Analyse des prototypischen Frameworks CrAc-Core zur Zuordnung von Aufgaben an freiwillige Helfer

David Hondl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 16. Juni 2017

David Hondl

Inhaltsverzeichnis

Vorwort

Diese Arbeit, sowie der evaluierte Prototyp, entstanden beide in Verbindung mit dem Dachprojekt *Cooperative Activities*. Dieses setzt sich mit unterschiedlichen Konzepten aus dem Bereich des *Freiwilligen-Managements* auseinander und kombiniert verschiedene Projekte, um eine einzigartige Plattform für freiwillige Helfer zu schaffen. Mehr Informationen zu *CrAc* sind in der Einführung dieser Arbeit zu finden oder online und mehrsprachig unter http://www.crac.at/crac-cooperative-activities-1.

Der gesamte Code des hier analysierten Frameworks *CrAc-Core*, sowie eine Liste aller Endpoints, die die *API* zur Verfügung stellt, und sämtliche Details der an ihm durchgeführten Tests können auf der *CD* nachgeschlagen werden, die dieser Arbeit beigelegt ist.

Abstract

This work presents the prototypical open-source framework *CrAc-Core*, which is used for assigning tasks to volunteers – based on *qualitative criteria* such as competences or skills – and analyzes it in detail.

First of all, this thesis goes into detail on the concept of *volunteering* and the community behind it. The values of volunteers and their demands regarding matching platforms are discussed and become a substantial part of the CrAc-Core as concrete requirements. Furthermore, this thesis deals with the question of the right technology for implementing such a platform, as well as the type of architecture – including the components and modules of the system – needed.

Of course, the most important aspect of the framework, the technology for assigning users and tasks, is discussed as well. The used approach is analyzed and the implementation and adjustments in the context of the CrAc-core are elaborated on. Finally the question gets answered, whether or not the resulting framework meets the previously discussed requirements, this is answered partly on the basis of tests.

Kurzfassung

Diese Arbeit stellt das prototypische Open-Source Framework *CrAc-Core* zur Zuordnung von Aufgaben an freiwillige Helfer – basierend auf *qualitativen Kriterien*, wie Kompetenzen oder Skills – vor und analysiert dieses detailliert.

Zunächst wird hierbei auf das Thema *Volunteering* und der dahinter stehenden Gemeinschaft eingegangen. Die Werte freiwilliger Helfer und deren Ansprüche an Matching-Plattformen werden erörtert und als konkrete Anforderungen zu einem fixen Bestandteil des CrAc-Cores. Desweiteren setzt sich diese Thesis mit der Frage nach der richtigen Technologie zur Umsetzung einer solchen Plattform auseinander, sowie mit der Art der benötigten Architektur und allen Komponenten und Modulen, aus der diese besteht.

Natürlich wird auch auf den wichtigsten Punkt des Frameworks eingegangen, die Technik zur Zuordnung von Benutzern und Aufgaben. Das hierfür verwendete Verfahren wird analysiert und die Umsetzung und Anpassungen im Kontext des CrAc-Cores werden diskutiert. Als letzter Punkt wird die Frage geklärt, ob das resultierende Framework den zuvor erörterten Anforderungen – unter anderem auf Basis von Tests – entspricht.

Kapitel 1

Einführung

Voluntarismus. Ein Phänomen so alt wie die Menschheit selbst. Während sich Kommunen und Völker ständig weiterentwickeln, ist das freiwillige Helfen ein zeitloses, unverändertes Konzept. Lediglich die Möglichkeiten der gemeinsamen Kommunikation, ein äußerst wichtiger Faktor für jedes größere Vorhaben, haben sich in Menge und Qualität verändert. Diese Veränderungen umfassen vor allem die Erfindung und Entwicklung der Telefonie. Eine Technik, die es freiwilligen Helfern aufgrund der Geschwindigkeit und Einfachheit der Kontaktaufnahme so einfach wie nie zuvor machte, gemeinsame Aktivitäten zu planen und auszuführen. Das Aufkommen der Mobiltelefonie verstärkte diesen Effekt einige Jahre danach noch.

Dies schlägt die Brücke zum Thema dieser Arbeit, denn der bis jetzt größte Wandel in der Kommunikation findet aktuell mit der intensiven Nutzung des Internets statt. Onlineportale ermöglichen die Vernetzung von Helfern aus der ganzen Welt und garantieren in Kombination mit einer riesigen Varianz portabler Computer-Systeme eine nahezu unbegrenzte Erreichbarkeit beteiligter Freiwilliger. Doch die Technologie beschränkt sich im Zeitalter moderner Informationstechnik nicht mehr einfach darauf, ein Mittel zur Kommunikation zu sein. Systeme mit Zugriff auf riesige Benutzerdatenmengen können Teile von Aufgaben selbst übernehmen oder dabei helfen, diese automatisch an den geeignetsten Mitarbeiter zu übertragen. Der in dieser Arbeit vorgestellte Prototyp ist eine webbasierte *Open-Source-Implementierung* des *Backends* eines solchen komplexen Systems, welches die zusätzlichen Herausforderungen und Probleme im Umfeld von *Freiwilliger Arbeit* aufzeigt und mögliche Lösungsansätze anbieten soll.

Zunächst wird etwas näher auf das Thema „Freiwilligen-Arbeit“ eingegangen. Nicht nur, um die zu beachtenden Faktoren in diesem Arbeitsumfeld aufzuzeigen, sondern auch um die nach wie vor bestehende Relevanz von freiwilligen Helfern in Österreich darzulegen. Ein Einblick in das Dachprojekt *Cooperative Activities* gibt anschließend einen kurzen Überblick darüber, wie es mit dem Prototypen in Verbindung steht und beleuchtet die Möglich-

keiten und Herausforderungen am Markt für Freiwillgen-Software und die Bedeutung des Projektes für Freiwilligen-Arbeit an sich näher.

1.1 Voluntarismus

Kurz zusammengefasst beschreibt Voluntarismus das menschliche Verhalten, gemeinnützig etwas beitragen, etwas einbringen zu wollen. Dies prägt sich in unterschiedlichen Kulturen und Sprachen zwar unterschiedlich aus, ist aber generell unabhängig von ihnen und in jeder Kultur zu finden [18].

Menschen haben meist sehr unterschiedliche Gründe für freiwilligen Dienst, tun es aber generell aus freien Stücken und im Geiste der Zusammengehörigkeit, ohne eine Entlohnung dafür zu erwarten. Allerdings bedeutet dieser Umstand nicht, dass der Helfer überhaupt nicht profitiert, er oder sie gewinnt Erfahrung (in verschiedenen Arbeitsbereichen) und knüpft Verbindungen zu anderen Menschen. Noch einmal ist anzumerken, dass die Nachfrage nach freiwilligen Helfern und der Wille zur Mitarbeit ein zeitloses Phänomen darstellt, das seit Menschengedenken existiert und gerade jetzt in Verbindung mit vergangenen und kommenden Krisen im humanitären Bereich relevanter ist denn je. Eine dazu passende Studie des *Ministeriums für Arbeit und Soziales* hat 2012 ergeben, dass nach wie vor 46% der über 15-jährigen Population freiwillige Arbeit in irgendeiner Form leistet [5].

An dieser Stelle muss nachdrücklich betont werden, dass der hier vorgestellte Prototyp zwingend die Werte dieser riesigen Gemeinschaft miteinbeziehen muss, um seine Ziele für sie und mit ihr zu erreichen. Einen guten Überblick über verschiedene Aspekte, welche bei einer bestehenden Menge an Freiwilligen berücksichtigt werden sollten – soweit dies möglich ist – bietet [9].

1.2 Das CrAc-Project

Genau aus diesem Grund wurde im November 2014 das Projekt *Cooperative Activities* ins Leben gerufen. Es handelt sich dabei um ein gemeinsames Projekt von Partnern aus dem akademischen Bereich und der Software-Industrie, welches von der *Austrian Research Promotion Agency*[1] finanziert wird. Das Ziel dieses Projektes ist die Erstellung einer webbasierten Software, die die organisatorische Arbeit in gemeinnützigen Vereinen wie *Das rote Kreuz*[2] oder *Freie Waldorfschule Linz*[3] reduziert. Es existieren zwar ähnliche Projekte und Produkte, allerdings ist die geplante Plattform von CrAc die erste dieses Ausmaßes, die komplett *unentgeltlich und Open-Source* veröffentlicht werden soll. Um Konkurrenz-Produkten nicht nur in dieser Hinsicht

[1] https://www.ffg.at/
[2] http://www.roteskreuz.at/home/
[3] https://www.fwsl.at/freie-waldorfschule-linz/schule/

voraus zu sein, führten Mitarbeiter von CrAc vor der konkreten Entwicklung einer Software eine Markt-Recherche über verschiedene etablierte Produkte durch.

Das Resultat, beschrieben in [11], gibt einen Überblick über den Markt und zeigt außerdem auf, warum verschiedene Produkte in verschiedenen Aspekten der Zielgruppe nicht gerecht werden. So erlauben verschiedenste Plattformen beispielsweise nicht das Erstellen von *komplexen Aufgaben*, ein angemessenes *Matching* von Aufgaben und Freiwilligen, sowie ein angepasstes Matching auf Basis von sich weiter entwickelnden Usern, hier auch *Evolution* genannt. Als Konsequenz daraus wurden Anforderungen abgeleitet, die erfüllt werden müssen, um die evaluierten Systeme zu übertreffen und das CrAc-Projekt markttauglich zu machen. Folgende Anforderungen werden auf Basis des Papers als Schwerpunkte an einen Prototypen des CrAc-Projektes [14] gestellt:

- *Dynamische Profile* repräsentieren den Benutzer und seine/ihre Kompetenzen und Errungenschaften.
- Ein *Matching-System* empfiehlt dem Benutzer basierend auf seinen/ihren Skills automatisiert verschiedenste Aufgaben.
- *Evolutions-Mechanismen* ermöglichen die eigenständige Weiterentwickeln des Systems auf Basis der Resultate von Aufgaben und Benutzer-Feedback.

1.3 CrAc-Core

Wie bereits zuvor angeschnitten, stellt der in dieser Thesis analysierte Prototyp die Implementierung eines solchen komplexen Systems für und in Kooperation mit dem CrAc-Projekt dar und bietet Lösungsansätze für dabei auftauchende Probleme. Die Software beschäftigt sich in ihrer Funktion als Backend rein mit der Verarbeitung von Daten und nicht deren grafischer Ausgabe für den Endnutzer (dem freiwilligen Helfer). Hier liegt eine bewusste Abgrenzung zur Frontend-Entwicklung vor, um eine unabhängige Entwicklung und Skalierbarkeit zu gewährleisten. Aufgrund seiner Funktionalität und der erwähnten Abgrenzung trägt der Prototyp einen eigenen internen Namen: *CrAc-Core*. Dieser wird zur Referenzierung in dieser Arbeit verwendet. Nachfolgend wird etwas Abstand vom reinen Konzept genommen und genauer auf den Prototypen selbst und dessen spezifische Anforderungen eingegangen.

1.3.1 Konkrete Anforderungen

Folgende aus den Anforderungsbereichen in Abschnitt 1.2 abgeleitete und zusätzlich hinzugefügte Ansprüche müssen im Kontext des Gesamtprojektes an den Prototypen gestellt werden, um der Zielgruppe und den Benutzern

gerecht zu werden. Auf ihre Erfüllung wird zu einem späteren Zeitpunkt (s. Abschnitt 7.4) erneut eingegangen.

Datenpersistenz

Der Prototyp muss das *Validieren* und *Persistieren* von Personen- und Aufgaben-bezogenen Daten ermöglichen. Diese werden für sämtliche Aktionen in CrAc benötigt und müssen damit jederzeit verfügbar sein.

Such- und Empfehlungsmechanismen

Das System muss über einen oder mehrere Mechanismen verfügen, die ein Zusammenführen oder auch *Matching* von Aufgaben und Benutzern nicht nur auf quantitativer, sondern auch auf qualitativer Ebene ermöglichen.

Entwicklungsmechanismen

Das System muss über einen Mechanismen verfügen, der es ihm erlaubt, die Resultate von Aufgaben, bezogen auf das Endergebnis und die Erfahrungen des involvierten Benutzers, in die Suchmechanismen einfließen zu lassen und diese so stetig weiter zu entwickeln.

Unabhängigkeit von eigenem Frontend

Der Prototyp soll die Möglichkeit zur Anbindung an beliebige weitere Applikationen bieten und nicht von einem einzigen Frontend abhängig sein. Somit benötigt er eine standardisierte Art, um mit *Requests* und *Responses* umzugehen, um daran angeschlossenen Systemen eine verlässliche Schnittstelle zu bieten. Der Zugriff muss Frontend-seitig einfach zu implementieren sein, einheitliche Antworten in standardisierten Übertragungsformaten zurückliefern und eine Unabhängigkeit zwischen Frontend und Backend gewährleisten.

Unabhängigkeit von Prototyp-Instanzen

Nicht nur soll der Prototyp unabhängig von den jeweiligen Frontends agieren, auch verschiedene Instanzen des Prototyps sollen unabhängig zueinander existieren und arbeiten können. Dies gewährleistet die Unversehrtheit von sensiblen Organisationsdaten, die auf der Datenbank der jeweiligen Instanz persistiert werden.

Austausch von gemeinsamen Daten

Trotzdem soll eine *Schnittstelle* zwischen verschiedenen Instanzen des Cores etabliert werden, die den Austausch und die Synchronisierung von gemeinsamen Daten ermöglicht, vor allem um multiple Eingaben bestimmter Daten

auf Seiten des Endnutzers zu vermeiden. Ausgeschlossen werden hierbei wie angemerkt sensible Organisationsdaten.

Code-Wiederverwendbarkeit

Auch als sehr wichtig ist die *Wiederverwendbarkeit des Source-Codes* einzustufen, da die weitere Entwicklung des Prototypen an einem gewissen Punkt durch andere Personen als den Urheber stattfindet. Relevant ist in diesem Zusammenhang auch, dass der Core den *Open-Source-Ansatz* verfolgt, was eine gleichzeitige Zusammenarbeit von unabhängigen Entwicklern ermöglicht. Die Wahl einer sehr verbreiteten, einfach strukturierten Programmier- oder Skriptsprache ist daher in Kombination mit einer Objekt-orientierten, gut durchdachten Architektur ein äußerst wichtiger Faktor.

Keine Monetarisierungsmöglichkeiten

Obwohl diese Anforderung einfach umsetzbar erscheint, muss sie bei jedem Bestandteil des CrAc-Cores miteinbezogen werden. Besonders der Matching-Prozess, aber auch die Anforderungen und der Outcome von Aufgaben im Allgemeinen dürfen nicht von Belohnungen in monetärer Form abhängig gemacht werden können.

Kapitel 2

Technische Basis

Nachdem auf die Grundprämisse und die allgemeinen und konkreten Anforderungen eingegangen worden ist, wird ab diesem Punkt der *CrAc-Core* an sich vorgestellt und die Teilbereiche analysiert. Für ein besseres Verständnis der Framework-internen Abläufe wird aus diesem Grund zuerst ein Überblick über die technische Basis und die verwendeten Technologien und Programmierparadigmen gegeben. Der Einsatz bestimmter Technologien, auf die auch in späteren Abschnitten Bezug genommen wird, stellt hierbei bereits eine teilweise Umsetzung der *Anforderungen* (s. Abschnitt 1.3.1) dar.

2.1 Sprache und Framework

Wie bei jedem Projekt stellt der erste Schritt der Entwicklung die Wahl eines Grundgerüstes dar. Dieses dient als Baukasten zur Umsetzung und muss passend für den Anwendungsfall gewählt werden. Da es sich beim CrAc-Core um einen *webbasierten Prototypen* handelt, ist es am effizientesten, eines der existierenden *Webframeworks* zu wählen. Diese speziell für den Anwendungsfall „Web-Applikation“ entwickelten Software-Pakete bieten eine Reihe von Grundfunktionen und Features, die unabhängig für jedes webbasierte Projekt notwendig sind. Nachfolgend einige der wichtigsten dieser Werkzeuge:

1. Management von Requests und Responses,
2. Routing zwischen Methoden und Endpoints,
3. Security-Werkzeuge.

Die Wahl ist dabei denkbar schwierig. Es existiert im Bereich webbasierter Frameworks eine *enorme Vielfalt* unterschiedlicher, untereinander konkurrierender Produkte. Diese bringen ihre jeweiligen Stärken und Schwächen mit, basierend auf den Konzepten und verwendeten Technologien, darunter auch die Programmier- oder Skriptsprache. Die Wahl der Programmiersprache geht also nahezu untrennbar Hand in Hand mit der Wahl des Frameworks,

da dieses das eigentlich benötigte *Tool-Kit* darstellt.

Was also sind die Anforderungen des Prototypen an ein potentielles Framework?

1. Die Programmier- oder Skriptsprache muss allgemein bekannt und einfach zu lesen und zu verstehen sein (s. Abschnitt 1.3.1).
2. Das Framework und die Sprache müssen Objekt-Orientierung forcieren (s. Abschnitt 1.3.1).
3. Es muss eine ausreichende Zahl an zuverlässigen *Libraries* für verschiedenste Anwendungsfälle existieren.
4. Das Framework muss eine stabile, ausreichend getestete Version bieten.
5. Das Framework darf die Skalierbarkeit des Systems nicht einschränken.
6. Das Framework muss über ausreichende Grundfunktionen (beispielsweise ausreichend Security-Funktionalität) verfügen.
7. Brauchbare Umsetzungen für ausgewählte und zu testende Matching-Konzepte müssen in der gewählten Sprache existieren.

Basierend auf diesen Punkten werden im CrAc-Core die Programmiersprache *Java*[1] und das Framework *Java Spring*[2] verwendet. Java erfüllt die Anforderungen nicht nur hinsichtlich der Fokussierung auf Objekt-Orientierung und einfach strukturiertem Code, sondern bietet auch eine große Anzahl an gut getesteten und einfach einzubindenden Libraries. Unter eben diesen befinden sich auch eine API für die Onthologie-Implementierung *Apache Jena* und eine API für die Volltext-Suchmaschine *Elasticsearch*, beide relevant hinsichtlich Matching (s. Abschnitt 1.3.1). *Spring* auf der anderen Seite bietet als etabliertes und mächtiges *Enterprise-Framework* alle Werkzeuge, die eine Applikation dieser Größenordnung benötigt. Das Framework ist außerdem ausreichend getestet, es sind mehrere stabile Versionen verfügbar und im Hintergrund steht eine riesige Community, die bei Bedarf und bei Problemen Hilfe anbietet. Als Zusatz wird die Spring-eigene Library *Boot* benutzt, die das ansonsten komplex zu konfigurierende Framework um eine Vielzahl an Auto-Konfigurationen erweitert. In Verbindung mit den klassischen *Java-Annotationen*[3] kann auf diese Weise eine einfache, transparente Projekt-Konfiguration vorgenommen werden. Verschiedene *Anleitungen* hierzu werden in [17] angeboten. Dies macht die Einarbeitung in den Code und das Verstehen der internen Abläufe einfacher und verbessert auf diese Weise die *Code-Wiederverwendbarkeit*. Mit der definitiven Auswahl der Basistechnologie kann nun auf einige relevante Unterpunkte eingegangen werden.

[1] https://java.com/de/about/whatis_java.jsp

[2] https://spring.io/

[3] https://docs.oracle.com/javase/tutorial/java/annotations/

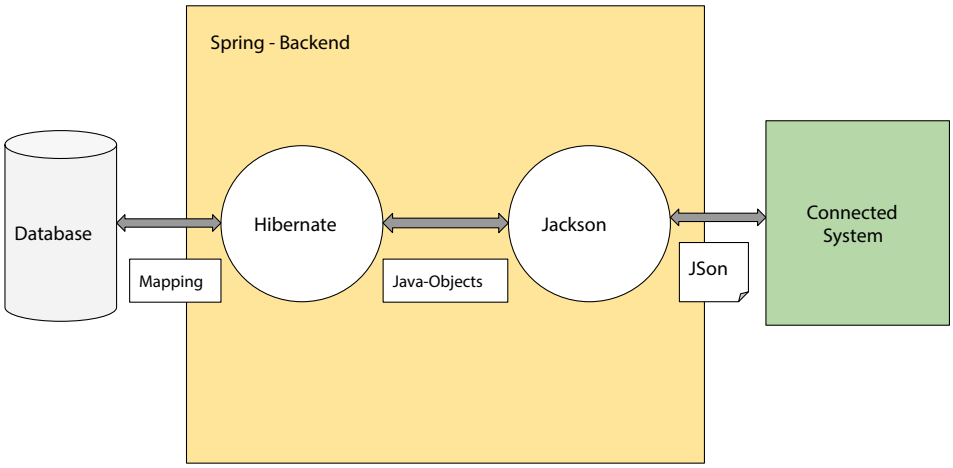


Abbildung 2.1: Daten-Mapping im Framework.

2.1.1 Datenbank

Genutzt wird die relationale Datenbank *Postgres*. Aufgrund der klar definierten, gleichbleibenden Struktur der Daten und deren Beziehungstypen, ist in diesem Fall eine relationale Datenbank eine bessere Wahl als eine NoSQL-Datenbank. Bedingt durch den Aufbau und das Zusammenspiel zwischen verschiedenen *CrAc-Core-Instanzen* (s. Abschnitt 6.1) verursachen auch die Limitierungen in der Skalierbarkeit von relationalen Datenbanken kein Problem. Letztlich stellt Postgres aufgrund seiner nützlichen Erweiterungen, allen voran das Paket *PostGIS* zum Verwalten von *Geo-Daten* für Aufgaben und Nutzer, den besten Kandidaten unter den relationalen Datenbanken dar.

2.1.2 Daten-Mapping

Der nächste Punkt ist das *Mapping*, das Umwandeln von Daten aus einer Form in eine andere. Dies ist vor allem wichtig, da die Weitergabe von Java-Objekten aus dem Crac-Core an ein angeschlossenes System in ihrer ursprünglichen Form nicht funktioniert. Sie müssen daher zuerst, je nach Anwendungsfall, in das richtige Format konvertiert werden. Wie der Datenfluss hierbei aussieht, wird in Abbildung 2.1 dargestellt.

SQL-Mapping

Für das Mapping zwischen Objekten in der Applikation und der Datenbank wird das Framework *Hibernate* verwendet. Dieses erlaubt eine Darstellung der Klassen und Objekte als Tabellen und Einträge in der angebundenen Datenbank und bietet zugleich, in Verbindung mit der *Java Persistence* API

(JPA), eine Schnittstelle zur Interaktion mit sämtlichen verfügbaren Daten.

JSON-Mapping

Eine weitere Art des Mappings wird benötigt, um die Daten des Prototyps an andere, beliebige Systeme zu kommunizieren. Das hierfür verwendete Format nennt sich *JSON*, ist definiert in [19] und erlaubt eine einfache und verständliche Übertragung. Um die Daten von Java nach JSON und vice versa zu übersetzen, wird der JSON-Prozessor *Jackson* genutzt. Dieser bietet einen Mapper, der die Generierung sowohl von Klassen-basierten Objekten aus ankommenden JSON-Daten als auch von JSON-Strings aus beliebigen Objekten erlaubt.

2.1.3 Schnittstellen

Um einen Datenaustausch in eine beliebige Richtung zu erreichen, müssen zugreifende Systeme die bereitgestellten Schnittstellen nutzen. Das Format für diesen Austausch ist, wie in Abschnitt 2.1.2 angesprochen, JSON. Die Schnittstellen des Prototypen verwenden das in [4] beschriebene *REST-Paradigma*, um eine einfach verständliche und wohldefinierte Kommunikation zu ermöglichen. So genannte *Endpoints*, deren Technologie von Spring zur Verfügung gestellt wird, ermöglichen den Zugriff auf eine oder mehrere Ressourcen, die verwendete URL-Struktur lässt Rückschlüsse auf die Ressource selbst und die Absicht des Zugriffs mittels *Request-Methode* (`GET`, `POST`, `PUT`, `DELETE`) zu.

2.1.4 Sonstige

Weitere wichtige verwendete Technologien umfassen:

- *Apache Maven*[4], als Build-Management-Tool, insbesondere um auf einfache Weise Libraries in den Core einzubinden,
- *Elasticsearch*[5], um Volltextsuche innerhalb des Cores zu ermöglichen (mehr dazu in Abschnitt 6.1).

2.1.5 Die Plattform Moodle

Anfangs schien *Moodle* als fertige Plattform ein geeigneter Kandidat für den CrAc-Core zu sein, vor allem durch die Menge an fertigen Features, die in der Initial-Installation bereits enthalten sind. Diese umfassen:

1. Eine weite Basis an *Management-Funktionalität* (für Aufgaben und Benutzer im System),
2. Eine große Auswahl an weiteren, fertigen Modulen,

[4] https://maven.apache.org/
[5] https://www.elastic.co/de/products/elasticsearch

3. Eine eingebaute *Administrator-Oberfläche*.

Die Abgeschlossenheit der Plattform und der einzelnen Module sind jedoch nicht nur als Vorteil zu sehen, sondern können bei der Einbindung von verschiedenen Konzepten hinderlich kann. Folgende Gründe führten schließlich zu einer Entscheidung gegen *Moodle*:

1. Die Plattform benötigt ein *zeitaufwändiges Einarbeiten* in das System für sämtliche zum Code beitragende Entwickler.
2. Die meisten Module hätten an die Anforderungen angepasst werden müssen, die Anpassung wäre vermutlich aufwändiger als das neue Schreiben der Module.
3. Das *Entkoppeln* von Frontend und Backend gestaltet sich schwierig (dies widerspricht den Anforderungen).

Kapitel 3

Komplexe Datentypen und deren Klassen

Nachdem die technische Basis nun etabliert ist, müssen verschiedene Datentypen und deren dementsprechende Klassen genauer betrachtet werden. Sehr wichtig ist hierbei die Trennung von zusammengesetzten Datentypen oder Datenstrukturen, die in der Architektur des CrAc-Cores als *Software-Module* dienen, und sämtlichen anderen. Um diese Datentypen, die keine kompletten *Module* darstellen, geht es in diesem Kapitel. Auf ihnen bauen alle Module und letztendlich der gesamte Prototyp auf. Sie stellen eine logische Repräsentation verschiedener realer Objekte dar, persistieren Daten und handhaben verschiedenste weitere Prozesse innerhalb der Gesamt-Instanz. Ihre Relevanz wird vor allem später bei der genauen Analyse der einzelnen System-Module noch einmal deutlich. Nachfolgend werden die Datentypen dargelegt und auf ihre Funktionalität, Attribute und Methoden wird eingegangen. Auch das relationale Datenmodell wird in Verbindung mit den zu persistierenden Klassen gezeigt. Dieses ähnelt – aufgrund der intensiven Nutzung von *Hibernate-Automatismen* – deren Aufbau sehr stark. Grundsätzlich wird zwischen folgenden vier Arten von Datentypen unterschieden.

3.1 Hauptdatentypen

Die erste Kategorie von Klassen stellt konkrete Repräsentationen von realen Objekten im Kontext des CrAc-Cores dar. Der Name *Hauptdatentypen* ist dahingehend sprechend gewählt, weil sich sämtliche Prozesse im System um die Verarbeitung und Speicherung von instantiierten Objekten dieser Klassen dreht. Persistiert wird mithilfe von *Hibernate*, Objekte von Hauptdatentypen existieren daher nicht nur im Speicher der Applikation. Verschiedene Methoden der Klassen sollen eine Interaktion mit diesen und ihren integrierten Workflows im Kontext des Frameworks ermöglichen.

3.1.1 User

Diese Klasse ist zuständig für die Speicherung persönlicher und System-bezogener Daten jedes CrAc-Plattform-Users und stellt damit seine virtuelle Repräsentation im CrAc-Core dar. Die gespeicherten Informationen teilen sich hierbei auf verschiedene Bereiche auf.

Identifikations-Daten

Zunächst zu den Informationen, die für eine erfolgreiche Identifizierung benötigt werden:

```
1  private long id;
2  private String name;
3  private String password;
```

Für die Identifizierung des *User-Objektes* wird System-intern eine einfache `id` genutzt. Die Kombination von `name` und `password` hingegen dient dazu, von außen via REST zugreifende Systeme, als einen der User des Systems zu identifizieren. Nur wenn diese Identifikation – durchgeführt von *Spring Security* – erfolgreich abgeschlossen werden kann, wird der Zugriff gewährt.

Persönliche Informationen

Folgende Attribute existieren rein dazu, den jeweiligen User im Kontext des CrAc-Cores zu repräsentieren:

```
1  private String lastName;
2  private String firstName;
3  private Date birthDate;
4  private String status;
5  private String phone;
6  private String address;
```

Rechte im CrAc-System

Jedes User-Objekt kann eine beliebige Menge an Rollen im CrAc-System zugewiesen bekommen, wobei jede Rolle über `id` und `name` (z.B. `USER` oder `ADMIN`) definiert ist. Wie erwähnt wird die zugreifende Software bei einer Request, mittels mitgeschickter Informationen, als eines der User-Objekt im CrAc-Core identifiziert (sofern die mitgeschickten Daten passen). Mittels einer einfachen Abfrage der Rollen dieses Objektes ist es Spring nun möglich, den Zugriff auf bestimmte Bereiche des CrAc-Cores zu erlauben oder zu verbieten:

```
1  Set<Role> roles;
```

Bis auf die Handhabung verschiedener Rollen und damit verbundener Restriktionen hinsichtlich Endpoint-Zugriffs, enthält die *User-Klasse* keinerlei komplexe Workflows, sondern dient rein als *Datencontainer*.

3.1.2 Task

Anders verhält es sich mit Aufgaben. Während Benutzer nur durch ihre Rollen im System definiert sind, werden Aufgaben und ihre Funktionen im Prototypen hauptsächlich von ihren verschiedenen *Zuständen* bestimmt. Auch hier dienen die Objekte im Core als Repräsentation realer Aufgaben und sollen diese als Klasse so gut wie möglich widerspiegeln.

