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

David Hondl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Master Thesis

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

Preface

Abstract

This master thesis presents and evaluates the prototype of a proposed open-source framework, with which the allocation of an amount of tasks to volunteers, based on qualitative criteria like competences and the user's skill-set, is possible.
It focuses on explaining why things have been done the way they ended up in the project and discusses the different approaches used.
It also elucidates why different other methods have been discarded along the creation of the framework.
Every aspect of the project will be dissected by - and presented in - the thesis.
In the end, the thesis should give an overview over the framework and clarify, if the presented prototype fits all the prerequisites this complex field of work imposes.

Kurzfassung

In Arbeit

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 Möglichkeiten 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. 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[3].
An dieser Stelle muss nachdrücklich betont werden, dass der hier vorgestellte Prototyp zwingend die Werte dieser riesigen Gemeinschaft miteinbezieht, um seine Ziele für sie und mit ihr zu erreichen.

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 finanziert wird. Das Ziel dieses Projektes ist die Erstellung einer webbasierten Software, die die organisatorische Arbeit in gemeinnützigen Vereinen 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 voraus zu sein, führten Mitarbeiter von CrAc vor der konkreten Entwicklung einer Software eine Markt-Recherche über verschiedene etablierte Produkte durch. Das daraus resultierende Paper [9] gibt einen Überblick über den Markt und zeigt außerdem auf, warum verschiedene Produkte in verschiedenen Aspekten der Zielgruppe nicht gerecht werden. Daraus lassen sich Anforderungen ableiten, 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 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 Der Prototyp, "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 Name: *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 Kapitel 1.2 abgeleitete und zusätzlich hinzugefügte Ansprüche müssen an den Prototypen gestellt werden, um der Zielgruppe und den Benutzern gerecht zu werden.

Datenpersistenz

Der Prototyp muss das Validieren und Persistieren von Personen- und Aufgabenbezogenen 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 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.

Einfach zu verstehende Arbeitsabläufe

Da der Prototyp nur das Backend für ein zugehöriges Web-Interface darstellt, muss er sich nicht direkt um die finale Ausgabe der Daten kümmern. Die unterliegenden Konzepte, die die Arbeitsabläufe innerhalb des CrAc-Cores beschreiben, müssen es allerdings dem Frontend und dadurch letztendlich dem Benutzer, einer - in den meisten Fällen - technisch nicht affinen Person, so einfach wie möglich machen, komplexe Schritte einfach zu bewältigen.

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.

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 jeder Komponente des Prototypen 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 Kapiteln Bezug genommen wird, stellt hierbei bereits eine teilweise Umsetzung der *Anforderungen (Kapitel 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.

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* ein 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. Allgemein bekannte, einfach zu lesende und verstehende Programmier -/ oder Skriptsprache (Anforderung 1.3.1)
2. Framework und Sprache müssen Objekt-Orientierung forcieren (Anforderung 1.3.1)
3. Eine ausreichende Zahl an zuverlässigen *Libraries* für verschiedenste Anwendungsfälle
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 (eg. Security-Aspekt) verfügen
7. Brauchbare Umsetzungen für ausgewählte und zu testende Matching-Konzepte müssen in der gewählten Sprache existieren

Basierend darauf wurden als Programmiersprache *Java* und als Framework *Java Spring* ausgewählt. Java erfüllt die Anforderungen nicht nur hinsichtlich der Fokussierung auf Objekt-Orientierung und einfach strukturiertem Code, sondern bietet auch eine unglaubliche 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 (Anforderung 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*[10] kann auf diese Weise eine einfache, transparente Projekt-Konfiguration vorgenommen werden. 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 Basis-Technologie kann nun auf einige recht relevante Sub-Punkte eingegangen werden.

2.1.1 Datenbank

Genutzt wird die relationale Datenbank *Postgres*. Aufgrund der klar definierten, gleichbleibenden Struktur der Daten und deren Beziehungstypen,

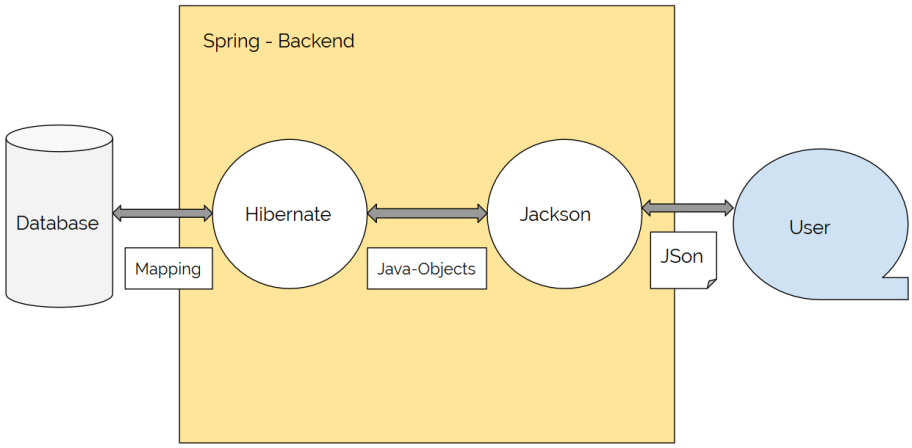


Abbildung 2.1: Daten-Mapping im Framework.

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* (erklärt in Kapitel 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 erste 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 Grafik 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 Übertragungsformat nennt sich *JSON* 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 Kapitel 2.1.2 angesprochen JSON. Die Schnittstellen des Prototypen verwenden das *REST-Paradigma*[7], 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*[2], als Build-Management-Tool, insbesondere um auf einfache Weise Libraries in den Core einzubinden
- *Elasticsearch*[6], um Volltextsuche innerhalb des Cores zu ermöglichen – mehr dazu in Kapitel 6.1

2.2 Engere Auswahl

Anbei eine Auswahl an Systemen, die es in die nähere Auswahl als Basistechnologie von CrAc-Core schafften.

Die Plattform Moodle, PHP als Skriptsprache

Anfangs schien Moodle ein geeigneter Kandidat für den Core zu sein, vor allem durch die Menge an fertigen Features, die in der Initial-Installation bereits enthalten waren. Diese umfassten:

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

Es stellte sich jedoch heraus, dass die Abgeschlossenheit der Plattform und der einzelnen Module nicht nur als Vorteil zu sehen ist, sondern bei der Einbindung von verschiedenen Konzepten hinderlich sein kann. Folgende Gründe führten schließlich zu einer Verwerfung von Moodle:

1. 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. Schwieriges Entkoppeln von Frontend und Backend

Kapitel 3

Komplexe Datentypen und deren Klassen

Nachdem die technische Basis in ihrer Gesamtheit 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-Komponenten* dienen, und sämtlichen anderen. Um diese Datentypen, die keine kompletten *Komponenten* darstellen, geht es in diesem Kapitel. Auf ihnen bauen alle Komponenten 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-Komponenten 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 werden 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

Informationen, die für erfolgreiche Identifizierung benötigt werden.

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

Die *id* wird System-intern für die Identifizierung des User-Objektes genutzt. Die Kombination von *name* und *password* hingegen dient als Identifikation von außen, um als Benutzer *Endpoints* des Systems aufzurufen.

Persönliche Informationen

Daten, die zur Darstellung des jeweiligen Nutzers genutzt werden.

```
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 System

Jeder User kann eine beliebige Menge an Rollen im System zugewiesen bekommen, wobei jede Rolle über ID und Namen (z.B. USER oder ADMIN) definiert ist. Mittels einer einfachen Abfrage dieser Rollen ist es *Spring* möglich, den Zugriff jedes Users 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.

```
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. Dies ermöglicht die Auftrennung von Aufgaben in Sub-Aufgaben in Form einer Baumstruktur, abgebildet von der übergeordneten *superTask* und einer Menge aus untergeordneten *childTasks*. 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.

- *Organisational:* Struktur-definierende Task, die zur Gliederung des Baumes dient
  - Erweiterung mit weiteren Organisational-Tasks möglich
  - Erweiterung mit Workable-Tasks möglich
- *Workable:* Unterste Ebene des Baumes und damit feinst-granulare Gliederung der Aufgabe
  - Erweiterung mit Shift-Tasks möglich
- *Shift:* Kann eine weitere Pseudo-Ebene unter *Workable* bilden, um diese in unterschiedliche zeitliche Schichten aufzuteilen

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

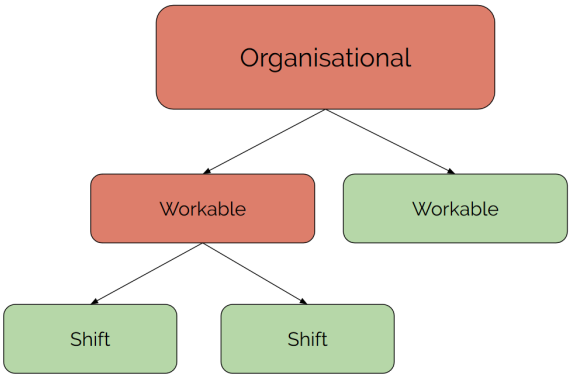


Abbildung 3.1: Eine Gesamtaufgabe als Baum dargestellt.

In Grafik 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. Da hier zwischen *Workable* und *Organisational* zu unterscheiden ist, müsste ohne einen festgelegten Typ jedes Mal auf Child-Tasks geprüft werden, was auch die nachträgliche Modifikation der Baum-Struktur verkomplizieren würde. Eben dieser Status einer Aufgabe wird im Attribut *taskState* persistiert. Es kann vier verschiedene Zustände einnehmen, die ineinander übergehen, um so dem System den Fortschritt innerhalb der Aufgabe mitteilen zu können.

- *Not_Published* beschreibt 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 die Flag *readyToPublish* ein, deren Bedingungen erfüllt werden müssen, um eine Aufgabe von *Not_Published* auf *Published* setzen zu können.

```
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 Kapitel 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 (eg. *Matching*) möglich gemacht. Weiters handelt es sich bei diesem Datentyp nicht um einen komplett selbst entwickelten, der sich an den Mindestanforderungen des ihn umgebenden Systems hält (wie Task und User), 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.

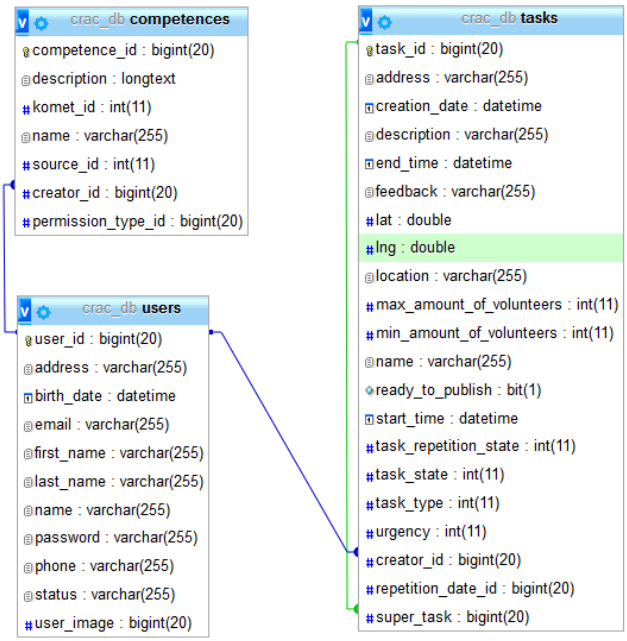


Abbildung 3.2: Die Hauptdatentypen

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

Der eigentliche Aufbau der *Kompetenz-Klasse* ist, wie hier zu sehen, recht primitiv und gewinnt erst mit den in der nächsten Sektion vorgestellten *Beziehungsdatentypen* an Tiefe. Hier offenbart sich auch das Potential hinter der *Vernetzung sämtlicher Hauptdatentypen*. Davor noch ein kurzer Blick auf das Modell der Hauptdatentypen in Grafik 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 Art von Kategorie. 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 Komponenten verarbeitet werden. Die Speicherung erfolgt auch hier wieder via *Hibernate* in die angeschlossene Postgres-Datenbank.

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 einige Meta-Daten persistiert.

```
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 der HR-XML-Recommendation 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.

```
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*
  - Beschreibt keine tatsächliche Teilnahme, sondern eine *Interessenbekundung*
  - Der User signalisiert mit dem *Folgen* eine mögliche Teilnahme an einer konkreten Aufgabe oder deren Sub-Aufgaben
  - Aufgaben jedes TaskTypes können ”gefollowed” werden
  - Erst möglich, wenn eine Aufgabe veröffentlicht, also ”gepublished” wurde, siehe Kapitel 3.1.2
- *Participating*
  - Beschreibt die *aktive, freiwillige Teilnahme* eines Users an einer Aufgabe
  - Nur an einer *Workable-Task* oder einer *Shift* kann ”participated” werden
  - Ein Benutzer kann an einer Aufgabe teilnehmen, wenn sein *Matching-Score* (siehe Kapitel 4) einem Wert über null entspricht
  - Ebenfalls erst möglich, wenn eine Aufgabe veröffentlicht wurde
- *Leading*
  - Beschreibt eine *führende* Tätigkeit innerhalb des Aufgaben-Baumes
  - 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
- In Grafik 3.3 ist ein Aufgaben-Baum mit verschiedenen verbundenen Usern abgebildet

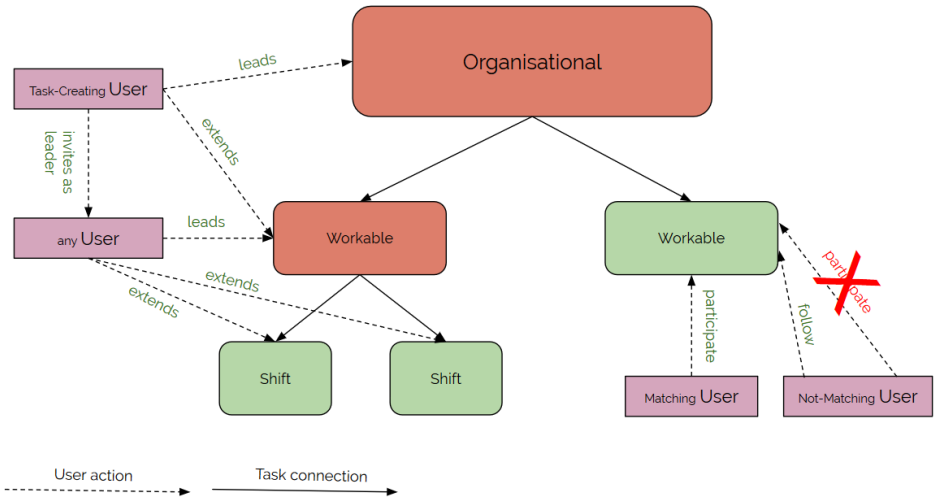


Abbildung 3.3: Eine Gesamtaufgabe mit verbundenen Benutzern als Baum dargestellt.

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 Kapitel 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 Repräsentiert die Verwandtschaft zwischen zwei beliebigen Kompetenzen, eine Information, die für das Matching in Kapitel 4 unverzichtbar ist.

