ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Entwicklung eines Cloud-basierten Servers für kollaboratives Checklisten-Management

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Jule Anna Ziegler
jule.ziegler@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Nicolas Mundbrod

2013

Fassung 11. November 2013

Satz: PDF-LaTeX 2ε

Kurzfassung

Die Beschleunigung und Flexibilität von Geschäftsprozessen erhält in jedem Unternehmensbereich eine immer stärker werdende Popularität. Die bisher mangelnde Prozessunterstützung von wissensintensiver Arbeit schwächt die Produktivität und Qualität von zukunftsweisenden Geschäftslösungen. Die Nutzung von Checklisten durch ein verteilt verfügbares System hat das Potential, die Strukturierung und Koordination von wissensintensiver Arbeit zu beschleunigen. Wiederkehrende ähnliche Aufgabenstellungen können durch die Verwendung von vordefinierten Vorlagen nachhaltig verbessert werden und das Wissensmanagement um die operative Ebene erweitern.
Diese Arbeit umschließt die Konzeption eines Cloud-basierten Servers zur Unterstützung wissensintensiver Arbeit durch den Einsatz von Checklisten. Im Rahmen des Projektes proCollab an der Universität Ulm findet die initiale Umsetzung in Form eines Prototypen statt. Dabei werden entscheidende Architekturparadigmen und Technologien verwendet, auf dessen Entwicklung der zu implementierende Prototyp basiert.

Abstract

The acceleration and flexibility of business processes received a predominant popularity in each business division and will become even more important in the future. Due to the lack of appropriate business process execution of knowledge-intensive work, the efficiency and quality of innovative business solutions is still weakened. Using checklists with a distributed system supports the potential to structure and coordinate knowledge-intensive processes. By applying predefined templates, similar tasks can be subject to sustainable improvement and extend knowledge-intensive management at the operational level.
This thesis describes the design of a cloud-based server in order to support knowledge-intensive work managed by the use of checklists. The initial implementation takes place on the basis of a prototype within the project proCollab at the University of Ulm. Based on the development of the prototype, essential architecture paradigms and technologies are introduced.

Danksagung

Als Erstes möchte ich meiner Mutter, meinen beiden Geschwistern Meike und Moritz sowie meinen Freunden zutiefst für ihre Unterstützung und Motivation danken. Ihr seid mir stets zur Seite gestanden und habt mich während meinem Studium bestärkt. Ich danke euch für eure unerschöpfliche Geduld und Bemühungen.

Ein ganz besonderer Dank gilt Prof. Dr. Reichert und Nicolas Mundbrod für Ihre Betreuung. Durch Ihre Expertise haben Sie mir stets dazu verholfen meine Recherchen zu hinterfragen und kritisch zu betrachten. In dieser Hinsicht möchte ich mich zusätzlich bei dem gesamten Team des Instituts für Datenbanken und Informationssysteme bedanken, das mich ebenfalls unterstützt hat.

Letztlich möchte ich meinen Teammitgliedern im Rahmen des Projekts *proCollab* für ihre Zusammenarbeit danken. Durch unser gemeinsames Wirken konnten wir die Forschungsarbeit meistern.

Inhaltsverzeichnis

1

Einleitung

"The most important contribution management needs to make in the 21st century is similiary to increase the productivity of knowledge work and knowledge workers."

Peter F. Drucker (1909 - 2005)

Ökonom, Professor und renommierter Schriftsteller

1.1 Motivation

Basierend auf der gestiegenen Produktivität in den Bereichen der Manufaktur im 20. Jahrhundert ist heutzutage ein immer stärker werdender Trend in der Automatisierung wissensintensiver Geschäftsprozesse zu erkennen [Dru07]. Um dem durch die Globalisierung heranwachsenden Marktdruck standhalten zu können, sind Unternehmen zunehmend daran interessiert, die Effizienz und Wirksamkeit ihrer Geschäftsprozesse zu optimieren [BHMS05]. Während ein Großteil der standardisierten, routinierten Arbeit bereits von Systemen übernommen oder in andere Länder ausgelagert wurde,

stehen gegenwärtig Länder mit hohem Entwicklungsgrad vor der Aufgabe, wissensintensive, unternehmensübergreifende Prozesse zu beschleunigen. Der Wandel hin zu einer wissensstarken Gesellschaft zeigt, dass Wissensarbeiter (im Allgemeinen: Wissenschaftler, Ingenieure und Entwickler) eine immer stärkere Rolle in unserer Wirtschaft spielen. Tagtäglich entwickeln sie anhand ihrer Expertise und Erfahrung vielversprechende Technologien und Geschäftslösungen. Dabei spielen hauptsächlich hochdynamische Faktoren wie Kosten, Zeit und verfügbare Ressourcen eine entscheidende Rolle für sie, die zu Beginn nicht endgültig planbar sind [MKR12].
Prozessmanagementsysteme beschränken sich typischerweise auf vordefinierte, planbare Prozesse und können so die täglichen Anforderungen eines wissensintensiven Arbeiters nicht adäquat umsetzen. Um einen Mehrwert für Unternehmen zu schaffen, sollen Geschäftsprozesse so entworfen werden, dass sie wirtschaftlich effizient sind, sich an die vorhandenen Ressourcen bestmöglich anpassen und die Durchlaufzeiten und -kosten reduzieren [BHMS05].
Der Mangel an wissensstarker Prozessunterstützung und das Nicht-Vorhandensein genauer Prozessinformationen bremst die Produktivität und verringert dadurch die Wettbewerbsfähigkeit. Die Chance bereits entstandene Prozesslösungen dynamisch bereitzustellen, diese zu automatisieren und als Vorlage für ähnliche Problemstellungen wiederzuverwenden, ist das Sprungbrett für Prozessautomation und -beschleunigung. Daher stellt der Wunsch die Arbeit am Arbeitsplatz zu erleichtern und damit zu beschleunigen eine der größten Herausforderungen des 21. Jahrhunderts dar. Sie kann aber durch den wachsenden Fortschritt der Informationstechnologie Realität werden [S$^+$10]. Die ursprüngliche prozessorientierte Vorgehensweise, Geschäftsabläufe mit Hilfe vordefinierter Kontrollflüsse und Aktivitäten zu beherrschen, eignet sich nicht für wissensintensive Prozesse und gilt daher als Antrieb für neue vielversprechende Ansätze. Es ist essentiell, dass neue innovative Herangehensweisen entwickelt werden, um Personen, die in wissensintensive Geschäftsprozesse involviert sind, besser zu unterstützen. Daher gilt es, ein prozessorientiertes Wissensmanagement aufzubauen und zu etablieren, um damit auf unvorhersehbare, dynamische Prozesse flexibel reagieren zu können [Gup12].

1.2 Problemstellung

Neben den routinierten, vordefinierten Prozessen, deren Logik bereits vor der Ausführung bekannt ist, laufen innerhalb eines Unternehmens viele Wissensprozesse ab, die hochdynamisch und flexibel sind und dadurch schwer strukturierbar sind. Das Ziel zur Erlangung maximaler Prozessunterstützung ist die fortschreitende Umsetzbarkeit besonders unstrukturierter Prozesse [GPSW03].
Die Entstehung wissensintensiver Prozesse lässt sich aus verschiedenen Einflussfaktoren ableiten (Abbildung 1.1).

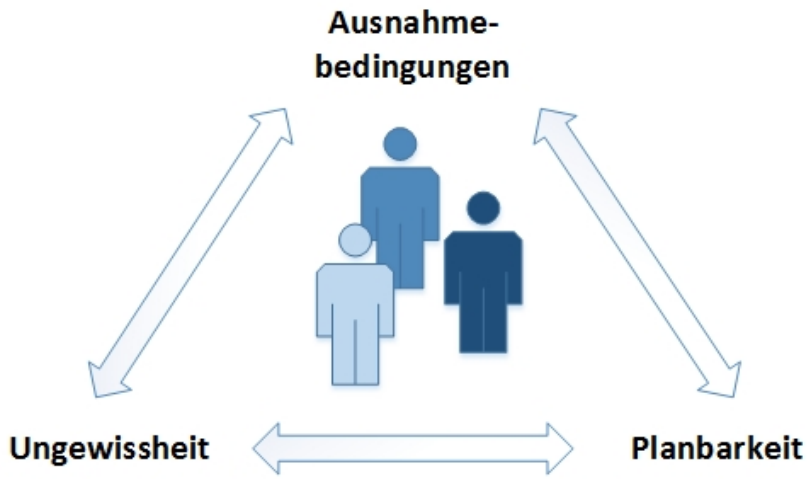


Abbildung 1.1: Abhängigkeitsverhältnis der Faktoren zur Entstehung wissensintensiver Arbeit

Neben dem hohen Grad an Dynamik zeichnen sich diese besonders dadurch aus, dass Entscheidungen erst zur Laufzeit von den *Prozessbeteiligten* getroffen werden. Entsprechende Prozesse können infolgedessen nicht komplett formal vorspezifiziert werden, sondern erwarten stattdessen eine dynamisch anpassbare Prozesskoordination.
Diese Art von Prozess ist in nahezu jedem Gebiet eines Unternehmens aufzufinden und wird hauptsächlich von Unternehmensbereichen in der Entwicklung, Forschung, Design

und dem Kundenservice repräsentiert. Besonders letzteres zeichnet sich durch die hohe Individualität eines Kunden aus, auf dessen Wünsche persönlich einzugehen ist. Individuelle, flexible Prozesse zu analysieren, zu modellieren und schließlich zu optimieren, konfrontiert jeden Sektor der Wirtschaft und wird langfristig Ziel jedes Unternehmens sein. [BHMS05]
Der größte Faktor mit dem umzugehen ist, ist die Bewältigung der *Ungewissheit* [S+10]. Komplexe Prozessabläufe sind von einer Menge nicht-handhabbaren Einflussfaktoren umgeben, auf die meist erst im gegenwärtigen Zustand reagiert werden kann. Wissensintensive Prozesse sind aus diesem Grund äußert schlecht planbar. Das Abhängigkeitsverhältnis verschiedenster Faktoren erfordert eine langsame Entwicklung des Prozesses und eine stetige Evaluierung und Anpassung der Aufgaben an sich ändernde Gegebenheiten. Die Zielfindung ist daher mit einer repetitiven Koordination und Neuausrichtung der Faktoren durchzuführen. Dies erfordert eine anpassungsfähige Planung sowie eine ständige Prüfung der momentanen Anforderungen, um die einzelnen Prozessschritte aufeinander abstimmen zu können.
Wissensintensive Prozesse zeichnen sich zusätzlich durch einen hohen Grad an *Ausnahmebedingungen* aus. Diese Variabilität ist durch die fehlende Modellierung nicht für andere Personen sichtbar. Die Generalisierung subjektiver Aktivitäten, Prozesse und Ergebnisse sowie deren Abbildung auf eine - für jeden verfügbare - Prozessbeschreibung stellt daher einen entscheidenden Meilenstein zum Wissensaustausch dar. Wissensintensive Prozesse unterstützen keine klassischen Techniken zur Geschäftsprozessoptimierung und sind daher auf einen größtmöglichen Innovationsgrad sowie eine Intensität an Kreativität angewiesen. [GPSW03]

1.3 Zielsetzung

Ein Bestandteil jedes Entwicklungsprojektes ist das *Qualitätsmanagement*. Damit dieses adäquat umgesetzt werden kann, werden regelmäßig Checklisten in den Entwicklungsprozess integriert. So können Checklisten aus einem Satz von Fragen bestehen, um die zu entwickelnden Personen an Aspekte hinsichtlich der Funktionalität, Zuverlässigkeit und Benutzerfreundlichkeit zu erinnern. Checklisten beschreiben keinen vordefinierten

Prozessablauf, da die umzusetzenden Anforderungen wissensintensiv und hochdynamisch sind [RW12]. Checklisten legen Anforderungen und Einschränkungen, sogenannte *Constraints*, zu Beginn fest.
Ausgangspunkt ist die klassische Checklistenverwaltung, ein manueller Prozess, der Checklisten in Papierform bereitstellt. Änderungen innerhalb einer solchen Checkliste lassen sich schwer handhaben, wodurch ein langwieriger Verwaltungsprozess entsteht. Soll ein Dokument aktualisiert und hinsichtlich seiner Anforderungen angepasst werden, hat dies ein erneutes Erstellen und Verteilen der Checklisten zur Folge. Checklisten aktuell zu halten ist belastend und es ist nicht selten, dass eines der Dokumente verloren geht und dadurch für Teammitglieder nicht mehr zugänglich wird. Personen stehen vor dem Problem, dass sie nur noch teilweise komplettierte oder veraltete Checklisten besitzen. Aus diesen Gründen hat sich herausgestellt, dass die Verwaltung von Checklisten in Papierform unzureichend ist und infolgedessen elektronisch für jeden zugänglich gemacht werden sollte.
In dem zu konzipierenden und zu implementierenden *proCollab-Prototypen* stellen Checklisten den zentralen Ansatz zur Bewältigung dynamischer, wissensintensiver Prozesse dar. Dabei leitet sich die Bezeichnung des Prototypen als „*proCollab-Prototyp*“ aus der Entstehung einer Forschungsarbeit an der Universität Ulm ab (siehe dazu Kapitel 2.2).
Es soll ein Verwaltungssystem für beliebige Entwicklungsprojekte entwickelt werden, die eine Menge von Checklisten beinhalten und in Bezug auf konkrete Anwendungsfälle entwickelt werden sollen. Dazu soll der proCollab-Prototyp ein durchgängiges Lebenszyklus-Management bereitstellen. Des Weiteren sollen gewisse Sicherheitsaspekte und Zugriffskontrollen standardisiert werden, um so Aspekte des Datenschutzes zu berücksichtigen.
Der Aufbau dieser Arbeit lehnt sich direkt an die Konzipierung und Implementierung des Servers an. Ziel dieser Systemarchitektur ist der modulare Aufbau des proCollab-Prototypen, um die beteiligten Komponenten unabhängig voneinander entwickeln zu können [Gei13, Köl13, Rei13, Thi13]. Die Erreichbarkeit des Servers soll anhand wohldefinierter Schnittstellen zwischen den einzelnen Modulen ermöglicht werden.

1.4 Aufbau der Arbeit

Die Arbeit ist in zwei Hauptabschnitte unterteilt. Zu Beginn werden die Grundlagen beschrieben, auf denen der zu implementierende proCollab-Prototyp basiert. Darauf aufbauend wird auf die konkrete Konzeption und Entwicklung des Systems eingegangen. Die Realisierung des Systems ist dabei modular aufgebaut und beschreibt die einzelnen angewandten Komponenten.

Kapitel 1 - *Einleitung*

Dieses Kapitel hat bereits aufgezeigt, warum aus heutiger Sicht, die Automatisierung wissensintensiver Prozesse unumgehbar geworden ist und was der direkten Umsetzung gegenwärtig im Weg steht.

Kapitel 2 - *Grundlagen*

Dieses Kapitel definiert den Begriff des *wissensintensiven Prozesses* und legt dar, warum Checklisten als Unterstützungsmöglichkeit dieser Prozesse vermutlich geeignet sind. Dabei wird auf konkrete Anwendungsfälle in der Praxis eingegangen, die bereits Checklisten nutzen, um die Komplexität des jeweiligen Themengebietes besser handhaben zu können. Des Weiteren wird die Terminologie, der im System angewandten Begriffe, definiert.

Kapitel 3 - *Anforderungen*

Dieses Kapitel beschreibt Anwendungsfälle aus Sicht eines Systemnutzers und fokussiert dabei, die als sehr hoch priorisierten Anforderungen an den proCollab-Prototypen. Funktionale und nicht-funktionale Anforderungen legen das Verhalten sowie dessen Güte fest.

Kapitel 4 - *Konzept*

Dieses Kapitel charakterisiert die Architektur und Funktionalität des Systems. Dabei wird zum einen auf die einzelnen Schichten innerhalb des Systems eingegangen, zum anderen auf dass, was die Architektur leisten muss, damit das System als verteilte Anwendung im Internet genutzt werden kann. Dieses Kapitel zeigt erstmals konkrete Umsetzungsmodelle des zu implementierenden proCollab-Prototypen auf.

Kapitel 5 - *Implementierung*

Dieses Kapitel umfasst die verwendeten Technologien und Standards auf denen das System basiert. Dabei wird auf das Prinzip der RESTful Webservices sowie der Umsetzung der einzelnen Komponenten im Detail eingegangen.

Kapitel 6 - *Fazit und Ausblick*

Dieses Kapitel blickt auf wesentliche Inhalte, die während der proCollab-Kollaboration entstanden sind, zurück und gibt einen Ausblick auf mögliche zukünftige Forschungsaspekte des Projekts. Es werden zentrale Fragestellungen und Ideen, die während der Entwicklung entstanden sind, aufgegriffen und dargestellt.

2

Grundlagen

Das Kapitel 2 - Grundlagen - veranschaulicht grundlegende Aspekte, die zum Verständnis des proCollab-Prototypen dienen und erläutert die Möglichkeit, wissensintensive Prozesse anhand von Checklisten zu unterstützen. Des Weiteren wird auf das Projekt *proCollab* an der Universität Ulm aufmerksam gemacht. Im Anschluss werden konkrete Anwendungsfälle in der Praxis veranschaulicht.

2.1 Wissensintensive Prozesse

Der Begriff des *Wissens* beschreibt einen komplexen Prozessablauf, der bisher einzig und allein im Kopf eines wissensintensiven Arbeiters vorhanden ist [RW12].
Ein *wissensintensiver Prozess* stellt einen Geschäftsprozess dar, dessen Ausführung hochdynamisch ist. Im Gegensatz zu vorspezifizierten, sich wiederholenden Prozessen, hängt die tatsächliche Ausführung und dessen Reihenfolge von den partizipierenden Prozessteilnehmern ab (siehe Kapitel 1.2).

In den ab Teilabschnitt 2.3 beschriebenen Anwendungsfällen, wird hervorgehoben, dass konkrete Anwendungsfälle nicht im Detail planbar sind und nur anhand Anforderungen, die durchzuführen sind, ausgerichtet werden können.
Wissensintensive Prozesse sind infolgedessen nicht vollständig vordefinierbar, da sie nicht durch eine Menge gegebener Aktivitäten abgebildet werden können [MKR12].
Durch die schlechte Planbarkeit und der Existenz von Ausnahmebedingungen werden wissensintensive Prozesse schwer strukturierbar, woraus die vorliegende Wissensintensität abgeleitet werden kann [GPSW03].
Diese Prozesse können durch Checklisten unterstützt werden, indem ein Checkfragen-Modell aufgebaut wird. Die Verwendung von Checklisten erlaubt eine partielle Spezifikation des Prozesses, zwängt den Prozess aber nicht in ein vorkonfiguriertes Modell [MKR12]. Der Prozess bleibt dennoch flexibel und anpassbar.

2.2 Projekt proCollab

Bei dieser Arbeit handelt es sich um eine kollaborative Forschungsarbeit unter der Leitung von Prof. Dr. Manfred Reichert und Nicolas Mundbrod.
Das Projekt umschließt eine Kooperation von fünf Studierenden der Universität Ulm im Rahmen ihrer Bachelorarbeit.
Aufsetzend auf die Konzeption des Systems ist gemeinsam ein initialer proCollab-Prototyp zu entwickeln. Dieser stellt grundlegende Funktionalität zur Verwaltung von Checklisten bereit, welche nach Abschluss der jeweiligen Arbeiten erweitert und komplettiert wird.
Das Ziel des proCollab-Projekts ist die modulare Konzeption und die damit einhergehende Implementierung eines initialen proCollab-Prototypen. Wohldefinierte Schnittstellen sollen für eine Kopplung der Komponenten sorgen. Dabei ist jedes Teammitglied für eine spezifische Komponente in der Entwicklung zuständig.
Im Kontext dieser Bachelorarbeit wird die Konzeption und Entwicklung der proCollab-Server-Komponente (*pCSK*) dargelegt. Diese stellt grundlegende Anwendungslogik für den proCollab-Prototypen bereit. Die pCSK spielt eine vermittelnde Rolle und ist dabei

zwischen der Datenhaltung und den einzelnen Applikationen angesiedelt. Dies erfordert stetige Iterations- und Evaluationsstrategien, um die Interaktion und Kommunikation zwischen der darüber- und darunterliegenden Schicht zu koordinieren und um eine reibungslose Übertragung der Daten von und zu den Applikationen zu gewährleisten.
Die Präsentationsschicht, welche in drei Applikationen unterteilt ist, wird dabei jeweils von einem Teammitglied übernommen. Jeder spezialisiert sich dabei auf eine Darstellungsart. Es steht eine Applikation für das Smartphone [Gei13], für das Tablet [Köl13] sowie für den Browser [Thi13] zur Verfügung. Zur Verwirklichung der Persistierung der Daten ist ein weiterer Kommilitone integriert [Rei13].

Um den proCollab-Prototypen erweitern zu können und dadurch auch für international Studierende zugänglich zu machen, orientiert sich die Konzipierung und die Implementierung des proCollab-Prototypen in englischer Sprache.
Im Umfang dieser Arbeit sind daher vermehrt englische Begrifflichkeiten aufzufinden, welche aber durch ein entsprechend deutsches Synonym charakterisiert werden.

2.3 Anwendungsfälle

Anwendungsfälle für Checklisten zur Unterstützung von wissensintensiven Prozessen lassen sich in einer Vielzahl von Branchen finden. Größtenteils werden Checklisten als Erinnerungsfunktion für die partizipierenden Prozessteilnehmer eingesetzt, um Prozessschritte, die anwendungsspezifisch durchgeführt werden müssen, festzuhalten. Außerdem ist die Qualitätssicherung dabei ein weiterer wichtiger Aspekt.

2.3.1 Anwendungsfall 1: Automobilindustrie

Bereich

Ein Kraftwagen- und Nutzfahrzeughersteller nutzt zur Entwicklung der mechatronischen Komponenten im Fahrzeug ein definiertes Vorgehensmodell [Mun13]. Die Mechatronik beschreibt dabei ein interdisziplinäres Verfahren, zur Kopplung von Bausteinen zwischen Software, Elektronik und Mechanik. Um Phasen und Prozessschritte adäquat

durchlaufen zu können, werden Listen zur Abarbeitung definiert. Diese Checklisten definieren die Ausführungsreihenfolge und die abzuarbeitenden Schritte, zur Sicherstellung vorgegebener Anforderungen im Qualitätsmanagement.

Problem

Der mechatronische Entwicklungsprozess in der Automobilindustrie beruht auf einem methodischen Phasenmodell (*V-Modell*), das die Entwicklung mechatronischer Systeme unterstützen und die zur Vielzahl miteinander interagierenden Komponenten verknüpfen muss [Mun13]. Insbesondere parallel ablaufende Entwicklungsprozesse, wie sie in der Automobilindustrie wiederzufinden sind, basieren auf einer interdisziplinären Methodik und zeichnen sich durch eine besonders hohe Komplexität aus, wodurch die Wissensintensität gesteigert wird. Um mechanische Komponenten mit elektronischen Geräten über Softwaremodule miteinander koppeln zu können, müssen Echtzeitdaten aufgenommen werden [HGP07]. Diese digital zu verarbeiten und letztendlich in Funktionalität umzusetzen, stellt eine Herausforderung für Ingenieure dar und kann nur anhand ihrer Expertise durchgeführt werden. Die einzelnen Verarbeitungsschritte laufen auf mehreren Ebenen ab, welche je nach Größe und Ausbaustufe der mechatronischen Komponente, oft schwer handhabbar werden.

Anwendung

Der Nutzfahrzeughersteller Daimler nutzt bereits eine Checklisten-Verwaltung basierend auf *IBM Rational Doors*, das Checklisten hierarchisch organisiert und zusammengehörige Phasen des V-Modells gruppieren kann. Dabei handelt es sich um ein Anforderungsanalyse-Werkzeug, das von einer verantwortlichen Person der Qualitätssicherung verwaltet und gepflegt wird. Die einzelnen Projektphasen werden dabei konfiguriert, als Excel-Datei exportiert und zentral für alle beteiligten Personen verfügbar gemacht. Nach Abschluss jeder Phase werden die beschriebenen Schritte wiederholt, um damit die Qualität mechatronischer Komponenten zu verbessern.

2.3.2 Anwendungsfall 2: Medizin

Bereich

Medizinische Prozesse, wie die Visite eines Patienten oder die Durchführung einer Operation, bedürfen einer überlegten Vorgehensweise. Die Abarbeitung von vorgegebenen Checklisten tragen zur Patientensicherheit bei und basieren auf Standards der Weltgesundheitsorganisation (WHO) sowie der Deutschen Gesellschaft für Chirurgie [Uni13]. Studien belegen, dass die Nutzung von Checklisten bei der Planung, Durchführung und Nachbehandlung von Operationen den Grad der Komplikationen nachweislich senkt [HWB+09].

Problem

Prozesse im medizinischen Bereich stellen für die beteiligten Ärzte eine besondere Herausforderung dar. Das Zusammenspiel zwischen der individuellen Betreuung eines Patienten und den zusätzlich vorgeschriebenen Maßnahmen, die bei jeder Behandlung eingehalten werden müssen, fördern die Komplexität des Prozessablaufs. Des Weiteren zeichnen sich medizinische Prozesse besonders durch das Auftreten individueller Krankheitsbilder als wissensintensiv aus und können daher zu Beginn nicht eindeutig festgelegt werden. Die zusätzlichen Aspekte, wie zum Beispiel Patientensicherheit und Sterilität, erhöhen die Problemstellung.

Bevor beispielsweise ein Patient operiert werden kann, ist eine Menge erforderlicher Schritte notwendig [Uni13]. Der Patient wird nach einer Reihe von Voruntersuchungen in eine Klinik überwiesen, gleichzeitig werden bei der Aufnahme des Patienten relevante Daten von der Klinikverwaltung erhoben. Ärzte ordnen anhand den Voruntersuchungen weitere Untersuchungen an und führen Gespräche mit dem Patienten. Es wird eine Blutuntersuchung durchgeführt, deren Daten an das Kliniklabor übermittelt und dessen Ergebnisse vom zuständigen Arzt ausgewertet werden müssen. Krankenschwestern verfolgen währenddessen den Gesamtzustand des Patienten und sammeln Informationen, um den Patienten auf die Narkose vorbereiten zu können. Zum Zeitpunkt der Operation müssen alle erforderlichen Daten ausgewertet worden sein und auf Abruf verfügbar sein. Während der Operation müssen die von der WHO spezifizierten Vorschriften zur Hygiene und Sterilität zu jeder Zeit gewährleistet sein.

„Zwischenfälle sind weniger auf mangelndes Fachwissen oder das Fehlhandeln und Versagen von Einzelnen, sondern in erster Linie auf Probleme beim Umsetzen des Wissens unter den Bedingungen der Versorgungsrealität und -komplexität sowie auf Defizite der Kommunikation und Teamkoordination zurückzuführen."

Bauer, Hartwig: Patientensicherheit,
In: Berlin Medical vom 01.10, S.8

Die notwendigen Schritte, die bis zur Durchführung der Operation abgearbeitet werden müssen, werden meist von mehreren Personen durchgeführt. Dies führt zu einer interdisziplinären Problematik, welche nur durch die Koordination der Arbeitsabläufe gelöst werden kann [Bau10].

Anwendung

Unter anderem nutzt die chirurgische Klinik in Ulm standardisierte Checklisten, um Prozesse in der Klinik zu optimieren und die Qualität zu erhöhen [Uni13]. Die Klinik in Ulm verwendet dabei die von der WHO empfohlene Checkliste und hat sie für ihren individuellen Gebrauch entsprechend angepasst.
Die Checkliste der WHO wurde durch das Projekt *Save Surgery Saves Lives* initiiert und findet derzeit weltweit Anwendung. Die Checkliste durchläuft drei vordefinierte Phasen. Jede Phase kann dabei ein potentielles Risiko für den Patienten darstellen, weshalb es essentiell ist, die Abarbeitung genau einzuhalten.
Atul Gawande, einer der Initiatoren der WHO-Checkliste, schreibt: „*Could a checklist be our soap for surgical care - simple, cheap, effective and transmissible?*“ [Gaw11]
Er beschreibt eine Checkliste „*Cleared for Takeoff*“, die Krankenschwestern und Ärzte vor einer Operation durchlaufen, um die wichtigsten Fakten zu klären:
Handelt es sich um den richten Patienten der an der zutreffenden Stelle operiert werden muss? Fühlt sich der Patient bereit und kann narkotisiert werden?
Die folgende Abbildung 2.1 fasst dabei relevante Aspekte einer Operation zusammen.

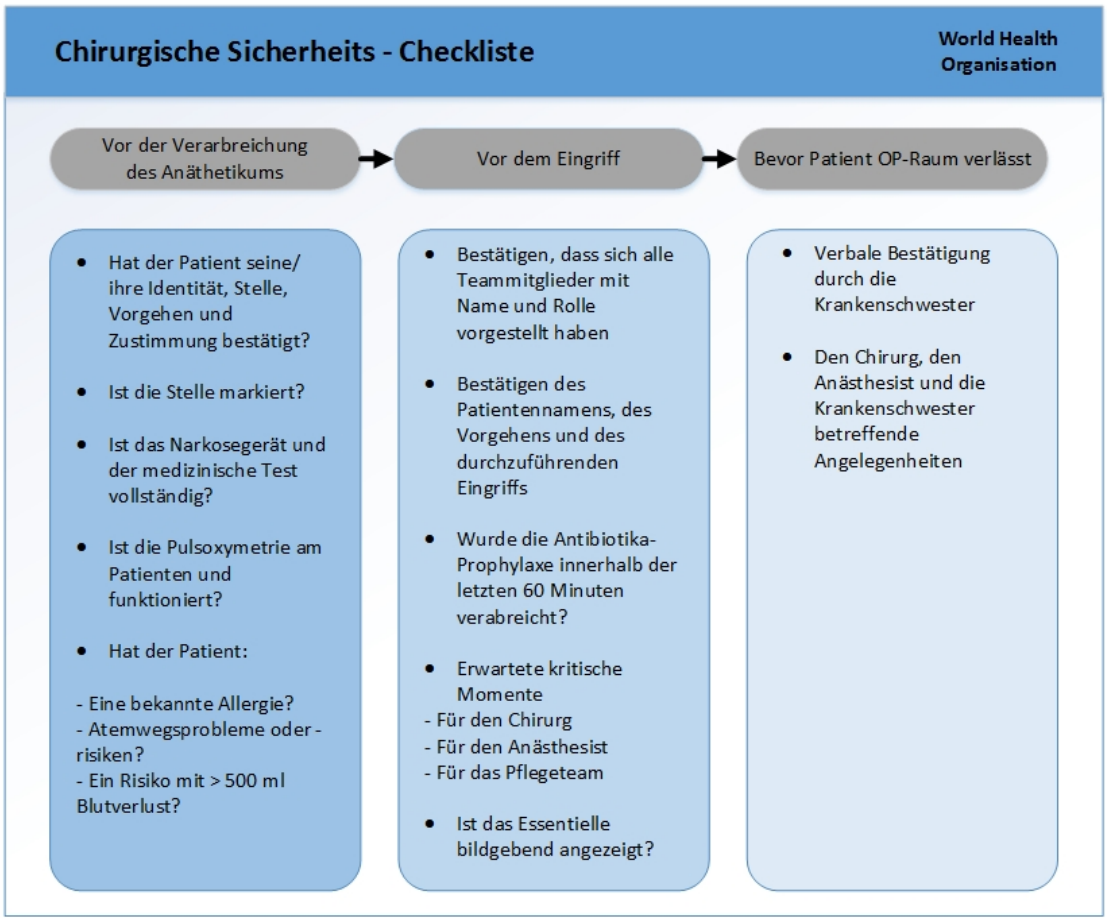


Abbildung 2.1: Die ins Deutsche übersetzte Checkliste der Weltgesundheitsorganisation (WHO) [Wor09]

Durch die Einführung simpler Fragen, die eine Operation nicht starten lassen, bevor diese verifiziert wurden, können Kommunikations- und Komplikationsprobleme während einer Operation verhindert werden. Dies trägt zur Steigerung der Patientensicherheit bei.

2.3.3 Anwendungsfall 3: Luftfahrt

Bereich

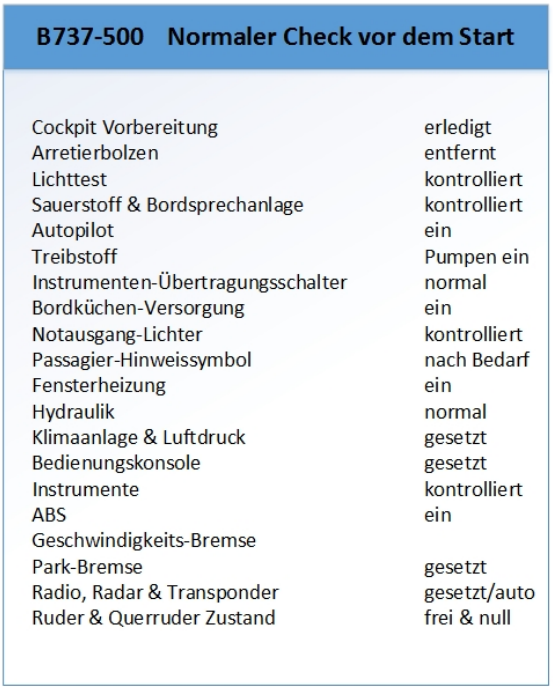
Die Luftfahrt verwendet seit langer Zeit Checklisten als Kontrollinstrument zur Einhaltung von vorgegebenen Sicherheitsstandards [Bau10]. Die stetige Weiterentwicklung der Bordtechnik hat dazu geführt, dass bereits zwei Personen alle Aufgaben übernehmen können. Dies bedarf jedoch trotzdem einer genauen Aufgaben- und Verantwortlichkeitsverteilung. Besonders technisch-intensive Prozesse wie der Start eines Flugzeuges oder dessen Landung laufen nach klar definierten Vorgehensweisen ab. Checklisten in der Luftfahrt müssen streng eingehalten werden, um die Risiken eines Absturzes zu minimieren und die Passagiersicherheit stets zu gewährleisten.

Problem

Aufgrund des hohen Sicherheitsrisikos und Gefahrenpotentials im Bereich der Luftfahrt reicht es nicht aus, sich einzig auf den Mensch oder die Technik zu verlassen. Die Vielzahl der im Cockpit vorkommenden Instrumente und deren unterschiedliche Ausgabevarianten (im Allgemeinen: optisch, akustisch, haptisch) führen zu einer sehr hohen Komplexität der Steuerung eines Flugzeugs. Einen weiteren Einflussfaktor stellen die psychologischen Aspekte des Piloten dar. Individuelle Schwächen wie Vergesslichkeit dürfen ebenfalls nicht außer Betracht gelassen werden.
Viele Aufgaben eines Piloten stellen wissensintensive Prozesse dar, da oftmals situationsabhängig gehandelt werden muss und dabei die Erfahrung ausschlaggebend ist. Das Prinzip der *Redundanz* ist dabei ein wichtiges Leitwort zur Gewährleistung von Sicherheit [Erk09].

Anwendung

Nachdem eine Verkehrsmaschine gelandet ist, muss diese für den nächsten Abflug vorbereitet werden. Checklisten in der Luftfahrt stellen sicher, dass alle relevanten Aspekte berücksichtigt werden und die Einstellungen der Bordinstrumente korrekt sind. Ein typischer Anwendungsfall ist die *Vorflugkontrolle* [Cer13], die die folgende Abbildung 2.2 exemplarisch darstellt.



B737-500 Normaler Check vor dem Start	
Cockpit Vorbereitung	erledigt
Arretierbolzen	entfernt
Lichttest	kontrolliert
Sauerstoff & Bordsprechanlage	kontrolliert
Autopilot	ein
Treibstoff	Pumpen ein
Instrumenten-Übertragungsschalter	normal
Bordküchen-Versorgung	ein
Notausgang-Lichter	kontrolliert
Passagier-Hinweissymbol	nach Bedarf
Fensterheizung	ein
Hydraulik	normal
Klimaanlage & Luftdruck	gesetzt
Bedienungskonsole	gesetzt
Instrumente	kontrolliert
ABS	ein
Geschwindigkeits-Bremse	
Park-Bremse	gesetzt
Radio, Radar & Transponder	gesetzt/auto
Ruder & Querruder Zustand	frei & null

Abbildung 2.2: Die ins Deutsche übersetzte Checkliste des Flugzeuges B737-500 [Erk09]

Anhand den vorgegebenen Punkten der Checkliste wird der technische Zustand eines Flugzeuges vor jedem Start überprüft. Dabei handelt es sich zum einen um einen Kontrollrundgang, zum anderen um einen Startcheck durch den Piloten. In der Luftfahrt existieren Checklisten zur Überprüfung der Komponenten und zur Einhaltung von Maßnahmen bei Zwischenfällen. Checklisten für Zwischenfälle schreiben Prozessschritte zur Ausführung vor, die den Piloten unterstützen, im Notfall richtig zu handeln.

2.4 Terminologie

Der zu implementierende proCollab-Prototyp basiert auf einer Menge von spezifischen Begrifflichkeiten, die im Verlauf dieser Arbeit wiederholt erscheinen. Die folgenden Definitionen grenzen die Begriffe eindeutig ein, zeigen analoge Ausdrucksweisen auf und konkretisieren den jeweiligen Anwendungskontext.

2.4.1 Organisatorischer Rahmen

Der *organisatorische Rahmen* (*OR*) repräsentiert den kontextuellen Bezug von Checklisten (siehe Kapitel 2.4.2) und ihren Nutzern. Generell kann ein OR mit einem *Projekt* oder einem *Fall* gleichgesetzt werden.
Ein OR wird zu Beginn definiert und leistet verbindliche Vorgaben für die Ablauforganisation. Ein Entwicklungsprozess kann nicht ohne einen OR stattfinden, da er relevante Angaben bezüglich der Aufgabenbeschreibung liefert. Jeder OR ist mit einem Namen versehen und beinhaltet Ressourcen, die für den OR benötigt werden. Zu jedem OR existiert eine verantwortliche Person, die Prozessbeteiligte zu dem OR hinzufügen kann. Zusätzlich verwaltet der Verantwortliche Termindaten. Er gibt zum einen an, wann der OR startet und zum anderen, bis wann ein OR fertiggestellt sein muss. Ein OR kann außerdem untergeordnete OR umschließen und demzufolge hierarchisch geschachtelt werden.

Ein *organisatorischer Rahmentyp* (*ORT*) ist eine Vorlage eines OR. Ein ORT enthält bereits einen vordefinierten Satz an Attributen. ORTs sind insbesondere hinsichtlich der Wiederverwendbarkeit sinnvoll. Oftmals entstehen ähnliche ORs, welche dann ohne erheblichen Mehraufwand anhand von existierenden ORTs reproduziert werden können. ORTs können von jedem Benutzer eingesehen und als Vorlage verwendet werden, um eine neue organisatorische Rahmeninstanz zu erstellen.

Eine *organisatorische Rahmeninstanz* (*ORI*) ist ein spezieller OR, der sukzessive mit projektspezifischen Daten gefüllt wird. ORIs können entweder unabhängig von einem

ORT erstellt oder anhand eines existierenden ORTs abgeleitet werden. ORIs besitzen die selben Attribute wie ein ORT. Ein ORI ist dabei ein konkreter, einmaliger OR, welcher nur von beteiligten Person eingesehen werden darf.

2.4.2 Checkliste

Jede *Checkliste* (*CL*) ist in einen OR integriert und fasst Checkfragen (siehe Kapitel 2.4.3) zusammen, welche festlegen, was getan werden muss, um eine CL erfolgreich abschließen zu können. CLs besitzen einen Zustand, welcher den Fortschritt des jeweiligen Prozesses dokumentiert (siehe auch Kapitel 4.1.2). Eine CL ist ein Anforderungskatalog, der die Anforderungen festlegt und diese verwaltet. Die Konzipierung der Spezifikation sollte allgemein gehalten werden, damit die Formulierung den Entwickler nicht einschränkt.
CLs können beliebige Funktionsbereiche in einem OR abbilden. Sie können beispielsweise in der Softwareentwicklung mehrere Entwicklungsphasen wie Anforderungsanalyse, Entwurf, Implementierung, Qualitätssicherung und Integration darstellen. Andererseits können sie sich auf Entwicklungsprozesse von speziellen Komponenten eines Systems, wie Software oder Hardware, beziehen. CLs können wiederum untergeordnete CLs innehaben und umfassen, für sich, zusätzlich eine Sammlung von mehreren Checkfragen.

Ein *Checklistentyp* (*CLT*) ist eine Vorlage einer CL. Analog zu ORTs (siehe auch Kapitel 2.4.1) besitzen CLTs bereits ausgefüllte Attribute. Ein CLT kann in einen ORT oder in einen übergeordneten CLT integriert sein.

Eine *Checklisteninstanz* (*CLI*) kann durch das individuelle Erstellen oder das Ableiten von einem CLT entstehen. Eine CLI kann nach dem Erstellen, beliebig verändert werden. Einer CLI können entweder weitere CLIs oder Checkfragen (siehe Kapitel 2.4.3) hinzugefügt werden.

2.4.3 Checkfrage

Eine *Checkfrage* (*CF*) ist ein Element einer CL und wird im Folgenden ebenso als *Item* bezeichnet.
Jede CF besitzt eine zentrale Fragestellung, welche im Optimalfall im Verlauf eines ORs beantwortet wird. Sie sind in einer CL logisch aufgelistet und bestimmen die abzuarbeitende Reihenfolge einer Anforderung. Dabei besitzen CFs verschiedene standardisierte Attribute [Mun13]. Gleichermaßen spiegeln sie eine Vollständigkeitsprüfung wieder. CFs sind geschlossene Fragen und können damit ausschließlich mit *ja* oder *nein* beantwortet werden und erwarten eine präzise und objektive Antwort.

Ein *Checkfragentyp* (*CFT*) ist eine Vorlage für eine CF. Ein Benutzer kann nach CFTs suchen und anhand diesen eine Checkfrageninstanz ableiten, welche er dann in eine CLI integriert.

Eine *Checkfrageninstanz* (*CFI*) kann durch das Ableiten von einem CFT entstehen oder individuell erstellt werden. CFIs besitzen die selben Attribute wie ein CFT. Im Gegensatz zu CFTs sind CFIs individuell angepasste CFs, welche in genau eine CLI integriert sind.

3

Anforderungen

Dieses Kapitel zeigt jene Anforderungen auf, die für die Realisierung des proCollab-Prototypen erforderlich sind. Dabei wird auf konkrete Anwendererzählungen von Benutzern eingegangen. Funktionale Anforderungen beschreiben das Verhalten des Systems und leisten einen wichtigen Beitrag über die geforderte Funktionalität. Nicht-funktionale Anforderungen definieren die Qualität, mit der der proCollab-Prototyp umgesetzt werden muss, um vom Benutzer akzeptiert zu werden.

3.1 Anwendererzählung

Eine Anwendererzählung (auch *User Story* genannt), ist eine schriftliche Beschreibung der Bedürfnisse eines Anwenders und stellt funktionale Anforderungen an das System in einer für den Benutzer verständlichen Sprache [Coh04]. User Stories bieten einen Mehrwert, da von anhand diesen überprüft werden kann, ob das Systemverhalten mit der Beschreibung übereinstimmt.

Im Rahmen des Projektes proCollab wurde eine große Anzahl von User Stories für das Gesamtsystem erstellt. Dabei gibt es verschiedene Einstufungen, welche zwischen den Prioritäten *sehr hoch*, *hoch*, *mittel* und *gering* differenzieren.
Im Folgenden werden Beispiele von Systemfunktionalitäten veranschaulicht, die mit der Priorität *sehr hoch* eingestuft sind. Es handelt sich um ausgewählte Funktionalität, welche im proCollab-Prototyp essentiell ist und einen hohen Stellenwert besitzt. Auf die User Stories mit geringerer Priorität wird im Folgenden nicht eingegangen. Weitere Systemfunktionen lassen sich daher aus Kapitel 5 entnehmen.

3.1.1 Anmeldung im System

Ein Benutzer des Systems kann sich nach der erfolgreichen Registrierung im System anmelden. Dazu gibt er seine festgelegte E-Mail Adresse sowie sein persönliches Passwort ein. Der Server prüft die Berechtigung anhand der übergebenen Attribute.
Der Benutzer erhält die Information, dass er im System angemeldet ist, sofern die eingegebene E-Mail und das Passwort mit dem aus dem Speicher übereinstimmt. Stimmen die Werte nicht überein, wird auf der Benutzeroberfläche ein Fehler angezeigt sowie die Möglichkeit den Vorgang zu wiederholen. Nach der erfolgreichen Anmeldung wird dem Benutzer eine Sitzung zugeteilt. Dadurch wird er nicht mehr erneut nach seinen Zugangsdaten gefragt.

3.1.2 Erstellen einer organisatorischen Rahmeninstanz

Jeder Benutzer des Systems kann eine ORI erstellen. Diese kann er entweder unabhängig von einem ORT oder basierend auf einem bereits existierenden ORT erstellen. Dazu lässt sich der Benutzer die Liste aller vorhandenen ORTs im System anzeigen und kann sich die Details zu einem speziellen ORT anzeigen lassen, indem er diesen selektiert. Stimmen die Angaben mit seinen Anforderungen überein, kann er eine neue ORI basierend auf einem ORT instanziieren lassen. Der Benutzer nimmt dadurch automatisch die Rolle des Managers in seinem erstellten ORI ein. Als Manager des Projektes hat er die Möglichkeit weitere ORIs, CLIs und CFIs der ORI hinzuzufügen.

3.1.3 Aktualisieren einer Checklisteninstanz

Besitzt ein Benutzer die erforderlichen Rechte darf er die Informationen zu einer CLI verändern. Dazu wählt er die CLI die er aktualisieren möchte aus und gibt die Daten, die er verändern möchte, in das Formular der Benutzeroberfläche ein. Ein Benutzer kann den Namen, den Text oder den Status einer CLI aktualisieren.
Eine CLI kann zusätzlich in ihrem Aufbau angepasst werden. Der Benutzer darf CLIs oder CFIs, die sich innerhalb einer CLI befinden, von einer Position an eine andere verschieben. Zum Beispiel kann eine CFI ausgewählt werden und mit einer anderen vertauscht werden. Dadurch hat der Benutzer die Möglichkeit eine CLI beliebig in ihrem Aufbau zu strukturieren.

3.2 Funktionale Anforderungen

Der zu implementierende proCollab-Prototyp stellt Funktionalität zu den Modulen *Benutzerverwaltung*, *Rahmenverwaltung* sowie *Checklistenverwaltung* bereit. Funktionale Anforderungen legen fest, welche Funktionen der proCollab-Prototyp erbringen muss. Im Folgenden werden die Anforderungen genauer spezifiziert und modular zusammengefasst.

3.2.1 Benutzerverwaltung

Innerhalb des proCollab-Prototypen stehen verschiedene Systemrollen zur Verfügung. Dabei wird zwischen den Systemrollen *Administrator*, *Manager*, *Employee*, *User* und *None* unterschieden. Administratoren können nur von bereits vorhandenen Administratoren erstellt werden. Registriert sich ein Benutzer im proCollab-Prototyp erhält dieser automatisch die Systemrolle User. User besitzen eingeschränkte Funktionalität und können nicht alle Systemfunktionen gleichermaßen nutzen (siehe Kapitel 3.2.2, 3.2.3 und 3.2.4) .
Administratoren des proCollab-Prototypen hingegen können das Benutzerprofil eines

jeden Benutzers editieren und auch löschen.
Generell soll der proCollab-Prototyp dem Benutzer ermöglichen, sein eigenes Profil zu verwalten. Dazu hat der Benutzer initial die Möglichkeit, sich zu registrieren und ein Profil anzulegen. Der Benutzer kann sich nach der erfolgreichen Registrierung und der Validierung seiner E-Mail Adresse am proCollab-Prototyp anmelden. Im System steht dem Nutzer eine Profilverwaltung zur Verfügung. Er kann seine persönlichen Daten (Vor- und Nachname, E-Mail Adresse und Passwort), einschließlich der zugehörigen Organisation anzeigen lassen oder diese aktualisieren. Wird die E-Mail Adresse geändert, ist eine erneute Validierung erforderlich. Nach einer Sitzung kann sich der Benutzer abmelden. Falls nicht, läuft die Sitzung, welche mit einer limitierten Gültigkeitsdauer versehen ist, automatisch ab und der Benutzer wird abgemeldet. Möchte ein Benutzer den proCollab-Prototypen nicht mehr nutzen, kann er seine Mitgliedschaft beenden. Dazu wird dem Benutzer eine E-Mail zugesandt, die bestätigt werden muss, um das Profil endgültig zu löschen. Nach dem Löschen steht dem betroffenen Benutzer das System nicht mehr zur Verfügung.

3.2.2 Verwaltung von ORTs und ORIs

Ein Administrator des proCollab-Prototypen ist für das Erstellen von ORTs zuständig. Dazu gibt er die Attribute des ORTs in ein Formular ein. Falls erforderlich, kann er den ORT in einen anderen ORT integrieren. Er besitzt die Möglichkeit sowohl ORTs als auch ORIs von einem existierenden ORT abzuleiten. Ein Administrator kann nach ORTs suchen, diese aktualisieren oder löschen. Wird ein ORT gelöscht, kann von diesem keine weitere ORI abgeleitet werden.

Ein Benutzer mit der Systemrolle User kann nur ORIs erstellen. Dazu muss er den Namen der zukünftigen ORIs angeben und kann zusätzliche Attribute (zum Beispiel: Start- und Enddatum oder das Ziel) setzen. Wie in Teilabschnitt 3.1.2 beschrieben, können ORIs entweder unabhängig oder anhand eines ORT erstellt werden. Leitet der Benutzer eine ORI von einem ORT ab, kann er diese anschließend spezifisch aktualisieren und anpassen. Er erhält dadurch automatisch die Rolle des *Managers* in

der ORI. Die Rolle *Manager* bezieht sich dabei auf die Rolle innerhalb einer ORI und sollte nicht mit der Systemrolle verwechselt werden. Im Folgenden wird dabei von der Rolle innerhalb einer ORI ausgegangen.
Verfügt ein Benutzer über die notwendige Rolle innerhalb einer ORI (im Allgemeinen: Manager), kann er die betreffende ORI verändern, löschen oder verschieben. Nur Manager einer ORI können die ORI mittels einer Bestätigung abschließen. Zusätzlich können einer ORI weitere ORIs als auch CLIs durch den Manager zugeordnet werden.

3.2.3 Verwaltung von CLTs und CLIs

Administratoren des proCollab-Prototypen sind wie auch in Kapitel 3.2.2 für die Verwaltung von CLTs zuständig. Er kann von einem CLT entweder einen neuen CLT oder eine neue CLI ableiten. Nach dem Erstellen eines CLTs muss dieser in einen existierenden ORT oder CLT eingefügt werden. Leitet ein Administrator eine CLI von einem CLT ab, muss diese analog entweder in eine ORI oder eine CLI eingefügt werden. Des Weiteren ist er für das Aktualisieren und Löschen von CLTs zuständig. Soll ein CLT gelöscht werden, sucht ein Administrator nach dem entsprechenden CLT und löscht diesen, woraufhin keine weiteren CLIs von dem CLT abgeleitet werden können.

Ein Manager kann innerhalb einer ORI oder CLI weitere CLIs erstellen. Analog zu der Vorgehensweise in Kapitel 3.2.2, kann eine CLI entweder unabhängig erstellt oder von einem CLT abgeleitet werden. Um eine CLI von einem existierenden CLT abzuleiten, wählt der Manager einen CLT aus einer bestehenden Liste aus und nimmt im Anschluss gegebenenfalls spezifische Anpassungen vor. Die CLI wird darauf folgend der jeweiligen ORI oder CLI hinzugefügt.
CLIs können von dem Manager der umfassenden ORI aktualisiert werden. Dazu stehen ihm die Attribute in einem Formular zur Verfügung, welche hinsichtlich der neuen Anforderungen angepasst werden können. Der Manager hat auch die Fähigkeit, CLIs aus einem Projekt zu löschen. Der Manager kann innerhalb einer CLI, weitere untergeordnete CLIs oder CFIs einfügen und diese beliebig platzieren oder verschieben.

3.2.4 Verwaltung von CFTs und CFIs

Im proCollab-Prototypen sind, wie zuvor in Kapitel 3.2.2 und 3.2.3 beschrieben, Administratoren für die Verwaltung von CFTs zuständig. Sie können CFTs erstellen und müssen diese einem CLT unterordnen. Soll ein CFT verändert oder gelöscht werden, sucht ein Administrator nach dem erforderlichen CFT und führt die gewünschte Operation aus. Handelt es sich dabei um eine Löschoperation, können keine weitere CFIs ableitet werden.

Nachdem ein Manager einer ORI eine CLI erstellt hat, kann er der CLI neue CFIs zuweisen. Um eine neue CFI zu erstellen, muss der Manager (vgl. Kapitel 3.2.2 und 3.2.3) diese entweder von einem CFT ableiten oder individuell erstellen. Der Manager der ORI kann nach CFIs innerhalb einer CLI suchen, indem er die Suchparameter (z.B. Name oder Text) eingibt und folglich, falls vorhanden, CFIs aufgelistet werden. Anschließend kann er die CFI aktualisieren, löschen oder verschieben. Beteiligte Personen, die an einer CLI arbeiten, können nur den Status einer CFI auf *abgeschlossen* setzen. Sie können allerdings nicht die weiteren Attribute einer CFI verändern.

3.3 Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen legen fest, *wie* die in Kapitel 3.2 definierten funktionalen Anforderungen des Systems zu erbringen sind. Da sich durch die Zielsetzung diese Arbeit mit der Konzeption und Realisierung der Anwendungslogik des proCollab-Prototypen beschäftigt, spezifiziert folgender Abschnitt die nicht messbaren Qualitätsmerkmale der pCSK [Bal09]. Um Qualitätsmerkmale adäquat zu definieren, wird auf den ISO/IEC Standard 9126-1 zurückgegriffen [Hor07]. Folgende Abbildung 3.1 unterscheidet mögliche Typen von nicht-funktionalen Anforderungen.

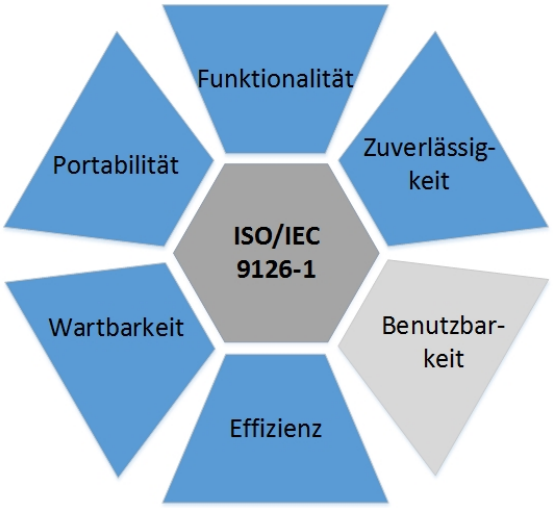


Abbildung 3.1: Nicht-funktionale Anforderungen nach ISO/IEC 9126-1 [Hor07]

Die *Benutzbarkeit* ist die Eigenschaft, den proCollab-Prototypen derart zu entwickeln, dass er für den Benutzer selbsterklärend, -bedienend und -erlernbar ist. Dieser Aspekt wird im Folgenden nicht erläutert, da er für die Konzeption der pCSK nicht relevant ist. In Bezug auf die Anwendungsschicht ist die Realisierung dieser Bedingung umso wichtiger [Gei13, Köl13, Thi13].
Damit der proCollab-Prototyp für den Benutzer *effizient* nutzbar ist, soll die pCSK ein adäquates Leistungsniveau bereitstellen. Die angefragten Ressourcen sollen für die Applikationen durch geringen Datenverkehr angemessene Antwortzeiten sicherstellen.
Die pCSK soll die Möglichkeit bieten, *änderungsfähig* und *wartbar* zu sein. Dadurch sollen Anpassungen der pCSK so durchgeführt werden können, dass die Funktionalität auf der Seite der Applikationen nicht beeinträchtigt wird. Erweiterungen der pCSK können neue Schnittstellen entstehen lassen, welche im Anschluss für die Applikationen zur Verfügung stehen. Die pCSK soll stabil sein und die Fähigkeit besitzen, trotz vorgenommenen Änderungen in der Programmausführung, für Endgeräte verfügbar zu sein.
Die *Portabilität* der pCSK beschreibt das Vorhaben, in andere Umgebungen übertragbar zu sein, um die pCSK plattformübergreifend ausführen zu können.
Der Aspekt der *Funktionalität* bezeichnet die Fähigkeit, die in Kapitel 3.2 spezifizierte

Systemfunktionalität so umzusetzen, dass folgende Bedingungen erfüllt werden: Die Funktionen des proCollab-Prototypen sollen angemessen sein und durch die pCSK bereitgestellt werden.
Die pCSK soll die Eigenschaft besitzen, trotz Fehlzuständen *zuverlässig* zu bleiben und fehlertolerant gegenüber Zuwiderhandlung der definierten Schnittstelle zu sein. Nach einer erfolgten Fehleingabe soll die pCSK die Möglichkeit zur Fehlerbehandlung anbieten und dem Benutzer die Chance geben, die betroffenen Daten zu korrigieren.

4

Konzept

Kapitel 4 - Konzept - spezifiziert die zugrundeliegende Architektur des Systems und beschreibt den Aufbau in Schichten. Um tiefer gehende Kenntnis über die interne Struktur des Systems zu erhalten, wird das Datenmodell sowie das zugehörige Zustandsmodell detailliert illustriert. Da die Anwendungsschicht ihre Funktionalität in Form von Services an die Client-Schicht bereitstellt, wird des Weiteren zu dem Paradigma der *Service-orientierten Architektur* Stellung genommen. *Cloud Computing* setzt die Verwendung von Services voraus, welche durch Service Prinzipien einer Service-orientierten Architektur gegeben werden.

4.1 Datenmodell

Das Datenmodell beschreibt den zugrundeliegenden Aufbau des proCollab-Prototypen und spezifiziert dabei die Entitäten sowie die einzelnen Zustandsmodelle von ORs, CLs und CFs.

4.1.1 Entitäten

Basierend auf den Anforderungen des Systems sind mehrere statische Entitäten ableitbar.
Die Entität *Person* besitzt die Attribute Vorname, Nachname, E-Mail Adresse und Passwort. Die Attribute müssen bei der Registrierung angegeben werden, können aber im System jederzeit vom Benutzer selbst verändert werden.
Manager können zu einer ORI, sowohl CLIs als auch CFIs hinzufügen. Die Funktion zum Erstellen und Verwalten von ORTs, CLTs und CFTs ist nur den Administratoren im System erlaubt. Intern wird mit Hilfe des Attributes *createdBy* festgelegt, durch wen eine Instanz erstellt wurde. Nachdem eine ORI erstellt wurde, werden dieser involvierte Personen hinzugefügt, welche an der ORI teilnehmen. Das Attribut *involved*, legt für jede einzelne Person fest, an welcher ORI sie beteiligt ist und welche Rolle sie in der ORI spielt.
Jeder Person ist eine *Role* im System zugeordnet. Mögliche Rollen sind *Administrator*, *Employee*, *Manager*, *User* und *None* (siehe auch Kapitel 3.2.1). Jeder Benutzer kann dabei nur eine Rolle annehmen, der Administrator besitzt jedoch die Möglichkeit die Rolle eines Benutzers im Verlauf zu ändern.
Zusätzlich zu jeder Systemrolle existieren Rollen innerhalb einer ORI. Jeder Benutzer kann eine ORI, unabhängig von seiner Rolle im System, erzeugen. Nach dem Erstellen nimmt der Benutzer die Rolle eines *Managers* innerhalb seiner angelegten ORI an.
Jede Person im System kann eine zugehörige *Organisation* angeben. Das Attribut *worksFor* innerhalb des Personenobjekts legt fest, zu welcher Organisation eine Person angehörig ist. Eine Organisation besitzt die Attribute Name und URL. Der Name spezifiziert die entsprechende Organisation, die URL gibt die Internetadresse der Organisation an. Jede Organisation besitzt eine Liste *employees* mit angestellten Personen.
Das folgende UML-Klassendiagramm in Abbildung 4.1 zeigt die Abhängigkeiten zwischen den beschriebenen Entitäten.

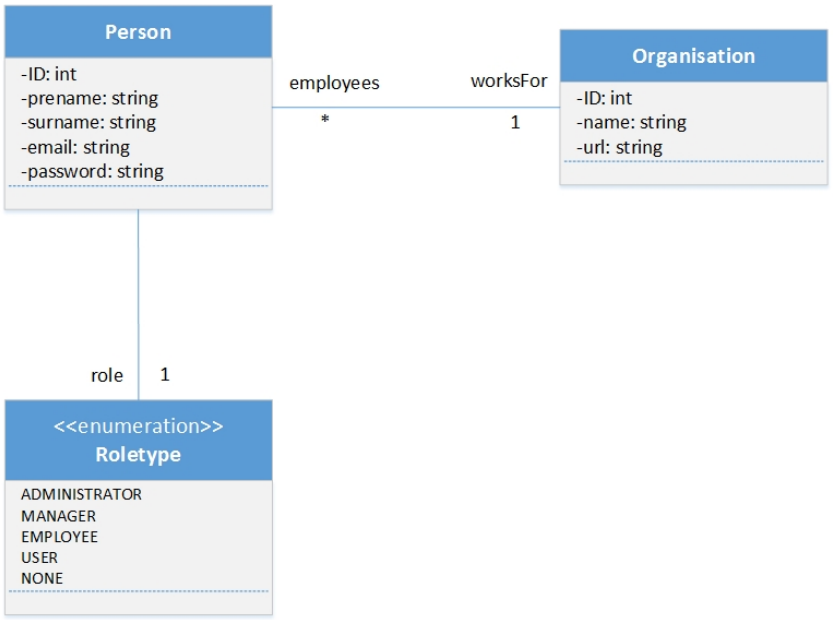


Abbildung 4.1: Teilausschnitt des UML-Klassendiagramms zur Darstellung der Abhängigkeiten zwischen Person, Organisation und Role

Eine ORI (*cFrameInstance*) besitzt einen Namen, sowie ein festgelegtes Ziel. Der *languageCode* gibt an, in welcher Sprache das Projekt verfasst wurde. Jedes Projekt erhält ein Erstelldatum, ein Startdatum und ein Enddatum. Der *daysOffset* gibt die Laufzeit der ORI in Tagen an und ist Differenz zwischen Start- und Enddatum. Das Attribut *createdBy*, gibt die Person an, durch die die ORI erstellt wurde. Der Abarbeitungszustand einer ORI wird durch das Attribut *state* festgelegt (siehe dazu Kapitel 4.1.2). Die untergeordneten ORIs werden als Kinder (*childFrames*) bezeichnet und enthalten eine Referenz zu der übergeordneten ORI. Jedes Projekt besitzt eine Liste *containLists*, welche auf die integrierten CLIs referenziert.
Die Entität *cFrameType* enthält die selben einfachen Datentypen wie die cFrameInstance und stellt einen ORT dar. Damit einsehbar wird, wie viele und welche ORIs von dem ORT abgeleitet wurden, existiert das Attribut *linkedFrameInstances*. Die Abhängigkeiten werden durch das UML-Klassendiagramm in Abbildung 4.2 aufgezeigt.

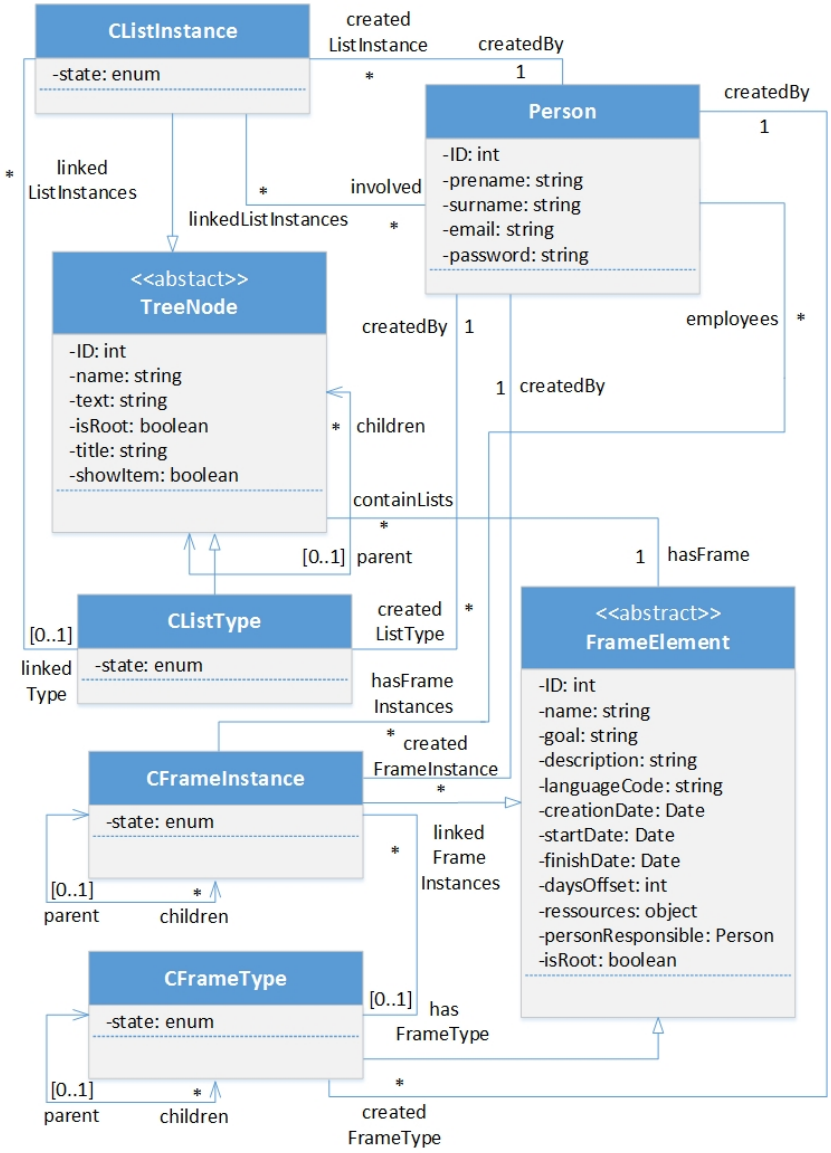


Abbildung 4.2: Abhängigkeitsverhältnis zwischen den Entitäten Person, ORT, ORI, CLT sowie CLI

Eine *cListInstance* charakterisiert eine CLI und verfügt über folgende einfache Datentypen: einen Namen, einen zugehörigen Text, sowie einen Titel. Das Attribut *state* beschreibt in welchem Status sich die CLI befindet (siehe dazu Kapitel 4.1.2). Das Attribut *hasType* gibt an von welchem CLT die CLI abgeleitet ist. *createdBy* referenziert diejenige Person, die die CLI erzeugt hat. Um zu wissen, welcher ORI die CLI zugeordnet ist, wird das Attribut *hasFrame* verwendet.
Eine CL, wie auch ein OR oder eine CF, ist intern innerhalb einer Baumstruktur repräsentiert. In Teilabschnitt 5.3 (Umsetzung der Komponenten) wird dabei näher auf die Repräsentation eines Baumes eingegangen. Damit diese Struktur verwaltet werden kann, werden den Entitäten zusätzliche Attribute hinzugefügt, die für die Systemnutzer keine Rolle spielen. Das Attribut *isRoot* gibt exemplarisch an, ob es sich bei einer CL um ein Hauptelement (im Allgemeinen: eine Wurzel) handelt oder nicht.
Der dazugehörige *cListType* besitzt analog dieselben einfachen Datentypen, unterscheidet sich daher allein durch die Referenzen zu verschiedenen Objekten. Der CLT wird durch die Liste *linkedListInstances* erweitert, welche auf die daraus abgeleiteten CLIs verweist.
Eine CFI, eine *cItemInstance*, besitzt einen Namen, einen Text sowie den Abarbeitungsstatus *state*, welcher angibt, in welchem Zustand sich die CFI befindet. CFIs sind immer innerhalb einer CLI angesiedelt und verwenden das Attribut *hasItemType*, um auf den CFT zu verweisen, durch den die CFI erstellt wurde. Das Attribut *createdBy* gibt an, von welcher Person die CFI erstellt wurde.
Der CFT, der *cItemType*, nutzt neben den selben einfachen Datentypen einer CFI die Liste *linkedItemInstances*, welche die aus dem CFT abgeleiteten CFIs innehält. Dies verdeutlicht die folgende Abbildung 4.3.

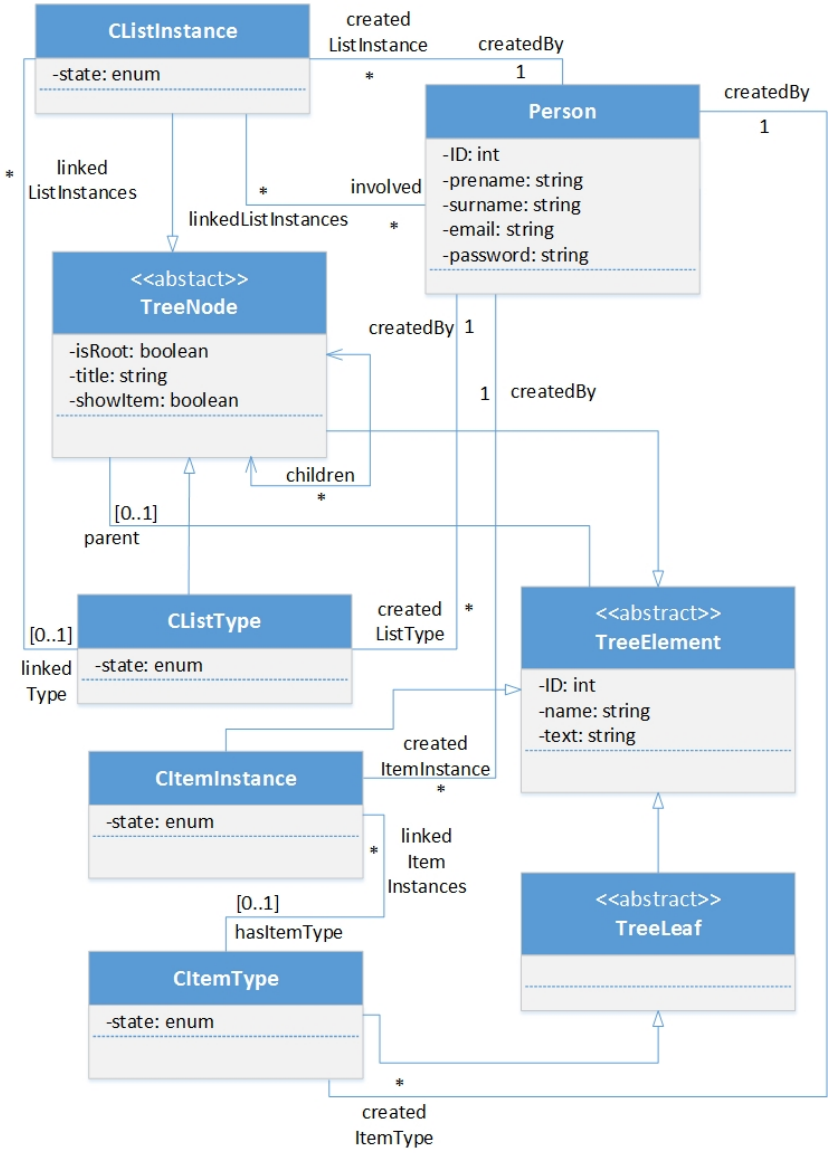


Abbildung 4.3: Abhängigkeitsverhältnis zwischen den Entitäten Person, CLI, CLT, CFI und CFT

4.1.2 Zustandsmodell

Wie in Kapitel 4.1.1 beschrieben, durchlaufen die Entitäten für ORTs, ORIs, CLTs, CLIs, CFTs und CFIs verschiedene Zustände. Dies ist vor allem von Seiten der Nutzer äußerst hilfreich, da zu jedem Zeitpunkt sichtbar ist, in welchem Abarbeitungsstatus sich eine Entität befindet. Zustandsübergänge werden durch ein Ereignis von außen getriggert und bilden mögliche Aktionsfolgen ab. Jede Entität enthält ein Attribut *state*, welches den momentanen Zustand repräsentiert. Für jeden Zustand ist eine Reihe von Vorgänger- und Nachfolgezustände definiert.

4.1.2.1 Zustandsmodell für CLTs und CFTs

Erstellt ein Administrator des proCollab-Prototypen einen neuen CLT oder CFT, befindet sich diese initial in dem Zustand *prepared*. Der CLT oder CFT ist exklusiv für den Ersteller sichtbar und kann nur von diesem modifiziert werden. Sobald der Typ gespeichert wird, wird er für alle andere Nutzer des Systems zugänglich. Der Zustand des Typs wechselt in den Zustand *available*. Nimmt der Ersteller des Typs erneut Änderungen vor, wechselt der Zustand wieder von *available* in *prepared*. Ein verfügbarer Typ kann aus dem Zustand *available* in den Zustand *archived* und vice versa übergehen, da der proCollab-Prototyp die Möglichkeit bietet, bereits archivierte Vorlagen wieder verfügbar zu machen und zu nutzen. Folgende Abbildung 4.4 zeigt das Verhalten von CLTs und CFTs:

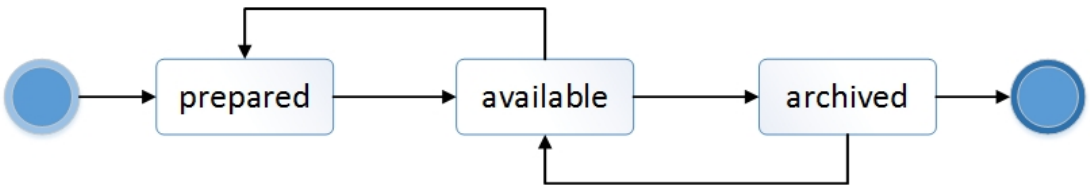


Abbildung 4.4: Zustandsdiagramm für CLTs und CFTs

4.1.2.2 Zustandsmodell für ORTs

Analog zu den CLTs und CFTs agiert das Zustandsmodell eines ORTs (siehe Abbildung 4.5). Zu Beginn, während der Bearbeitung eines ORTs, ist dieser für andere Systemnutzer nicht sichtbar und wird dies erst genau dann, sobald die Bearbeitung des organisatorischen Rahmens abgeschlossen und gespeichert wurde. Bei erneuter Bearbeitung des Erstellers wird wieder in den Zustand *prepared* gewechselt, andernfalls kann ein ORT archiviert werden, sofern er nicht mehr benötigt wird. Gleichermaßen können archivierte ORTs wieder zugänglich gemacht werden. Das Zustandsmodell (Abbildung 4.5) zeigt das Verhalten der Vorgänger- und Nachfolgezustände für ORTs.

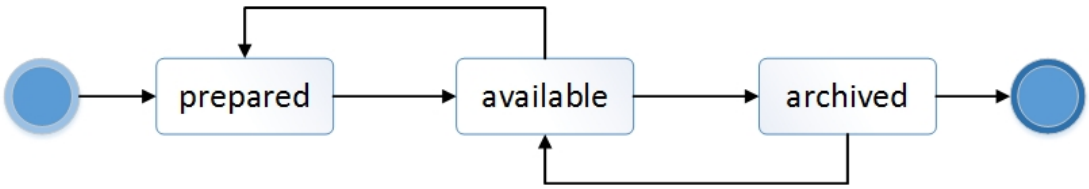


Abbildung 4.5: Zustandsdiagramm für ORTs

4.1.2.3 Zustandsmodell für ORIs

Nachdem eine ORI erstellt wurde, befindet sich diese anfänglich in dem Zustand *initialized*. Das Vorhaben kann entweder direkt wieder abgebrochen werden und in den Zustand *cancelled* versetzt werden, oder nach dem Hinzufügen von CLIs und CFIs den Zustand *running* annehmen. Der ORI befindet sich solange in dem Zustand *running*, bis alle CLIs und deren CFIs abgearbeitet worden sind. Danach wird von dem Verantwortlichen der betreffenden ORIs bestätigt, dass dieses vollendet wurde und somit in den Zustand *finished* gebracht werden kann. Es ist zu keiner Zeit möglich, in einen Vorgängerzustand zurückzukehren. Wurde eine ORI abgeschlossen oder abgebrochen, wird sie in beiden Fällen den Nachfolgezustand *archived* annehmen. Dies wird durch folgende Abbildung 4.6 skizziert.

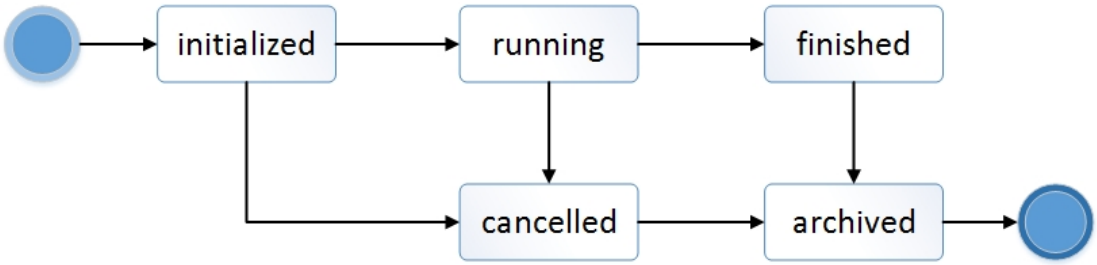


Abbildung 4.6: Zustandsdiagramm für ORIs

4.1.2.4 Zustandsmodell für CLIs und CFIs

Die Ausgangssituation ist das Erstellen einer CLI. Für CFIs ist dieser Prozess analog. Zu Beginn ist die CLI (oder CFI) automatisch in dem Zustand *open*. Wird die CLI erstmals bearbeitet, wird von dem Zustand *open* in den Zustand *in_process* gewechselt. Es ist möglich, dass die CLI wieder in den initialen Ausgangszustand *open* gebracht wird. CLIs die als *open* markiert sind, können als Nachfolgezustand zusätzlich *postponed* oder *dropped* annehmen. *Postponed* bedeutet, dass die Bearbeitung der CLI vorerst in den Hintergrund gestellt wird, wohingegen *dropped* das komplette Entfernen einer CLI aus einem Projekt bedeutet. Umgekehrt, kann aus einer zeitlich verschobenen oder sogar entfernten CLI wieder in den Zustand *open* zurückgekehrt werden, um deren Bearbeitung wieder zu beginnen. Aus dem Zustand *in_process*, kann wie auch aus dem Zustand *open* in *postponed* oder *dropped* gewechselt werden. Während von *postponed* wieder zurück *in_process* gewechselt werden kann, ist dies bei *dropped* nicht möglich. Wird eine CLI als *dropped* markiert, muss zuerst wieder in den Zustand *open* zurückgegangen werden. Nach erfolgreicher Abarbeitung der CLI wird in den Zustand *finished* gewechselt. Aus diesem Zustand kann nur noch der Zustand *archived* erreicht werden. Es können entweder erfolgreich abgearbeitete CLIs, die als *finished* markiert sind, oder abgebrochene CLIs, die als *dropped* markiert sind, archiviert werden. Das im Folgenden dargestellte Zustandsdiagramm (Abbildung 4.7) zeigt die möglichen Zustände von CLIs und CFIs.

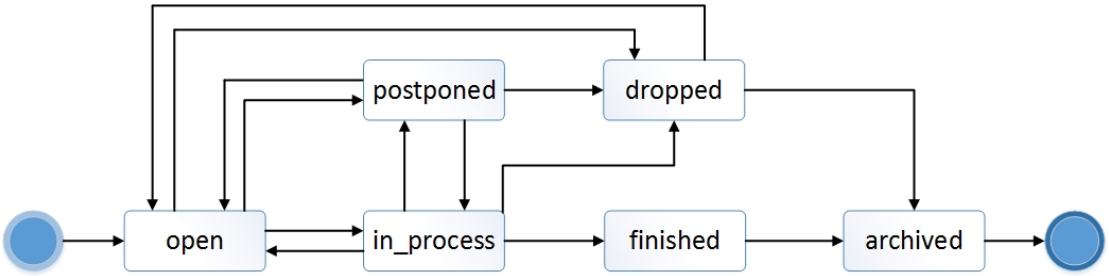


Abbildung 4.7: Zustandsdiagramm für CLIs und CFIs

4.1.3 Checklisten als Baumstruktur

CLs können sehr groß werden und müssen daher auf eine geeignete Art und Weise zur Verwaltung im proCollab-Prototypen repräsentiert werden. CLs liegen in einer Blockstruktur vor und sind hierarchisch aufgebaut. Aus Kapitel 2 geht hervor, dass eine CL in sich weitere CLs und CFs enthalten kann und deshalb beliebig tief geschachtelt werden darf. Abbildung 4.8 zeigt eine beispielhafte Darstellung einer CL in Blockstruktur.

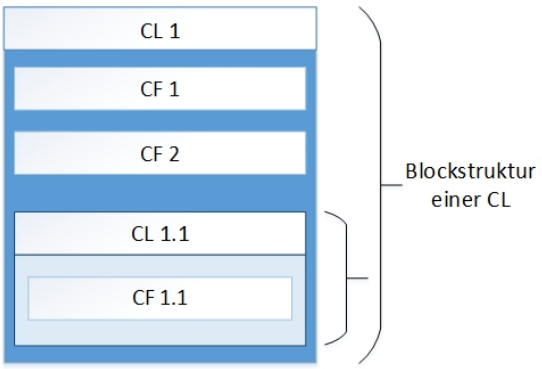


Abbildung 4.8: Darstellung einer CL in Blockstruktur

Generell lassen sich Blockstrukturen durch die Datenstruktur eines Baumes darstellen. Im Speziellen verkörpert das Modell von CL die Repräsentation eines n-ären Baumes. Ein *n-ärer Baum* ist eine dynamische zusammenhängende Datenstruktur und zwischen

0 und n Kinder besitzen. Ein n-ärer Baum lässt zu, dass jeder Knoten unterschiedlich viele Kinder auf beliebig vielen Ebenen besitzen darf. Diese Eigenschaft ist für den proCollab-Prototypen zwingend erforderlich, da jede CL individuell erstellt ist und deswegen unterschiedlich viele Knoten besitzen kann. Ein n-ärer Baum bietet damit den Vorteil, dass keine Bedingungen bezüglich der Anzahl der Knoten gestellt wird, die den proCollab-Prototypen einschränken könnte.
Jeder Baum enthält einen markierten Knoten, der die *Wurzel* darstellt. Dieser wird durch den Wahrheitswert *isRoot* festgelegt und besitzt logischerweise keinen Elternknoten.
Ein sogenannter *Elternknoten* ist derjenige Knoten, der sich, falls existent, eine Ebene über einem Knoten befindet und mit diesem durch eine direkte Kante verbunden ist. Alle Elternknoten und die Wurzel einer CL sind wiederum selbst CLs.
Analog wird von einem *Kindknoten* gesprochen, wenn ein Knoten, der durch genau eine Kante mit einem Elternknoten verbunden ist, eine Ebene unter dem Elternknoten liegt.
Ein *Blatt* ist ein Knoten der keine Kindknoten besitzt, während die Knoten, die zwischen der Wurzel und den Blättern liegen, *innere Knoten* sind. Ebenen werden in der Länge von der Wurzel zum entsprechenden Knoten gezählt. Ebene 0 stellt die Wurzel dar [MM09]. Abbildung 4.9 zeigt die Verwendung der Begrifflichkeiten in einem Schaubild auf.

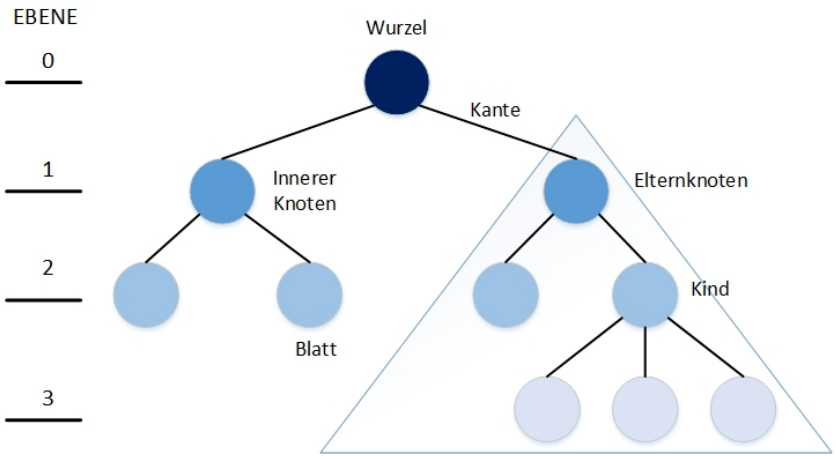


Abbildung 4.9: Struktur eines Baumes nach [MM09]

4.2 Service-orientierte Architektur

Die Voraussetzung zur Realisierung dynamischer, anpassungsfähiger Geschäftsprozesse stellt die Bereitstellung einer standardisierten Plattform dar.
Hervorgehend aus diesem Wandel hat sich in der Informatik das Architekturmuster der *Service-orientierten Architektur* (*SOA*) entwickelt, welche eine Menge von sogenannten Webservices bereitstellt.
Das Paradigma der Service-orientierten Architektur führt ein höheres Abstraktionsniveau ein und ist ein Ansatz für die Bewältigung und Pflege komplexer verteilter Systeme [Jos08]. Die Intention einer SOA ist das Erlangen von abstrahierten fachlichen Geschäftsregeln, um so Systeme zu strukturieren und zugänglich zu machen.
Der proCollab-Prototyp soll derart entwickelt werden, dass er flexibel, erweiterbar und skalierbar ist (siehe Kapitel 3), damit er auf Wachstum oder Änderungen schnellstmöglich reagieren kann. Eine SOA eignet sich für den proCollab-Prototypen, da alle benötigten Dienste zentral auf einer Plattform für alle beteiligten Komponenten zur Verfügung stehen sollen. Aus diesem Grund bietet das Prinzip der SOA dem proCollab-Prototypen einen Mehrwert, da infolgedessen eine neue Systemkomponente schnell durch Neukombination bereits existierender Services umgesetzt werden kann.

4.2.1 Service Prinzipien

Ein entscheidendes Konzept zur Realisierung einer SOA ist das Prinzip der *Services*.
Ein Service ist eine über das Internet aufrufbare, autonome und ortsunabhängige Softwarekomponente, dessen Spezifikation unabhängig von einer bestimmten Programmiersprache erfolgt [BHMS05].
Ein Service des proCollab-Prototypen bietet die Funktionalität, wie sie in Kapitel 3.2 beschrieben wurde, über eine Schnittstelle an und verbirgt infolgedessen die Implementierung hinter dieser Schnittstelle.
Im Folgenden wird aufgezeigt, welchen Nutzen Service-Orientierung für den proCollab-Prototypen hat, wie Services zusammenhängen und gegenseitig aufeinander einwirken [Erl05, Jos08, Mue09].

4.2.1.1 Wiederverwendbarkeit von Services

Es wird von Wiederverwendbarkeit gesprochen, sobald ein Service von mehreren Service-Konsumenten verwendet werden kann. Dieses Prinzip ist insbesondere für den proCollab-Prototypen von Bedeutung, da die Services für die mobilen Applikationen [Gei13, Köl13] als auch für das Backend [Thi13] zur Verfügung stehen sollen. Jede dieser Applikationen ist dabei ein Service-Konsument des proCollab-Prototypen. Anforderungen können durch wiederverwendbare Services schneller und kostengünstiger erfüllt werden. Wiederverwendbarkeit reduziert redundante Logik und baut diese langfristig ab. Die folgende Abbildung 4.10 zeigt exemplarisch, die Bereitstellung von Services innerhalb der Benutzerverwaltung.

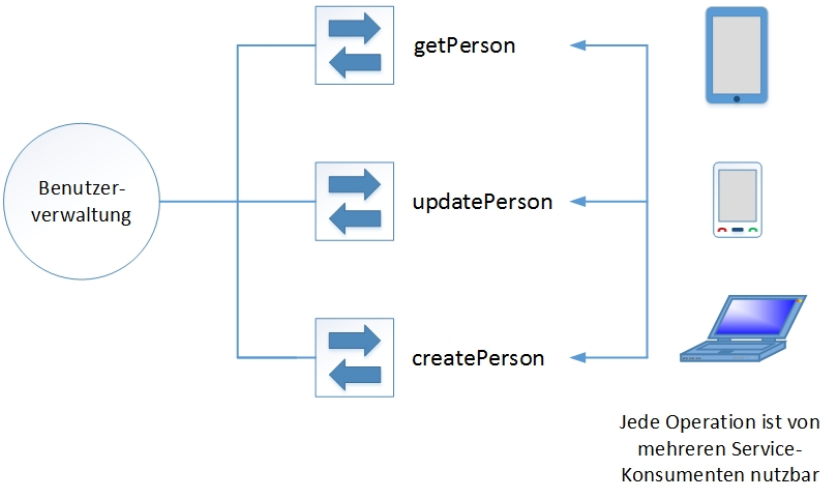


Abbildung 4.10: Wiederverwendbarkeit von Services nach [Erl05]

Abbildung 4.10 zeigt, wie Services der Benutzerverwaltung unabhängig von dem Service-Konsumenten genutzt werden können. Jeder Service steht als Funktionalität im proCollab-Prototyp bereit und kann von jedem Service-Konsumenten gleichermaßen genutzt werden. Das Vorhandensein dieser Grundlage spielt vor allem im Kontext der Service Komposition eine wichtige Rolle.

4.2.1.2 Standardisierte Service-Kontrakte

Ein Service Kontrakt ist die formelle Darstellung der Funktionalität eines Services, die zwischen Service-Konsument und -Erbringer festgelegt werden sollte. Der Kontrakt spezifiziert vollständig die Bedingungen und Regeln, welche eingehalten werden müssen, um mit einem Service interagieren zu können. Service-Konsumenten des proCollab-Prototypen kennen die zur Verfügung stehenden Schnittstellen sowie deren Eingabeparameter und Rückgabewerte, welche zur Interaktion mit einem Service benötigt werden. Bei einem korrekten Aufruf eines Services wird jedem Service-Konsumenten des proCollab-Prototypen eine garantierte Bereitstellung der Leistung zugesagt.

4.2.1.3 Service Auffindbarkeit

Damit ein Service aufgerufen werden kann, muss jeder Service automatisch auffindbar sein. Dazu registriert sich jeder Service des proCollab-Prototypen bei einem Verzeichnisdienst, der die Service-Schnittstellen wieder auffindbar ablegt. Je besser die Service-Schnittstelle ist, desto effektiver die Nutzung. Auffindbarkeit ist für den proCollab-Prototypen eine wichtige Voraussetzung, da dies das versehentliche Erstellen redundanter Dienste verhindert.

4.2.1.4 Lose Kopplung der Services

Das Prinzip der losen Kopplung beschreibt den Zustand der Unabhängigkeit zwischen Services, aber auch zwischen Service-Erbringer und -Konsument.
Lose Kopplung zwischen Services ist das Mittel, wodurch jeder Service Wissen über andere Services besitzt, aber dennoch unabhängig zu diesen bleiben soll. Es ist wichtig, dass die Services des proCollab-Prototypen unabhängig voneinander verwendet werden können, ohne dass intern Abhängigkeiten bestehen. Jeder Service des proCollab-Prototypen soll uneingeschränkt von anderen Services genutzt werden können. Dies verdeutlicht auch folgende Abbildung 4.11.

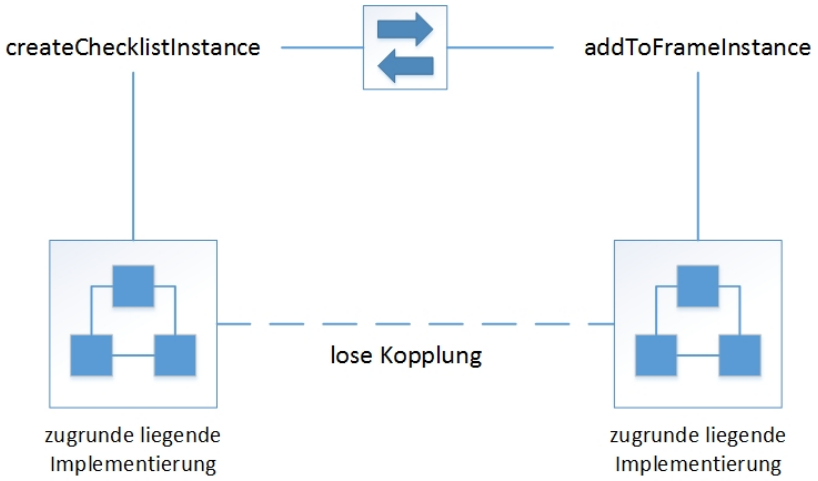


Abbildung 4.11: Lose Kopplung von Services nach [Erl05]

Die Abbildung 4.11 zeigt dabei exemplarisch zwei mögliche Services des proCollab-Prototypen, welche miteinander interagieren. Die zugrunde liegende Implementierung eines Services greift dabei auf Funktionen zurück, welche voneinander abhängig sein können. Dabei ist es wichtig, dass die Funktionalität des Services als Ganzes lose gekoppelt zu anderen Services aufgebaut ist. Je weniger Abhängigkeiten im proCollab-Prototypen vorhanden sind, desto minimaler sind die Nebeneffekte durch entstandene Fehlsituationen.
Gleichermaßen sollen die Services des proCollab-Prototypen derart implementiert werden, dass Service-Konsument und -Erbringer nur die Schnittstellen-Beschreibung benötigen, um miteinander zu kommunizieren. Dazu wird das Prinzip der Service Abstraktion benötigt.

4.2.1.5 Service Abstraktion

Der primäre Nutzen der Service Abstraktion ist, komplexe Prozesslogik zu kapseln und diese hinter einer Service-Schnittstelle zu verdecken. Damit agiert ein Service als eine Art „Black Box". Seine zugrundeliegenden Implementierungsdetails sind für den

Service-Konsument nicht sichtbar und der Service kann ohne detailliertes Wissen über die interne Verarbeitungslogik genutzt werden.
Die Übertragungsschicht des proCollab-Prototypen fungiert als eine solche zentrale Schicht, die Services anbietet. Die Client-Schicht kommuniziert mit den Services über die entsprechende Schnittstelle und nutzt diese ohne Kenntnis der zugrunde liegenden Logik. Service-Abstraktion führt zu einer Trennung zwischen der technischen und der fachlichen Ebene und soll in dem proCollab-Prototypen für langfristige Flexibilität sorgen. Die folgende Abbildung 4.12 zeigt die Service Abstraktion exemplarisch dargestellt.

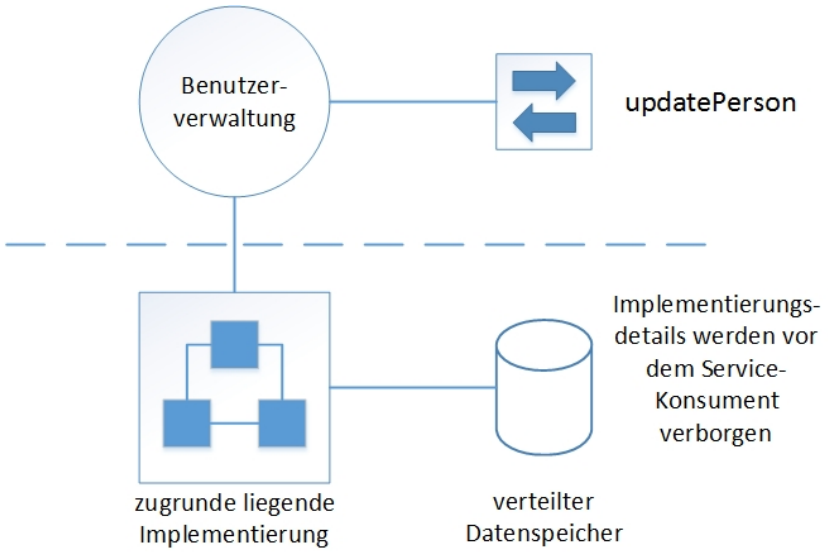


Abbildung 4.12: Service Abstraktion nach [Erl05]

Die Abbildung 4.12 zeigt einen Service des proCollab-Prototypen, welcher für Service-Konsumenten zur Verfügung steht. Das Prinzip der Service Abstraktion verbirgt die zugrundeliegende Implementierung und deren Datenmanipulationsfunktionen hinter einer Schnittstelle. Der Service-Konsument hat einzig und allein Kenntnis über die Schnittstelle.

4.2.1.6 Service Komposition

Services können in verschiedene Kategorien eingeteilt werden. Es wird von einem *Basis-Service* gesprochen, wenn ein Dienst eine einzige fachliche Funktionalität bereitstellt und nicht mehr weiter zerlegt werden kann. Basis-Services werden vor allem als Schnittstelle für Lese- und Schreibvorgänge zu bestehenden Daten verwendet.
Neben Basis-Services existieren *zusammengesetzte Services*. Ein zusammengesetzter Service ist die Komposition zweier oder mehrerer Basis-Services.
Dazu kann von folgendem Anwendungsfall ausgegangen werden: Ein Manager einer ORI möchte dieser eine neue CLI zuweisen. Insofern wird zuerst der Basis-Service *Erstellen einer CLI* aufgerufen und anschließend der Basis-Service *Hinzufügen zur ORI*. Beide Services gekoppelt bilden einen zusammengesetzten Service.
Services sind so zu implementieren, dass sie sowohl einzeln, als Basis-Service, sowie in Komposition mit mehreren Services, als zusammengesetzte Services, angewendet werden können. So kann einerseits eine fachliche Funktionalität aus mehreren Services bestehen, andererseits können höherwertige Funktionen durch die Komposition zweier oder mehrerer Services aggregiert werden.

4.2.1.7 Autonomie von Services

Autonomie ergibt sich aus der Grundlage, dass ein Service Kontrolle über die ihm bereitgestellte Verarbeitungslogik besitzt. Jeder Service des proCollab-Prototypen soll dabei autonom interagieren können und möglichst unabhängig von anderen Services sein.
Es ist zwischen zwei Arten von Autonomie zu unterscheiden:
Service-Level Autonomie besagt, dass sich mehrere Services möglicherweise zugrunde liegende Ressourcen teilen müssen.
Reine Autonomie hingegen garantiert dem Service, dass er vollständige Kontrolle und Nutzungsrecht der zugrunde liegenden Logik besitzt.
Durch das Prinzip der Autonomie kann ein Service des proCollab-Prototypen die zugrundeliegende Implementierung und die Ressourcen kontrollieren. Dadurch kann ein Service des proCollab-Prototypen autonom agieren, was ihn automatisch zuverlässiger und überschaubarer macht.

4.2.1.8 Zustandslosigkeit von Services

Services sollen zustandslos und möglichst einfach entwickelt werden. Als Zustandslosigkeit wird die Möglichkeit bezeichnet, nacheinander dieselbe Service-Operationen aufrufen zu können, ohne dass ein Zustand aufrecht gehalten werden muss.
Wird beispielsweise eine CLI ausgewählt und diese kurz darauf aktualisiert, soll nach dem Aufruf der betroffenen CLI kein Zustand gespeichert werden müssen. Der zweite Aufruf auf dieselbe CLI ist so zu implementieren, dass die Information zur CLI erneut übergeben werden muss. Der Service soll daher keine Informationen über ein- und ausgehende Nachrichten von Service-Konsumenten speichern. Ein Service nimmt nur kurzzeitig den Zustand zustandsbehaftet an, nämlich genau dann, um die Nachricht zu empfangen, diese zu interpretieren und zu verarbeiten.

4.3 Cloud-Computing

Cloud-Computing überführt Teile der IT Landschaft in das Internet als Verarbeitungsplattform und ermöglicht den Zugang zur Anwendung durch das Bereitstellen elektronisch erreichbarer Services [BKNT11]. Aus diesem Grund ist das Konzept einer SOA stärkste Voraussetzung für die Verwendung einer *Cloud*.
Eine weit verbreitete Definition des Cloud Computing stammt von der US-Standardisierungsstelle NIST (Nationales Institut für Standards und Technologie). Das NIST spezifiziert dabei fünf essentielle Eigenschaften [MG11]:

Diensterbringung auf Nachfrage
Die Provisionierung von Services, wie Rechenleistung und Speicherkapazität, kann durch den Service-Konsumenten automatisch und ohne menschliche Interaktion mit dem Dienstanbieter abgewickelt werden.

Umfassender Netzwerkzugang
Services sind über das Netzwerk verfügbar und können mit Standardtechnologien aufgerufen werden. Dabei sind sie an keinen spezifischen Client gebunden.

Pooling von Ressourcen

Die Ressourcen des Serviceanbieters sind in Pools verankert, um eine parallele Diensterbringung für die Konsumenten zu ermöglichen. Services sind unabhängig von der geografischen Lage und Service-Konsumenten wissen nicht, wo sich diese befinden.

Schnelle Elastizität

Ressourcen können schnell und elastisch - teilweise auch automatisch - beschafft werden und ermöglichen die Skalierbarkeit von Systemen. Daher erscheinen die Ressourcen für den Konsumenten unbegrenzt verfügbar zu sein.

Messbarer Service

Cloud-Anwendungen kontrollieren, messen und optimieren automatisch die Ressourcennutzung und stimmen diese nutzungsabhängig auf Konsumenten ab.

Die Konzeption sieht einen cloud-fähigen proCollab-Prototypen vor, da die Kollaboration nicht durch örtliche Grenzen eingeschränkt werden soll. Anwender können auf Nachfrage auf den proCollab-Prototypen über die ganze Welt verteilt zugreifen, da sich die Hardware nicht mehr in einem lokalen Rechenzentrum befindet. Die Cloud erscheint für den Anwender als ein zentraler abstrahierter Zugriffspunkt für alle Anfragen, welcher über definierte Schnittstellen und Protokolle wie HTTP erreicht wird. Cloud-Computing ermöglicht den Anwendern die notwendige Hardware für Rechenleistungen und Speicherkapazität nicht mehr selbst betreiben zu müssen, sondern bietet diese vielmehr als Dienst eines externen Anbieters an einem geografisch unabhängigen Ort an [BIT09].
Eine Webapplikation wie der proCollab-Prototyp soll zukünftig in der Cloud ausgeführt werden und dabei die dynamische Skalierbarkeit nutzen können. Werden mehrere Dienste gleichzeitig angefordert, werden diese im selben Moment geschaltet und bei Bedarf auf mehrere verteilte Serverinstanzen verteilt. Wird umgekehrt kein Dienst genutzt, ist kein Server in Betrieb. Dies führt zu einer Kosteneinsparung, da nur dann Anforderungen bearbeitet werden, wenn welche benötigt werden.

4.3.1 Grundvoraussetzungen

Cloud-Computing fasst bereits vorhandene Technologien zusammen und bietet diese als Internetdienste für Nutzer an [MRV11]. Die Konzeption des proCollab-Prototypen ist dabei auf die zukünftige Nutzung in der Cloud ausgelegt und passt sich damit perfekt den Grundvoraussetzungen an. Die folgende Abbildung 4.13 fasst die relevanten Standards zusammen:

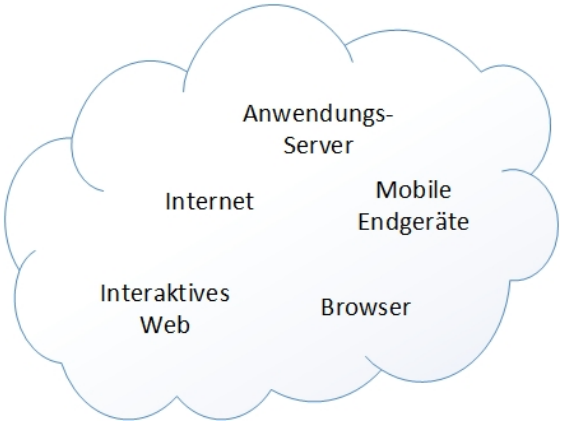


Abbildung 4.13: Standards für Cloud-Computing nach [MRV11]

Grundlegend ist die Verwendung eines *Anwendungs-Servers* auf dem der proCollab-Prototyp in Betrieb gesetzt wird. Der Anwendungsserver stellt Funktionalität in Form von Services bereit, die auf mehreren Servern verteilt werden, um die Datenübertragung zu steigern und Wartezeiten durch Warteschlangenschaltung zu vermeiden.
Verpflichtend dazu ist das Vorhandensein einer bestehenden *Internetverbindung*. Andernfalls kann das System nicht verwendet werden, da der Zugriff auf die Cloud nur über das Internet möglich ist.
Benutzer können über den *Browser* des Computers auf das System zugreifen und die Services des proCollab-Prototypen nutzen. Die Software muss dazu nicht lokal auf dem eigenen Computer vorhanden sein, sondern kann von einem beliebigen Computer mit einer verfügbaren Internetverbindung gestartet werden.
Mobile Endgeräte transportieren die Cloud-Anwendung an jeden geografischen Ort. Das System muss nicht über den Browser gestartet werden, sondern steht dem Nutzer

als Webapplikation auf dem Smartphone und Tablet bereit.
Das *Interaktive Web* ist Voraussetzung für eine zeitgemäße Darstellung des Cloud-Systems und für eine intuitive Benutzbarkeit des Systems. Dadurch können Informationen nicht nur angefragt werden, vielmehr können CLIs durch interaktive Funktionen beliebig zusammen geklickt werden.

4.3.2 Private Cloud

Der proCollab-Prototyp soll als *Private Cloud* für Anwender nutzbar sein. Dabei steht der proCollab-Prototyp als eigenständige alleinoperierende Cloud-Umgebung dem Unternehmen bereit. Da zum jetzigen Zeitpunkt eine Integration mit anderen Systemen nicht geplant ist, können Aspekte die die Datensicherheit und -zugriffsrechte zwischen mehreren Unternehmen koordinieren, vernachlässigt werden.
Private Clouds sind zugangsbeschränkt [BIT09]. Der proCollab-Prototyp steht nur dem jeweiligen Unternehmen zur Verfügung. Autorisierte Lieferanten oder Geschäftspartner einer Organisation können zusätzlich Zugang erhalten.
Hingegen der üblichen Konvention von Private Clouds, findet der Zugriff auf das System über das Internet statt. Häufig werden Private Clouds innerhalb eines Unternehmens über das Intranet angesprochen. Da Checklisten und Projekte jederzeit modifiziert werden können sollen, muss daher sichergestellt werden, dass eine Internetverbindung vorhanden ist. Für die Verwendung von mobilen Endgeräten sollte eine Zugriffsmöglichkeit über ein drahtloses oder ein mobiles Netz sichergestellt werden.

4.3.3 Platform as a Service

Cloud-Computing bietet generell die Möglichkeit zur Realisierung unterschiedlicher Ausprägungen. Der proCollab-Prototyp orientiert sich dabei an dem Servicemodell *Platform as a Service* (PaaS).
PaaS bezeichnet eine Cloud-Umgebung auf der die Anwendung angesiedelt wird, um später ausgeführt werden zu können. Zur Laufzeitumgebung steht eine Weboberfläche zum Bedienen der Anwendung bereit.

PaaS bietet die Möglichkeit, benutzerdefinierte Services in der Cloud in Betrieb zu nehmen. Die Konsumenten, müssen sich dabei nicht um die darunter liegende Infrastruktur kümmern. Damit die Services der Anwendungsarchitektur modular nutzbar werden, ist das Service-Prinzip der losen Kopplung einer SOA (siehe Kapitel 4.2.1) eine grundlegende Erfordernis. Erst dadurch werden die Dienste zur Nutzung in Geschäftsprozessen interoperabel und kombinierbar [BIT09].

4.4 System-Architektur

Die Architektur des proCollab-Prototypen ist in Schichten aufgebaut und basiert auf dem sogenannten *Model-View-Controller* Muster (*MVC*). Das MVC Muster bietet die Möglichkeit, das zu entwickelnde System auf modulare Art und Weise zu strukturieren [Buc09]. Folgende Abbildung 4.14 zeigt die System-Architektur des zu entwickelnden proCollab-Prototypen:

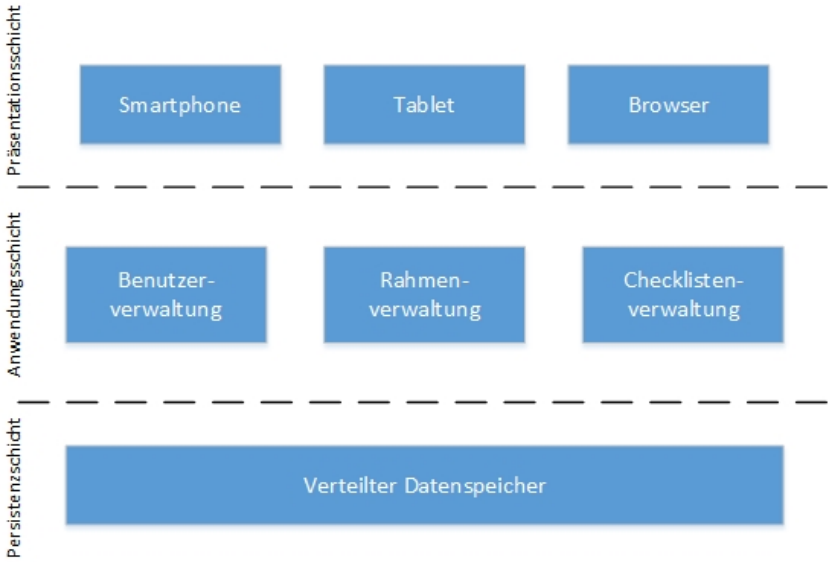


Abbildung 4.14: System-Architektur des Checklisten-Management-Systems

In dem zu entwickelnden proCollab-Prototypen ist es wichtig, eine einheitliche Interaktion mit der Präsentationsschicht (oder auch *Client-Schicht*) herzustellen. Der proCollab-Prototyp umfasst neben den drei Schichten (im Allgemeinen: Persistenzschicht, Anwendungsschicht und Präsentationsschicht) eine zusätzliche vierte Schicht, die den direkten Datenaustausch zwischen Client-Schicht und Controller steuert und in dem oben dargestellten Schaubild abstrahiert wurde. Dennoch spielt diese Schicht eine sehr große Rolle, da sie die Schnittstelle zu den Endgeräten darstellt. Diese Schicht wird dabei als Übertragungsschicht bezeichnet und befindet sich zwischen der Präsentations- und der Anwendungsschicht. Hinsichtlich dieser Schicht sind mehrere Technologien relevant [Red13c, Ado13, Vaa12].
Jede einzelne Schicht ist von den anderen streng gekapselt, es besteht eine klare Trennung zwischen den jeweiligen Komponenten. Das verschafft den Vorteil, dass jede einzelne Komponente unabhängig von den anderen entwickelt werden kann. Nichtsdestotrotz ist es unerlässlich einheitliche Schnittstellen zu definieren, die zur Kopplung der einzelnen Schichten führen. Der Aufbau der Architektur ist wie folgt realisiert:

4.4.1 Präsentationsschicht

Auf oberster Ebene befindet sich die Präsentationsschicht, analog zu der *View* des MVC Musters. Diese Schicht ist für die Interaktion mit dem Benutzer zuständig. Dabei wird auf Aspekte der Benutzerfreundlichkeit näher eingegangen, indem die grafische Benutzeroberfläche durch geeignete Wahl von Layout, Eingabeformulare und Farben entwickelt wird. Damit der proCollab-Prototyp für den Benutzer maximal zugänglich wird, verwaltet die Client-Schicht mehrere Präsentationen. Es ist zu unterscheiden zwischen der Anwendung für das Smartphone [Gei13], das Tablet [Köl13] und den Browser [Thi13]. Während das Smartphone und das Tablet eine touchbasierte Benutzereingabe besitzen, verwendet der Browser standardmäßig die Eingabe über Tastatur und Maus. In dieser Hinsicht müssen verschiedene Geräte auf unterschiedliche Eingabeereignisse abgestimmt sein, damit die Benutzerinteraktionen mit dem System adäquat umgesetzt werden können.

4.4.2 Anwendungsschicht

Die Client-Schicht steht in engem Kontakt mit der *Anwendungsschicht* (in MVC: *Controller*) und leitet Formulardaten aus Eingabefeldern an diese weiter. Dazu werden Eingabeereignisse, welche durch Benutzerinteraktionen entstehen, konvertiert und zur Ausführung an die pCSK übergeben. Die pCSK nimmt Aktionen, die von Eingabegeräten wie Maus, Tastatur und Touchpad entstehen, entgegen und veranlasst deren Verarbeitung und Ausführung. Die pCSK ist dabei für die Implementierung sämtlicher Funktionalität zuständig.
Der Umfang dieser Arbeit umfasst die Bereitstellung der pCSK welche zusätzlich Schnittstellen für das Backend und das Frontend bereitstellt. Folgende Abbildung 4.15 zeigt das Zusammenspiel der jeweiligen Komponenten:

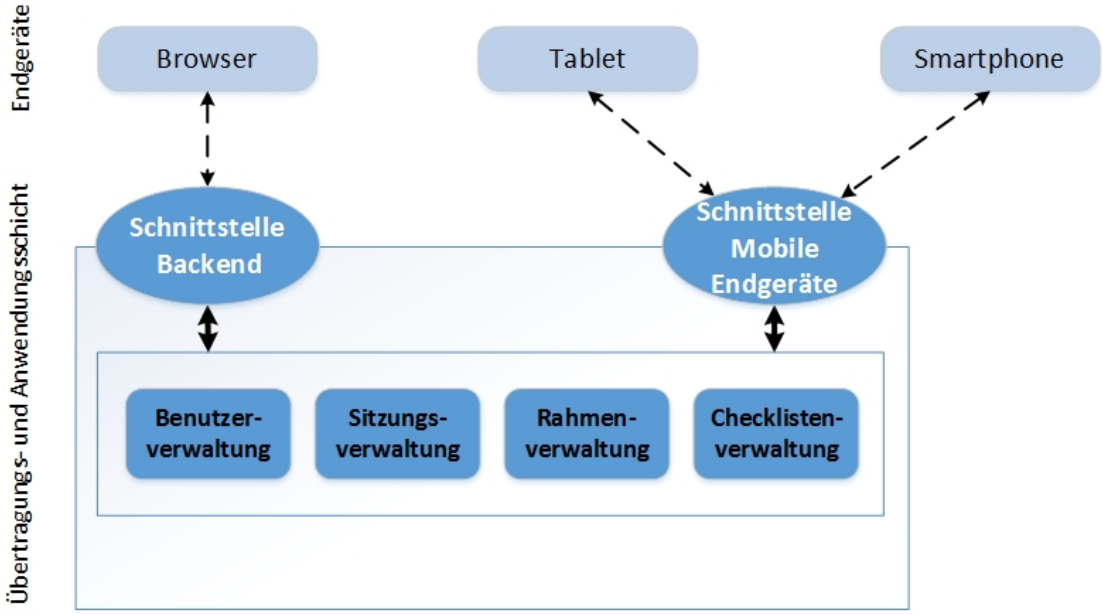


Abbildung 4.15: Teilausschnitt aus der System-Architektur

Als Teil dieser Arbeit werden die in der Abbildung 4.15 definierten Teilkomponenten implementiert. Dies beinhaltet die Umsetzung der *Benutzerverwaltung*, die geeignete Funk-

tionalität bereitstellt, um die Personenverwaltung innerhalb des proCollab-Prototypen adäquat umsetzen zu können. In dieser Hinsicht wird zusätzlich Funktionalität zur Sitzungsverwaltung benötigt. Eng damit gekoppelt sind die Methoden zur Zugriffsteuerung. Die pCSK stellt Funktionen zur Überprüfung von Rechten eines Benutzers bereit und prüft dies anhand definierten Rollen.
Die Komponente *Rahmenverwaltung* ist dafür verantwortlich, ORTs und ORIs im proCollab-Prototyp verwalten zu können. ORs werden durch ORIs und ORTs organisiert, welche von verantwortlichen Personen erstellt wurden.
Die umfassendste Komponente stellt die *Checklistenverwaltung* dar. Diese verwaltet sämtliche Funktionalität von CLs und CFs. CLs sind in einer Blockstruktur aufgebaut und werden daher intern als Baum repräsentiert. Warum sich genau diese Darstellungsform für Checklisten eignet, wird in Kapitel 4.1.3 näher beschrieben.
Die pCSK steht den Applikationen (Browser, Tablet, Smartphone) in der Prästentationsschicht über die Übertragungsschicht anhand definierter Schnittstellen zur Verfügung.

4.4.3 Persistenzschicht

Auf unterster Ebene befindet sich die *Persistenzschicht*. Dabei handelt es sich um eine Schicht zur Persistierung von Daten. Die Persistenzschicht des proCollab-Prototypen soll zukünftig auf verschiedenen Servern verteilt Daten anbieten und sichern können. Greifen Nutzer von unterschiedlichen Ländern auf den proCollab-Prototypen zu, steigert ein verteilter Datenspeicher die Performanz der Endnutzer, da die Datenlast auf mehrere Server verteilt werden kann und vorzugsweise der Server selektiert wird, der dem spezifischen Nutzer örtlich nahegelegen ist. Die Persistenzschicht nimmt dabei eine passive Rolle ein, das heißt, dass sie keine Anwendungsfunktion oder -logik enthält, sondern lediglich Klassen und Methoden, die den Zugriff auf das Datenmodell implementieren. Dabei handelt es sich um reine Datenmanipulationsfunktionen.

5

Implementierung

In Bezug auf die Implementierung der pCSK des proCollab-Prototypen werden verwendete Architekturmuster und Frameworks vorgestellt. Um Services adäquat dem Service-Konsumenten anbieten zu können, nutzt der proCollab-Prototyp ein REST[1] Framework. Es werden Standards zu REST Frameworks verglichen und aufgezeigt, warum sich JAX-RS zur Nutzung eignet. Kapitel 5.3 konkretisiert die Implementierung des proCollab-Prototypen und zeigt auf, wie einzelne Komponenten aufgebaut und umgesetzt sind.

5.1 Representational State Transfer

Zur Entwicklung der Services des proCollab-Prototypen bietet *REST* einen Lösungsansatz, der auf dem bekannten Anwendungsprotokoll HTTP[2] einer Webarchitektur basiert [FGM+99].

[1] Representational State Transfer
[2] Hypertext Transfer Protocol

REST beschreibt ein Programmierparadigma, das im Rahmen von Roy Fieldings Dissertation im Jahre 2000 entstanden ist [Fie00]. REST legt dabei den Schwerpunkt auf Kernprinzipien, anstatt eine konkrete Syntax festzulegen [Til11].
Ein REST-konformer Service zeichnet sich durch eine *eindeutige Identifikation* der *Ressourcen* (siehe Kapitel 5.1.1) mit dem Identifikationsmechanismus *URI*[3] aus. REST verwendet des Weiteren die *HTTP-Methoden*, wie sie in Kapitel 5.1.3 beschrieben werden, um Ressourcen anzusprechen. Ressourcen können dabei in unterschiedlichen *Repräsentationen* dargestellt werden. Welches Repräsentationsformat der proCollab-Prototyp verwendet, wird in Kapitel 5.1.2 genauer beschrieben. Zuletzt legt das Prinzip der statuslosen Kommunikation fest, dass der Zustand nicht serverseitig gehalten werden soll [Fie00]. Der Zustand soll also entweder von dem Service-Konsument aufrechterhalten werden oder als Ressourcenstatus zur Verfügung stehen. Der Grund für das Vermeiden eines Sitzungsstatus ist neben der Skalierbarkeit das Sicherstellen einer losen Kopplung (siehe Kapitel 4.2.1). Dadurch wird die Abhängigkeit zwischen Client und Server verringert.

5.1.1 Ressourcen

Jede Information in REST wird durch eine sogenannte *Ressource* dargestellt. Sie stehen daher im Mittelpunkt beim Entwurf von RESTful Webservices und bilden die Schnittstellen zur Funktionalität ab. Ressourcen können damit im proCollab-Prototypen eine Person oder eine ORI sein. Der konkrete Ressourcenentwurf beinhaltet im Speziellen die Schnittstellen nach Außen. Dabei werden die Methoden direkt auf die verfügbaren HTTP Methoden abgebildet [Til11].
Eine Ressource ist von zwei Haupteigenschaften geprägt: Sie ist *eindeutig identifizierbar* und kann in Form *mehrerer Repräsentationen* zur Verfügung stehen. Das Kapitel 5.1.2 spezifiziert dabei das verwendete Repräsentationsformat innerhalb des proCollab-Prototypen.
Durch das einheitliche Konzept der Vergabe von URI's wird gewährleistet, dass jede

[3] Uniform Resource Identifier

Ressource wieder eindeutig identifiziert werden kann [Til11]. Folgendes Beispiel zeigt die Struktur einer URI:

http://web2.dbis.info:8080/proCollabPrototype/rest/person/1234

In diesem Beispiel wird eine Person referenziert, die mittels dem definierten Schema und der am Ende stehenden ID eindeutig identifiziert werden kann. Dabei identifiziert eine URI eine Ressource stets eindeutig. Umgekehrt ist das aber nicht zwingend der Fall: Eine Ressource kann anhand mehreren ID's identifiziert werden.
In diesem Kontext kann daher zwischen mehreren Ressourcenkategorien unterschieden werden. Die drei im Folgenden genannten Kategorien bilden den Großteil aller verwendeten Ressourcen ab [Til11].

Primärressourcen

Primärressourcen beschreiben meist persistente Entitäten, die sich in der Entwicklung statisch verhalten. Sie verändern sich kaum und die Implementierung ist für den Aufrufenden einer Ressource, beispielsweise der Webbrowser, vollkommen transparent. Obwohl sie als Element der Schnittstelle identifizierbar sind, ist kein Rückschluss auf Implementierungsdetails möglich. Es ist zu beachten, dass Ressourcen nicht mit Datensätzen in einer Datenbank vergleichbar sind. Vielmehr bilden Ressourcen typischerweise komplexe Beziehungen zwischen Datensätzen im Hintergrund. Exemplarisch stellt eine Primärressource im proCollab-Prototyp eine Person dar. Mittels der HTTP GET Methode können nähere Informationen zu einer Person angezeigt werden, während mit der HTTP PUT Methode Änderungen an der Primärressource Person vorgenommen werden können. Im Folgenden werden einige wichtige Primärressourcen des Prototypen dargestellt, welche hier in der URI-Darstellung abgebildet sind:

http://web2.dbis.info:8080/proCollabPrototype/rest/person
http://web2.dbis.info:8080/proCollabPrototype/rest/frame
http://web2.dbis.info:8080/proCollabPrototype/rest/checklist
http://web2.dbis.info:8080/proCollabPrototype/rest/item

Subressourcen

Subressourcen sind Ressourcen, die Bestandteil einer anderen Ressource sind. Dies können beispielsweise individuelle Personen der Primärressource Person sein oder eine Liste von Personen, die zu einer bestimmten Organisation gehören. Auch Subressourcen sind eindeutig per URI identifizierbar und können mehrere Repräsentationen besitzen.

Listenressource

Jede Primärressource tritt meist in Verbindung mit einer oder mehreren *Listenressourcen* auf. Eine Listenressource ist zum Beispiel eine Liste von zugehörigen Personen zu einer ORI. Wird auf die Liste von zugehörigen Personen zu einer ORI eine HTTP GET Anfrage erfolgreich ausgeführt, erhält man die angeforderte Listenrespräsentation.

5.1.2 Repräsentationsformat JSON

Damit ein Service-Konsument Anwendungsdaten verarbeiten kann, muss jede Ressource mindestens eine oder mehrere Repräsentationen zur Verfügung stellen [Til11]. Ein Service-Konsument kann nur mit einer Ressource kommunizieren, wenn er die Repräsentation, die die Ressource anbietet, verarbeiten kann. Repräsentationsformate einer Ressource sind üblicherweise: XML[4], HTML[5] oder JSON[6].
Da die mobilen Applikationen als Frontend des proCollab-Prototypen in JavaScript implementiert sind, wird JSON als Repräsentationsformat verwendet. JSON wird inzwischen von fast jeder Programmierumgebung unterstützt und ist ein vollständig sprachenunabhängiges Format in Textform, das in den Grundzügen dem XML Repräsentationsformat ähnelt [RR07].
Die zwei grundsätzlichen Ideen von JSON sind, dass jedes Element aus einem Namen-Werte-Paar besteht und Daten beliebig geschachtelt werden können. JSON ist in der Lage vier primitive Datentypen (Strings, Zahlen, boolesche Werte, den Nullwert *Null*) sowie zwei Datenstrukturen (Objekte, Arrays) darzustellen [JSO13]. Dies zeigt folgendes Beispiel in Listing 5.1.

[4] Extensible Markup Language
[5] Hypertext Markup Language
[6] JavaScript Object Notation

```
{
"name":"Projekt Automobilprozess",
"description":"Herstellung von Kraftfahrzeugen",
"state":"initialized",
"containLists":
[
        {
        "name":"Checkliste Automobilprozess",
        "state":"open",
        "children"[],
        "id":32768
        }
],
"startDate":"2013-05-16",
"endDate":"2017-11-14",
"id" : 30000
}
```

Listing 5.1: Repräsentation einer ORI als JSON String

5.1.3 Standardisierte HTTP-Methoden

Die Kernidee für RESTful Webservices ist es, dass jede Operation dieselben HTTP-Methoden unterstützt und damit für alle Ressourcen in gleichem Maße gültig sind. Da die Methoden bereits standardisiert und optimiert sind, bieten sie einen entscheidenden Vorteil in der Performanz, da durch deren Verwendung eine zusätzliche Schicht eingespart werden kann. Durch die Ausprägung von REST unter Verwendung von HTTP haben sich folgende meist gebrauchte HTTP-Methoden herauskristallisiert: GET, POST, PUT und DELETE [RR07].

5.1.3.1 GET

Die wichtigste und meist verwendete Methode ist die Methode *GET* [Til11]. Sie ist für den Service-Konsument eine ausschließlich lesende Operation und dient dazu Informationen oder Dokumente, die durch eine URI bereitgestellt werden, in Ausprägung einer Repräsentation abzuholen. Im Folgenden wird eine HTTP-GET Anfrage dargestellt, wie sie bei dem Anfragen einer Person aussieht.

GET http://web2.dbis.info:8080/proCollabPrototype/rest/person/7

Die darauffolgende HTTP-Antwortnachricht enthält das Repräsentationsformat JSON und ist folgendermaßen aufgebaut:

---

200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Transfer-Encoding: chunked
Date: Thu, 10 Oct 2013 10:08:08 GMT

{“prename“ :“Tom“,“surname“:“Maier“,“email“:“Tom.Maier@test.de“,“id“:7}

Der obere Teil der HTTP-Antwortnachricht bezeichnet dabei die HTTP-Metadaten, während der untere Teil die Nutzdaten in JSON Format liefert.

5.1.3.2 PUT

Über die Methode *PUT* wird eine bereits vorhandene Ressource aktualisiert beziehungsweise erzeugt, falls sie nicht verfügbar ist. PUT wirkt sich unmittelbar auf die Ressource selbst aus und wird damit zur inversen Operation von GET.
PUT besitzt wie die Methoden GET und DELETE die Eigenschaft der Idempotenz. Die Definition einer idempotenten Operation besagt, dass die Seiteneffekte von mehreren identischen Aufrufen dieselben sind, wie der einmalige Aufruf der Operation [Til11]. Dies stellt sicher, dass im Falle von verloren gegangenen oder nicht angekommenen Anfragen

der Service-Konsument die Möglichkeit besitzt, die Anfrage erneut zu versenden. Diese bereitgestellte Garantie darf nur bei idempotenten Funktionen angewendet werden und ist folglich bei der Methode POST nicht erlaubt.

5.1.3.3 POST

Die Methode *POST* wird verwendet, um eine neue Ressource mit einer eindeutig identifizierbaren URI anzulegen. Dabei wird jene zuständige Ressource aufgerufen, die für das Erzeugen einer neuen Ressource verantwortlich ist. Oftmals handelt es sich um Listenressourcen (siehe Kapitel 5.1.1), die um ein zusätzliches Element erweitert werden. Neben der eigentlichen Bedeutung kann POST zusätzlich immer genau dann angewendet werden, wenn andere Methoden nicht geeignet sind die Verarbeitung zu gewährleisten, da POST keine Garantien bezüglich Sicherheit unterstützt. Des Weiteren ist POST nicht idempotent, da jede einzelne POST Anfrage den Service individuell modifiziert.

5.1.3.4 DELETE

Die Methode *DELETE* fordert die pCSK auf, die Ressource, die durch die Anfrage-URI identifiziert wird, zu löschen. Die Methode ist idempotent, da das mehrmalige Löschen keine weiteren Auswirkungen hat als das einmalige. Dabei stellt die Methode DELETE ein logisches Löschen dar. Dem Service-Konsument kann daher nicht garantiert werden, dass innerhalb der Persistenzschicht die Ressource nicht mehr existiert oder lediglich nur mit einem „gelöscht“ Attribut versehen wurde. Dies spielt für den Service-Konsument keine Rolle. Es sollte jedoch sichergestellt werden, dass der Server keine erfolgreiche Ausführung bezeugt, solange die Ressource noch für den Service-Konsument zugreifbar ist.

5.2 Verwendete Technologien

Zur in Kapitel 5.3 dokumentierten Umsetzung der pCSK ist ein grundlegendes Verständnis über die verwendeten Technologien und Frameworks erforderlich. Im Folgenden wird hierbei die Java Enterprise Edition sowie der JBoss Anwendungsserver näher erläutert. Des Weiteren wird ein Vergleich der vorhandenen JAX-RS Frameworks aufgeführt.

5.2.1 Java Enterprise Edition (JEE)

Der proCollab-Prototyp basiert auf der Plattform *Java Enterprise Edition* (*Java EE*) und nutzt einen Anwendungsserver von *JBoss*, der in Kapitel 5.2.2 illustriert wird.
Die Java EE ist eine Spezifikation zur beschleunigten Entwicklung von Webanwendungen auf Java Basis und setzt mehrere Schichten voraus, die weitgehend eigenständig voneinander sind. Diese trennen das System klar in verschiedene Aufgabenbereiche. Es wird von einer *Mehrschichtenanwendung* gesprochen [Sta10].
Java EE spezifiziert für die erleichterte Entwicklung eine Vielzahl von Funktionalitäten, die über wohldefinierte Schnittstellen bereitgestellt werden. Folgende Abbildung 5.1 zeigt die Architektur der Java EE inklusive ihrer vielzahligen Komponenten.

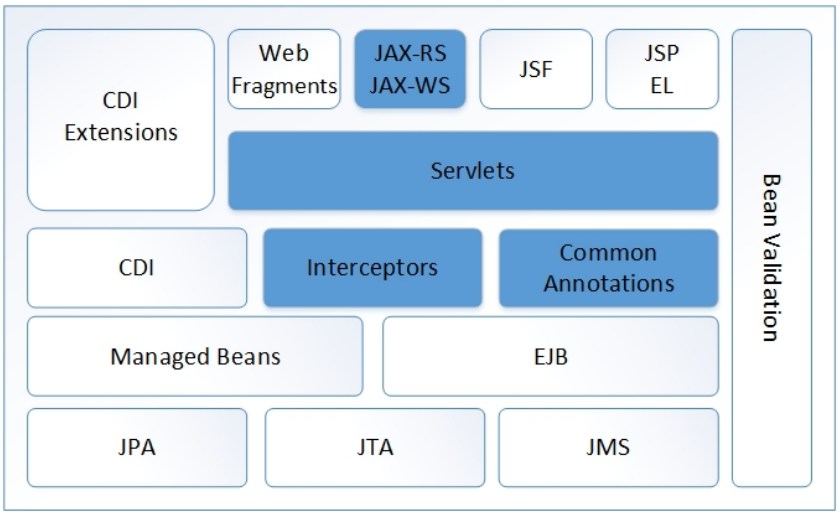


Abbildung 5.1: Architektur der Java Enterprise Edition nach [Gup12]

Wie in Abbildung 5.1 farblich dargestellt, werden für die Entwicklung der Übertragungs- und Anwendungsschicht des proCollab-Prototypen das Servlet-Modell (*Servlets*), die Annotationen (*Common Annotations*), die Komponente für RESTful Webservices (*JAX-RS*[7]) sowie die Interceptoren genutzt.

Die Persistenzschicht des proCollab-Prototypen, die Schnittstellen zur Persistierung bereitstellt, nutzt die grundlegenden Dienste der *Java Persistence API* (*JPA*) um Datenbankzugriffe und Transaktionssicherheit zu gewährleisten.

Das Frontend, das sich um die Darstellung für die Benutzer kümmert, setzt auf Web-Frameworks wie Vaadin und PhoneGap auf und arbeitet weitgehend unabhängig von der von Java EE bereitgestellten Funktionalität.

Java EE, im Speziellen JAX-RS, bietet eine intuitive Anwendung, da anstatt Konfigurationen größtenteils *Annotationen* für Java-Objekte festgelegt sind [Gup12]. Soll beispielsweise ein Java-Objekt als REST-Ressource veröffentlicht werden, muss die jeweilige Methode mit der Annotation „*@Path*“ versehen werden. Folgendes Codefragement in Listing 5.2 zeigt die Verwendung der Annotation @Path.

```
1 @Path("/{id}")
2 public Response get_Person(@PathParam("id") int id){
3 // code for requesting and delivering a specific person
4 }
```

Listing 5.2: Veröffentlichen einer REST-Ressource mittels der Annotation *@Path*

Um die ID der anzufragenden Person aus der URI in die Methode zu injizieren, wird zusätzlich „*@PathParam*“ verwendet. Der Wert *id* steht innerhalb der Methode zur Verwendung bereit.

[7] Java API for RESTful Web Services

5.2.2 JBoss Anwendungsserver

Der Anwendungsserver von *JBoss* ist nach dem Java EE-Standard implementiert und ist daher entsprechend als Grundlage für den ebenfalls nach dem Java EE-Standard entwickelten proCollab-Prototyp nutzbar [Red13a]. Für die Entwicklung des proCollab-Prototypen wurde konkret der JBoss Anwendungsserver 7.1 genutzt, der frei verfügbar und nutzbar ist.
Der JBoss Anwendungsserver ist für die Ausführung des proCollab-Prototypen zuständig und bietet zudem eine cloud-fähige Architektur an. Dadurch kann die durch die Konzeption festgelegte und in Kapitel 4.3 beschriebene Cloud-Fähigkeit zukünftig umgesetzt werden.
Des Weiteren ist die Verwendung des JBoss Anwendungsservers ist für den proCollab-Prototypen geeignet, da er eine Menge von Frameworks unterstützt, die für den proCollab-Prototypen relevant sind. Dazu gehören beispielsweise die angewandten Frameworks wie *RESTeasy* [Red13c] oder *Hibernate*. RESTeasy (siehe Kapitel 5.2.3.2) ist ein von JBoss bereitgestelltes Framework zur Entwicklung von RESTful Webservices und ist bereits im Anwendungsserver integriert und kann daher ohne zusätzliche Installation verwendet werden.

5.2.3 Vergleich REST-Frameworks

JAX-RS ist eine Programmierschnittstelle, die die Implementierung von RESTful Webservices (siehe Kapitel 5.1) auf Basis von Java realisiert. Die JAX-RS Spezifikation ist Teil der Java EE Version 6 und bietet eine Menge von Referenzimplementierungen an [Gup12]. Die Kapitel 5.2.3.1, 5.2.3.2 und 5.2.3.3 stellen drei wesentliche Referenzimplementierungen vor und bieten einen Vergleich an.

5.2.3.1 Restlet

Restlet ist ein REST-Framework, das zeitlich vor JAX-RS entwickelt wurde und steht als frei verfügbare Programmierschnittstelle zur Entwicklung von RESTful Webservices

bereit [Res13]. Das Framework bietet dieselbe einheitliche Schnittstelle für den Client als auch den Server an, wobei Restlet die nativen REST-Prinzipien (Ressourcen, Repräsentationen, URI, standardisierte HTTP Methoden) unterstützt.
Restlet kann alleinstehende Java Applikationen betreiben oder durch die Integration in Java EE innerhalb eines Servlet Containers ausgeführt werden [Sri12].
Restlet bietet eine Edition für das *Google Webtool Kit* (*GWT*) an, um die Schnittstelle ohne spezielle Erweiterungen in einem Browser laufen lassen zu können. Das Framework stellt des Weiteren eine verfügbare Edition für die PaaS *Google App Engine* (*GAE*) - zum in Betrieb nehmen auf einer Cloud Plattform - bereit, sowie einer Edition für Android, um Restlet Anwendungen auf einem mobilen Endgerät zu betreiben [Res13].
Restlet ist flexibel in der Konfiguration und kann über die Restlet API vollständig in Java eingerichtet werden. Durch die Definition von geeigneten Java Klassen können beliebige Methoden für den Server geschrieben werden.
Das Framework bietet mehrere Mechanismen zur Einhaltung von Sicherheit. Es unterstützt HTTP Basic und Digest, Amazon S3 und SMTP zur Authentifizierung und Autorisierung.

5.2.3.2 RESTEasy

RESTEasy ist die frei verfügbare Referenzimplementierung von JBoss und unterstützt das Erstellen von RESTful Webservices mit JAX-RS [Red13c]. Das Framework kann in jedem Servlet Container betrieben werden. RESTEasy unterstützt eine starke Integration mit dem Anwendungsserver von JBoss und erleichtert dadurch die Entwicklung, da Server als auch Framework von JBoss stammen. RESTEasy stellt eine zusätzliche Client Programmierschnittstelle bereit damit ein HTTP Client dieselben JAX-RS Annotationen verwenden kann. Das Framework unterstützt die Verwendung unterschiedlicher Repräsentationen wie XML oder JSON.
Wird der Anwendungsserver von JBoss verwendet, bringt dieser bereits die RESTEasy Implementierung mit. Zur Ausführung wird ein Controller benötigt, der die Klasse *javax.ws.rs.core.Application* erweitert [Red13b]. Diese Klasse muss alle Ressourcen bein-

halten, damit sie für den Client aufrufbar sind.
RESTEasy stellt ein erweitertes *Interceptoren-Modell* zur Verfügung. *Interceptoren* können vor oder nach dem Aufruf einer Ressource zwischengeschaltet werden und leisten beispielsweise als *PreProcessInterceptor* ihren Beitrag zur Authentifizierung [Bur10]. RESTEasy stellt zusätzlich *PostProcessInterceptoren* zur Verfügung, welche serverseitig angewandt werden und jede Serverantwort filtern bevor sie weitergeleitet werden. Häufig werden sie dazu verwendet, um innerhalb der Antwortnachricht den HTTP-Header zu modifizieren. Ein Interceptor wird dazu mit „*@Provider*“ und „*@ServerInterceptor*“ annotiert [Red13b]. Folgendes Listing 5.3 zeigt die Verwendung eines PostProcessInterceptors.

```
1  @Provider
2  @ServerInterceptor
3  public class Interceptor implements PostProcessInterceptor{
4    @Override
5    public void postProcess(ServerResponse response) {
6      MultivaluedMap<String,Object> header=response.getMetadata();
7
8      header.putSingle("Access-Control-Allow-Credentials", true);
9          header.putSingle("Access-Control-Allow-Methods", "*");
10         header.putSingle("Access-Control-Allow-Origin", "*");
11         header.putSingle("Access-Control-Allow-Headers", "*");
12
13 }
```

Listing 5.3: Implementierung eines *PostProcessInterceptors*

In diesem Beispiel liest der Interceptor den HTTP-Header der Serverantwort aus und erweitert diesen um vier weitere Felder. Jeder Aufruf einer JAX-RS Methode führt automatisch zum Aufruf des Interceptors und zum Setzen der erforderlichen Felder.

5.2.3.3 Jersey

Das REST-Framework *Jersey* ist eine frei verfügbare Referenzimplementierung von SUN und steht als portable JAX-RS Schnittstelle zur Entwicklung bereit [Ora13]. Jersey bietet, wie auch Restlet und RESTEasy, neben der serverseitigen Integration die Möglichkeit, eine Client-seitige Schnittstelle zu verwenden.
Eine neue Webapplikation kann in einem Servlet Container oder durch eine Java Applikation in Betrieb genommen werden. Jersey lässt die Verwendung von verschiedenen Repräsentationen (JSON, XML) zu [Sri12]. Das Framework unterstützt WADL, eine Beschreibungssprache für Webanwendungen in XML-Format. Diese kann sowohl für die Konfiguration, Code-Generation und Dokumentation verwendet werden [Bur10].

Alle drei vorgestellten Frameworks unterscheiden sich in Bezug auf die Anforderungen des proCollab-Prototypen nicht signifikant voneinander. Es ist daher abzuwägen, welche Zusatzfunktionen das jeweilige Framework besitzt und damit für das System vorteilhaft sind. Es ist ersichtlich, dass Restlet eine Vielzahl an verfügbaren Editionen (GWT, GAE, Android) unterstützt und damit ein mächtiges Werkzeug zur Umsetzung von RESTful Webservices ist. Auch die Referenzimplementierung Jersey stellt sehr umfangreiche Funktionalitäten zur Verfügung.
JBoss hingegen hat das Framework RESTEasy derart entwickelt, dass die Kopplung mit dem JBoss Anwendungsserver geschickt ausgenutzt werden kann. Wie in Kapitel 5.2.2 beschrieben, verwendet der proCollab-Prototyp zum in Betrieb nehmen den JBoss Anwendungsserver. Daher bietet sich die Kombination von JBoss Anwendungsserver und RESTEasy an. Außerdem ist die Möglichkeit JAX-RS Aufrufe, durch die von RESTEasy zusätzlich bereitgestellten Interceptoren, abzufangen, ein komfortables Werkzeug, um die Ressourcen vor nicht-autorisierten Zugriffen zu schützen.

5.3 Umsetzung der Komponenten

Die Implementierung der Anwendungsschicht ist modular strukturiert und wird im folgenden näher skizziert. Dabei werden die einzelnen Komponenten der Benutzer-, Sitzungs-, Rahmen- und Checklistenverwaltung detailliert (siehe Kapitel 5.3.2, 5.3.3, 5.3.4 und 5.3.5). Im folgenden Kapitel 5.3.1 wird die Implementierung der internen Baumstruktur spezifiziert.

5.3.1 Implementierung der Baumstruktur

Wie aus Kapitel 4.1.3 hervorgegangen ist, liegen CLs als Blockstruktur vor. Blockstrukturen werden in dem proCollab-Prototypen intern als Baumstruktur repräsentiert. Zusätzlich werden ORs in dem proCollab-Prototyp in einer Baumstruktur verwaltet. In den folgenden Kapiteln 5.3.1 und 5.3.1 werden dabei beide Datenstrukturen skizziert. Sie sind sich in dem Aufbau sehr ähnlich, werden aber dennoch als separate Baumstrukturen implementiert, um dem modularen Aufbau gerecht zu werden.

Baumstruktur für CLs und CFs

Für die Implementierung der Checklistenverwaltung wird eine teils vordefinierte Datenstruktur erweitert, die Bäume und deren Knoten verwaltet [Rou10]. Zur Repräsentation des Baumes wird das Prinzip der Vererbung verwendet.
CLIs und CLTs sowie CFIs und CFTs besitzen die abstrakte Klasse `TreeElement` mit Attributen, die sowohl in Checklisten als auch in Checkfragen benötigt werden. Die abstrakte Klasse `TreeElement` wird durch die beiden abstrakten Klassen `TreeNode` und `TreeLeaf` erweitert. `TreeNode` wird durch die Klassen `CListInstance` (repräsentiert eine CLI) und `CListType` (repräsentiert einen CLT) erweitert. `TreeLeaf` wird durch die Klassen `CItemInstance` (repräsentiert eine CFI) und `CItemType` (repräsentiert einen CFT) erweitert.
Ein `TreeNode` stellt dabei die inneren Knoten (die Checklisten) dar, während ein `TreeLeaf` die Blattknoten (die Checkfragen) darstellt. Ein innerer Knoten kann da-

bei wiederum mehrere innere Knoten oder Blattknoten enthalten. Diese sind in einer *ArrayList* gespeichert, welche im weiteren Verlauf als die *Kindesliste* (*children*) bezeichnet wird. Folgende Abbildung 5.2 zeigt das Datenmodell des n-ären Baumes.

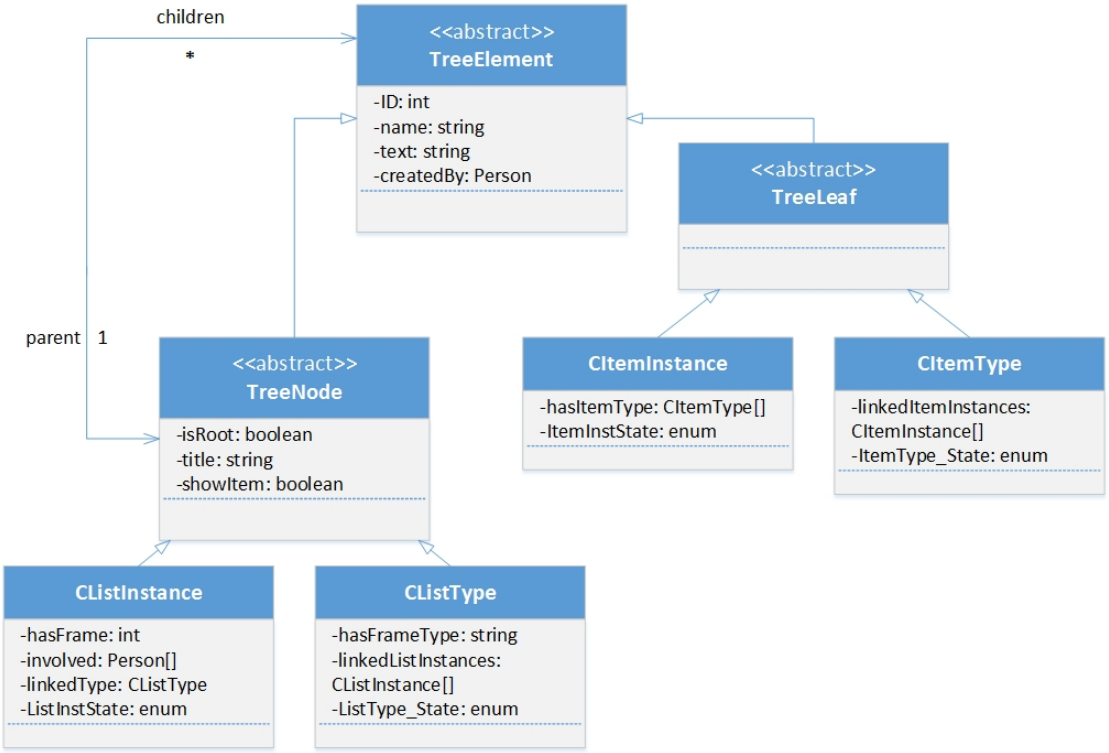


Abbildung 5.2: Repräsentation der Typen und Instanzen von Checklisten und Checkfragen

Da ein Smartphone oder ein Tablet aufgrund ihrer begrenzten Benutzeroberflächen nicht alle Inhalte wiedergeben kann, verwenden die mobilen Endgeräte eine vereinfachte Version des oben beschriebenen Datenmodells. Dieses verbirgt einige Attribute und macht dadurch das Datenmodell schlanker. Die folgende Abbildung 5.3 zeigt die Baumstruktur inklusive Vererbung des vereinfachten Datenmodells.

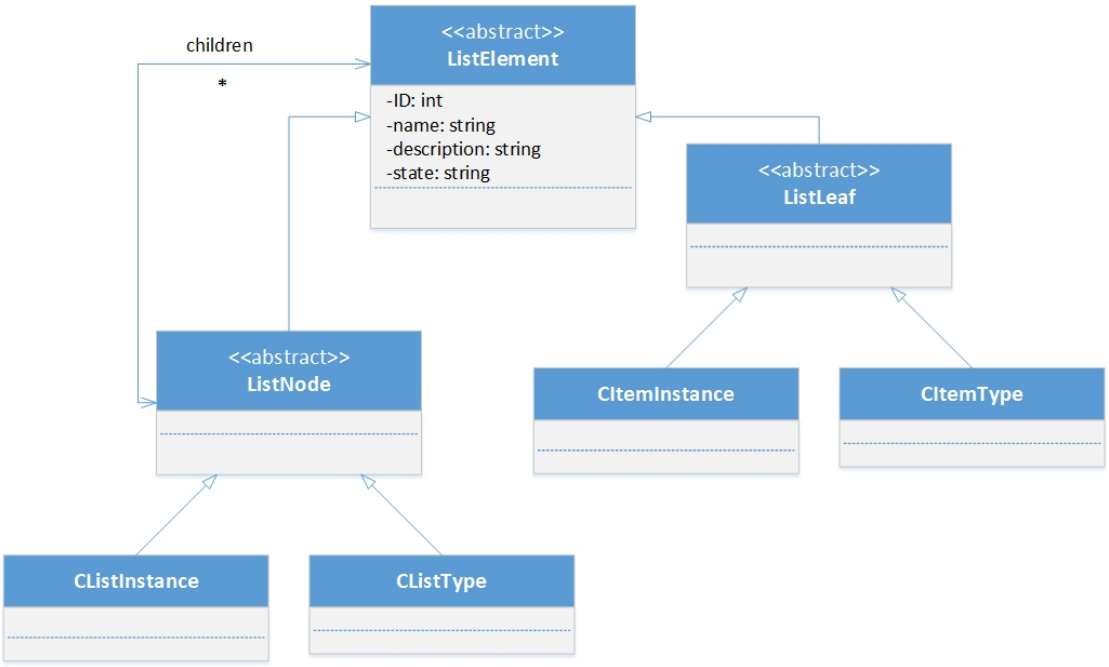


Abbildung 5.3: Vereinfachtes Datenmodell der proCollab-Anwendungen

Die mobilen Anwendungen verwenden dabei ausschließlich das vereinfachte Datenmodell. Da die pCSK die Java-Schnittstelle und die REST-Schnittstelle für die Präsentationsschicht bereitstellt, verfügt diese über die Implementierung beider Datenmodelle. Zum Datenaustausch und der Kommunikation über REST ist eine entsprechende Konvertierung vorzunehmen.
Die folgende Methode *mapItemTypeToMobile* ist Teil der Klasse `EntityMapperToMobile` und überträgt die relevanten Attribute eines CFTs auf das vereinfachte Datenmodell. Dadurch werden der REST-Schnittstelle nur die benötigten Attribute zur Verfügung gestellt. Listing 5.4 zeigt die Zuordnung anhand einem CFT.

```
1 public ItemType mapItemTypeToMobile(CItemType item){
2   ItemType mobileItemType = new ItemType();
3   mobileItemType.setID(item.getID());
4   mobileItemType.setName(item.getName());
5   mobileItemType.setDescription(item.getText());
6   mobileItemType.setState(item.getState());
7   return mobileItemType;
8 }
```

Listing 5.4: Übertragen der Attribute eines CFT auf das vereinfachte Datenmodell

Die Methode *mapItemTypeToMobile* erhält als Übergabeparameter einen CFT, der alle Attribute enthält. Es wird ein neues Objekt des vereinfachten Datenmodells erstellt und dabei nur die ID, der Name, die Beschreibung und der Zustand übernommen. Alle weiteren Attribute sind für die mobilen Anwendungen nicht relevant.
Nimmt der REST-Service eine Anfrage entgegen und erhält dabei das für mobile Anwendungen vereinfachte Datenobjekt des CFT, muss dieses zur Verarbeitung für die pCSK in die ursprüngliche Form konvertiert werden. Dazu werden die fehlenden Attribute wieder ergänzt, damit die pCSK die Baumstruktur verwalten kann. Die inverse Klasse `EntityMapperToServer` stellt dazu die Methode *mapItemTypeToServer* bereit.

CLIs oder CLTs können während der Bearbeitungsdauer stark anwachsen, da diesen kontinuierlich neue Elemente zugeordnet werden können. Wird von dem Frontend ein CLT oder eine CLI angefragt, kann entweder die komplette Checkliste mit allen Kindern zurückgegeben werden oder nur die Checkliste mit der ersten Kindesebene. Dadurch wird der Datenverkehr gering gehalten und Checklisten können bei Bedarf nachgeladen werden. Das Codefragement in Listing 5.5 zeigt einen Ausschnitt aus der Implementierung zum Durchlaufen der Kindesliste und dem Ausblenden der tiefer gelegenen Kinder.

```
1  for (int i = 0; i < children.size(); i++){
2    if (children.get(i) instanceof CListType){
3      newList.add(i, smallMapper.mapSmallListTypeToMobile(
4        (CListType) children.get(i)));
5    }
6    if (children.get(i) instanceof CListInstance){
7      newList.add(i, smallMapper.mapSmallListInstanceToMobile(
8        (CListInstance) children.get(i)));
9    }
10 }
```

Listing 5.5: Entfernen der untergeordneten Kinder

Die Methode *mapSmallListInstanceToMobile* beziehungsweise *mapSmallListTypeToMobile* bewirkt das Entfernen der tiefer gelegenen Kinder an der jeweiligen Position. Diese Funktionalität wird für Checkfragen nicht benötigt, da diese laut Definition keine weiteren Kinder besitzen können. Diese Funktion ist analog für die Rahmenverwaltung implementiert.

Baumstruktur für ORs

Parallel zu der Verwaltung von Checklisten in einer Baumstruktur werden ORs in einer separaten Baumstruktur organisiert. Ein OR kann mehrere untergeordnete ORs und CLs verwalten. Die ORIs und ORTs besitzen eine abstrakte Klasse `FrameElement`, die gemeinsame Attribute bereitstellt. Die erbenden Klassen `CFrameInstance` und `CFrameType` erweitern die abstrakte Klasse um die entsprechenden Attribute und repräsentieren damit semantisch ORIs und ORTs.
Die Baumstruktur verwaltet eine *ArrayList*, die die untergeordneten ORs speichert. Sie wird im Folgenden als *Kindesliste* (*children*) bezeichnet. Sie enthält ausschließlich Elemente vom Typ *FrameElement*.
Eine weitere *ArrayList* (*containLists*) enthält CLs, die einem ORT oder einer ORI zugeordnet sind. Diese enthält nur *TreeNodes*, die eine Wurzel darstellen (*isRoot = true*).

Das folgende Klassendiagramm (siehe Abbildung 5.4) zeigt die Repräsentation von ORTs und ORIs durch eine geeignete Baumstruktur.

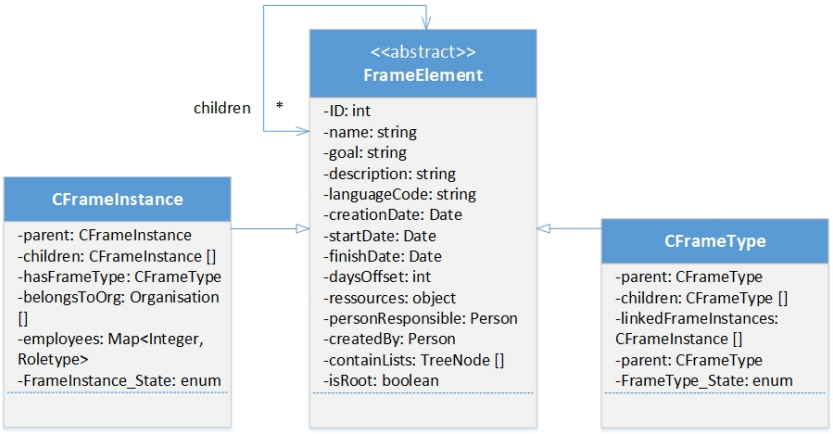


Abbildung 5.4: Repräsentation der ORs anhand eines Baumes

Analog zu den beiden Datenmodellen von CLs (siehe Kapitel 5.3.1) stellt auch das Datenmodell für ORs ein vereinfachtes Datenmodell bereit. Folgende Abbildung 5.5 zeigt das vereinfachte Datenmodell, nach Anwendung einer Funktion aus der Klasse `EntityMapperToMobile`.

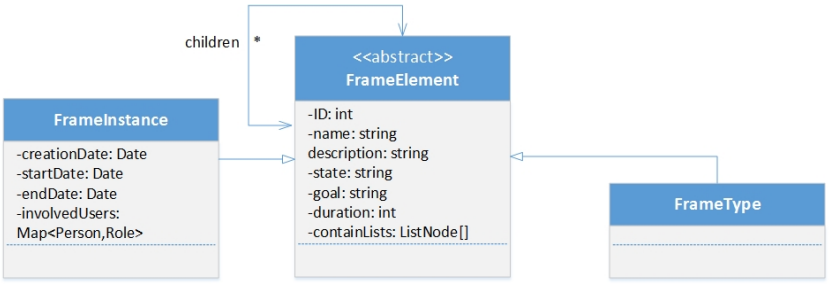


Abbildung 5.5: Vereinfachtes Datenmodell für ORs zur Kommunikation mit den mobilen Endgeräten

5.3.2 Benutzerverwaltung

Einem Benutzer stehen innerhalb des proCollab-Prototypen Funktionen zur Benutzerverwaltung bereit. Initial kann sich der Benutzer registrieren. Dazu steht eine *createPerson*-Methode zur Verfügung, die einen neuen Benutzer in der Datenbank anlegt. Die Attribute werden von Frontend-Anwendungen an die pCSK übergeben. Wird beim Erstellen einer Person eine Organisation innerhalb des JSON-Strings an den entsprechenden REST-Service übergeben, prüft die pCSK ob die Organisation bereits vorhanden ist. Ist dies der Fall, wird die Person der Organisation hinzugefügt, andernfalls wird eine neue Organisation mit den jeweiligen Attributen angelegt.
Benutzer können ihren eigenen Account editieren und Attribute wie den Vornamen oder den Nachnamen ändern. Die ID der zu ändernden Person wird dabei zum einen über die REST-Schnittstelle durch die URI übergeben, zum anderen befindet sich diese zusätzlich in dem übergebenen Personen-Objekt, das die neuen Attribute enthält. Es ist daher zu prüfen, ob diese ID's identisch sind. Das Codefragment in Listing 5.6 zeigt diese Behandlung innerhalb der Methode *update_Person*.

```
1  @PUT
2  @Path("/{id}")
3  public Response update_Person(@CookieParam("sessionID")
4    String sessionID, @PathParam("id") int personID, Person p){
5    // get person by sessionID
6      Person person = authenticationManager
7        .getLoggedPerson(sessionID);
8      if (person == null || !(person.getID() == personID) ||
9        !(personID == p.getID()) {
10       return Response.status(Status.UNAUTHORIZED).build();
11     }
12     // further code for updating a person
13 }
```

Listing 5.6: Aktualisieren einer Person durch die Methode *update_Person*

Stimmen die ID's nicht überein, wird eine HTTP-Antwortnachricht mit dem Statuscode *UNAUTHORIZED* übermittelt. Derselbe Fehlercode wird übergeben, sofern die Person nicht eingeloggt ist. Die Authentifizierung wird in Kapitel 5.3.3 näher erklärt.
Da das Frontend ein vereinfachtes Datenmodell verwendet, ist die Information über erstellte und involvierte ORIs und CLIs in diesem nicht verfügbar. Es müssen separate Methoden implementiert werden, die diese Informationen dem Frontend übermitteln. Um sich beispielsweise eine Liste aller erstellten CLIs zu einer Person anzeigen zu lassen, wird die URI */rest/person/createdLists/12* aufgerufen. Die Zahl 12 ist dabei die ID der jeweiligen Person. Zu der angefragten ID werden die erstellten CLIs ausgelesen, die sich in dem vollständigen Server-Datenmodell befinden. Die CLIs werden dann auf das Frontend-Datenmodell projiziert und als JSON-String versendet.
Zum Beenden der Mitgliedschaft eines Benutzers dient die Methode *delete_Person*. Die Person wird durch die Persistenzschicht entfernt und kann sich infolgedessen nicht mehr am proCollab-Prototyp anmelden.

5.3.3 Sitzungsverwaltung und Authentifizierung

Der proCollab-Prototyp besitzt eine individuelle Sitzungsverwaltung. Zur Umsetzung wird innerhalb des Personen-Objektes ein weiteres Attribut *lastSession* benötigt. Zusätzlich wird die Entität *Session* dem Datenmodell hinzugefügt. In der folgenden Abbildung 5.6 lässt sich die Abhängigkeit zwischen dem Sitzungsobjekt *Session* und dem Personen-Objekt erkennen. Eine Person wird durch eine eindeutige *sessionID* des Sitzungsobjekts referenziert.

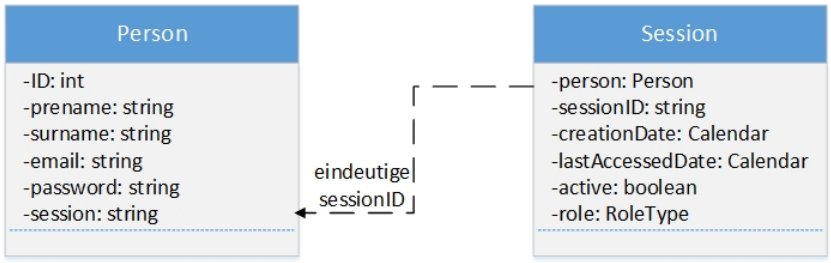


Abbildung 5.6: Abhängigkeitsverhältnis zwischen dem Personen- und Sitzungsobjekt

Die Klasse `AuthenticationManager` der pCSK definiert eine statische Variable, die das Ablaufdatum einer Sitzung festlegt, sowie eine Map, welche die momentan eingeloggten Personen im System hält. Loggt sich eine Person ein, wird eine neues Sitzungsobjekt generiert, das in einer HashMap gespeichert wird. Die folgende Methode *createSession* (siehe Listing 5.7) zeigt das Erstellen eines Sitzungsobjekts sowie das Setzen der jeweiligen Attribute.

```
1  private void createSession(Person person) {
2          Session personSession;
3          personSession = new Session();
4          // set all attributes of the session object
5          personSession.setActive(true);
6          personSession.setCreationDate(GregorianCalendar
7                  .getInstance());
8          personSession.setLastAccessedDate(GregorianCalendar
9                  .getInstance());
10         personSession.setUser(person);
11         // generate a new unique sessionID
12         UUID sessionID = UUID.randomUUID();
13         while (sessions.containsKey(sessionID)) {
14                 sessionID = UUID.randomUUID();
15         }
16         personSession.setSessionID(sessionID.toString());
17         // add sessionID to the map and to the specific person
18         person.setLastSession(sessionID.toString());
19         sessions.put(sessionID.toString(), personSession);
20 }
```

Listing 5.7: Erstellen eines Sitzungsobjekts durch die Methode *createSession*

Bei einem Login über die REST-Schnittstelle wird die in dem Sitzungsobjekt erzeugte sessionID als Cookie in der HTTP-Antwortnachricht versendet. Anschließend wird die sessionID im Cookie vom Service-Konsument stets mitgesendet und von Seiten der pCSK auf Gültigkeit geprüft. Das Binden der sessionID an die Antwortnachricht lässt sich wie folgt realisieren (siehe Listing 5.8).

```
1 NewCookie cookie = newCookie("sessionID", toLogIn
2         .getLastSession().toString());
3 return Response.status(Status.OK).entity(toLogIn)
4         .cookie(cookie).build();
```

Listing 5.8: Erstellen eines Cookies mit der zuvor erzeugten sessionID

Um auf das Cookie bei einer erfolgten Anfrage zugreifen zu können, wurde jeder REST-Service mit der Annotation „*@CookieParam*“ versehen. Dadurch kann der im Cookie gespeicherte Wert *sessionID* injiziert und an den Methodenparameter gebunden werden. Die folgende Methode *getLoggedInPerson* (siehe Listing 5.9) ist für das Auslesen des Cookies und für das Anfordern des jeweiligen Benutzers zuständig.

```
1 public Person getLoggedInPerson(String sessionID) {
2   //check if the session is valid and not expired
3   if(getSession(sessionID) == null || !checkSession(sessionID){
4     return null;
5   }else{
6     Person p = getSession(sessionID).getUser();
7     return p;
8   }
9 }
```

Listing 5.9: Anfordern eines Benutzers anhand der übergebenen sessionID

Der Aufruf dieser Methode findet in jedem bereitgestellten REST-Service statt. Die Methode fordert die Person anhand der übergebenden sessionID an und überprüft die Gültigkeit der Sitzung. Der Rückgabewert liefert die eingeloggte Person.

Das Anfordern einer Ressource erfordert die Verifikation des eingeloggten Benutzers. Um beispielsweise eine CLI anzufordern, muss der Benutzer in der übergeordneten ORI involviert sein. Dabei ist es ausreichend, wenn der Benutzer die Rolle User in der ORI innehält.
Sämtliche Methoden, die das Modifizieren, Einfügen oder das Löschen betreffen, dürfen nur ausgeführt werden, wenn die eingeloggte Person Manager der ORI ist. Die Authentifizierung lässt sich durch die Methode *isManagerInFrame* (siehe Listing 5.10) folgendermaßen durchführen.

```
1  public boolean isManagerInFrame(int frameID, Person p){
2    // request the specific CFrameInstance
3    CFrameInstance frame = persistence.frame_select(frameID);
4    if(frame == null){
5      return false;
6    }
7    if(frame.getEmployees().containsKey(p.getID())){
8      Roletype role = frame.getEmployees().get(p.getID());
9      if(role.equals(RoleType.MANAGER){
10       return true;
11     }
12   }
13   return false;
14 }
```

Listing 5.10: Authentifizierung anhand der Methode *isManagerInFrame*

Für jeden REST-Service (ausgenommen der Registrierung) wird erstens geprüft, ob der Benutzer eingeloggt ist und zweitens, ob er über die notwendige Rolle in der ORI verfügt. Sind die geforderten Bedingungen nicht erfüllt, kann der eigentliche Methodeninhalt nicht ausgeführt werden. Das Zwischenschalten der Authentifizierung vor dem Methodenaufruf kann anhand dem Löschen einer ORI verdeutlicht werden (siehe Listing 5.11).

```
@DELETE
@Path("/{id}")
public Response deleteFrameInstance(@CookieParam("sessionID")
  String sessionID, @PathParam("id") int id){
  Person p = authenticationManager.getLoggedPerson(sessionID);
  if(p==null || !(rightsManager.isManagerInFrame(id,p))){
    return Response.status(Status.UNAUTHORIZED).build();
  }
}
```

Listing 5.11: Überprüfung der Rolle dargestellt durch die Methode *deleteFrameInstance*

Ist die Person nicht eingeloggt oder verfügt nicht über die notwendige Rolle in der ORI, wird ein HTTP Statuscode *UNAUTHORIZED* zurückgegeben und das Abbrechen des Methodenaufrufs veranlasst.

5.3.4 Rahmenverwaltung

Innerhalb des proCollab-Prototypen steht das Modul Rahmenverwaltung zur Verfügung. Jeder Benutzer kann eine neue ORI erstellen, wodurch er automatisch die Position des Managers in der neu erstellten ORI erhält. Das Erstellen einer neuen ORI erzeugt einen neuen Wurzelknoten einer Baumstruktur. Das Attribut *isRoot* erhält den Wahrheitswert *true* und der Elternknoten *parent* wird auf *null* gesetzt.
Eine neue ORI kann zusätzlich anhand eines existierenden ORTs erstellt werden. Der bereitgestellte Service *instantiateRootFrame* (siehe Listing 5.12) kopiert durch die Methode *instantiateFrame* die Attribute des ORTs und erzeugt eine ORI, die genau jene Attribute erhält. Enthält der ORT weitere ORTs, CLTs und CFTs, werden diese ebenfalls durch die Methode *instantiateFrame* instanziiert.

```
1  public CFrameInstance instantiateRootFrame(int tID, Person p) {
2    // code for selecting the frame from persistence
3    // instantiate CFrameInstance from CFrameType
4    CFrameInstance instance = instantiateFrame(type, p);
5    instance.setParent(null);
6    instance.setRoot(true);
7    return persistence.frame_save(instance);
8  }
```

Listing 5.12: Instanziierung einer ORI

Ein Manager kann in dem ORI neue Personen hinzufügen oder entfernen. Die Methoden *addFrameManager* und *addUser* greifen dabei auf dieselbe Implementierung zurück. Durch das Übergeben der notwendigen Rolle wird die Methode *addPersonToFrame* aufgerufen, die in folgendem Codefragement dargestellt wird (siehe Listing 5.13).

```
1  public boolean addPersonToFrame(int parentID, int personID,
2    Roletype role) {
3    // code for selecting the person and frame from persistence
4    // add person to CFrameInstance
5    toAdd.getEmployees().put(personID, role);
6    boolean changed = persistence.frame_update(toAdd);
7    if (changed) {
8          return true;
9    }
10   return false;
11 }
```

Listing 5.13: Hinzufügen einer Person zu einer ORI

Nimmt der hinzuzufügende Benutzer bereits an der ORI teil und es findet ein erneutes Hinzufügen statt, wird die bereits vorhandene Rolle überschrieben. Dadurch kann der Benutzer entweder durch das Überschreiben der Rolle User zum Manager ernannt werden oder durch das Zuteilen der Rolle User in seiner Position zurückgestuft werden.

Eine höherwertige Funktion stellt das Verschieben von ORIs dar. Dabei lassen sich drei unterschiedliche Fälle betrachten:
Als erste Variante kann eine untergeordnete ORI innerhalb der als Wurzel fungierenden ORI verschoben werden. Die folgende Abbildung 5.7 zeigt die Verschiebung der grün markierten ORI an eine andere Position. Alle Knoten der Abbildung stellen ORIs dar. In diesem Fall befindet sich die Ursprungs- und Zielposition innerhalb der selben Kindesliste. Es muss daher nur die ORI von der ursprünglichen Position entfernt und an der neuen Position eingefügt werden. Die Referenz auf den Elternknoten bleibt unverändert.

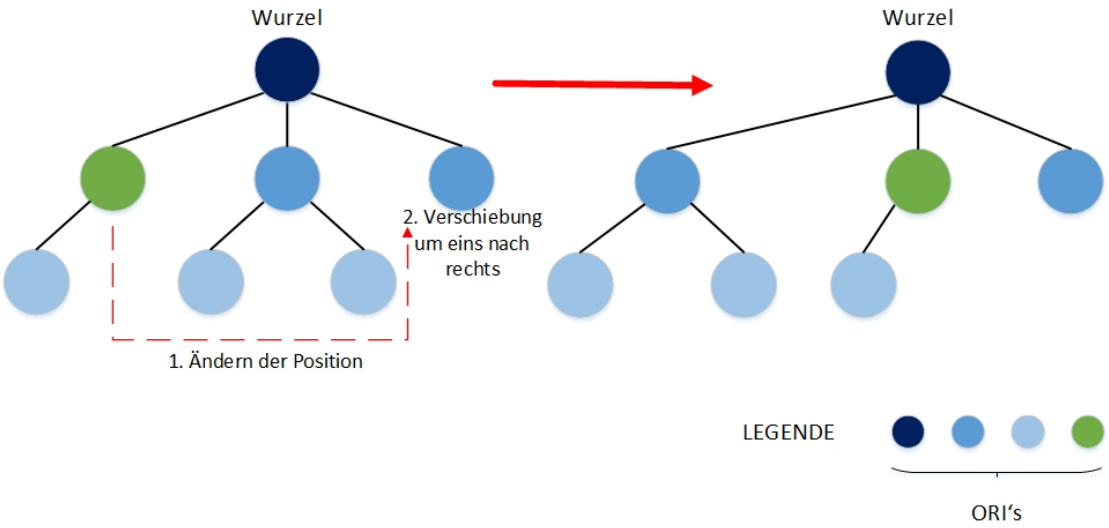


Abbildung 5.7: Verschieben einer untergeordneten ORI innerhalb einer ORI

Die zweite Variante ist das übergreifende Verschieben einer untergeordneten ORI in eine andere ORI. In Abbildung 5.8 wird der grün markierte Knoten dabei von ORI 1 in ORI 2 verschoben. Dabei muss sichergestellt werden, dass die Operation nur genau dann

stattfindet, wenn die jeweilige Person in beiden ORIs, die Stellung des Managers besitzt. Dazu wird die zu verschiebende ORI aus der übergeordneten ORI entfernt und der neuen ORI hinzugefügt. Dazu muss die Referenz zum neuen Elternknoten hergestellt werden indem die ORI als Kind hinzugefügt wird.

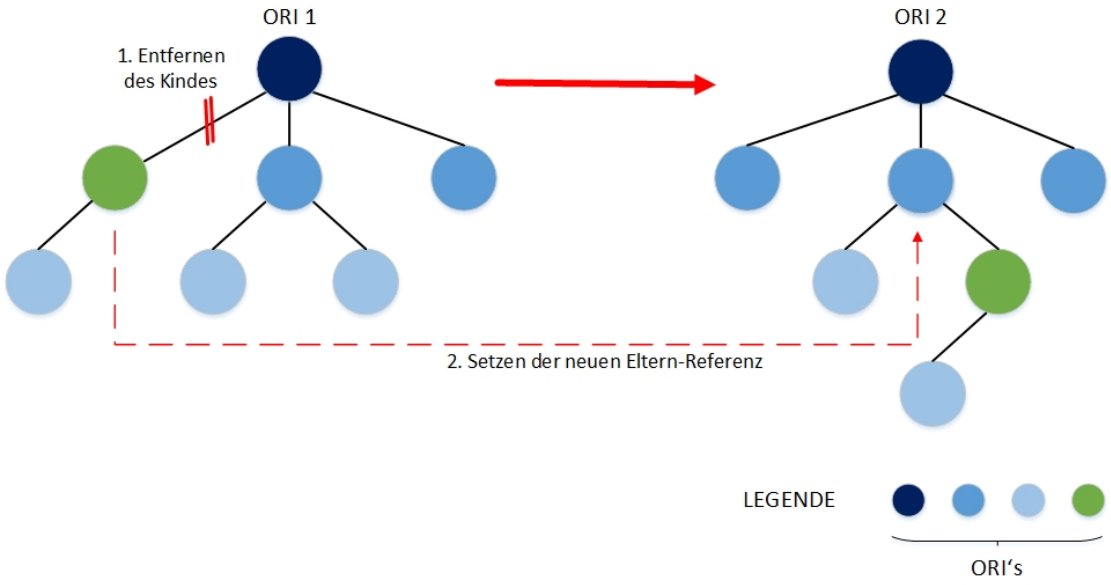


Abbildung 5.8: Verschiebung einer untergeordneten ORI in eine weitere ORI

Abbildung 5.9 zeigt die letzte Variante. Die ORI 1, die als Wurzel ausgezeichnet ist, soll in eine andere ORI 2 verschoben werden. Da die zu verschiebende ORI 1 eine Wurzel ist, muss diese aus keiner Kindesliste entfernt werden. Als Modifikation muss allerdings das Setzen des Attributes *isRoot* von *true* auf *false* erfolgen. Des Weiteren muss das Herstellen der Referenz auf ORI 2 erfolgen.

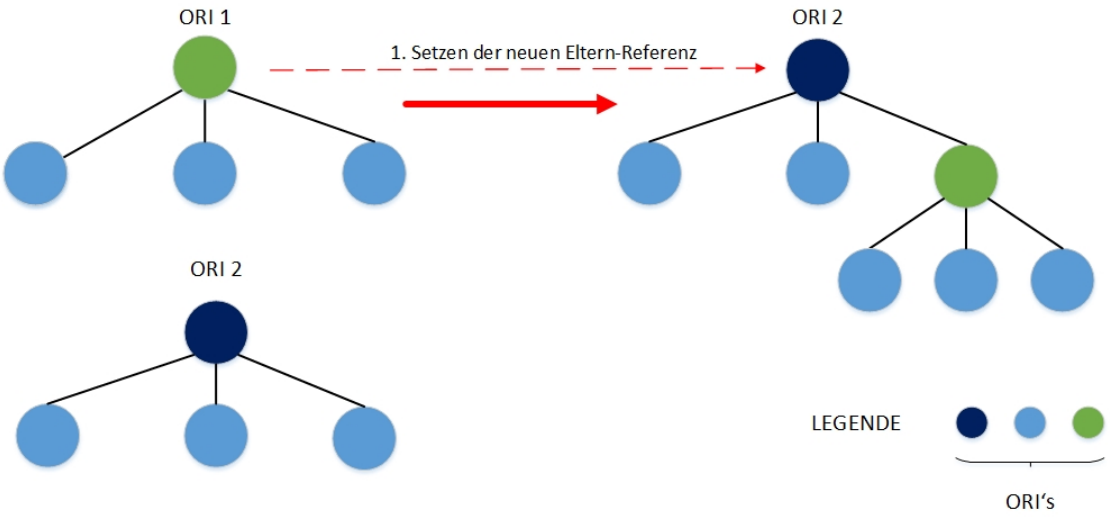


Abbildung 5.9: Integrieren eines Wurzel-Frames in ein Frame

5.3.5 Checklistenverwaltung

Nachdem ein Benutzer einen OR erstellt hat, können diesem OR neue CLs hinzugefügt werden. Das folgende Codefragement in Listing 5.14 zeigt eine Einfüge-Operation einer neuen CLI in eine bereits bestehende CLI.

```
1  public CListInstance insertListIntoList(int parentID,
2    CListInstance list, int position) {
3    // code for selecting the parentList from persistence
4    list.setHasFrame(parent.getHasFrame());
5    list.setIsRoot(false);
6    parent.addChildAt(position, list);
7    boolean changed = persistence.checklist_update(parent);
8    if(changed){
9      return parent;
10   }
11   return null;
12 }
```

Listing 5.14: Einfügen einer CLI in eine existierende CLI

Das Einfügen einer neuen CLI in eine CLI erfordert in allen Fällen das Setzen der zugehörigen ORI. Jede CLI muss diejenige ORI referenzieren, in die sie integriert ist, da CLIs nicht ohne eine zugeordnete ORI existieren dürfen. Das Attribut *isRoot* ist bei untergeordneten CLIs stets auf *false* gesetzt. Letztlich muss noch die Referenz zwischen Eltern- und Kindknoten hergestellt werden. Analog findet die Einfüge-Operation für CFIs statt.
Um eine CLI oder CFI mittels *GET* anfragen zu dürfen, muss der Benutzer in die jeweilige ORI involviert sein. Die Methode *isInvolvedInFrame* prüft analog zu der Methode *isManagerInFrame* (siehe Kapitel 5.3.3), ob der betreffende Benutzer an der ORI teilnimmt. Zusätzlich können Manager Personen zu CLIs hinzufügen und wieder entfernen.

Der Zustand von CLIs und CFIs kann wie auch bei ORIs verändert werden (siehe Kapitel 4.1.2). Wird der Zustand einer CLI von *finished* auf *open* geändert (siehe Listing 5.16), werden durch die Methode *uncheckList* alle untergeordneten CLIs und CFIs ebenfalls auf den Zustand *open* zurückgesetzt.

```
1  public CListInstance uncheckList(CListInstance list) {
2    list.setState(ListInstState.open);
3      if(list.hasChildren()){
4            int i = 0;
5            while(i < list.getNumberOfChildren()){
6              if(list.getChildren().get(i) instanceof
7                CItemInstance){
8                // set state of CItemInstance to open
9                    ((CItemInstance) list.getChildren().get(i))
10                     .setState(ItemInstState.open);
11               }
12               if(list.getChildren().get(i) instanceof
13                 CListInstance){
14                 // call recursively all CListInstances
15                 // and set state to open
16                 uncheckList(((CListInstance)list
17                   .getChildren().get(i)));
18               }
19               i++;
20         }
21 }
```

Listing 5.15: Zustandsänderung einer CLI

Bei dem Iterieren durch die Kindesliste einer CLI wird mittels dem Operator *instanceof* unterschieden, ob der Kindknoten eine CFI oder CLI ist. Die Methode wird solange rekursiv aufgerufen, bis alle Kindknoten in den Zustand *open* gesetzt wurden.

Analog zur Funktionalität der Rahmenverwaltung (siehe Kapitel 5.3.4) können CLIs und CFIs verschoben werden. Für das Verschieben von CLIs und CFIs ergeben sich zwei Möglichkeiten, die für beide identisch sind.
Eine CLI (bzw. CFI) kann innerhalb der selben Kindesliste einer CLI verschoben werden. In diesem Fall ändert sich lediglich die Position der CLI (bzw. CFI).
Die zweite Variante ist, eine CLI (bzw. CFI) übergreifend über zwei CLIs zu verschieben. Dazu muss die CLI (bzw. CFI) aus der Kindesliste entfernt und der neuen CLI hinzugefügt werden. Im Falle, dass sich die neue CLI in einer anderen ORI befindet, muss diese Referenz ebenfalls aktualisiert werden.
Zum Verschieben einer CLI ist noch eine weitere Konstellation denkbar: Eine untergeordnete CLI (isRoot = false) wird zu einer CLI, welche die Wurzel darstellt (isRoot = true), umgewandelt. Folgende Abbildung 5.10 zeigt wie die grün markierte CLI verschoben wird. Dabei stellen die in einem dunkleren blau markierten Knoten ORIs dar und die in einem helleren blau markierten Knoten CLIs.

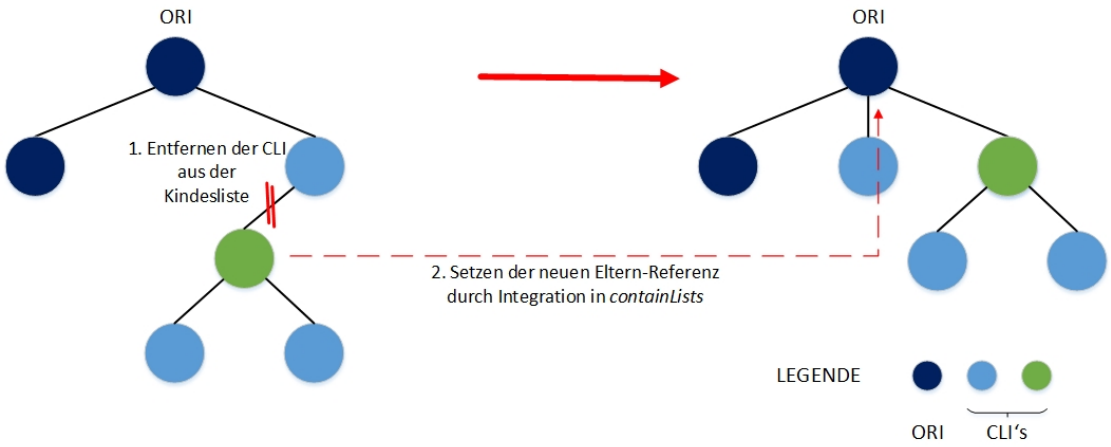


Abbildung 5.10: Umwandlung einer untergeordneten CLI zu einer alleinstehenden CLI

Zuerst wird die Referenz der CLI zu der übergeordneten CLI entfernt. In diesem Beispiel wird die ausgewählte CLI innerhalb der selben ORI zur CLI ernannt, welche die Wurzel darstellt. Es muss daher die Referenz auf die zugehörige ORI nicht aktualisiert werden. Im nächsten Schritt wird das Attribut *isRoot* auf *true* gesetzt und der *parent* auf *null*.

Letztlich wird die zu verschiebende CLI der *containLists* der ORI hinzugefügt.
Weitere Operationen können auf CLIs durchgeführt werden, die als Wurzel markiert sind. Hier steht die inverse Operation zu Abbildung 5.10 zur Verfügung. Eine CLI, welche als Wurzel markiert ist, kann in eine andere CLI integriert werden. Dadurch wird das Attribut *isRoot* auf *false* und der *parent* auf die jeweilige übergeordnete CLI gesetzt. Die Checkliste wird aus der *containLists* der ORI entfernt und in die Kindesliste der hinzuzufügenden CLI eingefügt.

5.3.6 Implementierung einer Suche

Ein Benutzer des proCollab-Prototypen besitzt die Möglichkeit nach Personen, ORIs, ORTs, CLIs, CLTs sowie CFIs und CFTs zu suchen. Die Suche nach ORTs, CLTs und CFTs steht allen Benutzern zur Verfügung. Wendet ein Benutzer die Suchfunktion für ORIs, CLIs und CFIs an, erhält er aus Gründen des Datenschutzes nur jene Instanzen, die ihm unmittelbar zugeordnet sind.

Jede Suche besitzt dabei die individuelle Eingabe von spezifischen Suchattributen. CLIs und CFIs können anhand den Attributen *name*, *text* und *state* gesucht werden, wohingegen die Suche nach dem *state* bei CLTs und CFTs nicht möglich ist.
Eine ORI kann anhand den Parametern *name*, *description* und *goal* gesucht werden. Werden Ergebnisse in der Datenbank gefunden, wird eine Liste von ORIs zurückgegeben. Die von der Persistenzschicht bereitgestellte Suchmethode selektiert alle Treffer unabhängig davon, ob der Benutzer legitimiert ist nach der ORI, CLI oder CFI zu suchen. Es ist daher notwendig die Ergebnisse zu filtern und nur diejenigen zurückzugeben, nach denen ein bestimmter Benutzer suchen darf. Folgendes Codefragement in Listing 5.17 zeigt einen Ausschnitt aus der Suchfunktion.

```
1  // search for CFrameInstances by name, description and goal
2  ArrayList<CFrameInstance> frameInstances = persistence
3    .frameInstance_search(frameInstance.getName(),
4      frameInstance.getDescription(),frameInstance.getGoal());
5
6      if(!frameInstances.isEmpty()){
7      // create resultList for Frontend
8        ArrayList<FrameInstance> mobileFrame =
9          new ArrayList<FrameInstance>();
10       // iterate through all results
11       // and check the role in the CFrameInstance
12       for (int i = 0; i < frameInstances.size(); i++){
13         for (Integer key : frameInstances.get(i)
14           .getEmployees().keySet()){
15             if(key.equals(p.getID())){
16                           mobileFrame.add(toMobileMapper
17                 .mapFrameInstanceToMobile
18                                   (frameInstances.get(i),-1));
19                         }
20                 }
21           }
22         }
23         // further code, if there is no result
```

Listing 5.16: Suchfunktion für ORIs

Um zu identifizieren, ob ein Benutzer die ORI im Suchergebnis angezeigt bekommt, wird die Liste der angestellten Personen einer ORI durchlaufen und nach dem entsprechenden Schlüssel, der PersonenID, gesucht. Ist diese vorhanden, wird die ORI der Resultatliste hinzugefügt.

6

Fazit und Ausblick

Das letzte Kapitel 6 fasst wesentliche Inhalte der Arbeit zusammen und gibt einen Ausblick auf die mögliche zukünftige Forschungsarbeit im Rahmen des Projektes proCollab. Kapitel 6.1 reflektiert die Konzeption und Entwicklung des proCollab-Prototypen sowie die Basis der verwendeten Technologien. Das finale Kapitel 6.2 greift Fragestellungen und Ideen auf, die während der Arbeit entstanden sind.

6.1 Fazit

Wie in Kapitel 1 beschrieben, erfordern Prozesse eine zunehmende Automatisierung, die besonders bei hochdynamischen Prozessen schwer zu erreichen ist. Zur Beschleunigung des Prozessaufbaus können Vorlagen in Form von ORTs und CLTs verwendet werden. Sich ähnelnde Prozesse können auf diese Weise schnell und einfach dupliziert werden. Im Gegensatz dazu zwängt das Konzept der Checklisten einen Prozess nicht in ein vordefiniertes Modell. Checklisten sind lose definiert und legen ausschließlich

fest, was in einem Prozess getan werden muss, nicht aber, wie die genaue Ausführungsreihenfolge auszusehen hat. Dies erleichtert die Strukturierung wissensintensiver Prozesse.
Kapitel 2 hat Anwendungsfälle in der Praxis illustriert, die bereits den Einsatz von Checklisten in unterschiedlichen Bereichen dokumentieren. Die Anwendung von Checklisten im Fachbereich der Automobilindustrie, wie es in Kapitel 2.3.1 dargestellt wurde, zeigt, dass durchaus Verbesserungspotential vorhanden ist. Das Projekt *proCollab* setzt auf dieser Problematik auf und kann die Effektivität von Checklisten in der Automobilindustrie steigern.
Wird in Anbetracht gezogen, dass Wissensarbeiter, wie zum Beispiel Forscher, ihre Forschungsarbeit häufig an unterschiedlichen Orten betreiben, ist die hohe Verfügbarkeit des proCollab-Prototypen unerlässlich. Kapitel 3 hat dabei sowohl funktionale als auch nicht-funktionale Anforderungen spezifiziert, die unter anderem den proCollab-Prototypen als einen cloud-fähigen proCollab-Prototypen vorsehen, um damit dem Aspekt der Verfügbarkeit gerecht zur werden. Anhand den definierten Anforderungen ist auch die Entscheidung für SOA entstanden. Durch das Anbieten von Services, wie es in Kapitel 4 beschrieben wurde, werden diese für mehrere Service-Konsumenten wiederverwendbar und somit verfügbar. Zur Umsetzung einer SOA wurde das von Roy Fielding beschriebene Paradigma für RESTful Webservices eingesetzt, das auf einer Menge von etablierten und bewährten Prinzipien beruht. Während der Entwicklung des proCollab-Prototypen hat sich REST als ein flexibles und individuell anpassbares Paradigma zur Implementierung bewährt. REST unterstützt eine Vielzahl von Repräsentationen und kann die Ressourcen daher in mehreren Formaten bereitstellen. Wie in Kapitel 5 dargestellt, verwendet der Prototyp die Repräsentation JSON, welches zudem ein leicht lesbares Datenformat ist.

Als abschließendes Fazit der Arbeit stellt die Konzeption und stetige Evaluation des proCollab-Prototypen eine wichtige und unerlässliche Basis zur Implementierung dar. Die Definition geeigneter Schnittstellen ist eine zentrale Voraussetzung zur Interoperabilität der einzelnen Schichten der Systemarchitektur. Die Konzeption der Architektur spielt für die Auswahl geeigneter Technologien eine signifikante Rolle. Dadurch können aus dem

umfassenden Angebot an verfügbaren Technologien die am besten korrespondierenden ausgewählt werden.
Der proCollab-Prototyp kann als Schritt hin zu der Unterstützung wissensintensiver Prozesse gesehen werden. Es ist offensichtlich, dass zukünftig noch weitere Technologien und Strategien entwickelt werden müssen, um dem immer stärker werdenden Bedürfnis der adäquaten Prozessunterstützung gerecht zu werden. Die heutige Informationsgesellschaft wird vermehrt durch heranwachsende Wissensarbeiter angetrieben, die eine verbesserte Prozessunterstützung mit Sicherheit begrüßen würden.

6.2 Ausblick

Während der Entwicklung des proCollab-Prototypen sind alternative Umsetzungsmöglichkeiten und Erweiterungen hervorgegangen, die in diesem Kapitel näher erläutert werden. Da es sich bei der Implementierung um einen Prototypen handelt, kann dieser im Verlauf der Forschung um diese Aspekte erweitert werden. Dabei wird insbesondere auf vier essentielle Kriterien eingegangen:

Erweiterung des Zustandsmodells
Das Zustandsmodell, wie es in Kapitel 4 beschrieben ist, ist bisher noch nicht vollständig in den proCollab-Prototypen integriert. Zum momentanen Zeitpunkt existieren die Zustände *open* und *finished*. Die bereits definierten Zustandsübergänge sind umgesetzt, finden aber noch keine Anwendung. Zukünftig soll es beispielsweise möglich sein ORIs zu archivieren oder laufende ORIs abzubrechen. Dadurch können noch präzisere Angaben zum Status einer ORI gemacht werden.

Einführung von Statuscodes und Fehlerbeschreibungen
Wichtig zur Erkennung und Identifizierung von Systemfehlern ist das Einführen einer einheitlichen Ausnahmebehandlung. Das Anbieten von standardisierten *Exceptions* und Statuscodes sollte bei der weiteren Entwicklung des proCollab-Prototypen nicht fehlen. Wächst der proCollab-Prototyp weiter, kann dies problematisch werden, da nicht mehr

unterschieden werden kann, wo der Fehler entstanden ist.
Es ist daher notwendig eine wohldefinierte Menge an Statuscodes einzuführen, die genau definieren, in welcher Schicht der Fehler entstanden ist und im Optimalfall eine Fehlerbeschreibung liefern, wie der Fehler behoben werden kann. Die Integration von Statuscodes sollte in der untersten Ebene, der Persistenzschicht, beginnen und sich durch alle Schichten ziehen. So kann ein möglicher Systemfehler erkannt und die Fehlerquelle gefunden werden.

Optimierung der internen Baumstruktur

Damit zukünftig die Einfüge- und Verschiebe-Operationen auf der internen Baumstruktur effizienter und präziser durchgeführt werden können, hat sich während der Entwicklung des proCollab-Prototypen eine Idee entwickelt, die auf dem Konzept von eindeutig durchnummerierten Knoten beruht. Abbildung 6.1 zeigt, wie jeder Knoten mit einer eindeutig gekennzeichneten Ziffer versehen wird.

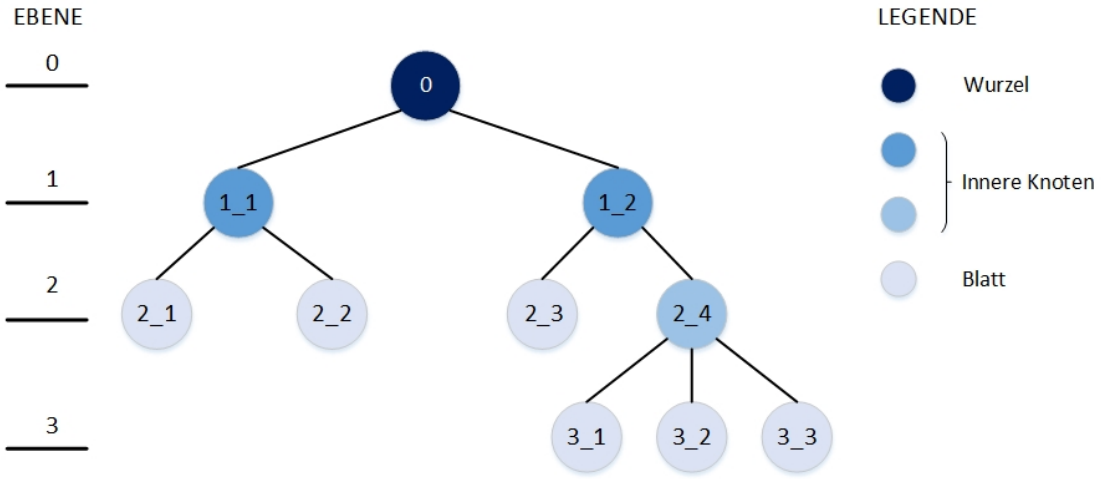


Abbildung 6.1: Zukünftig mögliche Darstellung der internen Baumstruktur

Die erste Ziffer markiert dabei die Ebene auf der sich der betroffene Knoten befindet, die zweite Ziffer kennzeichnet die Position des Knotens innerhalb der Ebene. Soll beispielsweise ein Knoten zwischen *1_1* und *1_2* eingefügt werden, muss nur die hintere

Ziffer des Knotens von *1_2* auf *1_3* geändert werden. Die Kinder der darunterliegenden Ebene können dabei unangetastet bleiben, da sich deren Identifizierung nicht ändert. Infolge dessen kann eine mögliche Einfüge-Operation folgendermaßen aufgebaut sein:

insert(newNode, 0, right1_1)

Die Methode zum Einfügen eines Knotens erhält zum einen den neu einzufügenden Knoten, des Weiteren die Identifikation des Elternknotens und zuletzt die Position. *right1_1* gibt dabei an, dass der neue Knoten rechts von dem bereits existierenden Knoten *1_1* eingefügt werden soll.

Erweiterung der Checklisten-Operationen

Der proCollab-Prototyp stellt momentan höherwertige Methoden wie das Verschieben von ORIs, CLIs und CFIs innerhalb der Baumstruktur bereit.
Bisher lässt sich die Baumstruktur des proCollab-Prototypen nur vertikal erweitern. Beispielsweise können einer CLI weitere CLIs und CFIs hinzugefügt werden. Es ist aber nicht möglich, eine CFI in eine CLI umzuwandeln. Diese Funktion ist vor allem dann sinnvoll, wenn einer CLI noch weitere Elemente hinzugefügt werden sollen. CFIs können laut Anforderungsspezifikation aber keine weiteren Kinder enthalten. Aus diesem Grund kann eine weitere Funktionalität sein CFIs zu CLIs zu erheben, um damit die Möglichkeit zu schaffen weitere Kinder hinzuzufügen.
Eine weitere höherwertige Funktion kann beispielsweise das Verzahnen von CLIs sein. Dabei werden werden zwei CLIs ineinander integriert, wobei als Resultat eine einzige zusammengeführte CLI entsteht. Die einzelnen Elemente der CLI werden dabei abwechselnd angeordnet. Diese Funktion ist insbesondere dann von Vorteil, wenn innerhalb eines Projektes ähnliche CLIs entstanden sind, die genauso gut durch eine CLI repräsentiert werden können. Der Benutzer kann die CLIs einfach zusammenführen, ohne eine der beiden löschen zu müssen und mühevoll die Elemente von Hand von der einen CLI in die Andere übertragen zu müssen.

Abbildungsverzeichnis

Quellcode

Literaturverzeichnis

[Ado13] ADOBE SYSTEMS: *PhoneGap: Open source framework for quickly building cross-platform mobile apps using HTML5, Javascript and CSS*. http://phonegap.com/. Version: 2013, Abruf: 19. Sept. 2013

[Bal09] BALZERT, Helmut: *Lehrbuch der Software-Technik*. Heidelberg : Springer, 2009

[Bau10] BAUER, Hartwig: Cockpit und OP-Saal: Checklisten verbessern Sicherheit. In: *Berlin Medical* (Jan. 2010), S. 8–12

[BHMS05] BERBNER, Rainer ; HECKMANN, Oliver ; MAUTHE, Andreas ; STEINMETZ, Ing R.: Eine dienstgüte unterstützende Webservice-architektur für flexible Geschäftsprozesse. In: *Wirtschaftsinformatik* 47 (2005), Nr. 4, S. 268–277

[BIT09] BITKOM: *Cloud Computing - Evolution in der Technik, Revolution im Business*. http://www.bitkom.org/files/documents/BITKOM-Leitfaden-CloudComputing_Web.pdf. Version: 2009, Abruf: 08. Aug. 2013

[BKNT11] BAUN, Christian ; KUNZE, Marcel ; NIMIS, Jens ; TAI, Stefan: *Cloud computing: Web-based dynamic IT services*. Berlin und Heidelberg : Springer, 2011

[Buc09] BUCANEK, James: *Learn Objective-C for Java Developers*. Berkeley and CA : Apress, 2009

[Bur10] BURKE, Bill: *RESTful Java with JAX-RS*. Sebastopol and CA : O'Reilly, 2010

[Cer13] CERVENKA, Tobias: *Pilotenberuf - Die Reportage: Der Outside-Check*. http://www.pilotline.ch/pilotenberuf_reportage_p4.html. Version: 2013, Abruf: 17. Juni 2013

[Coh04] COHN, Mike: *User stories applied: For agile software development*. Boston and Mass. [u.a.] : Addison-Wesley, 2004

[Dru07] DRUCKER, Peter F.: *Management challenges for the 21st century*. Amsterdam and London : Butterworth-Heinemann, 2007

[Erk09] ERKLAERT.DE: *Im Cockpit: Sicherheit doppelt und dreifach*. `http://www.erklaert.de/navigation/cockpit-sicherheit`. Version: 2009, Abruf: 17. Juni 2013

[Erl05] ERL, Thomas: *Service-oriented architecture: Concepts, technology, and design*. Upper Saddle River and NJ : Prentice Hall Professional Technical Reference, 2005

[FGM+99] FIELDING, Roy ; GETTYS, Jim ; MOGUL, Jeffrey ; FRYSTYK, Henrik ; MASINTER, Larry ; LEACH, Paul ; BERNERS-LEE, Tim: *Hypertext transfer protocol–HTTP/1.1*. `http://www.hjp.at/doc/rfc/rfc2616.html`. Version: 1999, Abruf: 21. Aug. 2013

[Fie00] FIELDING, Roy: *Architectural styles and the design of network-based software architectures*, University of California, Diss., 2000

[Gaw11] GAWANDE, Atul: *How to get things right*. London : Profile Books Ltd and TBS The Book Service Ltd, 2011

[Gei13] GEIGER, Sabrina: *Konzeption und Entwicklung einer auf Smartphones optimierten mobilen Anwendung für kollaboratives Checklisten-Management*, Universität Ulm, Bachelorarbeit, 2013

[GPSW03] GRONAU, Norbert ; PALMER, Ulrich ; SCHULTE, Karsten ; WINKLER, Torsten: Modellierung von wissensintensiven Geschäftsprozessen mit der Beschreibungssprache K-Modeler. In: *Wissensmanagement* 28 (2003), S. 315–322

[Gup12] GUPTA, Arun: *Java EE: Kurz & gut ; [behandelt Java EE 6]*. Beijing and Cambridge and Farnham and Köln and Sebastopol and Tokyo : O'Reilly, 2012

[HGP07] HEIMANN, Bodo ; GERTH, Wilfried ; POPP, Karl: *Mechatronik*. München and Wien : Fachbuchverlag Leipzig im Carl-Hanser-Verl, 2007

[Hor07] HORN, Thorsten: *Vorgehensmodelle zum Softwareentwicklungsprozess: Sechs Qualitätsmerkmale für Softwareprodukte nach ISO 9126 (DIN 66272)*. `http://www.torsten-horn.de/techdocs/sw-dev-process.htm`. Version: 2001-2007, Abruf: 15. Juli 2013

[HWB+09] HAYNES, Alex B. ; WEISER, Thomas G. ; BERRY, William R. ; LIPSITZ, Stuart R. ; BREIZAT, Abdel-Hadi S. ; DELLINGER, E P. ; HERBOSA, Teodoro ; JOSEPH, Sudhir ; KIBATALA, Pascience L. ; LAPITAN, Marie Carmela M. u. a.: A surgical safety checklist to reduce morbidity and mortality in a global population. In: *New England Journal of Medicine* 360 (2009), Nr. 5, S. 491–499

[Jos08] JOSUTTIS, Nicolai: *SOA in der Praxis: System-Design für verteilte Geschäftsprozesse*. Heidelberg : dpunkt-Verlag, 2008

[JSO13] JSON.COM: *Introducing JSON*. `http://www.json.com/`. Version: 2013, Abruf: 08. Juli 2013

[Köl13] KÖLL, Andreas: *Konzeption und Entwicklung einer auf Tablets optimierten mobilen Anwendung für kollaboratives Checklisten-Management*, Universität Ulm, Bachelorarbeit, 2013

[MG11] MELL, Peter ; GRANCE, Timothy: The NIST definition of cloud computing (draft). In: *NIST special publication* 800 (2011), Nr. 145, S. 7

[MKR12] MUNDBROD, Nicolas ; KOLB, Jens ; REICHERT, Manfred: Towards a System of Collaborative Knowledge Work. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, Springer, 2012, S. 31–42

[MM09] MEINEL, Christoph ; MUNDHENK, Martin: *Mathematische Grundlagen der Informatik: mathematisches Denken und Beweisen; eine Einführung*. Wiesbaden : Springer DE, 2009

[MRV11] METZGER, Christian ; REITZ, Thorsten ; VILLAR, Juan: *Cloud computing: Chancen und Risiken aus technischer und unternehmerischer Sicht*. München : Hanser, 2011

[Mue09] MUELLER, Willy: *SOA Prinzipien*. `http://www.e-services.admin.ch/dokumentation/00162/00166/index.html?lang=de`. Version: 2009, Abruf: 22. Juli 2013

[Mun13] MUNDBROD, Nicolas: *Interner Bericht: Nutzung von Checklisten zur Unterstützung der Entwicklung mechatronischer Systeme im Automobilbereich*. Universität Ulm, 2013

[Ora13] ORACLE CORPORATION: *Jersey: RESTful Web Services in Java*. `http://jersey.java.net/`. Version: 2010-2013, Abruf: 20. Juni 2013

[Red13a] REDHAT: *JBoss Application Server 7 - The Open Source Java application server reignited*. `http://www.jboss.org/jbossas`. Version: 2013, Abruf: 03. Juli 2013

[Red13b] REDHAT: *JBoss Documentation - Resteasy Userguide*. `http://docs.jboss.org/resteasy/2.0.0.GA/userguide/html/index.html`. Version: 2013, Abruf: 04. Juli 2013

[Red13c] REDHAT: *RESTeasy - Distributed peace of mind*. `http://www.jboss.org/resteasy`. Version: 2013, Abruf: 12. Juni 2013

[Rei13] REICH, Daniel: *Konzeption und Entwicklung eines Cloud-basierten Persistenz-Systems für kollaboratives Checklisten-Management*, Universität Ulm, Bachelorarbeit, 2013

[Res13] RESTLET: *Restlet Framework: The leading web API framework for Java*. `http://restlet.org/`. Version: 2013, Abruf: 25. Juli 2013

[Rou10] ROUGH BOOK: *Generic (n-ary) Tree in Java*. `http://vivin.net/2010/01/30/generic-n-ary-tree-in-java/`. Version: 2010, Abruf: 28. Juni 2013

[RR07] RICHARDSON, Leonard ; RUBY, Sam: *RESTful web services*. Farnham : O'Reilly, 2007

[RW12] REICHERT, Manfred ; WEBER, Barbara: *Enabling flexibility in process-aware information systems: Challenges, methods, technologies*. Heidelberg : Springer, 2012

[S+10] SWENSON, Keith D. u. a.: Mastering the Unpredictable. In: *How Adaptive Case Management Will Revolutionize the Way That Knowledge Workers Get Things Done, Glossary* (2010), S. 314–317

[Sri12] SRIPATHI, Triguna M.: *Comparing RESTful web services implementation with Jersey and Restlet*. `http://compare-tech.blogspot.de/2012/09/comparing-restful-web-services.html`. Version: 2012, Abruf: 20. Juni 2013

[Sta10] STARK, Thomas: *Java EE 5: Einstieg für Anspruchsvolle*. [München] and München/Germany : Addison-Wesley and Pearson Studium, 2010

[Thi13] THIEL, Norman: *Konzeption und Entwicklung einer Web-Applikation für kollaboratives Checklisten-Management*, Universität Ulm, Bachelorarbeit, 2013

[Til11] TILKOV, Stefan: *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. Heidelberg : dpunkt, 2011

[Uni13] UNI ULM INTERN: Chirurgische Kliniken nutzen Checkliste: 30 Häkchen für mehr Patientensicherheit. In: *Uni Ulm Intern* (Apr. 2013), S. 50

[Vaa12] VAADIN: *Vaadin: Java framework for building modern web applications*. `https://vaadin.com/home#peace`. Version: 2012, Abruf: 19. Sept. 2013

[Wor09] WORLD HEALTH ORGANIZATION: *Surgical Safety Checklist*. `http://whqlibdoc.who.int/publications/2009/9789241598590_eng_Checklist.pdf`. Version: 2009, Abruf: 14. Juli 2013

Name: Jule Anna Ziegler Matrikelnummer: 725114

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Jule Anna Ziegler