Aufgaben-bezogene Informationen

Ähnlich den Usern besitzen daher auch Aufgaben ein Set an Basis-Daten, die das jeweilige Objekt qualitativ beschreiben, diese werden im folgenden Code dargestellt:

```
1 private String name;
2 private String description;
3 private String address;
4 private Calendar startTime;
5 private Calendar endTime;
6 private int urgency;
7 private int amountOfVolunteers;
8 private TaskType taskType;
```

Darstellung der Struktur

Um einzelne Aufgaben nicht komplett zu überladen, steht der Klasse ein *Verknüpfungs-Mechanismus* zur Verfügung, der auf folgende Attribute zurückgreift:

```
1 private TaskState taskState;
2 private Task superTask;
3 private Set<Task> childTasks;
```

Dies ermöglicht die Auftrennung von Aufgaben in Sub-Aufgaben in Form einer Baumstruktur, welche aus übergeordneten `superTasks` und einer Menge aus untergeordneten `childTasks` besteht. Jede dieser Teilaufgaben kann im Baum eine gewisse Rolle annehmen, welche im Attribut `taskType` abgespeichert wird. Abhängig von diesem Attribut kann eine Task mit anderen Aufgaben bestimmter Typen erweitert werden. Welche Typen existieren und mit welchen Arten von Aufgaben sie erweiterbar sind, wird nachfolgend näher beschrieben.

- `ORGANISATIONAL`: Struktur-definierende Task, die zur Gliederung des Baumes dient.
  - Erweiterung mit weiteren Aufgaben des Typs `ORGANISATIONAL` möglich.
  - Erweiterung mit Aufgaben des Typs `WORKABLE` möglich.
- `WORKABLE`: Unterste Ebene des Baumes und damit feinst-granulare Gliederung der Aufgabe.

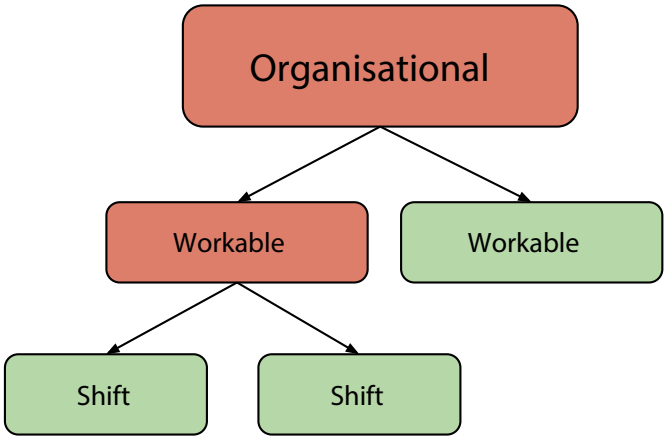


Abbildung 3.1: Eine Gesamtaufgabe als Baum dargestellt.

  - Erweiterung mit Aufgaben des Typs `SHIFT` möglich.
- `SHIFT`: Kann eine weitere Pseudo-Ebene unter einer Aufgabe des Typs `WORKABLE` bilden, um diese in unterschiedliche zeitliche Schichten aufzuteilen.

In Abbildung 3.1 ist ein solcher Baum beispielhaft abgebildet.

Darstellung des Status

Eine eindeutige Auftrennung des Aufgaben-Typs ist notwendig, um den Workflow bei einer Statusveränderung zu vereinfachen. Ohne einen zugewiesenen `TaskType` müsste der CrAc-Core bei jeder aufgabenspezifischen Änderung auf anderen Wegen (z.B. Prüfung auf vorhandene `childTasks`) herausfinden, welche Aufgabe welche Funktion (`WORKABLE` oder `ORGANISATIONAL`) inne hat.

Um eine Statusveränderung darstellen zu können, muss natürlich auch ein Status, oder `TaskState` existieren. Eben dieser Status einer Aufgabe kann vier verschiedene Zustände einnehmen, die ineinander übergehen, um so dem System den Fortschritt innerhalb der Aufgabe mitteilen zu können.

- `NOT_PUBLISHED`: Eine Aufgabe im Rohzustand, der sämtliche Daten (inklusive Basisdaten) fehlen.
- `PUBLISHED`: Eine mit Basisdaten bestückte Aufgabe, die sämtlichen Benutzern des CrAc-Cores öffentlich gemacht wird.
- `STARTED`: Eine Aufgabe, deren startTime eingetreten ist und deren Teilnehmer somit daran arbeiten.
- `COMPLETED`: Eine beendete und archivierte Aufgabe.

Abhängig vom Status sind bestimmte Aktionen auf den Tasks anwendbar

oder nicht anwendbar.

Eine spezielle Rolle nimmt das Attribut `readyToPublish` ein, deren Bedingungen erfüllt werden müssen, um eine Aufgabe von `NOT_PUBLISHED` auf `PUBLISHED` setzen zu können. Der folgende Code zeigt die beschriebenen Attribute und Methoden, welche zum Wechseln zwischen den unterschiedlichen Zuständen einer Aufgabe verwendet werden.

```
1  private TaskState taskState;
2  private boolean readyToPublish;
3
4  //Die zurückgegebene Integer signalisiert
5  //das Resultat der Status-Veränderung
6  //1... Aufgabe nicht zur Änderung bereit
7  //2... Baum der Aufgabe nicht zur Änderung bereit
8  //3... Statusänderung erfolgreich durchgeführt
9  public int publish() {}
10 public int start() {}
11 public int complete() {}
```

Weitere Informationen zu Aufgaben-Workflow sind bei den Beziehungs-Datentypen in Abschnitt 3.2 zu finden.

3.1.3 Competence

Der dritte Hauptdatentyp und gleichzeitig das verbindende Glied zwischen Task und User ist die Kompetenz. Erst durch die Nutzung dieser Klasse wird eine Interaktion zwischen den anderen Hauptdatentypen (z.B. *Matching*) möglich gemacht. Weiters handelt es sich bei diesem Datentyp nicht um eine komplett selbst entwickelte Klasse, sondern um eine Modifikation eines allgemeinen Standards [1]. Dieser Standard beinhaltet beschreibende Elemente, wie einen `name`, setzt aber auch *messbare Abhängigkeiten* zu Benutzern und Aufgaben voraus. Daraus folgt, dass eine Kompetenz, um ihren Zweck erfüllen zu können, mit einem oder mehreren Usern und/oder Tasks in *Verbindung* stehen muss und diese Verbindung beweisbar ist, sprich auf sie zugegriffen werden kann. Dies führt zu zwei Varianten von Zuordnungen.

- Die Verbindung einer Kompetenz mit einer Aufgabe ist gleichbedeutend mit einer *Voraussetzung* für diese Aufgabe.
- Die Verbindung einer Kompetenz mit einem Benutzer ist gleichbedeutend mit einer *gelernten Fähigkeit*.

Was den allgemeinen HR-XML-Standard und die Modifikation des CrAcCores unterscheidet, ist die *Auslagerung* der beschriebenen Verbindungen und der *Verzicht* auf einen verpflichtenden „Kompetenz-Beweis“. Ein zu erbringender Beweis für eine meist triviale gelernte Fähigkeit ist für das Freiwilligen-Umfeld größtenteils nicht erforderlich und sorgt im schlechtesten Fall für großen Mehraufwand. Sollte sich dieser Verzicht als Fehleinschätzung herausstellen, lässt sich das System auf Kompetenzen mit Beweispflicht erweitern.

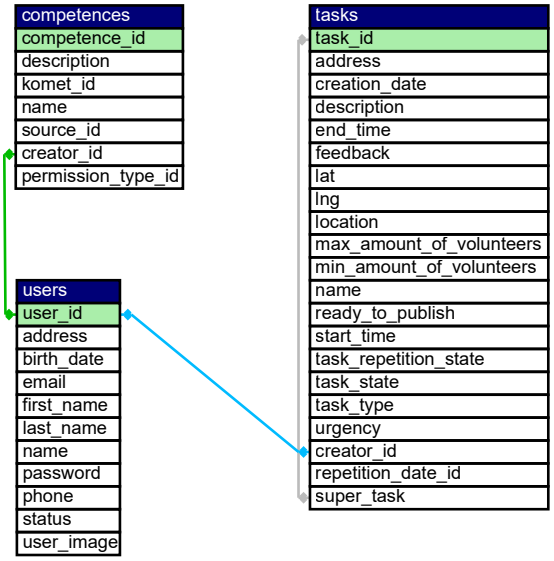


Abbildung 3.2: Die Hauptdatentypen.

Der eigentliche Aufbau der *Kompetenz-Klasse* ist, wie im folgenden Code zu sehen, recht primitiv und gewinnt erst mit den im nächsten Abschnitt vorgestellten *Beziehungsdatentypen* an Tiefe. Hier offenbart sich auch das Potential hinter der Vernetzung sämtlicher Hauptdatentypen.

```
1  private String name;
2  private String description;
```

Davor noch ein kurzer Blick auf das Modell der Hauptdatentypen in Abbildung 3.2.

3.2 Beziehungsdatentypen

Um die im Kompetenz-Standard beschriebenen Abhängigkeiten umzusetzen, aber so flexibel wie möglich zu halten, existiert im CrAc-Core eine weitere Kategorie an Datentypen. Diese *Beziehungsdatentypen* repräsentieren im Gegensatz zu den Hauptdatentypen keine konkreten Objekte, sondern deren Beziehungen. Die Existenz eines solchen Objektes dient jedoch nicht nur als Beziehungsbeweis, sondern auch zur Speicherung von zusätzlichen *Meta-Daten*, die in verschiedenen Prozessen und von verschiedenen Modulen verarbeitet werden. Die Speicherung erfolgt auch hier wieder via *Hibernate* in die angeschlossene Postgres-Datenbank. Der Ansatz der Anreicherung des Systems mittels Meta-Daten stammt aus dem Bereich des *Crowdsourcings*, dieser wird später in Abschnitt 4.1.3 genauer erklärt. Die einzelnen Werte die gespeichert werden bzw. ihre Bedeutung auf verschiedene Prozesse, sind

angelehnt an Vorschläge aus [10].

Competence-User-Relationship

Dieser erste, näher analysierte Beziehungsdatentyp repräsentiert die Verbindung zwischen einer Kompetenz und einem Benutzer. Wie bereits angemerkt, hat diese Verbindung den Stellenwert einer vom User gelernten Fähigkeit, die er nun einbringen kann. Zusätzlich zu diesem Beweis werden folgende Meta-Daten persistiert. Der folgende Code zeigt die Attribute der Beziehung:

```
1  // Wert zwischen -100 und 100
2  private int likeValue;
3  // Wert zwischen 0 und 100
4  private int proficiencyValue;
```

Das `likeValue` gibt Auskunft darüber, wie sehr einem User eine Kompetenz gefällt. Etwas zu mögen ist schließlich nicht gleichbedeutend damit, etwas zu können. Hier greift `proficiencyValue`, das aussagt, wie sehr der User die Kompetenz beherrscht. Dies gibt die Möglichkeit, zwischen verschiedenen Stufen einer Fähigkeit unterscheiden zu können.

Competence-Task-Relationship

Dem gegenüber steht die Verbindung zwischen Kompetenz und Aufgabe, die gleichbedeutend mit einer Anforderung an die Aufgabe ist und folgende Meta-Daten speichert:

```
1  // Wert zwischen 0 und 100
2  private int neededProficiencyLevel;
3  // Wert zwischen 0 und 100
4  private int importanceLevel;
5  private boolean mandatory;
```

Hier wird die Anforderung an den Benutzer, der an der jeweiligen Aufgabe teilnehmen möchte, noch spezifiziert. `neededProficiencyLevel` dient als Vergleichswert zum `proficiencyValue` des potentiellen Teilnehmers, während das `importanceLevel` die Wichtigkeit der Kompetenz im Kontext der Task festlegt. Eine spezielle Position nimmt die Flag `mandatory` ein, die eine Kompetenz für eine Aufgabe als verpflichtend auszeichnen kann.

User-Task-Relationship

Über die in [1] definierten Verbindungen hinaus kann die Idee der Verknüpfung von Datentypen noch weiter gesponnen werden. Da die Teilnahme eines Users an einer Aufgabe so oder so festgehalten werden muss, kann auch dies in Form einer solchen Relationship geschehen und eröffnet damit gleichzeitig eine ganze Reihe von weiteren Möglichkeiten. Anbei Attribute der Klasse:

```
1 private TaskParticipationType participationType;
2 private boolean completed;
```

Mithilfe dieser Klasse kann die Art der Teilnahme eines Users an einer Aufgabe festgehalten werden. Das Attribut `participationType` kann dabei verschiedene Zustände annehmen – dies ist relevant für den *Task-Workflow*.

- **Following**:
  - Dieser Typ beschreibt keine tatsächliche Teilnahme, sondern eine *Interessenbekundung*.
  - Der User signalisiert mit dem *Folgen* eine mögliche Teilnahme an einer konkreten Aufgabe oder deren Sub-Aufgaben.
  - Das Folgen einer Aufgabe durch einen Benutzer ist an Aufgaben jeden `TaskTypes` möglichen.
  - Erst möglich, wenn eine Aufgabe veröffentlicht wurde, siehe Abschnitt 3.1.2.
- **Participating**:
  - Dieser Typ beschreibt die aktive, freiwillige *Teilnahme* eines Users an einer Aufgabe.
  - Die aktive Teilnahme ist nur an einer *Workable-Task* oder einer *Shift* möglich.
  - Ein Benutzer kann an einer Aufgabe teilnehmen, wenn sein *Matchingergebnis* (siehe Kapitel 4) einem Wert über null entspricht.
  - Ebenfalls erst möglich, wenn eine Aufgabe veröffentlicht wurde.
- **Leading**:
  - Dieser Typ beschreibt eine *führende Tätigkeit* innerhalb des Aufgaben-Baums.
  - Dieser Typ kann in Verbindung mit jedem `TaskType` existieren.
  - Dieser Verbindungstyp gibt dem jeweiligen Benutzer das Recht, die Struktur der angeführten Aufgabe und deren Sub-Baumes verändern zu können.
  - Dies umfasst das *Hinzufügen/Löschen/Anpassen* von Aufgaben und das Hinzufügen von zusätzlichen *Leadern* auf einzelne Aufgaben-Knoten.
  - Der Ersteller der ersten Aufgabe in einem Aufgabenbaum wird automatisch Leader von dieser (und hat damit die Kontrolle über den gesamten Baum).
  - Diese Operationen können bereits vor *Veröffentlichung* einer Task durchgeführt werden (Zustand der Aufgabe kann `UN_PUBLISHED` sein).
  - Diese Rolle ist speziell auf die richtige Strukturierung und den Aufbau einer Task ausgelegt.

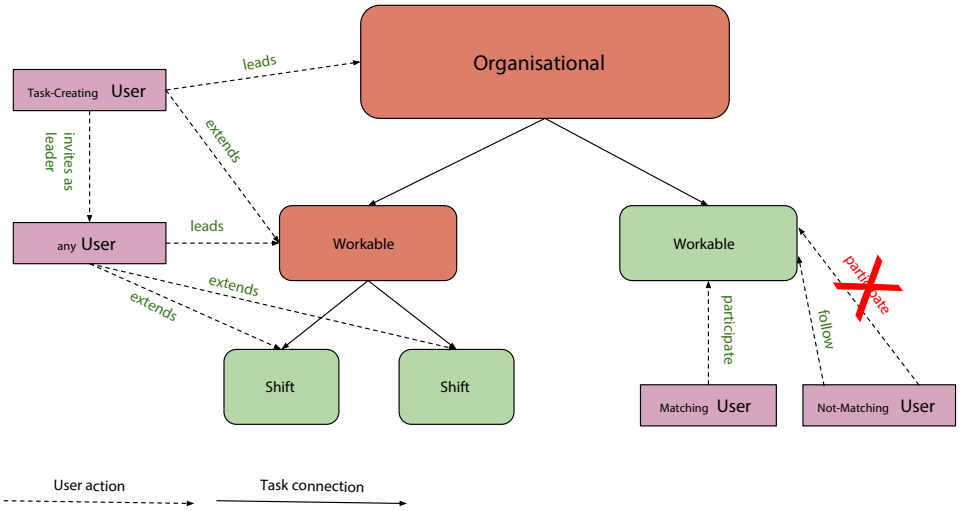


Abbildung 3.3: Eine Gesamtaufgabe mit verbundenen Benutzern als Baum dargestellt.

In Abbildung 3.3 ist ein Aufgaben-Baum mit verschiedenen verbundenen Usern abgebildet. An Aufgaben, an denen Freiwillige tatsächlich arbeiten, zeigt außerdem die Flag `COMPLETED`, ob die einzelnen User ihre Arbeit bereits beendet haben. Dies beschreibt einen Teil des Übergangs von `STARTED` zu `COMPLETED`, siehe Abschnitt 3.1.2.

Competence-Relationship

Beziehungsdatentypen müssen allerdings nicht immer explizit zwei oder mehr verschiedene Hauptdatentypen verbinden. Auch Verbindungen innerhalb einer einzelnen Hauptklasse können interessante und relevante Meta-Daten speichern. Dies betrifft unter anderem die Repräsentation der Beziehung zwischen zwei Kompetenzen. Diese stellt über den anbei dargestellten `CompetenceRelationshipType` die Verwandtschaft zwischen zwei beliebigen Kompetenzen dar, eine Information, die für das *Matching* (s. Kapitel 4) unverzichtbar ist.

```
1  private CompetenceRelationshipType type;
```

Diese enthält die Referenz zu einem `CompetenceRelationshipType`, welcher wiederum die tatsächlichen Beziehungsdaten, dargestellt in folgendem Code, speichert.

```
1  private String name;
2  private String description;
3  private double distanceVal;
```

Durch die Auslagerung vereinheitlichter Kompetenz-Verbindungstypen wird

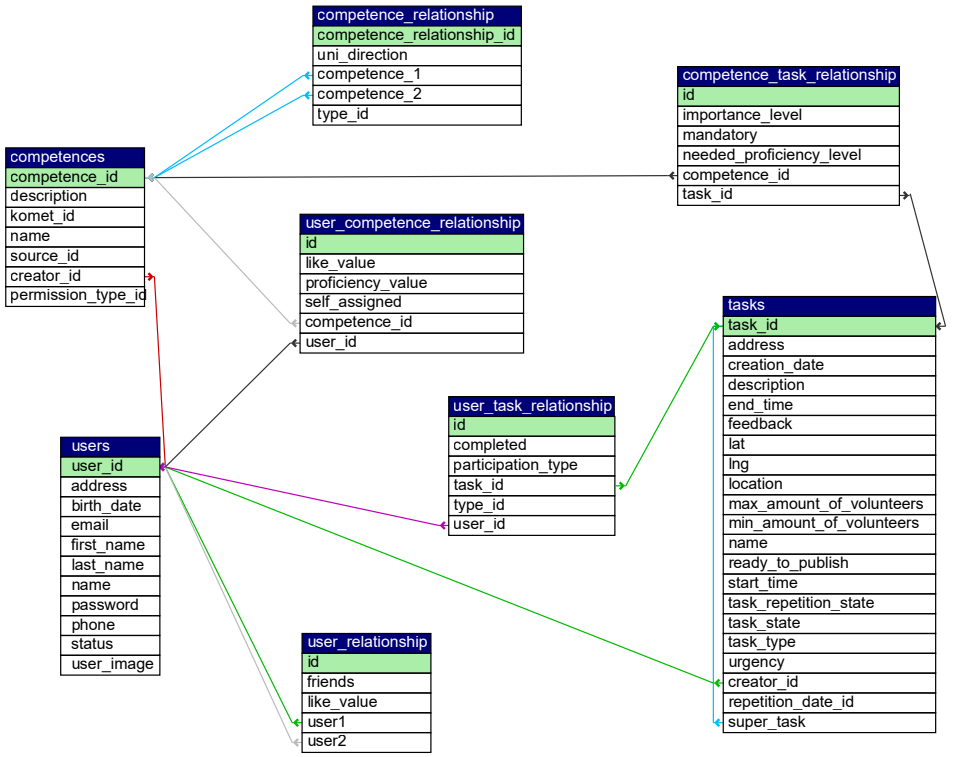


Abbildung 3.4: Die via Beziehungsdatentypen verbundenen Hauptdatentypen.

eine leichtere *Verständlichkeit* und ein *verringerter Aufwand* für den Endnutzer erreicht.

User-Relationship

In die gleiche Richtung geht die User-Relationship, eine Repräsentation der Beziehung zwischen zwei Benutzern. Statt des Verwandtschaftsgrades speichert sie allerdings, wie die Attribute im nachfolgenden Code zeigen, wie sehr zwei User zusammenpassen und sich mögen/miteinander arbeiten können.

```
1  private double likeValue;
2  private boolean friends;
```

Hierbei wird unterschieden zwischen dem Attribut `likeValue`, das unabhängig speichert, wie gern sich Benutzer mögen oder nicht mögen, und dem Attribut `friends`, welcher nur bei Zustimmung beider Beteiligter in der User-Relationship explizit auf befreundet gesetzt werden kann. Abbildung 3.4 zeigt erneut das Modell der Hauptdatentypen, diese sind diesmal jedoch verbunden mit den hier erklärten Beziehungsdatentypen.

3.3 Sonstige Datentypen

Es existiert noch eine weitere Kategorie von Datentypen im Core, deren gemeinsamer Funktionsumfang sämtliche nicht von den ersten beiden Datentypen abgedeckte Bereiche umfasst. Meist handelt es sich bei ihnen um Helper-Objekte integraler Bestandteile von Modulen und deren Prozessen. Da Objekte dieser Klassen nur existieren, um in ihrer Lebensspanne einzelne Core-interne Aufgaben auszuführen, werden sie auch in keinster Weise persistiert. Für eine bessere Veranschaulichung sind Datentypen der Kategorie *Sonstiges* noch einmal weiter unterteilt.

3.3.1 Schnittstellen

In die erste Kategorie fallen Klassen, die als Schnittstelle zu einer verbundenen Software-Instanz dienen und ein passendes Interface zum Zugriff darauf zur Verfügung stellen.

Crud-Repositories

Um mittels JPA Zugang zu der Datenbank zu erhalten, werden sogenannte *Crud-Repositories*[1] verwendet. Die genutzte, übergeordnete Klasse ist *Teil von JPA* und wird jeweils für die verschiedenen Haupt- und Beziehungsdatentypen vom CrAc-Core implementiert. Diese Repository-Klassen werden beim Start des Spring-Frameworks automatisch mit den nötigen Informationen befüllt und dienen jeweils dem Zugriff auf die Tabelle ihres Datentyps in der Datenbank. Durch die kombinierte Benutzung von *JPA und Hibernate* werden sämtliche SQL-Aufrufe maskiert und die Ergebnisse werden direkt in Java-Objekte konvertiert, die die Repositories an die aufrufende Instanz zurück liefern. Zum besseren Verständnis ein Beispiel:

```
1 @Transactional
2 public interface CracUserDAO extends CrudRepository<CracUser, Long>{
3    public CracUser findByName(String name);
4    public CracUser findByNameAndPassword(String name, String password);
5 }
```

Dies ist eine beispielhafte Implementierung eines solchen `CrudRepository`, welches Zugriff auf die User der Datenbank gewährt. Zusätzliche Zugriffsmethoden werden via *Methoden-Name* definiert, bei der Ausführung des Frameworks auf Korrektheit geprüft und im erzeugten Objekt umgesetzt. Weitere Standard-Methoden werden hinzugefügt. Basierend auf diesem Beispiel stehen am Ende folgende Methoden zur Verfügung:

```
1 public List<CracUser> findAll(){}
2 public CracUser findOne(Long id){}
```

[1] https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

```
3  public CracUser findByName(String name){}
4  public CracUser findByNameAndPassword(String name, String password){}
```

Auto-generiert stehen `findAll()`, um sämtliche User, und `findOne(Long id)`, um einen einzelnen User mithilfe seiner ID aus der Datenbank zu laden, bereit. Beide werden *unabhängig vom Datentypen* bei jedem CrudRepository-Objekt erzeugt. `findByName(String name)` und `findByNameAndPassword (String name, String password)` werden hingegen wie oben beschrieben, basierend auf ihrem Methoden-Name zusätzlich erzeugt.

Elastic-Connector

Der Elastic-Connector stellt eine Schnittstelle und Maske zur verbundenen Instanz von *Elasticsearch* dar. Im Hintergrund greift die Klasse auf die *Java-API* von Elasticsearch zurück und öffnet und schließt die Verbindung bei jedem Zugriff. Um ein konsistentes Softwaredesign zu erreichen und die Interaktion stark zu vereinfachen, ist der Elastic-Connector dem CrudRepository von JPA nachempfunden. Sie ist zudem *generisch* und abhängig vom zu speichernden Datentyp. Folgende Attribute müssen gesetzt werden, um ein Connector-Objekt zu erstellen:

```
1  //In der Elasticsearch genutzte Index
2  private String index;
3  //Im Index genutzter komplexer Datentyp
4  private String type;
5  //Address und Port um Zugriff zu Elasticsearch zu erhalten
6  private String address;
7  private int port;
```

`Index`, `Address` und `Port` sind fix vom CrAc-Core vorgegeben, das Attribut `Type` wird abhängig vom zu persistierenden Datentyp gesetzt.

```
1  //T bezeichnet den generischen Datent
2  indexOrUpdate(String id, T obj)
3  get(String id)
4  delete(String id)
5  query(String searchText)
```

Dem fertigen Objekt stehen einige Methoden zur Wartung `indexOrUpdate, get, delete`) und eine `query`, welche das Durchsuchen des Index mittels Volltextsuche ermöglicht, zur Verfügung.

3.3.2 Process-Helpers

Weiters existieren einige Klassen im Core, die den *Matching-Prozess*, also das intelligente Zuweisen von Aufgaben an Benutzer (s. Kapitel 4), sowie dessen Anpassungsmechanik (s. Kapitel 5) unterstützen bzw. die Grundlage dafür schaffen.

Worker

Objekte dieser Klasse bilden eine eigene Schicht an Prozessen im System. Diese „Arbeiter-Objekte“ werden für genau eine einzige Aufgabe instanziiert und nach deren Ausführung wieder zerstört. Die Erfüllung der Aufgabe geht möglicherweise, aber nicht zwangsweise mit einem Rückgabewert einher, der an das aufrufende System weitergegeben wird. Sämtliche *Worker* werden auf dieselbe Art und Weise aufgerufen und ausgeführt und müssen daher eine gleichbleibende Schnittstelle bieten. Um dies zu ermöglichen, *extenden* sie von einer übergeordneten Klasse (siehe folgender Code) und überschreiben die `run()`-Methode, die letztendlich vom Core ausgeführt wird.

```
1 public abstract class Worker {
2   private String workerId;
3   public Worker(){
4     this.workerId = CoreHelper.randomString(20);
5   }
6   public abstract Object run();
7 }
```

Jedes dieser *Worker-Objekte* erhält eine einzigartige `workerId` und einen auszuführenden Code (in der überschriebenen `run()`-Methode) , abhängig von der Worker-Klasse. Der Zugriff und die Modifikation sämtlicher Daten des Cores sind innerhalb eines Workers möglich. Die Aufteilung von Code in einzelne Worker dient vor allem der Modularisierung von Core-Funktionen. Verschiedene Implementierungen, vor allem wenn sie nicht durch die Controller-Struktur trennbar sind, können auf diese Weise klar voneinander getrennt gespeichert und ausgeführt werden. Weiters kann mithilfe einer übergeordneten Datenstruktur die Reihenfolge der Code-Ausführung bestimmt und modifiziert werden. Die Aufteilung in Worker macht den Core daher besser skalierbar und trägt zur geforderten Code-Wiederverwendbarkeit (s. Abschnitt 1.3.1) bei.

Matching-Matrix

Die Matching-Matrix kontrolliert die mathematischen Vorgänge während des Matching-Verfahrens und wird in diesem Zusammenhang meist, aber nicht zwangsweise, von einem implementierten *Worker* aufgerufen.

Innerhalb jedes instanziierten Objektes wird die Struktur aufgebaut, die nötig ist, um eine einzelne Task mit einem einzelnen Benutzer zu vergleichen. Im Umkehrschluss bedeutet dies, dass die Anzahl der Matrizen, die in einem kompletten Matching-Prozess benötigt wird, dem Produkt der jeweiligen Menge an Usern und Aufgaben entspricht. Diese werden mit Abschluss des Matchings automatisch von *Java* entfernt.

Wie der folgende Code der *Matching-Matrix-Klasse* zeigt, enthält der Datentyp nicht nur die Daten-Matrix in Form des zweidimensionalen Arrays `MatrixField[][] matrix`, sondern speichert auch Referenzen zum ver-

knüpften User in `CracUser u` und zur Aufgabe in `Task t`, sowie Informationen darüber, ob der Benutzer an der Aufgabe teilnehmen kann, in `boolean doable`. Die Funktionsweise hinter der Liste `mandatoryViolations`, welche die bei der Validierung der Matrix gefundenen Regelverstöße speichert, wird in Kapitel 4 näher erklärt.

```
1  private MatrixField[][] matrix;
2  private CracUser u;
3  private Task t;
4  private boolean doable;
5  private ArrayList<String> mandatoryViolations = new ArrayList<>();
```

Nachfolgende Mathoden werden genutzt, um die interne `matrix` via `buildMatrix()` zu erzeugen, sie via `markMandatoryViolation()` zu validieren und mit Hilfe von `applyFilters()` zu modifizieren.

```
1  private void buildMatrix() {}
2  private void applyFilters(FilterConfiguration m) {}
3  private void markMandatoryViolation() {}
```

Nach Ausführung aller Schritte kann die `calcMatch()`-Methode als Schnittstelle genutzt werden, um den aus der Matrix errechneten *Matching-Score* auslesen zu können. Weitere Informationen zur Matrix und ihrem Teil des Matching-Prozesses sind in Kapitel 4 zu finden.

```
1  public double calcMatch() {}
```

Matching-Filter

Mit der Matrix stark in Verbindung stehen die *Matching-Filter*, die einen weiterer Faktor beim Errechnen des finalen *Matching-Scores* darstellen. Um das Matching nicht rein Kompetenz-basiert zu gestalten, existieren verschiedene Filter-Klassen, die ähnlich den *Worker-Klassen* in einer als Schnittstelle dienenden `apply()`-Methode die Werte der Matching-Matrix nachträglich justieren. Die Art der Modifikation hängt rein vom Filter (siehe Code) und dessen implementiertem mathematischen Modell ab.

```
1  public abstract class CracFilter {
2    private String name;
3    public abstract double apply(MatrixField m);
4  }
```

Jeder der Filter leitet sich aus diese Klasse ab und überschreibt die `apply()`-Methode, die im laufenden Matching auf jedes Feld der Matrix angewandt wird. Auch hier wird auf Kapitel 4 verwiesen, in dem näher auf den *Matching-Prozess* eingegangen wird.

Notification

Ein weiterer wichtiger Faktor innerhalb des CrAc-Cores ist die Weitergabe von Informationen an beliebige Benutzer. Aus diesem Grund stellt die

Applikation verschiedene *Notification-Klassen* zur Verfügung. Objekte dieser Klassen können entweder von einem Benutzer oder dem System selbst erstellt werden, liegen im Applikations-Speicher und können von ihrem *Ziel-User* – der User, an den die Notification gerichtet ist – abgerufen werden. Nachfolgend der Code der abstrakten Klasse:

```
1  public abstract class Notification {
2
3    private String notificationId;
4    private Long targetId;
5    private Calendar creationTime;
6    private String name;
7    private NotificationType type;
8
9    public Notification(String name, NotificationType type, Long targetId)
       {
10     this.name = name;
11     this.type = type;
12     this.creationTime = Calendar.getInstance();
13     this.notificationId = NotificationHelper.randomString(20);
14     this.targetId = targetId;
15   }
16
17   public abstract String accept();
18   public abstract String deny();
19 }
```

Objekte der jeweiligen Klassen enthalten eine einzigartige `id`, einen `name` und einen `type` für die Identifikation, sowie – mit der `targetId` – die ID ihres Ziel-Users. Da es sich bei *Notification* um eine abstrakte Klasse handelt, können die implementierten Notifications beliebige weitere Attribute besitzen. Jede davon hat außerdem die Möglichkeit, in den beiden Methoden `accept()` – falls der Ziel-User die Notification bestätigt – und `deny()` – falls er sie ablehnt – beliebigen, Notification-abhängigen Code auszuführen.

3.3.3 Input/Output-Datentypen

In die letzte Kategorie fallen Datentypen, die keinen weiteren Zweck im Core erfüllen, als strukturierten Input oder Output zu erzeugen. Gebraucht werden diese Klassen in der Datenübertragung, um entweder vom zugreifenden System oder vom Core selbst mit Daten befüllt zu werden.

REST-Response

Die REST-Response Klasse existiert, um den *JSON-Output* des Prototypen konsistent zu halten. Jedes vom CrAc-Core versandte Dokument ist eine in JSON umgewandelte *REST-Response*.

Dies führt zu einer gleichbleibenden, Endpoint-unabhängigen Struktur, die dem Empfänger eine *Vorhersehbarkeit* in der Kommunikation mit der

Schnittstelle bietet. Gleichzeitig wird hiermit dem Teil von Anforderungen (s. Abschnitt 1.3.1) Genüge getan, der sich mit dem Datenaustausch auseinandersetzt. Der folgende Code zeigt die einzelnen Attribute der *REST-Response*:

```
1 private String type;
2 private RESTAction rest_action;
3 private boolean success;
4 private ArrayList<RESTError> errors;
5 private T object;
6 private HashMap<String, Object> meta;
```

Unabhängig vom aufgerufenen Endpoint wird immer versucht, sämtliche dieser Attribute zu füllen. `type` beschreibt, welche Art von Objekt in `object` zurück geliefert wird. Die `rest_action` beschreibt die Art des Zugriffs (GET, POST, PUT, DELETE). In `success` wird der Erfolg des Zugriffs aufgezeichnet und im Falle eines Fehlschlages in der Liste `errors` detailliert beschrieben.

Das letzte Attribut `meta` dient als Allzweck-Container, in dem zusätzliche, Endpoint-relevante Informationen aller Art gespeichert werden können. Anbei wird gezeigt, wie der Output eines solchen JSON-Objektes aussehen kann:

```
1  {
2    "type": "Task",
3    "rest_action": "GET",
4    "success": true,
5    "errors": [],
6    "object": {
7      "id": 3,
8      "name": "Titel der Aufgabe",
9      "description": "Beschreibung der Aufgabe",
10     ...
11   },
12   "meta": {}
13 }
```

Post-Options

Die Post-Options bieten dem Framework eine Möglichkeit zum Mappen von Input, der nicht dediziert für einen einzigen Datentypen bestimmt ist. Dies ist nötig, um ein *Mapping* via Jackson zu ermöglichen, da ansonsten direkt auf das JSON-Dokument zugegriffen werden müsste. Um dies zu vermeiden und damit den CrAc-Core konsistent zu halten, existieren die *Post-Options*.

Die Klasse enthält eine Vielzahl von Feldern, die je nach konsumierten Daten befüllt werden. Das daraus erzeugte Java-Objekt kann anschließend vom Core verwendet werden und bietet einen einfachen Zugriff auf den Input. Das spezielle an den Post-Options ist, dass sie komplett beliebig erweiterbar sind, jeder Endpoint nutzt nur jeweils die Attribute, die er selbst befüllt.

Styling-Klassen

Für einige Klassen, speziell Aufgaben, existieren eigene *Styling-Klassen*. Bei der Ausgabe durch ein REST-Response-Objekt wird nicht das eigentliche Objekt übergeben, sondern eine modifizierte Version. Dabei werden alle relevanten Daten an die entsprechende Styling-Klasse übergeben und das erzeugte, strukturell veränderte Objekt wird zum neuen Output. Dies hat den Sinn beliebige Meta-Daten ein- und ausblenden zu können, ohne direkt auf das für den Output gedachte JSON-Dokument zugreifen zu müssen. Ein Beispiel hierbei wäre die unterschiedliche Darstellung von *Aufgaben*, je nach aufgerufenem Endpoint.

Kapitel 4

Matching

Das Kernstück dieses Prototypen, des CrAc-Cores, ist die Zusammenführung verschiedener *Konzepte*, um ein Kompetenz-basiertes Matching zu ermöglichen. Die hierfür nötigen Datentypen und Klassen, sowie Aspekte einiger Prozesse sind bereits bekannt (s. Abschnitt 3.3.2) und werden hier genauer ausgeführt. Kompetenz-basiert bedeutet, dass das von den kombinierten Verfahren erzeugte Endergebnis den Datentyp „Kompetenz“ unbedingt als *Haupteinflussfaktor* beinhalten muss, da er mit seinen Beziehungen darstellt, wie geeignet verschiedene Benutzer für verschiedene Aufgaben sind und vice versa. Diese Konzepte, ihre mathematischen Grundlagen und Modelle, sowieso ihre Implementierungen im CrAc-Core und seinen Modulen werden in diesem Abschnitt präsentiert und analysiert. Sämtliche Ergebnisse des *Matching* werden als *Recommendation* gehandelt. Sie werden dem Benutzer also vorgeschlagen, nicht explizit zugewiesen. Da es sich um freiwillige Helfer und keine Angestellten handelt, können und sollen sie aus der zusammengestellten *Task-Liste* beliebig auswählen können. Weiters ist der Matching-Prozess Teil eines *Kreislaufs* (siehe Abbildung 4.1) bestehend aus *Matching, Ausführung, Feedback und Datenmodifikation*, der mit zunehmender Wiederholung einen immer genaueren Matching-Prozess ermöglicht.

Zum besseren Verständnis wird kurz auf den Kreislauf und seine vier Bestandteile eingegangen:

- **Matching**: Den Startpunkt markiert der Matching-Prozess in seiner Gesamtheit, siehe Kapitel 4.
- **Work**: Der nächste Schritt beschreibt die gesamte Interaktion des Benutzers mit der Aufgabe – von der Anmeldung bis zur Komplettierung. Der Workflow im Hintergrund ist beschrieben bei den Datentypen in Kapitel 3.
- **Feedback**: Der dritte Punkt beschreibt die Prozesse zuständig für die Evaluierung der User-Erfahrung. Dies umfasst sowohl die eigentliche Aufgabe, an der teilgenommen wurde, als auch die Mitarbeiter dieser, und ist das Kernthema in Abschnitt 5.2.

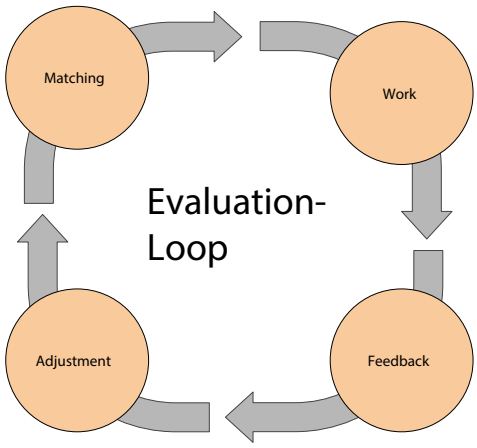


Abbildung 4.1: Der Matching - Feedback Kreisauf.

- **Adjustment**: Im letzten Schritt kümmert sich der Core um die Auswertung des Feedback und die Modifikation der im Matching verwendeten Daten (s. Abschnitt 5.2).

Nach diesem Überblick, nun zu unserem Einstiegspunkt im Kreislauf, dem *Matching*.

Besonders wichtig ist in diesem Zusammenhang die Qualität und die Relevanz der im Hintergrund verwendeten Daten und der Berechnungsmodelle.

- Die Ergebnisse des Matching-Prozesses müssen die Realität richtig widerspiegeln.
- Alle relevanten Daten müssen für den Prozess genutzt werden.
- Die Modelle und Daten müssen neuen Anforderungen angepasst werden können, sollten diese anfallen.

Um ein funktionierendes und skalierbares Matching zu ermöglichen, muss ein System gefunden werden, welches diesen gestellten Anforderungen standhalten kann. Bevor nachfolgend näher auf die umgesetzten Ansätze eingegangen wird, sind hier einige getestete Alternativen aufgelistet, die den Anforderungen im Endeffekt jedoch nicht standhalten konnten.

4.1 Alternative Ansätze

4.1.1 Onthologien

Bei der ersten Alternative handelt es sich um fertige Implementierungen so genannter *Onthologien* [6], welche Teil des *Semantic Web* [3] sind. Diese Systeme zur geordneten Datenspeicherung besitzen im Gegensatz zu herkömmlichen Datenbanken eine weitere Ebene, in der die Beziehungen zwi-

schen verschiedenen Datentypen und die Regeln über deren Zusammenhang beschrieben werden.

Mithilfe einer ausführlichen Beschreibung von Daten und Regeln kann die Onthologie nachfolgend durch *Inferenz* ableiten, in welchem Zusammenhang *andere* Daten stehen. Somit ist eine selbstständige Berechnung von zusammenhängenden Nutzern und Aufgaben möglich, [7] zeigt hier verschiedene Herangehensweisen und Ansätze. Trotz Alter und Potenzial des Konzeptes, sind fertig implementierte Onthologien nicht in jeder Programmier- und Skriptsprache zu einem befriedenden Grad umgesetzt. Wie bereits angemerkt gehört jedoch *Java* zu den Sprachen mit einer ausreichenden Implementierung und bietet mit *Apache Jena* eine API, die den einfachen Zugriff auf derartige Technologie unterstützt. Die Verwendung dieser API, in Verbindung mit dem Onthologie-Editor *Protégé*[1] und der Graphdatenbank *Neo4j*, wird in [10] detailliert dargestellt. Bei der in genannter Arbeit dargestellten Software *CrAc-core* handelt es sich im Übrigen nicht um den hier beschriebenen Prototypen, sondern lediglich um ein Experiment auf Basis der beschriebenen Techniken im Rahmen des CrAc-Gesamtprojektes. Lediglich die Namensgebung wurde übernommen. Obwohl der Einsatz einer implementierten *Onthologie* einiges an Potenzial verspricht, wurde die Idee im Zusammenhang mit dem hier präsentierten Prototypen zugunsten anderer Ideen verworfen. Dies sind die Gründe:

- Das System benötigt ein sehr aufwändiges Setup mit eigener Graph-Datenbank neben einer trotzdem notwendigen relationalen Backend-Datenbank.
- Ein großes, sich ständig weiter entwickelndes Projekt in der Onthologie-eigenen Sprache *OWL* zu beschreiben, bedeutet eine lange Einarbeitungszeit für jeden involvierten Entwickler.
- Das System ist eine Blackbox, deren Output akzeptiert werden muss, Anpassungen sind nur im OWL-File möglich.
- Scoring-Ergebnisse müssen im Nachhinein abhängig von den gewünschten Modifikationen angepasst werden, was den Mehrwert in Relation zum Mehraufwand mindert.

Trotz allem bergen fertig implementierte *Onthologien* großes Potential in diese Richtung. Um sie rein für einen Matching-Prozess zu verwenden, ist der Aufwand jedoch in jeglicher Hinsicht zu groß.

4.1.2 Elasticsearch als Matching-Engine

Dieser Abschnitt beschäftigt sich mit der – nach [15] – beliebten Suchmaschine *Elasticsearch* als Matching-Engine. Diese bietet eine große Menge an verschiedenen *Such-Queries* und ist ausgelegt für die *Volltext-Suche*, also optimiert für das Finden von Texten auf Basis von Teiltexten. Obwohl die

[1] http://protege.stanford.edu/

Funktionalität von Elasticsearch dahingehend nicht vereinbar mit dem Matching vom CrAc-Core wirkt, lassen sich die Queries auf verschiedenste Arten mit den Daten der Datentypen kombinieren, um eine *qualitative Suche* zu ermöglichen.

In durchgeführten Tests wurden sowohl Benutzer als auch Aufgaben zusammen mit ihren zugewiesenen Kompetenzen in einer Elasticsearch-Instanz gespeichert. Eine anschließende Volltextsuche, deren Ziel der Vergleich der zugewiesenen Kompetenznamen war, führte zu einem Matching-Score, der die Menge der jeweils im anderen Datentyp gefundenen Kompetenzen aufgrund ihres *Wortlauts* widerspiegelte.

Diese Methode ist daher optimal, um ein Matching auf Basis von *wortverwandten Kompetenzen* durchzuführen.

Es bleibt jedoch das Problem bestehen, dass ähnlich klingende Kompetenzen keinen Zusammenhang haben müssen, während nicht ähnlich klingende Kompetenzen sehr wohl einen Zusammenhang haben können. Ein Matching rein auf Basis von *Elasticsearch* hat keine Chance, diese Meta-Daten nur auf Basis von Volltextsuche miteinzubeziehen. Dies und einige weitere Gründe führten im Endeffekt zur Verwerfung der Idee von Elasticsearch als Matching-Engine:

- Reine Textsuche lässt inhaltlichen Zusammenhang zwischen Kompetenzen außen vor.
- Werte müssen in jedem Fall auf Basis der gewünschten Modifikationen nachbearbeitet werden.
- Blackbox lässt kein komplettes Nachvollziehen des Scores zu.

Zusammenfassend ist Elasticsearch also geeignet, um ein solches Matching auf Basis der *Kompetenz-Namen* durchzuführen, die fehlende Bewertung eines inhaltlichen Zusammenhangs macht die Engine für den CrAc-Core in dieser Hinsicht jedoch unbrauchbar.

4.1.3 Crowdsourcing als Ansatz

Weitere mögliche Ansätze stammen aus dem so genannten *Micro-Task Market*, auch Crowdsourcing genannt. Der Begriff fasst verschiedene Online-Plattformen zusammen, die den jeweiligen Benutzern – so genannten *Workern* [2] – kleine Aufgaben stellen und bei zufriedenstellender Lösung kleine Geldbeträge bezahlen. Konzepte und mathematische Modelle zum Matching in diesem Bereich [2, 13] nutzen eine Reihe an Informationen zur Berechnung passender Aufgaben, darunter fallen:

- Details der Aufgaben- und Benutzerbeschreibung,
- Performance- und Such-Verlauf des Benutzers,
- Feedback des Benutzers.

Letztendlich wurde ein Crowdsourcing-bezogener Ansatz verworfen, weil dieser zu wenig mit einem Skill-basierten Matching zu tun hat und sich zu sehr

auf monetäre Entlohnung fokussiert, was nicht mit den beschriebenen Anforderungen (s. Abschnitt 1.3.1) konform geht. Einzelne Ideen, wie das intensivere Miteinbeziehen von unterschiedlichen *Meta-Daten*, wurden jedoch in der finalen Umsetzung wieder aufgegriffen.

4.2 Umgesetzte Konzepte

Nun zur finalen Umsetzung im CrAc-Core. Diese entfernt sich gewollt von bereits implementierten Lösungen, um die – in den bereits getesteten Technologien – fehlende *volle Kontrolle* über das Matching, die damit verbundenen Scores und deren Modifikationen zu erhalten. Nunmehr verwendet wird ein vom Core selbst berechneter Matching-Grundwert auf Kompetenz-Basis, der abhängig von der Art der Suche von weiteren mathematischen Modellen modifiziert wird. Die konzeptuelle Grundlage für die Berechnung des Matching-Grundwertes entstammt [8]. Hier wird der Aufbau einer Onthologie-basierten Datenstruktur (jedoch nicht im Sinne einer fertigen Implementierung, sondern auf Basis der allgemeinen Onthologie-Definition, siehe Abschnitt 4.1.1) für die strukturierte Persistierung von Skills und deren Nutzung für *Meta-Datengewinnung* diskutiert. Teile der mathematischen Grundlagen zur Modifikation dieses Grundwertes lehnen sich an Vorschläge aus [10] an. Inwiefern sich die finale Umsetzung von den Ideen dieser Vorlagen unterscheidet, wird in den kommenden Abschnitten analysiert. Bevor zusammenfassend auf die Datenstruktur hinter den Kompetenzen eingegangen wird, muss der Unterschied zwischen zwei verschiedenen, oft auftretenden Begrifflichkeiten aufgezeigt werden.

- Das **Similarity-Value**: Ein Wert zwischen 0 und 1, welcher den *Ähnlichkeitswert* zweier Kompetenzen beschreibt, gespeichert von der Kompetenz-Beziehung (s. Abschnitt 3.2).
- Das **Matching-Value**: Ein Wert zwischen 0 und 1, welcher den *Kompatibilitätsgrad* zwischen einem Benutzer und einer Aufgabe auf Basis der Similarity-Values ihrer jeweiligen Kompetenzen beschreibt.

Die explizite Trennung beider Begriffe ist unbedingt erforderlich, da ihre Bedeutung auf den gesamten Prozess trotz gleichem Wertebereich eine komplett andere ist. Des weiteren gilt zu erwähnen, dass die Kompetenzen für das Matching auf Basis von [8, 10] in Form eines Kompetenz-Graphen vorliegen müssen.

4.2.1 Kompetenz-Graph

Diese auf dem *Skill-Graphen* aus [8] basierende Datenstruktur, in der in einem Graphen beliebige einfache Skills miteinander verknüpft werden, bildet das Rückgrat des Kompetenz-Matching. Die Skills selbst stellen die einzelnen Knoten des Graphen dar, während die Kanten von ihren Beziehungen un-

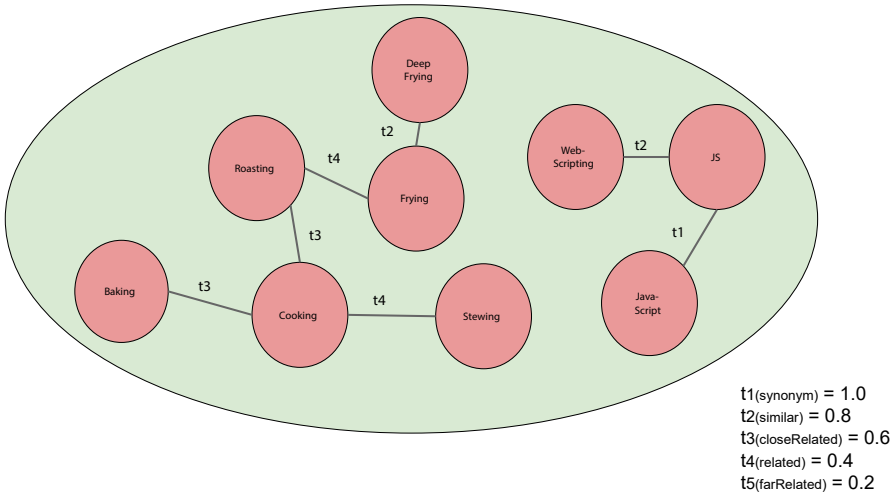


Abbildung 4.2: Beispiel für einen Kompetenz-Graphen.

tereinander repräsentiert werden. Diese verbinden jeweils genau zwei Skills und speichern zudem das angesprochenen *Similarity-Value*, welches die Verwandtschaft der Skills definiert. Diese Vorlage wird im CrAc-Core als *ungerichteter Graph* implementiert und erweitert. Anstatt einfacher *Skills* werden Objekte der *Kompetenz-Klasse* als Knoten des Graphen genutzt, als Kanten dient deren *Kompetenz-Beziehungen* (zum Vergleich siehe Abschnitt 3.1.3).

Das richtige Einpflegen von Inhalten in den Graphen ist von enormer Wichtigkeit, da die Struktur großen Einfluss auf das weitere Matching ausübt. Einmal korrekt eingefügt, kann der Graph jedoch beliebig oft und in beliebig vielen Instanzen vom CrAc-Core (wieder-) verwendet werden. In Abschnitt 6.1 wird näher auf ein einfaches Einpflegen von Kompetenzdaten eingegangen. In Abbildung 4.2 ist ein solcher Kompetenz-Graph abgebildet.

Nach dem abgeschlossenen Aufbau der Datenstruktur kann diese verwendet werden, um den Matching-Prozess zu initiieren, wobei die Struktur des Graphen nun genutzt wird, um von fix definierten auf dynamisch berechenbare *Similarity-Values* nicht direkt verbundener Kompetenzen zu schließen. Die folgende Liste zeigt die einzelnen, im Matchingprozess involvierten Schritte:

1. *Start des Prozesses* (mit dem Kompetenz-Graphen),
2. *Interpolation der Verwandtschaft* verschiedener Kompetenzen aus dem Graphen,
3. *Aufbau einer Matrix* aus den errechneten Werten,
4. *Modifikation der Matrix* auf Basis von Meta-Daten,
5. Auswertung der Matrix und bilden des *Base-Matching-Score*,

6. Ende des Prozesses (mit dem *finalen Matching-Score*).

Die einzelnen Schritte und die darin verwendeten mathematischen Ansätze und algorithmische Umsetzungen sind Thema der nachfolgenden Abschnitte.

4.2.2 Berechnung des Verwandtschaftsgrades

Mathematische Herangehensweise

Um die vom Graphen verwalteten Daten sinnvoll im Matching-Prozess verwenden zu können, müssen zunächst verschiedene Informationen daraus *interpoliert* werden. Am wichtigsten ist hierbei die Berechnung der Verwandtschaft zwischen beliebigen, im Graphen enthaltenen Kompetenzen. Diese muss für jede Kombination aus Benutzer- und Aufgaben-Kompetenzen durchgeführt werden, um die Überlappung beider Objekte mathematisch mit einem *Similarity-Wert* darstellen zu können.

Angenommen wird nun, dass der Pfad von einer beliebigen *Start-Kompetenz* zu einer beliebigen *Ziel-Kompetenz* bereits bekannt ist. Um den Verwandtschaftswert dieser beiden Kompetenzen zu errechnen, müssen die *Similarity-Values* sämtlicher zwischen der *Start-Kompetenz* und *Ziel-Kompetenz* liegenden Kanten multipliziert werden. Bei der Berechnung eines Ähnlichkeitswertes sg zweier *nicht-benachbarter* Kompetenzen c_1 und c_j müssen demnach sämtliche aus dem Graphen ablesbare Ähnlichkeiten

$$s_i = s(c_i, c_{i+1}) \tag{4.1}$$

der jeweils *benachbarten* Kompetenzen c_i und c_{i+1} zwischen der Startkompetenz und der im Abstand j entfernt liegenden Kompetenz multipliziert werden, was sich in der Form

$$sg(c_1, c_j) = \prod_{i=1}^{j-1} s_i \tag{4.2}$$

darstellen lässt (mit $s_i \in [0, 1]$). Aufgrund dessen muss auch jeder neu errechnete Similarity-Value sg in $[0, 1]$ liegen und kann nur *gleich groß* wie oder *kleiner* als die Einzelwerte sein, d. h.,

$$sg(c_1, c_j) \leq s_i, \tag{4.3}$$

für $i = 1, \ldots, j - 1$.

Die Berechnung wird für eine beliebige Schnittmenge aus Usern und Aufgaben durchgeführt, durch Anpassung dieser Schnittmenge ist dadurch bereits vor dem eigentlichen Matching ein Eingreifen in den Prozess möglich. Standardmäßig setzt sich die Schnittmenge aus *allen Usern* und den *bearbeitbaren, veröffentlichten Aufgaben* zusammen.

Algorithmische Umsetzung

Die algorithmische Umsetzung gestaltet sich in diesem Fall etwas aufwändiger als die mathematische Herangehensweise, da beim Beginn der Berechnungen die Lage sämtlicher Knoten zueinander unbekannt ist.

Als Konsequenz ist das erste Problem das Finden des Weges zwischen zwei beliebigen Kompetenzen des Graphen. Es wird daher ein Suchalgorithmus benötigt, der Schritt für Schritt alle Nachbarn (und deren Nachbarn) sämtlicher Kompetenzen prüft und den Weg zu allen anderen Kompetenzen berechnet. Es bieten sich einige Algorithmen dazu an:

- In [8] wird eine Implementierung des *Dijkstra-Algorithmus* für eine Berechnung sämtlicher Kompetenz-Paare des Graphen vorgeschlagen. Die Ergebnisse daraus werden indiziert, was bis zu einer Veränderung des Graphen ein weiteres Traversieren unnötig macht. Fraglich ist jedoch, ob der Einsatz dieses Algorithmus im Kontext des CrAc-Core-Matching die sinnvollste Option ist, vor allem wenn es darum geht, das *Similarity-Value* sämtlicher Kompetenz-Paarungen zusammenhängend zu berechnen und zu indizieren.
- Ein weiterer Kandidat wäre der *Floyd-Warshall-Algorithmus*, der letztendlich denselben Zweck erfüllt, jedoch alle möglichen Paare in einem Durchlauf findet und ohne die Probleme arbeitet, die ein *greedy Suchalgorithmus* wie der Dijkstra-Algorithmus inherent aufweist.
- Auch der *Bellman-Ford-Algorithmus* erfüllt die grundsätzlichen Anforderungen des Anwendungsfalles und ist ebenfalls nicht *greedy*.

All diese Algorithmen sind beschrieben und mit Beispielen dargestellt in [12, Kap. 4]. Im speziellen Fall des CrAc-Cores kann jedoch noch weiter optimiert werden. Dies ist zurückzuführen auf die Relevanz der Knoten-Verbindungen im Kontext des Matching-Prozesses. Ein zu kleines *Similarity-Value* kann, genau wie eine zu große Anzahl an Knotensprüngen zwischen zwei Kompetenzen, zu einer *Irrelevanz* der gefundenen Beziehung führen.

Diese existiert zwar noch auf einer mathematischen Ebene, macht jedoch keinen Sinn mehr auf einer Bedeutungsebene, weil die verbundenen Kompetenzen in der Realität zu weit voneinander entfernt sind, um ein *Matching* rechtfertigen zu können. Diese Irrelevanz kann dahingehend mit einem *Similarity-Value von 0* annotiert werden und ist damit gleichbedeutend mit *keiner Verbindung*. Aufgrund dessen soll der gewünschte Algorithmus keine Standard-Implementierung eines Such-Algorithmus sein und die folgenden Eigenschaften besitzen:

- Er ist nicht zielgerichtet, da sämtliche Kompetenz-Verbindungen in einem Durchlauf persistiert werden sollen.
- Ein Set aus Abbruchbedingungen regelt, wann ein Suchpfad endet, weil weitere Ergebnisse im Kontext der Suche irrelevant sind.
- Sämtliche aufgrund von Pfad-Terminierungen nicht gefundene Knoten-

Verbindungen zählen automatisch als *irrelevant* und damit 0.

Auf dieser Basis gibt es darum folgende Bedingungen, die zur Terminierung eines Suchpfades, nicht aber der gesamten Suche, führen:

- Um die Menge an besuchten Knoten zu reduzieren, darf jeder von der Start-Kompetenz ausgehende Pfad im Graphen nur eine festgelegte Anzahl von Schritten lang sein, bei einer Überschreitung wird der Pfad terminiert.
- Fällt der Verwandtschaftswert unter einen definierten Grenzwert, so gilt die Verwandtschaft als zu wenig ausgeprägt und der Pfad terminiert.
- Wird ein bereits besuchter Knoten erneut gefunden und der neue Verwandtschaftswert ist nicht größer und damit besser als der aktuelle, so ist der Pfad redundant und der Pfad terminiert, ohne den alten Wert zu ersetzen.
- Solange keine dieser Kriterien gebrochen wird, sucht der Algorithmus nach dem Ziel-Knoten und wendet den *Multiplikations-Ansatz* (Gleichung 4.2) an.

Da nun die Herangehensweise und Bedingungen geklärt sind, wird hier der Algorithmus zur Berechnung der Verwandtschaft zwischen einer beliebigen Start-Kompetenz und jeder weiteren Kompetenz näher vorgestellt.

Das Traversieren des Graphen, also das Aufrufen neuer Suchpfade, wird im CrAc-Core aufgrund der einfacheren Lesbarkeit des Codes und der besseren Anwendbarkeit der Abbruchkriterien rekursiv vorgenommen, die Abbruchbedingungen werden an jeden neuen Pfad mitübergeben und erneut überprüft.

Die Abbruchbedingungen der Rekursion lassen sich beliebig in jeder CrAc-Core-Instanz anpassen, `numberOfAllowedSteps` gibt den Wert der erlaubten Knotensprünge eines Pfades an, während `threshold` den nicht zu unterschreitenden *Similarity-Value* beschreibt.