```
1  private CompetenceRelationshipType type;
```

Diese enthält die Referenz zu einem *CompetenceRelationshipType*, welcher wiederum die tatsächlichen Beziehungsdaten speichert.

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

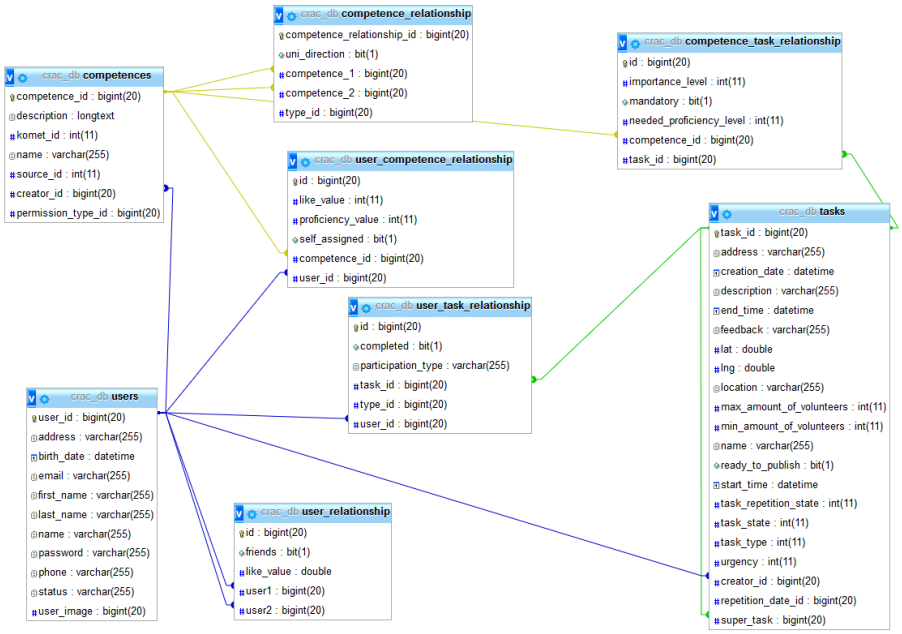


Abbildung 3.4: Die via Beziehungsdatentypen verbundenen Hauptdatentypen

Durch die Auslagerung vereinheitlichter Kompetenz-Verbindungstypen wird 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 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. Grafik 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 Komponenten 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 genutzt. 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.

```
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){}
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 (erklärt in Kapitel 4), sowie dessen Anpassungsmechanik (in 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 und überschreiben die 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 *ID* 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. Den Code in Worker aufzuteilen macht den Core daher besser skalierbar und trägt zur - in 1.3.1 angeforderten - Code-Wiederverwendbarkeit 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 in einem kompletten Matching-Prozess eine Menge von $Usern \times Tasks$ an Matrizzen erstellt wird, die danach wieder automatisch zerstört werden. Der Datentyp enthält nicht nur die Daten-Matrix, sondern implementiert auch verschiedene Verfahren, mit denen aus der erstellten Matrix ein einzelner Vergleichswert extrahiert werden kann. Weiters verfügt die Klasse über eine Option zur detaillierten Ausgabe.

Der Einfluss von Anpassungen der mathematischen Modelle des Matching-Prozesses lässt sich auf diese Art einfacher visualisieren.

```
1 private MatrixField[][] matrix;
2 private CracUser u;
3 private Task t;
4 private boolean doable;
5 private ArrayList<String> mandatoryViolations = new ArrayList<>();
6 private Set<UserCompetenceRel> userCompetences;
7 private Set<CompetenceTaskRel> taskCompetences;
```

Die Klasse speichert sämtliche Daten des verknüpften Users und der Task in verschiedenen Attributen.

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

Diese Information wird genutzt, um Matrix zu erzeugen (*buildMatrix()*), sie zu validieren (*markMandatoryViolation()*) und zu modifizieren (*apply-Filters()*).

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

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 im Kapitel 4 zu finden.

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 Methode die Werte der Matching-Matrix nachträglich justieren. Die Art der Modifikation hängt rein vom Filter 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 *extended* diese Klasse 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.

```
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 Namen und einen Typ für die Identifikation, sowie die ID ihres Ziel-Users. Da es sich im Code-Snippet um eine abstrakte Klasse handelt, welche die Eigenschaften lediglich weitervererbt, können die tatsächlich implementierten Notifications beliebige weitere Attribute besitzen. Jede Notification 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 Anforderung 1.3.1 Genüge getan, der sich mit dem Datenaustausch auseinandersetzt.

```
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": 116,
8      "name": "1 Blech Saures",
9      "description": "z.B. Schinkenkipferl (oder Käse) ... Pizza?",
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.

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. Das spezielle an den Post-Options ist, dass sie komplett beliebig erweiterbar sind, jeder Endpoint nutzt nur jeweils die Attribute, die er selbst befüllt.

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 wurden bereits in früheren Kapiteln angesprochen und 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 Komponenten werden in diesem Kapitel 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* 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 (dargestellt in Grafik 4.1) eingegangen:

- Matching
  Den Startpunkt markiert der Matching-Prozess in seiner Gesamtheit, beschrieben in 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.

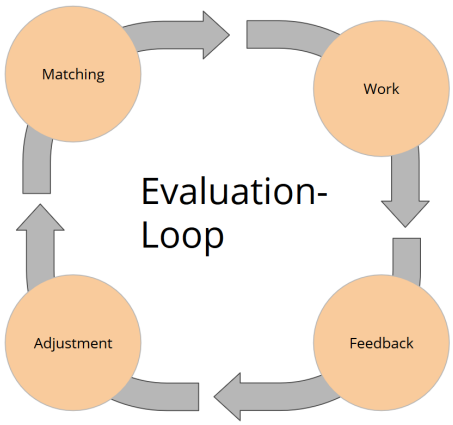


Abbildung 4.1: Der Matching - Feedback Kreisauf.

- 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 Kapitel 5.2.
- Adjustment
  Im letzten Schritt kümmert sich der Core um die Auswertung des Feedback und die Modifikation der im Matching verwendeten Daten, dies beschreibt Kapitel 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 so genannte *Onthologien*. Diese Systeme zur geordneten Datenspeicherung besitzen im Gegensatz zu herkömmlichen Datenbanken eine weitere Ebene, in der die Beziehungen zwischen 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. Onthologien sind Teil des *Semantic Web* und nicht in jeder Programmier- und Skriptsprache zu einem befriedenden Grad umgesetzt. Hier bietet Java mit *Apache Jena* eine API, die den einfachen Zugriff unterstützt. Obwohl der Einsatz einer *Onthologie* einiges an Potenzial verspricht, wurde die Idee im Zusammenhang mit dem *CrAc-Core* noch vor der eigentlichen Implementierung zugunsten anderer Ideen verworfen. Dies sind die Gründe:

- 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 *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

Diese Sektion beschäftigt sich mit der beliebten[5] 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 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 errechneten 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*.

4.2 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 einem Paper[8], welches den Aufbau einer Onthologie-basierten Datenstruktur für die strukturierte Persistierung von Skills und deren Nutzung für Meta-Datengewinnung diskutiert. Teile der mathematischen Grundlagen zur Modifikation dieses

Grundwertes lehnen sich Vorschlägen aus der Masterarbeit[12] eines weiteren Mitarbeiters des CrAc-Projektes an. Inwiefern sich die finale Umsetzung von den Ideen dieser Vorlagen unterscheidet, wird in den kommenden Sektionen dieses Kapitels 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 Ähnlichkeitsgrad zweier Kompetenzen beschreibt, gespeichert von der Kompetenz-Beziehung (erklärt in Kapitel 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 der Arbeiten[8, 12] in Form eines Kompetenz-Graphen vorliegen müssen.

4.2.1 Kompetenz-Graph

Diese auf dem Paper[8] basierende Datenstruktur, der auch ein Kapitel in der Thesis von Herrn Raab[12] gewidmet ist, bildet das Rückgrat des Kompetenz-Matching. Laut dem Paper werden in einem ungerichteten Graphen beliebige einfache Skills miteinander verknüpft. Die Skills selbst stellen die einzelnen Knoten des Graphen dar, während die Kanten von ihren Beziehungen untereinander 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 implementiert und erweitert. Anstatt einfacher *Skills* werden Objekte der *Competence-Klasse* als Knoten des Graphen genutzt, als Kanten dient deren *CompetenceRelationship*. Die Umsetzung übernimmt an diesem Punkt die Vorlage, reichert sie jedoch um die zusätzlichen Informationen beider Klassen an.
Das richtige Einpflegen von Inhalten in den Graphen ist von enormer Wichtigkeit, da die Onthologie-orientierte 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 Grafik 4.2 ist ein solcher Kompetenz-Graph abgebildet.

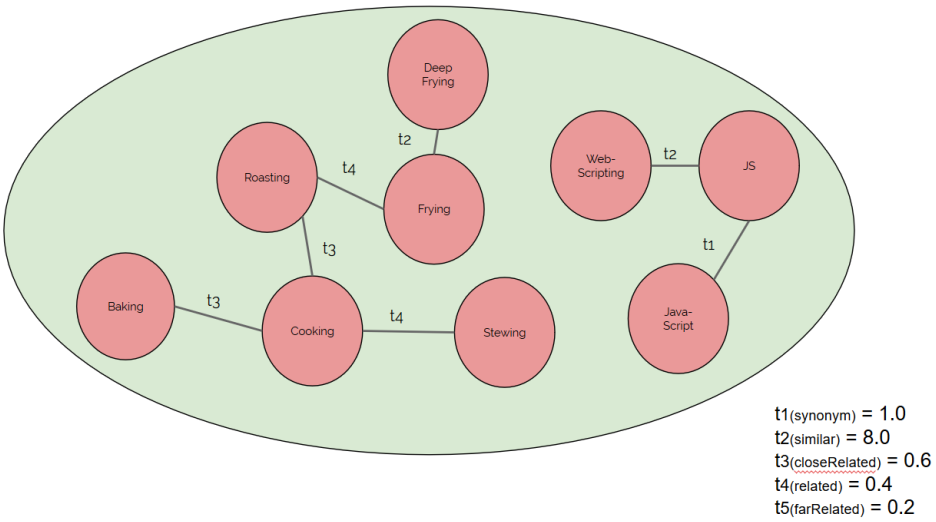


Abbildung 4.2: Beispiel für einen Kompetenz-Graphen.

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. Nachfolgend werden die einzelnen Schritte (vom Graphen bis zum Matching-Score) beschrieben.

- **Start des Prozesses (mit dem Kompetenz-Graphen)**
- Interpolation der Verwandtschaft verschiedener Kompetenzen aus dem Graphen
- Aufbau einer Matrix aus den errechneten Werten
- Modifikation der Matrix auf Basis von Meta-Daten
- Auswertung der Matrix und bilden des *Base-Matching-Score*
- Modifikation des *Base-Matching-Score* auf Basis von Meta-Daten
- **Ende des Prozesses (mit dem finalen *Matching-Score*)**

Nachfolgend werden die einzelnen Schritte, die verwendeten mathematischen Ansätze und ihre algorithmische Umsetzung näher vorgestellt.

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.

$$simVal(C_1, C_n) = \prod_{n=1}^{n} simVal(C_{n-1}, C_n)$$

Um den Verwandtschaftswert zweier beliebiger Kompetenzen zu errechnen, müssen die *Similarity-Values* sämtlicher zwischen der *Start-Kompetenz* und *Ziel-Kompetenz* liegenden Kanten multipliziert werden. Aufgrund des Wertebereiches sämtlicher Kanten muss auch jedes neu errechnete *Similarity-Value* immer zwischen 0 und 1 liegen und kann nur gleich groß oder kleiner als die bisher traversierten und multiplizierten Werte sein. Es gilt daher:

$$simVal(C_1, C_n) <= \{C_1, C_2, C_{n-1}, Cn\}$$

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 Such-Algorithmus benötigt, der Schritt für Schritt alle Nachbarn (und deren Nachbarn) sämtlicher Kompetenz prüft und den Weg zu allen anderen Kompetenzen berechnet. Es bieten sich einige Algorithmen dazu an:

- Das Paper[8] schlägt eine Implementierung des *Dijkstra-Algorithmus* für eine Berechnung sämtlicher Kompetenz-Paare des Graphen vor. 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 Dijkstra inherent aufweist.

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:

- 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 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 im Anhang nachgeschlagen werden.

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

Nun zur Methode, die rekursiv aufgerufen wird. 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.

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 von *Floyd-Welsh-Algorithmus* und Paper[8] nach erfolgreichem Abschluss als Index dient. Genauer analysiert wird dies bei den internen Komponenten in Kapitel 6.2. In der *AugmentedCompetenceCollection* werden sämtliche Suchergebnisse für alle bisher als Start-Knoten verwendete Kompetenzen gespeichert und mitgetragen.

```
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 && target.
       getSimilarity() >= threshold) {
5      collection.addCompetence(target);
6      //Überprüfung der Nachbarknoten
7      callChildren(collection, target);
8    }
9  }
```

```
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() > target.
       getSimilarity() ){
14         //Aktualisierung der Werte des mit dem Suchknoten
15         //verbundenen Knotens
16         updateValues(target, parent, sc.getDistance());
17         //Neuer rekursiver Aufruf mit der jeweiligen verbundenen
       Kompetenz
18         augmentIntern(collection, target);
19       }
20     }
21   }
22 }
```

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 letztens aktualisiert die *updateValues()*-Methode noch die Werte der verbundenen Knoten.

```
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 im Paper[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 keine Meta-Daten verloren gehen. 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 Grafik 4.2 veranschaulicht.

	Baking	Cooking	Stewing	Roasting	Frying	Deep Fry.
Baking	1	0.6	0.24	0.36	0.144	0.115
Cooking	0.6	1	0.4	0.6	0.24	0.192
Stewing	0.24	0.4	1	0.24	0.096	0.077
Roasting	0.36	0.6	0.24	1	0.4	0.32
Frying	0.144	0.24	0.096	0.4	1	0.8
Deep Fry.	0.115	0.192	0.077	0.32	0.8	1

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 in Kapitel 3.3.2 angesprochen und hier genauer erklärt. Die *CompetenceCollectionMatrix-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 Kapitel 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 Kapitel zu den Datentypen 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-Snippet präsentiert. Sämtliche hier verwendeten Attribute der Matrix-Klasse sind bei *Datentypen* in Kapitel 3 zu finden.

