81106–0 – 0–387–81106–0

[33] Stahl, Thomas (Hrsg.): *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. 2., aktualisierte und erw. Aufl. Heidelberg : dpunkt.-Verl., 2007. – XV, 441 S. : Ill., graph. Darst.. – ISBN 978–3–89864–448–8 – 3–89864–448–0

[34] Steinberg, Dave (Hrsg.): *EMF - Eclipse Modeling Framework*. 2. ed., rev. and updated, 1. printing. Upper Saddle River, NJ ; Munich [u.a.] : Addison Wesley, 2009 (The Eclipse series). – XXIX, 704 S. : Ill., graph. Darst.. – ISBN 978–0–321–33188–5 – 0–321–33188–5

[35] Tilkov, Stefan: *REST Anti-Patterns*. Version: July 2008. `http://www.infoq.com/articles/rest-anti-patterns`, Abruf: 3. August 2010

[36] Yergeau, F.: *UTF-8, a transformation format of ISO 10646*. RFC 3629 (Standard). `http://www.ietf.org/rfc/rfc3629.txt`. Version: November 2003 (Request for Comments)

[37] Zimmermann, H.: OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on Communications* 28 (1980), Nr. 4, 425–432. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1094702`

Erklärung zur Masterarbeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Steinfurt, den 24. August 2010

_____