Für dieses Code-Beispiel wurden für beide Attribute eher kleine Werte gewählt, außerdem wurde sämtlicher für das Beispiel nicht-relevante Code (z.B. Konsolen-Output) entfernt, dieser kann vollständig in der beigelegten CD nachgeschlagen werden.

```
1  int numberOfAllowedSteps = 3;
2  double threshold = 0.2;
```

Nun zur Methode, die rekursiv aufgerufen wird, `augmentCompetence()`. Diese prüft zuerst, ob eine der oben besprochenen Bedingungen verletzt wurde. Falls dies nicht der Fall ist, fährt sie fort. Um die Start-Kompetenz des jeweiligen Such-Pfades mit sämtlichen gefundenen Kompetenzen in Verbindung bringen zu können, werden sie gemeinsam mit den berechneten Similarity-Values in die Instanz eines Hilfs-Datentyps gespeichert, der *AugmentedCompetence*. Diese dient als Datencontainer und Schnittstelle.

```
1  //AugmentedCompetence target wird als nächster Knoten im Suchpfad
       untersucht
2  private void augmentCompetence(AugmentedCompetenceCollection collection,
        AugmentedCompetence target) {
3    //Überprüfung der Abbruchbedingungen
4    if (target.getStepsDone() <= numberOfAllowedSteps &&
5    target.getSimilarity() >= threshold) {
6      collection.addCompetence(target);
7      //Überprüfung der Nachbarknoten
8      callChildren(collection, target);
9    }
10 }
```

Jede dieser AugmentedCompetences enthält somit eine Hauptkompetenz (der Knoten, von dem aus die Suche startet) und eine Liste aus gefundenen, damit verbundenen Kompetenzen. Diese Datencontainer werden wiederum in einem weiteren Hilfsdatentyp – `AugmentedCompetenceCollection` – abgelegt, der die Suchergebnisse kumuliert und angelehnt an die Ergebnis-Matrix in [8] nach erfolgreichem Abschluss als Index dient. Genauer analysiert wird dies bei den internen Modulen in Abschnitt 6.2.

In der `AugmentedCompetenceCollection` werden sämtliche Suchergebnisse für alle bisher als Start-Knoten verwendeten Kompetenzen gespeichert und mitgetragen.

Wurde keine der Abbruchbedingungen verletzt, wird die nachfolgend als Code dargestellte `callChildren`-Methode aufgerufen:

```
1  public void callChildren(AugmentedCompetenceCollection collection,
       AugmentedCompetence parent) {
2    //Iteration durch die Liste der Kompetenzen, die mit dem Start-Knoten
       verbunden sind
3    List<CompetenceRelation> rels = parent.getComp().getRelations();
4    if (rels != null) {
5      for (SimpleCompetenceRelation sc : rels) {
6      //Überprüfung, ob der Knoten bereits in der Collection vorhanden ist
7      //ansonsten wird ein neues AugmentedCompetence-Objekt erzeugt
8        AugmentedSimpleCompetence target = collection.loadOrCreate(sc.
       getRelated());
9
10       //Hochzählen/Überprüfen der Abbruchbedinungen für
11       //die verbundenen Kompetenzen
12       target.setPaths(target.getPaths() + 1);
13       if(parent.getSimilarity() * sc.getDistance() >
14       target.getSimilarity() ){
15         //Aktualisierung der Werte des mit dem Suchknoten
16         //verbundenen Knotens
17         updateValues(target, parent, sc.getDistance());
18         //Neuer rekursiver Aufruf mit der
19         //jeweiligen verbundenen Kompetenz
20         augmentIntern(collection, target);
21       }
22     }
23   }
```

```
24 }
```

In der `callChildren()`-Methode werden die mit dem Startknoten verbundenen Kompetenzen untersucht und die Verwandtschaftswerte berechnet. Wichtig anzumerken ist, dass `target` und `parent` aufgrund ihrer logischen Position die Namen wechseln.

Als letzten Schritt aktualisiert die `updateValues()`-Methode die Werte der verbundenen Knoten, bevor die Rekursion die neuen *Start-Kompetenzen* aufruft.

```
1 private void updateValues(AugmentedSimpleCompetence target,
      AugmentedSimpleCompetence parent, double distance) {
2     target.setStepsDone(parent.getStepsDone() + 1);
3     target.setSimilarity(parent.getSimilarity() * distance);
4 }
```

Nach Anwendung dieses Algorithmus erhält man, wie bereits angemerkt, sämtliche Verwandtschaftswerte für eine beliebige Menge an Start-Kompetenzen im `AugmentedCompetenceCollection`-Objekt. Dieses dient als Schnittstelle und garantiert ein einfaches Weiterverarbeiten.

4.2.3 Werte-Matrix

Im Anschluss daran wird, wie in [8] beschrieben, aus den errechneten Similarity-Werten eine *Matrix* gebildet, deren Achsen die Kompetenzen der verglichenen Aufgabe und des Benutzers repräsentieren. Durch diese Struktur, die sowohl die Kompetenzen und ihre Verwandtschaft als auch die damit verbundene Aufgabe und den User enthält, wird garantiert, dass alle Meta-Daten erhalten bleiben. Bis zur Extraktion eines *Basis-Matching-Scores* werden alle weiteren Operationen rein auf dieser Matrix ausgeführt.

In Tabelle 4.1 wird eine solche Matrix, basierend auf dem linken Sub-Graphen, im Kompetenz-Graphen in Abbildung 4.2 veranschaulicht.

	Baking	Cooking	Stewing	Roasting	Frying	Deep Fry.
Baking	1.000	0.600	0.240	0.360	0.144	0.115
Cooking	0.600	1.000	0.400	0.600	0.240	0.192
Stewing	0.240	0.400	1.000	0.240	0.096	0.077
Roasting	0.360	0.600	0.240	1.000	0.400	0.320
Frying	0.144	0.240	0.096	0.400	1.000	0.800
Deep Fry.	0.115	0.192	0.077	0.320	0.800	1.000

Tabelle 4.1: Matrix, basierend auf einem Kompetenz-Graphen.

Implementierung

Im Folgenden wird auf die technische Umsetzung der Struktur eingegangen. Die als eigener Datentyp umgesetzte Matrix wird nachfolgend erneut

angesprochen (s. Abschnitt 3.3.2) und hier genauer erklärt.

Die *Matching-Matrix-Klasse* ist konzipiert für die automatische Anwendung der Similarity-Berechnung und das Befüllen des erstellten Objektes mit den Ergebnissen. Dies erfordert die Übergabe eines beliebigen Users, einer Aufgabe und einer Konfiguration aus Modifikationen, auf die im nächsten Abschnitt eingegangen wird. Um den Erhalt sämtlicher Informationen zu gewährleisten, speichert jedes Feld der – aus einem 2-dimensionalen Array bestehenden – Matrix die für das Feld relevanten Referenzen. Diese müssen daher nicht aus der Matrix abgeleitet, sondern nur vom Matrix-Feld ausgelesen werden, was in der weiteren Verarbeitung der Daten von großer Relevanz ist. Im Abschnitt der sich den Datentypen widmet, wird ebenfalls das Attribut `mandatory` in der Beziehung zwischen Aufgabe und Kompetenz angesprochen. Die Matrix prüft nun, ob der Benutzer die auf diese Weise markierte Kompetenz der Aufgabe auch selbst zugewiesen hat. Ist dies nicht der Fall, so besteht eine Regelverletzung, die vermerkt wird und letztendlich zu einem Matching-Score von 0 führt.

Die Erzeugung der Matrix-Struktur und deren Meta-Daten wird im folgenden Code präsentiert. Sämtliche hier verwendeten Attribute der Matrix-Klasse sind bei den *Datentypen* in Kapitel 3 zu finden. Es wird nun auf den *Konstruktor* der Klasse, welcher automatisch alle Daten verarbeitet und die Matrix erstellt, näher eingegangen.

```
1  public CompetenceCollectionMatrix(CracUser u, Task t,
       FilterConfiguration m) {
2    this.u = u;
3    this.t = t;
4    boolean doable = true;
5    this.taskCompetences = t.getCompetenceRelationships();
6    this.userCompetences = u.getCompetenceRelationships();
7
8    matrix = new MatrixField[userComps.size()][taskComps.size()];
9    buildMatrix();
10   markMandatoryViolation();
11   applyFilters(m);
12 }
```

In der `buildMatrix()`-Methode werden die korrekten Beziehungswerte aus dem *Competence-Cache-Modul* geladen. Dieses stellt die Suchergebnisse global zur Verfügung und wird in Abschnitt 6.2 detaillierter diskutiert. Die geladenen Beziehungswerte werden anschließend mit den Referenzen der Aufgabe und des Benutzers in das entsprechende Feld der Matrix persistiert.

```
1  private void buildMatrix() {
2    int uCount = 0;
3    for (UserCompetenceRel ucr : userComps) {
4      int tCount = 0;
5      for (CompetenceTaskRel ctr : taskComps) {
6        matrix[uCount][tCount] = new MatrixField(ctr, ucr,
7            CompetenceCache.getCompetenceSimilarity(ucr.getCompetence(),
       ctr.getCompetence()));
```

```
8         tCount++;
9       }
10      uCount++;
11    }
12  }
```

Die `markMandatoryViolation()`-Methode durchsucht die gesamte Matrix nach Verletzungen der *Mandatory-Regel* und markiert das Objekt, falls eine solche gefunden wird.

```
1  private void markMandatoryViolation() {
2    double[] columns = bestColumn();
3    for (int i = 0; i < columns.length; i++) {
4      CompetenceTaskRel t = null;
5      int c = 0;
6      for (CompetenceTaskRel ctr : taskComps) {
7        if (c == i) {
8          t = ctr;
9        }
10       c++;
11     }
12     if (t.isMandatory() && columns[i] < 1) {
13       this.doable = false;
14     }
15   }
16 }
```

Auf die `applyFilters()`-Methode wird im nächsten Abschnitt eingegangen.

4.2.4 Modifikation auf Basis von Meta-Daten

An den einzelnen Feldern der Matrix, welche jeweils einen Ähnlichkeitswert der beliebigen Kompetenzen i und j beschreiben, können nun Modifikationen aller Art vorgenommen werden. Abhängig von den Meta-Daten und den verfolgten Zielen werden unterschiedliche Modelle für die Re-Kalkulation der Werte genutzt. Bei den verschiedenen Modifikationen handelt es sich entweder um Berechnungen aus [10], welche für den CrAc-Core angepasst wurden, oder um Neuentwicklungen, die an [10] angelehnt sind. Für die Formel-Angaben wird hierbei immer von Werten im Bereich $[-1, 1]$ ausgegangen, bei den Angaben zur algorithmischen Umsetzung wird genauer auf die nötige Umrechnung mancher *Beziehungsdaten* aus dem Wertebereich $[-100, 100]$ eingegangen.

Grundsätzliche Implementierung

Sämtliche Modifikationen werden als Erweiterung des Datentyps *Filter* (s. Abschnitt 3.3.2) angelegt und überschreiben die relevanten Methode (dargestellt im folgenden Code), um ein für die Matching-Matrix *uniformes Interface* zu gewährleisten.

```
1  public abstract double apply(MatrixField m);
```

Basierend auf dem Konzept der Filter wird die Matrix Feld für Feld an die `apply()`-Methode der implementierten Filter in der konfigurierten Reihenfolge übergeben. Nachfolgend werden diese zusammen mit den mathematischen Konzepten erklärt.

Modifikation auf Basis von Vorlieben

Jeder Wert der Matrix wird basierend auf dem Like-Level (l) des Users gegenüber einer Kompetenz angepasst. Somit wird garantiert, dass der Benutzer bei gleicher Eignung für verschiedene Aufgaben diejenigen bevorzugt erhält, für deren zugewiesene Kompetenzen er sich interessiert.

Mathematische Herangehensweise

In ihrer Herangehensweise orientiert sich diese Modifikation am Vorschlag zur Anpassung der *Similarity-Values* aus [10] und setzt diesen in leicht veränderter Form um. Der angepasste Ähnlichkeitswert $sl(i,j)$ wird auf Basis des alten Wertes $sg(i,j) \in [0,1]$, unter der Berücksichtigung des *Like-Levels* $l \in [-1,1]$, mit der Gleichung

$$sl(i,j) = sg(i,j) \cdot (1 + (1 - sg(i,j)) \cdot l) \tag{4.4}$$

berechnet. Wie in Abbildung 4.3 zu sehen ist, werden die Similarity-Values bei steigendem oder sinkendem l des Users gegenüber der jeweiligen Kompetenz nach oben bzw. unten korrigiert. Die verwendeten Werte für l gehen hierbei von stark positiv (1 bzw. 0.5 und *rot* eingezeichnet) über neutral (0 bzw. *schwarz* eingezeichnet) zu stark negativ (-0.5 bzw. -1, *blau* eingezeichnet). Diese Korrektur wird, wie aus der Abbildung gut ersichtlich, schwächer, je näher das Similarity-Value den Extremwerten 0 und 1 kommt.

Algorithmische Umsetzung

Die Umsetzung entspricht beinahe einer exakten Kopie der Funktion. Der einzige Unterschied besteht darin, dass aufgrund der Speicherung des `likeLvl` im System als Integer zwischen -100 und 100 eine Umrechnung erfolgen muss und Werte per default in den Wertebereich 0 bis 1 getrimmt werden, sollte ein solcher Wert aus irgendeinem Grund Teil der Berechnung sein. Folgender Code zeigt die Implementierung der Modifikation:

```
1  @Override
2  public double apply(MatrixField m) {
3    double simVal = m.getVal();
4    int likeLvl = m.getUserRelation().getLikeValue();
5    double simValLike = simVal * (1 + (1 - simVal) * likeLvl);
6    if (simValLike > 1) {simValLike = 1;}
7    else if (simValLike < 0) { simValLike = 0;}
8    return simValLike;
9  }
```

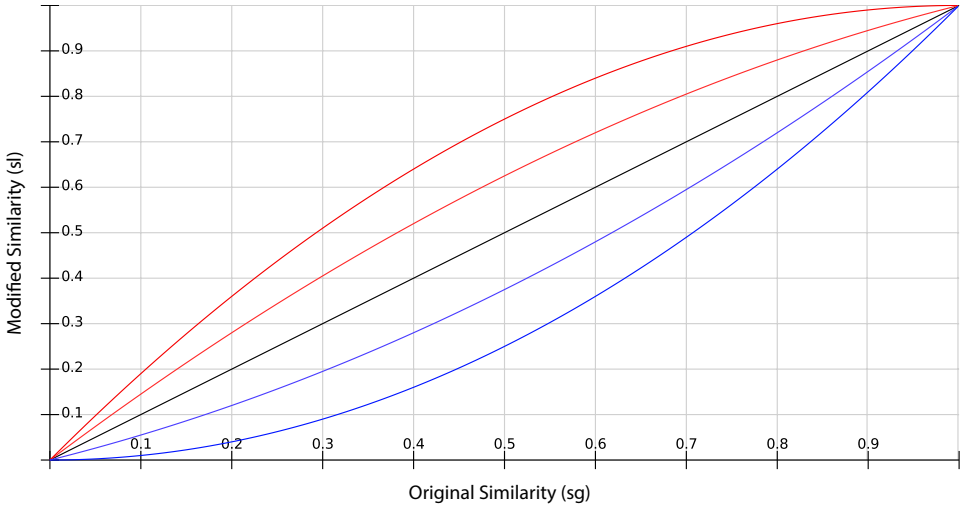


Abbildung 4.3: Die Like-Modifikation (Gleichung 4.4) als Funktion.

Modifikation auf Basis von Nutzer-Beziehungen

Sämtliche Werte werden basierend auf den Beziehungen zwischen dem in der Matrix repräsentierten Nutzer und den an der fraglichen Aufgabe bereits teilnehmenden Nutzern (u) angepasst. Dies hat zur Folge, dass Aufgaben, an denen bereits Freunde oder andere Freiwillige mit gutem Verhältnis eingeschrieben sind, vermehrt vorgeschlagen werden. Für etwaige Berechnungen wird hierbei der *durchschnittliche Like-Level* gegenüber allen beteiligten weiteren Freiwilligen genutzt.

Mathematische Herangehensweise

Hier wird dasselbe mathematische Modell wie in der *Modifikation auf Basis von Vorlieben* genutzt, dargestellt in Abbildung 4.3. Anstatt das *Competence-Like-Level* für die Anpassung der Werte zu nutzen, wird hierbei allerdings auf das durchschnittliche Like-Level des suchenden Users in Verbindung mit den bereits teilnehmenden Usern zurück gegriffen.

Ähnlich der vorherigen Modifikation wird auch hier der angepasste Ähnlichkeitswert $su(i,j)$ auf Basis des alten Wertes $sg(i,j) \in [0,1]$, diesmal unter Berücksichtigung des durchschnittlichen *User-Like-Levels* $u \in [-1,1]$, mit der Gleichung

$$su(i,j) = sg(i,j) \cdot (1 + (1 - sg(i,j)) \cdot u) \tag{4.5}$$

berechnet. Auch hier gilt die Funktion aus Abbildung 4.3.

Algorithmische Umsetzung

Zur Vermeidung von Rekalkulationen wird der für die Berechnung nötige Durchschnitt der *User-Like-Level* im Filter gecached. Um den Matching-Prozess so dynamisch wie möglich zu halten, wird dieser *Kompetenz-unabhängige* Wert trotzdem auf jedes *Similarity-Value* einzeln angewandt und nicht erst auf den finalen Matching-Score. Auf diese Weise kann bei einer nötigen Anpassung auf einer tieferen Ebene modifiziert werden. Folgender Code zeigt die Implementierung der Modifikation:

```
1  @Override
2  public double apply(MatrixField m) {
3    double simVal = m.getVal();
4    CracUser user = m.getUserRelation().getUser();
5    Task task = m.getTaskRelation().getTask();
6    calcRelatedVal(user, task);
7
8    double simValUserLike = simVal * (1 + (1 - simVal) * userLikeLvl);
9    if (simValUserLike > 1) { simValUserLike = 1; }
10   else if (simValUserLike < 0) { simValUserLike = 0; }
11   return simValUserLike;
12
13 }
```

Modifikation auf Basis von Fähigkeiten

Jeder Wert der Matrix, bestehend aus dem Similarity-Value zweier Kompetenzen, wird basierend auf dem *Proficiency-Level* des Benutzers (p_p) in Verbindung mit dem geforderten Proficiency-Level der Task (p_t) angepasst. Auf diese Weise werden zwei Dinge sicher gestellt: Zum einen werden Aufgaben höher gewertet, die der Benutzer besser lösen kann, und zum anderen können Aufgaben auf eine Weise konfiguriert werden, die es erlaubt, Benutzer zu bevorzugen, die ein Mindestlevel an *Eignung* mitbringen.

Mathematische Herangehensweise

In diesem Fall wird die Herangehensweise aus [10] komplett übernommen. Abhängig von der Differenz aus *Können* des Users und *Anforderung* der Task wird das Similarity-Value angepasst. Abbildung 4.4 zeigt die verwendete Funktion.

Bei dieser Modifikation errechnet sich der angepasste Ähnlichkeitswert $sp(i,j)$ ebenfalls auf Basis des alten Wertes $sg(i,j) \in [0,1]$ und bezieht die *Proficiency-Level* der Person $p_p \in [0,1]$ und der Aufgabe $p_t \in [0,1]$ mit ein. Hierfür wird die Gleichung

$$sp(i,j) = sg(i,j) \cdot (1 - (p_t - p_p)) \tag{4.6}$$

verwendet. Wie in Abbildung 4.4 zu erkennen ist, wird das *Similarity-Value* bei niedriger *Proficiency* seitens des Benutzern und hoher geforderter *Pro-*

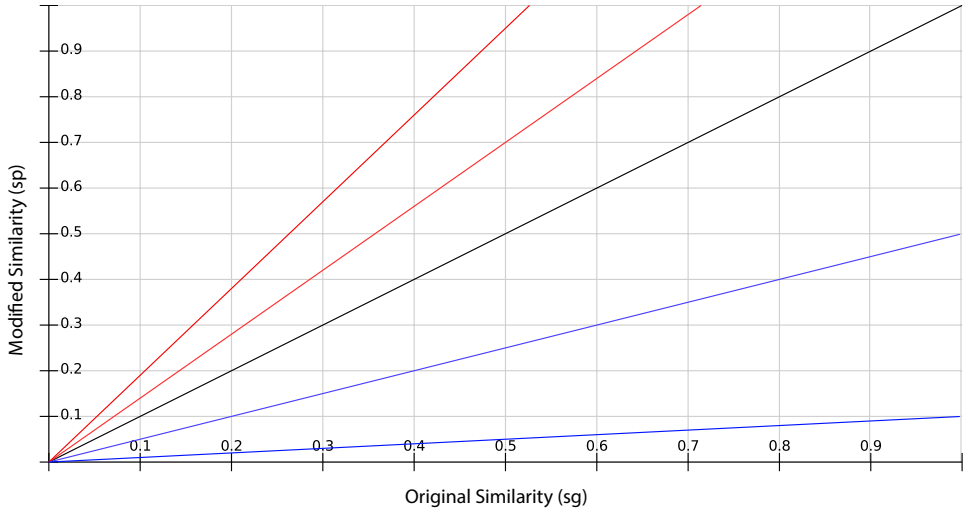


Abbildung 4.4: Die Proficiency-Modifikation (Gleichung 4.6) als Funktion.

ficiency seitens der Aufgabe stark abgeschwächt. Bei sich annähernden Werten wird dieser Effekt schwächer oder dreht sich um. Die verwendeten Proficiency-Values gehen hierbei von stark positiv (User-Proficiency 1 und Task-Proficiency 0.1 und *rot*) über neutral (User-Proficiency 0.5 und Task-Proficiency 0.5 und *schwarz*) bis hin zu stark negativ (User-Proficiency 0.1 und Task-Proficiency 1 und *blau*).

Algorithmische Umsetzung

Auch hier ist anzumerken, dass aufgrund des Wertebereiches, in der die *Proficiency-Levels* abgespeichert sind, eine Division notwendig wird. Ebenfalls werden Werte, die 0 unterschreiten oder 1 überschreiten, dem Standard-Wertebereich angepasst. Folgender Code zeigt die Implementierung der Modifikation:

```
1  @Override
2  public double apply(MatrixField m) {
3    double simVal = m.getVal();
4    int profLvlCTask = m.getTaskRelation().getNeededProficiencyLevel();
5    int profLvlCPerson = m.getUserRelation().getProficiencyValue();
6
7    double simValProf = simVal;
8    if (profLvlCPerson < profLvlCTask) {
9      simValProf = simVal * ( 1 - ((profLvlCTask / 100.0) - (
       profLvlCPerson / 100.0)));
10   }
11   if (simValProf > 1) { simValProf = 1; }
12   else if (simValProf < 0) { simValProf = 0; }
13   return simValProf;
```

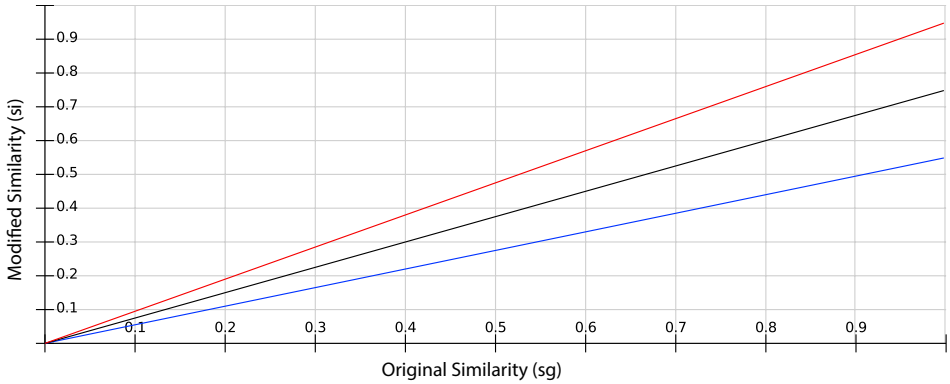


Abbildung 4.5: Die Importance-Modifikation (Gleichung 4.7) als Funktion.

```
14 }
```

Modifikation auf Basis der Wichtigkeit

Hierbei werden die Felder der Matrix auf Basis der Wichtigkeit (il) von den Kompetenzen für die Task unterschiedlich gewichtet. Dies garantiert eine möglichst genaue und flexible Einstellung von einzelnen Kompetenzen in Verbindung mit beliebigen Aufgaben und macht es für den CrAc-Core einfacher, geeignete Aufgaben für den suchenden User einzugrenzen. Der Ansatz stammt hierbei aus [8].

Mathematische Herangehensweise

Mithilfe dieser Modifikation werden die *Similarity-Values* unwichtiger Kompetenzen abgeschwächt, während wichtige Kompetenzen unangetastet bleiben. Bei dieser Modifikation errechnet sich der angepasste Ähnlichkeitswert $si(i, j)$ auf Basis des alten Wertes $sg(i, j) \in [0, 1]$ und bezieht das *Importance-Level* der Kompetenz hinsichtlich der Aufgabe $il \in [0, 1]$ mit ein. Hierfür wird die Gleichung

$$si(i, j) = sg(i, j) \cdot (\frac{il + 1}{2}) \tag{4.7}$$

verwendet. Abbildung 4.5 veranschaulicht den Abfall des Grundwertes bei hohem (0.9 und *rot* eingezeichnet), mittlerem (0.5 und *schwarz* eingezeichnet) und niedrigem (0.1 und *blau* eingezeichnet) *Importance-Level*.

Algorithmische Umsetzung

In diesem Fall gleicht die Implementierung der oben verwendeten Formel. Folgender Code zeigt die Implementierung der Modifikation:

```
@Override
public double apply(MatrixField m) {
  double simVal = m.getVal();
  int impLvl = m.getTaskRelation().getImportanceLevel();

  double simValImp = simVal * (( impLvl + 1) / 2);

  return simValImp;
}
```

Kapitel 5

Feedback und Modifikation

Mit dem erfolgreichen Beenden einer Aufgabe – unabhängig vom Outcome – endet der Gesamtprozess (s. Abbildung 4.1) noch nicht. Auf Basis der Erfahrung und des Ergebnisses aus der Teilnahme an einer Aufgabe soll weitere Information gewonnen werden, die zurück in den *CrAc-Core* fließt und dort Meta-Daten und im Endeffekt auch den Matching-Prozess für die betroffenen User anpasst. Zum Erreichen dieses Ziels werden mithilfe des Benutzer-Feedbacks dieselben Meta-Daten angepasst, die von den Matching-Filtern genutzt werden. Dieses Kapitel geht daher näher auf diese letzten beiden Schritte des Task-Feedback-Kreislaufs ein.

5.1 Evaluierung

Der erste Schritt für den Abschluss des Loops ist das Evaluieren der Erfahrung durch den Benutzer. Hierfür wird die Klasse *Evaluation* verwendet, die das Feedback des Users speichert und eine Schnittstelle für die Aufbereitung bietet. Sobald die jeweilige Aufgabe im System den Status *Completed* hat, kann das ihr zugeordnete *Evaluation-Objekt* von außerhalb des CrAc-Cores über einen separaten Endpoint angesprochen werden. Um den Aufwand für den Endnutzer gering zu halten, wird die Menge der Daten, die eingetragen werden können, auf folgendes Minimum reduziert.

```
1  private double likeValOthers;
2  private double likeValTask;
```

Diese Attribute beschreiben wie sehr dem User erstens die Aufgabe an sich und zweitens die Zusammenarbeit mit den anderen Freiwilligen gefallen hat. Die Befüllung von nur zwei Attributen (mit einem Wert zwischen -1 und 1) ist minimaler Aufwand und es lassen sich einige interessante Daten ableiten.

- **Training**: Eine Evaluierung bestätigt die definitive Teilnahme an einer Aufgabe, es kann daher von einer leichten Verbesserung des evaluierenden Benutzers betreffend die Kompetenzen der Aufgabe ausgegangen

werden.

- **Gefallen an einer Aufgabe**: Das Attribut *likeValTask* beschreibt, wie sehr die Aufgabe dem Benutzer gefallen hat. Abhängig von dieser Bewertung lässt sich Wert von jeder mit dem Task verknüpften Kompetenz anpassen.
- **Beziehung zu anderen Usern**: Dasselbe gilt für das Attribut *likeValOthers*, welches den Grad an Freundschaft zwischen allen Beteiligten anpasst.

Diese Werte auf jeweils alle Mitarbeiter und Kompetenzen zu beziehen bietet eine einfache Möglichkeit der Anpassung, die akkurater wird, je mehr Feedback dem System zur Verfügung steht, sprich je öfter der Kreislauf durchlaufen wird.

5.2 Daten-Modifikation

Nach abgeschlossener Evaluierung folgt die darauf basierende Modifikation der Beziehungsdaten im System. Ähnlich wie im *Matching-Prozess* werden mathematische Modelle genutzt, um die Daten aus dem jeweiligen *Evaluierungs-Objekt* mit den Meta-Daten zu kombinieren. Die Modifikationen betreffen zwei Arten von Beziehungen des Users, *User-Relationship* und *User-Competence-Relationship*. Die *User-Task-Relationships* anzupassen wäre sinnlos, da diese Beziehungen erst nach dem Matching-Prozess existieren und die beiden modifizierten Klassen diese in jedem Fall indirekt beeinflussen.

5.2.1 Modifikation der Kompetenz-Beziehungen

Wie bereits angemerkt können hier zwei verschiedene Werte dem Feedback entsprechend geändert werden. Zum einen der *Proficiency-Level*, der durch das nachgewiesene Training gesteigert werden kann und zum anderen das *Like-Level*, welches anhand der Evaluierung angepasst wird. Versucht das System eine Beziehung zu ändern, welche noch nicht existiert, wird diese mit niedrigen Startwerten erzeugt, um auf diese Weise den Lerneffekt zu simulieren.

Mathematische Grundlagen

Um das `proficiencyValue` statisch zu erhöhen, wird auf die einfache Gleichung

$$pv' = pv + 0.1 \tag{5.1}$$

zurückgegriffen. Die Variable $pv \in [0, 1]$ beschreibt hierbei den alten Wert und pv' den neuen. Wie in Abbildung 5.1 zu sehen ist, werden sämtliche Werte ausschließlich statisch um 0.1 erhöht.

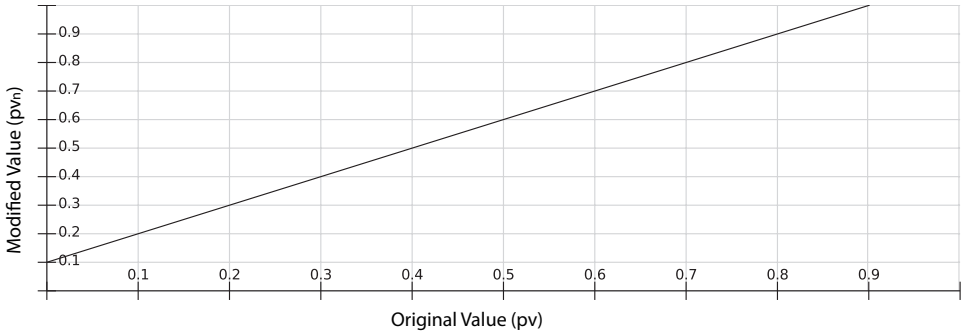


Abbildung 5.1: Die Proficiency-Modifikation auf Basis der Evaluierung (Gleichung 5.1) als Funktion.

Im Gegensatz dazu wird bei der Anpassung des *Like-Levels* ein Prozentsatz der Bewertung *(le)* der Aufgabe zu allen betroffenen Kompetenzen addiert. Auf diese Weise steigt der Wert bei häufiger positiver Bewertung schnell an, einzelne (eher zufällig betroffene) Kompetenzen werden allerdings nicht stark genug betroffen, um einen großen Unterschied im *Matching-Prozess* auszumachen. Das neue *Like-Value* lv' wird daher aus dem alten Wert $lv \in [-1, 1]$, in Abhängigkeit zur Bewertung der Evaluierung $le \in [-1, 1]$, unter Verwendung der Gleichung

$$lv' = lv + \frac{le}{4} \tag{5.2}$$

berechnet. Inwiefern die Werte innerhalb dieser Funktion angepasst werden, ist in Abbildung 5.2 zu sehen. Diese zeigt die Auswirkung der Modifikation bei einem positiven Wert (1 und *rot* eingezeichnet), einem neutralen Wert (0 und *schwarz* eingezeichnet) und einem negativen Wert (-1 und *blau* eingezeichnet).

Algorithmische Implementierung

Ähnlich der Implementierung der Matrix-Modifikationen wird auch hier zunächst der Wertebereich angepasst (Integer zwischen 0 bzw. -100 und 100). Werte, die diesen Bereich unter- oder überschreiten, werden abgefangen. Außerdem ist die Implementierung Teil einer *Worker-Einheit* (s. Abschnitt 3.3). Folgender Code zeigt die Implementierung der Modifikation:

```
1  //Iterieren der Kompetenz-Beziehungen und zuweisen der User-Kompetenz-
       Daten
2  for(CompetenceTaskRel ctr : task.getMappedCompetences()){
3    UserCompetenceRel ucr = userCompetenceRelDAO.findByUserAndCompetence(
       user, ctr.getCompetence());
4
5    //Gibt es zwischen dem angemeldeten Benutzer und der Kompetenz keine
       Beziehung, wird diese erstellt
```

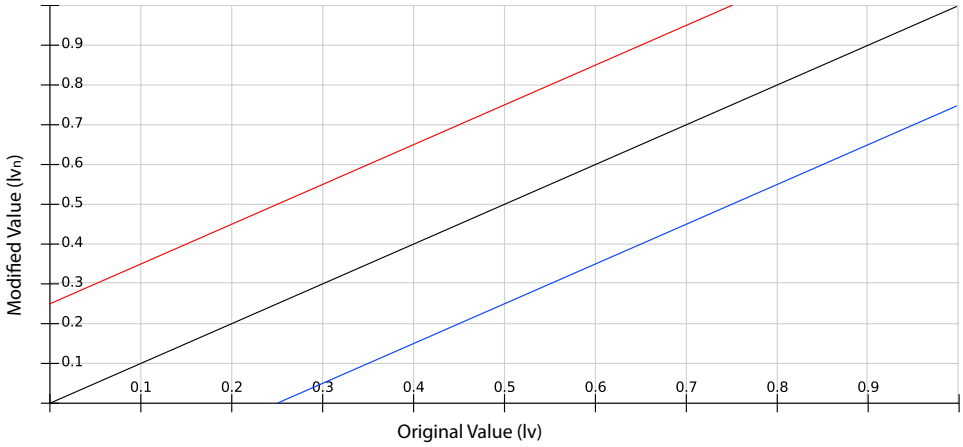


Abbildung 5.2: Die Like-Modifikation auf Basis der Evaluierung (Gleichung 5.2 als Funktion.

```
6    if (ucr == null) {
7      ucr = new UserCompetenceRel();
8      ucr.setCompetence(ctr.getCompetence());
9      ucr.setUser(user);
10     ucr.setLikeValue(0);
11     ucr.setProficiencyValue(0);
12   }
13
14   int likeValue = ucr.getLikeValue();
15   int profValue = ucr.getProficiencyValue();
16   //Anwenden der Like-Funktion
17   likeValue += (evaluation.getLikeValTask() / 4) * 100 ;
18   if(likeValue > 100){
19     likeValue = 100;
20     }else if(likeValue < -100){
21     likeValue = -100;
22   }
23   //Anwenden der Proficiency-Funktion
24   profValue += 10;
25   if(profValue > 100){
26     profValue = 100;
27   }else if(profValue < 0){
28     profValue = 0;
29   }
30   //Speichern der Daten
31   ucr.setLikeValue(likeValue);
32   ucr.setProficiencyValue(profValue);
33   userCompetenceRelDAO.save(ucr);
34 }
```

5.2.2 Modifikation der Benutzer-Beziehungen

Diese nächste Modifikation betrifft die Beziehung zwischen dem evaluierenden und sämtlichen anderen teilnehmenden Usern einer Aufgabe. Auf Basis des Wertes aus dem *Evaluation-Objekt* werden diese entweder auf- oder abgewertet. Auch hier werden neue Beziehungen erzeugt, sollten die zu modifizierenden User-Relationships noch nicht existieren.

Mathematische Grundlagen

Die mathematische Grundlage hierfür ist dieselbe wie für den *Like-Level* (le) gegenüber den Kompetenzen der Aufgabe. Demnach gilt

$$lv' = lv + \frac{le}{4} \tag{5.3}$$

für die Berechnung, wobei sich der neue *Like-Value* lv' wiederum aus dem alten Wert $lv \in [-1, 1]$ in Abhängigkeit zur Bewertung der Evaluierung $le \in [-1, 1]$ ergibt, diesmal werden jedoch jeweils die *Like-Levels* gegenüber anderen Benutzern anstatt gegenüber verschiedenen Kompetenzen herangezogen. Aufgrund dessen gilt Abbildung 5.2 auch für diese Funktion.

Algorithmische Implementierung

Auch der algorithmische Teil geht sehr ähnlich vor, es werden lediglich andere Beziehungen und Meta-Daten für die Kalkulation verwendet. Folgender Code zeigt die Implementierung der Modifikation:

```
1  //Iterieren der User-Beziehungen
2  for (UserTaskRel utr : evaluation.getTask().getUserRelationships()) {
3  //Überprüfung, ob der jeweilige Nutzer der angemeldete ist
4    if (user.getId() != utr.getUser().getId()) {
5    //Wenn nein, Zuweisung der Daten
6      UserRelationship ur = userRelationshipDAO.findByC1AndC2(utr.getUser
       (), user);
7      //Gibt es zwischen dem angemeldeten und diesem Benutzer keine
       Beziehung, wird diese erstellt
8      if (ur == null) {
9        ur = userRelationshipDAO.findByC1AndC2(user, utr.getUser());
10       if (ur == null) {
11         ur = new UserRelationship();
12         ur.setC1(user);
13         ur.setC2(utr.getUser());
14         ur.setLikeValue(0);
15         ur.setFriends(false);
16       }
17     }
18
19     double like = ur.getLikeValue();
20     double updated = like;
21
```

```
22      //Anwenden der Like-Funktion
23      updated += evaluation.getLikeValOthers() / 4;
24
25      if (updated > 1) {
26        updated = 1;
27      } else if (updated < -1) {
28        updated = -1;
29      }
30
31        //Speichern der Daten
32      ur.setLikeValue(updated);
33      userRelationshipDAO.save(ur);
34    }
35  }
```

Kapitel 6

Architektur

Nachdem die Voraussetzungen, die relevanten Datentypen und der Matching- und Evaluierungs-Prozess nun geklärt sind, beschäftigt sich dieses Kapitel mit der Umsetzung des Gesamtsystems. Im Folgenden werden die Software-Architektur und ihr Design, sowie sämtliche Module näher analysiert. Es handelt sich bei der Architektur um die Umsetzung eines objektorientierten, *modulbasierten* Ansatzes, der die bisher präsentierten Klassen zu einem funktionierenden Ganzen verknüpft und dabei externe Komponenten als Datenspeicher und zur Datenausgabe nutzt. Für mehr Übersichtlichkeit wird die gezeigte Architektur nachfolgend in zwei Kategorien aufgeteilt:

- **Externe Komponenten**: Diese sind nicht direkt Teil des *CrAc-Core*, sondern nur damit verbunden.
- **Interne Module**: Diese sind als Teil von *Java Spring* implementiert und gehören direkt zur CrAc-Core-Instanz.

Um einen ersten Überblick zu geben, werden im Folgenden in Abbildung 6.1 nicht nur die Module des Frameworks (interne Module) gezeigt, sondern auch jene Software-Instanzen, mit denen sie in Verbindung stehen und kommunizieren (externe Komponenten).

6.1 Externe Komponenten

In diese Kategorie fallen verschiedene Software-Instanzen, die essentiell für den CrAc-Core hinsichtlich Input, Verarbeitung von Daten und Output sind, jedoch eigenständig laufen und somit nur mit dem CrAc-Core verknüpft, nicht aber ein Teil davon sind. Wie diese Verknüpfung genau aussieht und welche Technologie verwendet wird, ist komplett abhängig von der jeweiligen Komponente und wird im Folgenden genauer erläutert.

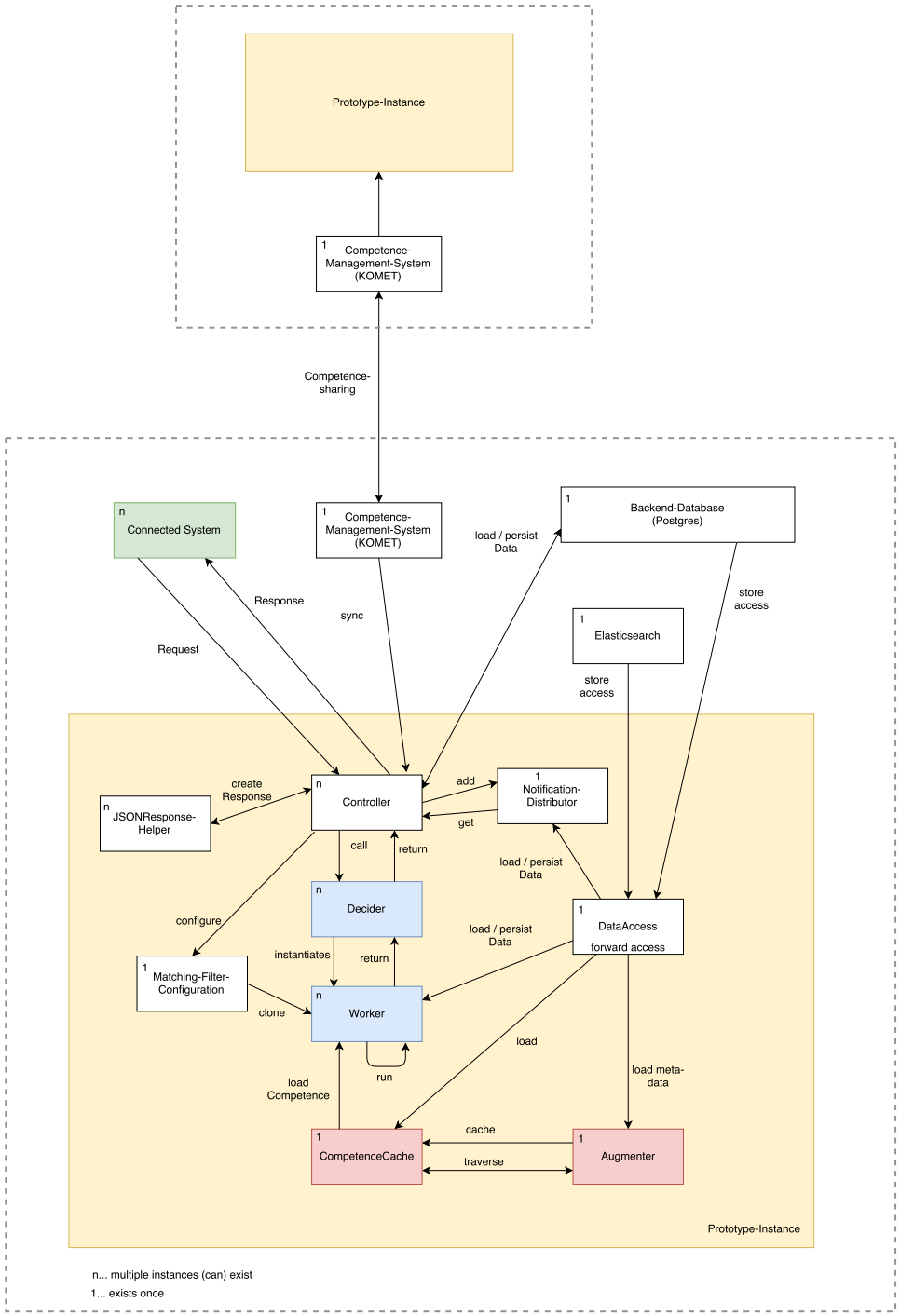


Abbildung 6.1: System-Diagramm des Frameworks.

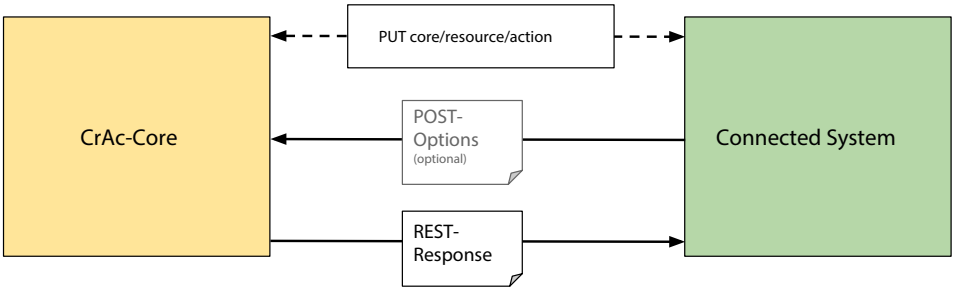


Abbildung 6.2: Zugriff eines angeschlossenen Systems via *PUT-Methode*.

Per API verknüpfte Systeme

Wie in Abschnitt 2.1.2 beschrieben, kann jede beliebige Software mit dem CrAc-Core verknüpft werden. Dies wird mithilfe einer definierten API, also einer Sammlung von Endpoints, realisiert. Jeder Aufruf (Request) eines solchen Endpoints setzt verschiedene, Endpoint-abhängige Prozesse innerhalb des Cores in Gang. Die Zuordnung von Requests und Methodenaufrufen übernimmt hierbei das *Spring-Framework* in sogenannten *Controller-Objekten*, (siehe den folgenden Abschnitt 6.2). Im klassischen Fall ist ein solches – per API verknüpftes – System eine Art von Frontend, das den CrAc-Core mit den benötigten Grunddaten (siehe Kapitel 3) befüllt und die Ergebnisse nach abgeschlossener Verarbeitung seitens des Cores formatiert ausgibt[1]. Dies ist allerdings keine Notwendigkeit, jede Art von Software – unabhängig von der verwendeten Technologie – kann die zur Verfügung gestellte API nutzen, solange sie deren Regeln befolgt. Die Schnittstelle besitzt folgende Eigenschaften:

- Endpoints, basierend auf einer definierten URL-Struktur erlauben den Zugriff auf Funktionalität,
- Anwendung des *REST-Prinzips* [4],
- Für den Datenaustausch wird das Format *JSON* verwendet,
- Das Ausgabe-Objekt ist eine *Rest-Response* (s. Abschnitt 3.3.3),
- Der Input verwendet beliebige Attribute der *Post-Options-Klasse*, nur erforderlich bei Request-Methoden mit Datenübergabe (s. Abschnitt 3.3.3).

Abbildung 6.2 zeigt den Zugriff eines beliebigen Systems auf den CrAc-Core via Endpoint.

[1] Für das CrAc-Projekt ist beispielsweise die Implementierung eines solchen Frontends als weiteres Teilprojekt in Arbeit.

Kompetenz-Management-System (KOMET)

Eine weitere externe Komponente setzt sich mit dem Datenaustausch zwischen verschiedenen Instanzen des CrAc-Cores auseinander. Das verwendete System, eine von einem Projektpartner modifizierte Version des Kompetenz-Management-Systems *KOMET*[2], erfüllt hierbei mehrere Rollen:

- Eine *administrative Oberfläche* für die Eingabe von Kompetenzdaten,
- Ein *Postkasten-System* zum Austausch von Kompetenzdaten zwischen verschiedenen CrAc-Core-Instanzen.

KOMET bietet ein maßgeschneidertes administratives Interface für das Anlegen und Warten System-relevanter Kompetenzdaten, darunter fallen die Kompetenzdaten an sich und ihre Abgrenzungen, die Kompetenz-Raster. Das Tool befindet sich in einem fertigen Zustand, wurde ausreichend getestet, ist beispielsweise auch als *Moodle-Extension* verfügbar und passt auf den Anwendungsfall CrAc-Core. Es ist daher zur Wartung der Kompetenzen sehr gut geeignet. Ein weiterer wichtiger Faktor ist Möglichkeit des Austauschs von Kompetenzen zwischen beliebigen KOMET-Instanzen. Damit ermöglicht das Kompetenz-Management-Tool ein *Postkasten-System* auf Basis von Kompetenz-Export und Import im XML-Format [16].

Für die effiziente Nutzung ist jedem CrAc-Core genau eine KOMET-Instanz zugewiesen, mit deren Datenbank eine permanente Verbindung aufgebaut wird. Diese Verbindung wird benötigt, um die Kompetenzen und Raster zu jedem Zeitpunkt mittels Endpoint-Aufruf synchronisieren zu können. Ausgelöst durch den dafür bereit gestellte API-Call, ruft *Spring* den SynchronizationController (s. Abschnitt 6.2) auf und startet einen Prozess, der in mehreren Schritten die Daten aus der KOMET-Datenbank lädt und die Kompetenzen der CrAc-Datenbank auf Aktivität und mögliche Inkonsistenzen prüft, bevor er beginnt sie mithilfe der neuesten Version der Kompetenzen zu aktualisieren.

In Abbildung 6.3 werden sowohl die Synchronisation zwischen CrAc und KOMET, als auch die die Datenübertragung zwischen mehreren KOMET-Versionen dargestellt.

Datenbank

Obwohl die Verknüpfung zwischen der Java-Applikation und der zugehörigen Datenbank gerade aufgrund der Nutzung von *Hibernate* und *JPA* sehr eng ist, stellt die Datenbank trotzdem eine externe Komponente dar, weshalb sie in dieser Sektion erneut erwähnt wird. Genutzt wird eine Instanz der relationalen Datenbank *Postgres*. Warum diese der beste Kandidat für den Anwendungsfall ist, wird in Abschnitt 2.1.1 erörtert. Wie die Datenbank mit dem *CrAc-Core* interagiert und inwiefern die Daten für diesen Prozess

[2] https://gtn-solutions.com/home/komet/

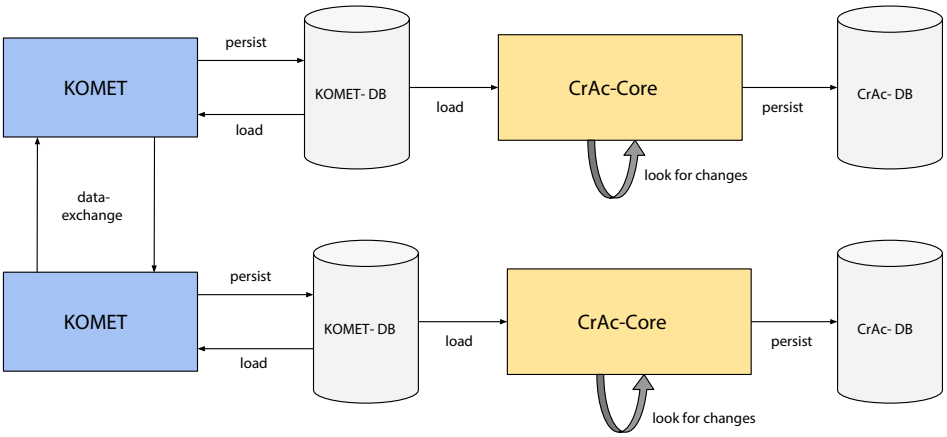


Abbildung 6.3: Daten-Synchronisierung zwischen KOMET und CrAc-Core.

transformiert werden müssen, ist in Abschnitt 2.1.2 näher beschrieben.

Volltext-Suchengine

Zwar ist die Schlüsselwort-Suche (auch nach mehreren Wörtern) in einer *Postgres-Datenbank* möglich, diese stößt jedoch besonders Performance-technisch sehr schnell an ihre Grenzen. Um eine performante und zuverlässige Volltext-Suche zu ermöglichen, wird folglich eine eigene Software benötigt. Hier fällt die Wahl eindeutig auf die Engine *Elasticsearch*, eine Software, die ebenfalls (siehe Abschnitt 4.1.2) für den *Matching-Prozess* getestet wurde.

Folgende Merkmale heben *Elasticsearch* besonders hervor:

- Open-Source,
- Skalierbar aufgrund des *Shard-Systems*,
- Sehr performant aufgrund der Lastaufteilung innerhalb des *Shard-Systems*,
- Verfügt über eine eigene *Java-API* als simple Applikations-Schnittstelle.

Als Bestätigung rangiert Elasticsearch im Ranking der besten Suchengines auf [15] seit geraumer Zeit mit Abstand auf dem ersten Platz. In Abbildung 6.4 wird gezeigt, wie der CrAc-Core mittels *Elastic-Connector*, erörtert in Abschnitt 3.3.1, auf die Elasticsearch-Instanz zugreift und Aufgabendaten persistiert bzw. Volltextsuchen durchführt. Der Zugriff auf Elasticsearch wird in Abbildung 6.4 dargestellt.

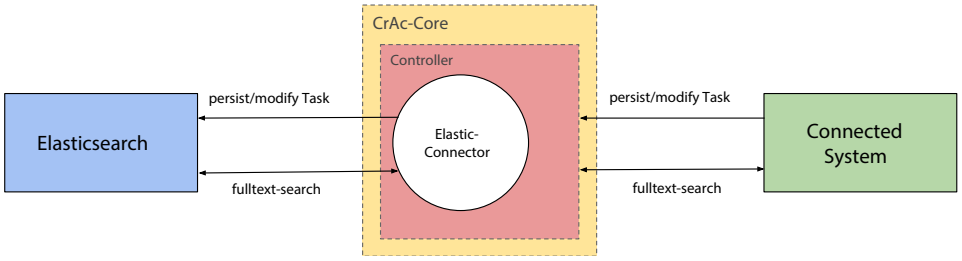


Abbildung 6.4: Zugriffsweg des Cores auf Elasticsearch.

6.2 Interne Module

Folgende Module sind Teil des eigentlichen *CrAc-Core* und damit in der in Kapitel 2 angeführten Technologie *Java Spring* unter Verwendung etablierter *Entwurfsmuster* der Softwareentwicklung erstellt. Sie bilden den Kern der Applikation und verarbeiten mithilfe aller bereits besprochenen Klassen und umgesetzten Algorithmen die Daten externer Komponenten (s. Abschnitt 6.1) in vielerlei Hinsicht.

Controller

Objekte dieser Klassen werden beim Start des CrAc-Cores automatisch instantiiert und bilden das Herzstück des Spring-Frameworks. Jedes mal, wenn ein *Endpoint* via REST genutzt wird, ruft *Spring* automatisch die damit verknüpfte Methode des jeweiligen Controllers auf. Im Code dieser Methode wird anschließend darüber entschieden, welche weiteren Module genutzt werden und welche Daten (als JSON) retourniert werden sollen. Controller sind Teil des überliegenden *Framework-Konzeptes MVC*, welches Spring zur Strukturierung von Klassen und Funktionalität verwendet. Somit sind sie nicht Teil der für den Prototyp umgesetzten Konzepte, müssen aber dennoch hier aufgrund ihrer Relevanz für die Abläufe im System erwähnt werden. Hinsichtlich Abbildung 6.1 muss erwähnt werden, dass beliebig viele Controller-Klassen erstellt werden können und erzeugt werden (daher Menge n). Spring schreibt hierbei nicht vor, wie die Controller-Klassen zu gliedern sind. Im CrAc-Core ist jeder Controller entweder für einen Datentyp oder eine Reihe ähnlicher Funktionalitäten zuständig, um den Prototyp so übersichtlich wie möglich zu halten. Diese sind nachfolgend aufgelistet.

1. Zuständig für Methoden einzelner *Haupt-Datentypen*:
    - CompetenceController, Zugriff auf Kompetenz-bezogene Methoden,
    - TaskController, Zugriff auf Aufgaben-bezogene Methoden,
    - UserController, Zugriff auf User- oder User-Relationship-bezogene

Methoden (mit beliebigen anderen Datentypen).

2. Zuständig für Methoden einzelner *anderer Datentypen*:
   - RoleController, Zugriff auf System-Rollen-bezogene Methoden,
   - EvaluationController, Zugriff auf Feedback-bezogene Methoden. Hier gehen Benutzer-Evaluierungen (s. Abschnitt 5.1) ein,
   - NotificationController, Zugriff auf Nachrichten-bezogene Methoden. Der Controller interagiert mit den Objekten der Klasse Notifications, siehe Abschnitt 3.3.2.
3. Zuständig für einzelne *Funktionsbereiche*:
   - AdminController, Zugriff auf – für Standard User – restriktierte Methoden zur Systemwartung,
   - FilterConfigurationController, Zugriff auf die Einstellungsmöglichkeiten der globalen Matching-Filter-Config, mehr zur Konfiguration später,
   - SynchronizationController, Zugriff auf Methoden, die das Synchronisieren von *KOMET*, das Indizieren der Kompetenzen oder das Laden von beliebigen Testdaten ermöglicht.

Der folgende Code zeigt nicht nur wie der Aufbau eines solchen Controllers und seiner Methoden aussieht, sondern auch, wie *Autowired* verwendet wird, die Strukturierung einer *URL* im Kontext des CrAc-Core funktioniert und wie die im Folgenden erklärte *JSON-Response-Helper* verwendet wird:

```
1  //Annotation, die die Klasse als Spring-Controller im REST-Stil markiert
       und dem Framework Bescheid gibt, sie beim Systemstart automatisch
     zu instantiieren
2  @RestController
3  //Annotation, die sämtliche URLs mit dem ersten Parameter "task" an
     diesen Controller weitergibt
4  @RequestMapping("/task")
5  public class TaskController {
6    //Annotation, die ein automatisches Befüllen und Instantiieren des
       Objektes bei Systemstart bewirkt
7    @Autowired
8    private TaskDAO taskDAO;
9    //Annotation, in der verschiedene Eigenschaften der Methode
       beschrieben werden
10   //"value" gibt den zweiten Parameter der URL an, trifft dieser zu,
       wird die Methode aufgerufen
11   //"method" gibt die Art der Request-Methode an
12   //"produces" gibt dem Framework zu verstehen, welche Art von Wert aus
       der Methode zurück gegeben wird
13   @RequestMapping(value = "/{task_id}", method = RequestMethod.GET,
       produces = "application/json")
14   //Annotation, die signalisiert, dass die Methode Output erzeugt
15   @ResponseBody
16   //Methoden-Deklaration, mit via Annotation zugewiesener Variable
```

```
17    public ResponseEntity<String> show(@PathVariable(value = "task_id")
        Long id) {
18      //Der JSONResponseHelper erzeugt aus dem aus der Datenbank geladenen
         Aufgaben-Objekt
19      //automatisch JSON im Format der REST-Response
20      return JSONResponseHelper.createResponse(taskDAO.findOne(id), true);
21    }
22  }
```

Ein Beispiel zur Veranschaulichung ist der Aufruf der URL `"GET core/-task/2"`.

1. *Spring* leitet den *Request* an den zuständigen, mit `"/task"` annotierten Controller weiter.
2. Der Controller wählt die Methode aus, die den Vorgaben entspricht.
   - Der Mapping-Parameter muss 2 entsprechen oder einem variablen Wert (in diesem Fall variabel).
   - Die Request-Methode muss `GET` entsprechen.
3. Die Methode wird ausgeführt und eine REST-Response wird generiert.
4. Das Resultat der Methode wird als *Response* des Endpoints an den Aufrufer gesendet.

Data-Access

Das Data-Access-Modul verwaltet den Zugang zu Bereichen der *Datenbank und Elasticsearch*. Auf Basis von *JPA* wird beim Starten der Java-Applikation, mit Hilfe von *Autowiring*[3] (s. Abschnitt 6.2), für jeden in der Datenbank persistierten Datentypen des *CrAc-Cores*, ein Objekt als Schnittstelle angelegt. Diese Crud-Repository-Objekte (s. Abschnitt 3.3.1) können im Anschluss genutzt werden, um Daten als Objekte des jeweiliges Datentyps aus der Datenbank zu laden oder zu modifizieren. Die Repositories sind jedoch nur in Controllern (und weiteren zum Systemstart automatisch erstellten Objekten) verfügbar, während später instanziierte und unabhängige Module keinen Zugriff haben. Zwei mögliche Lösungen für dieses Problem bieten sich an.

1. Die Weitergabe der Repository-Referenz an jedes Modul bzw. deren Methoden aus dem Controller.
2. Ein globales Objekt als Schnittstelle zur Sammlung sämtlicher Referenzen.