```
1  //Dieser Konstruktor verarbeitet automatisch alle Daten und erstellt die
       Matrix
2  public CompetenceCollectionMatrix(CracUser u, Task t,
       FilterConfiguration m) {
3    this.u = u;
4    this.t = t;
5    boolean doable = true;
6    this.taskCompetences = t.getCompetenceRelationships();
7    this.userCompetences = u.getCompetenceRelationships();
8
9    matrix = new MatrixField[userComps.size()][taskComps.size()];
10   buildMatrix();
11   markMandatoryViolation();
12   applyFilters(m);
13 }
```

In der *buildMatrix()-Methode* werden die korrekten Beziehungswerte aus der *CompetenceStorage-Komponente* geladen. Diese stellt die Suchergebnisse global zur Verfügung und wird in Kapitel 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            CompetenceStorage.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 Kapitel eingegangen.

4.2.4 Modifikation auf Basis von Meta-Daten

An den einzelnen Feldern der Matrix 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. Alle verwendeten Code-Snippets sind aufbereitet, um den Code besser verständlich zu machen, der tatsächliche Code kann im Anhang nachgelesen werden.

Grundsätzliche Implementierung

Sämtliche Modifikationen werden als Erweiterung des Datentyps *Filter* angelegt und überschreiben die relevanten Methode, um ein für die Matching-Matrix *uniformes Interface* zu garantieren.

```
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 "Gefällt mir"-Basis

Jeder Wert der Matrix wird basierend auf dem ”Like”- Level 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 der *Masterarbeit von Markus Raab*[12]. Dieser wurde leicht verändert, um den Anforderungen des *CrAc-Core* stand zu halten.

  $$simValLike(C_T, C_P) = simVal * (1 + (1 - simVal) * likeLvl)$$

  Wie in Grafik 4.3 zu sehen ist, werden die *Similarity-Values* bei steigendem oder sinkendem *likeLvl* des Users gegenüber der jeweiligen Kompetenz nach oben bzw. unten korrigiert. Diese Korrektur wird schwächer, je näher das Similarity-Value den Extremwerten null und eins 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.

```
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 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 angepasst. Dies hat zur Folge, dass Aufgaben, an denen bereits Freunde oder andere Freiwillige mit gutem Verhältnis eingeschrieben sind, vermehrt vorgeschlagen werden.

- Mathematische Herangehensweise
  Hier wird dasselbe mathematische Modell wie in der ”Gefällt mir”-Modifikation genutzt, dargestellt in Grafik 4.3. Anstatt das LikeLevel in Verbindung mit der Kompetenz für die Anpassung der Werte zu nutzen, wird hierbei allerdings auf das durchschnittliche LikeLevel des suchenden Users in Verbindung mit den bereits teilnehmenden Usern zurück gegriffen.

  $$simValUserLike(C_T, C_P) = simVal * (1 + (1 - simVal) * userLikeLvl)$$

- Algorithmische Umsetzung
  Um die Berechnung des genannten Durchschnitts in der algorithmischen Umsetzung effizienter zu gestalten, wird dieser im Filter gespeichert und nur bei wechselnden Usern rekalkuliert.

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.

```
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 }
```

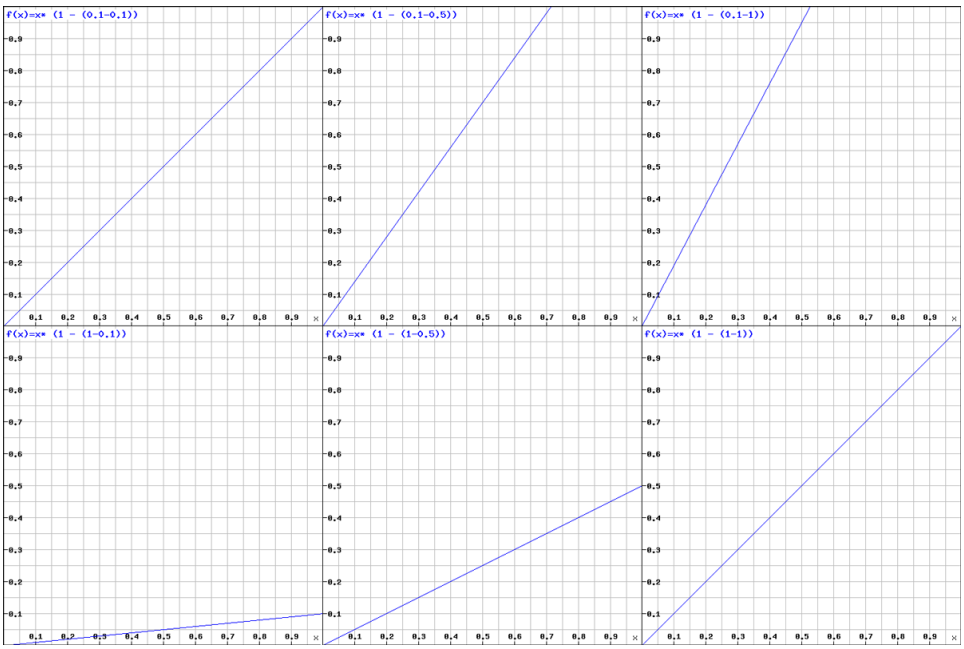


Abbildung 4.4: Die ”Proficiency”-Modifikation als Funktion

Modifikation auf Basis von Fähigkeiten

Jeder Wert der Matrix, bestehend aus dem *Similarity-Value* zweier Kompetenzen, wird basierend auf dem "Proficiency"-Level des Benutzers in Verbindung mit dem geforderten "Proficiency"-Level der Task 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 in der Arbeit von Markus Raab[12] komplett übernommen. Abhängig von der Differenz aus Können des Users und Anforderung des Tasks, wird das *Similarity-Value* angepasst. Abbildung 4.4 zeigt die verwendete Funktion.

$$simValProf(C_T, C_P) = simVal * (1 - (profLvlC_T - profLvlC_P))$$

- Algorithmische Umsetzung

```
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 * ((double) 1 - (((double) profLvlCTask /
       100) - ((double) profLvlCPerson / 100)));
10   }
11   if (simValProf > 1) { simValProf = 1; }
12   else if (simValProf < 0) { simValProf = 0; }
13   return simValProf;
14 }
```

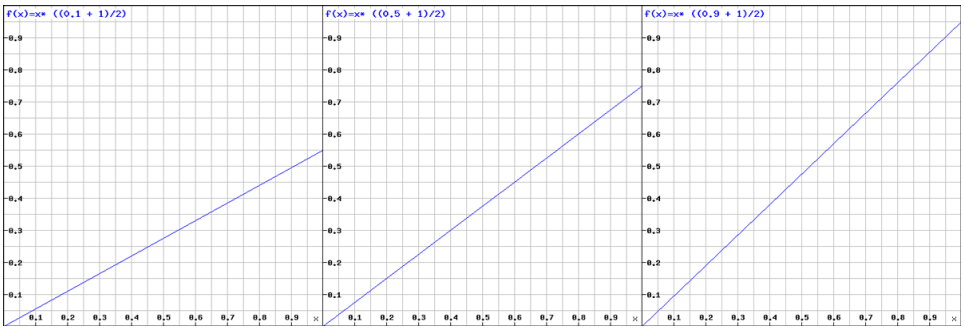


Abbildung 4.5: Die "Importance"-Modifikation als Funktion

Modifikation auf Basis von "Importance"

Hierbei werden die Felder der Matrix auf Basis der Wichtigkeit 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.

- Mathematische Herangehensweise
  In diesem Fall werden die Similarity-Values unwichtiger Kompetenzen abgeschwächt. Das verwendete System lässt dahingehend die *Similarity-Values* wichtiger Kompetenzen unverändert und erniedrigt alle anderen abhängig von ihrem *impLvl*. Dieses wird um einen vom Grundwert abhängigen Anteil erhöht, um die Abschwächung von Similarity-Values nicht zu extrem zu gestalten. Dargestellt ist sie in Grafik 4.5.

$$simValImp(C_T, C_P) = simVal * (\frac{impLvl_T + 1}{2})$$

- Algorithmische Umsetzung

```
1  @Override
2  public double apply(MatrixField m) {
3    double simVal = m.getVal();
4    int impLvl = m.getTaskRelation().getImportanceLevel();
5
6    double simValImp = simVal;
7    // do only if the value is not 1, since 1 means that the user
       possesses
8    // the competence
9    if (simVal != 1) {
10     simValImp = simVal * (( impLvl + 1) / 2);
11   }
12   return simValImp;
13 }
```

Kapitel 5

Feedback und Evolution

Mit dem erfolgreichen Beenden einer Aufgabe – unabhängig vom Outcome – endet der Gesamtprozess (dargestellt in Grafik 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. Auch hier wieder der Verweis auf den konkreten Code im Anhang.

5.1 Evaluierung

Der erste Schritt für den Abschluss des Loops ist das Evaluieren der Erfahrung durch den Benutzer. Hierfür wird die Utility-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 ein 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 der jeweiligen Evaluation und die Meta-Daten zu kombinieren. Die Modifikationen betreffen zwei Utility-Klassen 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 *proficiencyValue*, der durch das nachgewiesene Training gesteigert werden kann und zum anderen das *likeLevel*, welches anhand der Evaluierung angepasst wird.

Mathematische Grundlagen

Um das *proficiencyValue* zu erhöhen, wird auf eine sehr einfache Funktion zurückgegriffen, die eine Situations-unabhängige Steigerung des Wertes vornimmt.

$$profValNew(C,U) = profValOld(C,U) + 0.1$$

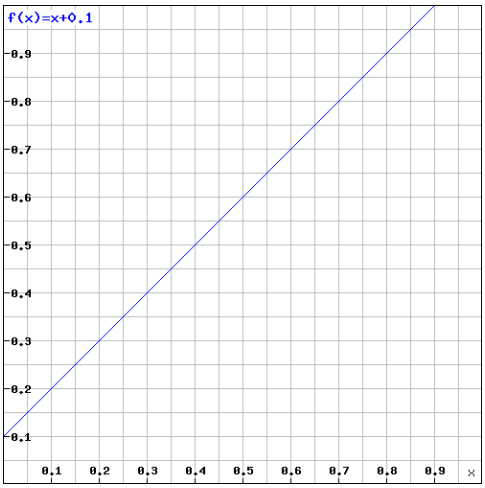


Abbildung 5.1: Die ”Proficiency”-Anpassung auf Basis der Evaluierung als Funktion

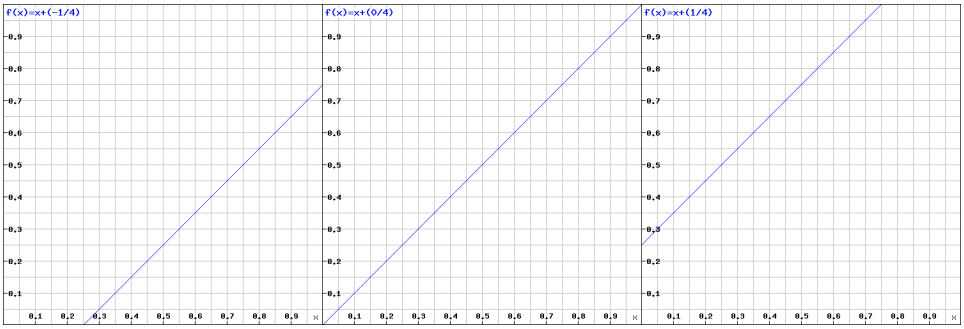


Abbildung 5.2: Die ”Like”-Anpassung auf Basis der Evaluierung als Funktion

Wie in Grafik 5.1 zu sehen ist, werden sämtliche Werte ausschließlich statisch erhöht.

Im Gegensatz dazu wird bei der Anpassung des *likeLevels* ein Prozentsatz der Bewertung 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.

$$likeValNew(C,U) = likeValOld(C,U) + \frac{likeEval(T,U)}{4}$$

Inwiefern die Werte innerhalb dieser Funktion angepasst werden, ist in Grafik 5.2 zu sehen.

Algorithmische Implementierung

Ähnlich der Implementierung der Matrix-Modifikationen wurde 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* (wie beschrieben in Kapitel 3.3).

```
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    int likeValue = ucr.getLikeValue();
5    int profValue = ucr.getProficiencyValue();
6    //Anwenden der Like-Funktion
7    likeValue += (evaluation.getLikeValTask() / 4) * 100 ;
8    if(likeValue > 100){
9      likeValue = 100;
10     }else if(likeValue < -100){
11     likeValue = -100;
12   }
13   //Anwenden der Proficiency-Funktion
14   profValue += 10;
15   if(profValue > 100){
16     profValue = 100;
17   }else if(profValue < 0){
18     profValue = 0;
19   }
20   //Speichern der Daten
21   ucr.setLikeValue(likeValue);
22   ucr.setProficiencyValue(profValue);
23   userCompetenceRelDAO.save(ucr);
24 }
```

5.2.2 Modifikation der Benutzer-Beziehungen

Dies 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.

Mathematische Grundlagen

Die mathematische Grundlage hierfür ist dieselbe wie für den *likeLevel* gegenüber den Kompetenzen der Aufgabe.

$$likeValNew(U_1, U_2) = likeValOld(U_1, U_2) + \frac{likeEval(U_1, U_2)}{4}$$

Dahingehend gilt Visualisierung 5.2 auch für diese Funktion.

Algorithmische Implementierung

Auch der algorithmische Teil geht sehr ähnlich vor, mit dem Unterschied, dass andere Beziehungen und Meta-Daten für die Kalkulation verwendet werden.

```
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-Prozess nun geklärt sind, beschäftigt sich dieses Kapitel mit der Umsetzung des Gesamtsystems und der Komponenten. Im Folgenden werden die Software-Architektur und ihr Design, sowie sämtliche Komponenten näher analysiert und diskutiert. Für mehr Übersichtlichkeit werden die Komponenten in zwei Kategorien aufgeteilt:

- Externe Komponenten, die nicht direkt Teil des *CrAc-Core*, sondern nur damit verbunden sind
- Interne Komponenten, welche als Teil von *Java Spring* implementiert sind und direkt zur Prototyp-Instanz gehören

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

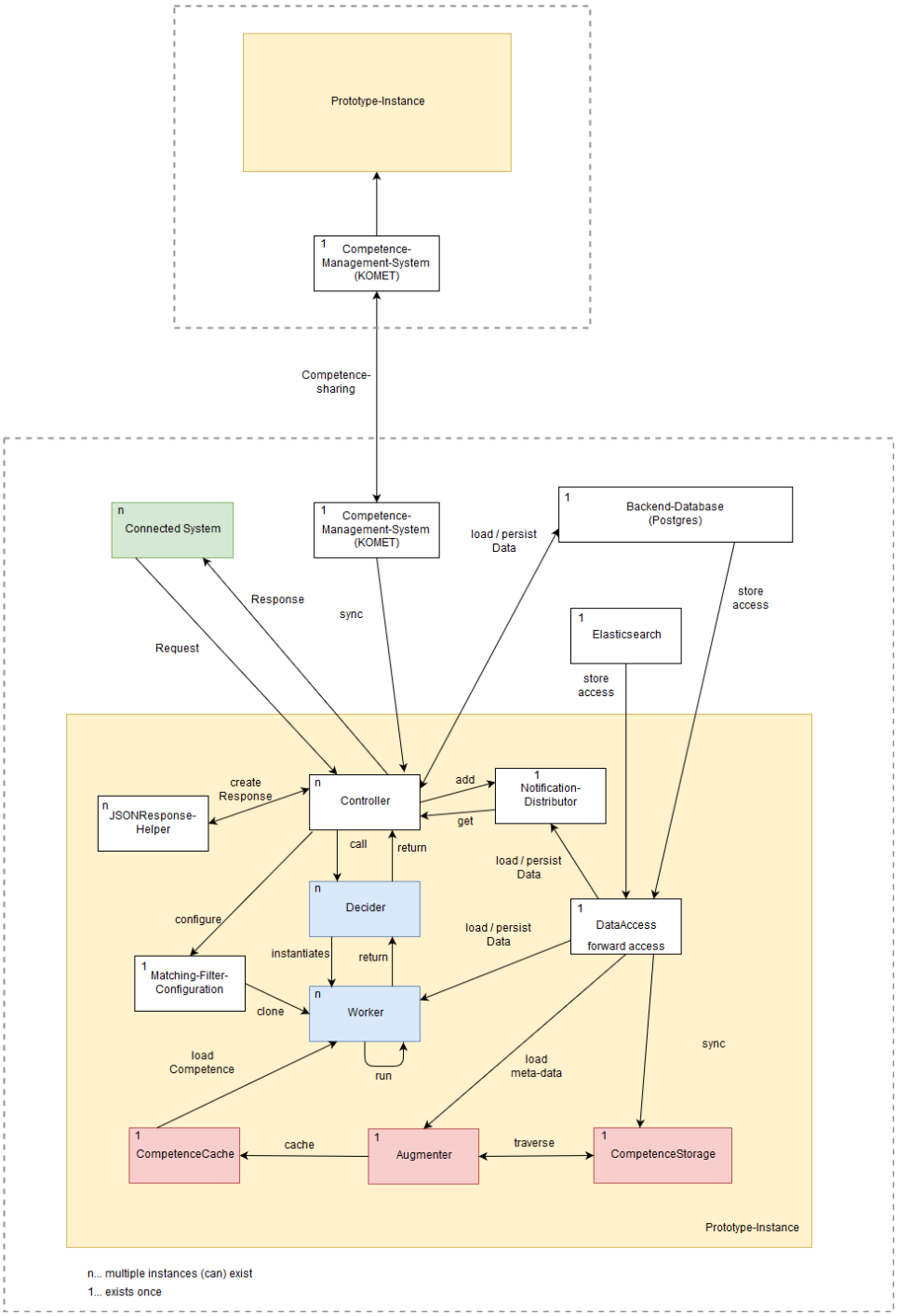


Abbildung 6.1: System-Diagramm des Frameworks.

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.

Per API verknüpfte Systeme

Wie in Kapitel 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*, diese werden im Anschluss an dieses Kapitel in Sektion 6.2 näher betrachtet. Im klassischen Fall ist ein solches – per API verknüpftes – System eine Art von Frontend, das den CrAc-Core mit den benötigten Grunddaten (nachzulesen in Sektion 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*[7]
- Für den Datenaustausch wird das Format *JSON* verwendet
- Das Ausgabe-Objekt ist eine *Rest-Response* (beschrieben in Kapitel 3.3.3)
- Der Input verwendet beliebige Attribute der *Post-Options-Klasse*, nur erforderlich bei Request-Methoden mit Datenübergabe (beschrieben in Kapitel 3.3.3)

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

[1] Für das CrAc-Projekt[4] ist beispielsweise die Implementierung eines solchen Frontends als weiteres Teilprojekts in Arbeit

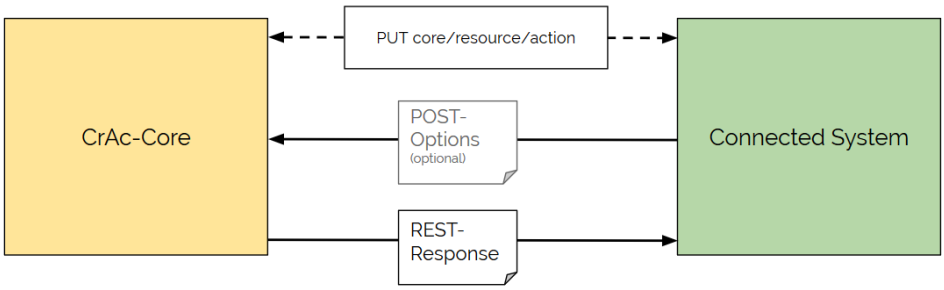


Abbildung 6.2: Zugriff eines angeschlossenen Systems via *PUT-Methode*

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*[11], 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.
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 (in Kapitel 6.2 erklärten) SynchronizationController 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.

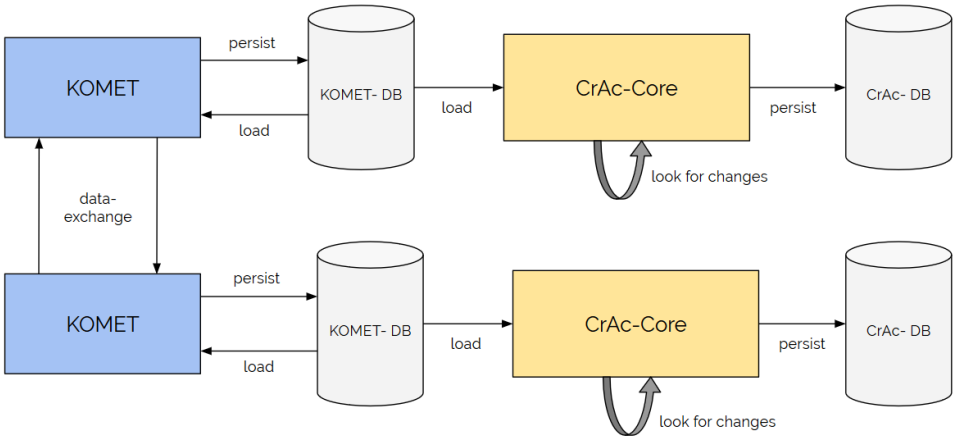


Abbildung 6.3: Daten-Synchronisierung zwischen KOMET und CrAc-Core

In Grafik 6.3 werde sowohl die Synchronisation zwischen CrAc und KOMET, als auch die die Datenübertragung zwischen mehreren KOMET-Versionen visualisiert.

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 Kapitel 2.1.1 erörtert. Wie die Datenbank mit dem *CrAc-Core* interagiert und inwiefern die Daten für diesen Prozess transformiert werden müssen, ist in Sektion 2.1.2 näher beschrieben.

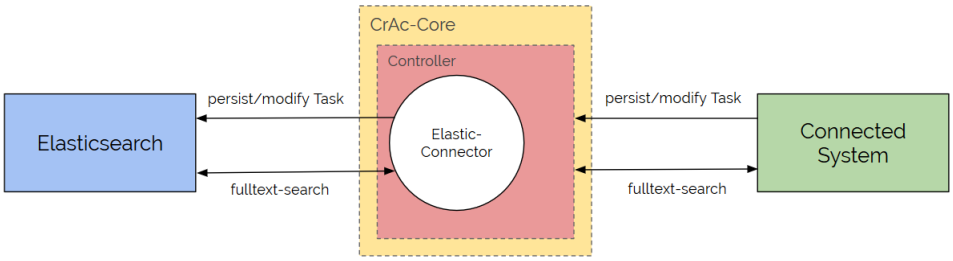


Abbildung 6.4: Zugriffsweg des Cores auf Elasticsearch

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 – wie beschrieben in Sektion 4.1.2 – für den *Matching-Prozess* getestet wurde. Folgende Merkmale lassen *Elasticsearch* im Fall des *CrAc-Cores* konkurrenzlos erscheinen:

- 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 seit geraumer Zeit mit Abstand auf Platz 1 [5]. In Grafik 6.4 wird gezeigt, wie der CrAc-Core mittels *Elastic-Connector*, erörtert in Sektion 3.3.1, auf die Elasticsearch-Instanz zugreift und Aufgabendaten persistiert bzw. Volltextsuchen durchführt.
Der Zugriff auf Elasticsearch wird in Grafik 6.4 dargestellt.

6.2 Interne Komponenten

Folgende Komponenten sind Teil des eigentlichen *CrAc-Core* und damit in der in Kapitel 2 angeführten Technologie *Java Spring* erstellt. Sie bilden den Kern der Applikation und verarbeiten mithilfe aller bereits besprochenen Klassen und umgesetzten Algorithmen die Daten externer Quellen 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 Komponenten 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 Grafik 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 wie in Kapitel 5.1 beschrieben ein)
    - NotificationController, Zugriff auf Nachrichten-bezogene Methoden (interagiert mit den – in Kapitel 3.3.2 – beschriebenen Notifications)

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

Im folgenden Code-Snippet wird nicht nur gezeigt, 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

Die Data-Access-Komponente verwaltet den Zugang zu Bereichen der *Datenbank und Elasticsearch*. Auf Basis von *JPA* wird beim Starten der Java-Applikation, mit Hilfe von *Autowiring[13]* (dargestellt in Code-Snippet 6.2), für jeden in der Datenbank persistierten Datentypen des *CrAc-Cores*, ein Objekt als Schnittstelle angelegt. Diese Crud-Repository-Objekte (veranschaulicht in Sektion 3.3.1) können im Anschluss genutzt werden, um Daten als Objekte des jeweiliges Datentyps aus der Datenbank zu laden oder zu modifizieren. Aufgrund ihrer Natur[13] sind diese Repositories jedoch nur in Controllern (und weiteren zum Systemstart automatisch erstellten Objekten) verfügbar, während später instanziierte und unabhängige Komponenten keinen Zugriff haben. Zwei mögliche Lösungen für dieses Problem bieten sich an.

1. Weitergeben der Repository-Referenz an jede Komponente bzw. deren Methoden aus dem Controller
   Dies funktioniert zwar, macht das Gesamtsystem jedoch zusehends unübersichtlicher, da eine große Anzahl an Komponenten 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 zu umgehen und ein schöneres Softwaredesign, sowie eine verbesserte Lesbarkeit zu erreichen, wird auf *Ansatz 2* zurück gegriffen.
2. Ein globales Objekt mit sämtlichen Referenzen
   Die Referenzen sämtlicher Zugänge werden beim Start des *CrAc-Core* vom SynchronizationController aus automatisch in die *Data-Access-Komponente* gespeichert. Von dort 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 in der *Data-Access-Komponente* auch die Zugänge zu den einzelnen Indizes, in Form von *Elastic-Connector-Objekten*, hinterlegt sind. Die Zugriffsmethoden sind generisch und erlauben das Hinzufügen und Laden sämtlicher Objekte auf Basis ihrer Klasse. Der Code dafür kann in der Code-Zusammenfassung im Anhang nachgelesen werden.

JSON-Response-Helper

Diese Komponente ist für den strukturierten Output im Format *JSON* zuständig, welchen sie auf Basis des Jackson-Mappers und der *REST-Response-Klasse* (erläutert in Sektion 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-Response-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 (in Kapitel 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 Komponenten 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 Komponenten *Cache* und *Storage*, 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 Kapitel 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. 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 Storage um die Daten aus dem *Traversierungsverfahren*, beschrieben in Sektion 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 Komponenten.

Eine solche stellt der *Augmenter* dar. Dieser nutzt – als zweite Teilkomponente der Kompetenz-Verarbeitung – die Implementierung des *Traversier-Algorithmus*, um zusätzliche Informationen aus dem *Kompetenz-Graphen* zu interpolieren. Hierbei wird exklusiv mit der *Cache-Komponente* interagiert, welche Zugriff auf den Graphen und einen Speicherort für die Resultate bietet. Damit stellt der Augmenter das Bindeglied im Prozess des *Verarbeitungsprozesses* dar.
Für einen besseren Überblick zeigt Grafik 6.5, in welcher Reihenfolge die Prozesse der *Kompetenz-Verarbeitung* abgearbeitet werden, beginnend bei

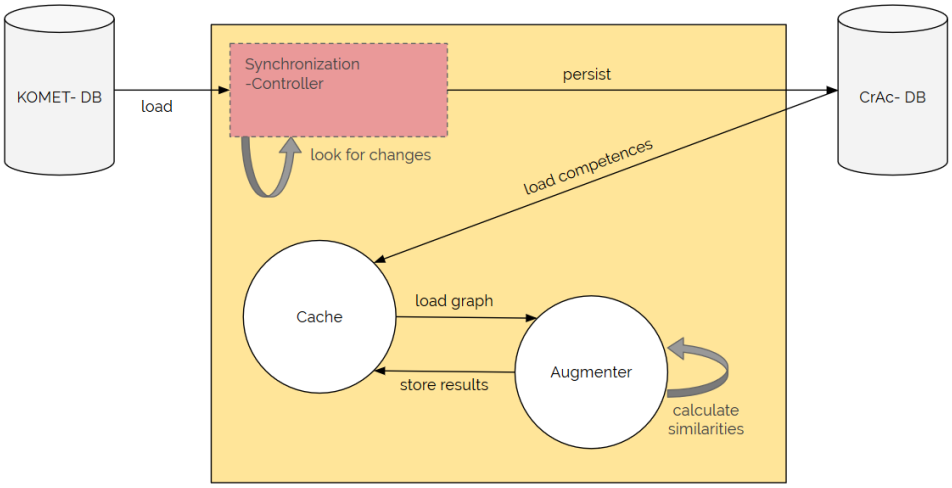


Abbildung 6.5: Prozesskette zur Kompetenz-Verarbeitung

der Synchronisierung der *KOMET-Daten*.

Decider

Die Decider-Komponente 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 sind erklärt in 3.3.2 und 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. Diese Komponente ist jedoch nicht nur 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 (eg. Benutzung eines Thread-Pools)
  - die Reihenfolge der Ausführung (Festlegung möglich in beliebigen Datenstrukturen)
- Komponente ist beliebig erweiterbar durch das Einbinden und Kombinieren von Worker-Einheiten

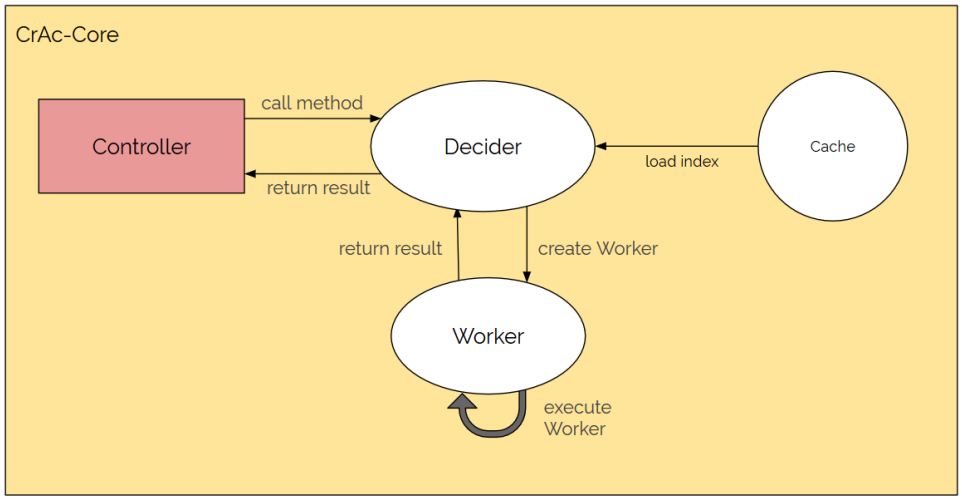


Abbildung 6.6: Abbildung der Worker-Decider-Struktur

In Grafik 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 – in Sektion 3.3.2 erklärten – Implementierungen der *Notification-Klasse*. Um die Interaktion mit diesen einfacher zu gestalten, besitzt der Distributor mehrere Funktionen:

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

Notifications werden bewusst im Speicher 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 hin-

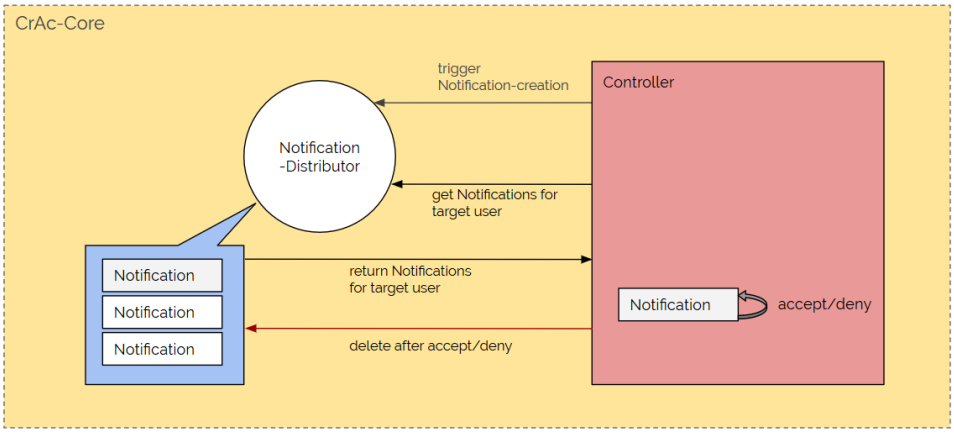


Abbildung 6.7: Abbildung der Notification-Distributor-Interaktionen

sichtlich Skalierbarkeit sehr ähnlich, da er mit der Menge an Datentypen, für die er eine Schnittstelle darstellt, mitwächst. In Grafik 6.7 sind der Distributor und seine Interaktion mit den Notifications dargestellt.

Matching-Filter-Konfiguration

Auch die letzte Komponente dieses Kapitels, sie komplettiert den CrAc-Core, hat mit dem Matching-Prozess zu tun. Wie der Name der Komponente bereits aussagt, enthält sie die globale Konfiguration sämtlicher *Matching-Filter*. Jeder dieser Filter modifiziert – wie in Sektion 5.2 diskutiert – 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 festgelegten 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-Snippet werden die wichtigen Teile der Klasse und ihre Funktionsweise gezeigt.

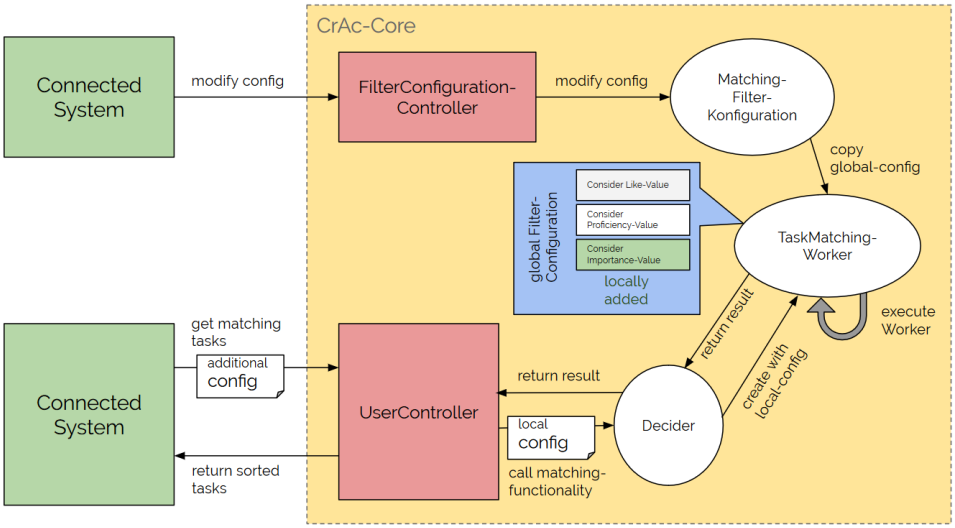


Abbildung 6.8: Prozess-Kette der Matching-Filter-Konfiguration

```
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 Grafik 6.8 die Prozess-Kette von der Setzung der globalen Konfiguration bis zum *Matching*.

Kapitel 7

Use-Case und Tests

Da das System und sämtliche Komponenten, 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 der Komponenten und Klassen näher beleuchtet und die einzelnen Schritte der Informationsverarbeitung werden zusammenhängend dargestellt. Weiters werden Tests auf Basis verschiedener *Personas* 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 einer eigenen Sektion aufgearbeitet dargestellt werden. Im Anschluss werden mögliche Anpassungen auf Basis der Werte diskutiert.

7.1 Use-Case

Zunächst zum Use-Case und den darin existierenden Benutzern:

- Ein Admin, der die Kompetenzen anlegt und synchronisiert
- Ein Creator, der eine Aufgabe anlegt
- Ein Standard-Benutzer, der nach einer passenden Aufgabe sucht

Folgende Schritte passieren nun im *CrAc-Core*, um das gewünschte Ziel - eine passende Aufgabe zu finden, auszuführen und evaluieren - zu erreichen. Alle Schritte sind außerdem nach ausführendem User sortiert.

Der Administrator

Als ersten Schritt zur Verwendung des Cores kümmert sich ein beliebiger Administrator um das Anlegen und Laden der Kompetenzen.

1. Die nötigen Daten müssen in der verbundenen KOMET-Instanz existieren
   - Die Raster müssen händisch angelegt oder
   - von einer anderen KOMET-Version kopiert werden
2. Mit dem Endpoint *"GET core/synchronization/competence"* wird der Synchronisierungs -/ und Indizierungsprozess eingeleitet
   (a) *Spring* leitet die Anfrage aufgrund des Parameters *"/synchronization"* an den zugewiesenen *SynchronizationController* weiter
   (b) Der Controller wählt aufgrund des Parameters *"/competence"* die zugewiesene Methode *syncCompetences()* aus, diese führt mehrere Dinge hintereinander aus, angefangen bei der Synchronisation der Datenbanken
      i. In der Methode werden die Informationen aus der KOMET-Datenbank geladen
      ii. Eine Schleife prüft und validiert alle Daten
      iii. Bei positiver Validierung in die *CrAc-Core*-DB persistiert
   (c) Nach dem Kopieren werden nun die Core-internen Komponenten genutzt, um die Kompetenzen für das Matching aufzubereiten
      i. Laden der Kompetenzen aus der CrAc-DB
      ii. Vereinfachen und Ablegen der Kompetenzen als simplere Version in der *Storage-Komponente*
      iii. Traversieren und Indizieren der Kompetenzen, sowie Ablegen der Ergebnisse im *Cache* via *Augmenter-Komponente*
   (d) Informationen über den Prozess und die Ergebnisse in Form von JSON an den aufrufenden Admin

Der Creator

Als nächstes müssen eine oder mehrere Aufgaben im System angelegt werden, um ein *Matching* möglich zu machen. Es existiert hierfür eine eigene Benutzer-Rolle im Systeme, der ein Anlegen von Aufgabenbäumen erlaubt ist. Für das Beispiel wird ein einzelner Aufgabebaum angelegt, die Funktionsweise das Anlegen weiterer ist dieselbe.

1. Erstellen eines Aufgabenbaumes mittels Endpoint *"POST /core/task"* und den Daten der Aufgabe als *JSON-Dokument*
   (a) *Spring* leitet die Anfrage aufgrund des Parameters *"/task"* an den zugewiesenen *TaskController* weiter
   (b) Der Controller wählt aufgrund des fehlenden Parameters und der Request-Methode die zugewiesene Methode *createTask()* aus
   (c) Die übergebenen JSON-Daten werden per Jackson in ein Task-Objekt konvertiert, Jackson validiert hierbei die Attribute, besonders wichtig ist das Attribute *TaskType*, welches die Art und Erweiterbarkeit der Aufgabe beschreibt
   (d) Der anlegende Creator erhält die *Leader-Rolle* des obersten Knoten im Baum
   (e) Die Aufgabe wird in der Datenbank gespeichert
2. Erweitern des Aufgabenbaumes mittels Endpoint *"POST /core/task/supertask_id/extend"* und den Daten der erweiternden Aufgabe als *JSON-Dokument*, dies ist nur möglich, wenn die Auflagen des *TaskType* erfüllt werden (beschrieben in Sektion 3.1.2)
   (a) *Spring* wählt aufgrund der Parameter die Methode *extendTask()* aus
   (b) Die übergebenen JSON-Daten werden per Jackson in ein Task-Objekt konvertiert, Jackson validiert hierbei die Attribute
   (c) Um die Aufgabe als Teil des Baumes zu etablieren, wird die zu erweiternde Aufgabe als *superTask* der neuen gesetzt
   (d) Die Aufgabe wird in der Datenbank gespeichert
3. Hinzufügen von Kompetenzen (optional, aber empfohlen, um den Matching-Prozess nutzen zu können) durch einen Aufruf des Endpoint *"PUT core/task/task_id/competence/require"*, die Daten der zuzuweisenden Kompetenzen werden via JSON übertragen
   (a) *Spring* wählt aufgrund der Parameter die Methode *requireCompetences()* aus
   (b) Die Daten der zuzuweisenden Kompetenzen wird in ein Array an *Aufgaben-Kompetenz-Beziehungen* konvertiert, Jackson validiert hierbei die Attribute

   (c) Sämtliche positiv validierten Beziehungen werden gespeichert
   (d) Sollte ein beliebiger Teil dieser Beziehungen bereits in der Datenbank existieren, werden diese aktualisiert

4. Zuletzt das Freigeben der Aufgabe, um diese für einen suchenden User auffindbar zu machen, wobei angenommen wird, dass die bis jetzt angegebenen Daten positiv validiert wurden und das Attribut *readyToPublish* der Aufgaben im Baum aus diesem Grund den Wert *true* trägt; dieses wird eingeleitet durch den Aufruf des Endpoint *"PUT core/task/task_id/state/publish"*
   (a) *Spring* wählt aufgrund der Parameter die Methode *changeTaskState()* aus
   (b) Es wird geprüft, ob ein Wechsel des *TaskState* von auf den Zustand *Published* möglich ist
   (c) Wenn ja, wird der Zustand gewechselt und die Aufgabe ist öffentlich einsehbar
5. Der aufrufende Benutzer bekommt bei jedem Endpoint-Aufruf Informationen über den Prozess und die Ergebnisse in Form von JSON zurück geliefert

Der Standard-User

Nun da die nötigen Daten in richtiger Form existieren, kann zum Hauptteil übergegangen werden, dem Suchen, Ausführen und Evaluieren des freiwilligen Helfers.

- Mit dem Endpoint *"GET core/task/find"* wird der Suchprozess nach Aufgaben eingeleitet
    1. *Spring* wählt aufgrund der Parameter die Methode *findTasks()* im *TaskController* aus
    2. Die Methode instantiiert eine *Decider-Einheit*
    3. Dessen *findTasks()-Methode* wird wiederum aufgerufen, welche einen *TaskMatchingWorker* erzeugt und mittels Aufruf der überschriebenen *run()-Methode* ausführt, darin werden folgende Schritte durchgeführt
        (a) Aufgaben werden geladen und vorsortiert
        (b) Die globale Matching-Filter-Konfiguration wird kopiert
        (c) Für jede Aufgabe wird eine *Matching-Matrix* erzeugt, die Matching-Filter-Konfiguration wird übergeben
            i. Die Matrix lädt die nötigen *Similarity-Values*, basierend auf den Kompetenzen des Users und einer Task, aus der *Storage/Cache-Komponente*
            ii. Das 2-Dimensionale Array wird aufgebaut und auf *Mandatory-Violations* untersucht
            iii. Die übergebenen *Matching-Filter* werden auf das Array angewandt
            iv. Der ermittelte *Matching-Score* kann ausgelesen werden
        (d) Der Worker eliminiert alle Aufgaben mit einem Score von 0 und sortiert den Rest
        (e) Die *Decider-Einheit* gibt die Ergebnisse des Workers an die aufrufende Methode weiter
        (f) Diese nutzt den *JSON-Response-Helper*, um die Response des Endpoints zu generieren
            i. *createResponse()* mit den Parametern *Objekt, Success* wird verwendet
            ii. Diese ruft die Hauptmethode *createResponse()* auf
            iii. Die Attribute werden geprüft und das *REST-Response-Objekt* wird instantiiert
            iv. *Jackson* transformiert das Objekt in ein *JSON-Dokument*

Die *findTasks()-Methode* schickt das transformierte Objekt, welches die gefundenen Aufgaben enthält an den Aufrufer des Endpoints

- Um an einer der retournierten Aufgaben teilzunehmen, wird der Endpoint *"PUT core/task/task_id/add/participate"* aufgerufen
  1. *Spring* wählt aufgrund der Parameter die Methode *addToUser()* im *TaskController* aus
  2. Es wird geprüft, ob an der Aufgabe teilgenommen werden darf
  3. Es wird geprüft, ob der Benutzer bereits teilnimmt oder der Aufgabe folgt (ob bereits eine User-Task-Relationship besteht)
  4. Die Relationship wird entweder erzeugt, oder angepasst und in der Datenbank gespeichert
- Das Abschließen einer Aufgabe funktioniert via dem Endpoint *"PUT core/task/task_id/done/true"*
  1. *Spring* wählt aufgrund der Parameter die Methode *singleUserDone()* im *TaskController* aus
  2. Das Attribut *completed* der *User-Task-Relationship* wird auf *true* gesetzt und die Relationship wird gespeichert
  3. Der Status aller Teilnehmer wird geprüft, sollten alle fertig sein, werden die Leader der Aufgabe informiert, um sie abzuschließen
- Für die Evaluierung einer Aufgabe und sämtlichten Beteiligten, wird der Endpoint *"PUT core/evaluation/evaluation_id"* aufgerufen, wobei die ID jedem Benutzer mittels *Notification* zugesandt wird; die Daten werden als JSON-Dokument übergeben
  1. *Spring* wählt aufgrund der Parameter die Methode *evaluateTask()* im *EvaluationController* aus
  2. Die übergebenen JSON-Daten werden per Jackson in ein Task-Objekt konvertiert, Jackson validiert hierbei die Attribute
  3. Die Werte der erzeugten Evaluierung werden in der Datenbank gespeichert
  4. Eine *Decider-Einheit* wird erzeugt, um mittels Feedback-Daten und *Workers* Meta-Daten anzupassen
  5. Die *evaluateUsers()-Methode* des Deciders wird aufgerufen
     (a) Ein *UserRelationEvolutionWorker* wird instantiiert und die *run()-Methode* wird ausgeführt
     (b) Der Worker lädt die *User-Relationships* für alle beteiligten Freiwilligen, nicht existierende werden neu erstellt
     (c) Die Meta-Daten werden laut Modell angepasst
     (d) Die Relationships werden gespeichert

6. Die *evaluateTask()-Methode* des Deciders wird aufgerufen
   (a) Ein *UserCompetenceRelationEvolutionWorker* wird instantiiert und die *run()-Methode* wird ausgeführt
   (b) Der Worker lädt die *User-Competence-Relationships* für allen der Aufgabe zugewiesenen Kompetenzen
   (c) Die Meta-Daten werden laut Modell angepasst
   (d) Die Relationships werden gespeichert

- Der aufrufende Benutzer bekommt bei jedem Endpoint-Aufruf Informationen über den Prozess und die Ergebnisse in Form von JSON zurück geliefert

7.2 Tests

Der nächste Teil dieses Kapitels beschäftigt sich mit einiges Tests am CrAc-Cores. Diese fokussieren sich auf den Kernaspekt, nämlich das Matching- und die Evaluierungsmechanismen. Um die Tests so aussagekräftig wie möglich zu gestalten wird ein realitätsnahes, detailliertes Testszenario verwendet, welches im Folgenden erklärt wird.

7.2.1 Aufbau

Zuerst also zum Aufbau. Um die Tests durchführen zu können, werden als Grundgerüst Datensätze sämtliche Hauptdatentypen (Kapitel 3.1) benötigt. Diese müssen für die Nutzung aller Features – durch die Anpassung von Meta-Daten – mittels ihrer jeweiligen Beziehungsdatentypen (Kapitel 3.2) verknüpft außerdem sein. Hierbei wird auf die Daten in derselben Reihenfolge eingegangen, in der sie auch in der Realität erstellt werden würden. Zunächst also zur Struktur der Kompetenzen und ihrer – den Graphen bildenden – Kompetenz-Beziehungen, diese sind in Abbildung 7.1 visualisiert. Die Bezeichnungen der Kantengewichtungen in der Grafik stehen für folgende Werte:
t1 → 1.0, *t2* → 0.8, *t3* → 0.6, *t4* → 0.4, *t5* → 0.2
Als nächstes zu den Benutzern und ihren Beziehungen untereinander – dargestellt in Grafik 7.2 – , der Wert der Kanten bezieht sich hierbei auf das *LikeLevel* der jeweiligen Beziehung:
An dieser Stelle muss ebenfalls auf die Beziehungen zwischen Benutzern und Kompetenzen eingegangen werden, welche für die gelernten Fähigkeiten der jeweiligen Person im System stehen. Die Darstellung der Daten erfolgt der Übersichtlichkeit halber als Aufzählung.
Die Abkürzungen stehen für folgende Attribute:
LikeVal → LikeValue, *ProfVal* → ProficiencyValue

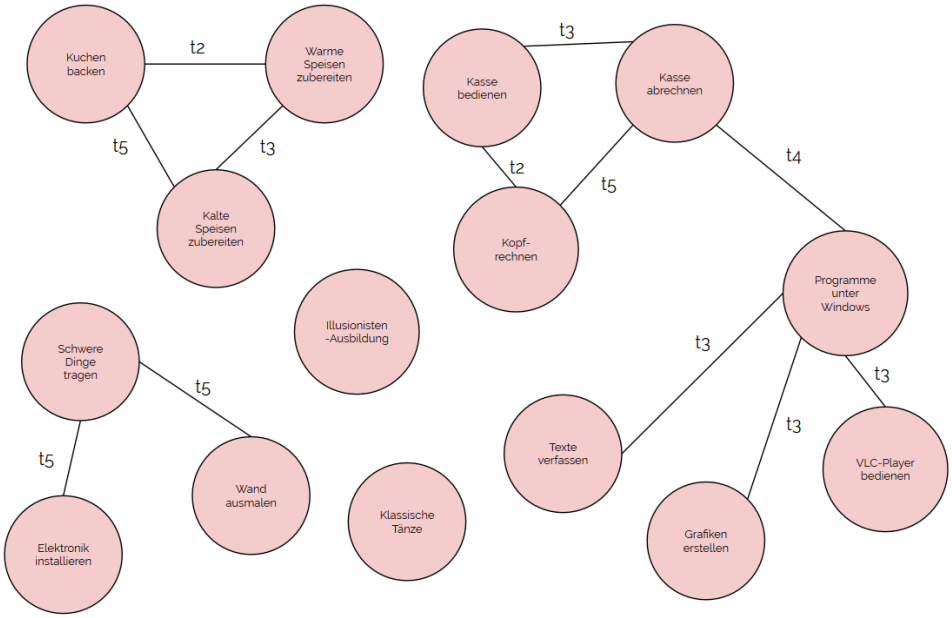


Abbildung 7.1: Die 15 Kompetenzen und ihre Beziehungen untereinander

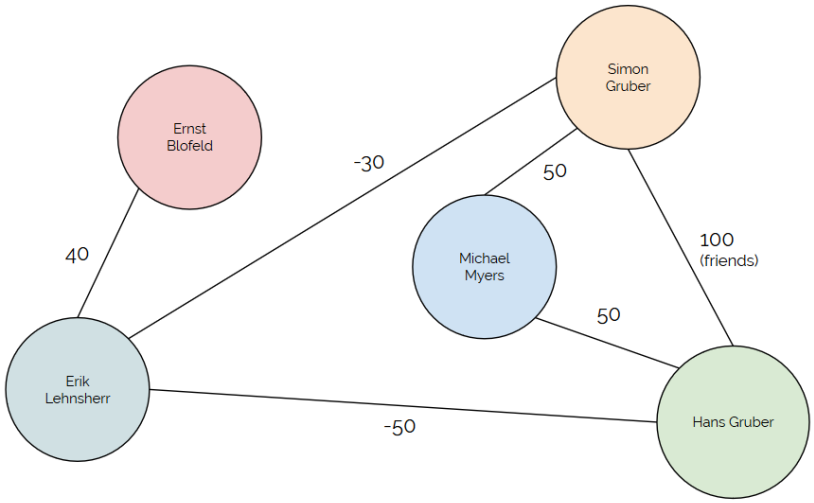


Abbildung 7.2: Die 5 Benutzer und ihre Beziehungen untereinander

- Hans Gruber
  - "Schwere Dinge tragen", LikeVal: 60, ProfVal: 100
  - "Klassische Tänze", LikeVal: 90, ProfVal: 70
  - "Kopfrechnen", LikeVal: 30, ProfVal: 50

- Simon Gruber
  - ”Elektronik installieren”, LikeVal: 100, ProfVal: 100
  - ”Kalte Speisen zubereiten”, LikeVal: 90, ProfVal: 70
  - ”Schwere Dinge tragen”, LikeVal: 50, ProfVal: 50
  - ”Illusionisten-Ausbildung”, LikeVal: -50, ProfVal: 50
- Michael Myers
  - ”Kasse bedienen”, LikeVal: 90, ProfVal: 100
  - ”Kasse abrechnen”, LikeVal: 90, ProfVal: 100
  - ”Kopfrechnen”, LikeVal: 50, ProfVal: 100
  - ”Kuchen backen”, LikeVal: 0, ProfVal: 30
- Erik Lehnsherr
  - ”Texte erfassen”, LikeVal: 100, ProfVal: 80
  - ”Grafiken erstellen”, LikeVal: 100, ProfVal: 90
  - ”Illusionisten-Ausbildung”, LikeVal: -50, ProfVal: 60
  - ”Bedienen von Windows-Programmen”, LikeVal: 30, ProfVal: 30
  - ”VLC-Player bedienen”, LikeVal: 30, ProfVal: 30
- Ernst Blofeld
  - ”Wand ausmalen”, LikeVal: 80, ProfVal: 20
  - ”Kuchen backen”, LikeVal: 80, ProfVal: 100
  - ”Kalte Speisen zubereiten”, LikeVal: 90, ProfVal: 100
  - ”Warme Speisen zubereiten”, LikeVal: 100, ProfVal: 100
  - ”Schwere Dinge tragen”, LikeVal: 30, ProfVal: 30

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[1] ausgewählt. Grafik 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 Organisational-Tasks nicht für das Matching gedacht sind, ist es hier 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 Modifikation auf Kompetenz-Basis erfolgen können, weder im Matching, noch in der Evaluierung.

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

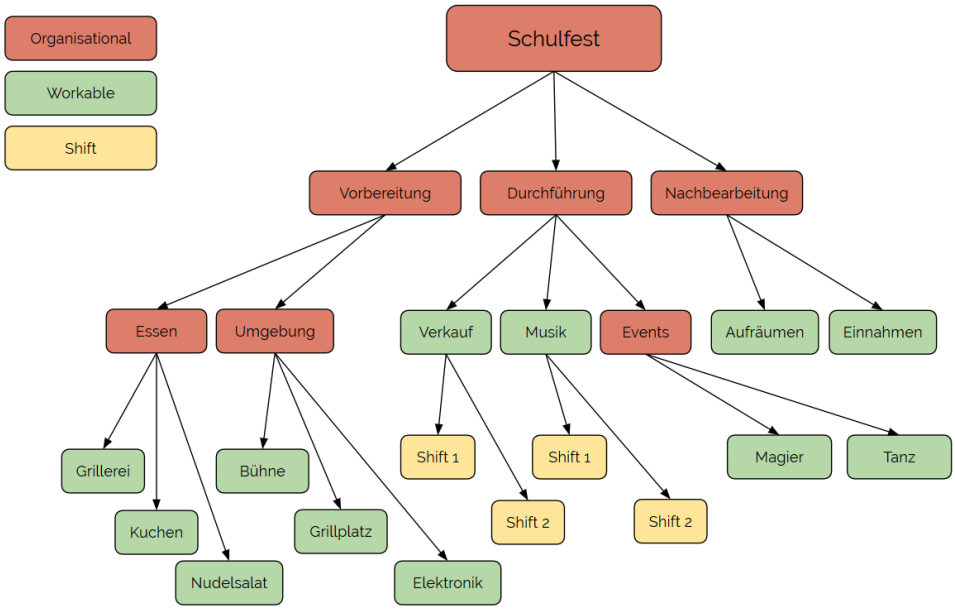


Abbildung 7.3: Der Aufgabenbaum, inklusive einer Legende der TaskTypes

Darum sollten nur sehr einfach zu lösende Aufgaben, ohne erkennbaren Lehrwert oder Anspruch, auf die Kompetenz-Zuweisung verzichten.
Die Abkürzungen stehen für folgende Attribute:
ImpLvl → ImportancyLevel, *ProfVal* → ProficiencyValue

1. Verkauf von Essen, Getränken und Tombola-Losen am Schulfest
   - ”Kopfrechnen”, ImpLvl: 70, ProfVal: 50
   - ”Kasse bedienen”, ImpLvl: 70, ProfVal: 40
2. Musikalische Untermalung am Schulfest
   - ”Bedienen von Windows-Programmen”, ImpLvl: 90, ProfVal: 50
   - ”VLC-Player bedienen”, ImpLvl: 100, ProfVal: 40
3. Aufräumen am Vormittag nach dem Schulfest
   - Keine Kompetenzen zugewiesen
4. Die Einnahmen der Kasse am Vormittag nach dem Schulfest rechnen
   - ”Kasse abrechnen”, ImpLvl: 100, ProfVal: 60
   - ”Kopfrechnen”, ImpLvl: 80, ProfVal: 60
5. Vorbereitung des Kuchens
   - ”Kuchen backen”, ImpLvl: 100, ProfVal: 60

6. Vorbereitung der Grillerei
    - Keine Kompetenzen zugewiesen
7. Vorbereitung des Nudelsalates
    - ”Kalte Speisen zubereiten”, ImpLvl: 100, ProfVal: 60
8. Vorbereitung des Grillplatzes
    - ”Schwere Dinge tragen”, ImpLvl: 100, ProfVal: 60
    - ”Wand ausmalen”, ImpLvl: 100, ProfVal: 30
9. Vorbereitung der Bühne
    - ”Schwere Dinge tragen”, ImpLvl: 100, ProfVal: 70
    - ”Elektronik installieren”, ImpLvl: 100, ProfVal: 60
10. Vorbereitung der Elektronik
    - ”Schwere Dinge tragen”, ImpLvl: 60, ProfVal: 20
    - ”Elektronik installieren”, ImpLvl: 100, ProfVal: 90
11. Auftritt als Illusionist
    - ”Illusionisten-Ausbildung”, ImpLvl: 100, ProfVal: 100, *mandatory*
12. Vorführung klassischer Tänze
    - ”Klassische Tänze”, ImpLvl: 100, ProfVal: 90

Nachdem die Test-Daten nun hinreichend bekannt sind, wirft die nächste Sektion einen Blick auf die Tests selbst.

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 – erklärt in Kapitel 4.2.4 – notwendig, die Benutzer 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. Dieses wird anhand von zwei Usern erörtert und ist visualisiert in Abbildung 7.4 und 7.5. Die restlichen Testergebnisse sind im Anhang dieser Thesis zu finden.

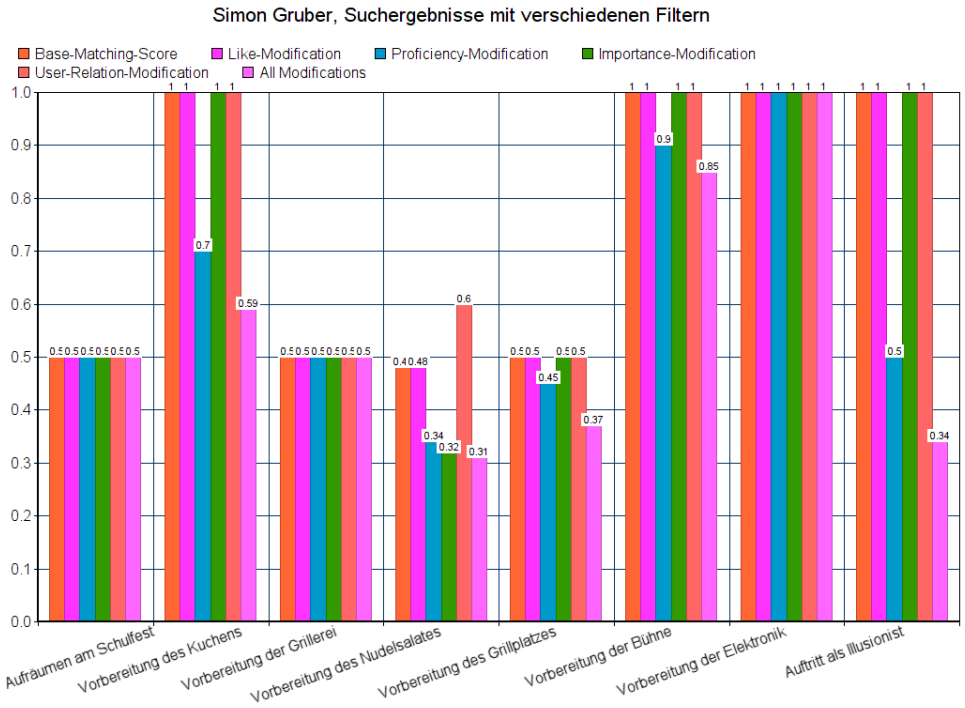


Abbildung 7.4: Ergebnisse von Simon Gruber mit unterschiedlichen Filtern

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

Fall 1 tritt bei den beiden Aufgaben "Aufräumen am Schulfest" und "Vorbereiten der Grillerei" 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" 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.

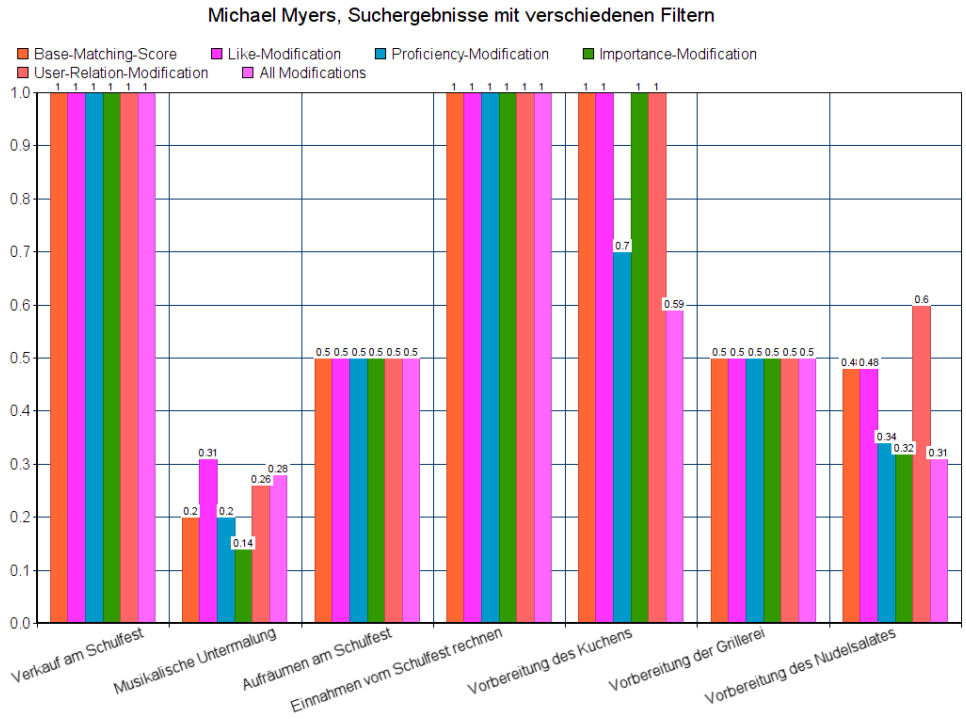


Abbildung 7.5: Ergebnisse von Michael Myers mit unterschiedlichen Filtern

Eine weitere Auffälligkeit stellt die Tatsache dar, dass sich der Ergebniswert beim Matching mit allen Filtern in den meisten Fällen sehr von den restilichen Scores unterscheidet. Dies ist in fast allen Aufgaben beider Grafiken 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 (Richtung 0 und Richtung 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" in Abbildung 7.4) indem sie schwächer wird und ändert die Werte ansonsten gemäß den gegebenen Meta-Daten (Aufgabe "Musikalische Untermalung" in Abbildung 7.5).

Der Proficiency-Filter schwächt den Score bei zu geringen Werten ab (Aufgabe "Vorbereitung des Nudelsalat" in Abbildung 7.5 und Aufgabe "Vorbereitung des Kuchens" 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" 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 beispielsweise beide dargestellten Freiwillige, die Aufgabe "Vorbereitung des Nudelsalates" 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" in Abbildung 7.4. Der Filter passt hier lediglich Similarity-Values an, die vom Matching – aufgrund eines besseren Wertes – verworfen werden, 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 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 kann und wie er die Beziehungen zu anderen Freiwilligen ausbaut. Auch hier wird das in 7.2.1 beschriebene Test-Setup verwendet.

Durchführung

Die Regeln der Durchführung für 10 Iterationen:

- Pro Iteration führt jeder der Test-Users 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 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

Generell werden so viele Aktionen der Benutzer wie möglich dem Zufall überlassen, um die Ergebnisse – so wie in der Realität – unabhängig vom Tester zu machen.

Ergebnisse

Auch im zweiten Test werden stellvertretend für die 5 Test-User wieder zwei in Grafik und Text präsentiert. Auch hier wieder der Verweis auf die vollständigen Tests im Anhang der Thesis. Aufgrund der schieren 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 folgende Aufgaben absolviert und evaluiert[2], die Nummern der Aufgaben entsprechen ihrer Nummerierung in Sektion 7.2.1:

	it1	it2	it3	it4	it5	it6	it7	it8	it9	it10
Task	3	1	8	9	10	8	2	9	12	1
User-Eval	0.45	0.35	0.35	-0.45	-0.4	0.05	0.3	-0.15	-0.2	-0.8
Task-Eval	0.25	0.2	0.2	0.55	-0.05	-0.2	-0.85	-1	0.55	0.7
User No.	3	5	5	2	1	1	5	1	4	4

Tabelle 7.1: Absolvierte und evaluierte Aufgaben von Hans Gruber

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

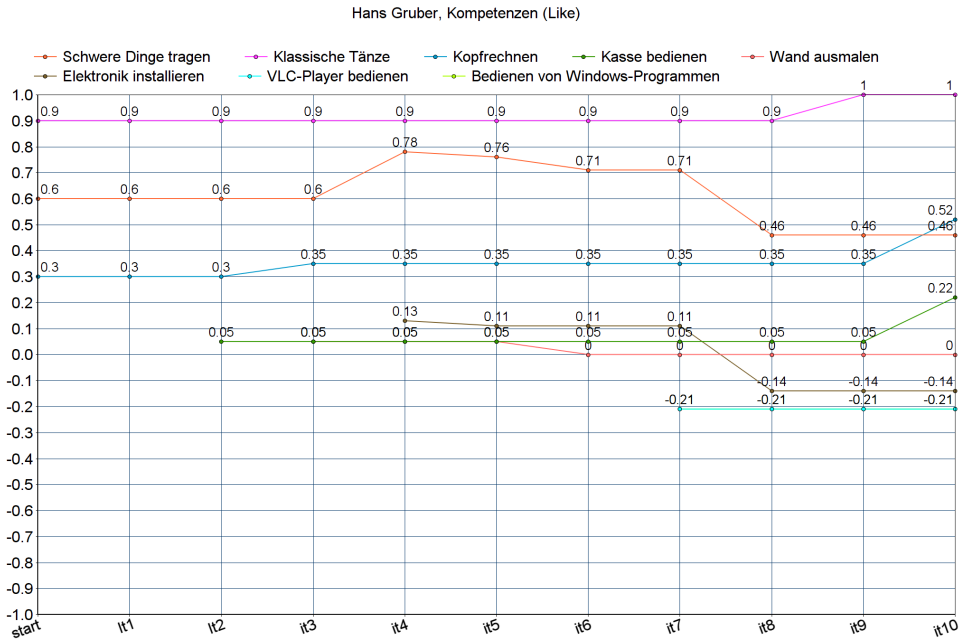


Abbildung 7.6: Veränderung des Competence-Like-Levels von Hans Gruber

	it1	it2	it3	it4	it5	it6	it7	it8	it9	it10
Task	10	10	10	9	8	7	10	6	8	10
User-Eval	-0.1	0.4	-0.2	-0.55	0.85	-0.05	0.2	-0.25	0.05	-0.65
Task-Eval	0.25	-0.7	1	0.95	0.35	-0.35	-0.5	-0.65	0.85	-0.65
User No.	5	3	1	4	3	5	1	4	2	5

Tabelle 7.2: Absolvierte und evaluierte Aufgaben von Ernst Blofeld

Im Fall dieses Tests macht es am meisten Sinn, einen Blick auf die Auswirkungen der Evaluierung zu werfen. Zunächst ist hier schön sichtbar in allen Abbildungen zu erkennen, wann neue Kompetenzen gelernt und andere Benutzer kennen gelernt wurden. In den Grafiken 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 – diese betreffen allesamt Like-Levels, entweder zu anderen Usern oder Kompetenzen – genau mit den positiven und negativen Evaluierungen beider Freiwilliger in Verbindung gebracht werden.

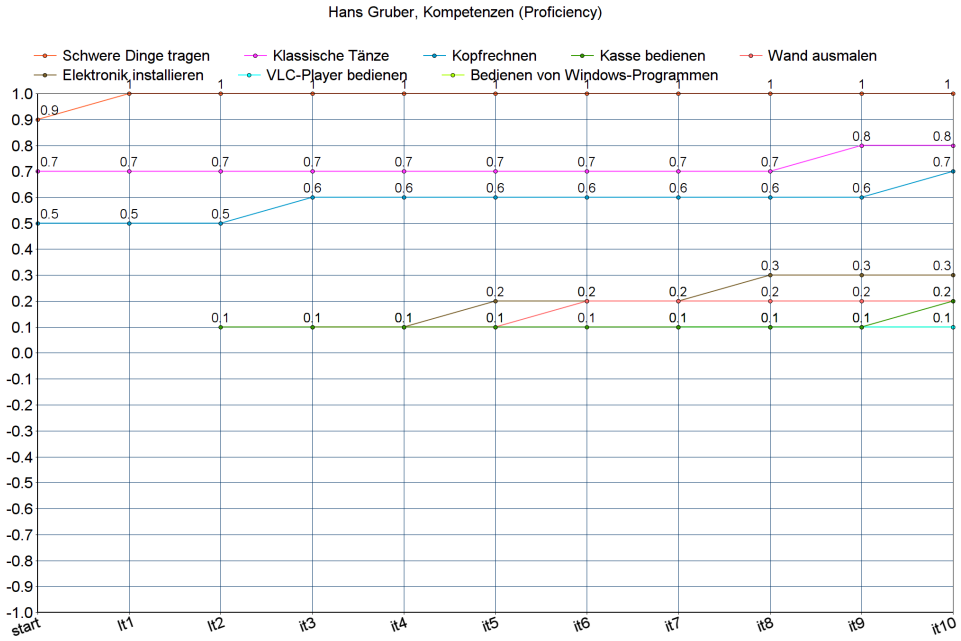


Abbildung 7.7: Veränderung des Competence-Proficiency-Levels von Hans Gruber

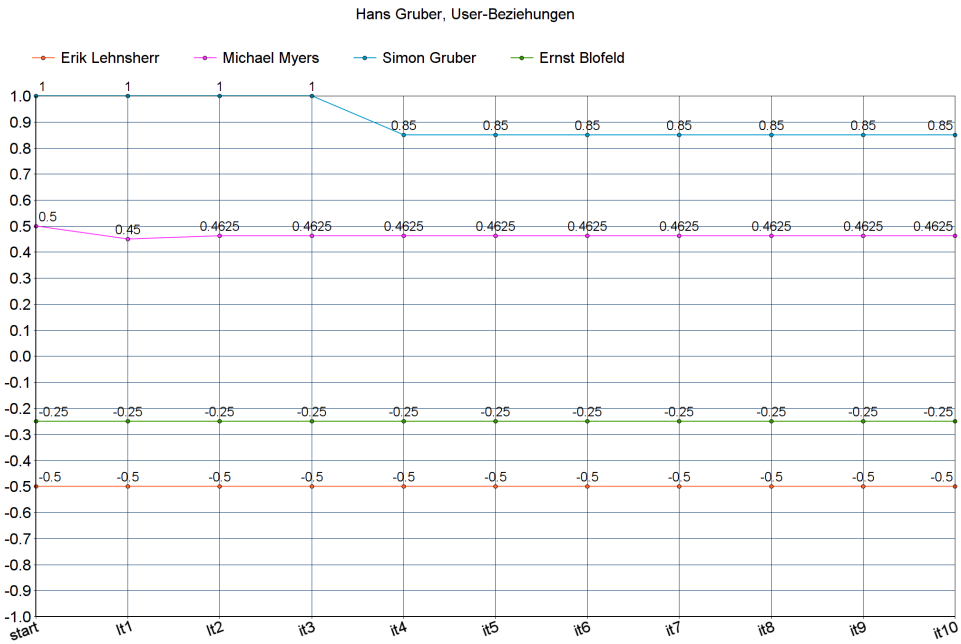


Abbildung 7.8: Veränderung des User-Like-Levels von Hans Gruber

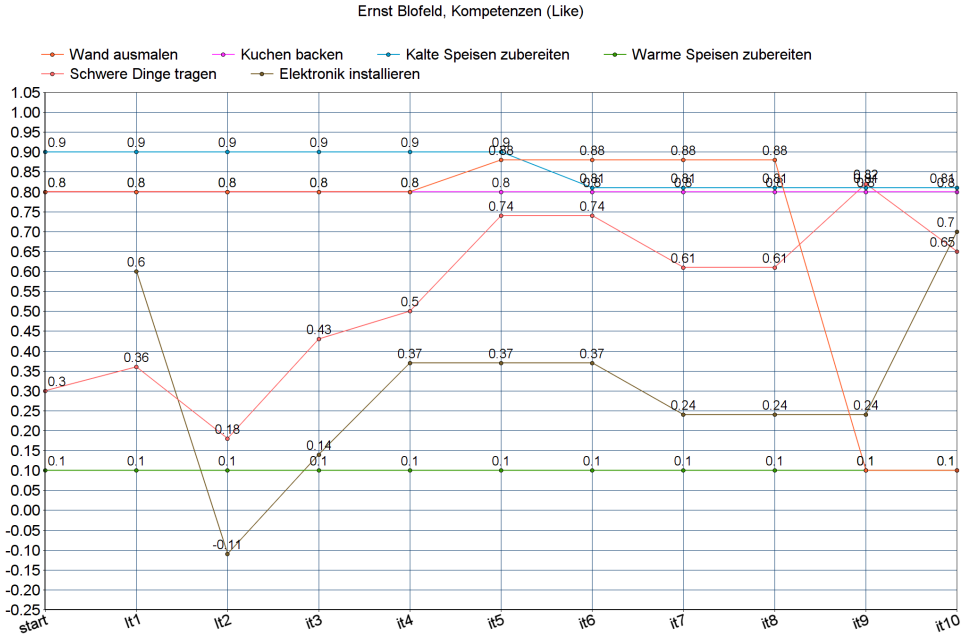


Abbildung 7.9: Veränderung des Competence-Like-Levels von Ernst Blofeld

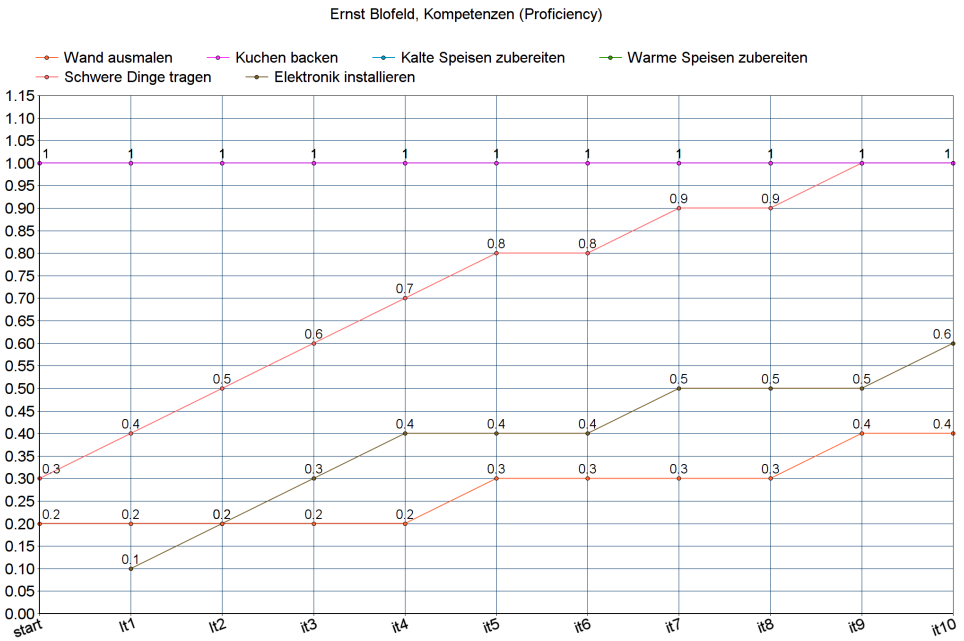


Abbildung 7.10: Veränderung des Competence-Proficiency-Levels von Ernst Blofeld

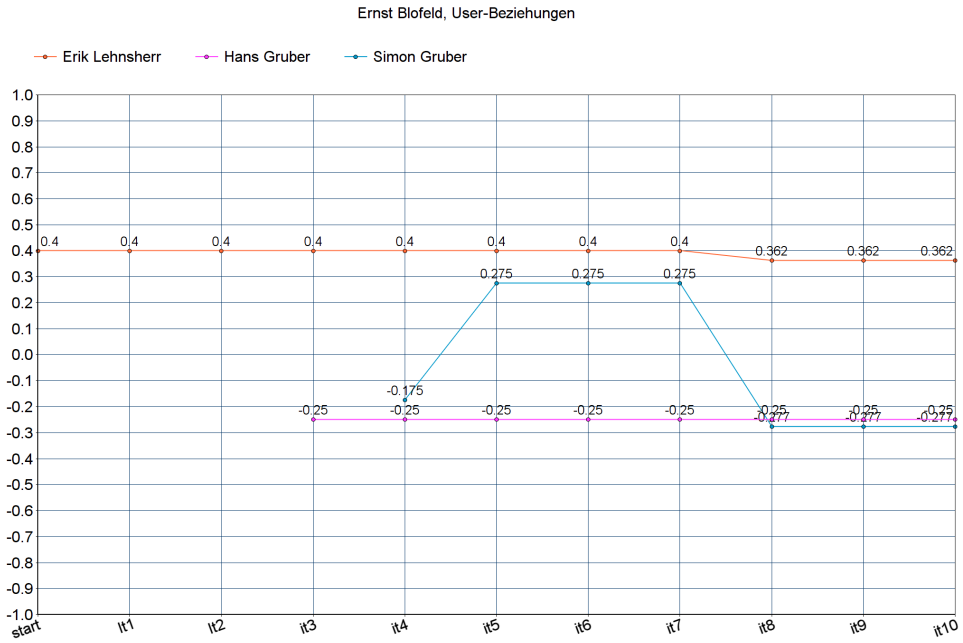


Abbildung 7.11: Veränderung des User-Like-Levels von Ernst Blofeld

7.3 Mögliche Adaptionen

Zusammenfassend kann – basierend auf den Ergebnissen – gesagt werden, dass der CrAc-Core seine Aufgabe, nämlich das sich dynamisch weiterentwickelnde Matching mithilfe des *Matching-Evaluation-Loops*, erfüllt. Die Filter nutzen die Meta-Daten auf die gewünschte Weise aus 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.
Zuvor noch eine *wichtige Anmerkung*: Die in dieser Sektion genannten Punkte sind in der für die Thesis verwendeten Version des CrAc-Cores nicht umgesetzt, 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 von 1. Während ein festgelegter Wertebereich – insbesondere beim Persistieren der Daten – Sinn macht, verliert man beim Matching wertvolle Informationen. Eine Aufgabe mit einem potentiellen Endergebnis von 1.5 sollte über einer Aufgabe mit dem potentiellen Endergebnis von 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 direkten Modifikation der Matrix-Felder durch die Matrix-Filter. Dies führt dadurch, dass die Filter bereits modifizierte Felder weiter verändern, schnell zu sehr großen oder sehr kleinen Werten. Würden die Filter jeweils auf dem Basis-Score arbeiten und ihre Ergebnisse im Nachhinein kombinieren könnte 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 hilfreich ist und 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 zugewiesen Kompetenzen funktionieren.

Einbezug von Ortsdaten ins Matching

In die gleiche Kategorie – dem nachträglichen 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 verwender 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 potentielle Frust-Faktor bei betroffenen Freiwilligen sollte bei einem automatisierten Verlernen nicht vernachlässigt werden.

Kapitel 8

Conclusion

Anhang A

Technical Details

Anhang B

CD-ROM/DVD Contents

Anhang C

Change History

Anhang D

LaTeX Source Code

Quellenverzeichnis

Literatur

[1] Chuck Allen. *Competencies (Measurable Characteristics) Recommendation*. visited last on 5.1.2017. Feb. 2004. URL: www.ec.tuwien.ac.at/~dorn/Courses/KM/Resources/hrxml/HR-XML-2_3/CPO/Competencies.html (siehe S. 15).

[2] *Apache Jena*. visited last on 8.5.2017. 2017. URL: https://jena.apache.org/ (siehe S. 9).

[3] Soziales und Konsumentenschutz Bundesministerium für Arbeit. *Freiwilliges Engagement in Österreich*. 2012. URL: http://www.freiwilligenweb.at/sites/default/files/fwe_in_oe_-_bundesweite_bevoekerungsbefragung_2012.pdf (siehe S. 2).

[4] *Cooperative Activities*. visited last on 15.3.2017. 2017. URL: http://www.crac.at/crac-cooperative-activities-1 (siehe S. 53).

[5] *DB-Engines*. visited last on 10.1.2017. 2017. URL: http://db-engines.com/en/ranking/search+engine (siehe S. 30, 56).

[6] *Elasticsearch*. visited last on 10.1.2017. 2017. URL: https://www.elastic.co/de/products/elasticsearch (siehe S. 9).

[7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm/ (siehe S. 9, 53).

[8] Bin Zhu Hexin Lv. *Skill Ontology-based Semantic Model and its Matching Algorithm*. 2006. URL: http://ieeexplore.ieee.org/abstract/document/4127158/?reload=true (siehe S. 31, 32, 35, 37, 38).

[9] B. Proell J. Schoenboeck M. Raab und W. Schwinger u.a. *A Survey on Volunteer Management Systems*. 2016. URL: http://www.crac.at/media/files/hicss_new.pdf (siehe S. 2).

[10] *Java Annotations*. visited last on 8.5.2017. 2017. URL: https://docs.oracle.com/javase/tutorial/java/annotations (siehe S. 7).

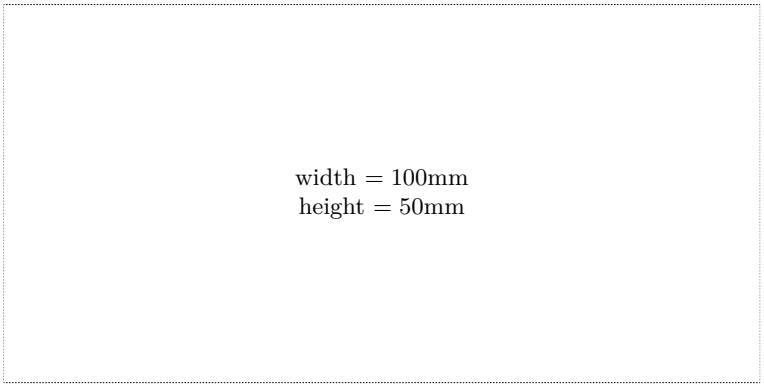
[11] *Kompetenzraster-Erfassungs-Tool*. visited last on 15.3.2017. 2017. URL: https://gtn-solutions.com/home/komet/ (siehe S. 54).

[12] Markus Raab. *A prototypical approach for a competency-based task allocation system in the context of voluntary organizations*. 2016 (siehe S. 32, 41, 44).

[13] *Spring-Autowired*. visited last on 12.4.2017. 2017. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html (siehe S. 59).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —