Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

ANDREAS BAUER
MASTER THESIS

MANAGING ORGANIZATION DATA FOR PATCH-FLOW MEASUREMENT

Eingereicht am 2. Juni 2017

Betreuer:
Maximilian Capraro M.Sc
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 2. Juni 2017

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 2. Juni 2017

Abstract

Open source practices and the establishment of an open source like culture within organizations is also called Inner Source (IS). The existing software Collaboration Metric Suite (CMSuite) provides metrics about collaboration between software projects.

These metrics can validate the application of IS in organizations. However, the underlying model of the CMSuite currently only supports simple hierarchical organizational structures. Organizations with a more complex structure can not be correctly mapped. In this thesis, a model was designed and integrated into the CMSuite, that fulfills the requirements of a complex organizational structure. For this purpose, two case studies, which show the weaknesses of the current model, were studied.

Finally, it was shown that dealing with complex organization structures is not a problem for the CMSuite anymore.

Zusammenfassung

Werden Open Source Praktiken und die Etablierung einer Open Source ähnlichen Kultur innerhalb einer Organisation angewendet, so wird dies auch als Inner Source (IS) bezeichnet. Die bestehende Software Collaboration Metric Suite (CMSuite) stellt Metriken für die Zusammenarbeit zwischen Softwareprojekten bereit. Anhand dieser Metriken kann die Anwendung von IS in Organisationen validiert werden.

Das der CMSuite zugrunde liegende Datenmodell unterstützt derzeit aber nur einfache hierarchische Organisationsstrukturen. Organisationsformen mit einem komplexeren Aufbau können somit nicht korrekt abgebildet werden. In dieser Arbeit wird ein Organisationsmodell konzipiert und in die CMSuite integriert. Dieses Datenmodell wird den Anforderungen an eine komplexe Organisationsstruktur gerecht. Hierzu werden unter anderem zwei durchgeführte Fallstudien, welche die Schwächen des gegenwärtigen Datenmodells aufzeigen, analysiert.

Zum Schluss wird gezeigt, dass der Umgang mit komplexen Organisationsstrukturen, die der Unternehmenspraxis entsprechen, für die CMSuite kein Problem mehr darstellt.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Hibernate ORM . . . 3
- 2.2 Liquibase . . . 4
- 2.3 Representational State Transfer . . . 5
  - 2.3.1 Constraints . . . 5
  - 2.3.2 Ressourcen . . . 6

3 Anforderungen **8**
- 3.1 Anforderungsanalyse des Organisationsmodells . . . 8
- 3.2 Anforderungsanalyse Webclient . . . 11

4 Architektur und Design **13**
- 4.1 Architektur der CMSuite . . . 13
- 4.2 Design des Organisationsmodells . . . 14
  - 4.2.1 Entwurfsmuster . . . 14
  - 4.2.2 Resultierendes Modell . . . 16
- 4.3 Design des Webclients und einer REST-API . . . 18
  - 4.3.1 REST-API . . . 18
  - 4.3.2 Webclient . . . 20

5 Implementierung **21**
- 5.1 Implementierung des Organisationsmodells . . . 21
  - 5.1.1 Refactoring am Datenmodell . . . 21
  - 5.1.2 Verlinkung von OrgElement-Objekten durch OrgLinks . . . 22
  - 5.1.3 Einführung von Dimensionen . . . 23
  - 5.1.4 Zeitliche Änderung der Beziehung zwischen Organisationselementen . . . 24
  - 5.1.5 Inner-Source-Projekte und Personen . . . 25

5.1.6 Umgang mit Dimensionen und Zeiten 25
5.1.7 Unit Tests 27
5.2 Implementierung des Webclients 29
5.2.1 REST Backend 29
5.2.2 Angular Frontend 30

6 Evaluation **32**
6.1 Evaluation des Organisationsmodells 32
6.2 Evaluation des Webclients 36

7 Fazit **38**

Anhänge **39**
Anhang A Beispiel eines Liquibase ChangeSet 39
Anhang B Beispiel für die Dokumentation einer REST-Ressource . . 40
Anhang C Inhalt des Datenträgers 41

Abbildungsverzeichnis **42**

Tabellenverzeichnis **42**

Literaturverzeichnis **44**

Abkürzungsverzeichnis

OS	Open Source
IS	Inner Source
CMSuite	Collaboration Metric Suite
SCM	Source Code Management
REST	Representational State Transfer
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
HTML	Hypertext Markup Language
RDBMS	relationales Datenbankmanagementsystem
ORM	Object/Relational Mapping
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
XML	Extensible Markup Language
DAO	Data Access Object
DTO	Data Transfer Object
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
TDD	Test-Driven Development

1 Einleitung

1.1 Motivation

Inner Source (IS) bezeichnet die Anwendung von Open Source (OS) Praktiken und die Etablierung einer Open Source ähnlichen Kultur innerhalb einer Organisation. Das bedeutet aber nicht, dass diese Organisation auch OS-Software entwickeln wird, sondern die Software proprietär bleiben kann und nur intern die Entwicklung, gemäß dem Open Source Gedanken, geöffnet wird. Die Zusammenarbeit innerhalb der Organisation kann hierbei durch das Messen des sogenannten Patch-Flow quantifiziert werden. Die Patches (kleine Codeänderungen), die den Patch-Flow bilden, können ein Bugfix, neue Funktionalitäten oder Ergänzungen des Codes sein (Capraro & Riehle, 2016).

Die bereits existierende Software CMSuite stellt durch das Messen des Patch-Flows Metriken zur Validierung von Inner Source bereit. Ein Bestandteil dieser Software ist der Patch-Flow-Crawler, dessen Aufgabe es ist, die Patches und weitere Informationen aus Source Code Management (SCM) Systemen oder anderen Datenquellen zu erfassen und für eine spätere Analyse aufzubereiten. Zu den Patches wird eine Abbildung der Organisationsstruktur sowie weitere Informationen benötigt, die die Software-Projekte mit den zugehörigen Organisationsstrukturen verknüpfen.

Bei der Durchführung von zwei Fallstudien traten Schwächen in zwei Problemfeldern des Patch-Flow-Crawlers in Erscheinung. Zum einen ist es mit dem Patch-Flow-Crawler zugrunde liegenden Organisationsmodell nicht möglich, komplexere Organisationsstrukturen, also solche die nicht einer Baumstruktur entsprechen, abzubilden. Diese Abbildung ist in der Realität allerdings nötig, da echte Organisationen meist komplexer sind. Des Weiteren ist der Import von Organisationsstrukturen problematisch. Händische Anpassungen einer importierten Organisationsstruktur können in einigen Fällen notwendig sein. Zusätzlich wäre eine manuelle Erstellung und vor allem eine Anpassung von Strukturen eine sinnvolle Ergänzung. Für Anwenderinnen und Anwender ist derzeit aber weder das eine noch das andere möglich.

Der Beitrag dieser Arbeit besteht aus den folgenden Punkten:

- Strukturierte Analyse der Schwächen des gegenwärtigen Organisationsmodells der CMSuite
- Konzipierung eines Organisationsmodells für komplexe Organisationsstrukturen
- Integration des entwickelten Organisationsmodells in die bestehende CMSuite
- Implementierung von REST-Services, die Schnittstellen zum Umgang mit dem Organisationsmodell bereitstellen
- Implementierung eines Webclients, welcher die Erstellung und Bearbeitung von Organisationsstrukturen ermöglicht
- Evaluation des erstellten Organisationsmodells und des Webclients

1.2 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel gegliedert, die nachfolgend beschrieben werden:

Kapitel 2, Grundlagen: Hier werden alle nötigen Grundlagen über die verwendeten Technologien vermittelt, die zum Verständnis dieser Arbeit notwendig sind.

Kapitel 3, Anforderungen: Die Anforderungen an ein Organisationsmodell und einen Webclient werden definiert und näher beschrieben.

Kapitel 4, Architektur und Design: Im Rahmen dieses Kapitels wird der Designentwurf für ein Organisationsmodell und einen Webclient vorgestellt. Zudem wird zum besseren Verständnis die Architektur der CMSuite und des Patch-Flow-Crawlers vorgestellt.

Kapitel 5, Implementierung: Auf die umgesetzte Implementierung des im vorherigen Kapitels erstellten Designentwurfs wird in diesem Kapitel Bezug genommen.

Kapitel 6, Evaluation: Die Evaluation der umgesetzten Lösung soll im Kapitel 6 beschrieben werden.

Kapitel 7, Fazit: Abschließend rundet ein Fazit die Arbeit ab.

2 Grundlagen

In diesem Kapitel werden nötige Begrifflichkeiten und Grundlagen erläutert, um Verständnisprobleme in den weiteren Kapiteln zu vermeiden.

2.1 Hibernate ORM

Hibernate ORM[1] ist ein Object/Relational Mapping (ORM) Framework, welches dabei hilft, eine Anwendung zu persistieren. Persistents bedeutet in diesem Fall vereinfacht, dass die Daten der Anwendung den Anwendungsprozess überleben (Hibernate ORM, 2017).

Die Persistierung von Objekten ist eine wesentliche Anforderung an Java-Anwendungen. In der Java-Welt wird mit Objekten gearbeitet, die einen Zustand und ein Verhalten besitzen. Dementgegen werden in der Welt der relationalen Datenbanken Daten in Form von Tabellen gespeichert. Es ist immer eine herausfordernde Aufgabe, die Java-Objekte in die Welt der relationalen Datenbanken zu überführen. Dieser Vorgang wird oft als Objekt-Relationals-Mapping bezeichnet (Konda, 2014).

ORM ist nicht die Lösung für alle Persistierungs-Aufgaben, kann aber einer Entwicklerin oder einem Entwickler viel Arbeit bei der Persistierung von Objekten ersparen. Unter anderem müssen keine komplexen SQL-Abfragen[2] mit vielen JOIN-Befehlen[3] geschrieben werden oder Werte zwischen Objekten und JDBC-Abfragen[4] kopiert werden. Zusätzlich können ORM-Lösungen auch Optimierungstechniken wie Caching bereitstellen, welche selbst nicht trivial zu implementieren sind (Bauer, King & Gregory, 2015).

[1] http://hibernate.org
[2] Structured Query Language (auf Deutsch: Strukturierte Abfrage-Sprache)
[3] Mittels JOIN können mehrere Tabellen zusammengefügt werden.
[4] Java Database Connectivity (JDBC) ist eine Datenbankschnittstelle der Java-Plattform.

2.2 Liquibase

Liquibase[5] ist eine Open-Source-Bibliothek, mit der Änderungen an Datenbanken verwaltet, verfolgt und ausgeführt werden können. Alle Änderungen an der Datenbank werden hiebei in einer für Menschen lesbaren Form gespeichert und können in einem Source Code Management System aufgenommen werden. Hierdurch kann die Datenbank versioniert werden. Die Kommunikation kann mit jeder Datenbank erfolgen, die über den JDBC-Standard angesprochen werden kann (Voxland, 2011).

Die grundlegenden Konzepte von *Liquibase* sind laut Voxland (2017):

Changelog Datei

Alle Änderungen durch *Liquibase* sind auf die DatabaseChangeLog-Datei zurückzuführen. Dabei kann diese Datei in den Formaten XML[6], JSON[7], YAML[8] oder SQL vorliegen.

ChangeSet

Ein *ChangeSet* stellt einen Satz von zusammenhängenden Änderungsoperationen dar. *Liquibase* überprüft bei der Ausführung, welche *ChangSets* in der *databasechangelog* Tabelle bereits durchgeführt wurden und führt nur diejenigen *ChangSets* aus, die noch nicht durchgeführt wurden. Jedes *ChangeSet* wird als eigenständige Transaktion ausgeführt und kann im Fehlerfall durch einen Roll-Back-Mechanismus rückgängig gemacht werden. Des Weiteren kann jedes *ChangeSet* eindeutig durch eine Kombination aus einem *id-tag* und dem *author-tag* identifiziert werden. Ein *ChangeSet* kann aus mehreren *Changes* bestehen. Ein Beispiel für ein *ChangeSet* ist dem Anhang A zu entnehmen.

Changes

Changes beschreiben, welche Änderungen an der Datenbank durchgeführt werden sollen.

[5] http://www.liquibase.org
[6] Extensible Markup Language
[7] JavaScript Object Notation
[8] YAML ist eine vereinfachte Auszeichnungssprache

Preconditions

Ein *ChangeSet* oder ein *ChangeLog* kann mit einer Vorbedingung versehen werden, die es zu erfüllen gilt.

Contexts

Mit *Contexts* kann ein *ChaneSet* für einen speziellen Kontext markiert werden und so nur in diesem ausgeführt werden. Somit können beispielsweise unterschiedliche *ChangeSets* für das Produktions- und Testumfeld definiert werden.

2.3 Representational State Transfer

Fielding (2000) entwickelte einen netzwerkbasierten Architekturstiel, den er als Representational State Transfer (REST) veröffentlichte. REST soll dabei den Anforderungen, die eine moderne Webarchitektur aufweist, besser gerecht werden. Dabei stehen besonders verteilte Hypermedia-Anwendungen im Fokus. Hierzu definierte Fielding (2000) sechs Voraussetzungen, die sogenannten Constraints, die eine REST-Architektur erfüllen muss. Ein Webservice kann als *RESTful* bezeichnet werden, wenn dieser auf der REST-Architektur basiert und demzufolge alle Constraints erfüllt sind.

2.3.1 Constraints

Client-Server: Beschreibt eine klare Trennung von Verantwortungsbereichen. So bietet der Server Dienste an, welcher der Client nutzen und nach seinen Wünschen verarbeiten kann. Das Hauptargument für diese Aufteilung ist eine gute Skalierbarkeit über mehrere Server, was für das Web von entscheidender Bedeutung ist. Zudem können Client und Server unabhängig voneinander entwickelt und, wenn nötig, ausgetauscht werden.

Stateless: Bedeutet, dass der Server keine Informationen über die Kommunikationsdauer hinaus speichern darf. Der Client muss bei jeder Anfrage an den Server alle Informationen, die zur Bearbeitung notwendig sind, mitschicken. Ein Vorteil der zustandslosen (stateless) Kommunikation ist, dass der Server keinerlei Session-Informationen zu den einzelnen Clients speichern muss.

Cache: Bereits geladene Daten können wiederverwendet werden, um die Netzwerk- und Serverlast zu reduzieren. Notwendig hierfür ist eine explizite oder implizite

Kennzeichnung der Daten als *cacheable* beziehungsweise *non-cacheable*. Der Client überprüft bei einer Anfrage an den Server, ob die benötigten Informationen sich im Cache befinden und verzichtet dann auf die Anfrage, falls die Informationen nicht veraltet sind.

Uniform Interface: Eine einheitliche Schnittstelle für unterschiedliche Komponenten ist der Hauptunterschied zu anderen Netzwerk-basierten Architekturen. Einerseits können die Ressourcen durch einen Ressource-Identifier angesprochen werden und andererseits existieren universelle Methoden, mit denen auf die Ressourcen zugegriffen werden kann. Ziel ist eine sogenannte lose Kopplung der Komponenten.

Layered System: Eine weitere Maßnahme um die Skalierbarkeit von Web-Systemen zu erhöhen, ist die Nutzung von hierarchischen Schichten. Jede Schicht bietet der darüberliegenden Schicht Dienste an und nutzt selbst Dienste der darunterliegenden Schicht. Dadurch wird die Komplexität der Architektur und die Abhängigkeiten zwischen den Komponenten reduziert. Es entsteht aber auch ein Overhead, da eine Anfrage mehrere Schichten durchlaufen muss.

Code-On-Demand (Optional): Damit ist es dem Client möglich, Funktionalitäten durch das Nachladen von Code zu adaptieren. Hierzu muss der Client aber eine entsprechende Laufzeitumgebung zur Verfügung stellen. Der entstehende Grad an Flexibilität wird aber nur selten benötigt, was dazu geführt hat, dass dieses Constraint keine wirkliche Voraussetzung darstellt und als optional gekennzeichnet ist.

2.3.2 Ressourcen

Webber, Parastatidis und Robinson (2010) und Tilkov (2011) bezeichnen Ressourcen als den zentralen Bestandteil der REST-Architektur. Dabei besitzt eine Ressource eine eindeutige Identifikation und eventuell mehrere Repräsentationen.

Identifikation von Ressourcen

Zur Identifikation von Ressourcen wird der Uniform Resource Identifier (URI) verwendet. Eine Ressource wird hierbei genau durch eine URI identifiziert. Die URIs selbst entsprechen in ihrer Form einem Uniform Resource Locator (URL)[9], mit der Ressourcen im World-Wide-Web adressiert werden.

[9]URL ist eine Teilmenge von URI, beschrieben im RFC3986 durch Network Working Group (2005)

Repräsentation von Ressourcen

Da eine Ressource in mehreren Repräsentationen vorliegen kann, hat der Client die Möglichkeit, eine bestimmte Repräsentationsform anzufordern. Der Server liefert dann auf eine Anfrage eine Repräsentation der Ressource - nie die Ressource selbst - zurück. Dabei wird mitgeteilt, in welcher Form die Ressource tatsächlich zurückgegeben wird. Dies ist unter anderem wichtig, falls das vom Client angefragte Format nicht verfügbar war. Gebräuchliche Repräsentationsformen sind JSON[10], XML[11], HTML[12] und reiner Text.

[10] definiert im RFC7159 durch Internet Engineering Task Force (IETF) (2014)
[11] Extensible Markup Language
[12] Hypertext Markup Language

3 Anforderungen

3.1 Anforderungsanalyse des Organisationsmodells

Die Anforderungsanalyse des Organisationsmodells begann mit der Analyse von zwei Fallstudien, welche die Erfahrungen mit dem gegenwärtigen Organisationsmodell aus der Sicht von zwei Firmen beschrieben. Hieraus wurden die Schwächen bei der Abbildung von Organisationsstrukturen erst aufgelistet und in einem zweiten Schritt zusammengefasst und strukturiert. Als Nächstes wurden die entstandenen Anforderungen in Zusammenarbeit mit dem Betreuer auf ihre Relevanz hin überprüft.

In der Tabelle 3.1 werden diese Anforderungen aufgelistet, bevor genauer auf die einzelnen Punkte eingegangen wird.

Tabelle 3.1: Anforderungen an das Organisationsmodell

Nr.	**Kurzbeschreibung**
A1.1	Unterstützung für mehrdimensionale Organisationsstrukturen
A1.2	Markieren einer Dimension als primäre Dimension
A1.3	Zeitliche Änderung der Organisationsstruktur
A2.1	n:m Beziehung von Personen zu Organisationselementen
A2.2	Zeitliche Änderung der Beziehung zwischen Personen und Organisationselementen
A3.1	n:m Beziehung von Inner-Source-Projekten zu Organisationselementen
A3.2	Zeitliche Änderung der Beziehung zwischen Inner-Source-Projekten und Organisationselementen

A1.1 Unterstützung für mehrdimensionale Organisationsstrukturen

Die derzeitige Abbildung von Organisationsstrukturen erfolgt als einfache hierarchische Baumstruktur. Ein Organisationselement kann hierbei genau ein übergeordnetes Element und n untergeordnete Elemente besitzen, wie in Abbildung 3.1

dargestellt wird[1]. Hiermit ist es möglich, Organisationsformen, die einem Einliniensystem entsprechen, korrekt abzubilden. Bei einem **Einliniensystem** ist jede Stelle nur durch eine einzige Verbindungslinie mit ihrer vorgesetzten Instanz verbunden. Dementsprechend erhällt sie nur von einer einzigen Instanz Anweisungen (Thommen & Achleitner, 2012). Durch Fayol (1929) ist dies auch als Prinzip der *Einheit der Auftragserteilung* bekannt.

Organisationsformen, die aber einem **Mehrliniensystem** gleichkommen, können hingegen durch das gegenwärtige Datenmodell nicht korrekt abgebildet werden. In einem idealtypischen Mehrliniensystem ist jede Stelle einer Mehrzahl von übergeordneten Instanzen unterstellt und entspricht somit dem Prinzip der Mehrfachunterstellung (Thommen & Achleitner, 2012).

Eine häufig vorkommende Organisationsform in der Unternehmenspraxis ist die **Matrixorganisation**. Die Matrixorganisation gehört zu den Mehrliniensystemen. Sie ermöglicht es zugleich zwei bedeutende Organisationsziele, wie z.B. technische Brillanz und starke Kundenorientierung, zu erreichen. Solche komplexeren Organisationsstrukturen sind erforderlich, um mit komplexer werdenden Organisationsumwelten umgehen zu können (Wohlwender, 2014).

A1.2 Markieren einer Dimension als primäre Dimension

Wenn eine mehrdimensionale Organisationsstruktur besteht, sollte eine Dimension als primäre Dimension markiert werden können. Dies ist unter anderem notwendig, da die Software zur Auswertung des Patch-Flows nur mit einer Dimension umgehen kann. Zudem ist so der Anwenderin und dem Anwender ersichtlich, welche Dimension in erster Linie die Organisationsstruktur bildet.

A1.3 Zeitliche Änderung der Organisationsstruktur

Ein weiterer negativer Aspekt des Organisationsmodells sind die statischen Beziehungen zwischen Organisationselementen untereinander, den zugewiesenen Personen und der Zuständigkeit für Inner-Source-Projekte. Organisationen sind aber nicht statisch, sondern stehen mit ihrer Umwelt in vielfältiger Weise in Beziehung und werden entsprechend von dieser beeinflusst (Scherm, 2007). Daher ist es notwendig, dass das Organisationsmodell mit den zeitlichen Änderungen dieser Beziehungen umgehen kann. Dies gilt auch für die beiden anderen zeitlichen Änderungen von Beziehungen, die in den Anforderungen **A2.2** und **A3.2** definiert wurden.

[1] Die Wurzel stellt eine Ausnahme mit keinem übergeordneten Element dar.

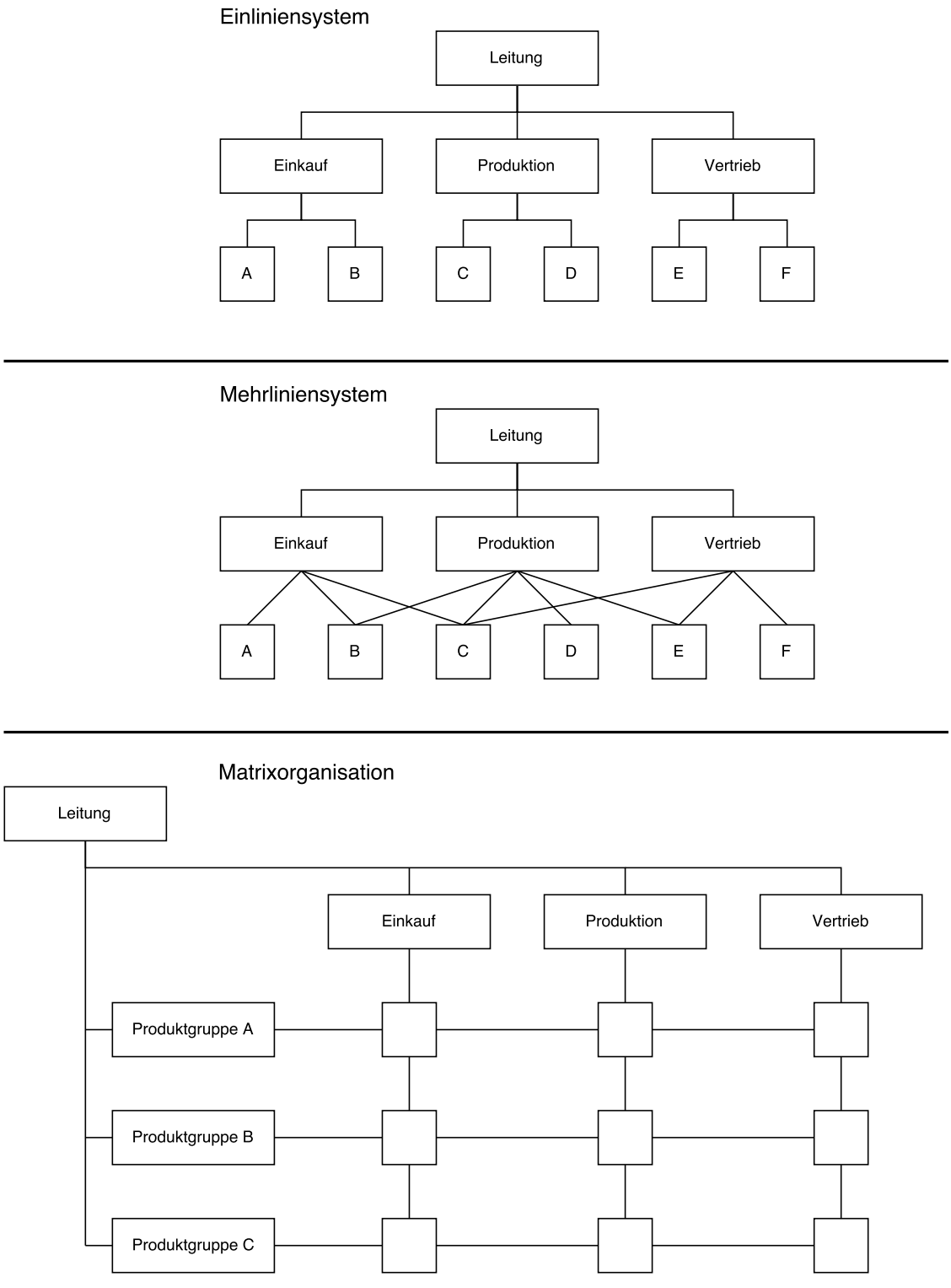

Abbildung 3.1: Organisationsstrukturen im Vergleich

A2.1 n:m Beziehung von Personen zu Organisationselementen

Zwischen Organisationselementen und Personen besteht derzeit eine *1:n* Beziehung. Daher ist eine Person nur einem Organisationselement zugewiesen. Um ein komplexeres Szenario abbilden zu können, müsste es möglich sein, eine Person mehreren Organisationselementen zuzuweisen. So könnte beispielsweise eine Person einer Abteilung zugeordnet werden und zugleich Mitglied eines Projektteams sein.

A3.1 n:m Beziehung von Inner-Source-Projekten zu Organisationselementen

Ähnlich der vorherigen Anforderung besteht auch hier eine *1:n* Beziehung zwischen Organisationselementen und Inner-Source-Projekten. Szenarien, in denen ein Inner-Source-Projekt mehreren Organisationselementen zugeordnet ist, sollten nicht ausgeschlossen werden.

3.2 Anforderungsanalyse Webclient

Aus der Analyse der Fallstudien ergab sich auch der Bedarf eines Webclients, der die Bearbeitung bestehender Organisationsstrukturen ermöglicht. Dies ist notwendig, da ein durch den Import erstandener inkonsistenter Datensatz nicht händisch korrigiert werden kann.

Daher soll ein minimaler Webclient erstellt werden, welcher Anwenderinnen und Anwender die Erstellung und Bearbeitung von Organisationsstrukturen ermöglicht. Eine Zusammenfassung der Anforderungen an einen Webclient ist in der Tabelle 3.2 zu sehen, auf die anschließend näher eingegangen wird.

Tabelle 3.2: Anforderungen an den Webclient

Nr.	**Kurzbeschreibung**
A4.1	Bereitstellen einer REST-API für den Webclient
A4.2	CRUD-Operationen für relevante Entitäten
A4.3	Verlinkung zwischen Entitäten bearbeiten und erstellen

A4.1 Bereitstellen einer REST-API für den Webclient

Voraussetzung für die Verwendung eines Webclients, welcher Daten des Organisationsmodells nutzen kann, ist eine zuvor bereitgestellte API in der CMSuite.

Als technologische Vorgabe durch den Betreuer soll diese API einer REST-API entsprechen.

Mittels einer REST-API können Funktionalitäten als Services umgesetzt und für verschiedene Clients jeglicher Plattformen angeboten werden. Die Verwendung einer REST-API ist eine populäre Wahl. Denn wie DuVander (2013) zeigt, sind die meisten der im Jahre 2010 bekannten Web-APIs REST basierend.

A4.2 CRUD-Operationen für relevante Entitäten

Für relevante Entitäten sollten die grundlegenden Datenbankoperationen Create, Read, Update und Delete (CRUD) für Anwenderinnen und Anwender durch den Webclient bereitgestellt werden. Relevante Entitäten sind diejenigen, welche für den Umgang mit Organisationsstrukturen notwendig sind. Im Konkreten sind dies Entitäten, die den folgenden Klassen entsprechen:

`OrgElement`, `OrgElementType`, `OrgDimension`, `Person` und `InnerSourceProject`

A4.3 Verlinkung zwischen Entitäten bearbeiten und erstellen

Durch die Verlinkung von Organisationselementen entsteht die eigentliche Organisationsstruktur. Das Bearbeiten und Erstellen von Organisationsstrukturen setzt daher voraus, dass die Verlinkungen zwischen den Organisationselementen angepasst werden kann.

Front-End-Webapplikationsframework

Eine weitere technische Vorgabe ist die Nutzung von *Angular* (Version 2.x) als Front-End-Webapplikationsframework.

4 Architektur und Design

4.1 Architektur der CMSuite

Zu Beginn soll hier auf die Architektur der bereits bestehenden Software CMSuite und dessen Komponente Patch-Flow-Crawler kurz eingegangen werden. Im Konkreten wird der Prozess der Persistierung von Objekten und der grobe Aufbau des Patch-Flow-Crawlers beschrieben.

Die CMSuite ist aus mehreren Komponenten modular aufgebaut. Das im Rahmen dieser Arbeit entwickelte Organisationsmodell wird in die Komponente Patch-Flow-Crawler integriert. Auf weitere Komponenten soll hier aber nicht eingegangen werden, weil sie im Rahmen dieser Arbeit keine zentrale Bedeutung aufweisen.

Persistierung von Objekten

Objekte der CMSuite werden in der relationalen Datenbank *PostgreSQL* gespeichert. Die Abbildung 4.1 veranschaulicht die Verbindung der verschiedenen Technologien, die zur Persistierung verwendet werden. Durch *Liquibase* wird anhand der Changelog-Datei das Datenbankschema initial erstellt und anfallende Änderungen darauf angewandt.

Soll nun ein Java-Objekt in der Datenbank gespeichert oder aus dieser gelesen werden, muss dies über ein entsprechendes Data Access Object (DAO) geschehen. Beispielsweise gibt es ein *OrgElementDao*, welches Methoden anbietet, um *OrgElement*-Objekte zu speichern und zu lesen. DAO ist ein Entwurfsmuster, das jeglichen Zugriff auf eine Datenquelle kapselt und abstrahiert. So agiert das DAO als ein Adapter zwischen den Objekten und der Datenquelle (Alur, Malks & Crupi, 2003).

Die DAOs wiederum greifen nicht direkt auf die *PostgreSQL* Datenbank zu, sondern nutzen *Hibernate ORM* zur Verbindung mit der Datenbank. *Hibernate ORM* bewerkstelligt die Umwandlung von Java-Objekten in Entitäten des relationalen

Datenbankmodells und umgekehrt. So wird eine manuelle objektrelationale Abbildung (ORM) vermieden.

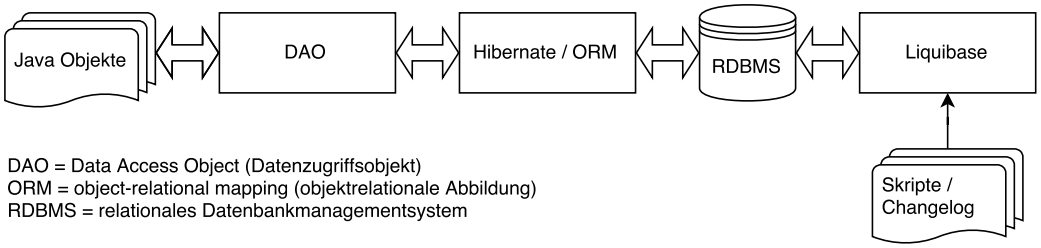

Abbildung 4.1: Persistierung von Objekten in der CMSuite

Patch-Flow-Crawler

Zentraler Bestandteil der CMSuite ist der Patch-Flow-Crawler, welcher die notwendigen Daten zur Analyse ermittelt. Hierzu können mittels unterschiedlicher Plugins, Patches aus unterschiedlichen Quellen durch einen Crawl-Vorgang erfasst werden. Ein Crawl-Vorgang wird in drei Schritten durchgeführt.

Beginnend mit dem *PreStep* werden die Quellen/Repositories für die weiteren Schritte aufgenommen.

In dem eigentlichen *Process*-Schritt werden zu jedem einzelnen Patch entsprechend die Inner-Source-Projekte, Personen und Organisationselemente in Beziehung gesetzt und in der Datenbank gespeichert.

Zum Schluss können im *PostStep* eventuell notwendige Aufräumarbeiten durchgeführt werden.

4.2 Design des Organisationsmodells

Hier soll nun ein objektorientiertes Design vorgestellt werden, welche die vorangestellten Anforderungen berücksichtigt.

4.2.1 Entwurfsmuster

Fowler (1997) beschreibt wie organisatorische Hierarchien modelliert werden können. Mit seinem *Accountability Pattern* (Verantwortlichkeits-Entwurfsmuster) schlägt er ein Modell vor, welches sich eignet Organisationshierarchien, Verträge und Dienstverhältnisse abzubilden. Eine Darstellung des *Accountability Pattern* ist in

Abbildung 4.2 zu sehen. Das *Accountability Pattern* selbst ist zusammengesetzt aus dem *Party Pattern* (Party-Entwurfsmuster) und dem *Organization Structure Pattern* (Organisationsstruktur-Entwurfsmuster). Ein Datenmodell, welches nun mehrere Hierarchien unterstützen soll, kann mittels der Verwendung von typisierten Beziehungen, wie sie im *Organization Structure Pattern* vorgeschlagen werden, realisiert werden. Die Abbildung 4.3 zeigt die Verwendung von typisierten Beziehungen in einem Modell.

Das Konzept der typisierten Beziehungen zur Modellierung von Hierarchien und Strukturen eignet sich um, die zuvor definierten Anforderungen (Kapitel 3.1) umsetzten zu können.

Als ein Nachteil des *Accountability Pattern* sei zu erwähnen, dass es zusätzliche Komplexität in das Modell bringt. Daher sollte es bei einfachen Hierarchien/Strukturen nicht unnötigerweise verwendet werden.

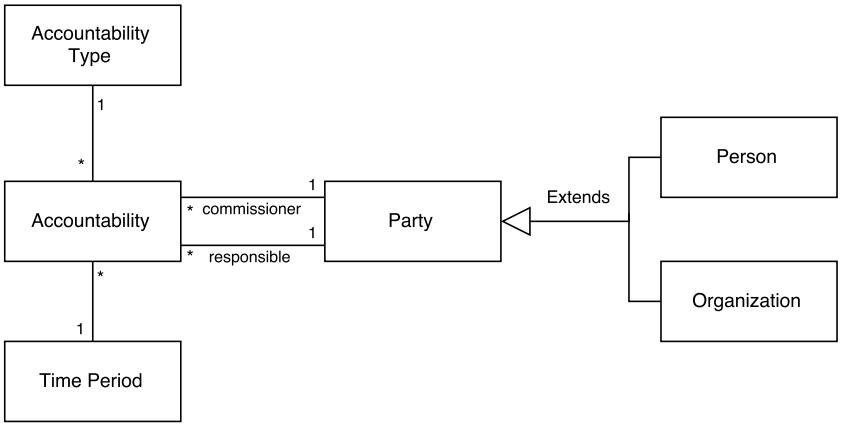

Abbildung 4.2: Accountability Pattern (Fowler, 1997)

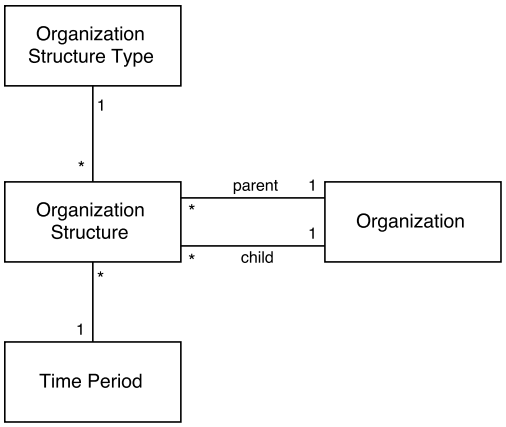

Abbildung 4.3: Typisierte Beziehungen des Organization Structure Pattern (Fowler, 1997)

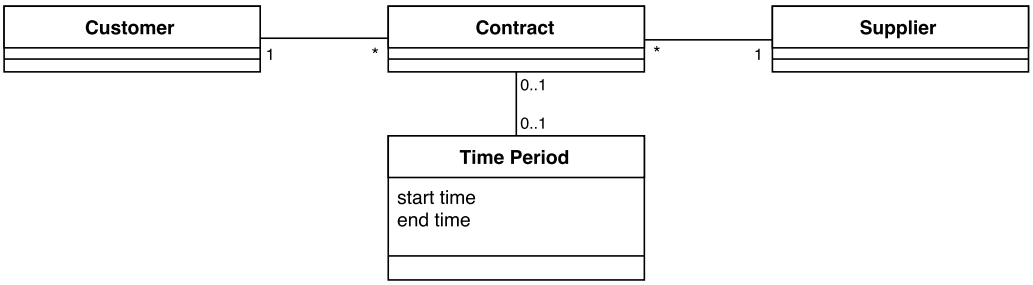

Abbildung 4.4: Temporale Assoziation zwischen zwei Vertragspartnern nach dem Temporal Association Pattern (Carlson, Estepp und Fowler, 1998)

Zeitliche Änderung von Beziehungen

Das *Organization Structure Pattern* berücksichtigt bereits, dass sich Organisationsstrukturen über die Zeit ändern können. Carlson, Estepp und Fowler (1998) gehen genauer auf die zeitabhängigen Assoziationen ein. Hier stellen sie das *Temporal Association Pattern* vor, welches zeigt wie ein Modell aussehen kann, in welchem Beziehungen zwischen Objekten sich über die Zeit ändern können.

Wie in Abbildung 4.4 zu sehen ist, beinhaltet ein `Time Period` Objekt zwei Attribute, die den Beginn und das Ende eines Zeitraumes definieren. Dieses `Time Period` Objekt wiederum ist dem Objekt, welches die Assoziation bildet zugeordnet.

Bei einer Implementierung dieses Modells für relationale Datenbanken ist ein separates `Time Period` Objekt meistens nicht sinnvoll. Hier können die Attribute des `Time Period` Objektes direkt in das Objekt, welches die Assoziation bildet, verschoben werden (Carlson et al., 1998). Bei dem Beispiel in der Abbildung 4.4 wäre dies das `Contract`-Objekt. Dadurch wird eine Tabelle in der relationalen Datenbank eingespart.

4.2.2 Resultierendes Modell

Der finale Designentwurf für das Organisationsmodell, welches Konzepte des *Organization Structure Pattern* und *Temporal Association Pattern* beinhaltet, kann der Darstellung 4.5 entnommen werden. Dieses objektorientierte Design wird den definierten Anforderungen gerecht werden. Zum besseren Verständnis sollen hier die unterschiedlichen Objekte, die in der Darstellung zu sehen sind, kurz erläutert werden.

Die Beziehung zwischen `OrgElement` und `OrgLink` spielt eine zentrale Rolle in diesem Organisationsmodell. Mithilfe dieser Beziehung wird die eigentliche Hierarchie einer Organisation abgebildet. Ein Objekt des Typs `OrgElement` entspricht

in den meisten Fällen einer Abteilung. `OrgLink` beinhaltet auch die Information über den zeitlichen Gültigkeitsbereich dessen. Mit der zusätzlichen Beziehung `OrgDimension` zu `OrgLink` können Verlinkung zwischen `OrgElement`-Objekten mit einer Dimension beschrieben werden. Dies ist erforderlich, um die verschiedenen Sichten auf Organisationsstrukturen zu kennzeichnen. Zur besseren Klassifikation von `OrgElement`-Objekten kann durch `OrgTyp` ein Type wie z.B. eine Projektgruppe oder Division definiert werden.

`PersonLink` und `InnerSourceProjectLink` unterscheiden sich von `OrgLink` dahingehend, dass sie nicht mit Dimensionen behaftet werden können. Dimensionen im Kontext von Personen und InnerSourceProjekten bilden kein sinnvolles Szenario.

Die Klassen `Repository`, `CrawlRun` und `Patch` spielen in dieser Arbeit keine Rolle und müssen daher nicht angepasst werden.

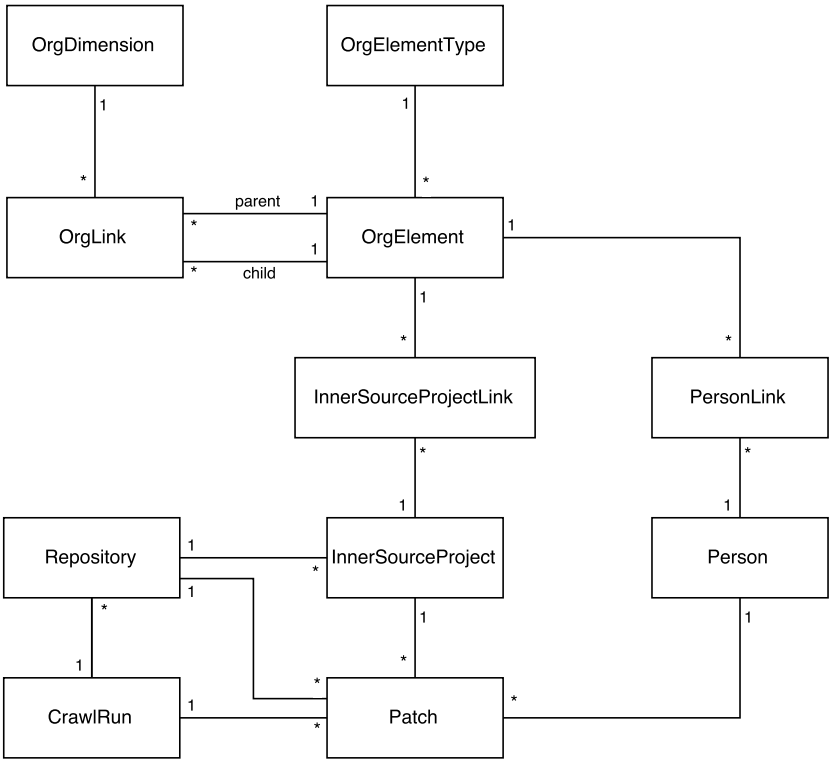

Abbildung 4.5: Finaler Designentwurf des Organisationsmodells

4.3 Design des Webclients und einer REST-API

4.3.1 REST-API

Eine REST-API die den definierten Anforderungen gerecht werden soll, ist in den folgenden Tabellen zu sehen. Die Tabelle 4.1 stellt alle CRUD-Operationen für die `OrgElement`-Entitäten sowie Operationen zur Verlinkung der Entitäten an ein `OrgElement` dar. Für die weiteren Entitäten `OrgElementTyp`, `OrgDimension`, `Person` und `InnerSourceProject` werden dann nur noch die CURD-Operationen benötigt. In der Tabelle 4.2 sind exemplarisch für diese Entitäten die REST-Endpunkte zu sehen.

Hingegen sind die REST-Endpunkte für Verlinkungen etwas eingeschränkter. Wie in der Tabelle 4.3 zu sehen ist, können beispielsweise `OrgLink`-Entitäten nicht direkt über die `OrgLink`-Ressource angelegt werden. Die Verlinkung soll explizit nur über die `OrgElement`-Ressource stattfinden. Zudem können die *Link-Entitäten nicht vollständig modifiziert werden, sondern nur der zeitliche Gültigkeitsbereich neu gesetzt werden.

Bezeichnung von Ressourcen

Für die Bezeichnung von Ressourcen sind laut Mulloy (2012) und Papapetrou (2014) folgende Best Practices zu berücksichtigen. Es sollten stets Nomen für die Bezeichnung von Ressourcen verwendet werden. Zudem sollte eine Vermischung der Plural- und Singularform vermieden werden. Hierbei wird die Pluralform oft bevorzugt. Wenn möglich sollten domänenspezifische Bezeichnungen genutzt werden.

HTTP Verben

Die Interaktion mit den Ressourcen findet über die durch HTTP bereitgestellten Verben statt. Diese wurden in der HTTP-Spezifikation[1] spezifiziert. Die Verben GET, POST, PUT und DELETE eignen sich hervorragend, um die gewünschten CRUD-Operationen abzubilden. Diese feste Anzahl an Operationen, die für alle Ressourcen gültig sind, entspricht auch dem Uniform-Interface-Constraint für REST-Architekturen.

GET = Read, Abruf von Informationen

POST = Create, Anlegen von neuen Informationen

[1] spezifiziert im RFC2616 durch Network Working Group (1999)

PUT = Update, Aktualisieren von Informationen

DELETE = Delete, Löschen von Informationen

Tabelle 4.1: REST-API der OrgElement-Ressource

OrgElement		
HTTP-Verb	**URL**	**Description**
GET	/orgelements	Get all OrgElements
GET	/orgelements/{id}	Find OrgElement by ID
POST	/orgelements	Create new OrgElement
PUT	/orgelements/{id}	Update an existing OrgElement
PUT	/orgelements/{id}/childlinks	Add OrgElement as child
PUT	/orgelements/{id}/parentlinks	Add OrgElement as parent
PUT	/orgelements/{id}/personlinks	Add Person
PUT	/orgelements/{id}/ ↩ innersourceprojectlinks	Add InnerSourceProject
DELETE	/orgelements/{id}	Delete a OrgElement

Tabelle 4.2: REST-API der Person-Ressource

Person		
HTTP-Verb	**URL**	**Description**
GET	/persons	Get all Persons
GET	/persons/{id}	Find Person by ID
POST	/persons	Create new Person
PUT	/persons/{id}	Update an existing Person
DELETE	/persons/{id}	Delete a Person

Tabelle 4.3: REST-API der OrgLink-Ressource

OrgLink		
HTTP-Verb	**URL**	**Description**
GET	/orglinks	Get all OrgLinks
GET	/orglinks/{id}	Find OrgLink by ID
PUT	/orglinks/{id}/validation	Update validation date of OrgLink
DELETE	/orglinks/{id}	Delete a OrgLink

4.3.2 Webclient

Der Webclient selbst soll alle Funktionalitäten, die durch die REST-API angeboten werden, durch seine graphische Oberfläche für Anwenderinnen und Anwender zugänglich machen. Bei der Gestaltung der Oberfläche wird auf eine simple Struktur mit einem einheitlichen Bedienungskonzept geachtet.

5 Implementierung

In diesem Kapitel wird nun auf die konkrete Implementierung eines überarbeiteten Organisationsmodells und eines Webclients eingegangen. Im vorherigen Kapitel wurde anhand der Anforderungen ein Designentwurf erstellt, an dem sich die Implementierung orientieren soll.

5.1 Implementierung des Organisationsmodells

5.1.1 Refactoring am Datenmodell

Der eigentlichen Umsetzung der Anforderungen ging ein Refactoring-Schritt voraus. In dem finalen Designentwurf des Organisationsmodells hat sich die Benennung `OrgUnit` in `OrgElement` und `Component` in `Inner-Source-Project` geändert. Diese Umbenennung erfolgte einerseits an den entsprechenden Klassen der `Commons`-Komponente, sowie am relationalen Datenbankmodell. Dadurch, dass diese Klassen oft von anderen Klassen referenziert wurden, waren Änderungen an über 100 Klassen notwendig.

Des Weiteren wurde der Primärschlüssel der `Person`-Relation geändert. Bisweilen diente das Attribut `token`, welches aus Merkmalen der Person abgeleitet ist, als natürlicher Schlüssel. Nun wurde ein neues *id* Attribut eingeführt, welches als Surrogatschlüssel die Primärschlüsselfunktion übernimmt. Dies hat zu einer Vereinheitlichung des Datenmodells geführt, da die anderen Relationen alle Surrogatschlüssel als Primärschlüssel nutzen.

Für jede Änderung am relationalen Datenmodell wurde ein ChangeSet in der DatabaseChangeLog-Datei erstellt. Liquibase wendet die Änderungen an der Datenbank an, so wie es bereits im Grundlagenkapitel 2.2 beschrieben wurde.

Dabei wurde beim Refactoring wie folgt vorgegangen. Zu Beginn wurde die komplette Refactoring-Aufgabe in mehrere, kleinere und voneinander unabhängige Arbeitspakete unterteilt. Nach der Umsetzung eines dieser Arbeitspakete enstand

stets ein stabiler Softwarestand. Diese Pakete wurden dann sukzessive abgearbeitet. Nach der Empfehlung von Fowler et al. (2012) wurde vor dem durchführen von Änderungen erst sichergestellt, dass der betroffene Code durch Tests abgedeckt wird. So können Fehler bei der Umsetzung eines Refactoringschrittes schnell bemerkt werden.

5.1.2 Verlinkung von OrgElement-Objekten durch OrgLinks

Im finalen Designentwurf spielt die Eltern-Kind-Beziehung zwischen `OrgElement`-Objekten eine zentrale Rolle. Zunächst soll die `OrgLink`-Klasse eingeführt werden, um diese Beziehungen abzubilden. Die Funktionalität bleibt zunächst dieselbe. Darauf aufbauend werden später die definierten Anforderungen umgesetzt.

Die Klasse `OrgLink` ist simpel aufgebaut und besteht zunächst nur aus den Attributen `id (Integer)`, `parent (OrgElement)` und `child (OrgElement)`. Durch die Verwendung von `OrgLink`-Objekten als Verbindungsstück zwischen `OrgElement`-Objekten mussten bis dahin simple Getter- und Setter-Methoden erweitert werden. So müssen zusätzlich `OrgLink`-Objekte erstellt oder `OrgElement`-Objekte aus ihnen extrahiert werden. Im Listing 5.1 sind diese erweiterten Methoden zu sehen.

Persistierung von OrgElement-Objekten

Die Persistierung der `OrgElement` Objekte erfolgte kaskadiert. Wenn ein `OrgElement` Objekt durch das `OrgElementDao` gespeichert wurde, so wurden durch *Hibernate* dessen Eltern- und Kinderobjekte automatisch mit abgespeichert. Auf diese Funktionalität wurde nun aber bewusst verzichtet, da nicht mehr klar ist, wann das Speichern eines Objektes durchgeführt wird. Das `OrgElementDao` wurde hierfür um folgende Methoden erweitert:

`saveIfNotExist(OrgElement)`: Speichert ein OrgElement, falls dieses noch nicht gespeichert wurde.

`saveWithAncestors(OrgElement)`: Speichert das OrgElement und rekursiv alle seiner Eltern-OrgElemente.

`saveWithDescendants(OrgElement)`: Speichert das OrgElement und rekursiv alle seiner Kind-OrgElemente.

Get Rich

Objekte die über die DAOs geladen werden, beinhalten nicht automatisch alle weiteren Objekte, die sie referenzieren. Wird zum Beispiel ein `OrgElement` über die Methode `getByToken(...)` des `OrgElementDao` geladen, führt ein Zugriff auf dessen Inner-Source-Projekte zu einer `NullPointerException`. Dieses Verhalten ist beabsichtigt, um nicht bei jedem Laden eines Objektes unnötig weitere Objekte zu initialisieren und diese dann im Speicher zu halten. In den DAOs gibt es deswegen Methoden, wie `getRichByToken(...)`, welche das Objekt mit allen benötigten referenzierten Objekten zurückgeben. So kann über die DAOs sinnvoll entschieden werden, wie Objekte geladen werden sollen. Die Get-Rich-Methoden in den DAOs waren bereits vorhanden und mussten nur an die Änderungen des Organisationsmodells angepasst werden.

Obwohl durch die neue Verlinkung von `OrgElement`-Objekten sich die bestehende Methodensignaturen nicht änderten, waren Anpassungen in vielen Klassen, die die `OrgElement`-Objekte verwendeten, notwendig. Diese Anpassungen sind auf die oben beschriebenen Punkte zur Persistierung und dem Laden von Objekten zurückzuführen.

Listing 5.1: Komplexe Getter- und Setter-Methoden der OrgElement-Klasse

```
1  public void addChild(OrgElement _child) {
2      if (_child != null) {
3          OrgLink link = OrgLink.create();
4          link.setChild(_child);
5          childLinks.add(link);
6      }
7  }
8
9  public Set<OrgElement> getChildren() {
10     Set<OrgElement> orgElements = new HashSet<OrgElement>();
11     for (OrgLink link : childLinks) {
12         orgElements.add(link.getChild());
13     }
14
15     return orgElements;
16 }
```

5.1.3 Einführung von Dimensionen

Durch die Einführung von Dimensionen kann die Unterstützung von komplexeren/mehrdimensionalen Organisationsstrukturen erfolgen. Repräsentiert werden

Dimensionen durch die Klasse `OrgDimension`. Eine Dimension besteht aus einem Namen und einer Kennzeichnung, ob diese die primäre Dimension ist.

Null-Dimensionen:

Die Verwendung von Dimensionen ist optional. Bei der Abbildung von einfachen hierarchischen Organisationsstrukturen kann daher darauf verzichtet werden. Im `OrgElement` wird hierzu der Wert des Attributs `dimension` auf `null` belassen. Haben zwei `OrgElement`-Objekte in unterschiedlichen Dimensionen ein gemeinsames übergeordnetes `OrgElement` wie z.B. die Unternehmensleitung so wird diese Beziehung in der Null-Dimension verortet.

Diese Implementierung zum Umgang mit Dimensionen wurde gewählt, um einerseits keine Dimension anlegen zu müssen, falls diese eigentlich nicht benötigt wird. Zum anderen ist es eine simple Lösung, um unterschiedliche Dimensionen, wie oben beschrieben, auf eine zu reduzieren.

Primäre Dimension

Wird das Attribut `isDefault` in einer Dimension auf den Wert `true` gesetzt, so wird diese Dimension als primär gekennzeichnet. Es darf nur eine primäre Dimension pro Organisation existieren. Daher wird beim Speichern einer primären Dimension über das `OrgDimensionDao` überprüft, ob bereits eine primäre Dimension existiert. `OrgDimensionDao` bietet zudem die Methoden `getDefaultDimension()` und `hasDefaultDimension()` an.

5.1.4 Zeitliche Änderung der Beziehung zwischen Organisationselementen

Exemplarisch soll für die drei Assoziations-Klassen (`OrgLink`, `PersonLink`, `InnerSourceProjectLink`) die `OrgLink`-Klasse in Abbildung 5.1 den Aufbau dieser Klassen darstellen. Der Aspekt der zeitlichen Änderung von Beziehungen ist nicht direkt in den Link-Klassen verankert. Hierzu wurde das Interface `TemporalLink` geschaffen, welches einheitliche Methoden zum Umgang mit dem zeitlichen Gültigkeitsbereich zur Verfügung stellt. Dabei besitzen die Methoden `isValidNow()`, `isValid(Date)` und `isOverlappingWithRange(Date, Date)` zur Validierung des Links, eine Default-Implementierung. So entstand eine Trennung der Entitäts-Klassen und deren gemeinsamen Eigenschaften.

Die Implementierung mittels Java 8 Interface mit Default-Methoden (siehe Oracle Corporation, 2017) wurde einer abstrakten Klasse bevorzugt, da hiermit einzelne

Aspekte eines Links abgebildet werden können, ohne eine Vererbungshierarchie bilden zu müssen. Sollen die Link-Klassen um eine weitere Eigenschaft erweitert werden, so kann dies durch ein neues Interface geschehen, welches jeweils eine Eigenschaft repräsentiert.

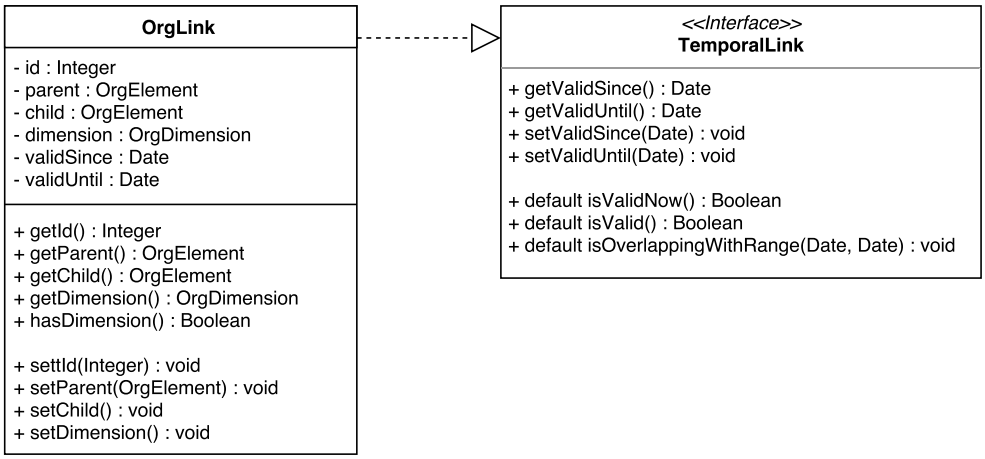

Abbildung 5.1: Klassendiagramm der OrgLink-Klasse

Gültigkeit eines Links

Durch die Methoden `getValidSince()` und `getValidUntil()` können die Zeiten abgerufen werden, die den Gültigkeitsbereich eines Links definieren. Beträgt der Wert `Null`, so besteht keine zeitliche Begrenzung in die entsprechende Richtung.

5.1.5 Inner-Source-Projekte und Personen

Die Verlinkung von Inner-Source-Projekten und Personen zu Organisationselementen erfolgt auf gleicher Weise wie die zwischen Organisationselementen untereinander. Die entsprechenden Klassen zur Verlinkung sind `PersonLink` und `InnerSourceProjectLink`. Im Gegensatz zum `OrgLink` besitzen diese Klassen aber keine Dimension.

5.1.6 Umgang mit Dimensionen und Zeiten

Durch die entstandene Möglichkeit, eine komplexere Organisationsstruktur abbilden zu können, stieg die Komplexität beim Umgang mit Organisationselementen. Einfache Getter- und Setter-Methoden genügen hier nicht mehr. Wie an den Listings 5.3 und 5.2 zu sehen ist, werden nun mehrere Methoden für den Umgang mit Dimensionen und Zeiten benötigt.

Getter-Methoden

Bei den Getter-Methoden muss genau spezifiziert werden, aus welcher Dimension und zu welcher Zeit Objekte zurückgegeben werden sollen. Zu diesen Methoden werden zusätzliche Methoden für gängige Abfragen, wie z.B. `getCurrentParent(...)` `oder getCurrentParentInWildcardDimension()`, angeboten.

Als Konvention beinhalten Methoden, die die aktuell gültigen Objekte zurückgeben, das Schlüsselwort `Current` im Methodennamen.

Add-Methoden

Exemplarisch für Add-Methoden ist im Listing 5.2 eine `addParent()` Methode der `OrgElement`-Klasse zu sehen. Hier muss nun explizit die Dimension und der Zeitraum der Gültigkeit mit übergeben werden. Vor dem eigentlichen Verlinken als Parent-Element werden die Argumente mittels Assertion-Methoden auf ihre Gültigkeit überprüft und im Falle von ungültigen Argumenten wird eine `IllegalArgumentException` geworfen. Die Assertion-Methode `assertParentLinkNotExisting(...)` überprüft, ob bereits ein Link zu dem übergebenen `OrgElement`-Objekt existiert, welches sich in der übergebenen Dimension zu dem übergebenen Gültigkeitsbereich befindet. Anschließend wird das Link-Objekt erstellt und zu dem Set `parentLinks` hinzugefügt.

Listing 5.2: Parent Add-Methoden der OrgElement-Klasse

```
1  public void addParent(OrgElement _parent, OrgDimension
       _dimension, Date _validSince, Date _validUntil) {
2      assertNotNull(_parent);
3      assertParentLinkNotExisting(_parent, _dimension, _validSince,
       _validUntil);
4
5      OrgLink link = OrgLink.create();
6      link.setParent(_parent);
7      link.setValidSince(_validSince);
8      link.setValidUntil(_validUntil);
9      link.setDimension(_dimension);
10     parentLinks.add(link);
11 }
```

Listing 5.3: Parent Getter-Methoden der OrgElement-Klasse

```
1  public Set<OrgElement> getCurrentParents() {
2      return getParents(DateUtil.now());
3  }
4
5  public Set<OrgElement> getParents(Date _date) {
6      return parentLinks.stream()
7          .filter(l -> l.isValid(_date))
8          .map(l -> l.getParent())
9          .collect(Collectors.toSet());
10 }
11
12 public OrgElement getCurrentParent(OrgDimension _orgDimension) {
13     return getParent(_orgDimension, DateUtil.now());
14 }
15
16 public OrgElement getParent(OrgDimension _orgDimension, Date
       _date) {
17     for (OrgLink link : parentLinks) {
18         boolean isSame = _orgDimension == link.getDimension();
19         boolean isEquals = link.hasDimension()
20                                 &&
       link.getDimension().equals(_orgDimension);
21
22         if (link.isValid(_date) && (isSame || isEquals)) {
23             return link.getParent();
24         }
25     }
26     return null;
27 }
28
29 public OrgElement getCurrentParentInWildcardDimension() {
30     return getParent(null, DateUtil.now());
31 }
```

5.1.7 Unit Tests

Zur Sicherstellung der Softwarequalität wird mithilfe von Tests ein Programm oder ein Teil dessen mit der Absicht ausgeführt, eventuell vorhandene Fehler zu finden. Hierzu werden die Ergebnisse der Tests mit den spezifizierten Anforderungen verglichen. Unter den verschiedenen Arten von Tests sind besonders zwei

davon für Entwicklerinnen und Entwickler interessant. Einerseits die **Unit Tests** zum Testen einzelner Programmbausteine. Andererseits **Integrationstests**, welche das fehlerfreie Zusammenwirken von Systemkomponenten überprüfen (Inden, 2012).

Die Qualität des Patch-Flow-Crawlers wird mittels Unit Tests, Integrationstests und Systemintegrationstests[1] sichergestellt. Während der Anpassung des Organisationsmodells wurden kontinuierlich Unit Tests hinzugefügt, die neue Funktionalitäten testeten. Teilweise hat eine testgetriebene Entwicklung (TDD[2]), wie sie Beck (2003) vorschlägt, stattgefunden. Dabei wurden erst die Unit Tests, die das Verhalten einer neuen Funktionalität prüfen, geschrieben. Danach wurde die eigentliche Funktionalität implementiert und so lange angepasst, bis sie alle Tests besteht.

In der Tabelle 5.1 ist die Testabdeckung der wichtigsten Klassen des Patch-Flow-Crawlers aufgelistet. Zum Vergleich ist die Testabdeckung vor der Implementierung des neuen Organisationsmodells in der Tabelle 5.2 zu sehen. Aus diesem Vergleich ist ersichtlich, dass sowohl die Anzahl der Tests als auch deren Testabdeckung erhöht wurde.

Tabelle 5.1: Testabdeckung wichtiger Klassen des Patch-Flow-Crawlers

Klasse	**Tests**	**Methoden [%]**	**Zeilen [%]**
OrgElement	68	100 %	97 %
OrgDimension	5	87 %	67 %
Person	16	95 %	89 %
InnerSourceProject	15	86 %	84 %
OrgLink	6	100 %	100 %
PersonLink	6	100 %	100 %
InnerSourceProjectLink	6	100 %	100 %
TemporalLink (Interface)	22	100 %	100 %
OrgElementDao	26	88 %	97 %

Tabelle 5.2: Testabdeckung wichtiger Klassen des Patch-Flow-Crawlers vor der Implementierung des neuen Organisationsmodells

Klasse	**Tests**	**Methoden [%]**	**Zeilen [%]**
OrgElement	11	86 %	73 %
Person	6	88 %	67 %
InnerSourceProject	4	73 %	55 %
OrgElementDao	16	71 %	65 %

[1] Smoke Tests
[2] Test-Driven Development

5.2 Implementierung des Webclients

5.2.1 REST Backend

Das REST-API Backend wurde in die Komponente Datamanager der CMSuite integriert. Java bietet hierfür mit JAX-RS[3] eine Schnittstelle, um RESTful-Web-Services umzusetzen. Die bereitgestellten Funktionen der API entsprechen dem im vorherigen Kapitel erstelltem Design und sollen daher hier nicht erneut behandelt werden.

Ablauf bei einem Request

Erreicht ein Request das REST-Backend so wird die entsprechende Klasse, die eine Ressource repräsentiert wie zum Beispiel `OrgElementResource`, aufgerufen. Von hier aus werden Service-Klassen aufgerufen, die mithilfe der DAOs die gewünschte Funktion ausführen. Zuletzt wird ein Response an den Client zurückgeschickt, welches die angeforderten Daten im JSON-Format beinhaltet[4].

API-Dokumentation

Eine Dokumentation ist ein wichtiger Bestandteil einer API. Für Entwickler sollte klar ersichtlich sein, welche Funktionalitäten die API bereitstellt und wie mit ihr umzugehen ist. Derzeit existiert kein Standard zur Dokumentation von REST-APIs, aber es sind viele Formate, Frameworks und Best Practices entstanden.

In dieser Arbeit wurde eine Dokumentation durch Dateien im Markdown-Format[5] gewählt. Ein Beispiel hierfür ist im Anhang B zu sehen. Diese Art der Dokumentation wurde aus folgenden Gründen gewählt. Einerseits müssen keine extra Frameworks und Tools in das Projekt eingebunden werden, dessen Lizenzen eventuell mit der eigenen Lizenz inkompatibel sind. Andererseits können Markdown-Dateien einfach in Plattformen wie GitHub[6] und GitLab[7] als Wiki-Seiten eingebunden werden. Als Nachteil ist zu erwähnen, dass die Dokumentation bei Änderungen händisch angepasst werden muss und nicht automatisch aus dem Code generiert werden kann.

[3] JAX-RS 2.0 wurde im JSR-339 durch Chinnici und Sandoz (2013) spezifiziert.
[4] Nur bei einem GET-Request werden Daten zurückgeliefert.
[5] vereinfachte Auszeichnungssprache
[6] https://github.com/
[7] https://about.gitlab.com/

Sollte die CMSuite in Zukunft dahingehend erweitert werden, dass die API-Dokumentation während des Buildprozesses mit erstellt wird, so sollte das Framework *Swagger*[8] in Betracht gezogen werden. Bei *Swagger* werden die zur Dokumentation benötigten Informationen als Annotationen dem Code beigefügt. Aus diesen Annotationen wird durch den Buildprozess eine Swagger-Specification generiert. *Swagger UI* kann diese generierten Dateien als Seite darstellen und ermöglicht es zudem, die API direkt zu testen.

5.2.2 Angular Frontend

Aufbau des Userinterface

Das Userinterface des Webclients stellt Seiten für jede Entität, die in der Anforderungsanalyse als relevant definiert wurden (Anforderung A4.2), bereit. Zunächst ist eine Übersicht der Entitäten, wie sie in Abbildung 5.2 zu sehen ist, dargestellt. Von hier führt der *Details*-Link zu einer detaillierten Ansicht der entsprechenden Entität, dargestellt in Abbildung 5.3. Bei Organisationselementen werden beispielsweise zu den identifizierenden Attributen (ID, Name, Token) auch Verlinkungen zu Personen, InnerSourceProjekten und weiteren Organisationselementen angezeigt. Diese Verlinkungen können invalidiert oder gelöscht werden. Im rechten oberen Bereich ist stets eine Anzeige der auf die Entität anwendbaren Funktionen zu sehen.

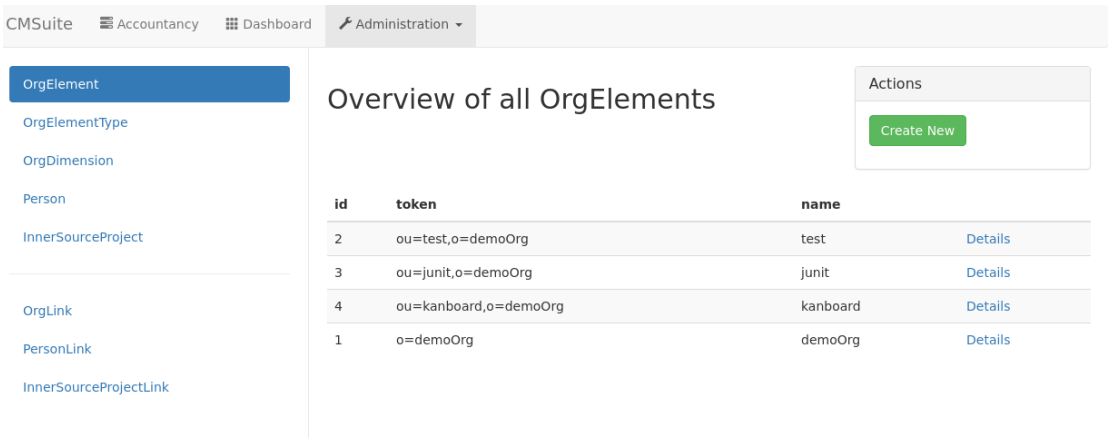

Abbildung 5.2: Screenshot Webclient: Übersicht aller OrgElement-Entitäten

[8] http://swagger.io/

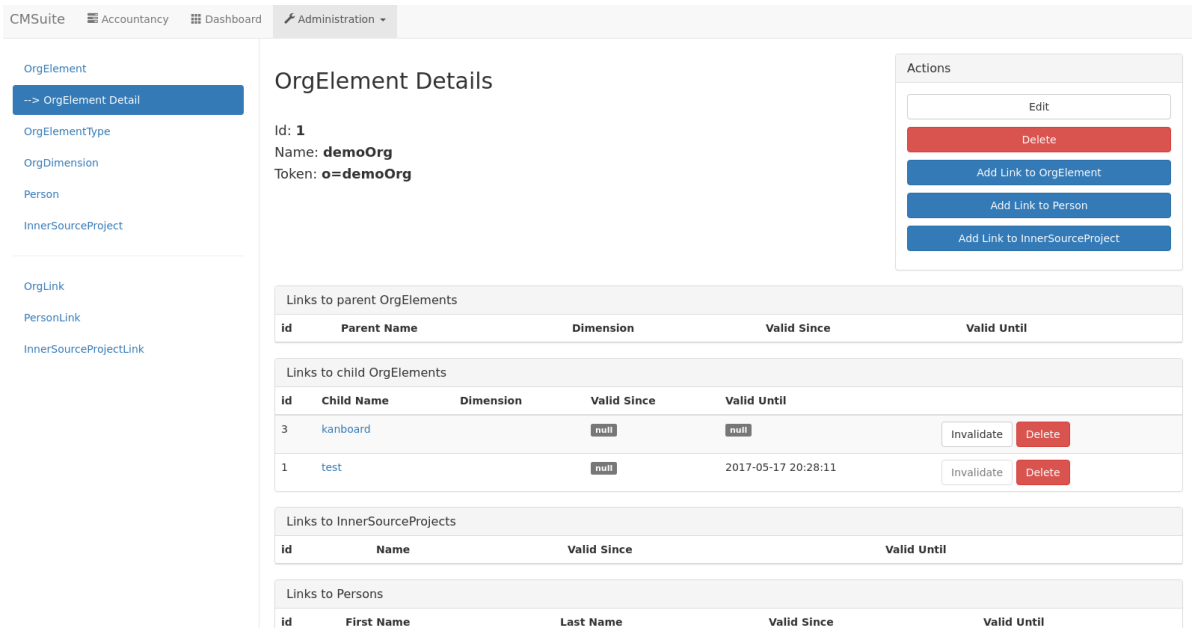

Abbildung 5.3: Screenshot Webclient: Detailansicht einer OrgElement-Entität

Services zur Kommunikation mit der REST-API

Eine Möglichkeit zur Kommunikation mit der REST-API wird an vielen Stellen des Webclients benötigt. Wiederverwendbare Funktionen können in *Angular* durch Services bereitgestellt werden. Services sind Klassen[9], die eine klar definierte Aufgabe erfüllen und die mittels Dependency Injection in Komponenten eingebunden werden können (Angular Team, 2017).

Im Webclient wurde der Service `BasicRestService` erstellt, der vier generische Methoden zur Interaktion mit der REST-API anbietet. Diese Methoden entsprechen den HTTP-Verben GET, POST, PUT und DELETE. Die auf die Ressourcen zugeschnittene Services, wie beispielsweise der `PersonService`, nutzen dann den `BasicRestService` für die Kommunikation.

[9] bei der Verwendung von TypeScript, andernsfalls sind es JavaScript-Objekte

6 Evaluation

Die Umsetzung eines neuen Organisationsmodells und eines Webclients haben sich stets an den definierten Anforderungen orientiert. Hier soll nun evaluiert werden, ob die gestellten Anforderungen eingehalten und im Zuge dessen die Zielsetzungen dieser Arbeit erreicht wurden.

6.1 Evaluation des Organisationsmodells

In der Tabelle 3.1 wurden bereits alle Anforderungen zum Organisationsmodell zusammengefasst. Im Folgenden wird geprüft, inwieweit das entwickelte Organisationsmodell den Anforderungen gerecht wird. Hierbei wird auf die Fallstudien Bezug genommen. Die involvierten Industriepartner werden aus Gründen der Anonymität, als A und B bezeichnet.

A1.1 Unterstützung für mehrdimensionale Organisationsstrukturen

Mehrdimensionale Organisationsstrukturen können flexibel durch die Verlinkung verschiedener Organisationselemente mittels der `OrgLink`-Objekte abgebildet werden. Beispielhaft ist in Abbildung 6.1 ein Organisationselement zu sehen, welches zwei übergeordnete Organisationselemente in unterschiedlichen Dimensionen besitzt. Team A ist hier sowohl der Projektgruppe X als auch der Entwicklung unterstellt. Durch die Dimensionen ist ersichtlich, dass Team A hierbei der Entwicklung disziplinarisch unterstellt ist und aus einer Projektsicht der Projektgruppe X zugeordnet ist. Eine Beschränkung der maximal zu nutzenden Dimension gibt es hierbei nicht. Die Abbildung von Matrixorganisationen und anderen komplexeren Organisationsstrukturen stellt daher kein Problem dar. Somit wird diese Anforderung erfüllt.

Beide Industriepartner benötigen jeweils zwei Dimensionen zur Modellierung ihrer Organisation. A hat hierbei eine disziplinarische und eine Feature-orientierte

Dimension. Bei B hingegen existiert eine disziplinarische und eine Team-orientierte Dimension.

A1.2 Markieren einer Dimension als primäre Dimension

Mit dem `isDefault` Attribut der `OrgDimension`-Klasse lässt sich leicht bestimmen, ob es sich bei einer Dimension um die Primäre handelt. Zudem kann über das `OrgDimensioDao` die primäre Dimension direkt zurückgegeben werden. Soll nur eine Dimension betrachtet werden, so kann den Verlinkungen gefolgt werden, die sich auf die primäre Dimension beziehen. Schlussfolgernd kann festgehalten werden, dass diese Anforderung erfüllt wurde.

Besonders Industriepartner A legt Wert darauf, dass eine Dimension primär betrachtet werden kann.

A1.3 Zeitliche Änderung der Organisationsstruktur

Ein bis dahin vernachlässigter Aspekt ist die Änderungen der Organisationsstruktur über die Zeit. Die Evaluation dieser Anforderung soll anhand der Abbildung 6.2 stattfinden. Es wir ein Organisationselement (Team A) gezeigt, welches über die Zeit unterschiedlichen Organisationselementen zugeordnet ist. Hierbei ist die Verlinkung zur Projektgruppe X bis zum Datum "2017-06-02 12:00:00" gültig. Hingegen beginnt die Verlinkung zur Projektgruppe Y erst ab diesem Datum. So wurde der Wechsel der Projektgruppe nachvollziehbar abgebildet. Eine Änderung der Organisationsstruktur kann mit der gewählten Implementierung auch in die Zukunft geplant werden. Zudem kann die gültige Organisationshierarchie zu einem beliebigen Zeitpunkt ermittelt werden. Auch hier wurde die Anforderung erfüllt. Gleiches gilt für die Anforderungen **A2.2** und **A3.2**, welche auf gleiche Weise umgesetzt wurden und deshalb nicht erneut beschrieben werden.

A2.1 n:m Beziehung von Personen zu Organisationselementen

Ebenso wurde die Anforderung, dass Personen mehreren Organisationselementen zugeordnet werden können, umgesetzt. Ein Rollenmodell für Personen war nicht Teil der Anforderung, könnte aber eine sinnvolle Erweiterung des Organisationsmodells sein. So wäre klar, in welcher Funktion eine Person agiert.

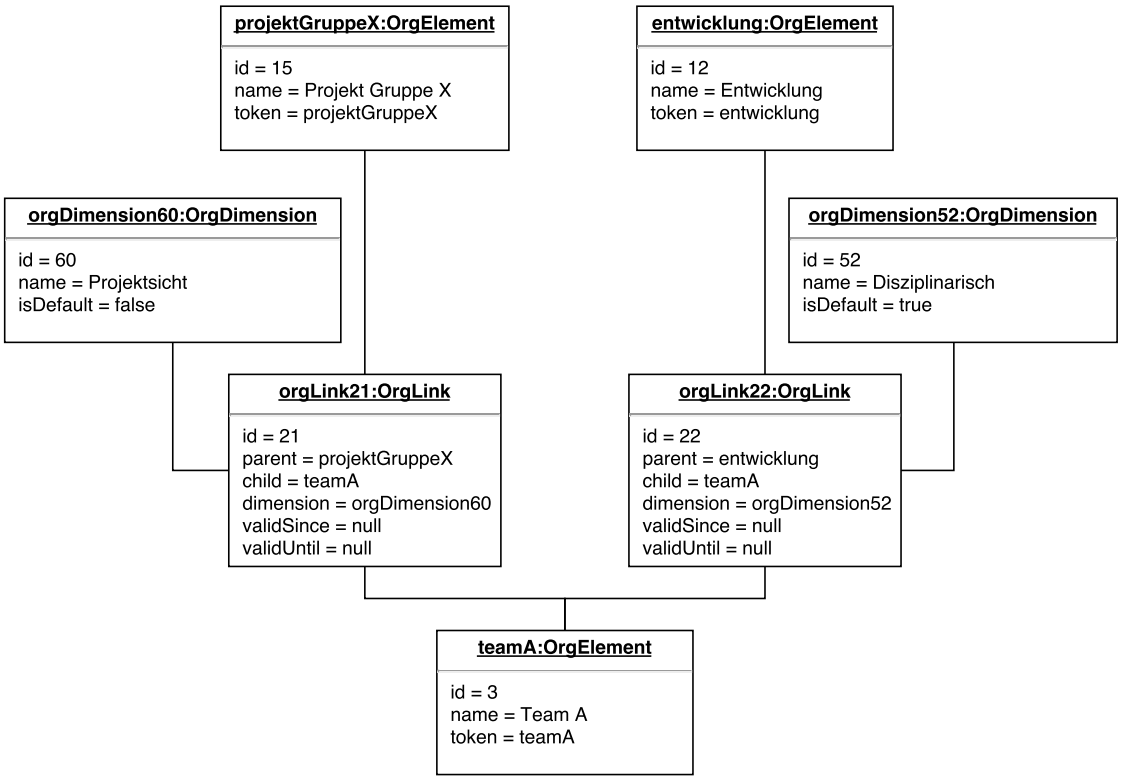

Abbildung 6.1: Objektdiagramm eines Organisationselements, welches zwei übergeordneten Organisationselementen in jeweils unterschiedlichen Dimensionen zugeordnet ist

A3.1 n:m Beziehung von Inner-Source-Projekten zu Organisationselementen

Auch die Inner-Source-Projekte können jetzt mehreren Organisationselementen zugewiesen werden. Dies ist beim Industriepartner B der Fall. Zudem kann es sein, dass ein Inner-Source-Projekt nirgendwo zugeordnet ist und somit der kompletten Organisation angehört.

Integration in die CMSuite

Das entwickelte Organisationsmodell ist vollständig in die CMSuite integriert und hat das bisherige Modell ersetzt. Somit wurde nicht eine Lösung für den Patch-Flow-Crawler geschaffen, sondern eine für die gesamte CMSuite.

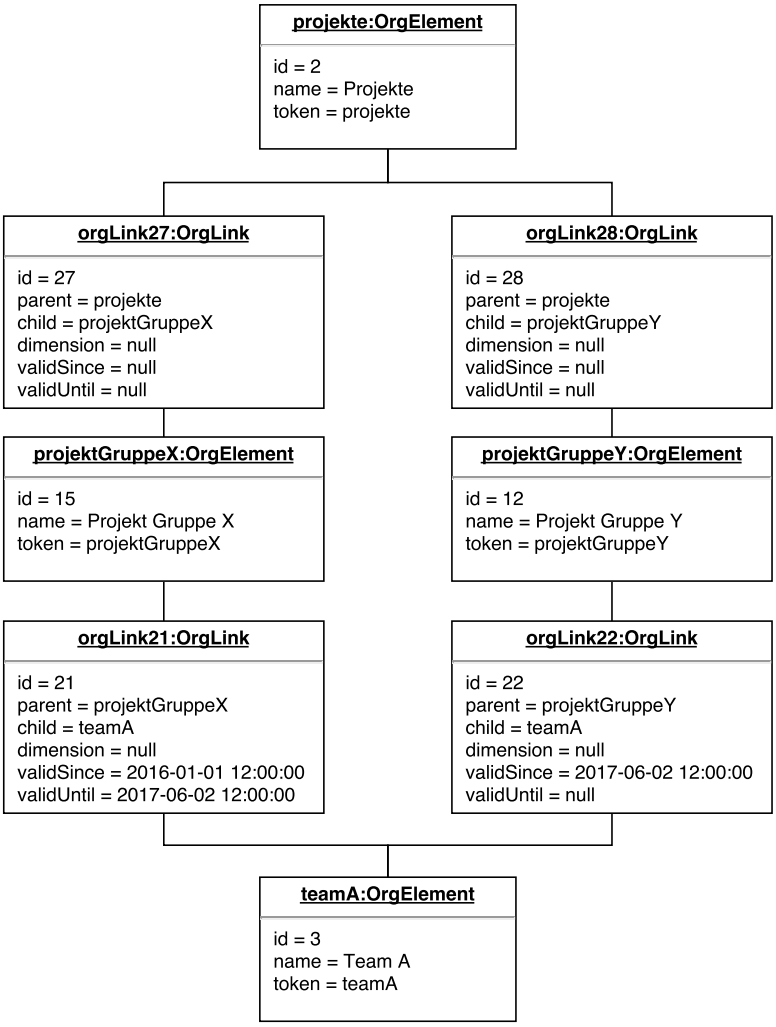

Abbildung 6.2: Objektdiagramm eines Organisationselements, welches über die Zeit unterschiedlichen Organisationselementen zugeordnet ist

6.2 Evaluation des Webclients

Die Umsetzung der Anforderungen soll auch hier anhand der Tabelle 3.2 evaluiert werden.

A4.1 Bereitstellen einer REST-API für den Webclient

Die implementierte REST-API stellt alle nötigen Funktionalitäten zur Verfügung, welche der Webclient für die Anforderungen A4.2 und A4.3 benötigt. Es soll nun auf den Unterschied zwischen REST und RESTful eingegangen werden.

Wie im Kapitel 2.3 bereits beschrieben wurde, kann ein Webservice als RESTful bezeichnet werden, wenn dieser alle von Fielding (2000) definierten Constraints berücksichtigt. Eine RESTful-API wurde im Rahmen dieser Arbeit aber nicht gefordert. So besitzt die umgesetzte Implementierung keinen Caching-Mechanismus, wie es durch das Cache-Constraint gefordert wird. Zusätzlich muss für eine RESTful-API das *Hypermedia As The Engine Of Application State* (HATEOAS) Prinzip umgesetzt werden. Wie aus dieser Bezeichnung hervorgeht, nimmt hierbei Hypermedia als treibende Kraft Einfluss auf den Zustand der Anwendung. Auf technischer Ebene wird dies durch Links in der Antwort eines Webservices umgesetzt, die zu weiteren Ressourcen führen. Fielding (2008) beklagt, dass der Hypermedia-Gedanke oft vernachlässigt wird.

Über diese Arbeit hinaus könnte die API um die fehlenden Aspekte erweitert werden, sodass diese schlussendlich einer RESTful-API entspricht.

A4.2 CRUD-Operationen für relevante Entitäten

Des Weiteren ist es gelungen, für jede Entität die CRUD-Operationen durch den Webclient zugänglich zu machen. Beim Erstellen und Bearbeiten von Entitäten werden Forms mit einer Validierung verwendet. So müssen Pflichtfelder ausgefüllt sein, bevor die Aktion durchgeführt werden kann. Zudem ist es nicht möglich, das Attribut `ID` der Entitäten zu manipulieren. Bei einer Auflistung von Entitäten, werden Einträge die unvollständig sind, farblich hervorgehoben, was eine Erleichterung bei der Fehlersuche darstellt.

A4.3 Verlinkung zwischen Entitäten bearbeiten und erstellen

Auch die Verlinkungen zwischen den Entitäten können durch den Webclient erstellt, gelöscht und invalidiert werden. Wie in Abbildung 5.3 dargestellt wurde,

sind die Verlinkungen in der Detailansicht für Entitäten ersichtlich. Zudem ist eine extra Auflistung der Link-Entitäten gegeben.

Ein Szenario, in dem Änderungen an einer bestehenden Organisationsstruktur durchgeführt werden oder kleinere Organisationsstrukturen erstellt werden, kann durch den Webclient abgedeckt werden. Soll hingegen eine große Organisationsstruktur durch den Webclient erstellt werden, wäre eine erweiterte Übersicht der Verlinkungen wünschenswert. Eine übliche Baumstruktur als Mittel zur Anzeige ist hierbei nicht ausreichend. Für mehrdimensionale Strukturen müsste ein Konzept erarbeitet werden, wie diese sinnvoll präsentiert werden können.

7 Fazit

In dieser Arbeit wurde gezeigt, wie durch das entwickelte Organisationsmodell komplexere Organisationsstrukturen abgebildet werden können. Das von der CMSuite bis dahin verwendete Organisationsmodell war nicht in der Lage, die Strukturen komplexerer Organisationen korrekt abzubilden. Ein weiterer Aspekt, der bis dahin vernachlässigt wurde, ist die Dynamik in Organisationen und somit in ihrer Struktur. Weder Änderungen in der Organisationshierarchie noch die Zuordnung von Personen und Inner-Source-Projekten konnte durch das Organisationsmodell berücksichtigt werden. Um ein neues Modell, das der Unternehmenspraxis gerecht wird, zu entwickeln, wurden zuerst zwei Fallstudien analysiert. Bei dieser Analyse wurden geeignete Anforderungen definieren.

Das entwickelte Organisationsmodell bietet nun eine flexible Gestaltung von komplexeren Organisationsstrukturen. Hierzu gehören mehrdimensionale Strukturen, die keiner einfachen Hierarchie entsprechen. Zudem ist es möglich, die genannten dynamischen Aspekte nachvollziehbar abzubilden. Beides wurde durch den Einsatz von typisierten Beziehungen ermöglicht. Features wie Dimensionen müssen jedoch nicht genutzt und berücksichtigt werden, falls nur einfache Hierarchien abgebildet werden sollen.

Durch die Integration des Organisationsmodells in die CMSuite und so in dessen Patch-Flow-Crawler wurde eine bessere Grundlage zur Analyse des Patch-Flows geschaffen. Denn durch eine exaktere Abbildung der tatsächlichen Organisationsstrukturen werden Fehlinterpretationen vermieden, wodurch Qualität der Metriken weiter verbessert wird.

Zusätzlich wurde mit dem Webclient ein Tool geschaffen, dass es Anwenderinnen und Anwendern erlaubt, in abgebildete Organisationsstrukturen einzugreifen. So können etwaige unvollständig importierte Datensätze manuell korrigiert oder komplett neue Organisationsstrukturen angelegt werden.

Anhang A Beispiel eines Liquibase ChangeSet

```
<!-- Add InnerSourceProjectLink relation -->
<changeSet id="17" author="abauer">
<dropForeignKeyConstraint
    baseTableName="innersourceproject"
    constraintName="fk_isp_hostorgelement"/>
<dropColumn
    tableName="innersourceproject"
    columnName="host_id" />

<createTable tableName="innersourceprojectlink">
    <column name="id" autoIncrement="true"
        type="integer">
        <constraints primaryKey="true"
            nullable="false" />
    </column>
    <column name="orgelement_id"
        type="integer" />
    <column name="innersourceproject_id"
        type="integer" />
    <column name="valid_since"
        type="TIMESTAMP WITH TIME ZONE" />
    <column name="valid_until"
        type="TIMESTAMP WITH TIME ZONE" />
</createTable>

<addForeignKeyConstraint
    baseColumnNames="orgelement_id"
    baseTableName="innersourceprojectlink"
    constraintName="fk_isplink_orgelement"
    onDelete="RESTRICT" onUpdate="CASCADE"
    referencedColumnNames="id"
    referencedTableName="orgelement" />
<addForeignKeyConstraint
    baseColumnNames="innersourceproject_id"
    baseTableName="innersourceprojectlink"
    constraintName="fk_isplink_innersourceproject"
    onDelete="RESTRICT" onUpdate="CASCADE"
    referencedColumnNames="id"
    referencedTableName="innersourceproject" />
</changeSet>
```

Anhang B Beispiel für die Dokumentation einer REST-Ressource

Get all root OrgElements

Get all OrgElements that have no parent. Returns root OrgElements and its children.

- **URL**

  /orgelements/roots

- **Method:**

  `GET`

- **URL Params**

  Optional:

  `levels=[alphanumeric]`

  Levels of children to get. Default value is 4.

- **Success Response:**

  - **Code:** 200 OK

    Content:

```
[{
"id": 1,
"token": "o\u003ddemoOrg",
"name": "demoOrg",
"type": null,
"children": [
  {
    "id": 4,
    "token": "ou\u003dkanboard,o\u003ddemoOrg",
    "name": "kanboard",
    "type": null,
    "children": [...],
    ...
  }
],
"innerSourceProjects": [],
"members": [],
"parentLinks": [],
"childLinks": [...],
"memberLinks": [],
"innerSourceProjectLinks": []
}]
```

Anhang C Inhalt des Datenträgers

Auf dem beigefügten Datenträger befinden sich alle Source-Code-Dateien, sowie die digitale Version dieser Arbeit als PDF- und LaTeX-Datei.

Quellcode: Der gesamte Quellcode zu dieser Arbeit befindet sich in der komprimierten Datei *cmsuite.zip*.

API-Dokumentation: In dem Ordner *api-dokumentation* ist die Dokumentation der REST-API zu finden.

Arbeit als PDF-Datei: Die PDF-Version dieser Arbeit wurde im Root-Verzeichnis abgelegt.

Arbeit als LaTeX-Datei: Der Order *thesis_tex* beinhaltet alle LaTeX-Dateien.

Abbildungsverzeichnis

3.1 Organisationsstrukturen im Vergleich 10

4.1 Persistierung von Objekten in der CMSuite 14
4.2 Accountability Pattern (Fowler, 1997) 15
4.3 Typisierte Beziehungen des Organization Structure Pattern (Fowler, 1997) 15
4.4 Temporale Assoziation zwischen zwei Vertragspartnern nach dem Temporal Association Pattern (Carlson, Estepp und Fowler, 1998) 16
4.5 Finaler Designentwurf des Organisationsmodells 17

5.1 Klassendiagramm der OrgLink-Klasse 25
5.2 Screenshot Webclient: Übersicht aller OrgElement-Entitäten . . . 30
5.3 Screenshot Webclient: Detailansicht einer OrgElement-Entität . . 31

6.1 Objektdiagramm eines Organisationselements, welches zwei übergeordneten Organisationselementen in jeweils unterschiedlichen Dimensionen zugeordnet ist 34
6.2 Objektdiagramm eines Organisationselements, welches über die Zeit unterschiedlichen Organisationselementen zugeordnet ist . . . 35

Tabellenverzeichnis

3.1 Anforderungen an das Organisationsmodell 8
3.2 Anforderungen an den Webclient 11

4.1 REST-API der OrgElement-Ressource 19
4.2 REST-API der Person-Ressource 19
4.3 REST-API der OrgLink-Ressource 20

5.1 Testabdeckung wichtiger Klassen des Patch-Flow-Crawlers 28
5.2 Testabdeckung wichtiger Klassen des Patch-Flow-Crawlers vor der Implementierung des neuen Organisationsmodells 28

Literaturverzeichnis

Alur, D., Malks, D. & Crupi, J. (2003). *Core J2EE Patterns - Best Practices and Design Strategies* (2. Aufl.). New Jersey: Prentice Hall Professional.

Angular Team. (2017). Angular Services. Zugriff 25. Mai 2017 unter https://v2.angular.io/docs/ts/latest/guide/architecture.html#!#services

Bauer, C., King, G. & Gregory, G. (2015). *Java Persistence with Hibernate* - (2nd edition.). Birmingham: Manning Publications Company.

Beck, K. (2003). *Test-driven Development - By Example* (01. Aufl.). Boston: Addison-Wesley Professional.

Capraro, M. & Riehle, D. (2016). Inner Source Definition, Benefits, and Challenges. *ACM Comput. Surv. 49*(4), 67:1–67:36. doi:10.1145/2856821

Carlson, A., Estepp, S. & Fowler, M. (1998). Temporal Patterns.

Chinnici, R. & Sandoz, P. (2013). JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services. Zugriff 31. Mai 2017 unter https://jcp.org/en/jsr/detail?id=339

DuVander, A. (2013). Growth In Web APIs Since 2005. Zugriff 21. Mai 2017 unter http://http://www.programmableweb.com/api-research

Fayol, H. (1929). *Allgemeine und industrielle Verwaltung* -. Muenchen: R. Oldenbourg.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation, University of California, Irvine,). Zugriff unter https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fielding, R. T. (2008). REST APIs must be hypertext-driven. Zugriff 21. Juni 2017 unter http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Fowler, M. (1997). *Analysis Patterns - Reusable Object Models* (1. Aufl.). Boston: Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. & Gamma, E. (2012). *Refactoring - Improving the Design of Existing Code* (1. Aufl.). Amsterdam: Addison-Wesley.

Hibernate ORM. (2017). Hibernate ORM. Zugriff 20. April 2017 unter http://hibernate.org/orm/what-is-an-orm/

Inden, M. (2012). *Der Weg zum Java-Profi : Konzepte und Techniken für die professionelle Java-Entwicklung* (2. aktualisierte und erweiterte Auflage). Heidelberg: dpunkt.verlag.

Internet Engineering Task Force (IETF). (2014). The JavaScript Object Notation (JSON) Data Interchange Format. Zugriff 21. Mai 2017 unter https://tools.ietf.org/html/rfc7159

Konda, M. (2014). *Just Hibernate - A Lightweight Introduction to the Hibernate Framework*. Sebastopol: Ö'Reilly Media, Inc.".

Mulloy, B. (2012). Web API Design - Crafting Interfaces that Developers Love. Zugriff 21. Juni 2017 unter http://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf

Network Working Group. (1999). Hypertext Transfer Protocol – HTTP/1.1. Zugriff 25. Mai 2017 unter https://www.ietf.org/rfc/rfc2616.txt

Network Working Group. (2005). Uniform Resource Identifier (URI): Generic Syntax. Zugriff 21. Mai 2017 unter http://tools.ietf.org/html/rfc3986#section-1.1.3

Oracle Corporation. (2017). Java 8 Interface Default Methods. Zugriff 2. Mai 2017 unter https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Papapetrou, P. (2014). Rest API Best(?) Practices Reloaded. Zugriff 21. Juni 2017 unter https://dzone.com/articles/rest-api-best-practices

Scherm, E. (2007). *Organisation - Theorie, Gestaltung, Wandel ; mit Aufgaben und Fallstudien*. Muenchen: Oldenbourg Verlag.

Thommen, J.-P. & Achleitner, A.-K. (2012). *Allgemeine Betriebswirtschaftslehre - Umfassende Einfuehrung aus managementorientierter Sicht* (7. Aufl.). Berlin Heidelberg New York: Springer-Verlag.

Tilkov, S. (2011). *REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien* (2. akt. und erw. Auflage). Heidelberg: dpunkt.

Voxland, N. (2011). Liquibase. *Methods and Tools*, 40–45.

Voxland, N. (2017). Liquibase. Zugriff 20. April 2017 unter http://www.liquibase.org

Webber, J., Parastatidis, S. & Robinson, I. (2010). *REST in Practice - Hypermedia and Systems Architecture*. Sebastopol: Ö'Reilly Media, Inc.".

Wohlwender, A. (2014). *Analyse der Wissenskommunikation in einer Matrixorganisation* - (2015. Aufl.). Berlin Heidelberg New York: Springer-Verlag.