Ansatz 1 funktioniert zwar, macht das Gesamtsystem jedoch zusehends unübersichtlicher, da eine große Anzahl an Modulen auf eine oder mehrere Tabellen zugreifen muss. Dies führt zu einer großen Menge an Übergabeparametern, die bei jedem Aufruf bedacht werden müssen. Um dieses Problem

[3] http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html

zu umgehen und ein schöneres Softwaredesign (s. Abschnitt 1.3.1), sowie eine verbesserte Lesbarkeit zu erreichen, wird auf *Ansatz 2* zurück gegriffen.

Hierbei werden die Referenzen sämtlicher Zugänge beim Start des CrAc-Core vom SynchronizationController aus automatisch in das *Data-Access-Modul* geladen. Von dort aus können sie aufgrund des verwendeten *Singleton-Patterns* vom gesamten Framework aus genutzt werden. Auch der Zugang auf die *Elasticsearch*-Instanz wird auf diese Weise geregelt, da im *Data-Access-Modul* auch die Zugänge zu den einzelnen Indizes, in Form von *Elastic-Connector-Objekten*, hinterlegt sind.

Der folgende Code zeigt die Attribute und generischen Methoden des Moduls:

```
1  private HashMap<Class<?>, CrudRepository<?, ?>> daos;
2  private HashMap<Class<?>, ElasticConnector<?>> connectors;
3
4  public static <T extends CrudRepository<?, ?>> void addRepo(T obj){}
5  public static <T extends CrudRepository<?, ?>> T getRepo(Class<T> t){}
6
7  public static <T> void addConn(ElasticConnector<T> obj, Class<T> t){}
8  public static <T> ElasticConnector<T> getConnector(Class<T> t){}
```

Die Maps `daos` und `connectors` speichern Referenzen beliebiger Objekte der Klassen *CrudRepository* und *ElasticConnector*. Beide besitzen mit `addRepo()` bzw. `addConnector()` eine Methode um neue Objekte hinzuzufügen, hierbei wird via *Reflection* auf den Typ des jeweiligen Objektes geschlossen und dieser zum Speichern genutzt. Außerdem bietet das Modul mit `getRepo()` und `getConnector()` jeweils eine Methode, um mittels übergebener Klasse nach bereits bestehenden Objekten zu suchen und dieses wie oben beschrieben in beliebigen Modulen zur Verfügung zu stellen. Ein mögliches Problem könnte hierbei der gleichzeitige Zugriff vieler Benutzer gleichzeitig sein, da dieses Objekt keine asynchrone Abarbeitung von Aufrufen unterstützt.

JSON-Response-Helper

Dieses Modul ist für den strukturierten Output im Format *JSON* zuständig, welchen sie auf Basis des Jackson-Mappers und der *REST-Response-Klasse* (s. Abschnitt 3.3.3) generiert. Sämtliche Methoden des JSON-Response-Helpers sind statisch deklariert und bieten globalen Zugriff innerhalb des CrAc-Core. Desweiteren überladen die Methoden eine einzige Hauptmethode, welche die übergebenen Attribute prüft und mit ihnen ein *REST-Response-Objekt* befüllt. Diese sieht wie folgt aus:

```
1  public static <T> ResponseEntity<String> createResponse(T obj, boolean
       success, String cause, ErrorCause error, HashMap<String, Object>
       meta, RESTAction action) {}
```

Alle weiteren Methoden nutzen beliebige Parameter-Kombinationen, die im Anschluss `createResponse()` aufrufen. Aufgrund dessen ist der *JSON-Re-*

sponse-Helper im Rahmen der Möglichkeiten der *REST-Response-Klasse* sehr leicht erweiterbar. Vorrangig wird die Klasse genutzt, um die JSON-Dokumente, die die Controller an die externen Systeme (s. Abschnitt 6.1) weiterleiten, so einheitlich wie möglich zu gestalten. Die damit einhergehende Konsistenz ist für eine einfache Anbindung der *CrAc-Core-Funktionen* äußerst wichtig.

Kompetenz-Verarbeitung

Diese zusammengehörigen Module sind besonders wichtig für die in Kapitel 4 besprochenen Konzepte, sprich den *Matching-Prozess*. Sie sind für den ersten Teil, nämlich das *Traversieren* sowie die *Indizierung* sämtlicher Kompetenz-Daten zuständig. Die Kompetenz-Verarbeitung der Applikation besteht aus zwei stark voneinander abhängigen Modulen, die aufeinander aufbauend alle nötigen Schritte durchführen und das Ergebnis letztendlich im CrAc-Core global zur Verfügung stellen. Dieser Prozess findet beim Starten des Frameworks statt und kann zusätzlich manuell von einem Administrator über den *SynchronizationController* ausgelöst werden. Die Wichtigkeit der Indizierung in zwei Punkten:

- Da auf das aufwändige Traversieren verzichtet werden kann, ist eine Reduzierung der Berechnungsschritte möglich.
- Es sind für das Matching keine Datenbank-Zugriffe bezüglich Kompetenzen nötig.

Aufgrund der beiden Module *Cache* und *Augmenter*, auf die nun näher eingegangen wird, kann der Aufwand im Live-Betrieb daher stark reduziert werden. Zunächst zum internen Kompetenzspeicher des CrAc-Cores, dem *Cache*. Dieser stellt das nötige Umfeld für das Traversieren des Graphen zur Verfügung und besitzt dafür zwei verschiedene Funktionen.

1. Zunächst dient er dazu, die Kompetenzen aus der Postgres-Datenbank in vereinfachter Form in den Speicher der Applikation zu laden und von dort aus zugänglich zu machen. Da für das Traversieren und die Indizierung *alle Kompetenzen* benötigt werden, kann die Anzahl der Datenbankzugriffe auf diese Weise auf einen einzigen umfangreicheren Zugriff reduziert werden.

   Desweiteren stellt der Cache ein zusätzliches *Security Layer* in Bezug auf die Kompetenzen dar. Diese sind, wie in Abschnitt 3.1.3 angemerkt, die einzigen Daten, deren eigentlicher Speicherort nicht die Datenbank des CrAc-Cores ist, weshalb bei der Synchronisierung genau darauf geachtet werden muss, keine Inkonsistenzen im laufenden System zu erzeugen. Mithilfe dieser Zwischenschicht ist es möglich, die relevanten Daten ein weiteres Mal nach dem eigentlichen Synchronisierungs-Prozess zu prüfen, bevor sie in den internen Speicher der Applikation kopiert werden.

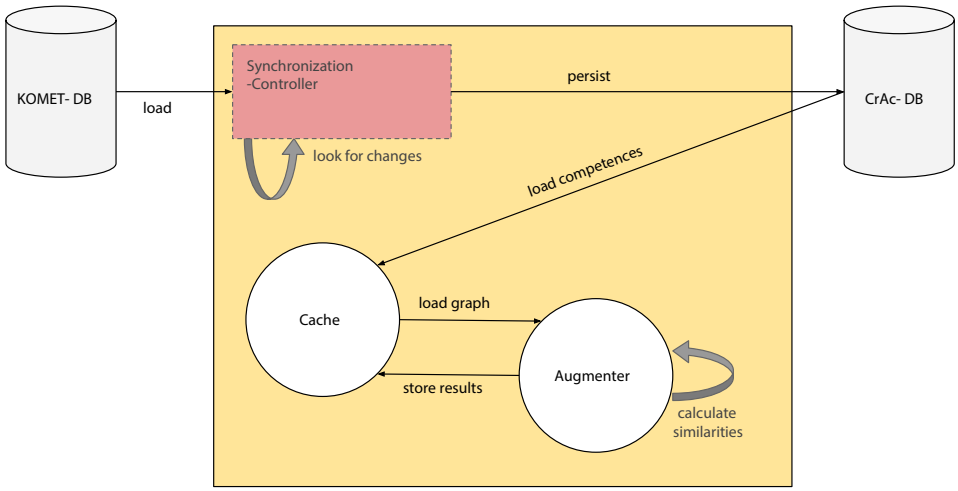


Abbildung 6.5: Prozesskette der Kompetenz-Verarbeitung.

Da erst dort tatsächlich auf sie zugegriffen wird, haben mögliche Fehler beim Kopieren zwischen den Datenbanken keine sofortige Auswirkung.

2. Die zweite Funktion erweitert den gerade erwähnten Cache um die Daten aus dem *Traversierungsverfahren*, beschrieben in Abschnitt 4.2.2. Der Cache dient hierbei als dynamischer Datenspeicher (Kompetenzen angereichert mit Informationen über Verwandte und die errechneten *Similarity-Values*) und globale Schnittstelle für weitere Module.

Eine solche stellt der *Augmenter* dar. Dieser nutzt – als zweites Teilmodul der Kompetenz-Verarbeitung – die Implementierung des *Traversier-Algorithmus*, um zusätzliche Informationen aus dem *Kompetenz-Graphen* zu interpolieren. Hierbei wird exklusiv mit dem *Cache-Modul* interagiert, welches Zugriff auf den Graphen und einen Speicherort für die Resultate bietet. Damit stellt der Augmenter das Bindeglied im *Verarbeitungsprozesses* dar.

Für einen besseren Überblick zeigt Abbildung 6.5, in welcher Reihenfolge die Prozesse der *Kompetenz-Verarbeitung* abgearbeitet werden, beginnend bei der Synchronisierung der *KOMET-Daten*.

Decider

Das Decider-Modul dient als Schnittstelle zum Zugriff auf die Worker-Ebene des CrAc-Core. Welche Funktionalität ansteuerbar ist, ist dabei komplett abhängig von den implementierten Worker-Klassen im System. Diese Worker (s. Abschnitt 3.3.2) umfassen in der hier vorgestellten Version des Cores die *Matching- und Evolutions-Funktionalität*. Abhängig von der gewünschten Funktionalität instantiiert der *Decider* im Hintergrund den richtigen Worker und lässt ihn seinen Code ausführen. Dieses Modul ist jedoch nicht nur

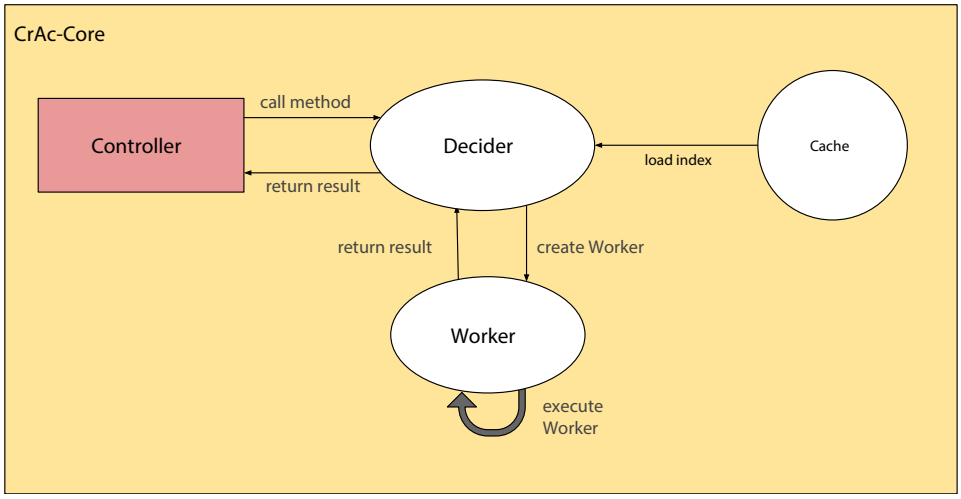


Abbildung 6.6: Abbildung der Worker-Decider-Struktur.

in ihrem Funktionsumfang via Worker erweiterbar, sondern kann zusätzlich dazu auf beliebige Datenstrukturen zurückgreifen bzw. diese implementieren und nutzen, um die an ein uniformes Interface gebundenen Worker nach beliebigen Regeln auszuführen. Hier die Vorteile des Deciders zusammengefasst:

- Vereinfachte Aufrufe durch Maskierung der Worker.
- Verkettung beliebiger Worker in Schnittstellen-Methoden.
- Ablegen beliebiger Worker in beliebiger Datenstruktur, dies beeinflusst:
  - die Zeit der Ausführung (z.B. durch Benutzung eines Thread-Pools),
  - die Reihenfolge der Ausführung (Festlegung möglich in beliebigen Datenstrukturen).
- Erweiterbarkeit durch die Einbindung weiterer Worker-Einheiten.

In Abbildung 6.6 ist die Decider-Einheit und ihr Zugriff auf die Worker-Schicht dargestellt.

Notification-Distributor

Ähnlich dem Decider, dient auch der Notification-Distributor dazu, Hintergrund-Prozesse und deren involvierte Objekte zu maskieren und eine Schnittstelle zu diesen zu bieten. In diesem Fall geht es um Objekte der unterschiedlichen Implementierungen der *Notification-Klasse*, siehe Abschnitt 3.3.2. Um die Interaktion mit diesen einfacher zu gestalten, besitzt der Distributor mehrere Funktionen:

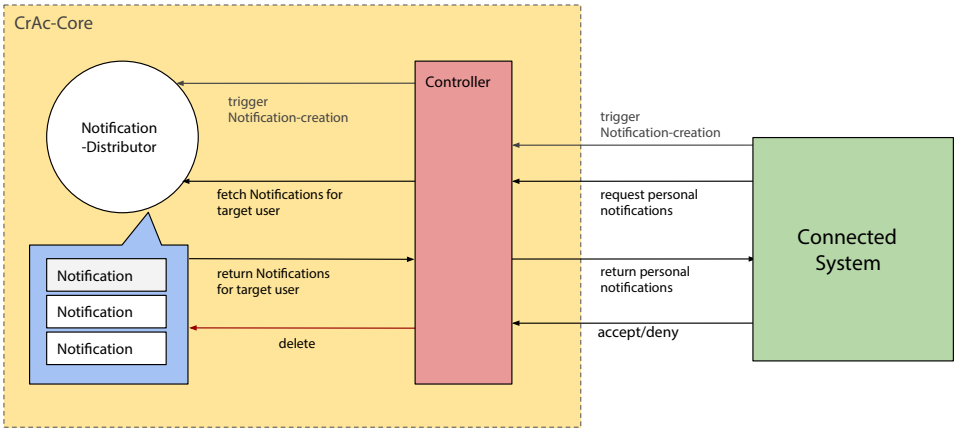


Abbildung 6.7: Abbildung der Notification-Distributor-Interaktionen.

- Das Modul dient als Maske zum instantiieren von unterschiedlichen Notifications.
- Das Modul cached die instantiierten Notifications, bis auf diese mit `accept()` oder `deny()` reagiert wird.
- Das Modul bietet eine globale Schnittstelle, mit der auf die gespeicherten Notifications zugegriffen werden kann.

Notifications werden bewusst im Cache der Applikation gelassen, um die Menge der Datenbankzugriffe zu reduzieren. Da es sich nur um voreingestellte Benachrichtigungen und nicht um persönliche Konversationen zwischen verschiedenen Usern handelt, werden diese nach Annahme oder Ablehnung komplett aus dem Core entfernt. Der Distributor ist dem Decider auch hinsichtlich Skalierbarkeit sehr ähnlich, da er mit der Menge an Datentypen, für die er eine Schnittstelle darstellt, mitwächst. In Abbildung 6.7 sind der Distributor und seine Interaktion mit den Notifications dargestellt.

Matching-Filter-Konfiguration

Auch das letzte Modul dieses Abschnitts, es komplettiert den CrAc-Core, hat mit dem Matching-Prozess zu tun. Wie der Name des Moduls bereits aussagt, enthält es die globale Konfiguration sämtlicher *Matching-Filter*. Jeder dieser Filter modifiziert (siehe dazu Abschnitt 5.2) die Basis-Werte im *User-Task-Matching*. Die Konfiguration enthält die Liste aller im Prozess anzuwendender Filter und dient als gobal zugängliche Schnittstelle zu diesen. Die Reihenfolge und Art der gesetzten Filter kann per *Endpoint* von einem Benutzer mit Administrator-Rechten angepasst werden.

Der CrAc-Core hat jedoch nicht nur die Möglichkeit einer einzelnen Konfiguration, denn diese ist auch auf Methode- und Endpoint-Ebene, abhängig von User-Input, anpassbar. Um dieses Ziel zu erreichen, werden die festge-

legten Filter bei jeder Nutzung der *Matching-Filter-Konfiguration* kopiert und übergeben. Diese Kopie kann problemlos in der zugreifenden Methode modifiziert und anschließend für den Matching-Prozess verwendet werden. In folgendem Code werden die wichtigen Teile der Klasse und ihre Funktionsweise gezeigt:

```
1  public class GlobalMatrixFilterConfig {
2
3    //Die globale Instanz
4    private FilterConfiguration config = new FilterConfiguration();
5    private static GlobalMatrixFilterConfig instance = new
       GlobalMatrixFilterConfig();
6
7    //Das Hinzufügen eines Filters
8    public static void addFilter(CracFilter filter){
9      instance.config.addFilter(filter);
10   }
11
12   //Das Leeren der Filter
13   public static void clearFilters(){
14     instance.config.clearFilters();
15   }
16
17   //Zugriff auf die Filter in Form einer Kopie
18   public static FilterConfiguration cloneConfiguration(){
19     return instance.config.clone();
20   }
21
22 }
```

Weiters zeigt Abbildung 6.8 die Prozess-Kette von der Setzung der globalen Konfiguration bis zum *Matching*.

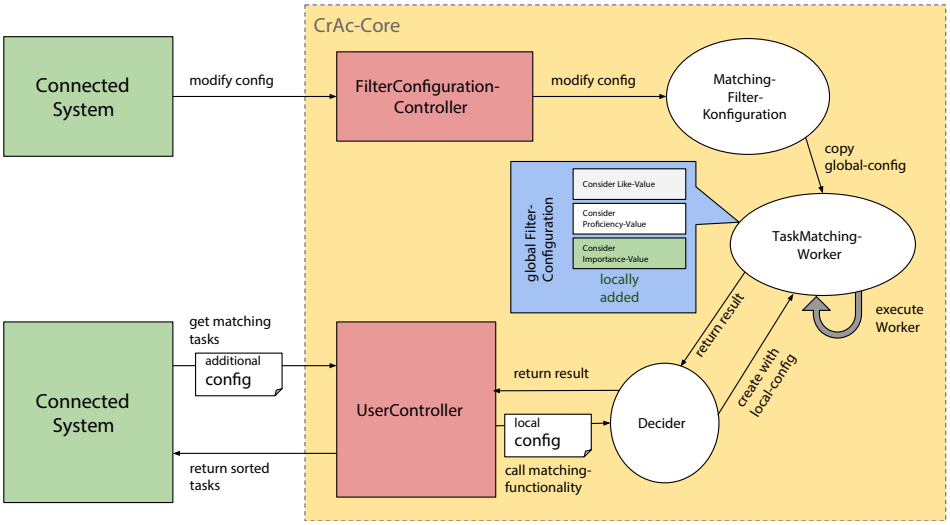


Abbildung 6.8: Prozess-Kette der Matching-Filter-Konfiguration.

Kapitel 7

Use-Case und Tests

Da das System und sämtliche Komponenten und Module, sowie die Klassen und Konzepte dahinter nun etabliert sind, kann der *CrAc-Core* in seiner Gesamtheit betrachtet werden. Zu diesem Zweck wird in einem *Use-Case* die Zusammenarbeit zwischen dem CrAc-Core und einem potenziellen *Frontend* näher beleuchtet. Weiters werden Tests auf Basis verschiedener *Personen* (s. Abschnitt 3.1.1) im System und deren Daten durchgeführt. Die Datensets durchlaufen eine definierte Menge an *Matching-Evaluations-Iterationen*, deren finale und zwischenzeitlichen Ergebnisse, sowie der messbare Einfluss der einzelnen Filter in einem eigenen Abschnitt aufgearbeitet dargestellt werden. Im Anschluss werden mögliche Anpassungen auf Basis der Werte diskutiert.

7.1 Use-Case

Zunächst zum Use-Case. Dieser dreht sich um den Kernaspekt des Frameworks, das *Matching* und die *Evaluierung* einer durchgeführten Task. Er zeigt, welche Benutzer nötig sind bzw. welche Schritte auf Seiten eben dieser durchgeführt werden müssen, um den gesamten Prozess zu durchlaufen. Zur Vereinfachung wird angenommen, dass der Zugriff auf die *API* des CrAc-Cores mittels passendem *Frontend* (siehe zum Vergleich Abschnitt 6.1) erfolgt. Weiters handelt es sich bei den agierenden Usern nicht nur um die Objekte im CrAc-Core, sondern um real agierende Personen, die diese Objekte als Repräsentation im System nutzen (siehe hierfür auch Abschnitt 3.1.2). Nachfolgend wird auf die User in dem Use-Case eingegangen. Es existieren:

- Ein Admin-User, der die Kompetenzen anlegt und synchronisiert,
- Ein Creator-User (eine zusätzliche Rolle, die einen Benutzer zum Anlegen von Aufgaben berechtigt), der eine Aufgabe anlegt,
- Ein Standard-User, der nach einer passenden Aufgabe sucht.

Folgende Schritte passieren nun im *CrAc-Core*, um das gewünschte Ziel -

eine passende Aufgabe finden, ausführen und evaluieren - zu erreichen. Alle Schritte sind außerdem nach ausführendem User sortiert.

Der Administrator

Als ersten Schritt kümmert sich ein Administrator um das Anlegen und Laden der Kompetenzen.

1. Zunächst müssen die nötigen Daten (Raster und Kompetenzen) in der verbundenen KOMET-Instanz existieren.
   (a) Dafür ruft der Administrator in seinem Browser die mit der jeweiligen CrAc-Plattform verbundene KOMET-Instanz auf.
   (b) Nun kann er entweder neue Daten anlegen und verknüpfen (s. Abschnitt 3.1.3), oder von einer weiteren KOMET-Instanz via *XML-Import* laden.
2. Um die Kompetenzen in CrAc verfügbar zu machen, muss der Administrator den *Synchronisierungs- und Indizierungsprozess* (s. Abschnitt 6.2) einleiten. Dieser wird im Admin-Bereich des – an die CrAc-Instanz angeschlossenen – Frontends per Button ausgelöst.
   (a) Das *Frontend* ruft nun die jeweilige Funktionalität des CrAc-Cores (s. Abschnitt 6.2) mittels API-Call auf.
   (b) Wie bei jedem *Request* (im Vergleich dazu Abschnitt 6.2) lädt *Spring* nun den zugehörigen Code und führt die Anfrage mithilfe der Module und zugehörigen Hilfsklassen aus.
   (c) Nach der Durchführung teilt der Core dem Frontend die Ergebnisse via *JSON* mit.
   (d) Auf dieser Basis teilt das Frontend dem Administrator mit, ob die Synchronisierung fehlerfrei verlaufen ist oder nicht.

Der Creator

Als Nächstes müssen eine oder mehrere Aufgaben im System angelegt werden, um ein *Matching* möglich zu machen. In diesem Fall wird ein einzelner Aufgabenbaum mit mehreren hierarchisch untergeordneten Einzelaufgaben (s. Abschnitt 3.1.2 angelegt.

1. Hierfür ruft der Creator den (einem Administrator oder Creator vorbehaltenen) Bereich des Frontends auf, der die Erstellung eines Aufgabenbaums ermöglicht, bzw. die nötige *Eingabemaske* bietet.
   (a) Das Frontend ruft nun den entsprechenden Endpoint der API auf und übergibt die Daten der Eingabemaske.
   (b) Das Backend speichert die Aufgabe und meldet das Ergebnis dem Frontend via *JSON*.

   (c) Der Creator ist nun *Leader* (s. Abschnitt 3.1.2) an der obersten `ORGANISATIONAL`-Task im von ihm erstellten Aufgabenbaum.

2. Nach positiver Rückmeldung wechselt der Creator nun in die – vom Frontend angebotene – *Detailansicht* seiner gerade erstellten Aufgabe.
3. Da er *Leader* dieser ist, hat er das Recht, weitere Unteraufgaben hinzuzufügen, auch hierfür nutzt er die zur Verfügung stehende *Eingabemaske*.
   (a) Das Frontend ruft für jede abgeschickte Aufgabe den entsprechenden Endpoint der API auf und übergibt die Daten.
   (b) Das Backend speichert die Aufgaben und ihre Position im Baum und meldet das Ergebnis dem Frontend via *JSON*.
4. Nach Anlegen des Baumes ist der nächste Schritt das Zuweisen der für die Ausführung nötigen Kompetenzen, dafür wechselt der Creator nacheinander auf die Detailansicht der *ausführbaren* Aufgaben der Typen `WORKABLE` und `SHIFT`.
   (a) Das Frontend nutzt hier zunächst eine per API angestoßene Funktion des CrAc-Cores, die ein Set aus vorgeschlagenen Kompetenzen auf Basis von (im Vorfeld gewählten) Kompetenz-Arealen zur Verfügung stellt.
   (b) In einer separaten Eingabemaske können diese ausgewählt und mitsamt gewünschten Beziehungswerten an den Core geschickt werden.
   (c) Dort werden im Anschluss auf Basis dieser Daten neue *Competence-Task-Relationships* (s. Abschnitt 3.2) erstellt und gespeichert. Die Kompetenzen sind der jeweiligen Aufgabe nun zugewiesen.
5. Als letzter Schritt müssen die Aufgaben des Baumes nun freigegeben werden. Dies ist nötig, damit sie im *Matching-Prozess* berücksichtigt werden.
   (a) Hierfür wird (wie beim Anlegen von Unteraufgaben) zurück in die – vom Frontend zur Verfügung gestellte – *Detailansicht* des obersten Knotens gewechselt und der jeweilige Knopf zur *Freigabe* der Aufgabe ausgelöst.
   (b) Im Hintergrund wird wiederum ein entsprechender API-Call durchgeführt, der den CrAc-Core anweist, die Aufgaben im Baum von `NOT_PUBLISHED` auf `PUBLISHED` zu setzen (falls möglich).

Der Standard-User

Nun, da die nötigen Daten in richtiger Form existieren, kann zum dritten Teil dieses Use-Case übergegangen werden, dem Suchen, Ausführen/Beenden und Evaluieren von Aufgaben durch den freiwilligen Helfer.

1. Zunächst sucht der User nach einer passenden Aufgabe. Dafür ruft er den Bereich des Frontends auf, in dem diese Aufgaben strukturiert angezeigt werden.
   (a) Der nun ausgeführte API-Call setzt im Core den *Matching-Prozess* (s. Kapitel 4) in Gang.
   (b) Die gefundenen Aufgaben werden sortiert in Form von *JSON* zurückgeliefert und anschließend im angesprochenen Bereich des Frontends dargestellt.
2. Mit einem Klick auf eine der Aufgaben wechselt das Frontend in die Detailansicht, dort wird anschließend per Button die *Teilnahme* an dieser Aufgabe bestätigt.
   (a) Der durchgeführte Aufruf des Frontends löst eine erneute Überprüfung auf Seiten des CrAc-Cores aus.
   (b) Bei positivem Ausgang wird eine neue *User-Task-Relationship* des Typs `PARTICIPATING` erstellt, welche den Benutzer an die gewählte Aufgabe bindet.
3. Um die Aufgabe nach der realen Ausführung als beendet zu erklären, wird eine Options-Box der Aufgaben-Detailansicht im Frontend genutzt.
   (a) Der durchgeführte Aufruf passt das Attribut `completed` in der jeweiligen *User-Task-Relationship* an.
   (b) Haben alle Teilnehmer die Aufgabe beendet, werden die *Leader* der Aufgabe davon in Kenntnis gesetzt (s. Abschnitt 6.2).
   (c) Bei Abschluss kann entweder ein Leader einen Aufruf zum Evaluieren an alle Teilnehmer senden (s. Abschnitt 3.3.2) oder jeder Teilnehmer kann die Evaluierung selbst anfordern. Beides wird in der Detailansicht der abgeschlossenen Aufgabe ausgelöst.
4. Der letzte Schritt ist nun das *Evaluieren* der Aufgabe.
   (a) Zunächst muss die Benachrichtigung zum Evaluieren im Frontend angenommen werden.
   (b) Diese leitet an eine eigene Eingabemaske weiter, welche die jeweiligen Felder zur Bewertung enthalten.
   (c) Alle Informationen der Maske werden per API-Call an den CrAc-Core gesendet.
   (d) Dieser prüft die Evaluierung und setzt den Mechanismus zur Modifikation der involvierten Daten in Gang (siehe dazu Kapitel 5).

7.2 Tests

Der nächste Abschnitt beschäftigt sich mit den *durchgeführten Tests* am CrAc-Core. Es handelt sich hierbei nicht um Tests, welche die fehlerfreie Ausführung von Plattform- und Modul-Funktionalität betreffen (beispielsweise Unit-Tests), da das einwandfreie Funktionieren des CrAc-Cores bereits im Vorfeld sicher gestellt wurde, sondern um das Testen der Kernaspekte der Plattform, nämlich *Matching- und Evaluierungsmechanismen*, sowie die Analyse der Ergebnisse. Um aussagekräftige Testergebnisse zu erhalten, wird ein realitätsnahes, detailliertes Szenario verwendet, welches im Folgenden erklärt wird. Zuvor jedoch zu einigen Schwierigkeiten den Aufbau betreffend.

Schwierigkeiten

Probleme beim Aufbau von Tests ergeben sich vor allem aus der *Komplexität* und Vielzahl der durchzuführenden Schritte am CrAc-Core, sowie durch ein *fehlendes Frontend*, was die Interaktion mit realen Testern sehr schwierig macht. Zwar ist es möglich, automatisiert immer wieder die einzelnen Teile des *Matching-Evaluation-Loops* (s. Kapitel 4) durchzugehen, dies erfordert aber entweder das Implementieren einer eigenen Software, die mit den jeweiligen *Endpoints* des CrAc-Cores interagiert, oder ein eigenes Modul im CrAc-Core, das seinerseits die ansonsten in den Controller-Methoden liegenden Aufrufe tätigt. Beide Ansätze sind sehr zeitaufwändig, beziehungsweise widersprechen der Art, auf die der CrAc-Core genutzt werden sollte.

Tests durch reale Benutzer stellen ihrerseits ein Problem dar, da zur Zeit der Entstehung dieser Arbeit kein fertiges Frontend existiert. Aufgrund dieser Tatsache müssten echte User direkt mit der *API* des CrAc-Core kommunizieren, beispielsweise mithilfe eines Tools wie *Postman*[1]. Die ungewohnte Oberfläche, sowie das fehlende Verständnis für das verwendete Tool und die sichtbaren Rohdaten (in Form von JSON) würden das Testen mit der Zielgruppe schwierig gestalten.

Um die hier präsentierten Probleme zu umgehen, werden die Tests am CrAc-Core zwar für verschiedene User direkt mithilfe eines Tools an den entsprechenden Endpoints der API durchgeführt, es existiert jedoch lediglich ein einzelner (technisch geschulter) Tester, der die *Aktionen aller Test-User* am System vornimmt. Außerdem zeichnet der Tester die Ergebnis-Daten der Testläufe auf, dies ist wichtig für die nachfolgende *Aufbereitung*.

7.2.1 Aufbau

Nun zum Aufbau. Um die Tests durchführen zu können, werden als Grundgerüst Datensätze sämtlicher Hauptdatentypen (s. Abschnitt 3.1) benötigt.

[1] https://www.getpostman.com/

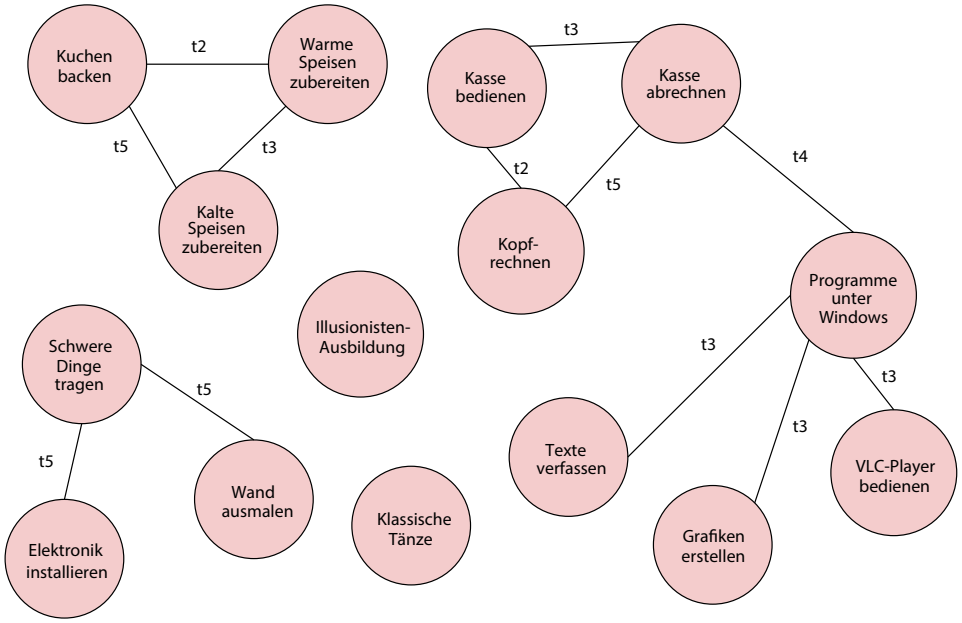


Abbildung 7.1: Die 15 Kompetenzen und ihre Beziehungen untereinander.

Diese müssen für Anpassungen ihrer jeweiligen Meta-Daten mittels Beziehungsdatentypen (s. Abschnitt 3.2) verknüpft sein. Die Erstellung der Daten erfolgt hierbei in der selben Reihenfolge, in der sie auch in der Realität erfolgen würde.

Die angegebenen Werte werden im CrAc-Core, wie in Abschnitt 3.2 beschrieben, eigentlich im Intervall $[-100, 100]$ persistiert und nur für die Berechnungen und Modifikationen in den Wertebereich $[-1, 1]$ übersetzt. Für eine verbesserte Übersichtlichkeit der Ergebnisse werden sie hier allerdings von Anfang an im Bereich $[-1, 1]$ angegeben.

Zunächst daher zur Struktur der Kompetenzen und ihrer – den Graphen bildenden – Kompetenz-Beziehungen. Der sich daraus bildende Graph (s. Abbildung 7.1) nutzt für die Kanten fünf unterschiedliche Gewichtungen. Diese stehen jeweils für die Werte $t_1 = 1.0$, $t_2 = 0.8$, 0.6 (t_3), 0.4 (t_4) und 0.2 (t_5).

Die 5 Benutzer der durchgeführten Tests sind namentlich und zusammen mit ihren Beziehungen untereinander in Abbildung 7.2 dargestellt, der Wert der Kanten bezieht sich hierbei auf das *Like-Level* der jeweiligen Beziehung.

Es muss ebenfalls auf die Beziehungen zwischen Benutzern und Kompetenzen eingegangen werden, welche für die gelernten Fähigkeiten der jeweiligen Person im System stehen. Diese sind in Tabelle 7.1 dargestellt. Die Abkürzung l steht hierbei für *Like-Level*, die Abkürzung p_u für *Proficiency-Level*.

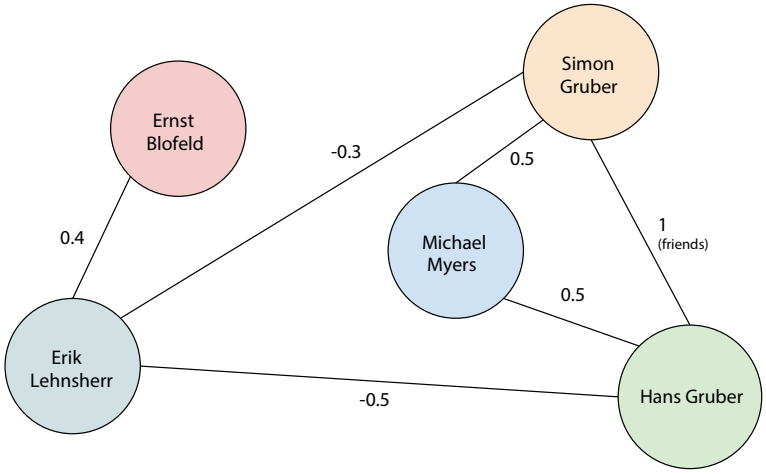


Abbildung 7.2: Die 5 Benutzer und ihre Beziehungen untereinander.

User-Name	Kompetenz-Name	l	p_u
Hans Gruber	Schwere Dinge tragen	0.6	1.0
	Klassische Tänze	0.9	0.7
	Kopfrechnen	0.3	0.5
Simon Gruber	Elektronik installieren	1.0	1.0
	Kalte Speisen zubereiten	0.9	0.7
	Schwere Dinge tragen	0.5	0.5
	Illusionisten-Ausbildung	-0.5	0.5
Michael Myers	Kasse bedienen	0.9	1.0
	Kasse abrechnen	0.9	1.0
	Kopfrechnen	0.5	1.0
	Kuchen backen	0.0	0.3
Erik Lehnsherr	Texte erfassen	1.0	0.8
	Grafiken erstellen	1.0	0.9
	Illusionisten-Ausbildung	-0.5	0.6
	Bedienen von Windows-Programmen	0.3	0.3
	VLC-Player bedienen	0.3	0.3
Ernst Blofeld	Wand ausmalen	0.8	0.2
	Kuchen backen	0.8	1.0
	Kalte Speisen zubereiten	0.9	1.0
	Warme Speisen zubereiten	1.0	1.0
	Schwere Dinge tragen	0.3	0.3

Tabelle 7.1: Tabelle der User-Competence-Relationships.

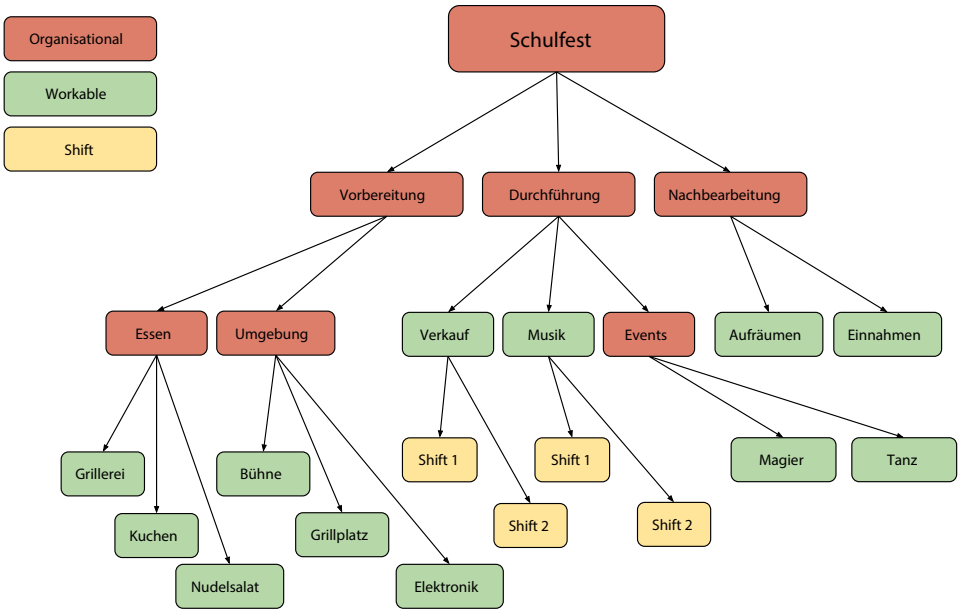


Abbildung 7.3: Der Aufgabenbaum inklusive einer Legende der Task-Types.

Der letzte fehlende Aspekt sind die Aufgaben selbst und hierbei ist das verwendete Gesamt-Szenario von großer Bedeutung. Um einen möglichst großen Rahmen an Aktivitäten abzudecken und die Zuordnung der Beziehungen einfach und verständlich zu halten, wurde hierfür ein Schulfest[2] ausgewählt. Abbildung 7.3 zeigt den auf diese Weise entstandenen Aufgabenbaum inklusive der jeweiligen `TaskTypes`.

Auch hier muss auf die Beziehungen zwischen dem Datentyp – Task – und den Kompetenzen eingegangen werden. Diese stellen die benötigten Fähigkeiten dar, um die Aufgabe auch ausführen zu können. Kompetenzen müssen jedoch nicht in jedem Fall vergeben werden. Da Tasks des Typs `ORGANISATIONAL` nicht für das Matching gedacht sind, ist eine Zuweisung in diesem Fall nicht notwendig. Auch für ausführbare Aufgaben ist die Kompetenz-Zuweisung nicht zwangsweise notwendig, der CrAc-Core vergibt dann als Matching-Score den Mittelwert 0.5. Der Nachteil dabei ist, dass für solche Aufgaben keinerlei Modifikationen auf Kompetenz-Basis erfolgen können, weder im Matching, noch in der Evaluierung. Darum sollten nur sehr einfach zu lösende Aufgaben, ohne erkennbare lehrreiche Erfahrung oder Anspruch, auf die Kompetenz-Zuweisung verzichten. Die Beziehungen sind dargestellt in Tabelle 7.2, die Abkürzung *il* steht hierbei für *Importancy-*

[2]Hierbei handelt es sich außerdem um einen realen Anwendungsfall für das CrAc-Projekt nach Fertigstellung.

Level, die Abkürzung p_t für *Proficiency-Level*.

Task-ID	Aufgaben-Name	Kompetenz-Name	il	p_t
1	Verkauf am Schulfest	Kopfrechnen	0.7	0.5
		Kasse bedienen	0.7	0.4
2	Musikalische Untermalung am Schulfest	Bedienen von Windows-Programmen	0.9	0.5
		VLC-Player bedienen	1.0	0.4
3	Aufräumen nach dem Schulfest	Keine Kompetenzen zugewiesen!		
4	Die Einnahmen der Kasse rechnen	Kasse abrechnen	1.0	0.6
		Kopfrechnen	0.8	0.6
5	Vorbereitung des Kuchens	Kuchen backen	1.0	0.6
6	Vorbereitung der Grillerei	Keine Kompetenzen zugewiesen!		
7	Vorbereitung des Nudelsalates	Kalte Speisen zubereiten	1.0	0.6
8	Vorbereitung des Grillplatzes	Schwere Dinge tragen	1.0	0.6
		Wand ausmalen	1.0	0.3
9	Vorbereitung der Bühne	Schwere Dinge tragen	1.0	0.7
		Elektronik installieren	1.0	0.6
10	Vorbereitung der Elektronik	Schwere Dinge tragen	0.6	0.2
		VLC-Elektronik installieren	1.0	0.9
11	Auftritt als Illusionist	Illusionisten-Ausbildung	1.0	1.0
12	Vorführung klassischer Tänze	Klassische Tänze	1.0	0.9

Tabelle 7.2: Tabelle der Task-Competence-Relationships.

Nachdem die Test-Daten nun hinreichend bekannt sind, wirft der nächste Abschnitt einen Blick auf die Tests und ihre Ergebnisse.

7.2.2 Filter-Test

Der erste Test beschäftigt sich mit dem Effekt der Modifikations-Filter auf den Matching-Score.

Durchführung

Hierfür wird mehrmals die Suche der Benutzer nach Aufgaben unter komplett gleichen Bedingungen durchgeführt. Der einzige Unterschied liegt in den abwechselnd aktivierten Filtern. Weiters ist es für das Testen des *User-Relation-Filters* (s. Abschnitt 4.2.4) notwendig, den Benutzern einige der Aufgaben zuzuweisen. Dies erfolgt zufällig. Die Zuweisung für diesen Test sieht folgendermaßen aus (die Benutzer nehmen an den aufgelisteten Aufgaben teil):

- Hans Gruber:
  - Verkauf von Essen, Getränken und Tombola-Losen am Schulfest.
  - Vorbereitung des Grillplatzes.
- Simon Gruber:
  - Vorbereitung des Nudelsalates.
  - Vorbereitung der Elektronik.
- Erik Lehnsherr:
  - Musikalische Untermalung am Schulfest.
  - Vorbereitung der Elektronik.
- Michael Myers:
  - Verkauf von Essen, Getränken und Tombola-Losen am Schulfest.
  - Die Einnahmen der Kasse am Vormittag nach dem Schulfest rechnen.
- Ernst Blofeld:
  - Vorbereitung des Kuchens.
  - Vorbereitung des Nudelsalates.

Ergebnisse

Normalerweise werden die Ergebnisse für Aufgaben, an denen der jeweilige User bereits partizipiert, gefiltert. Diese Funktion ist für eine umfangreichere Auswertung dieses Tests jedoch deaktiviert. Nun zu den Resultaten des Tests. Diese werden anhand von zwei Usern erörtert und sind visualisiert in den Abbildungen 7.4 und 7.5. Hierbei werden *Task-IDs* entsprechend der Nummerierung in Tabelle 7.2 verwendet, es werden für mehr Übersichtlichkeit jedoch immer *ID* und *Name* jeder Aufgabe genannt. Die restlichen Testergebnisse sind in der beigelegten CD dieser Thesis zu finden.

Zunächst fallen die Ergebnisse auf, die sich unabhängig vom Filter nie verändern. Dies kann mehrere Gründe haben:

1. Es sind keine Kompetenzen zugewiesen, daher bleibt der Wert auf 0.5.

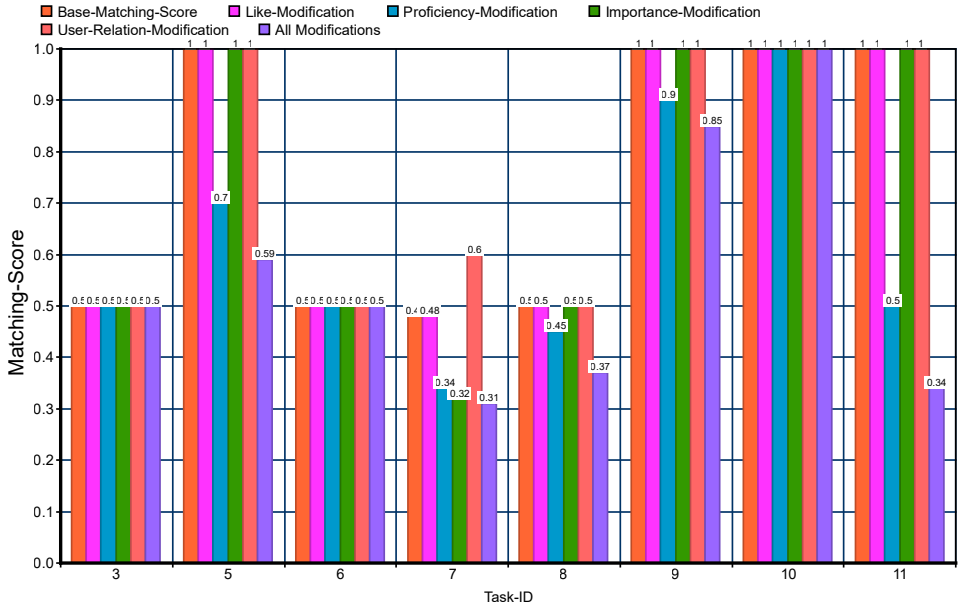


Abbildung 7.4: Ergebnisse von Benutzer „Simon Gruber“ mit unterschiedlichen Filtern.

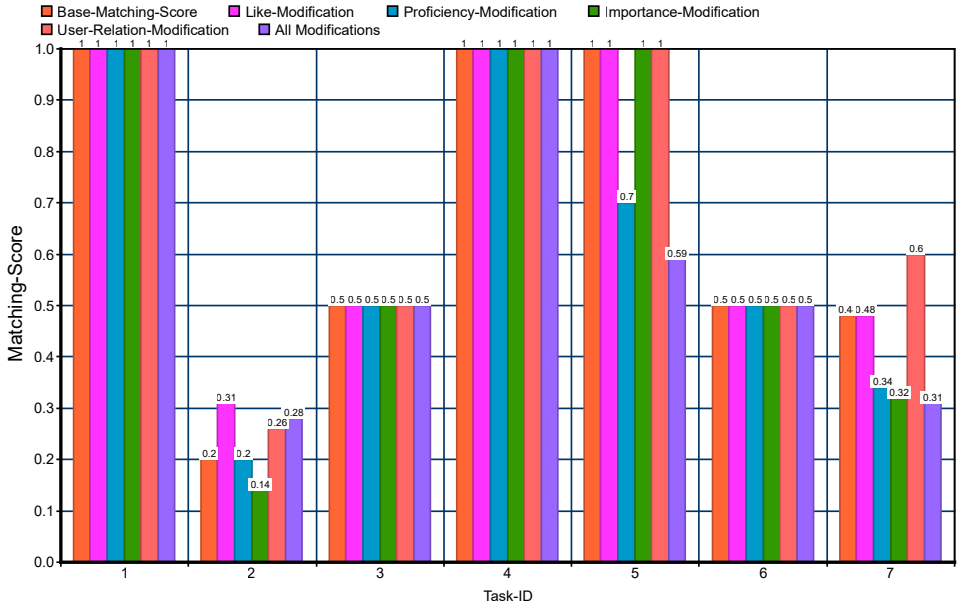


Abbildung 7.5: Ergebnisse von Benutzer „Michael Myers“ mit unterschiedlichen Filtern.

2. Die Filter modifizieren zwar *Similarity-Values*, diese sind jedoch nicht relevant für das Matching, weil das System für dieselbe Kompetenz einen besseren Wert findet, oder weil der Wert bei jeder Modifikation einen Extremwert (0 oder 1) überschreiten würde (s. Abschnitt 4).

Fall 1 tritt bei den beiden Aufgaben „Aufräumen am Schulfest“ (ID 3) und „Vorbereiten der Grillerei“ (ID 6) auf. Da beide Aufgaben weder Ansprüche haben noch eine lehrende Erfahrung bieten, sind ihnen keinerlei Kompetenzen zugewiesen. *Fall 2* tritt vor allem dann auf, wenn ein Benutzer alle Auflagen einer Aufgabe erfüllt oder übertrifft. Ein Beispiel hierfür wäre das Ergebnis zu „Verkauf am Schulfest“ (ID 1) in Abbildung 7.5. Die Werte aus allen – zum Matching verwendeten – Meta-Daten sind hier so hoch, dass die einzelnen Scores zurück auf 1 reduziert werden müssen und der Wert dadurch gleich bleibt. Eine weitere Auffälligkeit stellt die Tatsache dar, dass sich der Ergebniswert beim Matching mit allen Filtern in den meisten Fällen sehr von den restlichen Scores unterscheidet. Dies ist in fast allen Aufgaben beider Abbildungen zu beobachten und liegt daran, wie die Filter abgearbeitet werden. Diese modifizieren die Einträge der Matrix der Reihe nach. Schwächt ein Filter einen Wert ab, so rechnet der nächste Filter mit dem abgeschwächten Wert weiter, nicht mit dem Basis-Wert. Dies kann sich in beide Richtungen (Wert 0 und Wert 1) aufschaukeln. Die Reihenfolge der Filter spielt daher eine große Rolle für das Endergebnis.

Die anderen Ergebnisse entsprechen weitestgehend den Erwartungen. Die Modifikation des *Like-Levels* passt sich den Extremwerten des Matchings an (Aufgabe „Auftritt als Illusionist“ (ID 11) in Abbildung 7.4), indem sie schwächer wird und ändert die Werte ansonsten gemäß den gegebenen Meta-Daten (Aufgabe „Musikalische Untermalung“ (ID 2) in Abbildung 7.5).

Der *Proficiency-Filter* schwächt den Score bei zu geringen Werten ab (Aufgabe „Vorbereitung des Nudelsalat“ (ID 7) in Abbildung 7.5 und Aufgabe „Vorbereitung des Kuchens“ (ID 5) in Abbildung 7.4) und der *Importance-Filter* sorgt für einen niedrigeren Score, wenn der Benutzer einen schlechten Matching-Score für – in Relation – wichtige Kompetenzen aufweist (Aufgabe „Vorbereitung des Nudelsalat“ (ID 7) im Falle beider User).

Der letzte Filter – der *User-Relation-Filters* – bezieht die Daten für die Modifikation aus der Beziehung des Suchenden zu anderen Teilnehmern, daher müssen in diesem Fall die Werte der Aufgaben heran gezogen werden, an denen auch andere User partizipieren. Hier beeinflussen die beiden dargestellten Freiwilligen die Aufgabe „Vorbereitung des Nudelsalates“ (ID 7). Der Wert in Abbildung 7.5 ist deshalb so hoch, weil Simon Gruber bereits an der Aufgabe teilnimmt und beide in einer positiven Beziehung zueinander stehen. Ein Beispiel für einen Werte-Anstieg, der keinen Einfluss auf das Gesamt-Matching hat, zeigt hier die Aufgabe „Vorbereitung des Grillplatzes“ (ID 8) in Abbildung 7.4. Der Filter passt hier lediglich Similarity-Values an, die vom Matching – aufgrund eines besseren Wertes – verworfen wer-

den, der Rest sind Extremwerte (0 oder 1). Die Werte, die am Ende des Matchings den Score ergeben, werden nicht angetastet.

7.2.3 Test des Matching-Kreislaufs

Der zweite Test wirft einen genauen Blick auf die Auswirkung des *Matching-Evaluation-Loops* (siehe Kapitel 4) in Bezug auf die Beziehungsdaten der involvierten Benutzer. Es soll die Frage beantwortet werden, wie viel die involvierten Benutzer laut dem System bei 10 Teilnahmen im selben Projekt lernen können und wie sie die Beziehungen zu anderen Freiwilligen ausbauen. Auch hier wird das in Abschnitt 7.2.1 beschriebene Test-Setup verwendet.

Durchführung

Die Regeln der Durchführung für 10 Iterationen:

- Pro Iteration führt jeder der Test-User eine Aufgabe durch.
- Die Gesamtheit der Iterationen ist in zwei Sets zu je 5 aufgeteilt.
  - In den ersten 3 Iterationen jedes Sets werden die Aufgaben für alle Benutzer zufällig gewählt, um das oft unvorhersehbare menschliche Verhalten zu berücksichtigen.
  - In den letzten beides Iterationen sucht der Tester die Aufgaben für alle Benutzer aus, um gewisse Interaktionen zwischen mehreren Benutzern und Aufgaben zu forcieren.
- Die Reihenfolge, in der die Benutzer ihre Aufgaben zugewiesen bekommen, ist zufällig. Dies ist wichtig, da sich das Matching durch den *User-Relation-Filter* bei unterschiedlicher Reihenfolge der Anmeldung unterscheiden kann.
- Die Evaluierung jeder Aufgabe durch den jeweiligen Benutzer ist ebenfalls zufällig.

Wie oben bereits beschrieben, werden alle Aktionen der Personen vom Tester durchgeführt. Hierbei wird, wie in der Beschreibung angemerkt, so viel wie möglich dem Zufall überlassen, um die Ergebnisse so wie in der Realität unabhängig vom Tester zu machen.

Ergebnisse

Auch im zweiten Test wird, stellvertretend für alle, auf zwei Test-User genauer eingegangen. Hier der Verweis auf die vollständigen Tests in der beigelegten CD der Thesis. Aufgrund der großen Menge an Daten aus den Iterationen sind die Ergebnisse auf jeweils drei Abbildungen verteilt, welche die *Proficiency- und das Like-Level* zu verschiedenen Kompetenzen, sowie das *Like-Level* gegenüber den anderen Usern zeigt. Die beiden User haben in den 10 Durchläufen die in Tabelle 7.3 und Tabelle 7.4 folgenden Aufgaben

absolviert und evaluiert[3], die IDs der Aufgaben entsprechen ihrer Nummerierung in Tabelle 7.2.

Iteration	1	2	3	4	5	6	7	8	9	10
Task	3	1	8	9	10	8	2	9	12	1
User-Eval	0.45	0.35	0.35	−0.45	−0.40	0.05	0.30	−0.15	−0.20	−0.80
Task-Eval	0.25	0.20	0.20	0.55	−0.05	−0.20	−0.85	−1.00	0.55	0.70
User No.	3	5	5	2	1	1	5	1	4	4

Tabelle 7.3: Absolvierte und evaluierte Aufgaben von Hans Gruber.

Iteration	1	2	3	4	5	6	7	8	9	10
Task	10	10	10	9	8	7	10	6	8	10
User-Eval	−0.10	0.40	−0.20	−0.55	0.85	−0.05	0.20	−0.25	0.05	−0.65
Task-Eval	0.25	−0.70	1.00	0.95	0.35	−0.35	−0.50	−0.65	0.85	−0.65
User No.	5	3	1	4	3	5	1	4	2	5

Tabelle 7.4: Absolvierte und evaluierte Aufgaben von Ernst Blofeld.

Im Fall dieses Tests macht es am meisten Sinn, einen Blick auf die Auswirkungen der Evaluierung (für die Funktionsweise siehe Kapitel 5) zu werfen. Zunächst ist hier schön sichtbar in allen Abbildungen zu erkennen, wann neue Kompetenzen gelernt und andere Benutzer kennen gelernt wurden. In den Abbildungen 7.7 und 7.10 kann außerdem die stetige Steigerung in der *Proficiency* bereits gelernter Kompetenzen beim Abschluss verschiedener Aufgaben beobachtet werden. Schlussendlich kann der Anstieg und Abfall der Kurven in den Abbildungen 7.6, 7.8, 7.9 und 7.11, welche alle das *Like-Level* entweder zu anderen Usern oder Kompetenzen betreffen, genau mit den positiven und negativen Evaluierungen der beiden Freiwilligen in Verbindung gebracht werden.

7.3 Mögliche Adaptionen

Zusammenfassend kann – basierend auf den Ergebnissen – gesagt werden, dass der CrAc-Core seine Aufgabe, nämlich das Bereitstellen eines sich dy-

[3] *User No.* bezeichnet die Nummer in der Reihenfolge, in der die User ihre Aufgabe annehmen, dies ist relevant für den *User-Relation-Filter*.

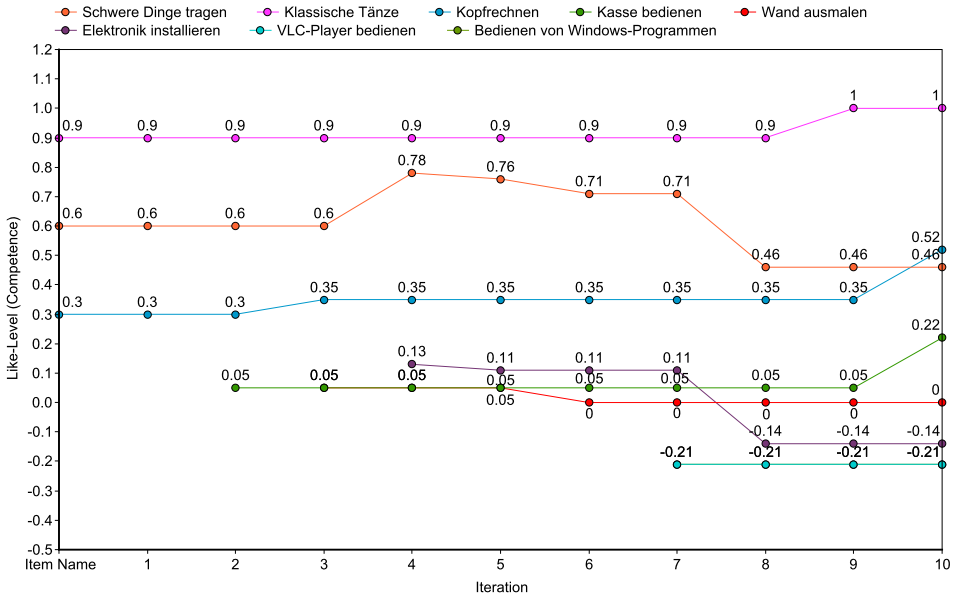


Abbildung 7.6: Veränderung des Competence-Like-Levels von Benutzer „Hans Gruber“.

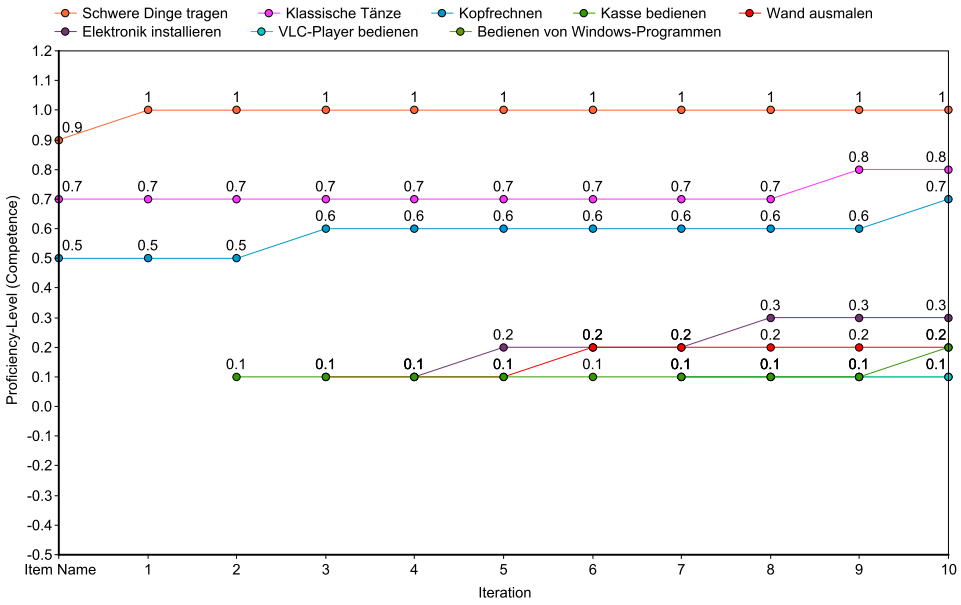


Abbildung 7.7: Veränderung des Competence-Proficiency-Levels von Benutzer „Hans Gruber“.

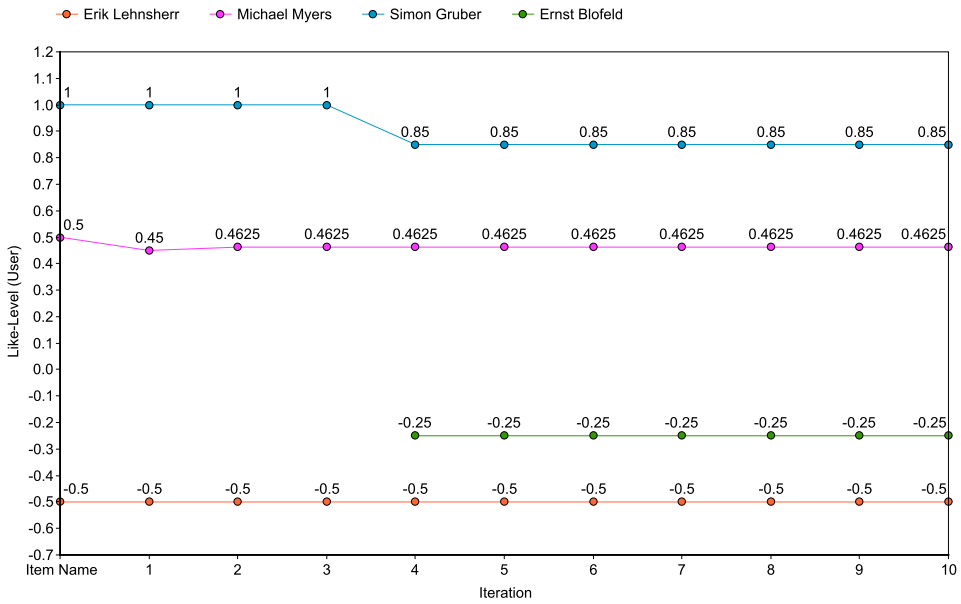


Abbildung 7.8: Veränderung des User-Like-Levels von Benutzer „Hans Gruber“.

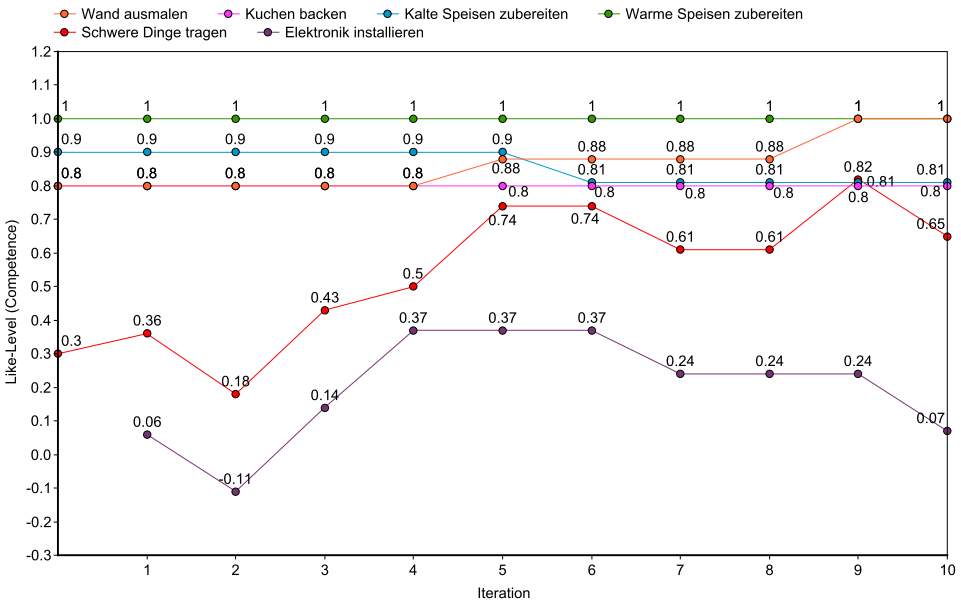


Abbildung 7.9: Veränderung des Competence-Like-Levels von Benutzer „Ernst Blofeld“.

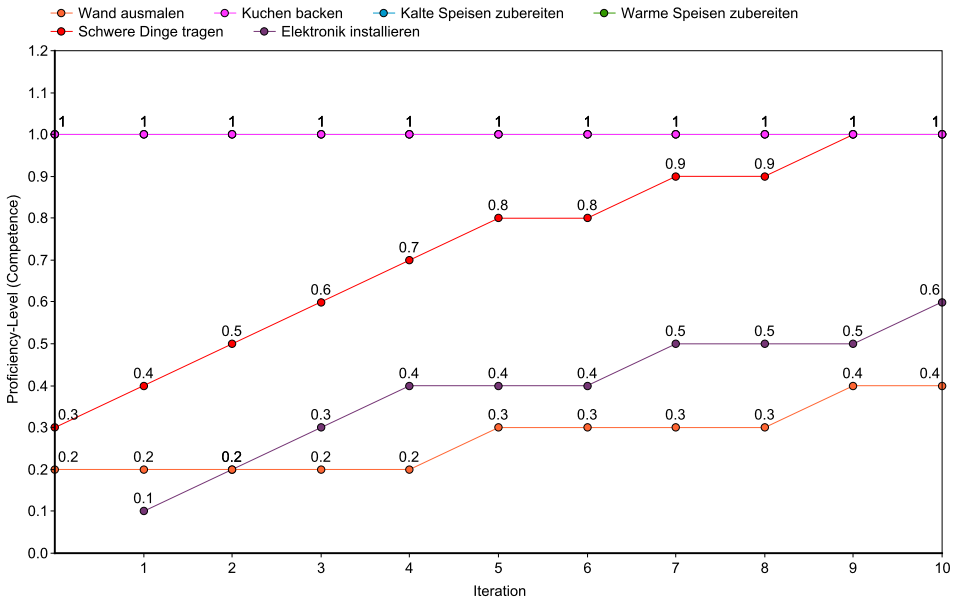


Abbildung 7.10: Veränderung des Competence-Proficiency-Levels von Benutzer „Ernst Blofeld“.

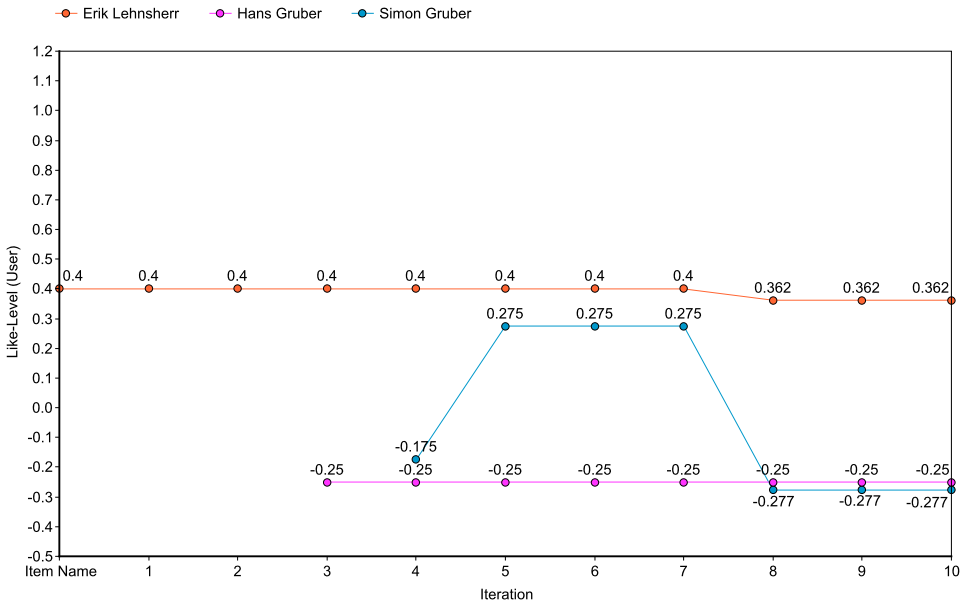


Abbildung 7.11: Veränderung des User-Like-Levels von Benutzer „Ernst Blofeld“.

namisch weiterentwickelnden Matchings mithilfe des *Matching-Evaluation-Loops*, erfüllt. Die Matching-Filter modifizieren die Meta-Daten auf die gewünschte Weise und die Funktionen in der Evaluierungsphase entwickeln sie auf die geplante Weise weiter. Trotz der Erfüllung dieser Ziele sind während der Tests einige Punkte aufgetaucht, die möglicherweise an einem gewissen Punkt Adaptionen an verschiedenen Stellen im System erfordern könnten. Auf diese Punkte wird nachfolgend eingegangen. Hierbei ist anzumerken, dass die in diesem Abschnitt genannten Punkte in der für die Thesis verwendeten Version des CrAc-Cores nicht umgesetzt sind, da sie teils größere interne Umstellungen erfordern. Ihre Umsetzung ist für die Funktionsfähigkeit des CrAc-Core außerdem *nicht notwendig*, sie könnte allerdings zur weiteren *Optimierung* der Ergebnisse beitragen.

Obergrenze der Werte im Matching

Der erste Punkt betrifft das Anpassen der Werte bei einer Überschreitung des Grenzwertes 1. Während ein festgelegter Wertebereich, insbesondere beim Persistieren der Daten, Sinn macht, verliert man beim Matching wertvolle Informationen. Bei zwei unterschiedlichen Aufgaben, welche beide einen höheren *Matching-Score* als 1 erhalten haben, sollte die Aufgabe mit dem höheren *potenziellen Score* (beispielsweise 1.5) eigentlich über der Aufgabe mit dem niedrigeren *potenziellen Score* (beispielsweise 1.3) gereiht werden. Dies kann nicht passieren, wenn beide auf den Wert 1 reduziert werden. Somit wäre ein *Überschreiten der Obergrenze* im Matching möglicherweise sinnvoll – jedoch nur mit Anpassungen der Filter.

Umstellung des Modifikation-Systems

Ein weiterer Punkt ist die direkte Modifikation der Matrix-Felder durch die Matrix-Filter. Dies führt schnell zu sehr großen oder sehr kleinen Werten, da die Filter bereits modifizierte Felder weiter verändern. Würde stattdessen jedes Mal der *Basis-Score* zur Berechnung verwendet, so könnten die einzelnen Filter-Ergebnisse *im Nachhinein kombiniert* und ein Mittelwert erreicht werden. Dies würde auch die Reihenfolge, in der die Filter in der Konfiguration liegen, für das Matching irrelevant machen.

Umstellung des User-Like-Filters

Während der Filter an sich funktioniert, so muss die Implementierung als Matching-Filter hinterfragt werden. Da der Filter nichts mit den einzelnen Kompetenzen zu tun hat, muss er nicht auf dieser Ebene des Matchings arbeiten und könnte stattdessen einfach im Nachhinein den *finalen Score modifizieren*. Dies würde im Gegensatz zur aktuellen Umsetzung auch bei Aufgaben ohne zugewiesene Kompetenzen funktionieren.

Einbezug von Ortsdaten ins Matching

In die gleiche Kategorie, nämlich das nachträgliche Anpassen des Matching-Scores, könnte auch ein *Orts-basierter Filter* fallen. Speziell durch die bereits im Projekt existierende Funktionalität von *PostGIS* könnten Aufgaben, welche in der Nähe des jeweiligen Benutzers liegen, eine bessere Reihung im Matching erhalten.

Abflachung des Like-Level-Filters

Bei Scores, die in die Nähe von Grenzwerten kommen, flacht die Funktion, die zur Berechnung im *Like-Level-Filter* verwendet wird, ab. Dies führt zwar dazu, dass der Grenzwert nie überschritten wird, macht ihn in gewissen Szenarien jedoch auch *irrelevant*. Besonders wenn ein Benutzer eine stark negative Einstellung gegenüber einer Kompetenz hat, wird dies bei einem hohen Base-Matching kaum berücksichtigt. Die Umstellung dieser Berechnung ist insbesondere interessant in Verbindung mit der bereits besprochenen Aufhebung einer Werte-Obergrenze im Matching.

Senkung des Proficiency-Levels

Ein letzter Punkt wäre die *Absenkung des Proficiency-Levels* des Benutzern in Bezug auf verschiedene Kompetenzen. Auf diese Weise könnte bei fehlender Praxis ein langsames Verlernen simuliert werden. Allerdings wäre ein Absenken nur bei gewissen Kompetenzen sinnvoll und sollte zusammen mit der Zeiteinheit, nach deren Ablauf das Absenken stattfindet, in der Kompetenz selbst konfiguriert werden. Auch der potenzielle Frust-Faktor bei betroffenen Freiwilligen sollte bei einem automatisierten Verlernen nicht vernachlässigt werden.

7.4 Erfüllte Anforderungen

Nachdem die einzelnen Module und Klassen, sowie die Tests am Gesamtsystem nun dargelegt sind, wird erneut auf die in Abschnitt 1.3.1 gestellten Anforderungen eingegangen und erörtert, ob und wie sie erfüllt wurden.

Datenpersistenz

Die erste Anforderung wird durch die Verwendung verschiedener Mechanismen und *Libraries* erfüllt. Darunter fallen die verwendete Datenbank *PostgreSQL*, sowie die Datenbank-Schnittstelle *JPA* und das involvierte Datenmapping via *Hibernate*. Für einen genaueren Blick auf die verwendeten Technologien siehe Abschnitt 2.1.1.

Such- und Empfehlungsmechanismen

Eine der wichtigsten Anforderungen, nämlich der *Matching-Mechanismus*, wird als Teil des *Matching-Evaluation-Loops* (s. Kapitel 4) realisiert und kombiniert als Bestandteil verschiedener Module und Klassen unterschiedliche Konzepte. Die darauf basierenden Tests dieses Kapitels bestätigen eine Erfüllung dieser Anforderung.

Entwicklungsmechanismen

Ähnlich dem vorherigen Punkt sind auch die *Entwicklungsmechanismen* (s. Kapitel 5) als Teil des *Matching-Evaluation-Loops* und verschiedener Module und Klassen implementiert. Auch in diesem Punkt bestätigen die Tests eine Erfüllung der Anforderung.

Unabhängigkeit von eigenem Frontend

Die Unabhängigkeit vom Frontend wird durch das Abkoppeln des CrAc-Cores von einer Frontend-orientierten Datenausgabe erreicht. Stattdessen wird eine eigene *API* verwendet, die zwar einem CrAc-spezifischen Frontend vollen Zugriff auf alle Funktionen des Cores erlaubt, jedoch auch von jeder anderen Plattform, unabhängig von der verwendeten Technologie, integriert werden kann (s. dazu Abschnitt 6.1). Diese API verwendet das *REST-Prinzip* und kommuniziert via *JSON*.

Unabhängigkeit von Prototyp-Instanzen

Dieser Punkt wird bedingt durch die *Architektur* (s. Kapitel 6) des CrAc-Cores erfüllt. Jede Instanz hat nur die Möglichkeit, auf ihre eigenen internen Module und externen Komponenten zuzugreifen, eine Schnittstelle zum direkten Austausch zwischen verschiedenen Cores wird (bis auf eine Ausnahme) nicht angeboten.

Austausch von gemeinsamen Daten

Diese Ausnahme bildet der Austausch gemeinsamer, nicht-sensibler Daten, sprich *Kompetenzen*. Umgesetzt ist dies durch Verwendung der Software *KOMET*, die dem CrAc-Core als Postkastensystem für Kompetenzen dient (s. Abschnitt 6.1).

Code-Wiederverwendbarkeit

Die Erweiterbarkeit und Wiederverwendbarkeit des eigentlichen *Core-Programmcodes* wird durch einen modularen Aufbau, eine eindeutige Gliederung von Funktionalität, sowie die Erweiterbarkeit verschiedener Klassen und Module gewährleistet.

Keine Monetarisierungsmöglichkeiten

Diese Anforderung wird erfüllt, da im gesamten CrAc-Core Rücksicht darauf genommen wird, keine monetäre Entlohnung irgendeiner Art möglich zu machen. Dies reicht von den verwendeten *Open-Source-Technologien* über die Art des verwendeten *Matching-Verfahrens* bis zu den Hauptklassen, die das Anlegen einer solchen Entlohnung nicht zulassen.

Kapitel 8

Schlussbemerkungen

Abschließend soll noch einmal auf die verschiedenen Kapitel dieser Arbeit eingegangen werden. Auch die anfangs gestellten Anforderungen, sowie die mögliche weitere Entwicklung des *Matching-Frameworks* werden angesprochen.

Diese Thesis hat sich mit dem prototypischen Matching-Framework *CrAc-Core* auseinandergesetzt, welches eine Zuordnung von Aufgaben an freiwillige Helfer, basierend auf deren Kompetenzen und Skills, ermöglichen soll. Hierbei wurde zunächst auf den Bereich *Volunteering* eingegangen und basierend auf einer Markt-Recherche betreffend ähnlicher Volunteering-Plattformen die Frage geklärt, welche Funktionalität ein solches Framework bieten muss. Basierend auf diesen Ergebnissen wurden schließlich bindende Anforderungen an den CrAc-Core gestellt.

Als nächster Schritt dieser Arbeit wurden die Klassen des Frameworks, allen voran die für das Matching notwendigen Hauptdatentypen *User*, *Competence* und *Task* analysiert. Hierbei wurde nicht nur näher auf die Klassen an sich eingegangen, sondern auch auf deren Beziehungsdatentypen, die als Speicher für eine Reihe an Matching-relevanten Meta-Daten dienen. Auch auf die internen Verbindungen von Objekten solcher Typen und den damit einhergehenden Workflows (z.B. interner Task-Workflow) wurde dabei geachtet.

Ebenso wurde ein Einblick in den Kreislauf aus Matching, Bearbeitung der Aufgabe, Evaluierung und Anpassung der gerade angesprochenen Meta-Daten gegeben, der das Herzstück des gesamten Systems bildet und für eine dynamische Zuordnung von Task und User, auf Basis von sich immer weiter entwickelnden Profilen, sorgt. An dieser Stelle wurde auch auf weitere Matching-relevante Klassen eingegangen, wie beispielsweise die *Matching-Matrix*, welche die eigentlichen Berechnungen durchführt, oder die *Matching-Filter*, die genau diese Matrix modifizieren.

Sowohl *externe Komponenten*, als auch *interne Module* des CrAc-Core wurden auf Basis ihres Aufbaus und ihrer Funktionsweise analysiert und ihre

Rollen im Gesamtsystem inklusive den Verknüpfungen zu anderen Teilen des CrAc-Cores wurden grafisch und textuell dargestellt.

Verschiedene Tests am System zeigen außerdem sowohl die Auswirkungen aktivierter *Matching-Filter* im Suchprozess als auch die Veränderung bestehender Meta-Daten auf Basis des angesprochenen *Matching-Evaluation-Loops*.

Letztendlich kann aufgrund der implementierten Komponenten, Klassen und verwendeten Konzepte von einer Erfüllung aller gestellten Anforderungen gesprochen werden. Die dahingehend durchgeführten Tests bestätigen dies, zeigen jedoch ebenfalls auf, dass nach wie vor Potenzial in der Weiterentwicklung dieses Prototyps liegt und Adaptionen bereits funktionierender Mechanismen die Ergebnissen weiter verbessern können. Obwohl also die grundsätzlichen Anforderungen erfüllt sind, sollte der CrAc-Core weiterhin entwickelt und neu auftauchenden Anforderungen angepasst werden.

Anhang A

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Dateien

Pfad: /

Thesis_Hondl.pdf . . .	PDF-Version dieser Thesis

A.2 Projekt-Dateien

Es folgt die Dateistruktur des Projekts *CrAc-Core*:

Pfad: /crac-core

pom.xml	Enthält die Abhängigkeiten zu externen Libraries.
README.md	Enthält Informationen zu Projekt und Endpoints.
.gitignore	Enthält die von Git zu ignorierenden Dateinamen und Ordner.

Pfad: /crac-core/src/main/resources

application.properties .	Enthält grundlegende Einstellungen der Spring-Applikation.
import.sql	Enthält SQL-Anweisungen, die beim Aufsetzen der Datenbank ausgeführt werden.

Pfad: /crac-core/src/main/java/crac

*.java . . . . . Dateien enthalten die Hauptkonfiguration der Applikation.

Pfad: /crac-core/src/main/java/crac/components/matching

*.java . . . . . Dateien enthalten die Hauptklassen für den Matching-Prozess.

Pfad: /crac-core/src/main/java/crac/components/matching/configuration

*.java . . . . . Dateien enthalten die Klassen für die Konfiguration des Matching-Prozesses.

Pfad: /crac-core/src/main/java/crac/components/matching/filter

*.java . . . . . Dateien enthalten die Filter-Klassen.

Pfad: /crac-core/src/main/java/crac/components/matching/filter

*.java . . . . . Dateien enthalten die Worker-Klassen.

Pfad: /crac-core/src/main/java/crac/components/notifier

*.java . . . . . Dateien enthalten die Klassen des Notification-Systems.

Pfad: /crac-core/src/main/java/crac/components/notifier/notifications

*.java . . . . . Dateien enthalten die Notification-Klassen.

Pfad: /crac-core/src/main/java/crac/components/storage

*.java . . . . . Dateien enthalten die Cache-Klassen.

Pfad: /crac-core/src/main/java/crac/components/utility

*.java . . . . . Dateien enthalten Klassen mit zusätzlicher Funktionalität.

Pfad: /crac-core/src/main/java/crac/controllers

*.java . . . . . Dateien enthalten die Controller-Klassen.

Pfad: /crac-core/src/main/java/crac/enums

*.java . . . . . Dateien enthalten die enum-Klassen.

Pfad: /crac-core/src/main/java/crac/filters

*.java . . . . . Dateien enthalten die Spring-Filter.

Pfad: /crac-core/src/main/java/crac/models/db

*.java . . . . . Dateien enthalten die Klassen, die mittels Hibernate in die CrAc-Core-Datenbank persistiert werden.

Pfad: /crac-core/src/main/java/crac/models/komet

*.java . . . . . Dateien enthalten die Klassen, die mittels Hibernate in die KOMET-Datenbank persistiert werden.

Pfad: /crac-core/src/main/java/crac/models/input

*.java . . . . . Dateien enthalten spezielle Input-Klassen für die DB-Models.

Pfad: /crac-core/src/main/java/crac/models/output

*.java . . . . . Dateien enthalten spezielle Output-Klassen für die DB-Models.

Pfad: /crac-core/src/main/java/crac/models/storage

*.java . . . . . Dateien enthalten Klassen, die für Storage relevant sind.

Pfad: /crac-core/src/main/java/crac/models/utility

*.java . . . . . Dateien enthalten Klassen mit zusätzlicher Funktionalität.

A.3 LaTeX-Projekt

Pfad: /tex/HagenbergThesis-master/

. . . . . . Dateien des LaTeX-Projekts.

A.4 Bilder

Pfad: /tex/HagenbergThesis-master/images

*.pdf . . . . . Alle Bilder der Thesis.

A.5 Online-Literatur

Pfad: /literature

competenciesRecommendation.pdf	Chuck Allen. Competencies (Measurable Characteristics) Recommendation.
xml.pdf	W3C World Wide Web Consortium. Extensible Markup Language (XML).
cracAt.pdf	CrAc - Cooperative Activities.
searchEngineRank.pdf .	DB-Engines. DB-Engines Ranking of Search Engines.
jsonStandard.pdf	Ecma International. The JSON Data Interchange Format.
springGuides.pdf	Spring by Pivotal. Spring Guides.
volunteerismReport.pdf	SWVR Report Team. State of the World's Volunteerism.

A.6 Sonstiges

Pfad: /

Endpoints.pdf	Liste der API-Endpoints
Tests.xls	Sämtliche Daten aus den Tests

Quellenverzeichnis

Literatur

[1] Chuck Allen. *Competencies (Measurable Characteristics) Recommendation*. Feb. 2004. URL: http://xml.coverpages.org/HR-XML-Competencies-1_0.pdf (siehe S. 15, 17).

[2] Vamshi Ambati, Stephan Vogel und Jaime Carbonell. „Towards Task Recommendation in Micro-Task Markets“. In: Proceedings of the 11th AAAI Conference on Human Computation. San Francisco, USA: ACM, 2011, S. 80–83 (siehe S. 31).

[3] Tim Berners-Lee, James Hendler und Ora Lassila. „The Semantic Web“. *Scientific American* 284.5 (2001), S. 28–37 (siehe S. 29).

[4] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. Irvine: University of California, Department of Computer Science, 2000 (siehe S. 9, 55).

[5] *Freiwilliges Engagement in Österreich, Bundesweite Bevölkerungsbefragung*. Studienbericht, Bundesministerium für Arbeit, Soziales und Konsumentenschutz, Institut für empirische Sozialforschung. 2013. URL: http://www.freiwilligenweb.at/sites/default/files/fwe_in_oe_-_bundesweite_bevoekerungsbefragung_2012.pdf (siehe S. 2).

[6] Thomas R. Gruber. „A Translation Approach to Portable Ontology Specifications“. *Knowledge Acquisition* 5.2 (1993), S. 199–220 (siehe S. 29).

[7] Fuad Mire Hassan u. a. „Ontology Matching Approaches for eRecruitment“. *International Journal of Computer Applications* 51 (2012), S. 39–45 (siehe S. 30).

[8] Hexin Lv und Bin Zhu. „Skill ontology-based semantic model and its matching algorithm“. In: Proceedings of the 7th International Conference on Computer-Aided Industrial Design and Conceptual Design. Hangzhou, China: IEEE, Nov. 2006, S. 1–4 (siehe S. 32, 35, 37, 38, 45).

[9] Mark Andrew Mitchell und Susan Taylor. „Internal marketing: key to successful volunteer programs“. *Nonprofit World* 22.1 (2004), S. 25–26 (siehe S. 2).

[10] Markus Raab. „A prototypical approach for a competency-based task allocation system in the context of voluntary organizations“. Magisterarb. Hagenberg, Austria: Upper Austria University of Applied Sciences, Kommunikation Wissen Medien, Aug. 2016, S. 39–84 (siehe S. 17, 30, 32, 40, 41, 43).

[11] Johannes Schoenboeck u. a. „A Survey on Volunteer Management Systems“. In: Proceedings of the 49th Hawaii International Conference on System Sciences (HICSS). Koloa, Hawaii, USA: IEEE, 2016, S. 767–776 (siehe S. 3).

[12] Robert Sedgewick und Kevin Wayne. *Algorithms*. 4. Aufl. Reading, MA: Addison-Wesley, 2011 (siehe S. 35).

[13] Man-Ching Yuen, Irwin King und Kwong-Sak Leung. „Task Recommendation in Crowdsourcing Systems“. In: Proceedings of the First International Workshop on Crowdsourcing and Data Mining. Beijing, China: ACM, 2012, S. 22–26 (siehe S. 31).

Online-Quellen

[14] *CrAc - Cooperative Activities*. URL: http://www.crac.at/crac-cooperative-activities-1 (besucht am 23. 06. 2017) (siehe S. 3).

[15] DB-Engines. *DB-Engines Ranking of Search Engines*. URL: http://db-engines.com/en/ranking/search+engine (besucht am 23. 06. 2017) (siehe S. 30, 57).

[16] *Extensible Markup Language (XML)*. URL: https://www.w3.org/XML/ (besucht am 23. 06. 2017) (siehe S. 56).

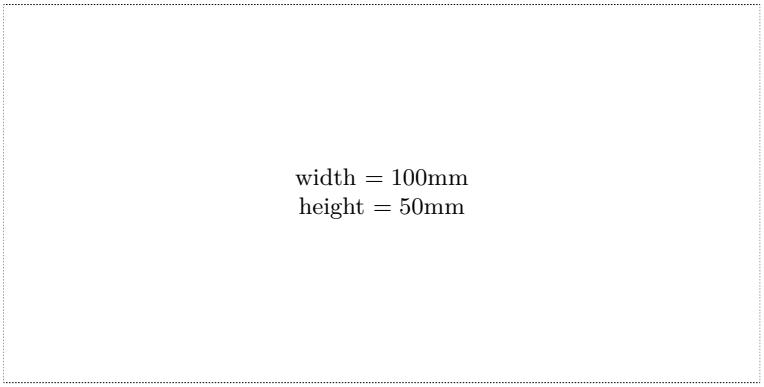
[17] *Spring Guides*. URL: https://spring.io/guides (besucht am 23. 06. 2017) (siehe S. 7).

[18] SWVR Report Team. *State of the World's Volunteerism*. 2011. URL: https://www.unv.org/sites/default/files/2011%20State%20of%20the%20World%27s%20Volunteerism%20Report%20-%20Universal%20Values%20for%20Global%20Well-being.pdf (siehe S. 2).

[19] *The JSON Data Interchange Format*. 2013. URL: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (besucht am 23. 06. 2017) (siehe S. 9).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —