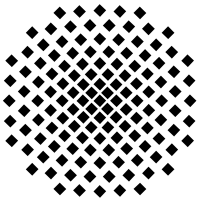
Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart



Diplomarbeit Nr. 3454

REST Testbed

Nick Eisenbraun

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Florian Haupt
begonnen am:	04.02.2013
beendet am:	06.08.2013
CR-Klassifikation:	C.2.4, D.2.1, D.2.5, D.2.11

Kurzfassung

Web Services, die gemäß dem Architekturstil REST entworfen werden, zeichnen sich durch Eigenschaften wie Interoperabilität, lose Kopplung, Wiederverwendbarkeit, Leistung und Skalierbarkeit aus. In verteilten Systemen werden deswegen oft REST-basierte Web Services eingesetzt. Verteilte Systeme haben höhere Fehleranfälligkeit als Standalone-Anwendungen und diese Erkenntnis sollte beim Entwicklungsprozess durch ausreichende Testszenarien berücksichtigt werden. Bei der Entwicklung von REST-basierten Client-Anwendungen wird ein REST-basierter Web Service benötigt, um die Funktionalitäten der Client-Anwendung zu testen. In dieser Diplomarbeit werden Anforderungen an ein Testbed zum Testen von REST-basierten Client-Anwendungen gestellt. Es wird eine Architektur zu diesem Testbed entworfen und anschließend ein Testbed prototypisch implementiert. Bei der Entwicklung des Testbeds werden die Eigenschaften wie Erweiterbarkeit und Konfigurierbarkeit der Funktionalitäten des REST-basierten Web Services sichergestellt. Durch die berücksichtigte Erweiterbarkeit kann das Testbed leicht um neue Funktionalitäten ergänzt werden. Die Konfigurierbarkeit erlaubt das Anpassen der funktionalen und nicht-funktionalen Eigenschaften des Testbeds, um die Erfüllbarkeit der an eine Client-Anwendung gestellten funktionalen und nicht-funktionalen Anforderungen zu überprüfen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AJDT AspectJ Development Tools

API Application Programming Interface

AOP Aspect-Oriented Programming

BAM Business Activity Monitoring

CRUD Create, Retrieve, Update, Delete

CSV Comma-Separated Values

ESB Enterprise Service Bus

GUI Graphical User Interface

GWT Google Web Toolkit

HATEOAS Hypermedia as the Engine of Application State

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IEEE Institute of Electrical and Electronics Engineers

JAX-RS Java API for RESTful Web Services

JSON JavaScript Object Notation

LPML Literate Programming Markup Language

MIME Multipurpose Internet Mail Extensions

MVC Model View Controller

RDF Resource Description Framework

RFC Request for Comments

QoS Quality of Service

REST Representational State Transfer

ROA Resource-Oriented Architecture

SAWSDL Semantic Annotations for WSDL and XML Schema

SSL Secure Sockets Layer

SOA Service-Oriented Architecture

SOA4ALL Service Oriented Architectures for All

SOAP Simple Object Access Protocol *(deprecated)*

UDDI Universal Description, Discovery and Integration

URI Uniform Resource Identifier

W3C World Wide Web Consortium

WADL Web Application Description Language

WSDL Web Services Description Language

XML eXtensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Representational State Transfer (REST) Web Services stellen eine einfache Alternative zur Realisierung von Web Services dar. Die Einfachheit bezieht sich dabei auf die Nutzung von bekannten Standards wie HyperText Markup Language (HTML), eXtensible Markup Language (XML), Uniform Resource Identifier (URI) und Multipurpose Internet Mail Extensions (MIME). Eine ressourcenorientierte Architektur (ROA) stellt eine konkrete REST-konforme Architektur zur Umsetzung REST-konformer Web Services dar. Bei ROA stehen Ressourcen im Mittelpunkt, die eindeutig identifizierbar und somit adressierbar sind. Alle Ressourcen verfügen über eine einheitliche Schnittstelle, die sich durch die Create, Retrieve, Update, Delete (CRUD)-Operationen auszeichnet. Unter Verwendung von HTTP bei der Implementierung von REST Web Services sind das die entsprechenden HTTP-Methoden. Ressourcen können über verschiedene Repräsentationen in Formaten wie HTML, XML, JavaScript Object Notation (JSON) etc. dargestellt werden. Unter Verwendung von Links in Repräsentationen kann auf andere Ressourcen verwiesen werden, die in einer Beziehung zu der betrachteten Ressource stehen (PZL08, S. 807). Eine weitere Eigenschaft von REST ist die gute Skalierbarkeit im Web (Til11, S. 4). Legt man auf diese Eigenschaften Wert, so werden REST Web Services bevorzugt eingesetzt.

Wie bei jeder Softwareentwicklung treten auch bei der Entwicklung von REST-basierten Client-Anwendungen Schwierigkeiten auf. Bei Client/Server-Anwendungen besteht aufgrund der Verteilung der Komponenten zusätzliche Fehleranfälligkeit (SW02, S. 17, 19). Ein Netzwerk, über welches die verteilten Komponenten kommunizieren, kann überlastet oder nicht verfügbar sein. Die Anwendungen sollten auf verschiedene Fehlersituationen vorbereitet sein und nicht nur auf das spezifizierte Verhalten. Es gibt verschiedene Verfahren, um ein Anwendungssystem während ihres Lebenszyklus auf Fehler zu untersuchen. Um qualitative Ergebnisse zu liefern, braucht man Testwerkzeuge zur Überprüfung der funktionalen und nicht-funktionalen Eigenschaften der in der Entwicklung befindender Anwendung. Um eine Client-Anwendung zu testen, wird an erster Stelle eine Server-Anwendung vorausgesetzt, mit der die Client-Anwendung kommunizieren soll. Genau darauf konzentriert sich diese Diplomarbeit. Es soll ein konfigurierbares Testbed mit REST-basierten Web Services zum Verifizieren der funktionalen und nicht-funktionalen Eigenschaften von REST-basierten Client-Anwendungen entwickelt werden.

Unter Testbed wird hierbei eine Experimintierumgebung verstanden, die als Plattform zur Förderung experimenteller Arbeitsweisen dient. Gemäß Institute of Electrical and Electronics Engineers (IEEE) Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, wird der Begriff Testbed mit einer Umgebung assoziiert, die Hardware, Instrumente, Simulatoren, Software-Tools und andere unterstützende Elemente zum Durchführen von Tests beinhaltet (IEE90).

1.1 Motivation

Das Testen ist ein bedeutsamer Teil der Softwareentwicklung. Die Entwicklung soll während des ganzen Entwicklungsprozesses kontinuierlich durch verschiedene Tests unterstützt werden. Zu solchen Tests gehören Modultest (Unit Test), Integrationstest etc. Für gewünschte Tests werden passende Web Services beziehungsweise Tools gebraucht, die oft erst speziell an eine einzelne Client-Anwendung zugeschnitten implementiert werden müssen. In manchen Fällen könnte man auf eine neue Implementierung von REST-basierten Web Services verzichten, indem man auf bereits vorhandene Implementierungen wie von einem externen Service-Anbieter ausweicht. Diese sind jedoch oft mit Nutzungskosten oder anderen Limitierungen verbunden, was die Experimentierfreiheit deutlich einschränkt. Ein konfigurierbares REST Testbed unterstützt somit als Testwerkzeug bei der Entwicklung einer REST-basierten Client-Anwendung, ohne unnötige Einschränkungen, vor allem in den früheren Entwicklungsphasen, zu verursachen. Die entwickelten Komponenten in der Client-Anwendung können auf diese Weise getestet, evaluiert sowie demonstriert werden. Dazu muss ein Testbed umfassende Funktionalität bieten, konfigurierbar und beobachtbar sein. Durch reproduzierbare Testszenarien genauso wie Simulation von Fehlerfällen kann das Testbed bei dem Testprozess und somit bei der Suche nach Fehlverhalten die Tester und Entwickler unterstützen.

1.2 Aufgabenstellung

In dieser Diplomarbeit sollen zunächst die bereits skizzierten Anforderungen aus dem Abschnitt 1.1 an ein entsprechendes REST Testbed detailliert erhoben und aufbereitet werden. Anschließend soll ein Überblick über bestehende Testbeds und ihre Anwendbarkeit auf die vorliegende Problemstellung gegeben werden. Weiterhin soll ein Konzept sowie eine Architektur eines REST Testbeds entworfen und anschließend prototypisch implementiert werden.

1.3 Eingrenzung

Das Ziel dieser Diplomarbeit ist die Entwicklung eines konfigurierbaren REST Testbeds mit umfassender Funktionalität zum Testzweck. Dieser dient zur Überprüfung der Erfüllbarkeit von funktionalen und nicht-funktionalen Anforderungen, die an REST-basierte Client-Anwendungen gestellt werden. Mit der Konfigurierbarkeit ist die Möglichkeit zum Steuern von funktionalen beziehungsweise nicht-funktionalen Eigenschaften gemeint. Die Abbildung 1.1 zeigt Komponenten des Testbeds. Es werden zwei Schnittstellen zur Verfügung gestellt. Über die Schnittstelle *ConfigurationAPI* sollen die funktionalen und nicht-funktionalen Eigenschaften des REST Testbeds angepasst und über die Schnittstelle *TestServiceAPI* sollen die Ressourcen angefragt werden. Basierend auf der in Abschnitt 1.2 beschriebenen Problemstellung soll ein REST-basierter Web Service zum Testen von Client-Anwendungen die

Schnittstelle *TestServiceAPI* implementieren. Dabei sollen die über die Schnittstelle *ConfigurationAPI* konfigurierten funktionalen und nicht-funktionalen Eigenschaften sich auf den REST-basierten Web Service auswirken. Nach einer passenden Konfiguration aller im Testbed vorhandenen funktionalen und nicht-funktionalen Eigenschaften an bestimmte Testfälle, kann der REST-basierte Web Service zum Testen der Client-Anwendungssysteme in Anspruch genommen werden.

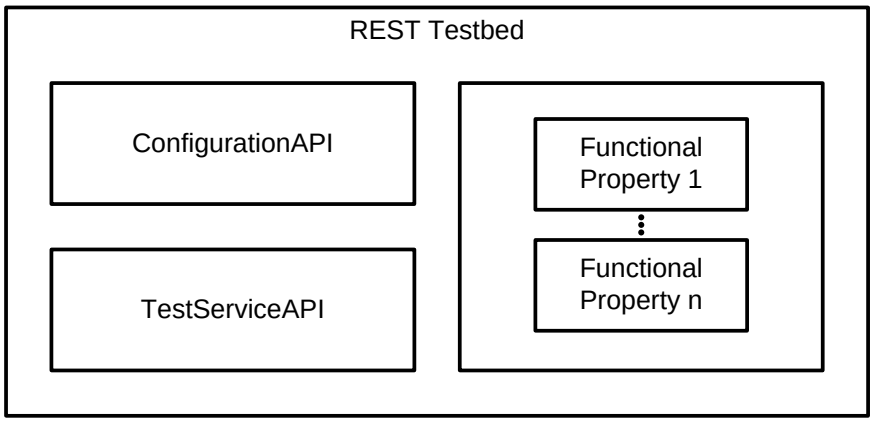


Abbildung 1.1: Überblick über Komponenten des Testbeds

1.4 Aufbau der Arbeit

Der Rest der Ausarbeitung dieser Diplomarbeit ist wie folgt aufgebaut. Um das REST Testbed besser zu verstehen, bietet das Kapitel 2 einen Einblick in die Konzepte des REST-Architekturstils, einiges an Grundwissen aus dem Bereich Software-Prüfung und der aspektorientierten Programmierung. Kapitel 3 präsentiert einige verwandte Arbeiten, welche in Bezug auf die Problemstellung von REST Testbed diskutiert werden. In Kapitel 4 werden die an das zu entwickelnde REST Testbed gestellten Anforderungen detailliert erhoben und aufbereitet. Aufbauend auf den Erkenntnissen aus Kapitel 3 und den Anforderungen aus Kapitel 4 wird in Kapitel 5 ein Konzept ausgearbeitet. Zu diesem Konzept wird eine komponentenbasierte Architektur zum REST Testbed mit den Erweiterungskomponenten vorgeschlagen. Kapitel 6 widmet sich dem Design vom REST Testbed. In diesem Kapitel sind die Klassendiagramme von dem komponentenbasierten Testbed sowie von den einzelnen Erweiterungskomponenten zu finden. Die implementierungstechnischen Details zum REST Testbed und den Erweiterungskomponenten sind in Kapitel 7 beschrieben. Das letzte Kapitel beinhaltet eine Zusammenfassung dieser Diplomarbeit und gibt einen Überblick über die noch anstehende Verbesserungs- und Erweiterungsmöglichkeiten des REST Testbeds.

2 Grundlagen

Dieses Kapitel bietet einen Einblick in die Eigenschaften des REST-Architekturstils. Es wird einiges an Grundwissen der Software-Prüfung und der aspektorientierten Programmierung, anhand einiger Konstrukte aus der aspektorientierten Sprache AspektJ, vermittelt.

2.1 REST und ROA

Representational State Transfer (REST) beschreibt einen Architekturstil von verteilten Anwendungssystemen. Ressourcenorientierte Architektur (ROA) stellt eine konkrete REST-konforme Architektur und somit eine Alternative zur Umsetzung REST-konformer Web Services dar. Die Kernbestandteile von ROA sind Ressourcen, deren Namen und Repräsentationen genauso wie Verweise zwischen Ressourcen. Zu den Eigenschaften von ROA gehören Adressierbarkeit, Zustandslosigkeit, Verbindungshaftigkeit und die einheitliche Schnittstelle. Auf die genannten Bestandteile und Eigenschaften von ROA wird später in diesem Kapitel eingegangen.

REST- und SOAP-basierte Web Services sind zurzeit die gängigsten Alternativen, die Web Services zu realisieren, und je nach gestellten Anforderungen wird die eine oder andere bevorzugt (Ric07, S. XIV-XV). Ein Unterschied zwischen REST und SOAP Web Services besteht darin, wie die Dokumente übertragen werden. Ein Beispiel von Web Services, die mit HTTP-Protokoll umgesetzt sind, verdeutlicht diesen Unterschied. Bei REST Web Services werden Dokumente in einen HTTP-Umschlag verpackt. SOAP Web Services verpacken Dokumente in einen SOAP-Umschlag, der wiederum in einen HTTP-Umschlag verpackt wird (Ric07, S. 23).

Im Folgenden wird das Interesse voll und ganz den REST-konformen Web Services gewidmet. Der Architekturstil REST ist vor allem wegen seiner Leichtgewichtigkeit für die Software-Entwickler sehr attraktiv. Als nächstes wird das geschichtliche Entstehen des Begriffs REST erläutert und die REST-Prinzipien aufgeführt und diskutiert.

Es werden explizit einige Technologien wie HTTP, URI und XML genannt und für die Implementierung von REST-konformen Web Services herangezogen. Diese erfüllen die an den REST-Architekturstil gestellten Voraussetzungen. Dabei wird betont, dass es beim REST sich um ein Architekturstil handelt und dieser grundsätzlich mit keiner Technologie verbunden ist (Ric07, S. 89).

2.1.1 Geschichtliches

Der Begriff REST wurde von Roy Thomas Fielding im Jahr 2000 in seiner Dissertation *Architectural Styles and the Design of Network-based Software Architectures* geprägt. Er verglich

verschiedene Architekturen der verteilten Systeme auf deren Eigenschaften und definierte REST als Folgerung seiner Forschungen. REST war eine Zusammenstellung von verschiedenen Designkriterien (Ric07, S. XII-XIV,90). Im fünften Kapitel seiner Dissertation hat er den REST-Architekturstil durch weiter unten diskutierte Architekturprinzipien beschrieben (Fie00).

2.1.2 Ressourcen

Wie bereits in Abschnitt 2.1 erwähnt, gehört ROA zu den REST-konformen Architekturen und konzentriert sich auf die Ressourcen. Ein Objekt oder Liste von Objekten, die mit einem Modell beschrieben werden können, werden als Ressource interpretiert. Die Ressourcen bekommen einen eigenen eindeutigen URI und werden dadurch adressierbar. Die Ressourcen können und sollten auch Verweise auf andere Ressourcen beinhalten, mit denen sie in einer Beziehung stehen (Ric07, S. 95). Dieses Prinzip der Verlinkung wird auch als *Hypermedia as the Engine of Application State* bezeichnet. Eine weitere Eigenschaft einer Ressource ist deren Repräsentationen. Für jede Ressource kann eine Menge von verschiedenen Repräsentationen bereitgestellt werden, woraus man sich durch *Content Negotiation* auf eine Darstellung einigt. Im weiteren Verlauf werden die oben angeschnittenen Eigenschaften von Ressourcen genauer beschrieben (Fie00).

Identifizierbarkeit

Unter Ressourcen sind Objekte oder Listen von Objekten gemeint. Damit diese Objekte beziehungsweise Listen von Objekten abrufbar sind, müssen sie adressierbar sein. Das hat bei REST zur Folge, dass alle Ressourcen gemäß Request for Comments (RFC) 2396 (BL98) einen URI besitzen. Mit URIs können die Ressourcen global eindeutig identifiziert und somit abrufbar gestaltet werden (Ric07, S. 92-97) (PZL08, S. 807).

Hypermedia as the Engine of Application State

Mit Hypermedia as the Engine of Application State (HATEOAS) werden die Verlinkungen von Ressourcen assoziiert. Es wird nicht nur eine Repräsentation einer Ressource alleine vom Server zum Client übertragen, sondern auch Verlinkungen auf andere Ressourcen. Besteht bei einer Ressource ein Zusammenhang mit anderen Ressourcen, so sollen der Repräsentation zusätzlich die entsprechenden Links auf diese Ressourcen eingebettet werden. Diese zusätzlichen Informationen sind als Zustandsübergänge von dem Client zu interpretieren (Ric07, S. 106-109) (Fie08).

Repräsentationen

Die Repräsentation einer Ressource schließt die Darstellung von Daten genauso wie deren Metadaten ein. Bei der Benutzung des HTTP-Protokolls werden dazu MIME-Typen verwendet. Die wohl gängigsten Formate zur Repräsentation von Ressourcen sind HTML, XML und

JSON. Bei der Inhaltsvereinbarung (Content Negotiation) zum Beispiel mit Hilfe des HTTP-Headers bei der HTTP-Anfrage wird zwischen Client und Server über den bevorzugten MIME-Typ zur Repräsentation der Ressource ausgehandelt (Fel10, S. 53-54) (Fie00).

Selbstbeschreibende Nachrichten

Diese Eigenschaft ist die direkte Forderung der zustandslosen Kommunikation zwischen Client und Server. Die zustandslose Kommunikation bedeutet, dass der Zustand der Interaktion zwischen Client und Server nicht vom Server verwaltet wird. Deswegen sollen die Nachrichten alle notwendigen Daten und Metadaten enthalten, um die gewünschte Aufgabe erledigen zu können. Der Zustand einer Anwendung wird somit mit der Repräsentation der Ressource beschrieben (BS07). Das hat zur Folge, dass der Inhalt der Nachrichten auch von den Zwischenknoten auf dem Pfad zwischen Client und Server interpretiert werden kann. Diese Eigenschaft ist auch wichtig, um dem Caching gerecht zu werden (Til11, S. 144). Wenn ein Zwischenknoten feststellen kann, dass eine angefragte Repräsentation einer Ressource im Cache von diesem Knoten verfügbar und noch aktuell ist, dann kann dieser Knoten diese Repräsentation aus eigenem Cache dem Anfragesteller übermitteln und somit den Server entlasten.

2.1.3 Architekturprinzipien

Dem REST sind einige Designkriterien zu Grunde gelegt, die bei der Entwicklung von REST-konformen Web Services zu beachten sind, um die Vorteile des Webs nutzen zu können (Fie00). Diese Kriterien werden nun kurz vorgestellt.

Client-Server-Architektur

Durch die Standardisierung der Kommunikation werden Client und Server voneinander entkoppelt. So können mehrere Clients die Dienste eines Servers in Anspruch nehmen. Der Client spielt eine aktive Rolle bei diesem Konzept. Er kann eine Anfrage an den Server stellen. Der Server ist dabei eine passive Komponente, die auf Anfragen des Clients wartet. Beim Eintreffen der Anfragen werden entsprechende Prozeduren ausgeführt und dem Client die angemessene Antwort geliefert. Dieses Prinzip trägt durch die Trennung der Angelegenheiten der Benutzerschnittstelle von den Aufgaben der Datenhaltung zu der Portabilität des Clients über mehrere Plattformen hinweg bei und auf dem Server wird die Skalierbarkeit durch die Vereinfachung der Komponenten verbessert. Dadurch wird es möglich die Komponenten voneinander unabhängig zu entwickeln (Fie00, S. 78).

Zustandslosigkeit

Prinzip der Zustandslosigkeit bezieht sich auf die zustandslose Kommunikation zwischen Client und Server. Das bedeutet, dass einzelne Anfragen aus einer Sequenz der Anfragen

an einen Server unabhängig voneinander erfolgen und somit als einzelne Transaktionen zu betrachten sind. Für die Kommunikation verwendet REST das zustandslose HTTP-Protokoll. Nach diesem Prinzip muss der Client nicht an einen bestimmten Server gebunden sein. Beim Ausfall eines Servers kann der Client seine Arbeit einfach mit einem anderen Server mit einem gleichen Dienst fortsetzen. Dieses Prinzip ermöglicht somit die Skalierbarkeit, trägt zu der Einfachheit und der Transparenz bei (Ric07, S. 98-101) (Fie00) (PZL08, S. 3).

Cache

Die bereits empfangenen Ressourcen können auf dem Client oder den Zwischenknoten auf dem Pfad zum Server gespeichert werden. Wenn bei den angefragten Ressourcen, die sich schon aus den alten Anfragen auf dem Client oder Zwischenknoten befinden, festgestellt werden kann, dass diese immer noch aktuell sind, dann können diese von dem Client verwendet werden. So kann der Server entlastet werden und man vermeidet unnötige Kommunikationszeiten. Dieses Prinzip ermöglicht folglich gute Skalierbarkeit und verbessert die Effizienz (Fie00, S. 79-81).

Einheitliche Schnittstelle

Unter der einheitlichen Schnittstelle werden die Ressourcen-Operationen verstanden. REST ist nicht an das Web gebunden und hängt nicht von den HTTP-Methoden ab. Dennoch ist oft die Rede von den Web Services, und deswegen werden gleich auch die Web-Technologien hinzugezogen. Die Informationen bezüglich der Methoden auf die mit URIs adressierten Ressourcen verbergen sich in der HTTP-Methode (Ric07, S. 90,101). Allgemein gesehen wird bei einer einheitlichen Schnittstelle nach dem CRUD-Prinzip gearbeitet. Dabei steht CRUD für die gebräuchlichsten Operationen beim REST: Create, Retrieve, Update und Delete (PZL08, S. 807). Diese einzelnen Operationen bezogen auf das Web entsprechen im engeren Sinne den HTTP-Methoden POST, GET, PUT und DELETE (Fie09):

- Create (POST, PUT): Zum Erstellen einer neuen Ressource.
- Retrieve (GET): Zum Holen einer Repräsentation einer Ressource.
- Update (PUT): Zum Verändern von Eigenschaften einer Ressource.
- Delete (DELETE): Zum Löschen einer Ressource.

Die HTTP-Methoden PUT und POST können beide für das Erstellen einer neuen Ressource verwendet werden. Es gibt aber eine Unterscheidung, wann die eine oder andere Methode einzusetzen ist. Will man darauf Einfluss nehmen, welche URI die neue Ressource bekommen soll, so ist die PUT-Methode zu verwenden. Überlässt man die Benennung der Ressource dem Server, dann soll man die POST-Methode einsetzen (Ric07, S. 110-116) (Fie00, S. 81-82) (Fie09).

Die POST-Methode hat auch eine andere Funktion, die außerhalb der Grenzen von REST liegt. Bei der einheitlichen Schnittstelle stellen HTTP-Methoden die Methoden-Informationen

dar. Es ist aber nicht immer möglich, die Methoden-Information in der HTTP-Methode zu übermitteln. Ein Beispiel dafür sind Ressourcen-Repräsentationen in HTML-Format mit HTML-Formularen. Ein HTML-Formular erlaubt nur die HTTP-Methoden GET und POST. Die Methoden-Information zum Ändern oder zum Löschen der Ressource muss in diesem Fall auf eine andere Weise übermittelt werden. Für solche Fälle wird die POST-Methode zur Übertragung der Daten verwendet. Die Methoden-Information kann dabei im URI, HTTP-Header oder Entity-Body kodiert sein. Die Methoden-Information befindet sich in diesem Fall nicht in der HTTP-Methode und die Schnittstelle ist somit nicht mehr einheitlich. (Ric07, S. 112-116)

Es gibt noch andere HTTP-Methoden, die weniger gebräuchlich sind (Ric07, S. 111-112):

- HEAD: Zum Holen der Metadaten beziehungsweise HTTP-Header einer Ressource.
- OPTIONS: Zum Prüfen von unterstützten HTTP-Methoden einer Ressource.
- TRACE: Zum Debuggen von Proxys.
- CONNECT: Zum Weiterleiten von anderen Protokollen über einen HTTP-Proxy.

Die einheitliche Schnittstelle wird durch die Identifizierbarkeit der Ressourcen, Manipulierbarkeit von Ressourcen über Repräsentationen, selbstbeschreibende Nachrichten und HATEOAS ermöglicht (Fie00, S. 82). Die Ressourcen werden mithilfe von URIs eindeutig identifiziert und sind somit adressierbar. So können die Repräsentationen von Ressourcen mit der GET-Methode angefragt werden. Die Repräsentationen können Manipuliert und mit der PUT-Methode an den Server geschickt werden, damit die Änderungen in die Ressourcen auf dem Server einfließen. Mit der POST- oder PUT-Methode können mithilfe von URIs neue Ressourcen angelegt und mit der DELETE-Methode gelöscht werden. Die Repräsentationen von Ressourcen beinhalten alle Daten und Metadaten zur Erledigung der gewünschten Aufgabe. Zu den Daten von Ressourcen gehören auch Verlinkungen von Ressourcen. Über diese Links (URIs) kann der Übergang zu anderen Ressourcen mit der GET-Methode erfolgen.

Layered System

Mit der Schichtenarchitektur eines Systems (Layered System) werden die Funktionalitäten nach dem Abstraktionsgrad in hierarchische Schichten zerlegt. Die Kommunikation bei dieser Architektur erfolgt nur zwischen benachbarten Schichten. Dabei ruft eine Schicht die Funktionalitäten der direkt darunter liegender Schicht ab. Dafür sind die Schnittstellen bei jeder Schicht definiert, die nur der direkt darüber liegender Schicht bekannt sind. Mit der Schichtenarchitektur kann man gegen die Komplexität bei großen Systemen vorgehen. Nachteile dabei sind die mit der Anzahl der steigenden Schichten ebenfalls steigenden Latenzzeiten und Aufwand. Durch Caching der Daten auf den Zwischenknoten auf dem Pfad vom Client zum Server kann jedoch eine Leistungssteigerung erzielt werden, welche die oben genannten Nachteile in den Schatten stellt (Fie00, S. 82-84).

Code On Demand

Ein weiteres optionales Designkriterium von REST ist *Code On Demand*. Repräsentationen der Ressourcen können auch Quellcode beziehungsweise ausführbare Skripte beinhalten. REST-basierten Anwendungen, die *Code On Demand* unterstützen, können somit durch die Benutzung der Web Services in der Funktionalität erweitert werden. So können zum Beispiel die Methoden zum Validieren von Benutzereingaben erst während des Einsatzes von relevanten Web Services der Client-Anwendung übertragen werden. Auf diese Weise wird ein Teil der Funktionalitäten der Client-Anwendung auf dem Server gelagert. Die Client-Anwendung wird um diese Funktionalitäten erst bei Bedarf erweitert (Fie00, S.84-85).

2.1.4 Sicherheit und Idempotenz

Es wurde bereits beschrieben, welche HTTP-Methoden zur Implementierung des CRUD-Prinzips eingesetzt werden. Man kann jedoch diese Methoden auch falsch anwenden, was mit Risiken oder Nachteilen verbunden ist. An dieser Stelle wird auf die Sicherheit und Idempotenz eingegangen und erklärt, wieso sie so wichtig sind.

Sicherheit

Bei richtigen Implementierung und Einsatz der GET-Methode dürfen keine Ressourcen manipuliert werden. Diese HTTP-Methode ist nur für lesende Zugriffe gedacht. Der Client sollte beim Verwenden dieser Methode bei den Anfragen keine Sorgen haben, irgendwas bei dieser Anfrage zerstören zu können. Die GET-Methode gehört zu den sicheren Methoden von der Sicht des Clients und sollte deswegen auch so beim Server implementiert werden. In der Praxis gibt es aber auch Nebeneffekte oder beabsichtigte Veränderung der Ressourcen bei dieser Methode vorzufinden. Es werden bei kleinen Nebeneffekten Log-Datei oder der Zähler der Anfragen verändert und bei einigen Web-Service-Anbietern funktioniert die komplette Manipulation der Ressourcen basierend auf der GET-Methode. Die Nebeneffekte dürfen keine gravierenden Veränderungen von Ressourcen mit sich bringen und der Client sollte dafür auch nicht verantwortlich gemacht werden (Ric07, S. 116-118).

Idempotenz

Mit Idempotenz ist die Eigenschaft einer Operation festgelegt, bei welcher der Ressourcenzustand sich bei mehrfacher Anwendung der Operation nicht mehr ändert. So sollte nach dem mehrfachen Anwenden der GET-, PUT- oder DELETE-Methoden der Ressourcenzustand gleich der einmaliger Anwendung der Methode entsprechen. Dabei soll beachtet werden, dass der Ressourcenstatus bei der PUT-Methode nicht mit relativen, sondern mit absoluten Werten verändert werden darf, um diese Bedingung zu erfüllen. Wenn ein Attributwert einer Ressource 5 ist und man diesen Wert auf 2 setzen möchte, dann soll die Änderung mit dem absoluten Wert 2 und nicht mit dem relativen Wert -3 erfolgen. In einem unzuverlässigen Netzwerk kann eine Änderungsanfrage mehrfach empfangen werden. Die Änderung mit

einem relativen Wert würde bewirken, dass der Attributwert kleiner als 2 wird. Beim absoluten Wert bleibt der Attributwert 2 auch beim mehrfachen Empfang der Änderungsanfrage. Idempotenz ermöglicht zuverlässige HTTP-Anfragen über ein unzuverlässiges Netzwerk. Bei ausstehender Antwort auf die HTTP-Anfrage, sei die verwendete Methode GET-, PUT- oder DELETE, kann ohne weitere Sorgen eine erneute Anfrage abgeschickt werden (Ric07, S. 116-118).

2.2 Software-Prüfung

Jedes in der Entwicklung befindende Anwendungssystem durchläuft verschiedene Testphasen. Bei den Vorgehensmodellen wie Wasserfallmodell in Abbildung 2.1 sieht man unterschiedliche Phasen im Lebenszyklus einer Softwareanwendung. Bevor man zur nächsten Phase im Lebenszyklus übergeht, wird die aktuelle Phase auf die Erfüllbarkeit überprüft. Die Überprüfung beinhaltet darüber hinaus auch integrierten Testprozesse. Beim Aufdecken von Fehlern kann man auch wieder zu der vorherigen Phase wechseln. Die Tests sind im Lebenszyklus einer Softwareanwendung sehr wichtig und sollen in jeder Phase des Lebenszyklus durchgeführt werden (FLS07, S. 15-17).

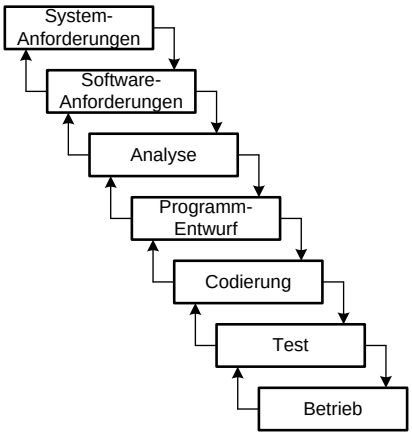


Abbildung 2.1: Testphasen im Software-Lebenszyklus (FLS07, S. 16)

Die Software-Prüfung ist ein sehr umfangreiches Thema und es gibt viele Methoden, Techniken und Verfahren dazu. In diesem Abschnitt wird nur das Grundwissen der Software-Prüfung vermittelt und ein Verfahren für das automatisierte Testen präsentiert. Nachfolgend wird auf die Phasen des Testablaufs eingegangen.

2.2.1 Phasen des Testablaufs

Mit Abbildung 2.2 wird der Zusammenhang der einzelnen Phasen eines Testprozesses graphisch zum Ausdruck gebracht. In der Planungsphase wird festgelegt, was und in welchem

Umfang getestet wird. Es wird der Aufwand geschätzt, die Termine festgelegt und die ausführende Personen bestimmt. Weiterhin werden in dieser Phase Überlegungen über das Ziel, Art und Umfang der notwendigen Tests und auch über die erwarteten Ergebnisse gemacht. Nach der Planungsphase erfolgt der Testablauf, indem Tests vorbereitet, ausgeführt und ausgewertet werden. Die einzelnen Phasen des Testablaufs werden in den nachfolgenden Abschnitten genauer beschrieben. Dem Testablauf folgt die Analyse des nach dem Testlauf erstellten Testberichts. Es werden die in dem Testbericht aufgeführten Fehler analysiert, die Rückschlüsse auf Verbesserungspotenzial in dem Entwicklungsprozess und auf Programmeinheiten zur Überarbeitung erlauben.

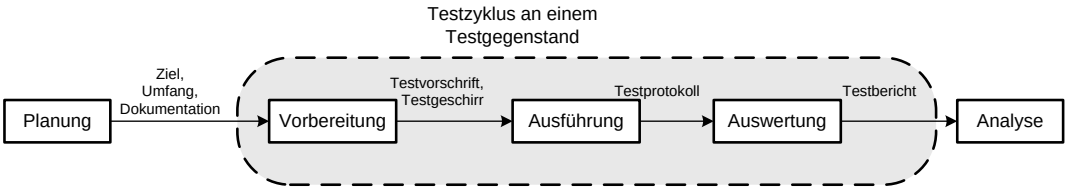


Abbildung 2.2: Semantischer Testablauf (FLS07, S. 37)

Testvorbereitung

Es werden geeignete Testfälle abhängig von der zu prüfenden Funktionalität beziehungsweise funktionaler oder nicht-funktionaler Eigenschaft ausgewählt. Diese Testfälle werden spezifiziert, indem die erforderlichen Vorbedingungen bezüglich eines Prüflings und einer Testumgebung definiert, die Eingabedaten und die erwarteten Ausgabedaten bestimmt werden. Danach wird das Testgeschirr, wie Testdaten und Testwerkzeuge, bereitgestellt. Siehe dazu Abbildung 2.2. Zum Schluss dieser Phase wird nun die Reihenfolge der Testfälle bestimmt. Die Reihenfolge der zu bearbeitenden Testfälle ist optimal, wenn der Aufwand für die Umgestaltungen der Testumgebung für alle zu bearbeitende Testfälle in der Summe, verglichen mit dem Aufwand bei anderen Reihenfolgen, minimal ist (FLS07, S. 37-39).

Testausführung

In dieser Phase werden alle Testfälle, die in der Testvorbereitungsphase vorbereitet waren, ausgeführt und dabei gewonnenen Ergebnisse aufgezeichnet. Am Ende der Testausführungsphase wird ein Testprotokoll, siehe Abbildung 2.2, angefertigt, welches die Daten über den Prüfling, die verwendeten Testfälle, das Testgeschirr und die Ergebnisse der Prüfung beinhaltet (FLS07, S. 39-40).

Testauswertung

Protokollierten Testergebnisse werden ausgewertet, indem sie mit den spezifizierten erwarteten Werten verglichen werden. Als Ergebnis der Testauswertungsphase wird ein Testbericht,

siehe Abbildung 2.2, erstellt. In ein Testbericht gehören die Angaben zu den Testfällen, die Schlussbewertungen und die Verweise auf relevante Dokumente. Testbericht bildet die Grundlage für die Fehlersuche und Fehlerbehebung (FLS07, S. 40).

2.2.2 Qualitätssicherungsmaßnahmen

Software-Qualitätssicherung zerfällt in drei Bereiche. Die *organisatorische Maßnahmen*, die sich bei einem Projekt beziehungsweise Prozess mit der Steigerung der Erfolgsaussichten beschäftigen, die *konstruktive Maßnahmen*, die sich auf die Vermeidung der Fehler konzentrieren, und die *analytische Maßnahmen*, die für das Auffinden von Fehlern zuständig sind. Mit dem REST Testbed sollen bereits entwickelten und noch in der Entwicklung befindenden REST-basierten Client-Anwendungen getestet werden, somit wurden die organisatorischen und die konstruktiven Maßnahmen bereits getroffen. Bei der Qualitätssicherung von REST-basierten Client-Anwendungen mit dem REST Testbed steht das Prüfen beziehungsweise Finden von Fehlern im Vordergrund. Es findet also die Verifikation des entwickelten oder in der Entwicklung befindenden Produktes statt (FLS07, S. 20) (Tie09, S. 471-472).

2.2.3 Prüfungsverfahren

Bei der Softwareprüfung gibt es statische und dynamische Testverfahren. Die statischen Verfahren beschäftigen sich mit der Analyse sowie Prüfung von Systembeschreibungen und sind nicht Teilaufgaben des REST Testbeds. Das REST Testbed unterstützt dynamische Verfahren zur Software-Prüfung. Die Clients sollen Anfragen mit vorgegebenen Daten an den Server schicken. Wenn die Soll-Werte von einem Testergebnis noch nicht bekannt sind, so kann der Test manuell vorbereitet, ausgeführt und ausgewertet werden. Sind Soll-Werte bereits bekannt, so sollte man automatisierte Testfälle erstellen, um Aufwand der Testabläufe zu reduzieren. Durch Abweichungen des Ist-Wertes vom Soll-Wert kann auf diese Weise Fehler aufgedeckt werden, die protokolliert und an das Entwicklungsteam weitergeleitet werden sollen. Der Regressionstest ist ein sehr bekanntes automatisiertes Testverfahren, das für die wiederkehrenden Testabläufe eingesetzt wird (FLS07, S. 22-24).

2.2.4 Regressionstest

Die in der Entwicklung befindende Anwendung wird in zeitlichen Abständen erneut getestet. Zum einen werden neu implementierten Funktionalitäten mit neuen Testskripten überprüft, zum anderen wird mit den alten Testskripten überprüft, ob nicht mit neuen Implementierungen auch Bugs eingebaut wurden. Dabei werden bereits bekannte Soll-Ergebnisse von den früheren Tests, die als korrekt empfunden wurden, manifestiert und bei den Testläufen herangezogen. Der so festgehaltene Soll-Wert wird in einem Testfall zum Vergleich mit dem Ist-Wert genommen. Diese Testfälle können dann überprüft werden, indem die Testskripte von diesen Testfällen automatisch nacheinander ausgeführt werden (FLS07, S. 31-33).

2.3 Aspektorientierte Programmierung

Bei der Umsetzung des Testbeds wurde aspektorientierte Sprache AspectJ benutzt. Um die Implementierung des Testbeds nachvollziehen zu können, werden in folgenden Abschnitten einige Konstrukte von AspectJ erläutert. Der Umfang der Einführung in AspectJ wird kurz gehalten und umfasst nur grundlegende Informationen zum Verständnis des Testbeds.

Die aspektorientierte Programmierung baut auf der objektorientierten Programmierung auf und ermöglicht die Trennung der Geschäftslogik von den zusätzlichen Funktionalitäten. Auf diese Weise können zum Beispiel die Bearbeitung der HTTP-Anfragen zum Auslesen der Ressourcen um weitere Funktionalitäten wie Logging oder Autorisierung erweitert werden. Die zusätzlichen Funktionalitäten werden dabei in eigenem Modul gekapselt und der ursprüngliche Quellcode bleibt unberührt. Dieser Ansatz wird auch als *Separation of Concerns* bezeichnet und kann gegen die steigende Komplexität bei wachsenden Systemen eingesetzt werden. Durch die aspektorientierte Programmierung können Funktionalitäten, die sich sonst der Modularisierung erfolgreich widersetzen und so über die verschiedenen Module in einem System verteil sind, die sogenannten *Crosscutting Concerns*, modularisiert werden (Böh06, S. 7,14-15,17).

Ein Aspekt bei der aspektorientierten Programmierung ist die Erweiterung des Klassen-Konzepts. Die Aspekte sind vergleichbar mit den Klassen bei der objektorientierten Programmierung, in denen zusätzlich zu dem gewöhnlichen Java-Code weitere Sprachkonstrukte definiert werden (Böh06, S. 25).

2.3.1 Joinpoint

Ein Joinpoint stellt ein Ereignis dar, das einen Punkt im Programm definiert, der einer Erweiterung oder Modifikation unterzogen werden soll. Zu den Joinpoints gehören folgende Ereignisse mit dem entsprechenden Schlüsselwort in AspectJ (Böh06, S. 25):

- Aufruf einer Methode (**call**)
- Ausführen einer Methode (**execution**)
- Behandeln einer Exception (**handler**)
- Zugriff auf eine Variable (**set**/**get**)
- Initialisierung einer Klasse (**staticinitialization**/**preinitialization**/**initialization**)

2.3.2 Pointcut

Ein Pointcut stellt einen Sprachkonstrukt zum Vereinigen von Joinpoints. Es können also mehrere durch die Verbindungspunkte definierten Ereignisse zu einer Menge zusammengefasst werden. Dabei gibt es in AspectJ einige logische Operationen und Wildcards, die das Bilden einer Joinpoint-Menge erlauben (Böh06, S. 25-26).

Wildcards (Böh06, S. 56-57):

- Pluszeichen (+): Vertritt alle Unterklassen beziehungsweise Schnittstellen des vorgegebenen Typs
- Stern (*): Repräsentiert eine Folge beliebiger Zeichen (außer dem Punkt)
- Zwei Punkte (..): Repräsentieren eine Folge beliebiger Zeichen (einschließlich dem Punkt)

Logische Operationen (Böh06, S. 58-59):

- Vereinigung (||): Zum Vereinigen von Ausdrücken
- Schnitt (&&): Zum Bilden einer Schnittmenge der Ausdrücke
- Verneinung (!): Zum Verneinen eines Ausdrucks

Der nachfolgende Beispiel zeigt eine praktische Anwendung von Joinpoints mit Wildcards und logischen Operationen:

```
pointcut GET() : execution(* services.testing.get(..)) || execution(* services.config.get(..));
```

In diesem Beispiel wird ein Pointcut *GET()* definiert. Dieser Jointcut wird ausgelöst, wenn eine *get()*-Methode der Klassen *testing* oder *config* im Paket *services* ausgeführt wird. Die zwei Sterne (*) in diesem Ausdruck stehen für beliebige Typen der Rückgabewerte. Die Doppelpunkte (..) weisen auf eine beliebige Anzahl der Übergabeparameter der Methoden *get()*. Mit dem ODER-Operator (||) werden Joinpoint-Mengen der einzelnen execution-Ausdrücke zu einer Menge vereinigt.

Eine Alternative für den beschriebenen Ausdruck könnte aber auch so aussehen:

```
pointcut GET(..) : execution(* services.*.get(..));
```

In diesem Fall steht anstatt des Namens einer Klasse ein Stern. Damit werden alle Klasse im Paket *services* umfasst. Wenn es keine andren Klassen im Paket *services* existieren oder zumindest keine Klassen mit der Methode *get()* gibt, dann sind die zwei beschriebenen Pointcuts *GET()* äquivalent.

2.3.3 Advice

Unter einem Advice versteht man eine Methode, die mit Auslösen eines definierten Ereignisses im Programmcode beziehungsweise mit dem Erreichen eines Joinpoints, ausgeführt werden soll. Dabei kann diese Methode auf unterschiedliche Weise ausgeführt werden (Böh06, S. 26-27) (Kna07):

- before-Advice: Ausführung vor einem *Joinpoint*
- after-Advice: Ausführung nach einem *Joinpoint*

Für ein Beispiel der praktischen Anwendung wird der bereits in Abschnitt 2.3.2 definierter Jointcut GET() verwendet:

before() : GET() // to do ;

Wenn ein Joinpoint des Pointcuts GET() ausgelöst wird, dann wird vor der Ausführung der *get()-Methode* (siehe Abschnitt 2.3.2) die an der Stelle *// to do* definierte Aktion ausgeführt.

2.3.4 Ausführungsreihenfolge

Mit der Reihenfolge der Ausführung kann die Beziehung zwischen den Aspekten festgelegt werden. Die Zeile unten beschreibt einen Konstruktor, mit dem der Vorrang eines Aspekts vor einem anderen Aspekt definiert werden kann:

define precedence: *AspectPatternList*

Definiert man eine Liste von Aspekten mit *AspectPatternList*, so haben links stehenden Aspekte aus der Liste Vorrang vor rechts stehenden Aspekten. Die *before-Advices* der Aspekte mit dem höheren Vorrang werden früher wie die *before-Advices* der Aspekte mit dem niedrigeren Vorrang abgearbeitet. Die *after-Advices* der Aspekte mit dem höheren Vorrang werden dagegen nach den *after-Advices* der Aspekte mit dem niedrigeren Vorrang ausgeführt. Mehrere definierten *Advices* innerhalb eines Aspekts, die von der gleichen Art sind, werden in der Reihenfolge deren Definition ausgeführt (Böh06, S. 150-155).

In Abschnitt 7.3 wird beispielhaft ein Aspekt in Bezug auf das Testbed demonstriert und erklärt, bei dem einige der oben beschriebenen Sprachkonstrukte sich wiederfinden. Man wird dabei deutlicher das Zusammenspiel der einzelnen Sprachkonstrukte in einem Aspekt in Verbindung zueinander verstehen können.

3 Verwandte Arbeiten

In diesem Kapitel werden die Arbeiten betrachtet, die sich mit Aufsetzen von Testumgebungen befassen, die zum Testen von Web Service basierten Client-Anwendungen herangezogen werden können. Die kurzen Beschreibungen von diesen Projekten gewähren einen schnellen Einblick in diese Arbeiten und es werden Aspekte angesprochen, die beim Vorhaben dieser Diplomarbeit von Bedeutung sind. Die erste betrachtende Arbeit konzentriert sich primär auf das Testen von SOAP-basierten Client-Anwendungen. Der zweite Projekt baut auf dem ersten Projekt auf und bietet auch die Funktionalitäten zum Testen von REST-basierten Client-Anwendungen. Beide Projekte weisen einige Ansätze auf, die bei der Entwicklung des REST Testbeds von Interesse sind.

3.1 GENESIS

An der Universität Wien wird schon seit 2008 im Bereich von SOA Testbeds geforscht. Dazu wurden bereits mehrere Arbeiten veröffentlicht. Die Publikation *GENESIS - A Framework for Automatic Generation and Steering of Testbeds of ComplexWeb Services* beschäftigt sich mit dem Aufsetzen von Testbeds für serviceorientierte Architektur (SOA), die auf Basis von SOAP-basierten Web Services umgesetzt wird. Dabei wird ein Framework namens GENESIS präsentiert. Dieses Framework erlaubt die Spezifikation und das Steuern von Testbeds von einem zentralisierten Front-End und die automatische Generierung von verteilten Testbeds in Back-End. Die Funktionalität von SOAP-basierten Web Services kann bei GENESIS mit Plug-ins erweitert werden (JTD08).

SOA konzentriert sich

Mit GENESIS gelieferten Plug-ins:

- QOSPlugin: Simuliert die nicht-funktionalen Eigenschaften respektive die Quality of Service (QoS)-Parameter
- BPELPlugin: Ausführung von zusammengesetzten Prozessen innerhalb von Web Service Operationen
- LogPlugin: Protokollieren von Web Service Aufrufen
- RegistryPlugin: UDDI-Registrierung von Web Services

Es können neue Web Service Beschreibungen angefertigt und einem Testbed zum Erstellen eines Web Services übergeben werden. Die erstellten Web Services können vorhandene Plug-ins in Anspruch nehmen, um verschiedene Verhalten zu simulieren. Durch das Simulieren von QoS-Eigenschaften kann die Suche nach fehleranfälligen Komponenten unterstützt werden.

Das Verhalten des Testbeds in diesem Framework kann während der Laufzeit verändert werden. Das Framework unterstützt automatisches Testen, indem die Parameter von Plug-ins sich mit Hilfe einer Java API während der Laufzeit verändern lassen. Bei der Publikation *Script-based Generation of Dynamic Testbeds for SOA* geht es um die weitere Forschung im gleichen Bereich und die Entwicklung der zweiten Version des Testbeds. Die Funktionalität in GENESIS2 wurde erweitert. Es kann nicht nur ein Testbed, sondern auch weitere Komponenten wie Clients generiert werden. Dazu mussten jedoch das Konzept und die Architektur in der zweiten Version überdacht und verändert werden. Das Framework GENESIS2 wurde generisch gehalten, um in der Zukunft eine Grundlage für die Forschung auch in den nicht-SOA Bereichen zu bieten (JTD08) (JD10).

Mit GENESIS2 gelieferten Plug-ins:

- WebServiceGenerator: Erstellt SOAP-basierte Web Services
- WebServiceInvoker: Ruft entfernte SOAP-basierte Web Services
- CallInterceptor: Zur Bearbeitung von SOAP-basierten Aufrufen auf der Nachrichtenebene
- DataPropagator: Bietet eine automatisierte Replikation von den Daten und Funktionen unter den Back-End-Hosts
- QOSEmulator: Emuliert die Quality of Service (QoS)-Eigenschaften
- SimpleRegistry: Zur globalen Registrierung und Abfrage von Web Services
- ClientGenerator: Zum Aufsetzen von Testbeds mit Standalone-Clients

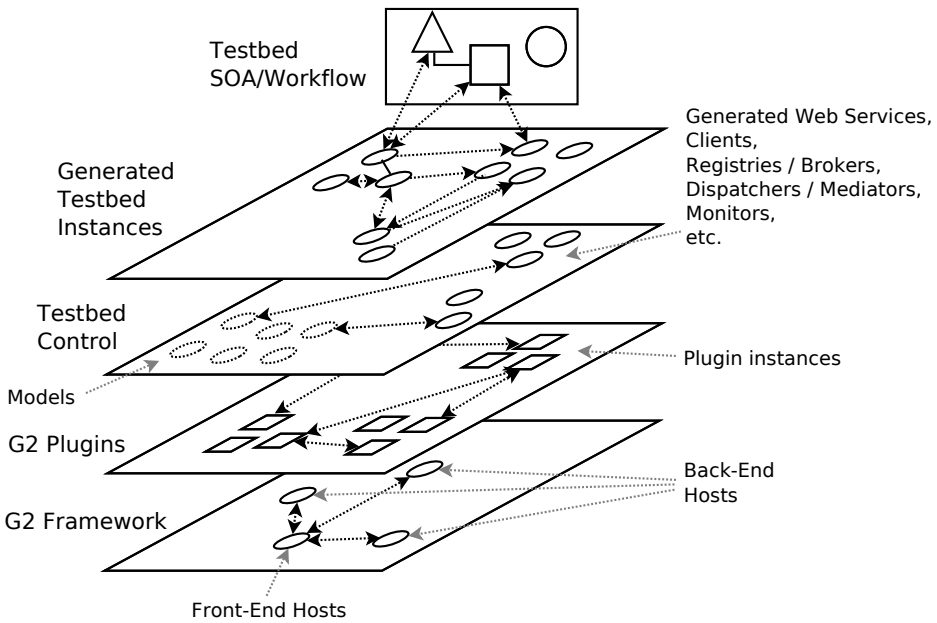


Abbildung 3.1: Konzept von GENESIS2 (GEN)

Abbildung 3.1 zeigt in einer Schichtendarstellung das Konzept von GENESIS2. In der Schicht *G2 Framework* sind das Front-End-Host und die Back-End-Hosts dargestellt. Auf dem Front-End-Host werden die Spezifikationen zu den Testbeds erstellt und in dieser Schicht findet auch die Steuerung von Testbeds statt. Das kann nach der Empfehlung der GENESIS2-Entwickler in der kompakten Skriptsprache Groovy erfolgen. Mithilfe der Spezifikationen werden auf den Back-End-Hosts Testbed-Instanzen generiert. Diese Testbed-Instanzen findet man in der Schicht *Generated Testbed Instances*. Die Schicht *G2 Plugins* verdeutlicht den modularen Ansatz zur Realisierung von den Funktionalitäten basierend auf einem Plug-in-Konzept. In der Schicht *Testbed Control* findet die individuelle Anpassung von den gewünschten Plug-ins bei einzelnen Testbed-Instanzen statt.

Laut der Publikation *Script-based Generation of Dynamic Testbeds for SOA* soll GENESIS2 auch für REST-basierte Web Services geeignet sein. Es soll für dieses Vorhaben dann ein weiteres Plug-in entwickelt werden, das für die Erstellung von REST-basierten Web Services zuständig ist. Des weiteren braucht man weitere Plug-ins für die Realisierung von funktionalen und nicht-funktionalen Eigenschaften, die die Funktionalität von dem erstellten REST-basierten Web Service erweitern würden. Im Unterschied zu GENESIS2 handelt es sich bei REST Testbed um die Erstellung eines Testbeds und nicht wie bei GENESIS2 um die automatische Generierung von mehreren verteilten Testbeds. Die Spezifikation und die Steuerung der Testbeds in GENESIS2 kann von einer entfernter Maschine durchgeführt werden, auf der eventuell auch das zu testende Client-Anwendungssystem läuft. In diesem Fall ist es möglich ein Test Framework aufzusetzen, das die Spezifikation, das Steuern von den Testbeds und die Steuerung des zu testenden Client-Anwendungssystem zum Durchführen von Testaufrufen in den Testfällen verwalten könnte. Vor allem bei den Regressionstests werden solche Testfälle erstellt, siehe auch Abschnitt 2.2.4. Die Spezifikation und Steuerung von Testbeds in GENESIS2 findet programmiertechnisch statt. Für einen programmierunfähigen Software-Tester ist dieses Konzept nicht geeignet, denn es gibt keine Möglichkeit die Steuerung über eine graphische Benutzeroberfläche durchzuführen. Das komponentenbasierte Konzept von GENESIS kann aber auch auf das REST Testbed angewandt werden. Durch die Plug-ins kann die Funktionalität des REST Testbeds somit leicht erweitert werden.

Eine weitere Publikation der Universität Wien, die auf dem Framework GENESIS beziehungsweise GENESIS2 aufbaut und sich auf die aspektorientierte Programmierung konzentriert, heißt *Automating the Generation of Web Service Testbeds using AOP*. Die aspektorientierte Programmierung AOP wird dabei dazu verwendet, um Aufrufe externer SOAP-basierter Web Services während der Laufzeit abzufangen. Dabei wird das empfangene WSDL-Spezifikation analysiert und der beschriebene Web Service im Testbed generiert. Das WSDL-Dokument wird manipuliert, so dass es auf das nun im Testbed befindende Duplikat vom Original-Service verweist, und im Anschluss an die SOA weitergeleitet. Dabei werden Aufrufe der externen Web Services simuliert, um Kosten oder weitere Limitierungen beim Testen zu vermeiden (JD11).

In unserem Projekt brauchen wir zwar keine Aufrufe externer Web Services, und doch bietet die AOP auch bei unserer Problemstellung einige Vorteile. Man kann eine Anwendung auch ohne aspektorientierter Programmierung komponentenorientiert gestalten, jedoch ist Aspect-Oriented Programming (AOP) ein sehr praktisches Konzept, das wir nutzen können. Die *Joinpoints*, an denen Aspekte verwoben werden, bilden die Schnittstellen. Für die

Definition von einem *Joinpoint* siehe Abschnitt 2.3.1. So kann mittels Aspekt-Konstruktoren die Funktionalität des Testbeds sehr komfortabel erweitert oder manipuliert werden, ohne ursprünglichen Quellcode zu verändern. Des weiteren lassen sich *Crosscutting Concerns* mit AOP modularisieren, die sich mit anderen Programmierparadigmen der Modularisierung erfolgreich entziehen, siehe auch Abschnitt 2.3.

3.2 SOA4ALL

Das Forschungsprojekt Service Oriented Architectures for All (SOA4ALL) konzentrierte sich auf die Entwicklung eines vollwertigen webbasierten verteilten Systems namens SOA4ALL Studio, siehe dazu Abbildung 3.2. Dieses Projekt begann im Jahr 2008 und erstreckte sich über drei Jahre. Die Weiterentwicklung wird nicht mehr verfolgt und mittlerweile trifft man auf tote Verweise in den Dokumentationen und auf der Webseite. Durch die Erstellung von konfigurierbaren Testbeds zum Testen der entwickelten Komponenten soll der Entwicklungsprozess mit SOA4ALL Studio unterstützt werden. Mit SOA4ALL Studio können die entwickelten Komponenten auf die funktionalen Eigenschaften und andere charakteristische Eigenschaften wie Skalierbarkeit und Leistung, die sogenannten nicht-funktionalen Eigenschaften, validiert werden. Die Infrastruktur von SOA4ALL Testbed basiert auf dem Projekt GENESIS. Das Projekt GENESIS erfüllte bereits einige Anforderungen, die an SOA4ALL Studio gestellt wurden, und wegen der Erweiterbarkeit mit Plug-ins bot GENESIS eine gute Grundlage für den Entwicklungsstart von SOA4ALL Studio. Das SOA4ALL Studio ist aus den integrierten Werkzeugen, englisch Tools, aufgebaut.

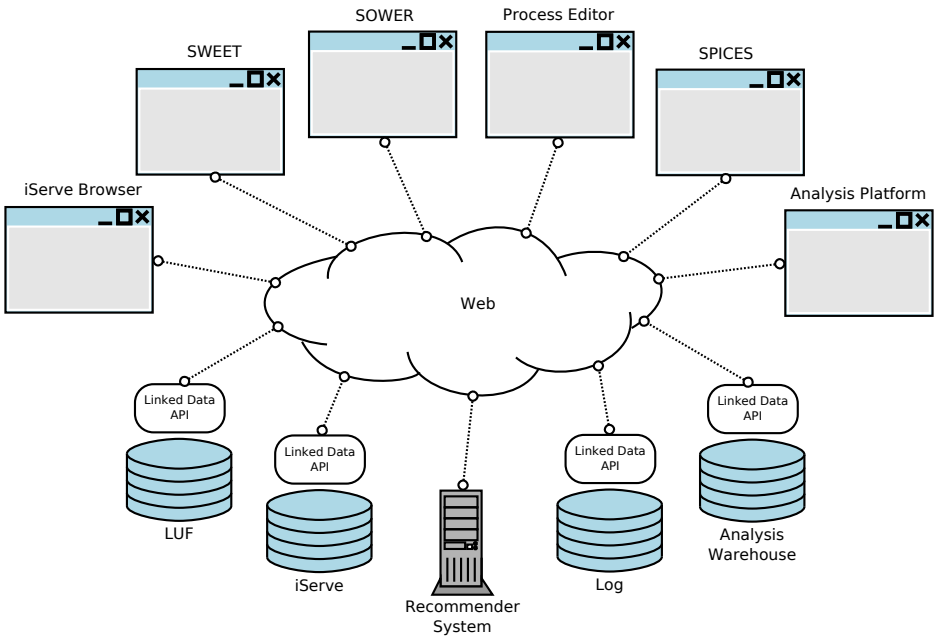


Abbildung 3.2: Architektur von SOA4ALL mit integrierten Tools (Atoa)

Die integrierten Tools sind nach Aufgabenbereichen, entsprechend der drei Phasen des Lebenszyklus eines Services, in drei Segmente unterteilt. Sie bieten Unterstützung beim Lösen unterschiedlicher Aufgaben innerhalb der drei wichtigsten Phasen entlang des Lebenszyklus eines Services.

Aufgabenbereiche des *SOA4ALL Studio*:

- Provisioning Platform: Die Bereitstellung von semantischen Services durch eine Beschreibung oder durch eine Zusammensetzung von existierenden Web Services.
- Consumption Platform: Das Auffinden und Aufrufen von Web Services.
- Analysis Platform: Die Analyse der Ausführung von Web Services.

Diesen drei Segmenten sind verschiedene Tools zugeordnet. Weiter unten sind die Tools nach den oben genannten Segmenten aufgezählt und es werden die Aufgabenbereiche der jeweiligen Tools zusammengefasst. Die Aufgaben der Tools werden je nach Wichtigkeit im Hinblick auf das REST Testbed mehr oder weniger detailliert beschrieben.

Komponenten von *Provisioning Platform* (Atoa):

- iServe: Eine Plattform zur Veröffentlichung von semantischen Beschreibungen von Web Services als *Linked Data*. Bei *Linked Data* wird das Web für die Verbindung mit den relevanten Daten benutzt.
- Process Editor (Composer): Eine Anwendung zur Modellierung von Prozessen und zum Zusammensetzen von Prozessen aus den semantischen Beschreibungen von SOAP- und REST-basierten Web Services mit der Modellierungssprache Literate Programming Markup Language (LPML).
- SOWER: Dieser Editor ermöglicht die manuelle Beschreibung von SOAP-basierten Web Services mit semantischen Informationen.
- SWEET (Semantic Web sErvice Editing Tool): Ein Editor zur Unterstützung der Erstellung von semantischen Beschreibungen von den REST-basierten Web Services (Atoa).

Zur Erstellung von REST-basierten Web Services gibt es in *SOA4ALL Studio* zwei Schablonen. Das *SOA4ALL Studio* benutzt bereits existierende Web Application Description Language (WADL)-Spezifikationen von bekannten REST Application Programming Interface (API)-Anbietern wie eBay REST API etc., um die nötigen Elemente eines REST-basierten Web Services zu erstellen. Abbildung 3.3 demonstrieren beispielhaft ein Fragment, in dem ein Link auf eine WADL-Spezifikationen angegeben ist.

```
<servicetemplates>
  <service name="newsSearchServiceTemplate" type="REST">
    <definition href="NewsSearchService.wadl">
  </service>
</servicetemplates>
```

Abbildung 3.3: Definition eines REST-basierten Web Service (SKA+09)

Nach der Erstellung eines REST-basierten Web Service werden die dazugehörigen Ressourcen definiert. Abbildung 3.4 demonstriert beispielhaft eine praktische Anwendung.

Mit *SWEET* können die oben erklärten Beschreibungen zur Erstellung von REST-basierten Web Services über eine graphische Benutzeroberfläche sehr komfortabel erstellt werden. Der Editor *SWEET* wird in zwei Versionen angeboten. Bei einer Version handelt es sich um ein Plug-in mit abgespeckter Funktionalität für den Webbrowser Firefox. Die andere vollwertige Version wurde als ein Teil des Kernmoduls *Dashboard* entwickelt und kann über einen beliebigen Webbrowser benutzt werden (Atoa) (SWE).

Zur Veröffentlichung und zum Auffinden von semantischen Beschreibungen von Web Services wird iServe verwendet. Unterstützt werden semantische Beschreibungen von SOAP- genauso wie REST-basierten Web Services. Dazu werden einige standardisierte Formate zur Beschreibung von Web Services von den entsprechenden Plug-ins von iServe in ein eigenes Format transformiert. Es werden Formate wie Semantic Annotations for WSDL and XML Schema (SAWSDL) nach World Wide Web Consortium (W3C) (FL07), WSMO-Lite nach W3C (FFK+10) zur semantischen Beschreibung von SOAP-basierten Web Services, MicroWSMO zur Beschreibung von REST-basierten Web Services (LSS+10) und teilweise OWL-S (MBH+04)

```
<resources base="http://api.search.yahoo.com/NewsSearchService/V1/" >
  <resource path="newsSearch" >
    <method name="GET" id="search" >
      <request>
        <param name="appid" type="xsd:string" style="query" required="true" />
        <param name="query" type="xsd:string" style="query" required="true" />
        <param name="type" style="query" default="all" >
          <option value="all" />
          <option value="any" />
          <option value="phrase" />
        </param>
        <param name="results" style="query" type="xsd:int" default="10" />
        <param name="start" style="query" type="xsd:int" default="1" />
        <param name="sort" style="query" default="rank" >
          <option value="rank" />
          <option value="date" />
        </param>
        <param name="language" style="query" type="xsd:string" />
      </request>
      <response>
        <representation mediaType="application/xml" element="yn:ResultSet" />
        <fault status="400" mediaType="application/xml" element="ya:Error" />
      </response>
    </method>
  </resource>
</resources>
```

Abbildung 3.4: Ressourcendefinition für ein REST-basiertes Web Service (SKA+09)

unterstützt. Es stehen zwei Zugriffsmöglichkeiten auf iServe zur Verfügung. Der Zugriff auf die Funktionalitäten von iServe kann über die mit Google Web Toolkit (GWT) entwickelte Anwendung namens *iServe Browser* oder über eine REST-basierte API erfolgen. Dabei bietet die REST-basierte API eine HTML-Darstellung, welche eine benutzerfreundliche manuelle Steuerung unterstützt, und eine Resource Description Framework (RDF)-basierte Darstellung zur automatisierten Steuerung (iSe).

Komponenten von *Consumption Platform* (Atoa):

- anSWERS (a novel Semantic Web-enabled Recommender System): Ein wissensbasiertes System, welches einem Benutzer anhand seines Benutzerprofils und Charakteristiken der Web Services eine Empfehlung zur Nutzung eines ausgewählten Web Services unterbreiten kann.
- LUF (Linked User Feedback): Der Service LUF sammelt Feedback zu den APIs, das in Form einer Bewertung, der Kommentare und Markierungen von den Anwendungsbenutzern entsteht.
- SPICES (Semantic Platform for the Interaction and Consumption of Enriched Services): Ein Web-basiertes Tool für die Interaktion zwischen dem Endbenutzer und den SOAP- beziehungsweise REST-basierten Web Services.

Mit *SPICES* können die mit dem oben erwähnten Tool *iServe* hinterlegten Beschreibungen von SOAP- und REST-basierten Web Services durchsucht werden. Die zugehörigen Web Services können mit dem *SPICES* aufgerufen, bewertet und kommentiert werden.

Komponenten von *Analysis Platform* (Atoa):

- K-Analytics (Knowledge Analytics): Ein webbasiertes Tool zur Visualisierung von *Linked Data* Services und Analysedaten.

- SENTINEL (A Semantic Business Process Monitoring Tool): Ein Tool, das den Stand der Technik in Business Activity Monitoring (BAM) voranbringt, indem es extensiv die Semantik-Technologien zum Unterstützen der Integrität und Ableitung des Geschäftswissens aus den von IT-Systemen generierten Low-Level-Prüfprotokollen benutzt.

Mit den Tools der *Analysis Platform* können die Prozesse beobachtet werden. Es sind unterschiedliche Metriken wie Zeitverzögerung, Aufruffrequenz, Leistung und Benutzerwahrnehmung definiert, die beim Bearbeiten von Prozessen festgehalten werden. Diese Daten sollen den Testern beziehungsweise den Entwicklern helfen, kritische Komponenten zu finden.

Abbildung 3.5 zeigt die Architektur von SOA4ALL mit einzelnen Komponenten. Beim *SOA4ALL Studio* sind die drei oben beschriebenen Plattformen abgebildet. Des weiteren ist die Komponente *User Management* zum Verwalten von Benutzerprofilen dargestellt. Die Informationen aus den Benutzerprofilen helfen bei der Suche nach gewünschten Web Services in *iServe*. Die Komponenten von *SOA4ALL Studio* können über graphische Benutzerschnittstellen und auch über Service-Schnittstellen gesteuert werden. Der im Zentrum dargestellte Enterprise Service Bus (ESB) dient als Infrastruktur-Dienstleister und Integrationsplattform.

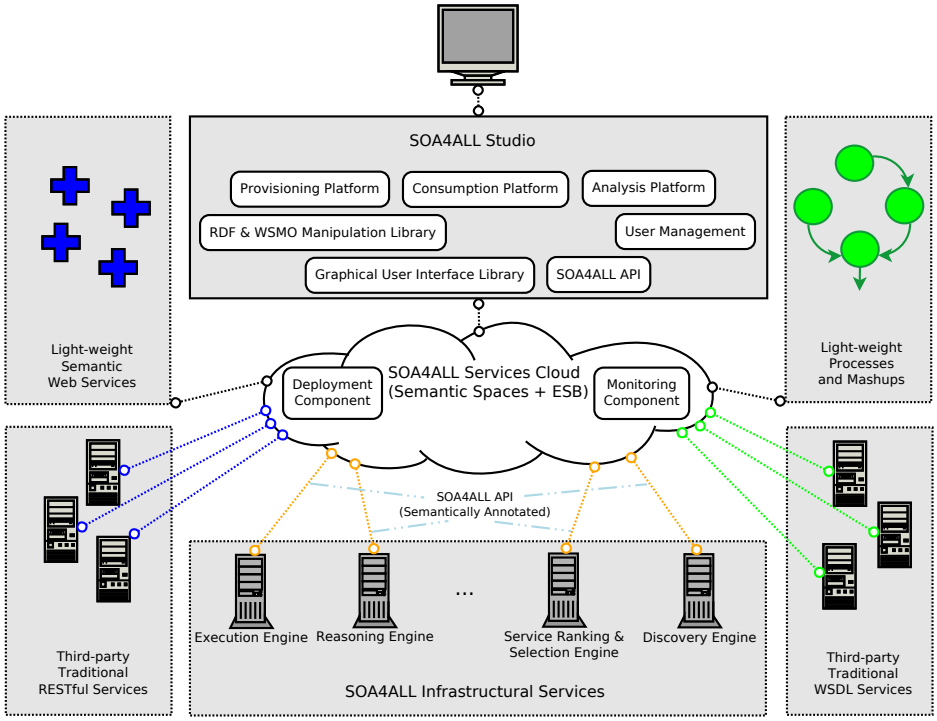


Abbildung 3.5: Architektur von SOA4ALL (Atob)

Beim *SOA4ALL* handelt es sich um ein sehr umfangreiches Projekt, das drei Jahre dauerte und bei dem viele Entwickler beteiligt waren. Es sind einige interessante Komponenten und Ansätze dabei zu finden, die auch beim REST Testbed berücksichtigt werden sollten. Der Zugriff auf *iServe* kann entweder über einen graphischen Editor oder direkt über eine REST-basierte API durchgeführt werden. Zum Konfigurieren der Plug-ins im REST Testbed

könnte ein graphischer Editor eine benutzerfreundliche Schnittstelle für die manuelle Konfigurationsmöglichkeit darstellen. Eine REST-basierte API ist eine passende Schnittstelle für das programmiertechnische Konfigurieren für automatisierte Konfigurationsoption. Beim *SOA4ALL* werden zwei oben angegebene Fragmente (siehe Abbildungen 3.3 und 3.4) benutzt, um ein REST-basiertes Web Service zu erstellen. Dabei wird zuerst anhand einer WADL-Spezifikation ein Web Service erstellt und dann wird die zugehörige Ressource definiert. Beim REST Testbed dagegen sollen die Web Services anhand des gegebenen Datenbestands, aus dem die Ressourcen auszulesen sind, definiert werden. Zur Erstellung von Beschreibungen von Web Service werden beim *SOA4ALL Studio* einige bekannte REST APIs wie eBay REST API etc. benutzt. Das REST Testbed soll dagegen auf keinen bestimmten Datenbestand fixiert sein. Die Funktionalität des REST Testbeds soll zum Benutzen verschiedener Datenquellen wie Datenbanken, Comma-Separated Values (CSV)-Dateien, Web Service APIs etc. erweiterbar sein. Der Ansatz zur Erstellung eines Web Services mittels einer semantischen Beschreibung, wie in den oben angegebenen Fragmenten (siehe Abbildungen 3.3 und 3.4), kann also nicht zum Lösen der gleichen Aufgabe übernommen werden. Dieser Ansatz kann aber bei der vom REST Testbed bereitgestellter REST-basierter API zum Manipulieren von Parametern der Plug-in-Konfigurationen angewandt werden.

4 Anforderungen

In diesem Kapitel werden alle an das zu entwickelnde REST Testbed gestellten Anforderungen detailliert erhoben und aufbereitet, die bereits in Kapitel 1 teilweise angesprochen wurden.

Beim REST Testbed handelt es sich um eine Server-Anwendung, bei der zwei Schnittstellen, englisch API, bereitgestellt werden sollen. Die Schnittstelle *TestServiceAPI* soll für die Ressourcenanfragen für die Client-Anwendung zu Testzwecken zur Verfügung stehen. Die Schnittstelle *ConfigurationAPI* soll für das Konfigurieren der funktionalen und nicht-funktionalen Eigenschaften der Schnittstelle *TestServiceAPI* zuständig sein (siehe Abbildung 1.1). Die API zum Testen der Client-Anwendung soll nach den Konzepten aus Abschnitt 2.1 einen REST-basierten Web Service darstellen, der sich gemäß der aktuellen Konfiguration entsprechend verhalten soll. Die im Folgenden aufgelisteten funktionalen und nicht-funktionalen Eigenschaften sollen von dem REST Testbed abgedeckt werden.

4.1 Funktionale Anforderungen

Mit diesen funktionalen Eigenschaften sollen an die zu testende Client-Anwendung gestellten funktionalen und nicht-funktionalen Anforderungen abgedeckt und getestet werden können. Jede dieser funktionalen Eigenschaften des REST Testbeds soll konfigurierbar sein. Man soll sie aktivieren, konfigurieren und wieder deaktivieren können. Weiter unten sind die funktionalen Eigenschaften aufgelistet, die das REST Testbed bereitstellen soll:

- Die über die Schnittstelle *TestServiceAPI* angefragten Ressourcen sollen aus einem realitätsnahen Datenbestand ausgelesen werden. Die Datenbestands- genauso wie Berechtigungsparameter sollen konfigurierbar sein. Auch die URIs, mit denen die Ressourcen in dem Datenbestand identifiziert werden, sollen definiert und einer Ressource zugeordnet werden können. Das Testbed soll den Benutzern das Auslesen, Modifizieren, Löschen und Anlegen von Datensätzen in dem Datenbestand erlauben.
- Für jede Ressource sollen verschiedene Repräsentationen, etwa HTML nach der HTML 4.01 Spezifikation von W3C (RHJ99), XML in der Version 1.0 nach W3C (BPSM$^+$08) und JSON nach RFC 4627 (Cro06), von den Benutzern des Testbeds über die Schnittstelle *ConfigurationAPI* explizit auswählbar sein.
- Jede Anfrage soll protokolliert werden können. Je nach gewünschtem Grad sollen unterschiedliche Informationen wie Zeitpunkt, Bearbeitungszeit der Anfragen etc. ins Protokoll einfließen. Die gesammelten Informationen sollen bei Bedarf persistent abgespeichert werden und für Testbed-Benutzer für das spätere Einsehen zugänglich sein. Der Testbed-Benutzer soll die Freiheit haben, über die Schnittstelle *ConfigurationAPI*, den

Grad der Informationen und den Speicherort zum Ablegen der Protokolle zu bestimmen. Dieses Protokoll soll die Entwickler der Client-Anwendung bei der Fehlersuche und der Bewertung der Konzepte unterstützen.

- Der Testbed-Benutzer soll über die Schnittstelle *ConfigurationAPI* imstande sein, der Client-Anwendung mitzuteilen, ein Cookie anzulegen. Dabei soll der Testbed-Benutzer die Parameter des Cookie bestimmen können. Die möglichen Parameter und deren Anwendung sind in der Publikation RFC 6265, *HTTP State Management Mechanism*, (KM97) beschrieben.
- Des Weiteren soll der Testbed-Benutzer in der Lage sein, das Verhalten der Client-Anwendung auf Cache-Nutzung zu testen. Deswegen sollen HTTP-Header, die sich auf das Caching-Verhalten der Client-Anwendung auswirken können, über die Schnittstelle *ConfigurationAPI* gesetzt und manipuliert werden können. Die für Caching zuständigen HTTP-Header können dem Kapitel 13 der Publikation RFC 2616, *Hypertext Transfer Protocol – HTTP/1.1*, (Fie09) entnommen werden.
- Es sollen verschiedene Verfahren, etwa wie HTTP-Authentifizierungsverfahren *Basic Access Authentication (HTTP Basic)* und *Digest Access Authentication (HTTP Digest)*, zur Autorisierung der HTTP-Anfragen vom Testbed bereitgestellt werden. Die erforderlichen Parameter für die Authentifizierung der eingehenden HTTP-Anfragen sollen vom Testbed-Benutzer über die Schnittstelle *ConfigurationAPI* konfigurierbar sein. Die oben erwähnten HTTP-Authentifizierungsverfahren sind in der Publikation RFC 2617, *HTTP Authentication: Basic and Digest Access Authentication*, (FHBH+99) definiert.
- Die vom Testbed an die Client-Anwendung gelieferten HTTP-Statusmeldungen auf die HTTP-Anfragen sollen simuliert werden können. Es sollen vom Testbed-Benutzer bei Bedarf vorgegebene HTTP-Statusmeldungen der Serie 2xx, 3xx, 4xx oder 5xx erstellt und an die Client-Anwendung geliefert werden. Dabei soll die Möglichkeit zum Festlegen eines Gültigkeitsbereichs bestehen, binnen dessen die Konfiguration innerhalb des Testbeds wirksam sein soll. Die HTTP-Statusmeldungen sind in Kapitel 10 der Publikation RFC 2616, *Hypertext Transfer Protocol – HTTP/1.1*, (Fie09) definiert.

4.2 Nicht-funktionale Anforderungen

Die an das REST Testbed gestellten nicht-funktionalen Anforderungen, die bei der Architektur, Design und Implementierung berücksichtigt werden sollen, sind nachfolgend aufgelistet:

- Flexibilität: Das Verhalten des Testservices soll an die Bedürfnisse der Client-Anwendung anpassbar sein. Somit soll der Testbed-Benutzer in der Lage sein, das Verhalten vom Testbed durch das Aktivieren, Konfigurieren und Deaktivieren von funktionalen und nicht-funktionalen Eigenschaften von der Schnittstelle *TestServiceAPI* zu steuern.
- Erweiterbarkeit: Die angestrebten funktionalen und nicht-funktionalen Eigenschaften des Testbeds in dieser Diplomarbeit decken nur einen Bruchteil der Technologien ab. Der Umfang des Testbeds soll leicht erweiterbar sein, beispielsweise durch das Hinzufügen neuer funktionaler oder nicht-funktionaler Eigenschaften. Deswegen sollen die

funktionale beziehungsweise nicht-funktionale Eigenschaften als Plug-ins beim REST Testbed integriert werden.

- Bedienbarkeit: Bei der Gestaltung der Konfigurationsschnittstelle *ConfigurationAPI* sollen die Möglichkeiten zur manuellen Manipulation von Parametern eines Plug-ins wie zum Beispiel über eine graphische Benutzeroberfläche und das automatisierte Verändern von Parametern wie zum Beispiel durch Ausführen der Skripte berücksichtigt werden. Deshalb soll eine graphische Benutzerschnittstelle über einen Web-Browser für die manuelle Konfiguration des REST Testbeds bereitgestellt werden. Zusätzlich zur Beschreibung der Parameter soll bei falschen Benutzereingaben auch die Fehlerbeschreibung dem Testbed-Benutzer mitgeteilt werden.
- Technische Anforderungen: Das REST Testbed soll in der Programmiersprache Java mithilfe der Spezifikation Java API for RESTful Web Services (JAX-RS) (HS07) realisiert werden. Konkret wird Java in Version 6 verwendet. Als Webcontainer wird Apache Tomcat benutzt.

4.3 Anwendungsfälle und Anwendungsfall-Diagramm

Die Abbildung 4.1 beschreibt grob das zu entwickelnde System anhand von definierten Anwendungsfällen. Das Testbed soll in der Lage sein, die Liste der verfügbaren Plug-ins, welche konfigurierbare funktionale und nicht-funktionale Eigenschaften darstellen, auszulesen und dem Tester bekannt zu geben. Des Weiteren sollen manipulierbare Parameter einzelner Plug-ins von dem Testbed ausgelesen und dem Tester mitgeteilt werden können. Das Testbed soll auch imstande sein, die vom Tester manipulierten Parameter-Werte eines Plug-ins persistent zu übernehmen. Die oben beschriebenen Aufgaben sollen über die Schnittstelle *ConfigurationAPI* unterstützt werden.

Mit der Schnittstelle *TestServiceAPI* sollen Plug-ins mit den geltenden Parametern der aktuellen Konfiguration abgearbeitet werden, solange die Plug-ins aktiviert sind. Im Anwendungsfall-Diagramm wird die Schnittstelle *TestServiceAPI* mit dem Anwendungsfall *Resource anfragen* assoziiert. Die Anwendungsfälle, die von dem Anwendungsfall *Plug-in aufrufen* abgeleitet sind, beziehen sich auf die zu implementierende Plug-ins.

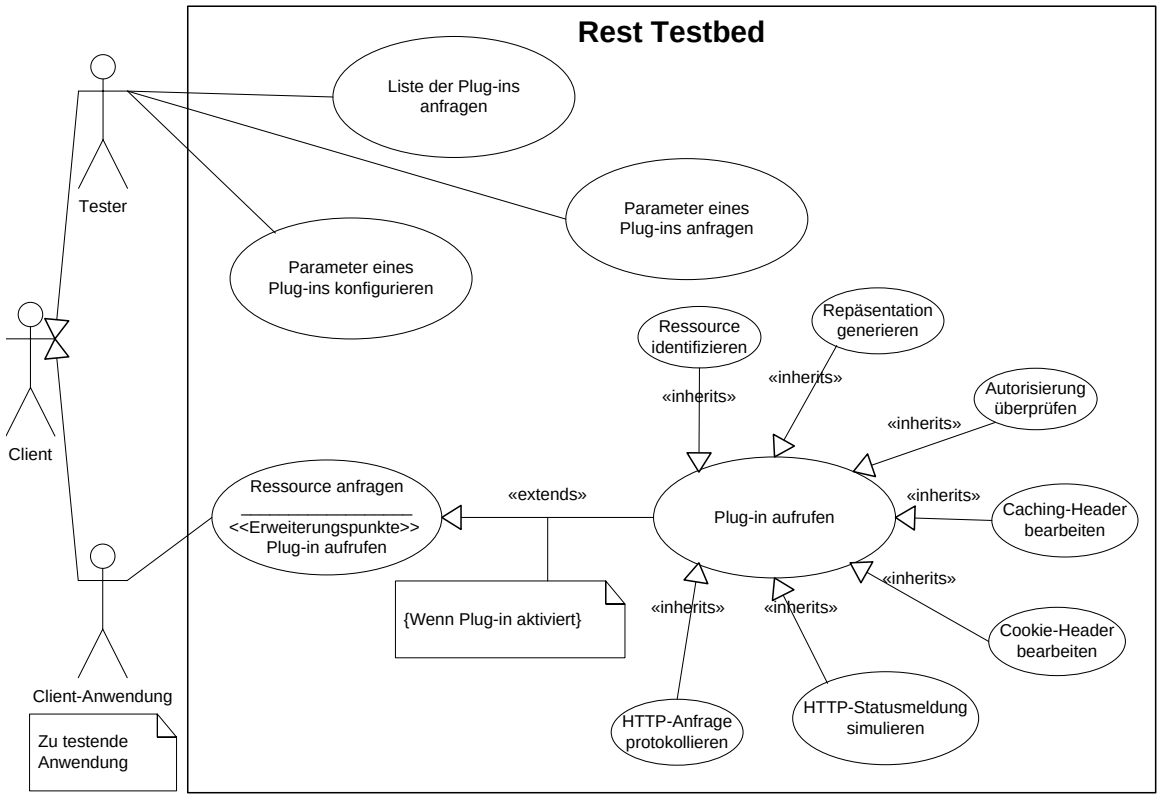


Abbildung 4.1: Anwendungsfall-Diagramm zum REST Testbed

Die Anfragen an das Testbed können unter verschiedenen Bedingungen auch unterschiedliche Verhaltensabläufe respektive Szenarien implizieren. Anwendungsfälle stellen eine Zusammenfassung verschiedener Szenarien unter deren Bedingungen dar (CD08). Weiter unten werden

Anwendungsfälle zur Liste der funktionalen Anforderungen aus Abschnitt 4.1 erstellt. Bei der Nummerierung der Anwendungsfälle ist Folgendes zu beachten. Die erste Zahl bezieht sich auf die Schnittstelle, über welche die beschriebene Funktionalität zu erreichen ist. Bei Nummern 1.x handelt es sich um die Schnittstelle *ConfigurationAPI* und bei 2.x geht es um die Funktionalität der Schnittstelle *TestServiceAPI*. Gemäß der nicht-funktionalen Anforderung der Erweiterbarkeit an das Testbed werden die konfigurierbaren Eigenschaften aus Abschnitt 4.1 weiter als Plug-ins bezeichnet.

Name und Nummerierung	Liste der Plug-ins anfragen 1.1
Beschreibung	Die Liste der Namen aller verfügbaren Plug-ins zur Erweiterung der Funktionalität der Schnittstelle *TestServiceAPI* über die Schnittstelle *ConfigurationAPI* anfordern.
Beteiligte Akteure	Tester (T) und System (S)
Auslöser	T hat Namen der Plug-ins angefragt.
Vorbedingungen	-
Nachbedingung	T hat Namen aller Plug-ins erhalten.
Standardablauf	1. T macht Anfrage nach den Namen der Plug-ins. 2. S liefert die Namen aller Plug-ins.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *ConfigurationAPI* erreichbar sein.

Tabelle 4.1: Anwendungsfall 1.1: Liste der Plug-ins anfragen

Name und Nummerierung	Parameter eines Plug-ins anfragen 1.2
Beschreibung	Die Parameter eines Plug-ins zur Erweiterung der Funktionalität der Schnittstelle *TestServiceAPI* über die Schnittstelle *ConfigurationAPI* anfordern.
Beteiligte Akteure	Tester (T) und System (S)
Auslöser	T hat Parameter eines Plug-ins angefragt.
Vorbedingungen	-
Nachbedingung	T hat die Parameter des angefragten Plug-ins erhalten.
Standardablauf	1. T macht Anfrage nach den Parametern eines Plug-ins. 2. S liefert die Liste der Parameter des gewünschten Plug-ins.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *ConfigurationAPI* erreichbar sein.

Tabelle 4.2: Anwendungsfall 1.2: Parameter eines Plug-ins anfragen

Name und Nummerierung	Parameter eines Plug-ins konfigurieren 1.3
Beschreibung	Die Parameter eines Plug-ins zur Erweiterung der Funktionalität der Schnittstelle *TestServiceAPI* werden den Bedürfnissen vom Tester einer Client-Anwendung angepasst.
Beteiligte Akteure	Tester (T) und System (S)
Auslöser	T fordert S die modifizierten Parameter eines Plug-ins zu übernehmen.
Vorbedingungen	-
Nachbedingung	Die von T eingegebenen Parameter sind konsistent mit den von S übernommenen Parametern.
Standardablauf	1. T passt die Parameter zum gewünschten Plug-in an. 2. T fordert S zur Übernahme der modifizierten Parameter. 3. S gibt positives Feedback über die übernommenen Parameter.
Erweiterungen	3a. S gibt negatives Feedback für die Übernahme der Parameter mit Hinweisen auf fehlerhafte Eingaben. 3a1. T identifiziert die Ursache. 3a2. T beseitigt die Ursache. 3a3. T geht zum Schritt 2.
Hinweis	Diese Funktionalität soll über die Schnittstelle *ConfigurationAPI* erreichbar sein.

Tabelle 4.3: Anwendungsfall 1.3: Parameter eines Plug-ins konfigurieren

Name und Nummerierung	Ressource anfragen 2.1
Beschreibung	Eine Anfrage nach einer Ressource an den Testservice zum Testen der Client-Anwendung bezüglich der Erfüllbarkeit der funktionalen und nicht-funktionalen Anforderungen stellen.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	C soll getestet werden.
Vorbedingungen	Die Parameter der Plug-ins wurden an den aktuellen Testfall angepasst.
Nachbedingung	S hat die Anfrage bearbeitet und C hat darauf eine Antwort bekommen.
Standardablauf	1. S bearbeitet die Anfrage mit der Behandlung der Anwendungsfälle, die von dem Anwendungsfall *Plug-in aufrufen* abgeleitet sind. 2. S liefert C eine Antwort.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein.

Tabelle 4.4: Anwendungsfall 2.1: Ressource anfragen

Name und Nummerierung	Plug-in aufrufen 2.2
Beschreibung	Ein Plug-in zum Testen der Client-Anwendung bezüglich der Erfüllbarkeit einer funktionalen beziehungsweise nicht-funktionalen Anforderung aufrufen.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	S ruft ein Plug-in auf.
Vorbedingungen	Plug-in ist aktiviert.
Nachbedingung	Plug-in wurde behandelt.
Standardablauf	1. S bearbeitet das Plug-in.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein.

Tabelle 4.5: Anwendungsfall 2.2: Plug-in aufrufen

Name und Nummerierung	Ressource identifizieren 2.3
Beschreibung	Überprüfung der Autorisierung des Zugriffs auf den Datenbestand und gegebenenfalls die Identifizierung der angefragten Ressource im Datenbestand.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Auslesen der Ressourcen aus einem Datenbestand ist aktiviert.
Nachbedingung	Plug-in zum Auslesen der Ressourcen aus einem Datenbestand wurde ausgeführt.
Standardablauf	1. S überprüft die Zugriffsberechtigung auf den Datenbestand. 2. S identifiziert die Ressource.
Erweiterungen	2a. S stellt einen nicht-autorisierten Versuch des Zugriffs auf den Datenbestand. 2a1. S teilt eine Fehlermeldung dem C mit. 2b. S kann die angefragte Ressource nicht identifizieren. 2b1. S teilt eine Fehlermeldung dem C mit.
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 1. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.6: Anwendungsfall 2.3: Ressource identifizieren

Name und Nummerierung	Repräsentation generieren 2.4
Beschreibung	Die Repräsentation der mit dem Anwendungsfall *Ressource identifizieren* ausgelesener Ressource wird generiert.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Generieren der Ressourcen-Repräsentationen ist aktiviert. Die Ressource ist aus dem Datenbestand ausgelesen.
Nachbedingung	Plug-in zum Generieren der Ressourcen-Repräsentationen wurde ausgeführt.
Standardablauf	1. S vergleicht den gewünschten Format zur Generierung der Repräsentation der Ressource mit den aktivierten Formaten. 2. S generiert die Repräsentation der entsprechenden Ressource.
Erweiterungen	2a. S stellt eine Nicht-Übereinstimmung der Formate. 2a1. S teilt eine Fehlermeldung dem C mit.
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 2. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.7: Anwendungsfall 2.4: Repräsentation generieren

Name und Nummerierung	Autorisierung überprüfen 2.5
Beschreibung	Die Autorisierung des Ressourcenaufrufs überprüfen
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Überprüfen der Autorisierung der Anfragen ist aktiviert.
Nachbedingung	Plug-in zum Überprüfen der Autorisierung der Anfragen wurde ausgeführt.
Standardablauf	1. S vergleicht das gewünschte Authentifizierungsverfahren mit dem eingestellten Authentifizierungsverfahren. 2.S stellt eine autorisierte HTTP-Anfrage fest.
Erweiterungen	1a. S stellt eine Abweichung des gewünschten Authentifizierungsverfahrens mit dem eingestellten Authentifizierungsverfahren fest. 1a1. S teilt eine Fehlermeldung dem C mit. 2a. S stellt eine nicht-autorisierte HTTP-Anfrage fest. 2a1. S teilt eine Fehlermeldung dem C mit.
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 6. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.8: Anwendungsfall 2.5: Autorisierung überprüfen

Name und Nummerierung	HTTP-Statusmeldung simulieren 2.6
Beschreibung	Eine vorkonfigurierte Statusmeldung als Antwort auf ankommende Anfragen simulieren.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Simulieren von HTTP-Statusmeldungen ist aktiviert.
Nachbedingung	Plug-in zum Simulieren von HTTP-Statusmeldungen wurde ausgeführt.
Standardablauf	1. S ließt die Parameter der zu generierenden Statusmeldung. 1. S erstellt eine HTTP-Statusmeldung und schickt sie an C.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 7. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.9: Anwendungsfall 2.6: HTTP-Statusmeldung simulieren

Name und Nummerierung	Caching-Header bearbeiten 2.7
Beschreibung	Setzen der HTTP-Header zum Steuern des Verhaltens der Client-Anwendung zur Nutzung von Cache.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Setzen der Caching-Header ist aktiviert.
Nachbedingung	Plug-in zum Setzen der Caching-Header wurde ausgeführt.
Standardablauf	1. S ließt die vorkonfigurierten Parameter der HTTP-Header zur Cache-Nutzung. 2. S erweitert die HTTP-Header der HTTP-Antwort um weitere Header zur Cache-Nutzung.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 5. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.10: Anwendungsfall 2.7: Caching-Header bearbeiten

Name und Nummerierung	Cookie-Header bearbeiten 2.8
Beschreibung	Mitteilen der Cookies an die Client-Anwendung.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Setzen der Cookie-Header ist aktiviert.
Nachbedingung	Plug-in zum Setzen der Cookie-Header wurde ausgeführt.
Standardablauf	1. S ließt die vorkonfigurierten Parameter der mitzuteilenden Cookies. 2. S erweitert die HTTP-Header der HTTP-Antwort um weitere Header mit Cookies.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 4. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.11: Anwendungsfall 2.8: Cookie-Header bearbeiten

Name und Nummerierung	HTTP-Anfrage protokollieren 2.9
Beschreibung	Protokollieren der HTTP-Anfrage einer Ressource.
Beteiligte Akteure	Client-Anwendung (C) und System (S)
Auslöser	Eine Ressource wurde von C angefragt.
Vorbedingungen	Plug-in zum Protokollieren der HTTP-Anfragen ist aktiviert.
Nachbedingung	Plug-in zum Protokollieren der HTTP-Anfragen wurde ausgeführt.
Standardablauf	1. S ließt die vorkonfigurierten Parameter zum Protokollieren der HTTP-Anfragen. 2. S protokolliert die HTTP-Anfrage mit dem vorgegebenen Informationsgrad in der voreingestellen Ausgabequelle.
Erweiterungen	-
Hinweis	Diese Funktionalität soll über die Schnittstelle *TestServiceAPI* erreichbar sein. Dieser Anwendungsfall bezieht sich auf die 3. funktionale Eigenschaft aus Abschnitt 4.1

Tabelle 4.12: Anwendungsfall 2.9: HTTP-Anfrage protokollieren

4.4 Sequenzdiagramm

In der Abbildung 4.2 sind einige mögliche Abläufe beschrieben, um das Testbed besser zu verstehen. Die Abläufe sind in einer logischen Reihenfolge von oben nach unten dargestellt, diese ist jedoch nicht zwingend erforderlich. Wie am Anfang dieses Kapitels beschrieben, kann der Benutzer über die Schnittstellen *ConfigurationAPI* und *TestServiceAPI* mit dem REST Testbed kommunizieren. Auf dem Sequenzdiagramm sind vier unterschiedliche Hauptabläufe zum Konfigurieren der Plug-ins und zu einem Testaufruf dargestellt. Bei der Initialisierung des Testbeds werden auch alle Plug-ins initialisiert und in einer Liste abgelegt, auf welche die Schnittstellen *ConfigurationAPI* und *TestServiceAPI* Zugriff haben. Wie die Verwaltung der Liste der Plug-ins realisiert wird, ist zu diesem Zeitpunkt noch nicht festgelegt und ist auf dem vorliegenden Diagramm nicht abgebildet.

Der dargestellte Ablauf dient dem Auslesen der Namen aller Plug-ins des Testbeds. Mit der gestellten Anfrage *getPluginNames()* über die Schnittstelle *ConfigurationAPI* wird in der *ConfigurationAPI* aus der Liste der verfügbaren Plug-ins von jedem Plug-in der Name ausgelesen und die Liste der Namen aller Plug-ins wird an den Anfragesteller geschickt.

Der zweite Ablauf von oben beschreibt das Auslesen der Parameter einzelner Plug-ins. Dabei muss der Name des gewünschten Plug-ins übergeben werden. Wird das gewünschte Plug-in gefunden, dann werden alle Parameter dieses Plug-ins dem Client geliefert.

Bei dem dritten Ablauf von oben handelt es sich um das Überschreiben alter Parameter eines Plug-ins mit neuen Werten. Nachdem die Parameter von dem Plug-in übernommen sind,

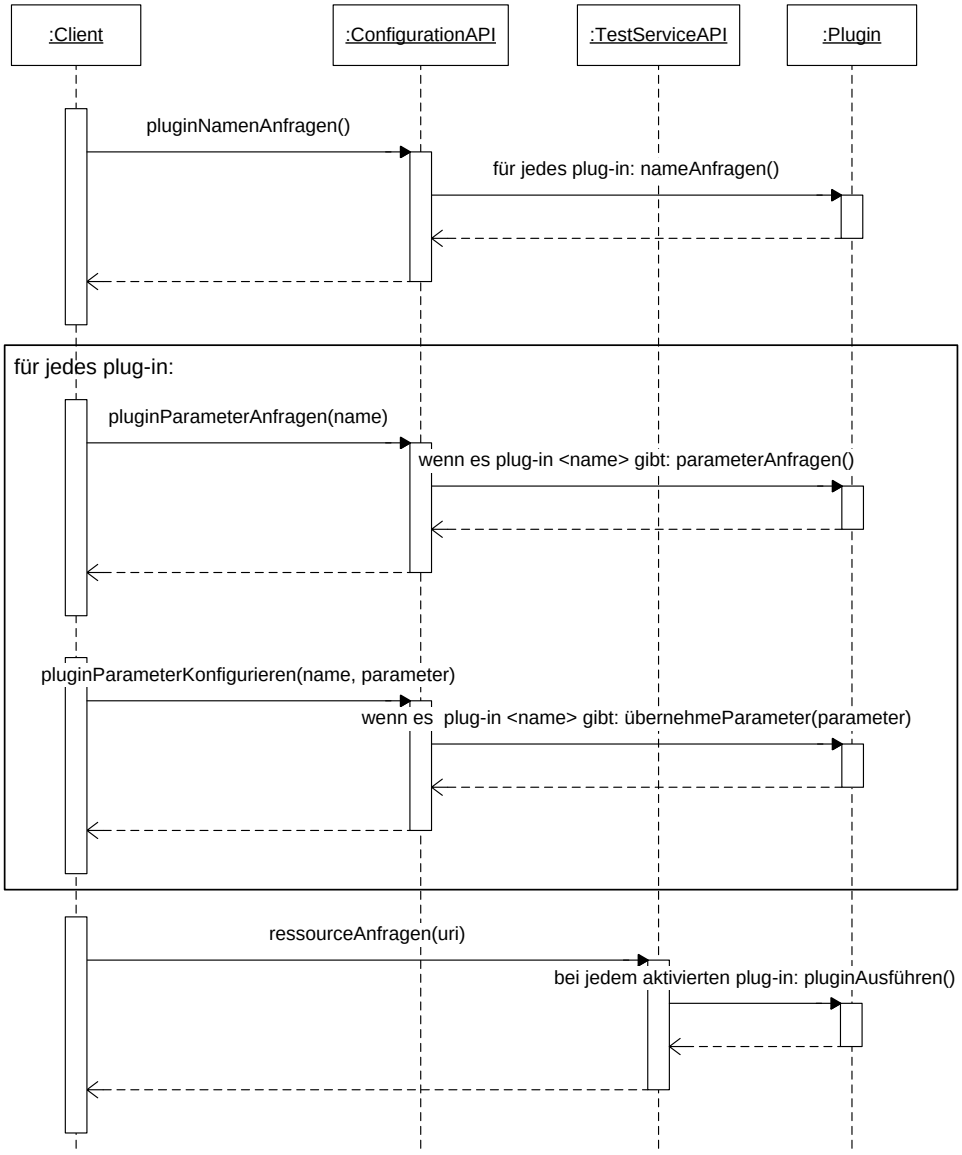


Abbildung 4.2: Sequenzdiagramm zum REST Testbed

werden sie auch persistent abgespeichert. So bleiben die Konfigurationen der Erweiterungen auch über dem Neustart des Testbeds bestehen.

Die mittleren Abläufe zum Auslesen und Anpassen der Parameter eines Plug-ins sollen für alle verfügbaren Plug-ins wiederholt werden. Danach ist das Testbed an ein bestimmtes Testfall angepasst und der Testaufruf kann vorgenommen werden.

Der unterste Ablauf beschreibt den Aufruf des Testservices über die Schnittstelle *TestServiceAPI*. Dem Aufruf des Testservices folgt die Überprüfung der einzelnen Plug-ins aus der Liste der verfügbaren Plug-ins in der *TestServiceAPI* nacheinander. Wird bei einem Plug-in festgestellt, dass sein Status auf *Aktiviert* konfiguriert ist, so wird dieses Plug-in bei der Bearbeitung der Anfrage beziehungsweise der Erstellung der Antwort ausgeführt.

5 Konzept und Architektur

In diesem Kapitel wird auf das Zusammenwirken der einzelnen Komponenten der Architektur vom REST Testbed eingegangen. Es werden die relevanten Konzepte zur Gestaltung des REST Testbeds dargelegt und das Datenmodell zur Verwaltung der Konfigurationen von Plug-ins vorgestellt.

Beim REST Testbed handelt es sich um eine Web-Anwendung, die auf einem Server ausgeführt wird. Diese Tatsache impliziert die Verwendung des Client-Server-Konzepts. Das Client-Server-Konzept bezieht sich primär auf die Unterteilung der Funktionalitäten in zwei Zuständigkeitsbereiche und nicht in zwei Standorte. Der Client kann auf der gleichen Maschine laufen wie der Server. Wenn man die üblichen Client/Server-Anwendungen von einem Anwendungssystem betrachtet, so sind sie oft aufeinander zugeschnitten. Beim REST Testbed soll durch die Konfigurierbarkeit der Plug-ins die Anpassung der Eigenschaften des Testbeds an eine Client-Anwendung stattfinden. Es ist keine mehrfache Benutzung des Testbeds durch mehrere Testbed-Benutzer, im Gegensatz zu den üblichen Server-Anwendungen, vorgesehen, denn die Konfiguration der mit den Plug-ins realisierten funktionalen und nicht-funktionalen Eigenschaften muss durch einen Tester kontrolliert durchgeführt werden.

In Abbildung 5.1 wird die grobe Architektur vom REST Testbed, genauso wie die relevanten Werkzeuge für die Konfiguration des Testbeds und für das Testen von Client-Anwendungen skizziert. Die rechte Seite der Abbildung namens *Testbed Server* liefert einen groben Überblick über die Komponenten des REST Testbeds. Mit der linken Seite der Abbildung namens *Client-Side* geht diese Abbildung über die Grenzen des REST Testbeds hinaus, hilft jedoch es besser dem Testprozess zuzuordnen.

Mit *Client-Side* wird das zu testende Client-Anwendung mit möglichen Hilfsmitteln zum Unterstützen des Testvorgangs dargestellt. Die Aufgabenbereiche der Software-Komponenten dieser Seite der Abbildung fallen nicht in die Zuständigkeit dieser Diplomarbeit und werden deswegen nicht im Detail erläutert. Die Komponente *ConfigurationAPI* stellt die Konfigurationsschnittstelle für das Testbed dar. Über unterschiedliche Repräsentationen von Konfigurationsressourcen kann das Testbed beziehungsweise jedes einzelne Plug-in konfiguriert werden. Dabei werden mit *Client-Side* zwei Arten der Konfiguration verdeutlicht. Die eine Art der Konfiguration wird von dem Tester (Strichmännchen) über die graphische Benutzeroberfläche (Configuraiton GUI) manuell durchgeführt. Die andere Art bezieht sich auf die automatisierte Durchführung der Konfiguration mittels eines *Test-Skripts*. Die Einstellungen von Plug-ins werden in dem mit *Plug-ins Configurations* bezeichnetem Speicher persistent abgelegt. Auf diese Weise können die Konfigurationen von Plug-ins auch bei einem Neustart des Servers bestehen bleiben.

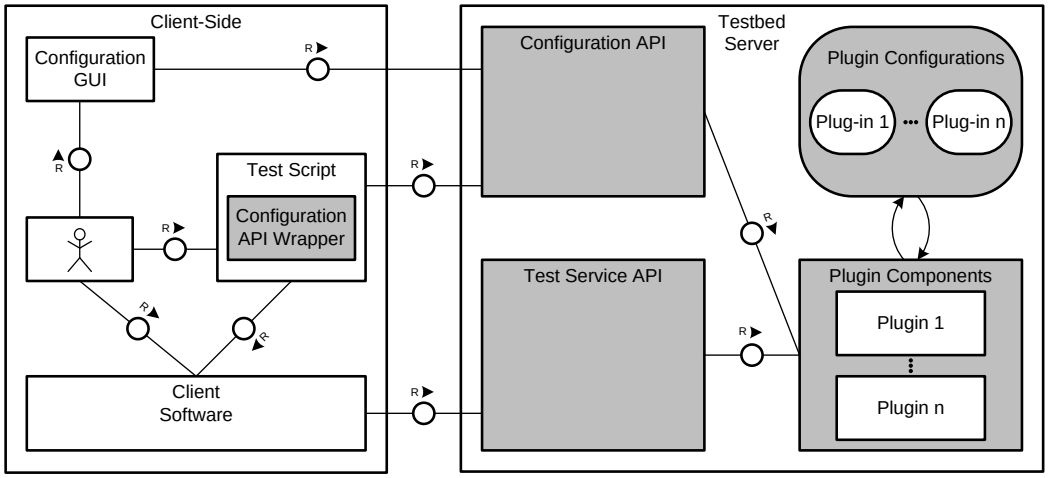


Abbildung 5.1: Die Architektur von dem Testbed

5.1 Plug-in-Konzept

Das REST Testbed basiert auf dem Plug-in-Konzept. Die Entwicklung des Testbeds kann dadurch modular gestaltet werden, was auch die Erweiterbarkeit des Testbeds zusichert. Zu den Kern-Plug-ins gehören die Plug-ins zur Verwaltung von Ressourcen (siehe Anwendungsfall 2.3), zur Erstellung von Repräsentationen von Ressourcen (siehe Anwendungsfall 2.4) und zum Protokollieren von Aufrufen (siehe Anwendungsfall 2.9). Die anderen Plug-ins haben oft Abhängigkeiten bezüglich dieser drei Kern-Plug-ins, denn bei den meisten Aufrufen des Testservices wird eine Ressource aus dem Datenbestand ausgelesen und die Repräsentation von dieser Ressource an die Client-Anwendung geschickt. Dabei wird dieser Prozess ab der Ankunft der Anfrage bis zum Abschicken der Antwort protokolliert. Die meisten weiteren Plug-ins erweitern die Funktionalität dieser Kern-Plug-ins (siehe Anwendungsfall 2.5, Anwendungsfall 2.7 und Anwendungsfall 2.8). Die Ausnahmen bei dem oben beschriebenen Vorgang bildet das Plug-in zum Simulieren einer HTTP-Statusmeldung (siehe Anwendungsfall 2.6), welches keine Repräsentation und somit auch keine Ressource benötigt.

Beim REST Testbed wurden verschiedene Technologien beziehungsweise Standards zum Realisieren von funktionalen und nicht-funktionalen Eigenschafen eingesetzt. Diese Eigenschaften können genauso, nach den Bedürfnissen, mit anderen Technologien beziehungsweise Standards bewerkstelligt werden. Zum Autorisieren der Anfragen werden nach den gestellten Anforderungen die Verfahren *Basic Access Authentication* und *Digest Access Authentication* eingesetzt. Wenn bei der Client-Anwendung ein anderes Verfahren zum Autorisieren der Anfragen verwendet wird, so soll das REST Testbed um dieses Verfahren erweitert werden. Weiteres Beispiel liefert die Generierung der Repräsentationen von Ressourcen. Viele Client-Anwendungen arbeiten mit einer spezifischen Formatierung der Repräsentationen von Ressourcen. Um solche Client-Anwendung zu testen, soll das REST Testbed um die Generierung von Repräsentation mit der entsprechenden Formatierung erweitert werden. Die Entkopplung der einzelnen Komponenten des Testbeds wirkt sich in diesem Fall sehr

positiv auf die Erweiterbarkeit des REST Testbeds aus. Das Plug-in-Konzept erweist sich für diese und viele andere Fälle als sehr praktischer Ansatz bei der Entwicklung. Das Testbed lässt sich mit diesem Konzept leichter um neue Funktionalitäten, die zum Testen einer Client-Anwendung benötigt werden, erweitern.

Es wird zwischen folgenden Komponenten bei einem Plug-in unterschieden:

- Beobachter: Zum Definieren der Ereignisse, nach deren Erscheinung die Domainlogik abgearbeitet wird
- Domainlogik: Die implementierte Domainlogik, mit der sich das Plug-in beschäftigt
- Konfiguration: Zum Parametrisieren von Operationen der Domainlogik

Der Kern des REST Testbeds besteht aus der Verwaltung von Plug-in-Konfigurationen und dem Auslösen von Ereignissen zum Ausführen von Plug-ins. Die antreffenden Anfragen auf dem REST Testbed über die Schnittstellen *TestServiceAPI* lösen Ereignisse aus. Bei den Ereignissen handelt es sich um den Anfang und das Ende der Ausführung der in der Schnittstelle *TestServiceAPI* definierten Methoden. Wenn ein Ereignis, das heißt Bearbeitungsstart oder -ende einer Methode der Schnittstelle *TestServiceAPI*, ausgelöst wird und es gibt einen Beobachter eines Plug-ins, welches auf dieses Ereignis wartet, dann wird die entsprechende Domainlogik mit Hinzunahme der konfigurierbaren Parametern aus der Konfiguration dieses Plug-ins abgearbeitet. Einige Plug-ins müssen den anderen Plug-ins vorgeschaltet werden. Das ist der Fall bei dem Plug-in *Authorization*, das die Autorisierung der Anfrage vor der eigentlichen Bearbeitung überprüfen muss. Einige Plug-ins müssen erst auf die Abarbeitung anderer Plug-ins warten, bevor sie ausgeführt werden. Das Plug-in *Caching* kann erst dann ein *Entity-Tag* berechnen, wenn der Inhalt der HTTP-Antwort bereits erstellt wurde. Bei anderen Plug-ins wie Logging müssen einige Operationen unmittelbar nach dem Antreffen der Anfrage und andere unmittelbar vor dem Abschicken der HTTP-Antwort durchgeführt werden. Dementsprechend soll ein Plug-in zustande sein, deren Methoden zu zwei unterschiedlichen Zeiten der Bearbeitung einer HTTP-Anfrage ausführen zu können.

Die Beobachter nehmen Ereignisse wahr. Wenn ein erwartetes Ereignis von einem Beobachter festgestellt wird, dann wird die zugehörige Domainlogik ausgeführt. Bei den Beobachtern wird die Reihenfolge der Ausführung definiert. Abhängig davon, ob das Ereignis den Anfang oder das Ende der Bearbeitung der *TestServiceAPI*-Operation signalisiert, ändert sich die Reihenfolge der Anwendung der Domainlogik von den Plug-ins bei der Bearbeitung einer HTTP-Anfrage. Wenn es sich um das Start-Ereignis handelt, dann werden die nach dem Vorrang sortierten Plug-ins, angefangen mit dem Plug-in mit dem höchsten Vorrang, nacheinander ausgeführt. Wenn es jedoch um das Ende-Ereignis geht, dann wird die umgekehrte Reihenfolge der Ausführung von Plug-ins verglichen mit der Reihenfolge bei dem Start-Ereignis verwendet. Ein Plug-in mit dem höchsten Vorrang vor anderen bekannten Plug-ins wird beim Start-Ereignis also vor allen anderen Plug-ins und im Falle des Ende-Ereignisses als Letzter ausgeführt. Abbildung 5.2 stellt den Ablauf der Ausführung von Plug-ins graphisch dar.

Eine Anwendung, das nach dem Plug-in-Konzept entwickelt wurde, kann leichter in der Funktionalität erweitert werden. Durch die Modularisierung beim Plug-in-Konzept kann die Anwendung in unterschiedliche Module nach den Zuständigkeitsbereichen aufgeteilt werden.

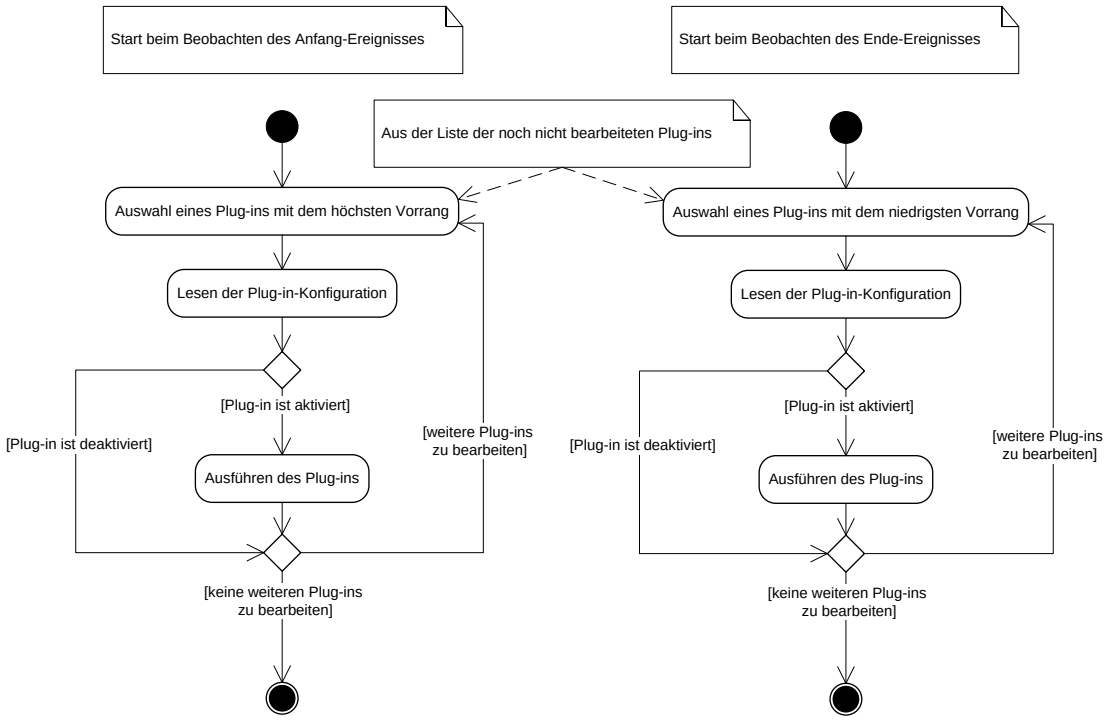


Abbildung 5.2: Abarbeitung von Plug-ins

Die Trennung der Module nach den Zuständigkeitsbereichen verringert die Komplexität bei der Weiterentwicklung und Fehlersuche.

5.2 MVC-Konzept

Bei der Umsetzung des Testbeds wird das Model View Controller (MVC)-Konzept angewandt. Bei diesem Konzept handelt es sich um eine Vorschrift zur Strukturierung einer Software-Entwicklung in drei Komponenten. Die Komponente *Model* beinhaltet dabei Daten und die notwendige Kernfunktionalität zur Verwaltung dieser Daten. Zu jeder Komponente *Model* kann es mehrere *Views* geben. Die Komponente *View* ist für die Darstellung der Daten aus einem *Model* zuständig. Die Komponente *Controller* sorgt dafür, dass die Änderungen in der *View* der Daten eines *Models* sich auch auf die Daten dieses *Model* auswirken. Zur automatischen Benachrichtigung der *Views* über Änderungen in den Daten wird beim MVC-Konzept der Entwurfsmuster Beobachter in *Models* verwendet (PBG04, S. 212) (HR02, 248-251). Auf die Benachrichtigung und somit das Entwurfsmuster Beobachter wird jedoch verzichtet. Diese Eigenschaft des vollwertigen Konzepts von MVC bringt beim REST Testbed keine Vorteile, denn die Konfiguration von den Plug-ins muss durch den Tester unter selbstständiger Kontrolle durchgeführt werden. Abbildung 5.3 zeigt die Komponenten des MVC-Architekturmusters.

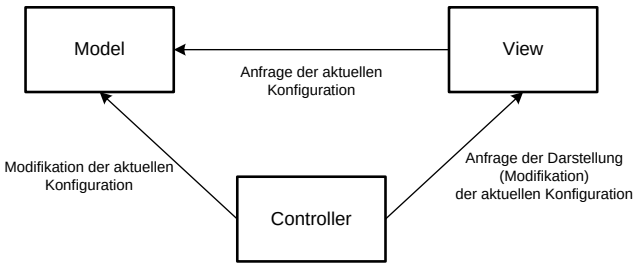


Abbildung 5.3: Model-View-Controller (HR02, S. 249)

Gemäß der Anforderungen sollen beim REST Testbed die Möglichkeiten zum manuellen und automatisierten Konfigurieren der Plug-ins berücksichtigt werden. Es können dafür verschiedene Repräsentationen von Konfigurationen verwendet werden. Bei diesen zwei Darstellungsformen handelt es sich um weiter unten beschriebene *Views* aus dem MVC-Konzept. Nach diesem Konzept bleibt das Testbed um weitere Darstellungen der Konfigurationsdaten leicht erweiterbar.

Beim Testen der Client-Anwendung werden Testressourcen über die Schnittstelle *TestServiceAPI* abgerufen. Dabei werden aktuelle Konfigurationen der Plug-ins überprüft und die entsprechenden Plug-ins für die Bearbeitung der Anfrage und zur Erstellung der Antwort mit vorgegebenen Parametern aus der ausgelesener Konfiguration aktiviert oder deaktiviert. Den Konfigurationen der Plug-ins ist eine definierte Struktur vorgegeben, die im Abschnitt 5.3 beschrieben wird. Dieses Datenmodell ist dabei der Komponente *Model* aus dem bereits erwähnten *Model View Controller* zugeordnet.

Bei der Entwicklung des REST Testbeds sollen zwei Ausgabeformate, und somit zwei *View*-Alternativen gemäß Abschnitt 4.1 realisiert werden. Das eine Ausgabeformat muss für das manuelle und das andere für das automatisierte Konfigurieren der Plug-ins gut geeignet sein. Mit dem MVC-Konzept kann das REST Testbed leicht um weitere *Views* erweitert werden, denn dieses Konzept impliziert diese Eigenschaft. Der *Controller* ist für die Übernahme von Modifikation der Konfigurationen von Plug-ins, genauso wie für das Bereitstellen der Hinweise auf fehlerhafte Eingaben an die *Views*, zuständig.

5.3 Datenmodell der Plug-in-Konfigurationen

Für die Beschreibung einer Konfiguration eines Plug-ins vom REST Testbed wird das in Abbildung 5.4 skizzierte Datenmodell verwendet. Diese Beschreibungen sollen persistent abgespeichert werden, um beim Neustarten des REST Testbeds die Plug-ins nicht erneut konfigurieren zu müssen.

Den Ausgangspunkt in dieser Abbildung stellt die Komponente *Plug-in Configuration* dar. Diese Komponente soll die Charakteristik einer konfigurierbaren funktionalen Eigenschaft, wie in Abschnitt 4.1 angegeben, beschreiben. Dabei soll jedes Plug-in einen eindeutigen

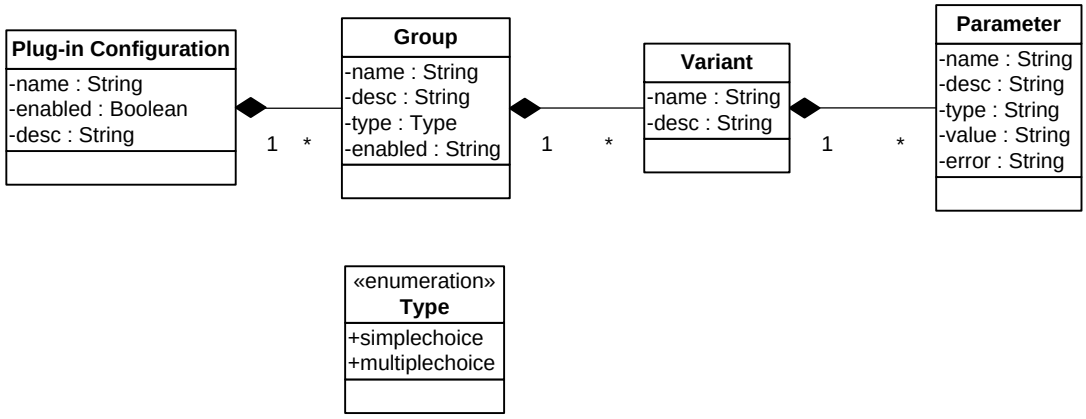


Abbildung 5.4: Datenmodell der Plug-in-Konfigurationen

Namen besitzen, um die einzelnen Plug-ins voneinander unterscheiden zu können. Das Attribut *enabled* sagt darüber aus, ob die Domainlogik vom entsprechenden Plug-in beim Bearbeiten der Operationen von *TestServiceAPI* abgearbeitet werden soll oder nicht. Das Attribut *desc* liefert die Beschreibung des Plug-ins, zu dem die Konfiguration gehört. Die Komponente *Plug-in Configuration* kann mehrere Komponenten *Group* haben, die wiederum mehrere Komponenten *Variant* besitzen können.

Eine Gruppe (Group) kann einem der zwei Typen zugeordnet werde. Der Typ *multiplechoice* erlaubt das voneinander unabhängige Aktivieren der Realisierungsvarianten (*Variant*). Der Typ *singlechoice* erlaubt das Aktivieren nur einer einzigen Realisierungsvariante einer gegebenen Gruppe. Das Attribut *enabled* der Komponente *Group* liefert dabei die Liste der aktivierten Realisierungsvarianten. Für eine angefragte Ressource kann das REST Testbed verschiedene Repräsentationen (zum Beispiel HTML- und XML-Repräsentation) generieren. In dem Fall kann sich der Client für eine oder mehrere Repräsentationen entscheiden. Deswegen handelt es sich dabei um eine Gruppe vom Typ *multiplechoice*. Bei der Autorisierung wird nur ein Verfahren zur Authentifizierung (zum Beispiel *HTTP Basic* oder *HTTP Digest*) festgelegt, so das der Client keine Wahlmöglichkeiten hat. In diesem Fall wird für die gegebene Gruppe der Typ *singlechoice* verwendet. Die Variable *desc* soll eine Beschreibung der jeweiligen Gruppe bereitstellen.

Die Menge der Realisierungsvarianten beziehungsweise die Realisierung der funktionalen oder nicht-funktionalen Eigenschaft eines Plug-ins wird in Abbildung mit der Komponente *Variant* dargestellt. Jede Realisierungsvariante muss einen eindeutigen Namen besitzen, um die einzelnen Varianten voneinander zu unterscheiden. Das Attribut *desc* soll durch eine Beschreibung der Realisierungsvarianten eine unterstützende Funktion bei der Auswahl der Varianten übernehmen. Jede Realisierungsvariante ist einer Gruppe *Group* zugeordnet und besitzt eine bestimmte Anzahl von benötigten Parametern.

Die Parameter einer Realisierungsvariante sind zum Parametrisieren der entsprechenden Realisierungsvariante gedacht. Die in den Attributen *value* gehaltenen Werte der einzelnen Parameter lassen sich über die Schnittstelle *ConfigurationAPI* verändern. Somit können die

Werte der Parameter je nach Testfall angepasst werden. Mit dem Testfall kann dabei das Testen einer funktionalen oder nicht-funktionalen Eigenschaft einer Client-Anwendung gemeint sein. Jeder Parameter besitzt einen eindeutigen Namen in der jeweiligen Variante und einen Typ. Das Attribut *type* beinhaltet die Bezeichnung des in dem Plug-in definierten Validierer von dem Attribut *value* des gleichen Parameters. Zu jedem Attribut *value* jedes Parameters soll ein Validierer bereitgestellt werden. Im Falle einer manuellen Konfiguration eines Plug-ins mit Hilfe einer graphischen Benutzeroberfläche sollen die Benutzereingaben beziehungsweise die Attribute *value* von Parametern clientseitig auf die Korrektheit überprüft werden. Das Attribut *desc* soll eine knappe und aussagekräftige Beschreibung des Parameters liefern. Die Beschreibungen der Parameter und die Benutzung der Validierer sollen die Benutzerfreundlichkeit beziehungsweise Bedienbarkeit beim manuellen Konfigurieren, vor allem über eine Graphical User Interface (GUI), den Testbed-Benutzer unterstützen. Die vom Client übermittelten Parameter sollen auch auf dem Server validiert werden. Über eventuell auftretende serverseitige Fehler und deren Ursache kann der Client über das Attribut *error* informiert werden. Im Gegensatz zu den anderen Attributen wird das Attribut *error* nicht persistent abgespeichert.

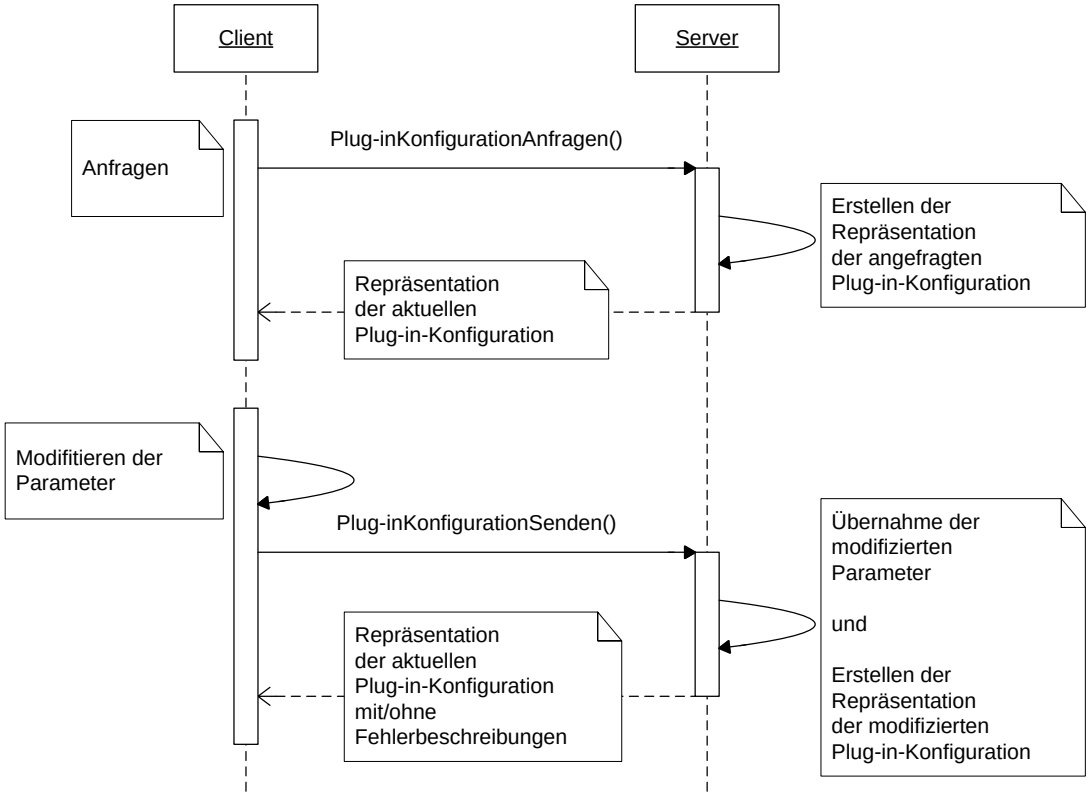


Abbildung 5.5: Szenario zum Modifizieren einer Plug-in-Konfiguration

Abbildung 5.5 beschreibt ein mögliches Szenario beim Manipulieren einer Plug-in-Konfiguration nach dem oben beschriebenen Datenmodell. Der Testbed-Benutzer fragt den aktuellen Zustand einer Plug-in-Konfiguration über eine Client-Anwendung an. Der Server generiert anhand einer Instanz nach dem Datenmodell der Plug-in-Konfigurationen die Reprä-

sentation der entsprechenden Plug-in-Konfiguration und übermittelt diese an die Client-Anwendung. Der Testbed-Benutzer führt nach eigenen Bedürfnissen Änderungen in der Plug-in-Konfiguration, indem die Werte des Attributs *value* in der vorliegenden Repräsentation verändert werden. Mit den in den Attributen *type* gelieferten Werten hat die Client-Anwendung die Möglichkeit die Benutzereingaben zu überprüfen und den Testbed-Benutzer über falsche Benutzereingaben zu informieren. Es ist auch sinnvoll, dass die Client-Anwendung das Absenden der neuen Werte verhindert, wenn die Benutzereingaben fehlerhaft sind. Wenn die modifizierten Werte an den Server abgeschickt und dort empfangen wurden, so sollten diese Werte auch auf dem Server genauer überprüft werden. Können die modifizierten Werte nicht übernommen werden, dann sollte in das Attribut *error* zu dem fehlerhaften Parameter die Fehlerbeschreibung geschrieben werden. Der Wert des Attributs *error* kann dann in die Repräsentation der Plug-in-Konfiguration eingearbeitet und an den Client geschickt werden.

6 Design

Mithilfe von Klassen- und Sequenzdiagrammen wird in diesem Kapitel das Design des REST Testbeds präsentiert. Dabei werden Zuständigkeitsbereiche der in Kapitel 5 vorgestellten Komponenten *Plug-in*, *Model*, *View* und *Controller* besser erklärt und es wird ein Konstrukt zum Verwalten der Konfigurationen und der Domainlogik von Plug-ins vorgestellt. Es wird darauf hingewiesen, dass in den Klassendiagrammen dieses Kapitels zugunsten der besseren Übersichtlichkeit nicht alle Attribute und Methoden bei den angegebenen Klassen aufgelistet sind.

6.1 Zentralisierter Container für Plug-ins

Wie bereits in Abschnitt 5.1 erwähnt, besteht ein Plug-in aus den Komponenten Beobachter, Konfiguration und Domainlogik. Zur Verwaltung der Domainlogik und der Konfigurationen von Plug-ins wird ein Konstrukt benutzt, das als ein zentralisierter Container zur Verfügung steht. Eine Plug-in-Konfiguration wird als Modell nach dem MVC-Architekturmuster realisiert, siehe dazu Abschnitt 5.2, und wird von den Komponenten wie *Views* und *Controllern* des MVC-Architekturmusters genauso wie von der Domainlogik der Plug-ins verwendet. Es wird sicher gestellt, dass es nur eine Instanz des Containers im REST Testbed gibt, auf die verschiedene Komponenten zugreifen können. Dafür kommt das Entwurfsmuster *Singleton* (Einzelstück) zum Einsatz. Das *Singleton*-Entwurfsmuster garantiert, dass es nur eine Einzelinstanz der Klasse *PluginContainer* gibt, und bietet einen globalen Zugriffspunkt auf diese Einzelinstanz. Mit diesem Entwurfsmuster wird es möglich nur eine Instanz der Klasse zum Verwalten einer Konfiguration von jedem Plug-in zu erstellen. Dasselbe gilt auch für die Instanzen der Klassen der Domainlogik von verfügbaren Plug-ins.

Die Klasse *PluginContainer* in Abbildung 6.1 realisiert das beschriebene Architekturmuster *Singleton* und bietet den Komponenten des REST Testbeds einen globalisierten Zugriff auf die Domainlogik von verfügbare Plug-ins und deren Konfigurationen. Weiterhin ist durch die Möglichkeit eines globalisierten Zugriffs auf die Plug-in-Konfigurationen über *PluginContainer* kein erneutes Auslesen der Konfigurationsdateien notwendig. Die erneuten Zugriffe auf die Plug-in-Parameter erfolgen deswegen schneller. Das Verwalten von zum Beispiel zwei Instanzen der Konfigurationen einer Konfigurationsdatei würde Synchronisation erfordern, die eine zusätzliche Fehlerquelle darstellen würde. Das bleibt durch die Anwendung des beschriebenen Entwurfsmusters erspart.

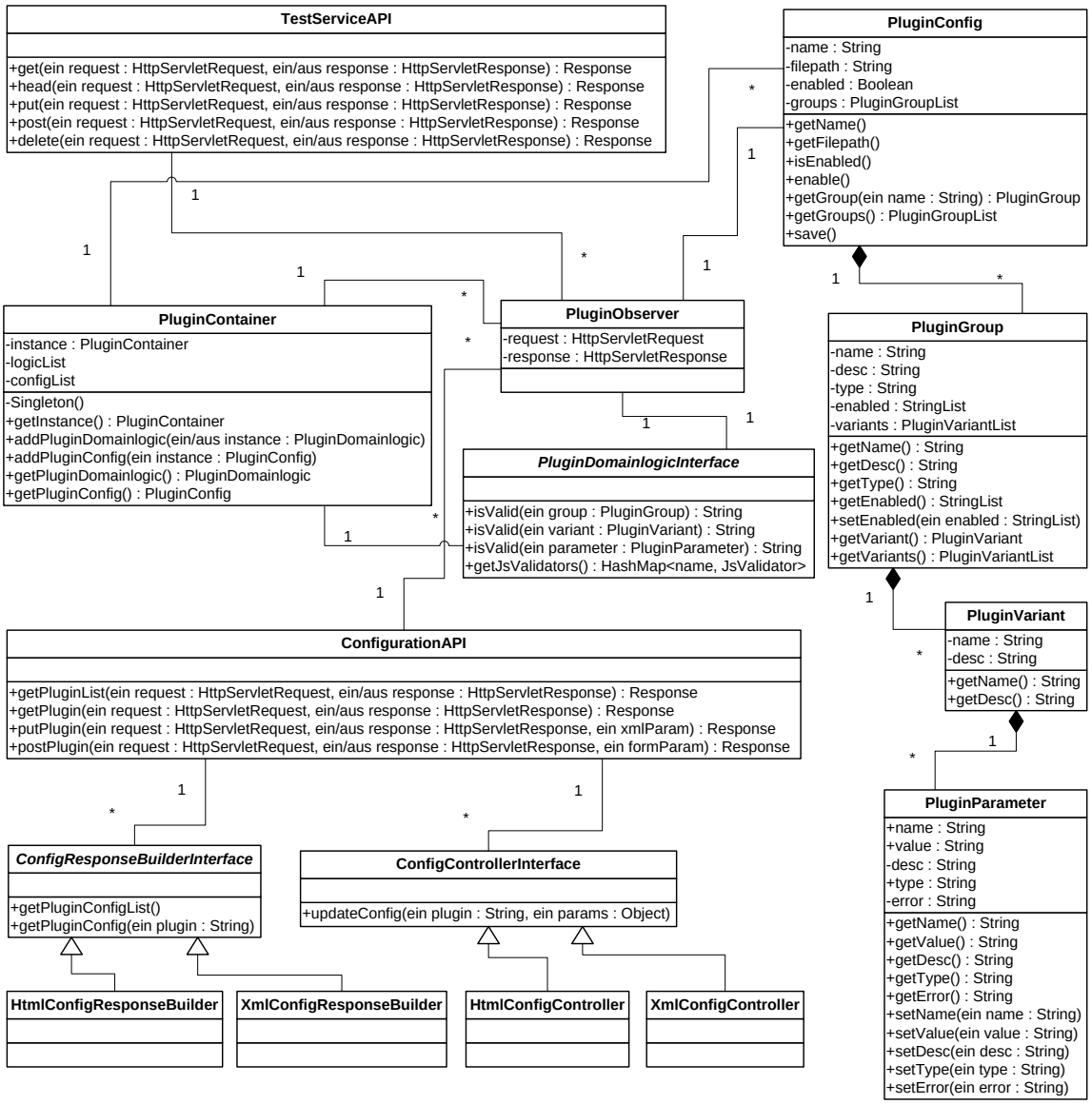


Abbildung 6.1: Klassendiagramm zum Testbed

6.2 Views von Plug-in-Konfigurationen

Die Schnittstelle *ConfigResponseBuilderInterface* in Abbildung 6.1 schreibt die Methoden vor, welche von den Klassen *Html-* und *XmlConfigResponseBuilder* zu implementieren sind. Die Klassen *Html-* und *XmlConfigResponseBuilder* sind für die Erstellung der Repräsentationen respektive Views von Plug-in-Konfigurationen aus den Instanzen der Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und *PluginParameter* zuständig. Die Methode *getPluginConfigList()* liefert eine View, welche die Namen der verfügbaren Plug-in-Konfigurationen beinhaltet. Bei der Methode *getPluginConfig(plugin)* wird eine View mit den Werten der angefragten Plug-in-Konfiguration generiert und geliefert. Das Sequenzdiagramm 6.2 stellt eine mögliche Interaktion der Komponenten bei der Anfrage einer Repräsentation einer Plug-in-Konfiguration über die Schnittstelle *ConfigurationAPI* dar. Mit der Methode *getPluginConfig(name)* wird der *XmlConfigResponseBuilder* zum Erstellen einer Repräsentation einer Plug-in-Konfiguration veranlasst. Der *XmlConfigResponseBuilder* fragt beim *PluginContainer* die gewünschte Plug-in-Konfiguration (*PluginConfig*) an und liest deren Parameter zum Erstellen der Präsentation. Aus der Domainlogik werden die clientseitigen Validierer der entsprechenden Parameter entnommen und der erstellenden Repräsentation hinzugefügt. Die erstellte Repräsentation der angefragten Plug-in-Konfiguration wird darauf dem Client übermittelt.

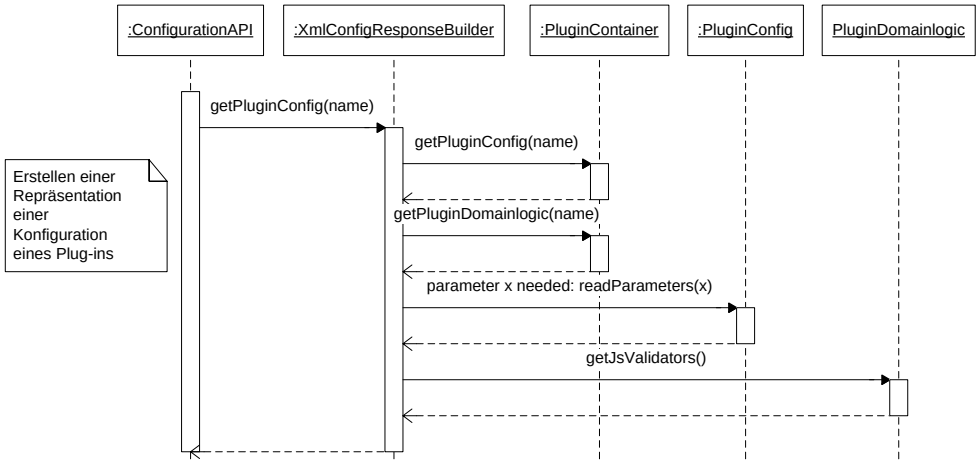


Abbildung 6.2: Auslesen von Plug-in-Konfigurationen über *ConfigurationAPI*

6.3 Controller von Plug-in-Konfigurationen

Die Benutzereingaben werden mit den *Controller*-Klassen, welche die Schnittstelle *ConfigControllerInterface* implementieren, in die Plug-in-Konfigurationen übernommen (siehe Abbildung 6.1). Die zu übernehmenden Parameter einer Plug-in-Konfiguration werden der Methode *updateConfig()* als Parameter übergeben. Diagramm 6.3 beschreibt eine mögliche Interaktion zwischen Komponenten beim Übernehmen der Benutzereingaben beziehungsweise neuen

Parametern in eine Plug-in-Konfiguration. Die Methode *updateConfig(name, params)*, welcher Benutzereingaben mitgeteilt werden, fragt die zu modifizierende Plug-in-Konfiguration (*PluginConfig*) und die Domainlogik implementierende Klasse beim *PluginContainer* an. Die Parameter der mitgeteilten Plug-in-Konfiguration werden dann nach der Überprüfung mit den *isValid(..)*-Methoden der Domainlogik vom *XmlConfigController* aktualisiert. Danach kann eine Repräsentation der aktualisierten Plug-in-Konfiguration wie in Abbildung 6.2 dargestellt generiert werden und dem Client übermittelt werden.

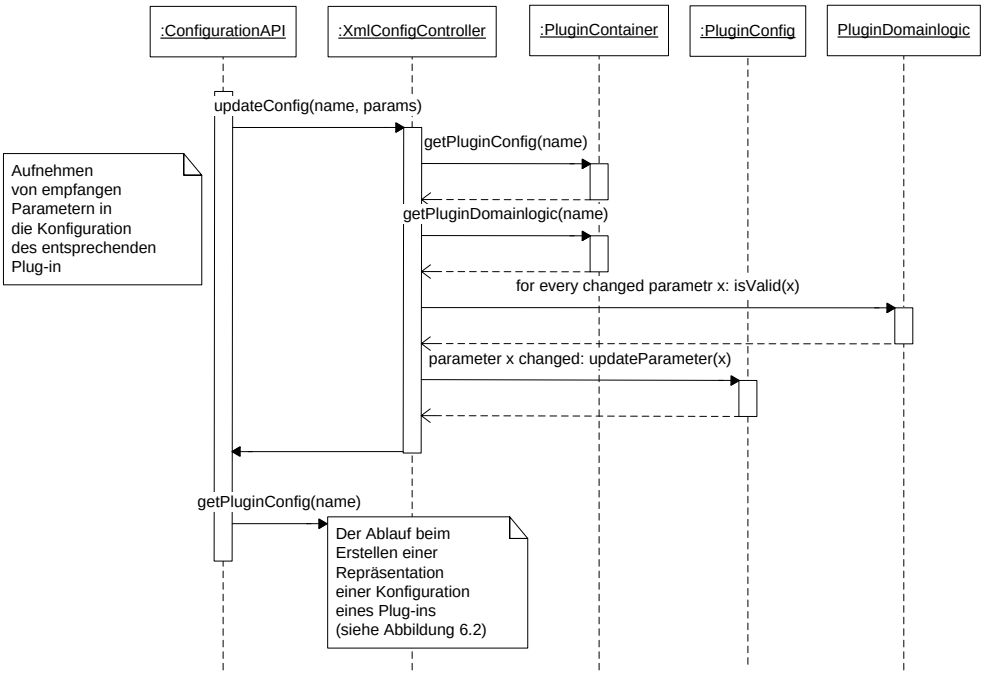


Abbildung 6.3: Modifizieren von Plug-in-Konfigurationen über *ConfigurationAPI*

6.4 Modelle von Plug-in-Konfigurationen

Bei der Klasse *PluginConfig* mit den Klassen *PluginGroup*, *PluginVariant* und *PluginParameter* in Abbildung 6 handelt es sich um die Realisierung des Datenmodells für Plug-in-Konfigurationen, die im Abschnitt 5.3 beschrieben ist.

Um das MVC-Konzept deutlicher zu machen, werden betroffene Klassen den einzelnen Komponenten der *Model-View-Controller*-Architektur zugeordnet. Die Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und *PluginParameter* gehören der Komponente *Model* an. Die Klassen, welche die Schnittstelle *ConfigResponseBuilderInterface* implementieren, stellen *Views* dar. Und die Klassen zum Manipulieren der Plug-in-Konfigurationen, welche die Schnittstelle *ConfigControllerInterface* implementieren, sind dem *Controller* zugeordnet.

6.5 Plug-ins

Wie oben bereits erwähnt, besteht ein Plug-in aus einem Beobachter, einer Domainlogik und einer Konfiguration. Im Folgenden werden die Domainlogik der einzelnen Plug-ins vorgestellt und die Aufgaben der Plug-in-Konfigurationen genauso wie Beobachtern erklärt.

6.5.1 Konfigurationen von Plug-ins

Eine Konfiguration eines Plug-ins ist auch ein Modell nach dem MVC-Konzept (siehe Abschnitt 6.4). Eine Plug-in-Konfiguration beinhaltet konfigurierbare Parameter, die zum Parametrisieren der Methoden der Domainlogik beim Bearbeiten der HTTP-Anfragen herangezogen werden (siehe auch Abschnitt 6.4).

6.5.2 Beobachter von Plug-ins

Über die Schnittstelle *TestServiceAPI* werden die HTTP-Anfragemethoden HEAD, GET, PUT, POST und DELETE abgefangen. Bei der Klasse *PluginObserver* handelt es sich um die in Abschnitt 5.1 beschriebenen Beobachter. Ein Beobachter erwartet die oben genannten HTTP-Anfragen und ist Teil von jedem Plug-in. Wenn von einem Beobachter ein erwartetes Ereignis wahrgenommen wird, dann wendet er gemäß der aktuellen Plug-in-Konfiguration die Domainlogik vom Plug-in zur Bearbeitung der HTTP-Anfrage an. Wenn gleichzeitig zwei oder mehrere Beobachter auf ein Ereignis reagieren, so werden die zugehörigen Plug-ins in der definierten Reihenfolge abgearbeitet. Das Konzept der Reihenfolge der Bearbeitung von Plug-ins ist in Abschnitt 5.1 beschrieben. Außerdem übernehmen die Beobachter die Aufgabe der Instanziierung der Domainlogik und der Konfiguration eines Plug-ins. Der Beobachter soll genauso diese Instanzen im *PluginContainer* ablegen, damit andere Komponenten diese Instanzen erreichen können ohne die Klassen erneut instanziieren zu müssen.

Sequenzdigramm 6.4 zeigt die Interaktion der Komponenten vom Testbed beim Abarbeiten der Domainlogik eines Plug-ins bei der Bearbeitung einer HTTP-Anfrage über die Schnittstelle *TestServiceAPI*. Ein Beobachter (*PluginObserver*) stellt ein erwartetes Ereignis fest. Im Regelfall handelt es sich bei den erwarteten Ereignissen um die Aufrufe einer definierten Methode der Schnittstelle *TestServiceAPI*. Der Beobachter überprüft daraufhin, ob die Domainlogik (*Domainlogic*) gemäß aktueller Plug-in-Konfiguration (*PluginConfig*) abgearbeitet werden soll. Wenn die Domainlogik abgearbeitet wird, dann wird die Plug-in-Konfiguration für die Parametrisierung der Methoden zur Bearbeitung der Domainlogik benutzt. Dieser Ablauf wird für alle Beobachter wiederholt, wenn mehrere Beobachter auf das gleiche Ereignis reagieren.

6.5.3 Domainlogik von Plug-ins

Dieser Abschnitt widmet sich der Domainlogik einzelner Plug-ins, mit denen funktionale Anforderungen an das Testbed aus Abschnitt 4.1 abgedeckt werden. Die Domainlogik wird

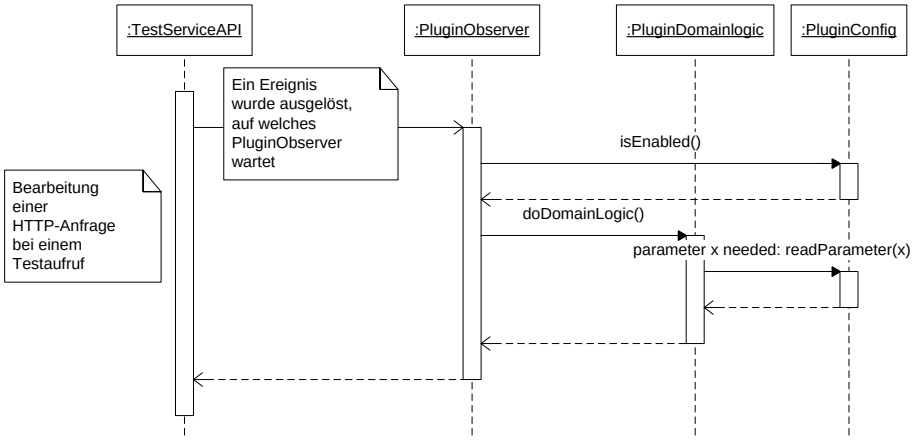


Abbildung 6.4: Bearbeitung einer HTTP-Anfrage über *TestServiceAPI*

mit der die Schnittstelle *PluginDomainlogicInterface* implementierenden Klasse und zusätzlich anderen in der Verbindung mit dieser Klasse stehenden Komponenten eines Plug-ins implementiert. Die Methoden *isValid(..)* und *getJsValidators()* werden als Teil der Domainlogik von den Plug-ins implementiert (siehe dazu Abbildung 6.1). Diese Methoden sollen die Benutzereingaben für Plug-in-Konfigurationen überprüfen und im Fehlerfall eine entsprechende Meldung erstellen. Die Methode *getJsValidators()* soll Validierer zur clientseitigen Überprüfung der Benutzereingaben liefern und die Methoden *isValid(..)* sollen für die serverseitige Überprüfung der Benutzereingaben zuständig sein.

Protokollieren (Plug-in *Logging*)

Das Protokollieren der Bearbeitung von HTTP-Anfragen (siehe Anwendungsfall 2.9) wird vom Plug-in *Logging* übernommen. Über das Attribut *level* in Abbildung 6.5 wird dem Logger die Stufe der Informationsdetails vermittelt. Der Logger soll unterschiedliche Ausgabequellen wie die Konsole und eine Textdatei für das persistente Abspeichern unterstützen. Die beschriebenen Eigenschaften wie die Stufe der Informationsdetails und die Ausgabequelle werden in der Methode *initialize()* dem Logger mitgeteilt. Für das Berechnen der Bearbeitungszeit einer HTTP-Anfrage wird das Hilfsattribut *processingStart* verwendet, welches sich die Zeit zu Beginn der Bearbeitung merken soll. Die Berechnung der Bearbeitungszeit umfasst die Zeiten der Bearbeitung einzelner Plug-ins. Das Plug-in Logging startet von allen Plug-ins als Erstes und endet als Letztes. Somit hat es die Möglichkeit die Gesamtzeit der Abarbeitung aller Plug-ins zu erfassen.

Ressourcen (Plug-in *Resources*)

Die Aufgabe dieses Plug-ins ist die Verwaltung der Ressourcen in den vorgesehenen Datenquellen (siehe Anwendungsfall 2.3). Die bestehenden Ressourcen sollen mit diesem Plug-in

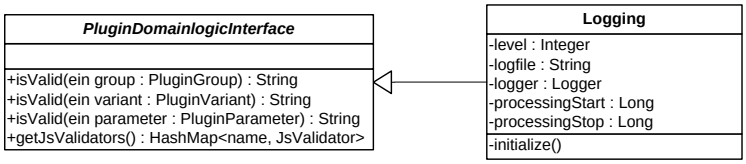


Abbildung 6.5: Protokollierung von HTTP-Anfragen

gemäß Anforderungen aus Abschnitt 4.1 ausgelesen, manipuliert, gelöscht genauso wie neue Datensätze angelegt werden können. Einige Plug-ins benötigen die ausgelesene Ressource zur Bearbeitung deren Domainlogik. Die Attribute der Ressourcen werden zum Beispiel für die Erstellung der Repräsentationen gebraucht, die im Plug-in *Representation* stattfindet. Deswegen werden die Attribute der Ressourcen in einer Instanz der Klasse *Resource* abgelegt. Die Benutzung der Klasse *Resource* soll bei jeder Implementierung zum Verwalten von Ressourcen in beliebiger Datenquelle stattfinden (siehe dazu auch Abbildung 6.5.3). Die aus den Datenquellen, zum Beispiel Datenbank, extrahierte Ressource zum Bearbeiten der Anfrage soll bis zur nächsten Anfrage in der Instanz der Klasse *Resources* zwischengelagert werde. Wenn andere Plug-ins zur Abarbeitung deren Domainlogik die mit diesem Plug-in ausgelesene Ressource brauchen, dann sollen diese Plug-ins auf die Instanz der Klasse *Resources* zugreifen und die Ressource mit der Methode *getResource()* auslesen können. Eine Ressource besteht aus Attributen, die einen Namen *name*, eventuell einen Wert *value* und möglicherweise einen Verweis *link* auf andere Ressourcen beinhalten. Zusätzlich wird auch der Datentyp vom Wert *value* im Attribut festgehalten. Die Bekanntgabe des Typs eines Attributs ist vor allem für die Bearbeitung der Ressourcenattribute von der Client-Anwendung von Vorteil.

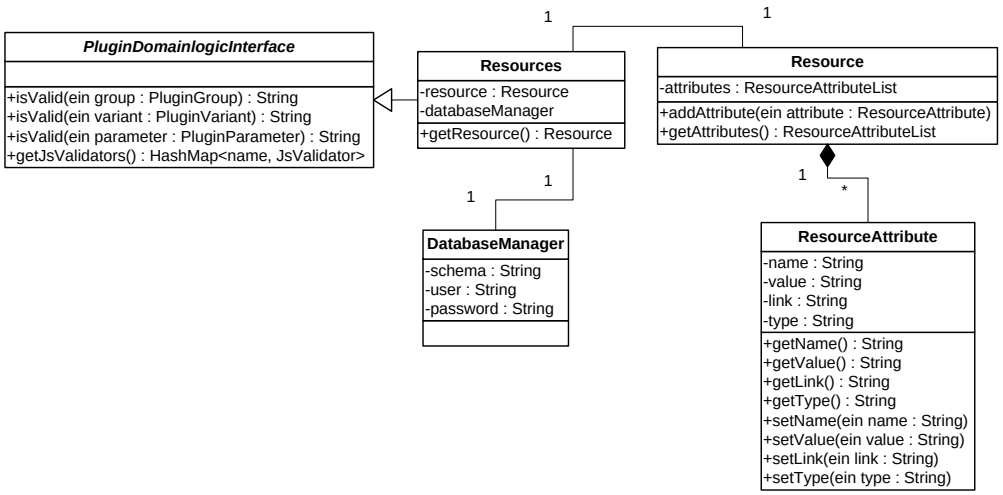


Abbildung 6.6: Ressourcenzugriff

Repräsentationen (Plug-in *Representation*)

Das Plug-in *Representation* beschäftigt sich mit der Generierung der Repräsentationen von Ressourcen genauso wie Extraktion der Ressourcen-Attribute aus den empfangenen HTTP-PUT- und HTTP-POST-Anfragen (siehe Anwendungsfall 2.4). Dieses Plug-in soll laut gestellten Anforderungen in Abschnitt 4.1 die Generierung der Repräsentation von Ressourcen in HTML-, XML- und JSON-Format unterstützen und dementsprechend auch Ressourcen-Attribute den empfangenen Repräsentationen der Ressourcen entnehmen können. Für die Generierung der Ressourcen-Repräsentationen sind die Klassen zuständig, welche die Schnittstelle *ResponseBuilderInterface* implementieren. Die Extraktion der Ressourcen-Attribute bei empfangenen Repräsentationen übernehmen die Schnittstelle *RequestReaderInterface* implementierenden Klassen. Die Instanz der Klasse *Resource*, welche die Attribute der angefragten Ressource beinhaltet, kann dem Plug-in *Resources* aus dem *PluginContainer* entnommen werden. Einige Plug-ins brauchen die von diesem Plug-in generierte Repräsentation der angefragten Ressource zur Bearbeitung deren Domainlogik. Der Inhalt der HTTP-Antwort beziehungsweise die Repräsentation der angefragten Ressource wird zum Beispiel für die Berechnung des *ETag*-Headers vom Plug-in *Caching* gebraucht. Deswegen wird die generierte Repräsentation von der Methode *generate()* jeder Klasse, welche die Schnittstelle *ResponseBuilderInterface* implementiert, an die Instanz der Klasse *Representation* übermittelt. Die Repräsentation der angefragten Ressource soll während der Bearbeitung einer HTTP-Anfrage im Attribut *responseContent* zwischengespeichert bleiben, um anderen Plug-ins zur Verfügung zu stehen (siehe dazu Abbildung 6.5.3).

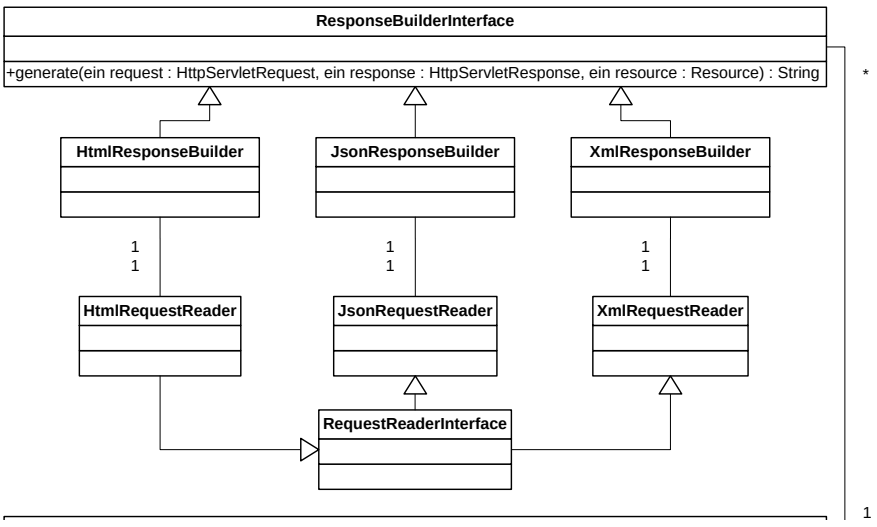


Abbildung 6.7: Generierung von Repräsentationen

Autorisierung (Plug-in *Authorization*)

Für die Überprüfung der Autorisierung der HTTP-Anfragen (siehe Anwendungsfall 2.5) ist das Plug-in *Authorization* zuständig. Gemäß der Anforderungen aus Abschnitt 4.1 werden

die Verfahren *HTTP Basic* und *HTTP Digest* realisiert. Die Klassen *HTTPBasic* und *HTTPDigest* implementieren die Schnittstelle *AuthorizationInterface*. Diese Schnittstelle schreibt den Klassen *HTTPBasic* und *HTTPDigest* vor, die Methode *isAuthorized()* zu implementieren. Diese Methode soll anhand der mit der HTTP-Anfrage empfangenen HTTP-Header und der in der Plug-in-Konfiguration vorliegenden Parameter entscheiden, ob die empfangenen HTTP-Anfragen bearbeitet werden dürfen.

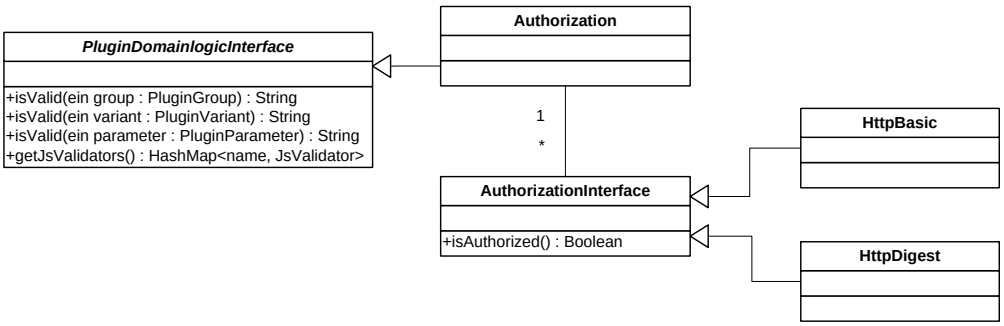


Abbildung 6.8: Autorisierung

Cookies (Plug-in *Cookies*)

Mit diesem Plug-in können neue Cookies über die Methode *addCookie()* angelegt und der HTTP-Antwort angehängt werden (siehe Anwendungsfall 2.8). Weiterhin besteht die Möglichkeit die mit der HTTP-Anfrage empfangenen Cookies zu löschen oder mit der HTTP-Antwort zurück an den Client zu schicken. Mit der Methode *deleteAll()* sollen alle empfangenen Cookies gelöscht werden. Die Methode *delete()* löscht nur die Cookies, deren Namen in der übergebenen Liste vorkommen. Die Methode *deleteAllExcept()* sorgt dafür, dass alle Cookies, bis auf bestimmte Ausnahmen, gelöscht werden. Die Liste der Ausnahmen wird in einer Liste als Parameter dieser Methode übergeben. Die Attribute der anzulegenden Cookies genauso wie die Liste der Cookies zum Löschen und zum Zurückschicken sollen der Konfiguration des Plug-ins *Cookies* entnommen werden.

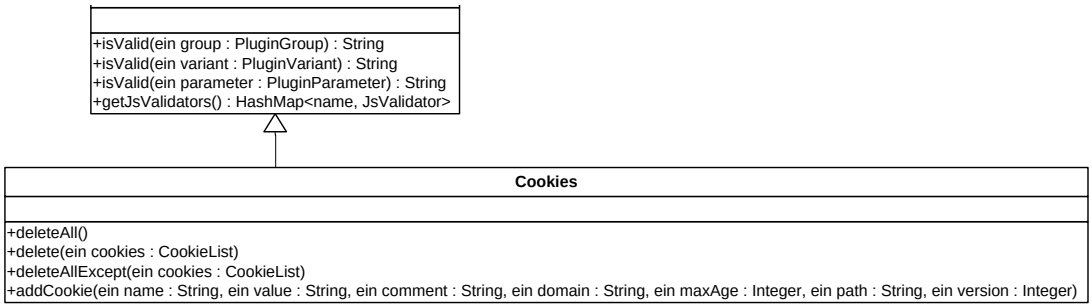


Abbildung 6.9: Cookies

Caching (Plug-in *Caching*)

Zur Steuerung des Verhaltens der Client-Anwendung bezüglich Cache (siehe Anwendungsfall 2.7) wird das Plug-in *Caching* eingesetzt. Dieses Plug-in verwaltet HTTP-Header, welche die Benutzung von Cache in der Client-Anwendung bewirken können. Anhand der vom Testbed geschickten HTTP-Header entscheidet die Client-Anwendung, ob sie den Cache benutzt oder nicht. Die Klasse *Caching* stellt Methoden zur Verfügung, welche die HTTP-Header mit gewünschten Werten versorgen. Die Bezeichnungen der Methoden wurden entsprechend der Bezeichnung von den HTTP-Headern in der *HTTP-1.1-Spezifikation* (Fie09, Kapitel 13) gewählt. Die gewünschten HTTP-Header mit deren Werten sind der Konfiguration des Plug-ins *Caching* zu entnehmen.

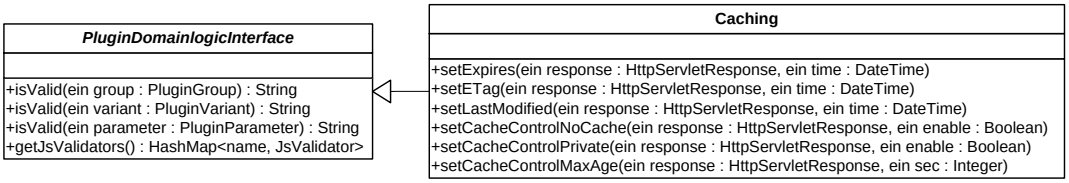


Abbildung 6.10: Steuerung von dem Caching-Verhalten der Client-Anwendung

HTTP-Statusmeldungen (Plug-in *Responsecodes*)

Zum Simulieren von HTTP-Statusmeldungen (siehe Anwendungsfall 2.6) steht das Plug-in *Responsecodes* zur Verfügung. Es sollen gemäß gestellter Anforderungen in Abschnitt 4.1 alle HTTP-Statusmeldungen simulierbar sein, die in der Publikation *HTTP-1.1-Spezifikation* (Fie09, Kapitel 10) definiert sind. Die Methoden zum Simulieren der HTTP-Statusmeldungen sind entsprechend der Benennung der *HTTP-1.1-Spezifikation* bezeichnet und können somit leicht der Beschreibung in der Spezifikation zugeordnet werden. Die zu simulierende HTTP-Statusmeldung und eventuelle Parameter sollen den definierten Methoden aus der aktuellen Konfiguration des Plug-ins *Responsecodes* übergeben werden.

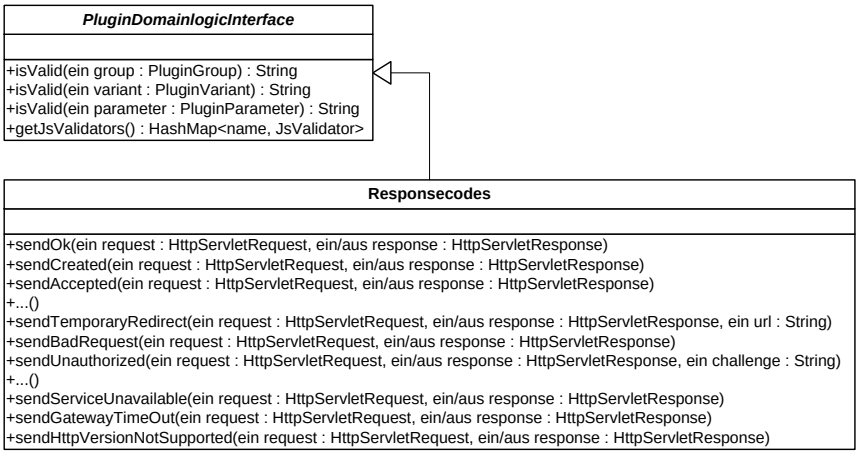


Abbildung 6.11: Simulation von HTTP-Statusmeldungen

7 Implementierung

In diesem Abschnitt wird auf die Implementierung der Komponenten des Testbeds eingegangen. Es wird die Implementierung der Konfigurationen, der Beobachter, der Domainlogik und weiterer Komponenten erklärt.

Die Schnittstellen *ConfigurationAPI* und *TestServiceAPI* in Abbildung 6.1 wurden mit Jersey (Ora) in der Version 1.14, Referenzimplementierung der Spezifikation JAX-RS 1.1 (HS07), implementiert. Über die Schnittstelle *TestServceAPI* werden Testanfragen zum Testen einer Client-Anwendung getätigt und über die Schnittstelle *ConfigurationAPI* findet die Verwaltung der Plug-in Konfigurationen statt.

7.1 Konfigurationsverwaltung

Für die Verwaltung von Plug-in-Konfigurationen wurde in Abschnitt 5.2 beschriebenes MVC-Architekturmuster angewandt. Nachfolgend wird die Implementierung der drei Komponenten des MVC-Musters genauer betrachtet.

7.1.1 Modell vom MVC-Architekturmuster

Die Komponente Modell des MVC-Musters entspricht dem *Modell für Plug-in-Konfigurationen* aus Abschnitt 5.3. Das Modell ist dafür zuständig, die Werte der Plug-in-Konfigurationen zu verwalten. Mit den Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und *PluginParameter* aus Abbildung 6.1 wird das in Abbildung 5.4 dargestellte Modell implementiert. Diese Klassen sind auch für die Verwaltung von Plug-in-Konfigurationsdateien implementiert, die in XML-Formaten vorliegen. Mithilfe dieser XML-Dateien werden die Plug-in-Konfigurationen persistent abgespeichert. Das in Abschnitt 5.3 beschriebene Modell wird mit zwei Dateien je Plug-in realisiert. In einer Datei werden konfigurierbare Parameter verwaltet, die zum Parametrisieren der Domainlogik von Plug-ins verwendet werden. Die andere Datei beinhaltet beschreibende Daten respektive Meta-Daten der konfigurierbaren Parametern.

Die Dateien, welche zum persistenten Abspeichern der Plug-in-Konfiguration dienen, sollen von jedem Plug-in bereitgestellt werden. Die Werte dieser Dateien werden nach dem Auslesen in den Instanzen der Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und *PluginParameter* (siehe Abbildung 6.1) im Arbeitsspeicher gehalten, sodass das Auslesen der Konfigurationsdateien bei erneuten HTTP-Anfragen keine weiteren Latenzen durch das Auslesen verursacht. Die Klasse *PluginConfig* steht gemäß *Modell für Plug-in-Konfigurationen* aus Abschnitt 5.3 für die Komponente *Plug-in Configuration* in Abbildung 5.4. Die Bezeichnungen der Klassen für

andere Komponenten wurden gemäß der Bezeichnungen der Komponenten in Abbildung 5.4 gewählt.

Zum Festhalten der Werte von Plug-in-Konfigurationen wurde bei der Implementierung der Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und *PluginParameter* ein Datentyp eingesetzt, welcher die Positionierung der eingefügten Elemente beibehält. Für die graphische Darstellung von Plug-in-Konfigurationen ist es von Vorteil die Konfigurationsdateien der Plug-ins passend zu strukturieren. Die Repräsentationen von Plug-in-Konfigurationen können benutzerfreundlicher gestaltet werden, indem die Positionierung der Elemente in den Konfigurationsdateien auf die Repräsentation dieser Konfigurationen übernommen wird.

Der Zugriff auf die Konfigurationsdateien tritt wieder auf, wenn einige Werte in den Instanzen der Klassen *PluginConfig*, *PluginGroup*, *PluginVariant* und beziehungsweise oder *PluginParameter* über die Schnittstelle *ConfigurationAPI* verändert wurden und in den Konfigurationsdateien persistent abgespeichert werden sollen.

7.1.2 Views vom MVC-Architekturmuster

Die *Views* nach dem MVC-Muster stellen die Repräsentationen der Plug-in-Konfigurationen beziehungsweise die oben beschriebene Modell-Instanzen dar. Für die Gestaltung der Repräsentationen wurde das Konzept *Hypermedia as the Engine of Application State* aus Abschnitt 2.1.2 angewandt. Im Wurzelelement befindet sich die Liste mit den Namen der verfügbaren Plug-ins, die aus den dazugehörigen Konfigurationsdateien ausgelesen werden. Jedem Namen wird ein Link zugeordnet, welcher auf die entsprechende Plug-in-Konfiguration verweist. Folgt man dem Link mit der HTTP-GET-Methode, so bekommt man die Repräsentation der jeweiligen Plug-in-Konfiguration.

Es wurden zwei *Views* implementiert: *HtmlConfigResponseBuilder* und *XmlConfigResponseBuilder*. Mit der Klasse *XmlConfigResponseBuilder* werden die Repräsentationen der Plug-in-Konfigurationen in XML-Format generiert. Die Klasse *HtmlConfigResponseBuilder* sorgt für die Generierung der Plug-in-Konfigurationen in HTML-Format. Für HTML-Repräsentationen wurde das HTML-Formular verwendet, um die Plug-in-Konfiguration mit einem Web-Browser ohne weitere Tools manuell bearbeiten zu können.

7.1.3 Controller vom MVC-Architekturmuster

Die modifizierten Repräsentationen werden an einen *Controller* gemäß dem MVC-Muster übergeben. Es sind zwei *Controller* implementiert: *XmlConfigController* und *HtmlConfigController*, siehe Abbildung 6.1. Abhängig vom Format, in dem die über die Schnittstelle *ConfigurationAPI* empfangene Plug-in-Repräsentation vorliegt, wird einer der beiden *Controller* verwendet. Die konfigurierbaren Werte aus einer Repräsentation können vom Testbed-Benutzer verändert und dann mit der HTTP-Methode POST beziehungsweise PUT an das REST Testbed abgeschickt werden. Für die XML-Repräsentationen der Plug-in-Konfigurationen wurde dafür gemäß der *HTTP-1.1-Spezifikation* (Fie09) die vorgesehene HTTP-PUT-Methode verwendet. Für die HTML-Repräsentation musste auf die HTTP-POST-Methode ausgewichen werden,

denn für die Übertragung der modifizierten Werte der HTML-Repräsentation wird das HTML-Formular benutzt, das nur die HTTP-Methoden GET und POST unterstützt. Ein *Controller* übernimmt die mit der POST- beziehungsweise PUT-Methode übertragenen neuen Werte für die Plug-in-Konfiguration im fehlerfreien Fall und liefert die aktualisierte Repräsentation der Plug-in-Konfiguration an die Client-Anwendung. Beim Auftreten eines Fehlers liefert der Controller die Repräsentation der Plug-in-Konfiguration mit den Fehlerbeschreibungen für die nicht übernommenen Werten an die Client-Anwendung. Die Fehlerbeschreibung wird im zusätzlich angehängten *error*-Attribut des Parameters übertragen, siehe auch Abschnitt 5.3.

7.2 Validierung der Benutzereingaben

Der Testbed-Benutzer führt Anpassungen an den Plug-in-Konfigurationen über die Schnittstelle *ConfigurationAPI* durch. Dabei können die Benutzereingaben fehlerhaft sein. Die Validierung der Benutzereingaben kann von der Client-Anwendung und vom Testbed gewährleistet werden. In Domainlogik-Klassen von Plug-ins, welche die Schnittstelle *PluginDomainlogicInterface* implementieren, werden die Methoden zum Validieren von Benutzereingaben definiert, siehe dazu Abbildung 6.1. Bei der Methode *getJsValidators()* handelt es sich um Validierer, welche für die Validierung der Benutzereingaben auf der Client-Anwendung gedacht sind. Die Methoden *isValid()* überprüfen die Benutzereingaben auf dem Testbed.

7.2.1 Validierung auf der Client-Anwendung

Die *type*-Attribute der Parameter in den Konfigurationsdateien der Plug-ins, siehe dazu Abschnitt 5.3, werden dazu verwendet, um die Benutzereingaben von der Client-Anwendung validieren zu können. Für diesen Zweck wird JavaScript eingesetzt, denn JavaScript kann von den Webbrowsern und genauso von den Eigenentwicklungen unter Verwendung von zusätzlichen JavaScript-Interpreter ausgeführt werden.

Die Instanzen der Klasse *JsValidator* beinhalten zwei String-Attribute. Bei einem String handelt es sich um eine JavaScript-Methode und bei dem anderen um einen Kommentar. Die JavaScript-Methode soll die Benutzereingaben validieren, indem einige Bedingungen überprüft werden. Wenn die Bedingungen nicht erfüllt sind, dann soll der definierte Kommentar einen Hinweis zur Unterstützung der korrekten Benutzereingabe liefern.

Für HTML-Repräsentationen von Plug-in-Konfigurationen wurde die JavaScript-Bibliothek *JQuery* (jQua) mit dem *JQuery Validation Plugin* (jQub) eingesetzt. Die definierten JavaScript-Methoden mit den jeweiligen Kommentaren werden in die HTML-Repräsentationen integriert. Wenn der Testbed-Benutzer Eingaben macht, werden diese Werte vom *JQuery Validation Plugin* überprüft. Erst nach der erfolgreichen Validierung können die aktualisierten Werte an das Testbed übermittelt werden. Wenn die Validierung nicht erfolgreich verläuft, dann werden die definierten Kommentare bei den Eingabefeldern mit fehlerhaften Eingaben eingeblendet. Diese Kommentare sollen den Testbed-Benutzer dabei unterstützen, die fehlerhaften Eingaben zu korrigieren.

Im Fall einer XML-Repräsentation werden die JavaScript-Validierer mit dem jeweiligen Kommentar mit der Repräsentation der Plug-in-Konfiguration mitgeliefert und stehen somit für eigene Umsetzung der clientseitigen Validierung zur Verfügung.

7.2.2 Validierung auf dem Testbed

Die empfangenen Benutzereingaben werden auf dem Testbed validiert, bevor sie in die Plug-in-Konfigurationen übernommen werden. Die Validierung auf dem Testbed findet mit Hilfe von *isValid(..)*-Methoden der Domainlogik von Plug-ins statt. Die drei *isValid(..)*-Methoden werden zum Validieren von den Instanzen der Klassen *PluginGroup*, *PluginVariant* und *PluginParameter* verwendet. Wenn die Validierung erfolgreich verläuft, dann liefern die *isValid()*-Methoden *null*. In einem Fehlerfall liefern die Methoden eine Textnachricht mit dem Hinweis auf die fehlerhafte Benutzereingabe. Diese Textnachricht wird im *error*-Attribut des fehlerhaften Parameters (*PluginParameter*) eingetragen, daraufhin in die betroffene Repräsentation der Plug-in-Konfiguration eingearbeitet und an die Client-Anwendung übermittelt.

7.3 Beobachter von Plug-ins

Die Beobachter im REST Testbed werden nach der aspektorientierten Programmierung mit Hilfe der Java-Erweiterung AspectJ Development Tools (AJDT) (Ecl) implementiert. Bei implementierten Beobachter-Komponenten handelt es sich folglich um Aspekte, siehe dazu Abschnitt 2.3. In einem Aspekt kann ein *Pointcut* (Schnittpunkt) definiert werden. Als *Pointcut* wird eine oder mehrere Methoden der Schnittstellen *ConfigurationAPI* und *TestserviceAPI* gewählt. Als *Advice* (Empfehlung) reichen bei den aktuellen Aspekten die Konstrukte *before* und *after* aus. Die Beschreibungen der Konstrukte sind in Abschnitt 2.3 nachzulesen. Abbildung 7.1 zeigt ein beispielhaftes Aspekt mit den Konstrukten, die bei der Implementierung der Beobachter verwendet wurden.

Es werden *Pointcuts init()*, *ConfigurationAPIMethod()* und *TestServiceAPIGet()* definiert. Die *Advice*-Konstrukte *before* und *after* von den *Pointcuts* legen den Ausführungspunkt für die in den *Advice*-Konstrukten definierten Aktionen fest. Wenn die Methode *get()* der Schnittstelle *TestServiceAPI* aufgerufen wird (*TestServiceAPI.get()*), so wird vor der Bearbeitung der *get()*-Methode, die an der Stelle *to do before 'get'* definierte Aktion ausgeführt. Danach folgt die Bearbeitung der *get()*-Methode. Die an der Stelle *to do after 'get'* definierte Aktion wird nach der Bearbeitung der *get()*-Methode ausgeführt. Das *Pointcut* für die Schnittstelle *ConfigurationAPI* reagiert auf alle Methoden (*ConfigurationAPI.*()*) und vor der Ausführung der aufgerufenen Methode wird die an der Stelle *to do before* definierte Aktion ausgeführt.

Der Konstruktor dieses Aspekts wird bei der Instanziierung ausgeführt. Die Instanziierung der Aspekte erfolgt jedoch erst beim Feststellen der Ausführung der in den *Pointcuts* angegebenen *Joinpoints*. Für die Definition von einem *Joinpoint* lese man Abschnitt 2.3. Manche Aspekte benötigen aber keine *Joinpoints* zu der Schnittstelle *ConfigurationAPI*. Zur Zeit eines Aufrufs einer Plug-in-Konfiguration von solchen Aspekten ist der Konstruktor noch nicht ausgeführt worden und die gewünschte Plug-in-Konfiguration kann sich noch nicht im *PluginContainer*

```
public aspect AspectB {

    declare precedence : LoggingA, LoggingB, LoggingC;

    public AspectB(){
        // Instantiate domainlogic and configuration of associated plug-in
        // and put this instances into pluginContainer.
    }

    pointcut init() : execution(ConfigurationAPI.new(..)) ||
                      execution(TestServiceAPI.new(..));

    before() : init() {}

    pointcut ConfigurationAPIMethod():
        execution(* ConfigurationAPI.*(..));

    before(): ConfigurationAPIMethod() {
        // to do before
    }

    pointcut ConfigurationAPIGet(HttpServletRequest request,
                                 HttpServletResponse response):
        execution(* TestServiceAPI.get(..)) && args(request, response);

    before(HttpServletRequest request, HttpServletResponse response):
        TestServiceAPIGet(request, response) {
        // to do before 'get'
    }

    after(HttpServletRequest request, HttpServletResponse response):
        TestServiceAPIGet(request, response) {
        // to do after 'get'
    }

}
```

Abbildung 7.1: Beobachter als Aspekt

befinden. Außer Aspekten werden die Plug-in-Konfigurationen auch von anderen Komponenten wie *Views* und *Controller* genutzt. Um diesen Komponenten die Plug-in-Konfigurationen mithilfe des *PluginContainers* bereitzustellen, muss jeder Aspekt das *Pointcut init()* beinhalten. Das *Pointcut init()* sorgt alleine nur dafür, dass bei erster HTTP-Anfrage über *ConfigfurationAPI* oder *TestServiceAPI* alle Aspekte initialisiert werden. Die Konstruktoren von Aspekten erstellen die Instanzen der Domainlogik und der Konfigurationen von Plug-ins. Diese Instanzen werden anschließend im *PluginContainer* abgelegt und stehen somit auch anderen Komponenten des Testbeds zur Verfügung.

Mit dem Konstruktor *declare precenence* wird die Reihenfolge der Ausführung von *Advices* von Aspekten definiert, die gleiche *Joinpoints* haben. Im gezeigten Aspekt *AspectB* wird definiert, dass der Aspekt *AspectA* Vorrang vor dem Aspekt *AspectB* hat. Der Aspekt *AspectB* hat wiederum Vorrang vor dem Aspekt *AspectC*. Die Reihenfolge der Ausführung kann bei den Aspekten auch mehrfach durch verschiedene Aspekte angegeben werden, darf aber nicht widersprüchlich sein.

Bei der Annahme, dass in den Aspekten *AspectA* und *AspectC* die Methode *get()* der Schnittstelle *TestServiceAPI* auch einem *Pointcut* zugeordnet wäre und auch die *Before-* und *After-Advices* für diesen *Pointcut* definiert wären, wird die HTTP-Anfrage in folgender Reihenfolge abgearbeitet. Zuerst wird die definierte Aktion vom *Before-Advice* des *TestServiceAPIGet-Pointcuts* vom Aspekt *AspectA*, dann vom Aspekt *AspectB* und danach vom Aspekt *AspectC* abgearbeitet. Danach folgt die Bearbeitung der angefragten Methode *get()*. Nach der Ausführung der *get()*-Methode werden die *after-Advices* der betrachteten Aspekte bearbeitet. Für die Durchführungen der Aktionen der *after-Advices* wird die umgekehrte Reihenfolge der Aspekte zur Ausführung der *before-Advices* angewandt. Es wird die Aktion vom *after-Advice* des *TestServiceAPIGet-Pointcuts* vom Aspekt *AspectC*, dann vom Aspekt *AspectB* und danach vom Aspekt *AspectA* ausgeführt.

7.4 Domainlogik von Plug-ins

Die entwickelten Plug-ins sind nachfolgend in einer Liste mit absteigendem Vorrang bei der Bearbeitung angegeben. Bei den Beschreibungen wird mit dem Bearbeiten der Plug-ins das Anwenden der Domainlogik der Plug-ins angedeutet.

- 1. Logging: Das Plug-in mit dem höchsten Vorrang. Alle anderen Plug-ins können den Logger des Plug-ins *Logging* zum Protokollieren der Bearbeitung verwenden. Bei dem *Before-Advice* wird die empfangene HTTP-Anfrage und mit dem *After-Advice* die abgeschickte HTTP-Antwort protokolliert.
- 2. Responsecodes: Mit diesem Plug-in werden HTTP-Statusmeldungen simuliert. Dieses Plug-in hat zweithöchsten Vorrang und wird mit dem *Before-Advice* abgearbeitet, denn bis auf das Protokollieren der Bearbeitung vor diesem Plug-in sind keine weiteren Bearbeitungen nötig. Wenn dieses Plug-in aktiviert ist, dann folgt keine Abarbeitung der weiteren Plug-ins (siehe dazu Abschnitt 7.4.2). Die nachfolgenden Plug-ins verwenden wie dieses Plug-in nur die *Before-Advices*.

- 3. Authorization: Die Abarbeitung dieses Plug-ins findet nur statt, wenn das Plug-in *Responsecodes* deaktiviert ist.
- 4. Resources: Vor der Bearbeitung dieses Plug-ins muss die Protokollierung und die Überprüfung der Autorisierung der HTTP-Anfrage stattgefunden haben.
- 5. Representation: Bei diesem Plug-in werden die Repräsentationen von Ressourcen generiert. Die Ressource muss vor der Bearbeitung dieses Plug-ins von dem Plug-in *Resources* aus einer Ressourcenquelle ausgelesen sein und zur Verfügung stehen. Somit hat das Plug-in *Resources* Vorrang vor dem Plug-in *Representation*.
- 6. Caching: Dieses Plug-in braucht in manchen Fällen, wie für die Berechnung des *Entity-Tag*, die Repräsentation der angefragten Ressource. Folglich hat das Plug-in *Representation* Vorrang vor dem Plug-in *Caching*.
- 7. Cookies: Vor der Ausführung dieses Plug-ins muss das Logging und die Autorisierung der HTTP-Anfrage durchgeführt worden sein. Somit hätte man dieses Plug-in auch bis zur Stelle 4 vorschieben können.

7.4.1 Protokollieren (Plug-in *Logging*)

Für das Protokollieren der Bearbeitung der HTTP-Anfragen wurde das Framework *log4j* (Apa) eingesetzt. Die zu protokollierenden Meldungen können verschiedenen Kategorien zugeordnet werde: *DEBUG*, *INFO*, *WARN*, *ERROR* und *FATAL*. Das folgende Beispiel zeigt die Verwendung des Frameworks *log4j*:

```
import org.apache.log4j.Logger;
Logger logger = Logger.getRootLogger();
logger.info("Root logger created.");
```

Bei der Ausgabe der protokollierten Meldungen kann der Informationsumfang gefiltert werden:

```
logger.setLevel(Level.INFO);
```

Es sind sieben Stufen des Informationsumfangs definiert, die weiter unten aufgelistet sind. Die Stufen sind nach unten in aufsteigender Reihenfolge der Wichtigkeit angegeben. Bei der Wahl einer Wichtigkeitsstufe werden alle Meldungen der gewählten Stufe genauso wie aller Stufen mit der höherer Wichtigkeit, in den aktivierten Ausgabequellen ausgegeben (Gup05). Die Stufe der zu protokollierenden Meldungen wird über die Schnittstelle *ConfigurationAPI* festgelegt.

Kategorien der Meldungen in *log4j*:

- *ALL*: Alle Meldungen werden ausgegeben. Keine Filterung.
- *TRACE*: Kommentare.
- *DEBUG*: Informationen für die Fehlersuche.
- *INFO*: Informationen zum regulären Programmablauf.

- *WARN*: Warnhinweise.
- *ERROR*: Fehler, die abgefangen werden.
- *FATAL*: Kritische Fehler, die zum Programmabbruch führen.
- *OFF*: Keine Ausgabe der Meldungen.

Beim Framework *log4j* können mehrere Ausgabequellen mittels *Appender* aktiviert werden, in welchen die protokollierten Meldungen ausgegeben werden (Gup05). Beim REST Testbed wurden *ConsoleAppender* für die Ausgabe in der Console und *FileAppender* für das persistente Abspeichern der Meldungen in einer Datei gewählt. Im folgenden Beispiel wird beim Framework *log4j* eine Konsole als Ausgabequelle aktiviert.

```
import org.apache.log4j.ConsoleAppender;
ConsoleAppender consoleAppender = new ConsoleAppender();
logger.addAppender(consoleAppender);
```

Die Appender *Console* und *FileAppender* können über die Schnittstelle *ConfigurationAPI* ein- und ausgeschaltet werden. Auch die Bezeichnung von der Datei zum persistenten Abspeichern der protokollierten Meldungen kann über die Schnittstelle *ConfigurationAPI* angepasst werden. Wenn die Bezeichnung der Datei dem Testbed-Benutzer bekannt ist, dann kann er auf dessen Inhalt mit einer HTTP-GET-Anfragen zugreifen.

7.4.2 HTTP-Statusmeldungen (Plug-in *Responsecodes*)

Zum Simulieren von HTTP-Statusmeldungen wurde die Klasse *WebApplicationException* aus dem Paket *javax.ws.rs.WebApplicationException* eingesetzt. Die HTTP-Statusmeldungen werden als Ausnahmefall generiert. Folgendes Beispiel zeigt eine Zeile aus dem Quellcode zum Simulieren der HTTP-Statusmeldung einer temporär verschobener Ressource:

```
import javax.ws.rs.WebApplicationException.WebApplicationException;
throw new WebApplicationException(Response.status(307).build());
```

Bei einigen HTTP-Statusmeldungen sollen spezifischen Header gesetzt werden. Bei der oben angegebenen Statusmeldung verlangt die *HTTP-1.1-Spezifikation* das Setzen des HTTP-Header *Location*, das auf den neuen Ort der angefragten Ressource verweist. Dies wird mit folgendem Befehl realisiert:

```
response.addHeader("Location", "http://...");
```

Bei der Instanz *response* handelt es sich um die vom Web Service bereitgestellte Instanz der Klasse *HttpServletResponse*. Die benötigten HTTP-Header müssen auf diese Weise vor der Generierung der HTTP-Statusmeldung gesetzt werden.

Welche HTTP-Statusmeldung mit welchen HTTP-Header generiert werden soll, wird aus der Konfiguration des Plug-ins *Responsecodes* ausgelesen. Der Testbed-Benutzer kann diese Konfiguration über die Schnittstelle *ConfigurationAPI* nach Bedarf anpassen.

7.4.3 Autorisierung (Plug-in *Authorization*)

Für die Überprüfung der Autorisierung von HTTP-Anfragen existieren viele standardisierte Verfahren. Bei dem REST Testbed wurden die Verfahren *HTTP Basic* und *HTTP Digest* implementiert. Der Austausch der Daten zur Überprüfung der Autorisierung erfolgt zwischen dem Client und Server mittels HTTP-Header. Wenn das Plug-in *Authorization* aktiviert ist, dann wird die HTTP-Anfrage auf den HTTP-Header untersucht, welches die Daten zur Überprüfung der Autorisierung beinhaltet. Bei den beiden oben angegebenen Verfahren wird das HTTP-Header *Authorization* verwendet. Im folgenden Beispiel wird das HTTP-Header *Authorization* der HTTP-Anfrage entnommen:

```
String auth = request.getHeader("Authorization");
```

Bei der Instanz *response* handelt es sich um die vom Web Service bereitgestellte Instanz der Klasse *HttpServletRequest*. Alle Daten zum Anwenden des aktivierten Authentifizierungsverfahrens können dem Attribut *auth* extrahiert werden. Wenn der *Authorization*-Header bei der HTTP-Anfrage nicht gesetzt ist oder das Authentifizierungsverfahren anhand der mitgelieferten Daten auf eine unautorisierte HTTP-Anfrage schließt, dann wird eine HTTP-Statusmeldung mit den für den Client notwendigen Daten abgeschickt. In diesem Fall handelt es sich um HTTP-Header *WWW-Authenticate*. Das Beispiel unten zeigt die beschriebene HTTP-Antwort für das Verfahren *HTTP Basic*:

```
response.addHeader("WWW-Authenticate", "Basic realm=\"testbed\"");
ResponseBuilder builder = Response.status(Status.UNAUTHORIZED);
throw new WebApplicationException(builder.build());
```

Es wird die in Abschnitt 7.4.2 beschriebene Klasse *WebApplicationException* eingesetzt. Dem HTTP-Header *WWW-Authenticate* müssen alle für das Authentifizierungsverfahren benötigten Daten übergeben werden, damit die Client-Anwendung zum angefragten Authentifizierungsverfahren eine HTTP-Anfrage mit dem korrekt aufgebauten HTTP-Header *Authorization* erstellen kann.

7.4.4 Ressourcen (Plug-in *Resources*)

Mit dem Plug-in *Resources* werden die Ressourcen in einem Datenbestand verwaltet. Als Datenquelle wurde im Testbed die relationale Datenbank verwendet. Es sind einige Methoden zum Verwalten der Daten in einer relationalen Datenbank implementiert, die den Grundbedarf für die Interaktion zwischen dem Testbed und einer relationalen Datenbank abdecken.

Die Klassen zum Verwalten von Ressourcen in einem bestimmten Datenbestand implementieren die Schnittstelle *Resource*, siehe dazu Abbildung 7.2. Mit der Methode *get()* werden die Attribute einer Ressource aus einer Datenquelle ausgelesen und in der Liste abgespeichert. Die Methode *put()* sorgt dafür, dass die modifizierten Werte der Ressourcen-Attribute auch in die Datenquelle übernommen werden. Mit der Methode *post()* werden die mit der HTTP-Anfrage erhaltenen Ressourcen-Attribute dazu verwendet, um eine neue Ressource

in der aktivierten Datenquelle anzulegen. Die *delete*-Methode löscht eine Ressource aus der aktuellen Datenquelle.

```
public interface Resource {

    public String getId();

    public void head(HttpServletRequest request);

    public ResourceAttribute[] get(HttpServletRequest request);

    public void put(HttpServletRequest request);

    public void post(HttpServletRequest request);

    public void delete(HttpServletRequest request);

}
```

Abbildung 7.2: Schnittstelle für Ressourcen-Klassen

Die Attribute einer Ressource werden in einer Liste mit den Instanzen der Klasse *Attribute* hinterlegt. Abbildung 7.3 zeigt einen Ausschnitt der Klasse *ResourceAttribute*. Ein Ressourcen-Attribut hat einen Namen (*name*) und eventuell kann es zusätzlich einen Verweis (*link*) auf eine andere Ressource enthalten. Weiterhin kann ein Ressourcen-Attribut einen Wert (*value*) besitzen, der möglicherweise modifiziert werden darf. Die Attribut-Werte (*value*) sind von einem bestimmten Typ (*type*), bei der Bekanntheit dessen die Validierung der Attribut-Werte außerhalb der Datenquelle erleichtert wird.

```
public class ResourceAttribute {

    private String name;
    private String value;
    private String link;
    private String type;

    ...

}
```

Abbildung 7.3: Ressourcen-Attribut

Mittels Mapping-Operationen werden die durch die URIs angefragten Ressourcen bestimmt. Die Mapping-Tabelle kann vom Testbed-Benutzer verwaltet werden. Eine Ressource, die unter einer URI verfügbar ist, kann nach der Manipulation der Mapping-Tabelle unter einer anderen URI zur Verfügung stehen. Die Mapping-Tabelle ist in der Plug-in-Konfiguration realisiert. Dazu werden die Instanzen der Klasse *PluginParameter* erstellt (siehe Abbildung 6.1). Das Attribut *name* der Klasse *PluginParameter* bekommt eine Bezeichnung, mit der eine Ressource aus der aktivierten Datenquelle assoziiert wird. Bei der implementierten Datenbankverwaltung kann das der Name einer Tabelle sein, falls mit der Ressource eine Tabelle gemeint ist. Das Attribut *value* nimmt einen auf die URI schließenden Ausdruck an. Es könnte nach der implementierten Logik zum Beispiel der Tabellen-Name als Erweiterung der URI der Ressourcen-Schnittstelle sein. Wenn die URI der Ressourcen-Schnittstelle http://localhost:8080/testbed/resources und der Tabellen-Name *customers* ist, dann wird mit

der URI http://localhost:8080/testbed/resources/customers mit der HTTP-GET-Methode die Repräsentation der Datenbank-Tabelle namens *customers* angefragt.

Der Ansatz der dynamischen Zuordnung der URIs zu den Ressourcen mithilfe einer Mapping-Tabelle hat auch einen Nachteil. Es werden auch Anfragen von der Testservice-Schnittstelle abgefangen, für die keine Ressourcen existieren. Das Überprüfen der Existenz der Ressourcen fällt in diesem Fall nicht in den Bereich des Jersey Frameworks. Die ankommenden Anfragen beim Testserviceaufruf werden erst in der eigenen Implementierung auf das Vorhandensein der angefragten Ressourcen überprüft. Die fehlende Überprüfung der Existenz von Ressourcen im Jersey Framework führt dazu, dass Jersey die WADL-Spezifikation nicht erstellen kann.

Das URI-Schema erlaubt nicht alle Zeichen. Deswegen wird für die Erstellung der Ausschnitte der URIs, die in der Mapping-Tabelle festgelegt werden, die Klasse *java.net.URLEncoder* verwendet. Mit dieser Klasse können einige nicht URI-konforme Ausdrücke durch das Ersetzen der unzulässigen Zeichen in URI-konforme Ausdrücke überführt werden. Wenn es den Methoden dieser Klasse nicht gelingt, wird der Testbed-Benutzer darauf hingewiesen.

Von der Komponente zum Verwalten von Ressourcen in einer relationalen Datenbank werden auch Funktionen zum Verwalten der Mapping-Tabelle geliefert. Durch den Wechsel der Datenquelle soll auch die Mapping-Tabelle neu erstellt werden. Wird eine Abweichung der in der aktuellen Mapping-Tabelle festgehaltenen Ressourcen zu den Ressourcen von der aktuellen Datenquelle festgestellt, so wird die Mapping-Tabelle aktualisiert. Dies erfolgt mithilfe der zur Verwaltung der Mapping-Tabelle bereitgestellten Funktionen, welche die Mapping-Tabelle mit den definierten Standardwerten füllen. Die Ressourcen-Bezeichnungen bleiben danach fest. Die Ressourcen-URIs können vom Testbed-Benutzer verändert werden. Die Funktionen zum Anpassen der Mapping-Tabelle an die Ressourcenquellen müssen von jeder Komponente zum Verwalten dieser bestimmten Datenquelle geliefert werden.

7.4.5 Repräsentationen (Plug-in *Representation*)

Die mit dem Plug-in *Resources* ausgelesene Ressource mit der Liste der Attribute, siehe dazu Abschnitt 7.4.4, wird vom Plug-in *Representation* verwendet. Die Liste der Attribute bildet die Grundlage für die Generierung einer Repräsentation dieser Ressource. Es wurden Klassen zur Generierung der Repräsentationen in Formaten HTML, XML und JSON namens *HtmlResponseBuilder*, *XmlResponseBuilder* und *JsonResponseBuilder* implementiert (siehe dazu Abbildung 6.7). Die ankommenden HTTP-Anfragen werden auf das HTTP-Header *Accept* untersucht. Wenn im HTTP-Header *Accept* das angegebene Repräsentationsformat der Ressourcen auch vom Testbed unterstützt wird, dann wird die Repräsentation der angefragten Ressource im gewünschten Format generiert und an die Client-Anwendung geschickt. Die implementierten Formate können über die Schnittstelle *ConfigurationAPI* de- und aktiviert werden. Wenn eine HTTP-Anfrage empfangen wird, die eine Repräsentation der Ressource in einem auf dem Testbed deaktivierten Format verlangt, dann wird eine HTTP-Statusmeldung über das nicht akzeptierte Format abgeschickt.

Abbildung 7.4 zeigt die Klasse zur Generierung von Repräsentationen der Ressourcen in JSON-Format. Die Klassen *JsonFactory* und *JsonGenerator* kommen aus dem Paket *org.codehaus.jackson*. Abbildung 7.5 demonstriert beispielhaft das Layout der mit der Klasse *JsonResponseBuilder* erstellten Repräsentation einer Ressource mit zwei Attributen.

```
public class JSONResponseBuilder {


    public void generate(HttpServletRequest request,
        HttpServletResponse response, Resource resource) throws Exception {

        response.setContentType("application/json");
        JsonFactory f = new JsonFactory();
        JsonGenerator g = f.createJsonGenerator(response.getOutputStream());
        g.writeStartObject();
        g.writeStringField("resource", resource.getId());
        g.writeObjectFieldStart("attributes");
        for (Attribute a : resource.get(request, response)) {
                    g.writeFieldName(a.getName());
                    g.writeStartObject();
                    g.writeStringField("value", a.getValue());
                    g.writeStringField("link", a.getLink());
                    g.writeStringField("type", a.getType());
                    g.writeEndObject();
        }
        g.writeEndObject();
        g.writeEndObject();
        g.close();
    }
```

Abbildung 7.4: Repräsentation in JSON-Format

```
{
  "resource": "http://...resourceid/",
  "attributes": {
    "name-1": {
      "value": "value-1",
      "link": "http://...link1/",
      "type": "type-1"
    },
    "name-2": {
      "value": "value-2",
      "link": "http://...link-2/",
      "type": "type-2"
    }
  }
}
```

Abbildung 7.5: Layout in JSON-Format

Die Klasse *XmlResponseBuilder* zur Generierung der Repräsentation in XML-Format zeigt Abbildung 7.6. Auch wie beim JSON-Format wird in Abbildung 7.7 beispielhaft das Layout der XML-Repräsentation einer Ressource mit zwei Attributen demonstriert.

Die Klasse *HtmlResponseBuilder* zur Generierung der Ressourcen-Repräsentationen in HTML-Format wurde nach dem gleichen Ansatz wie die Klasse *XmlResponseBuilder* implementiert und wird hier daher nicht präsentiert. Die so erstellten Repräsentationen werden bis zur nächsten HTTP-Anfrage zwischengespeichert, um anderen Plug-ins zur Verfügung zu stehen.

```
public class JSONResponseBuilder {

    public void generate(HttpServletRequest request,
        HttpServletResponse response, Resource resource) throws Exception {

        response.setContentType("application/json");
        JsonFactory f = new JsonFactory();
        JsonGenerator g = f.createJsonGenerator(response.getOutputStream());
        g.writeStartObject();
        g.writeStringField("resource", resource.getId());
        g.writeObjectFieldStart("attributes");
        for (Attribute a : resource.get(request, response)) {
                    g.writeFieldName(a.getName());
                    g.writeStartObject();
                    g.writeStringField("value", a.getValue());
                    g.writeStringField("link", a.getLink());
                    g.writeStringField("type", a.getType());
                    g.writeEndObject();
        }
        g.writeEndObject();
        g.writeEndObject();
        g.close();
    }
```

Abbildung 7.6: Repräsentation in XML-Format

```
<resource xlink:href="http://...resourceid/">
    <name-1 xlink:href="http://...link-1/" type="type-1">value-1</name-1>
    <name-2 xlink:href="http://...link-2/" type="type-1">value-2</name-2>
</resource>
```

Abbildung 7.7: Layout in XML-Format

7.4.6 Caching (Plug-in *Caching*)

Zum Steuern des Verhaltens der Client-Anwendung bezüglich der Verwendung von Caching können bestimmte HTTP-Header der HTTP-Antwort angehängt werden. Beim Plug-in *Caching* kann zum Beispiel *Entity-Tag* aktiviert werden. Für die Berechnung des *Entity-Tag* wird die vom Plug-in *Representation* zwischengespeicherte Repräsentation der Ressource herangezogen. Die zu setzenden Attribute für andere HTTP-Header werden aus der Plug-in-Konfiguration ausgelesen, die der Testbed-Benutzer beim Aktivieren der jeweiligen HTTP-Header angegeben haben muss. Das Setzen der HTTP-Header erfolgt wie schon in Abschnitten 7.4.2 und 7.4.3 demonstriert wurde.

7.4.7 Cookies (Plug-in *Cookies*)

Vom Testbed-Benutzer können Cookies mit dem Plug-in *Cookies* definiert werden. Die definierten Cookies werden mit der Klasse *javax.servlet.http.Cookie* erstellt:

Cookie cookie = new Cookie(**“name“**, **“value“**);
cookie.setComment(**“comment“**);
cookie.setDomain(**“/“**);
cookie.setMaxAge(10);
cookie.setPath(**“/testbed“**);

```
cookie.setVersion(1);
cookie.setSecure(false);
```

Die in der HTTP-Anfrage enthaltene Cookies werden automatisch der HTTP-Antwort angehängt. Um das zu verhindern, wird das maximale Alter der unerwünschten Cookies auf Null gesetzt:

```
Cookie[] cookies = request.getCookies();
for (Cookie cookie : cookies) {
    cookie.setMaxAge(0);
}
```

Es werden drei verschiedene Methoden zum Behandeln von Cookies, die mit der HTTP-Anfrage empfangen werden, bereitgestellt. Die eine implementierte Methode sorgt dafür, dass alle empfangenen Cookies nach dem oben beschriebenen Verfahren gelöscht werden. Die zweite implementierte Methode löscht nur die Cookies, deren Namen in der Plug-in-Konfiguration eingetragen sind. Die dritte Alternative löscht alle Cookies bis auf die Ausnahmen, deren Namen in der Plug-in-Konfiguration angegeben sind.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Testbed zum Testen von REST-basierten Client-Anwendungen entwickelt. Die funktionalen Anforderungen an das Testbed aus Abschnitt 4.1 sind mit Plug-ins realisiert. Dadurch wird das Testbed modular gehalten, was in Bezug auf Umfang, Komplexität und Erweiterbarkeit des Testbeds von Vorteil ist. Die Konfiguration der Plug-ins kann manuell mithilfe eines Webbrowsers mit der dafür vorgesehener HTML-Repräsentation erfolgen. Für das automatisierte Konfigurieren der Plug-ins steht die dafür besser geeignete XML-Repräsentation zur Verfügung. Die konfigurierten Parameter sind nach dem Modifizieren sofort wirksam und werden von den Plug-ins eingesetzt, ohne das Testbed neu starten zu müssen. Es wurden folgende Plug-ins implementiert:

- *Resources*: Liest, löscht, modifiziert Ressourcen in einer Datenquelle und legt neue Ressourcen an. Als Datenquelle wurde eine relationale Datenbank verwendet. Die Zuordnung von URIs zu Ressourcen wird in einer Tabelle gehalten, die ein Testbed-Benutzer an eigene Bedürfnisse anpassen kann.
- *Logging*: Die HTTP-Anfragen und -Antworten genauso wie deren Bearbeitungen können auf dem Testbed persistent protokolliert werden. Die Testbed-Benutzer können auf das Protokoll mithilfe eines Web Services zugreifen. Dadurch bekommt der Testbed-Benutzer die nötige Unterstützung zum Auffinden der Fehlerquellen bei Unzulänglichkeiten im Verhalten der getesteten Client-Anwendung.
- *Autorization*: Das Plug-in zum Autorisieren der HTTP-Anfragen unterstützt die Authentifikationsverfahren *Basic Access Authentication* und *Digest Access Authentication*.
- *Representation*: Damit können Repräsentationen von Ressourcen in HTML-, XML- und JSON-Format erstellt werden.
- *HTTP-Statusmeldungen*: Mit der Simulation von HTTP-Statusmeldungen kann das implementierte Verhalten der Client-Anwendungen mit dem gemäß der *HTTP-1.1-Spezifikation* beschriebenen Verhalten verglichen werden.
- *Cookies*: Durch die Möglichkeit zum Setzen der Cookies in den HTTP-Antworten kann die Funktionalitäten von Client-Anwendungen überprüft werden, die sich mit der Verwaltung von Cookies beschäftigen.
- *Caching*: Die Client-Anwendungen, die Vorteile von Caching nutzen, können darauf getestet werden, ob die HTTP-Anfragen beziehungsweise -Antworten zum erwarteten Verhalten bezüglich der Cache-Nutzung führen.

Es wurden Funktionalitäten beim REST Testbed mit Plug-ins realisiert, die bereits ein breites Spektrum an Testszenarien abdecken. Es können mit dem Testbed somit einige Testfälle an den zu testenden Client-Anwendungen bezüglich der Erfüllbarkeit der funktionalen und

nicht-funktionalen Anforderungen überprüft werden. Bei der Architektur wurde mit dem Plug-in-Konzept dafür gesorgt, dass das Testbed erweiterbar ist. Somit kann der Umfang des Testbeds nach Bedarf um weitere Bereiche durch neue Plug-ins ergänzt werden. Im folgenden Ausblick werden nun einige dieser möglichen Plug-in-Ergänzungen diskutiert beziehungsweise empfohlen.

Wegen dem Ansatz der dynamischen Zuordnung von URIs zu Ressourcen kann keine WADL-Spezifikation vom Jersey Framework erstellt werden, siehe Abschnitt 7.4.4. Wenn es die Möglichkeit bestehen soll, dass eine Client-Anwendung ein Web Service automatisiert anhand der WADL-Beschreibung benutzt, dann ist die Entwicklung eines weiteren Plug-ins zur Generierung der WADL-Beschreibung der Testservice-Schnittstelle erforderlich.

Für das automatisierte Konfigurieren der Plug-ins kann die XML-Repräsentation eingesetzt werden. Die XML-Repräsentation ist für die eigenen Entwicklungen von Konfigurationswerkzeugen zum REST Testbed besser geeignet als die HTML-Repräsentation. Unter der Benutzung der XML-Repräsentation kann ein Wrapper für die Konfigurationsschnittstelle des REST Testbeds entwickelt werden. Ein Wrapper für die Konfigurationsschnittstelle würde es ermöglichen, die Konfiguration der Plug-ins im REST Testbed elegant durchzuführen. Der Benutzer dieses Wrappers würde von der XML-Repräsentation nichts mitbekommen. Die Logik zum Verwalten der konfigurierbaren Werte von Plug-ins würde im Wrapper versteckt bleiben. Durch die Bereitstellung dieses Wrappers würden sich der Aufwand und die Fehlerquellen, die beim Erstellen eines Testfalls entstehen, enorm reduzieren.

Durch Erweiterung der Funktionalität vom Plug-in *Resources* können andere Ressourcenquellen wie CSV-, XML-Dateien sowie Web Services implementiert werden. Die Client-Anwendungen erfordern meist spezifische Darstellungen von Ressourcen, sodass die mit dem Testbed gelieferten Repräsentationen nur selten geeignet wären. Die Implementierungen von beispielhaften Repräsentationen des Plug-ins *Representation* können herangezogen werden, um eine eigene Implementierung zur Erstellung von passenden Repräsentationen der Ressourcen schnell zu bewerkstelligen. Das Plug-in *Autorization* kann durch weitere Verfahren wie OAuth und OAuth2 erweitert werden. Das REST Testbed unterstützt HTTP- und die Hypertext Transfer Protocol Secure (HTTPS)-Anfragen. Daher steht auch einer Erweiterung dieses Plug-ins um eine Authentifizierung durch ein Secure Sockets Layer (SSL)-Zertifikat nichts im Weg.

Literaturverzeichnis

[Apa] APACHE SOFTWARE FOUNDATION: *Apache Log4j 2*. http://logging.apache.org/log4j/2.x/. http://logging.apache.org/log4j/2.x/

[Atoa] ATOS RESEARCH & INNOVATION: *SOA4ALL Studio - Projektseite*. http://technologies.kmi.open.ac.uk/soa4all-studio/. http://technologies.kmi.open.ac.uk/soa4all-studio/

[Atob] ATOS RESEARCH & INNOVATION: *SOA4SOA - Enabling a Web of billions of serveces*. http://www.soa4all.eu/. http://www.soa4all.eu/

[BL98] BERNERS-LEE, T.: Uniform Resource Identifiers (URI) / MIT. Version: 1998. http://www.rfc-archive.org/getrfc.php?rfc=2396. 1998. – RFC 2396

[Böh06] BÖHM, Oliver: *Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP*. Dpunkt.Verlag GmbH, 2006. – ISBN 9783898643306

[BPSM+08] BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, C. M. ; MALER, Eve ; YERGEAU, François: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. http://www.w3.org/TR/REC-xml/. Version: November 2008

[BS07] BAYER, Thomas ; SOHN, Dirk M.: Eine Einführung: REST Web Services. In: *T3N Magazin* 8. Ausgabe (2007). http://t3n.de/magazin/rest-web-services-einfuhrung-219976/

[CD08] COCKBURN, A. ; DIETERLE, R.: *Use Cases effektiv erstellen*. mitp-Verlag, 2008 (mitp bei Redline). – ISBN 9783826617966

[Cro06] CROCKFORD, Douglas: The application/json Media Type for JavaScript Object Notation (JSON) / IETF. 2006. – RFC 4627

[Ecl] ECLIPSE FOUNDATION, INC.: *AJDT: AspectJ Development Tools*. http://www.eclipse.org/ajdt/. http://www.eclipse.org/ajdt/

[Fel10] FELIPE, L.O.: *Design and Development of a REST-based Web Service Platform for Applications Integration*, UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC), Diplomarbeit, 2010

[FFK+10] FENSEL, Dieter ; FISCHER, Florian ; KOPECKÝ, Jacek ; KRUMMENACHER, Reto ; LAMBERT, Dave ; VITVAR, Tomas: *WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web*. W3C Recommendation. http://www.w3.org/Submission/2010/SUBM-WSMO-Lite-20100823/. Version: August 2010

[FHBH+99] FRANKS, J. ; HALLAM-BAKER, P. ; HOSTETLER, J. ; LAWRENCE, S. ; LEACH, P. ; LUOTONEN, A. ; STEWART, L.: HTTP Authentication: Basic and Digest Access Authentication / Internet Engineering Task Force. Version: June 1999. http://www.rfc-editor.org/rfc/rfc2617.txt. 1999. – RFC 2617

[Fie00] FIELDING, Roy: *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. – S. 76-106

[Fie08] FIELDING, Roy: REST APIs must be hypertext-driven. (2008). http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[Fie09] FIELDING, Roy ; NETWORK WORKING GROUP (Hrsg.): *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group, 2009. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[FL07] FARRELL, Joel ; LAUSEN, Holger: Semantic Annotations for WSDL and XML Schema / World Wide Web Consortium. Version: 2007. http://www.w3.org/TR/sawsdl/. 2007. – W3C Working Draft

[FLS07] FRÜHAUF, K. ; LUDEWIG, J. ; SANDMAYR, H.: *Software-Prüfung: Eine Anleitung zum Test und zur Inspektion*. 6. Auflage. vdf, Hochschulverlag an der ETH Zürich, 2007 (vdf-Lehrbuch Informatik). – ISBN 9783728130594

[GEN] GENESIS - PROJEKTSEITE: *GENESIS - Generating SOA Testbed Instrastructures*. http://www.infosys.tuwien.ac.at/prototypes/Genesis/Genesis_index.html

[Gup05] GUPTA, S.: *Pro Apache Log4j*. Apress, 2005 (ITPro collection). – ISBN 9781430200345

[HR02] HORN, E. ; REINKE, T.: *Softwarearchitektur und Softwarebauelemente: eine Einführung für Softwarearchitekten*. Hanser, 2002. – ISBN 9783446213005

[HS07] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: The Java API for RESTful Web Services*. Java Specification Request (JSR) 311, Oktober 2007

[IEE90] IEEE COMPUTER SOCIETY. STANDARDS COORDINATING COMMITTEE AND INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS AND IEEE STANDARDS BOARD: *I.E.E.E. Standard Glossary of Software Engineering Terminology*. The Institute, 1990 (IEEE Std). – ISBN 9781559370677

[iSe] ISERVE - PROJEKTSEITE: *iServe - Where Linked Data Meets Services*. http://iserve.kmi.open.ac.uk/

[JD10] JUSZCZYK, Lukasz ; DUSTDAR, Schahram: Script-based generation of dynamic testbeds for soa. In: *ICWS*, IEEE Computer Society, 2010

[JD11] JUSZCZYK, Lukasz ; DUSTDAR, Schahram: Automating the Generation of Web Service Testbeds Using AOP. In: ZAVATTARO, Gianluigi (Hrsg.) ; SCHREIER, Ulf

(Hrsg.) ; PAUTASSO, Cesare (Hrsg.): *ECOWS*, IEEE Computer Society, 2011. – ISBN 978–1–4577–1532–7, S. 143-150

[jQua] JQUERY - PROJEKTSEITE: *jQuery*. http://jquery.com/. http://jquery.com/

[jQub] JQUERY VALIDATION PLUGIN - PROJEKTSEITE: *jQuery Validation Plugin*. http://jqueryvalidation.org/

[JTD08] JUSZCZYK, Lukasz ; TRUONG, Hong L. ; DUSTDAR, Schahram: GENESIS - A Framework for Automatic Generation and Steering of Testbeds of ComplexWeb Services. In: *ICECCS*, IEEE Computer Society, 2008. – ISBN 0–7695–3139–3, S. 131-140

[KM97] KRISTOL, D. ; MONTULLI, L.: *HTTP State Management Mechanism*. RFC 2109 (Proposed Standard). http://www.ietf.org/rfc/rfc2109.txt. Version: February 1997 (Request for Comments). – Abgelöst durch RFC 2965

[Kna07] KNABE, Christoph: *Aspektorientierte Programmierung mit AspectJ 5*. http://public.beuth-hochschule.de/~knabe/fach/ats/AOP-Skript.pdf. Version: 2007

[LSS+10] LAMPE, Ulrich ; SCHULTE, Stefan ; SIEBENHAAR, Melanie ; SCHULLER, Dieter ; STEINMETZ, Ralf: Adaptive matchmaking for RESTful services based on hRESTS and MicroWSMO. In: BINDER, Walter (Hrsg.) ; SCHULDT, Heiko (Hrsg.): *WEWST*, ACM, 2010 (ACM International Conference Proceeding Series). – ISBN 978–1–4503–0238–8, S. 10-17

[MBH+04] MARTIN, David ; BURSTEIN, Mark ; HOBBS, Jerry ; LASSILA, Ora ; MCDERMOTT, Drew ; MCILRAITH, Sheila ; NARAYANAN, Srini ; PAOLUCCI, Massimo ; PARSIA, Bijan ; PAYNE, Terry R. ; SIRIN, Evren ; SRINIVASAN, Naveen ; SYCARA, Katia: OWL-S: Semantic Markup for Web Services. (2004). http://eprints.ecs.soton.ac.uk/12687/

[Ora] ORACLE AMERICA, INC.: *Jersey: RESTful Web Services in Java*. https://jersey.java.net/. https://jersey.java.net/

[PBG04] POSCH, T. ; BIRKEN, K. ; GERDOM, M.: *Basiswissen Softwarearchitektur*. dpunkt-Verlag, 2004. – ISBN 9783898642705

[PZL08] PAUTASSO, Cesare ; ZIMMERMANN, Olaf ; LEYMANN, Frank: RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In: *WWW '08: Proceeding of the 17th international conference on World Wide Web*. New York : ACM, 2008 (Proceedings of the 17th international conference on World Wide Web), S. 805-814

[RHJ99] RAGGETT, Dave ; HORS, Arnaud L. ; JACOBS, Ian: *HTML 4.01 Specification*. W3C Recommendation. http://www.w3.org/TR/html4. Version: December 1999

[Ric07] RICHARDSON, Leonard: *Web Services mit REST*. O'Reilly Verlag GmbH & Co. KG, 2007. – ISBN 978–3897217270

[SKA+09] SCHREDER, Bernhard ; KRUMMENACHER, Reto ; ABELS, Sven ; PARIENTE, Tomás ; RICHARDSON, Marc ; VILLA, Matteo ; DI MATTEO, Giovanni: *D1.5.2 Setup SOA4All Testbeds*. `http://www.soa4all.eu/pdocs/deliverables/D1.5.2+SETUP+SOA4ALL+TESTBEDS.PDF`. Version: 2009. – Work Package: WP1 - SOA4All Runtime,

[SW02] SNEED, H.M. ; WINTER, M.: *Testen objektorientierter Software.: Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme.* Hanser Fachbuchverlag, 2002. – ISBN 9783446218208

[SWE] SWEET - PROJEKTSEITE: *SWEET - Semantic Web sErvice Editing Tool*. `http://sweet.kmi.open.ac.uk/`

[Tie09] TIEMEYER, E.: *Handbuch IT-Management: Konzepte, Methoden, Lösungen und Arbeitshilfen für die Praxis*. 3. Auflage. Hanser Fachbuchverlag, 2009. – ISBN 9783446418424

[Til11] TILKOV, S.: *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. 2. Auflage. Dpunkt.Verlag GmbH, 2011. – ISBN 9783898647328

Alle URLs wurden zuletzt am 02.08.2013 geprüft.

Erklärung

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

Stuttgart, 06.08.2013 ———————————

(Nick Eisenbraun)