Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Florian Johannßen

Ein generisches Matching-Modul in Scala für die flexible Integration in Marketplace-Systeme

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Florian Johannßen

Ein generisches Matching-Modul in Scala für die flexible Integration in Marketplace-Systeme

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Michael Böhm
Zweitgutachter : Prof. Dr. Birgit Wendholt

Abgegeben am 22. August 2011

Florian Johannßen

Thema der Bachelorarbeit

Ein generisches Matching-Modul in Scala für die flexible Integration in Marketplace-Systeme

Stichworte

Marketplace, Matching, Scala, Tupelraum, Metamodelle, parallele Algorithmen, Lift, REST, Webservices

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit dem Matching-Mechanismus für Marketplace-Systeme, indem Angebots- und Nachfrageprofile, die zueinander passen einander zugeordnet werden. Hierzu wurde ein Matching-Modul in der Programmiersprache Scala entwickelt, welches die Ähnlichkeiten der Profile anhand eines parallelen und mathematisch basierten Algorithmus berechnet. Bei der Implementierung wurde besonderen Wert auf die generische Eigenschaft des Moduls gelegt, indem durch eine generische Schnittstelle eine lose Kopplung zwischen Matching-Problem und Algorithmus erreicht wird.

Florian Johannßen

Title of the paper

A generic matching module in Scala for the integration in marketplace systems

Keywords

Marketplace, matching, Scala, Tuple Space, Metamodell, parallel Algorithm, Lift, REST, Web services

Abstract

This thesis deals with the matching problem in marketplace systems to match profiles of suppliers and demanders. For this purpose a matching module was implemented in Scala, which computes the similarity of theses profiles by using a parallel and mathematical-based algorithm. The module has a generic interface and is therefore independent from a client specific matching problem.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Ausgangssituation

Heutzutage findet der Austausch von Informationen, Waren und Dienstleistungen überwiegend im Sinne des E-Business online statt. Unter E-Business versteht man nach (Zimmermann, 2010) alle Formen von Geschäftsprozessen, die elektronisch abgebildet werden können, wie E-Commerce[1], E-Consulting[2], E-Intermediary[3], E-Payment[4] und E-Marketplace[5].

Diese Arbeit beschäftigt sich ausgiebig mit dem Begriff des E-Marketplace. Dieser stellt einen elektronischen Ort dar, welcher ermöglicht, dass Anbieter und Nachfrager über das Internet miteinander Informationen, Waren oder auch Dienstleistungen austauschen. Dieser Marketplace ist sowohl für die Verwaltung der Angebote und Nachfragen verantwortlich, als auch für das Matching[6] derjenigen Paare, die zueinander passen. Dabei stellt der Matching-Prozess den Kern eines Marketplace dar. Die zueinander passenden Angebots-Nachfrage Paare werden ermittelt, indem ein Matching-Mechanismus die Profilbeschreibungen beider miteinander vergleicht und die Ähnlichkeit zwischen ihnen berechnet.

Ein System, welches einen elektronischen Marktplatz verwendet, wird im Folgenden als Marketplace-System bezeichnet. Mit Hilfe solcher Systeme können folgende Beziehungen über den Marketplace abgebildet werden:

- Business-To-Business[7], wie Rohstoffhandel

[1] Elektronischer Handel

[2] Elektronische Beratung

[3] Elektronische Vermittlung

[4] Elektronische Bezahlung

[5] Elektronischer Marktplatz

[6] Zueinander passende Profilbeschreibungen finden

[7] Online-Handel zwischen Unternehmen

- Business-To-Consumer[8], wie Online-Banking, Online-Shops
- Consumer-To-Consumer[9], wie E-Bay, Dating-Services

Im IT-Beratungsunternehmen AKRA GmbH wird das Marketplace-System *SC-EEN* vertrieben. Dieses System ermöglicht es Firmen, sich bei Konferenzen anzumelden und ihre Angebots- und Nachfrageprofile zu veröffentlichen. Anhand dieser Profile werden durch einen Matching-Prozess diejenigen Firmen zusammen gebracht, die einen hohen Ähnlichkeitswert vorweisen können.

Des Weiteren hat die moderne Programmiersprache Scala in den letzten Jahren nach (Odersky, 2010) viel Aufmerksamkeit erregt, indem es die Paradigmen objektorientierte Programmierung und funktionale Programmierung in einer sehr eleganten Form miteinander vereint. Scala bietet Konzepte wie dem Aktor-Modell, um skalierbare und parallele Anwendungen auf einem sehr hohen Abstraktionsniveau zu realisieren. Ein weiterer wichtiger Grund, dass Scala mehr und mehr ins Rampenlicht rückt, ist nach (Arno Haase, 2010) die Interoperabilität mit der weitverbreiteten Programmiersprache Java, da Scala eine Programmiersprache ist, die von der Java virtuellen Maschine interpretiert wird. Die in diesem Abschnitt beschriebene Entwicklung, sowohl im Bereich des elektronischen Handelns als auch im Bereich der modernen Programmiersprachen, stellt die Beweggründe dar, um sich mit dem Thema Matching in Marketplace-Systemen mithilfe der Programmiersprache Scala zu beschäftigen.

1.2 Problemstellung

Die meisten Marketplace-Systeme implementieren diesen Matching-Prozess auf der Datenbankebene, speziell für ihr jeweiliges Anwendungsproblem angepasst, um eine bestmögliche Performanz zu erreichen. Dies hat zur Folge, dass eine sehr enge Kopplung zwischen dem Matching-Mechanismus und den Metadaten in Kauf genommen werden muss. Die Metadaten beschreiben die spezifischen Matching-Daten eines Anwendungsproblem. So würden die Metadaten einer Partnervermittlung anders strukturiert sein, als die Daten eines Job-Systems, bei dem Stellenanzeigen mit Bewerbern verglichen werden.

Die Problemstellung dieser Arbeit umfasst die Aufgabe, ein Matching-Modul in der Programmiersprache Scala zu implementieren, welches unabhängig von den Metadaten des Anwendungsproblems die Funktionalität des Matchings gewährleistet. Die Berechnung der Ähnlichkeiten zwischen den Angebots- und Nachfrageprofilen wird auf ein mathematisch basierendes Modell zurückgeführt. Hierbei soll die Art des Matching-Problems durch eine generische Schnittstelle eine

[8] Online-Handel zwischen Unternehmen und Verbraucher

[9] Online-Handel zwischen Verbrauchern

irrelevante Bedeutung bekommen, um eine möglichst geringe Abhängigkeit zwischen dem Anwendungsproblem und dem Matching-Algorithmus zu erreichen.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, ein Modul zu entwickeln, welches den Matching-Mechanismus generisch implementiert, so dass es unabhängig vom Anwendungsgebiet eingesetzt werden kann.

Um dieses Ziel zu erreichen, wird das Matching-Modul auf der Logikebene implementiert. Dadurch, dass die Implementierung auf der Logikschicht ineffizienter ist als die auf der Datenbankebene, wird parallele Programmierung eingesetzt, um die Performanz-Nachteile möglichst gering zu halten und letztendlich auch ein Online-Matching[10] zu ermöglichen.

Dadurch, dass die Qualitätsmerkmale Wiederverwendung, Austauschbarkeit und Flexibilität jeweils mit Performanz im Zielkonflikt stehen, muss man sich für die Implementierung einer dieser Anforderungen entscheiden. Da im Rahmen dieser Arbeit ein Konzept für eine anwendungsunabhängige Lösung des Matching-Problems entwickelt wird, wird nicht versucht die bestmöglichste Performanz für ein konkretes Matching-Problem zu erreichen. Stattdessen werden diejenigen Qualitätsmerkmale umgesetzt, die den generischen Aspekt des Moduls hervorheben.

[10] Kunden können in ihrem Browser den Matching-Prozess starten und die Ergebnisse sehen

1.4 Aufbau der Arbeit

Die Arbeit ist in sieben Kapitel gegliedert und beginnt damit, die theoretischen Grundlagen der behandelten Themengebiete zu erläutern. Anschließend findet eine Analyse statt, in der sowohl die funktionalen Anforderungen als auch die nicht-funktionalen Anforderungen aus Entwicklersicht spezifiziert werden. Im vierten Kapitel wird der Entwurf eines Matching-Moduls vorgestellt. Es handelt es sich hierbei um ein fachliches Konzept, welches beschreibt, wie man ein Matching-Modul gemäß der im *Kapitel 3* spezifizierten Anforderungen entwickelt. Anschließend folgt eine technische Realisierung dieses Konzeptes in der Programmiersprache Scala. Dabei werden die im *Kapitel 2* beschriebenen Grundlagen verwendet, um ein solches Matching-Modul zu implementieren.

Des Weiteren wird beschrieben, wie die Matching-Funktionalität des Moduls von einem technologisch-unabhängigen System verwendet werden kann. Dafür wird eine Systemarchitektur vorgestellt, die das Matching-Modul integriert und durch Hilfsmodule die Benutzbarkeit des Matchings für ein heterogenes Anwendungssystem gewährleistet. Nachdem gezeigt wurde, wie bestimmte Aspekte dieser Architektur in Form eines Matching-System umgesetzt werden können, werden zwei Anwendungsszenarien simuliert und deren Ergebnisse hinsichtlich der Performanz ausgewertet. Die Arbeit endet mit einer Zusammenfassung, in der die Kernaussagen noch einmal aufgeführt werden sowie einem Ausblick, der die Möglichkeiten zur Weiterentwicklung des Matching-Systems aufzeigt.

2 Theoretische Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, um die Entwicklung eines generischen Matching-Moduls anhand paralleler Programmierung in der Programmiersprache Scala nachvollziehen zu können. Dazu werden im ersten Abschnitt die Matching-Grundlagen erläutert, indem nach einer Themeneinführung vor allem auf ein mathematisches Modell eingegangen wird, welches eine Möglichkeit beschreibt, das Matching-Problem für Marketplace-Systeme formal zu definieren. Des Weiteren werden sowohl parallele Programmierparadigmen und Konzepte vorgestellt, als auch die funktionale Programmiersprache Scala. Diese Hilfsmittel können verwendet werden, um solch ein Matching-Modul zu implementieren. Da im *Kapitel 6* eine Möglichkeit gezeigt wird, wie ein technologisch heterogenes Anwendungssystem den Dienst des Matching-Moduls von außen verwenden kann, werden in den letzten beiden Abschnitten dieses Kapitels die Grundlagen von Webservices und Metamodellen erläutert.

2.1 Matching

Grundlegend wird der Begriff *Matching* nach (Vries, 2009) als eine Berechnung der Ähnlichkeit zwischen zwei Angebot- und Nachfrageprofilen beschrieben.

Dabei besteht die Herausforderung aus einer gegebenen Menge von Matching-Profilen, die jeweils anhand von Matching-Kriterien beschrieben werden, darin, herauszufinden welche dieser Angebot- und Nachfrage Paare einander zugeordnet werden können. Ein Matching-Kriterium eines Profils beschreibt entweder eine Suchanfrage oder ein Angebot, welche durch Attribute dargestellt werden. Ein Attribut bildet unabhängig davon, ob es eine Nachfrage oder ein Angebot eines Profils repräsentiert, eine Bezeichnung auf einen Wert ab. Dabei gibt es wesentliche Unterschiede zwischen den Datentypen der Nachfrageattribute, auf die im nächsten Abschnitt näher eingegangen wird. Eine Suchanfrage verfügt zusätzlich über eine Ähnlichkeitsrelation für numerische Attributwerte und Intervalle. Diese Relation gibt an, mit welcher Genauigkeit ein Angebot zu einer gegebenen Suchanfrage passen muss, um zugeordnet werden zu können.

Das Ergebnis eines Matchings von zwei Profilen ist ein numerischer Wert und wird als Matching-Grad bezeichnet. Dieser Matching-Grad repräsentiert typischerweise einen Wert zwischen 0.0 und 1.0. Je höher der Matching-Grad zweier Profile ist, desto besser passen diese zusammen. Da in Marketplace-Systemen die Priorisierung von Profilattributen eine zentrale Rolle spielt, wird jeder Nachfrage eine Gewichtung zugeordnet, die angibt, wie groß der Einfluss dieses Kriteriums in Bezug auf das Gesamtresultat ist.

2.1.1 Datentypen der Attribute

Im Folgenden wird eine Auswahl von möglichen Datentypen der Nachfrageattribute aufgelistet und anhand eines kleinen Beispiels in *Tabelle 1* deutlich gemacht, wie solche Nachfrage- und Angebotspaare aussehen können:

- **Numerische Werte**
  Bei einem Nachfrageattribut, bei dem der Wert vom numerischen Typ ist, wird zusätzlich eine Ähnlichkeitsrelation angegeben, die aussagt, in welcher Art und Weise das Angebot abweichen darf, um dennoch zugeordnet werden zu können.
- **Zeichenketten**
  Ein Attribut dieses Typs, bildet wie in *Tabelle 1* ersichtlich, eine Attributbezeichnung auf eine einfache atomare Zeichenkette ab.
- **Wahrheitswerte**
  Der Attributwert dieses Attributtyps kann entweder den Wert *wahr* oder den Wert *falsch* annehmen.
- **Aufzählungen**
  Eine Aufzählung repräsentiert eine Menge von Zeichenketten, aus der mehrere Werte ausgewählt werden können, was das Beispiel aus *Tabelle 1* verdeutlicht.
- **Freitext**
  Bei Freitextfeldern handelt sich um eine Attributart, die ein semantisches Matching erfordert. Um den Matching-Grad zwischen zwei Texten zu berechnen, können auf Ontologie basierte Verfahren eingesetzt werden.
- **Intervalle**
  Falls das Nachfrageattribut ein Intervall repräsentiert, ist von entscheidender Bedeutung zu wissen, welche Art von Intervall berücksichtigt wird. Dabei wird die Intervallart durch die Ähnlichkeitsrelation definiert.
- **Selektion**
  Die Selektion unterscheidet sich von der Aufzählung in der späteren Berechnung des Matching-Grades darin, dass aus der Auswahl einer Nachfrage nur ein Wert selektiert werden darf und bei einer Aufzählung mehrere Werte ausgewählt werden können. Um ein Matching-Volltreffer von 1.0 zu erzielen, muss der Wert des Angebotsattribut in der Auswahl der Nachfrage enthalten sein.

	Nachfrage			Angebot	
Datentyp	**Attribut**	**Ähnlichkeits-relation**	**Wert**	**Attribut**	**Wert**
Numerisch	Gehalt	>=	3000	Gehalt	3200
Zeichenkette	Land	keine	Deutschland	Land	Deutschland
Wahrheitswerte	Männlich	keine	wahr	Männlich	falsch
Aufzählungen	Hobbys	keine	{Fußball, Golf}	Hobbys	{Tennis, Golf}
Freitext	Hobbys	Verfahren-abhängig	Ich mag Golf	Hobbys	Ich mag kein Golf
Intervalle	Alter	offenes Intervall	[18,26]	Alter	25
Selektion	Hobbys	keine	{Golf, Fußball}	Hobbys	Golf

Tabelle 1: Datentypen der Profil-Attribute

Hierbei ist darauf hinzuweisen, dass die Wahl der richtigen Bewertungsfunktion abhängig vom Typ des Nachfrageattributs ist und beeinflusst damit die Berechnung des Matching-Ergebnisses.

2.1.2 Matching-Grad

Wie schon erwähnt, beschreibt der Matching-Grad die Güte eines Vergleichs zweier Profile und hat großen Einfluss auf das letztendliche Matching-Ergebnis. Um den Matching-Grad dieser zwei Profile zu berechnen, werden die Nachfrage-Attribute des einen mit den Angebots-Attributen des anderen Profils gegenübergestellt und geprüft, ob es gemeinsame Attribute gibt. Sobald Attribute mit gleicher Bezeichnung eines Angebot-Nachfrage Paares gefunden worden sind, kann der Matching-Grad anhand von Bewertungsfunktionen berechnet werden.

Der Matching-Grad zweier Profile setzt sich aus den einzelnen Matching-Graden der gemeinsamen Attribute zusammen. Es ist hervorzuheben, dass ein Matching-Grad zwischen einem Nachfrageattribut und einem Angebotsattribut nur berechnet werden kann, wenn diese Attribute die gleiche Attributs-Bezeichnung vorweisen können.

2.1.3 Mathematisches Modell

Dieses Modell nach (Vries, 2009) stellt eine formale Beschreibung des Matching-Problems dar. Es definiert Bewertungsfunktionen, die verwendet werden um den Matching-Grad eines Angebot-Nachfrage-Paares, zu berechnen.

Mathematisch gesehen besteht die Lösung des Matching-Problems darin, aus den beiden Mengen R = {n | n ist eine Nachfrage} und O = {a | a ist ein Angebot} die Matching-Grade der Angebot-Nachfrage-Paare zu berechnen.

Dabei betrachtet man folgenden Matching-Raum *M*, der das Kreuzprodukt zwischen den Anfragen und Nachfragen darstellt. Er beschreibt eine Menge aller Angebot-Nachfrage-Paare für die jeweils der dazugehörige Matching-Grad berechnet wird.

$$M = O \times R \text{ , wobei } M = \{(a , n) \mid a \in O \land n \in R\}$$

Abbildung 1: Mathematische Formulierung des Matching-Problems

Das mathematische Modell umfasst hauptsächlich eine Menge von Funktionen, die den Matching-Grad eines Angebot-Nachfrage Paares abhängig von den Attribut-Typen berechnet. Das Modell nach (Vries, 2009) berücksichtigt dabei numerische Attribute, Selektionen und Interessenfelder, wobei nur die ersten zwei Attributtypen in dieser Arbeit behandelt werden. Dabei gliedert das Modell die Menge der Nachfrage-Attribute in diese drei Kategorien, was zu drei unterschiedlichen Attribut-Mengen führt, welche folgende Tabelle zeigt.

Attribut-Art	Mengenbezeichnung
Numerische Attribute	*N*
Selektionen	*S*
Interessenfelder	*I*

Tabelle 2: Bezeichnung der Attributmengen

Für diese Arten von Attributen definiert das Modell Bewertungsfunktionen, die als Eingabe die Werte der Attribute entgegennehmen und als Ergebnis den Matching-Grad dieses Attribut-Paares zurückliefern. Als Voraussetzung dafür gilt, dass die Attribut-Bezeichnungen zweier Attribute eines Angebot-Nachfrage Paares identisch sein müssen. Im Folgenden werden die Bewertungsfunktionen, abhängig von den Datentypen der Attribute, mathematisch definiert.

2.1.3.1. Numerische Attributwerte

Dabei ist darauf hinzuweisen, dass dieses Modell ein numerisches Nachfrageattribut intern auf ein Intervall abbildet, was folgende Tabelle verdeutlicht. Die Intervallart ist abhängig von der Ähnlichkeitsrelation des Attributs. Um in *Zeile 1* einen Matching-Grad von 1.0 zu erlangen, muss der Wert des Angebot-Attributs *Berufserfahrung* größer gleich der linken Intervallgrenze des Nachfrage-Attributs sein.

Attribut-Bezeichnung	Attribut-Wert	Ähnlichkeitsrelation	Intervall
Berufserfahrung	5	>=	[5,∞]
Alter	18-22	>= , <=	[18,22]
Postleitzahl	22111	==	[22111,22111]

Tabelle 3: Numerische Attributwerte – Intervallabbildung

Nachdem das numerische Nachfrageattribut auf das entsprechende Intervall abgebildet worden ist, wird dieses Intervall zusammen mit dem Wert des Angebotsattributs der Bewertungsfunktion aus *Abbildung 3* übergeben, um den Matching-Grad zu berechnen.

Fuzzy-Level

Dieses Modell berücksichtigt nicht nur den Volltreffer-Match mit dem Matching-Grad von 1.0, sondern berechnet auch die Ähnlichkeit eines Angebot-Nachfrage Paares, die einen gewissen Grad voneinander abweichen. So wird bei einem Angebot von 4 Jahren Berufserfahrung in *Tabelle 3* zwar nicht ein Matching-Grad von 100 % ermittelt, jedoch wird die Ähnlichkeit dieses Angebot-Nachfrage Paares berücksichtigt und ein Matching-Grad von 0.66 berechnet. Für die Berücksichtigung der Abweichung wird nach dem Modell nach (Vries, 2009) ein Fuzzy[11]-Level $e \in [0.0,1.0]$ hinzugenommen. Je kleiner der Wert, desto negativer wirkt sich die Abweichung zwischen einem Angebot-Attribut und einem Nachfrage-Attribut bezogen auf den Matching-Grad aus. Die folgende Abbildung veranschaulicht, wie das Fuzzy-Level die Intervallgrenzen erweitert.

[11] Ungenau, undeutlich

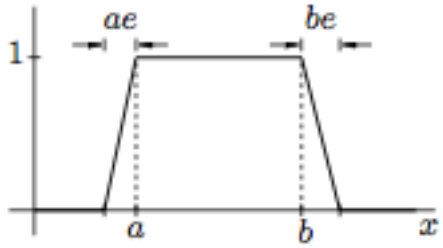

Abbildung 2: Fuzzy-Level e

Bewertungsfunktion

Die folgende Bewertungsfunktion zeigt, wie der Matching-Grad unter der Berücksichtigung des Fuzzy-Levels berechnet werden kann. Ein typischer Wert für *e* wäre 0.2, der in die Berechnung des Matching-Grades mit einfließt. Des Weiteren repräsentiert der Wert *a* die linke Intervallgrenze, der Wert *b* die rechte Intervallgrenze und der Wert *x* den Wert des Angebotsattributs.

$$\boldsymbol{fn}(\boldsymbol{x},\boldsymbol{a},\boldsymbol{b}) = \begin{cases} \dfrac{x}{a\cdot e} - \dfrac{1-e}{e}, (1-e)\cdot a < x < a \\ 1, h(x,a,b) \\ \dfrac{1+e}{e} - \dfrac{x}{b\cdot e}, b < x < (1+e)\cdot b \\ 0, sonst \end{cases}$$

Abbildung 3: Bewertungsfunktion für numerische Werte

Die Funktion *h(x,a,b)* stellt hierbei die Ähnlichkeitsrelation dar, die prüft, ob der Wert x im Intervall [a,b], abhängig von der Intervallart, enthalten ist. Die folgende Abbildung stellt die gängigsten Intervallarten nach (Wikipedia, 2011) grafisch dar.

Intervallart	Notation
offenes Intervall	$(a, b) \coloneqq \{x \in \mathbb{R} \mid a < x < b\}$
abgeschlossenes Intervall	$[a, b] \coloneqq \{x \in \mathbb{R} \mid a \leq x \leq b\}$
rechtsoffenes Intervall	$[a, b) \coloneqq \{x \in \mathbb{R} \mid a \leq x < b\}$
linksoffenes Intervall	$(a, b] \coloneqq \{x \in \mathbb{R} \mid a < x \leq b\}$
linkseitig unendliches abgeschlossenes Intervall	$(-\infty, b] \coloneqq \{x \in \mathbb{R} \mid x \leq b\}$
linkseitig unendliches offenes Intervall	$(-\infty, b] \coloneqq \{x \in \mathbb{R} \mid x < b\}$
rechtsseitig unendliches abgeschlossenes Intervall	$[a,\infty) \coloneqq \{x \in \mathbb{R} \mid a \leq x\}$

rechtsseitig unendliches offenes Intervall	$(a, \infty) := \{x \in \mathbb{R} \mid a < x\}$

Tabelle 4: Intervallarten

Im *Kapitel 2.1.3.5* wird anhand eines Beispiels gezeigt, wie die Bewertungsfunktion aus *Abbildung 3* den Matching-Grad eines Nachfrage-Angebot Paares berechnet.

2.1.3.2. Attribut-Werte als Selektionen

Wie bereits erwähnt, ist es bei einer Selektion, anders als bei einer Aufzählung, nur erlaubt aus einer gegebenen Menge von Werten genau einen Wert auszuwählen.

Attribut-Bezeichnung	Nachfrage	Angebot
Wohnort	{Hamburg, Kiel, Lübeck}	Hamburg
Hobbys	{Fußball, Tennis}	Handball

Tabelle 5: Attributwerte als Selektionen - Beispiel

Diese Eigenschaft beeinflusst auch die Berechnung des Matching-Grades. Sobald der Wert eines Angebot-Attributes in der Wertemenge einer Nachfrage enthalten ist, so wird wie in *Abbildung 4* ersichtlich, ein Matching-Grad von 100 % zurückgegeben. Andernfalls wird ein Fehl-Matching von 0 % verzeichnet. Dabei stellt φ die Wertemenge einer Nachfrage in Form einer Selektion dar und *x* den Wert eines Angebots.

Bewertungsfunktion

$$\boldsymbol{fs}(\boldsymbol{x}, \boldsymbol{\varphi}) = \begin{cases} 1.0 & , x \in \varphi \\ 0.0 & , \text{sonst} \end{cases}$$

Abbildung 4: Bewertungsfunktion für Selektionen

2.1.3.3. Totale Bewertungsfunktion

Der Matching-Grad stellt wie schon im *Kapitel 2.1.2* beschrieben einen Wert dar, der die Güte des Matchings zwischen einer Nachfrage und einem Angebot repräsentiert. Die verschiedenen Berwertungsfunktionen berechnen für ihre spezielle Attributart den Matching-Grad. Um den totalen Matching-Grad zweier Profile *(p1,p2)* zu berechnen, werden die einzelnen Matching-Grade akkumuliert und durch die Anzahl der Attribute dividiert. Folgende Abbildung zeigt die totale Bewertungsfunktion, die intern die Bewertungsfunktionen der jeweiligen Attribut-Arten aufruft und dessen Ergebnisse verarbeitet. Hierbei beschreibt das Modell eine Funktion, wobei alle Attribute gleich gewichtet sind. Die Funktion *fi(r,o)*, die den Matching-Grad für Interessenfelder berechnet, wird hierbei nur der Vollständigkeit halber aufgeführt.

$$\boldsymbol{f(p1,p2)} = \sum_{n \in N} \frac{fn(x,a,b)}{|N|+|S|+|I|} + \sum_{s \in S} \frac{fs(x,\varphi)}{|N|+|S|+|I|} + \sum_{i \in I} \frac{fi(r,o)}{|N|+|S|+|I|}$$

Abbildung 5: Totale Bewertungsfunktion

Gewichtung der Attribute

Da in den meisten Marketplace-Systemen die Attribute einer Nachfrage unterschiedlicher Priorisierung entsprechen, wird in *Abbildung 7* eine Funktion vorgestellt, die die Gewichtung der Attribute berücksichtigt.

$$\boldsymbol{wTotal} = \sum_{\boldsymbol{a} \in \mathbf{N}} \boldsymbol{w(a)} + \sum_{\boldsymbol{a} \in \mathbf{S}} \boldsymbol{w(a)} + \sum_{\boldsymbol{a} \in \mathbf{I}} \boldsymbol{w(a)}$$

Abbildung 6: Gesamtgewichtung aller Attribute einer Nachfrage

$$\boldsymbol{f(p1,p2)} = \sum_{\boldsymbol{a} \in \mathbf{N}} \frac{\boldsymbol{w(a)}}{\boldsymbol{wTotal}} \boldsymbol{fn(x,a,b)} + \sum_{\boldsymbol{a} \in \mathbf{S}} \frac{\boldsymbol{w(a)}}{\boldsymbol{wTotal}} \boldsymbol{fs(x,\varphi)} + \sum_{\boldsymbol{a} \in \mathbf{I}} \frac{\boldsymbol{w(a)}}{\boldsymbol{wTotal}} \boldsymbol{fi(r,o)}$$

Abbildung 7: Totale Bewertungsfunktion mit Gewichtungen

2.1.3.4. Profil-Matching

Im Folgenden wird ein Beispiel zeigen, wie ein Profil-Matching anhand des mathematischen Modells durchgeführt werden kann, wobei die Profile aus einer Menge von Matching-Kriterien bestehen. Wie schon in der Einführung der Matching-Grundlagen beschrieben, unterscheidet man zwischen den Matching-Kriterien Angebot und Nachfrage. Dabei besteht eine Nachfrage jeweils aus einem Attribut, einem Gewicht und einer Ähnlichkeitsrelation. Ein Angebot besteht hierbei nur aus einem Attribut. Anhand der folgenden zwei Abbildungen wird ein Beispiel demonstriert, wie man mittels der Berwertungsfunktionen des mathematischen Modells den entsprechenden Matching-Grad berechnet. Hierfür wird ein Anwendungsfall aus dem Online-Dating hinzugezogen, bei dem die Aufgabe darin besteht, die Ähnlichkeit der Matching-Profile mithilfe von Bewertungsfunktionen zu berechnen.

	Nachfrage			Angebot
Attribut	**Attribut-wert**	**Ähnlichkeits-relation**	**Gewichtung**	**Attributwert**
Alter	[18,22]	offenes Intervall	0.4	20
Geschlecht	weiblich	----------	0.6	männlich

Tabelle 6: Matching- Profil 1

	Nachfrage			Angebot
Attribut	**Attributwert**	**Ähnlichkeitsrelation**	**Gewichtung**	**Attributwert**
Alter	[18,20]	offenes Intervall	0.4	23
Geschlecht	männlich	----------	0.6	weiblich

Tabelle 7: Matching- Profil 2

Um den totalen Matching-Grad dieser zwei Profile zu berechnen, werden zuerst die Matching-Grade der einzelnen Nachfrageattribut-Angebotsattribut Paare berechnet. Im Folgenden wird der Matching-Grad des Paares *(Profil 1,Profil 2)* berechnet. Aufgrund der Asymmetrie des Matchings, erwartet man ein unterschiedliches Ergebnis, wenn man den Matching-Grad des Paares *(Profil 2, Profil 1)* berechnet.

Berechnung des Matching-Grades für das Attribut *Alter*

Da es sich bei diesem Attributtyp um ein Intervall handelt, wird die Bewertungsfunktion aus *Abbildung 3* verwendet, um den Matching-Grad zu berechnen. Wie im *Kapitel 2.1.3.1* beschrieben, wird ein Fuzzy-Level *e* von 0.2 verwendet. Des Weiteren ist die Ähnlichkeitsrelation zu berücksichtigen. Da es sich bei diesem Nachfrageattribut um ein offenes Intervall handelt, gibt die Bewertungsfunktion einen Match-Volltreffer von 1.0 zurück, sobald der Wert des Angebotsattribut ≥ der linken Intervallgrenze und ≤ der oberen Intervallgrenze liegt. Die folgende Abbildung zeigt, wie der Matching-Grad speziell für das Attribut *Alter* berechnet wird.

$$\boldsymbol{fn}(\mathbf{23},\mathbf{18},\mathbf{22}) = \begin{cases} \dfrac{23}{18\cdot e} - \dfrac{1-e}{e}\,, (1-e)\cdot 18 < 23 < 18 \\ 1\,, (18 \le 23 \le 22\,) \\ \dfrac{1+e}{e} - \dfrac{23}{22\cdot e}\,, 22 < 23 < (1+e)\cdot 22 \end{cases}$$

Abbildung 8: Berechnung des Matching-Grades für das Attribut *Alter*

Dadurch, dass der Wert des Angebotsattributs von Profil 2 etwas von der Nachfrage abweicht, wird ein Matching-Grad von *0.7728* zurückgegeben.

Berechnung des Matching-Grades für das Attribut *Geschlecht*

Da es sich bei diesem Attribut um eine Selektion handelt, wird die Bewertungsfunktion aus *Abbildung 4* zur Berechnung des Matching-Grades hinzugezogen.

$$\boldsymbol{fs}(\mathbf{weiblich},\{\mathbf{weiblich}\}) = \begin{cases} 1.0 & , weiblich \in \{\text{weiblich}\} \\ 0.0 & , \text{sonst} \end{cases}$$

Abbildung 9: Berechnung des Matching-Grades für das Attribut *Geschlecht*

Die Funktion $fs(x, \varphi)$ berechnet einen Matching-Grad von 1.0, da der Wert des Angebotsattributs in der Nachfragemenge der Selektion enthalten ist.

Berechnung des totalen Matching-Grades

Hierzu werden sowohl die Matching-Grade der einzelnen Attribut-Paare, als auch die Gewichtungen der Attribute berücksichtigt. Die nachfolgende Abbildung stellt das Matching-Ergebnis dar und zeigt, dass die Beziehung (Profil 1,Profil 2) zu ca. 90.91 % übereinstimmt.

$$\boldsymbol{f}(\boldsymbol{p1},\boldsymbol{p2}) = \frac{0.4}{1.0}\cdot 0.7728 + \frac{0.6}{1.0}\cdot 1.0 = 0.9091$$

Abbildung 10: Totaler Matching-Grad

2.2 Parallele Programmierung

Parallele Programmierung beschreibt nach (Timothy G. Mattson, 2008) ein Programmierparadigma, welches die Intention hat, ein Computerprogramm in einzelne voneinander unabhängige Ausführungseinheiten aufzuteilen, damit diese parallel abgearbeitet werden können, um eine Effizienzsteigerung zu erreichen.

Mittlerweile gibt es nicht mehr viele Möglichkeiten die Rechengeschwindigkeit einer Anwendung zu erhöhen. Traditionelle Strategien wie z.B. die Anzahl der Transistoren pro Chip und die Erhöhung der Taktrate zu erreichen, haben ihre Grenzen durch zu hohen Stromverbrauch und zu hohen Temperaturen erreicht. Immer mehr wird nun darauf Wert gelegt, die Computersysteme mit mehr Prozessorkernen auszustatten, um dadurch die Rechengeschwindigkeit zu erhöhen. Dies führt jedoch dazu, dass die Software, die auf solcher Hardware ausgeführt werden soll, dementsprechend auf die neuen Gegebenheiten der Mehrkernarchitekturen angepasst werden muss, welches eine nicht triviale Aufgabe darstellt.

Da die parallele Programmierung ein riesiges Gebiet umfasst, wird im Rahmen dieser Arbeit nur ein bestimmter Teil dieses Programmierparadigma behandelt. Man kann dieses Thema abhängig vom Abstraktionsniveau betrachten. Je nachdem, ob man ein System auf einer unteren, hardwarenahen Systemebene parallelisieren möchte, wird man gezwungen andere Architekturstile zu verwenden, als wenn man parallele Programmierung auf höherer Abstraktionsebene behandelt. Diese Arbeit beleuchtet die Architekturstile und Konzepte, die benötigt werden, um parallele Systeme auf einem höheren Abstraktionsniveau zu entwickeln.

2.2.1 Architekturstile

Im Folgenden werden einige Architekturstile erläutert, die dazu verwendet werden können, um parallele Anwendungen zu entwickeln.

2.2.1.1. Auftraggeber-Arbeiter

Dieses Architektur-Muster wird hauptsächlich für die Entwicklung paralleler Anwendungen verwendet. Es ist ein sehr beliebtes und anerkanntes Muster, um ein bestimmtes Anwendungsproblem zu parallelisieren. Dabei stellen sowohl der Auftraggeber, als auch die Arbeiter eigenständige Ausführungseinheiten dar. Die Idee dabei ist nach (Prankratius), dass der Auftraggeber eine Aufgabe erhält und diese in kleinere und voneinander unabhängige Teilprobleme zerlegt. Anschließend delegiert er diese Teilaufgaben weiter an seine Arbeiter. Die Arbeiter lösen ihr eigenes Teilproblem und schicken das Ergebnis zum Auftraggeber, der die Zwischenergebnisse zu einem Gesamtergebnis wieder zusammenfügt. Um eine möglichst lose Kopplung zwischen Auftraggeber und Arbeiter zu erreichen, wird wie nach (Sarstedt, 2010) beim Erzeuger-Verbraucher Architekturstil eine Datenstruktur verwendet, über die sie miteinander kommunizieren. Damit wird verhindert, dass

Auftraggeber und Arbeiter direkt miteinander kommunizieren müssen. Dementsprechend verwaltet diese Datenstruktur die Aufgaben und die Zwischenergebnisse. Sobald sich in der Datenstruktur Aufgaben befinden, nehmen sich aktive Arbeiter jeweils eine Aufgabe, bearbeiten diese und schreiben das Zwischenergebnis zurück in den Speicher. Nach (Prankratius) gibt es verschiedene Arten die Datenstruktur zu realisieren, wobei ein Tupelraum oder eine geteilte Warteschlange zu den gängigsten Arten gezählt werden.

2.2.1.2. Linda

Linda ist nach (Heidt, 2005) ein Paradigma zur parallelen und verteilten Programmierung und wurde nach (Michael Scherer) in den 80er Jahren unter dem Namen Linda von Prof. Gelernter an der Yale University entwickelt. Dieses Konzept dient zur Koordination von mehreren Prozessen, die auf einen gemeinsamen Speicher zugreifen. Die Idee ist es, dass die beteiligten Prozesse über einen zentralen Tupelraum miteinander kommunizieren. Der Tupelraum repräsentiert dabei einen zentralen Datenspeicher, indem Werte-Tupel abgelegt, gelesen und entfernt werden können. Ein solches Tupel kann folgende Struktur aufweisen:

- Tupel(„Addiere“,5,5)
- Tupel(„Angebot“,3500)
- Tupel(„Nachfrage“,“Mindestgehalt“,3000)

Der Tupelraum bietet eine asynchrone, anonyme und persistente Kommunikation zwischen Prozessen. Dies hat den Vorteil, dass sich die Kommunikationspartner nicht gegenseitig kennen müssen, da sie nur über den Tupelraum miteinander kommunizieren.

Implementierung eines Tupelraumes

Es gibt bereits mehrere Implementierungen dieses Paradigmas in verschiedenen Programmiersprachen. Die bekanntesten sind die Java-Implementierungen von Sun und IBM. Im Folgenden wird nach (J.M.Joller) anhand einer Grafik skizziert, wie eine Anwendung mit dem Zusammenspiel vom Auftraggeber-Arbeiter Muster und Java Space parallelisiert werden kann.

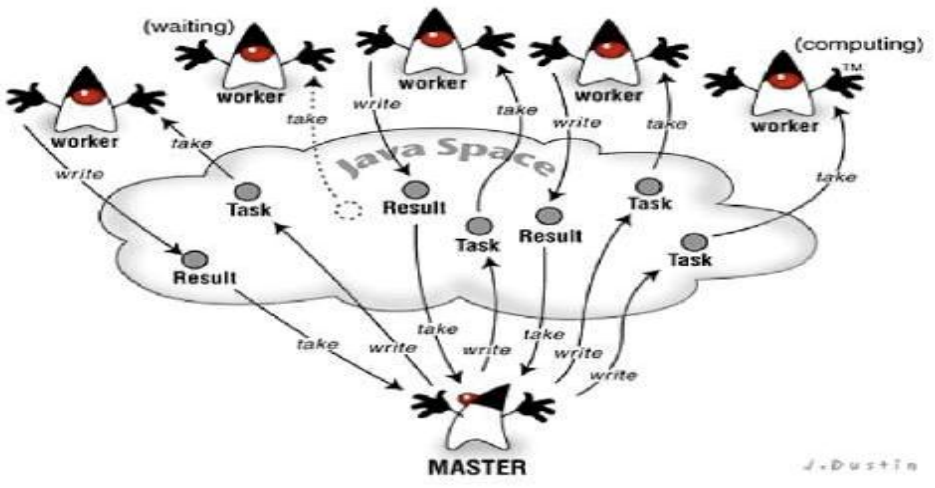

Abbildung 11: Implementierung eines Tupelraumes – Java Space

2.2.2 Aktor-Modell

Das Aktor-Modell ist nach (Esser, 2010) ein ereignisbasiertes und nachrichtenbasiertes Programmiermodell, welches ein Konzept vorstellt, parallele Programmierung auf einem hohen Abstraktionsniveau komfortabel und effizient umzusetzen. Durch die steigende Verfügbarkeit von Mehrkernprozessoren in den letzten Jahren, ist das Aktor-Konzept ein interessantes Modell für parallele Architekturen und wird auch von mehreren Programmiersprachen und Frameworks, wie Erlang, Scala und Akka unterstützt. Jedem Aktor wird eine Identifikationsnummer zugewiesen, die eine logische Adresse repräsentiert, damit sich die Aktoren gegenseitig adressieren können. Des Weiteren beinhalten sie eine Mailbox und einen Verteilungsmechanismus. Die Mailbox stellt eine Warteschlange dar, die die einkommenden Nachrichten von anderen Aktoren empfängt und speichert. Der Verteilungsmechanismus bildet die eingehenden Nachrichten auf das Protokoll des Aktors ab. Dies geschieht über das Konzept des Pattern-Matchings, welches in den Scala-Grundlagen erläutert wird. Da der Nachrichtenaustausch zwischen den Aktoren asynchron stattfindet, müssen die Aktoren nicht empfangsbereit sein. Die eingehenden Nachrichten werden solange in der Mailbox des jeweiligen Aktors gespeichert, bis dieser die Nachricht verarbeiten kann. Das Modell sieht vor, dass die Aktoren nicht gegenseitig auf ihre Zustände zugreifen, um Synchronisationsprobleme zu vermeiden. Ein Aktor besteht nach (Esser, 2010) aus folgenden Elementen:

- **Id**
  Identifikationsnummer eines Aktors
- **State**
  Zustand des Aktors
- **Mailbox**
  Synchronisierte Nachrichtenwarteschlange, in der die empfangenen Nachrichten zur Verarbeitung gelagert werden
- **Thread**
  Jeder Aktor wird durch einen Thread ausgeführt. Dies führt dazu, dass ein Aktor einen leichtgewichtigen Thread repräsentiert und wesentlich effizienter zu erzeugen und zu beenden ist.
- **Nachrichtenprotokoll**
  Die empfangenen Nachrichten werden mittel Pattern-Matching auf das Nachrichtenprotokoll abgebildet.

Die folgende Grafik zeigt nach (Esser, 2010), wie die eben beschriebenen Komponenten einem Aktor zugeordnet werden können und wie dieser mit anderen Aktoren über Nachrichten kommuniziert.

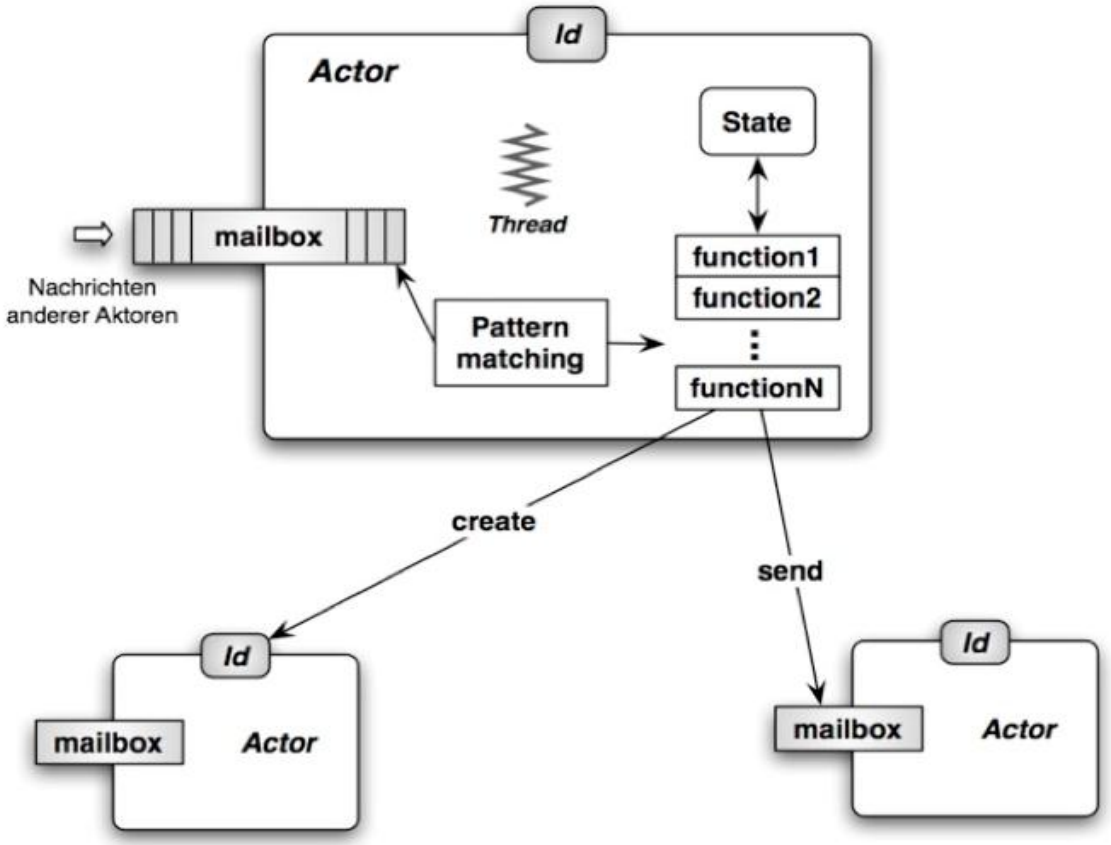

Abbildung 12: Aktor-Modell

2.3 Scala

Die moderne Programmiersprache Scala ermöglicht nach (Odersky, 2009) einen eleganten Weg objektorientierte Programmierung mit der Denkweise des funktionalen Paradigmas zu verschmelzen. Scala ist eine auf der JVM[12] basierende Programmiersprache, die letztendlich wie die Programmiersprachen Java, JRuby oder Closure von der JVM interpretiert werden. Als grundlegendes Prinzip stellt in Scala jede Variable ein Objekt dar. Hierbei wird nicht wie in Java zwischen primitiven Datentypen und Objekten unterschieden. Dadurch, dass Scala funktional inspiriert ist, stellt sie anhand von immutable objects[13], pure-functions[14] und referential transparency[15] die optimalen Voraussetzungen bereit für einfache parallele Programmierung auf einem hohen Abstraktionsniveau.

2.4 Grundlagen – Webservices

Webservices beschreiben eine Technologie, die es einem beliebigen Anwendungssystem ermöglicht einen Dienst über ein standardisiertes Netzwerk-Protokoll anzufordern und mit diesem Dienst Daten auszutauschen. Zur Datenübertragung können standardisierte Transportprotokolle wie HTTP[16] oder FTP[17] verwendet werden. Dadurch, dass die Daten durch standardisierte Beschreibungssprachen wie XML[18] oder Json[19] beschrieben werden und die Kommunikation über solch ein standardisiertes Netzwerkprotokoll geschieht, ist es für ein Anwendungssystem unerheblich zu wissen, in welcher Technologie der Webservice implementiert worden ist und auf welchem Betriebssystem dieser läuft. Für die Realisierung eines Webservice stehen vorwiegend die Technologien Simple Object Access Protocol (SOAP) und das Representational State Transfer (REST) zur Auswahl.

---

[12] Java virtuelle Maschine

[13] Unveränderbare Variablen, die nach der Initialisierung nicht mehr geändert werden dürfen

[14] Eine Funktion mit dieser Eigenschaft erzeugt keine Seiteneffekte

[15] Eine Funktion ist funktional transparent, wenn ihr Ergebnis ausschließlich von den Argumenten beeinflusst wird

[16] Hypertext Transfer Protocol

[17] File Transfer Protocol

[18] Extensible Markup Language

[19] JavaScript Object Notation

2.4.1 REST

REST beschreibt nach (Fielding, 2000) einen Architekturstil zur Realisierung von serviceorientierten Systemen. Dabei repräsentiert REST einen leichtgewichtigen Ansatz zur Realisierung eines Webservices, der auf den etablierten Internetstandards wie HTTP, URL[20] und URI[21] basiert. Es ist ein ressourcenbasierter Ansatz, bei dem jede Ressource, wie ein Bild oder eine XML-Datei über eine eindeutige URL adressierbar und veränderbar ist. Da REST als einziges Kommunikationsprotoll HTTP einsetzt, unterstützt es auch dessen Methoden, wie in *Tabelle 8* dargestellt.

HTTP- Methode	Beschreibung
GET	Lesender Zugriff auf eine Ressource an, dessen Adressierung über eine URL erfolgt
POST	Erzeugen einer neuen Ressource
PUT	Manipulierung einer Ressource
DELETE	Löschen einer Ressource

Tabelle 8:HTTP- Methoden

REST hat sich für die Realisierung des Matching-Systems als Webservice im *Kapitel 6* REST aufgrund dessen Leichtgewichtigkeit und Einfachheit gegenüber SOAP durchgesetzt.

2.4.2 Datenaustauschformate

Die folgenden Datenaustauschformate beschreiben jeweils ein Format für den plattform- und implementationsunabhängigen Austausch von Daten zwischen Systemen heterogener Technologie.

2.4.2.1. Extensible Markup Language

XML ist nach (Liam Quin, 2011) eine Beschreibungssprache für Daten und ist das Standarddatenformat für den Datenaustausch zwischen heterogenen Systemen.

[20] Uniform Resource Locator

[21] Uniform Resource Identifer

2.4.2.2. JavaScript Object Notation

Json bietet nach (1 Wikipedia , 2011) ein einfaches und gut lesbares Datenformat, welches für den Datenaustausch zwischen unterschiedlichen Systemen verwendet werden kann. Damit wird neben XML eine leichtgewichtige Variante vorgestellt, mit der Daten zwischen heterogenen Systemen über ein bestimmtes Transportprotokoll übertragen werden können. Im Vergleich zur Markup-Language XML, ist Json im Wesentlichen nicht so ausdrucksstark. Dessen Stärke besteht darin mittels eines objektorientierten Beschreibungsstils Daten einfach mit geringem Overhead im Json-Format zu definieren.

2.4.3 Jetty

Das Produkt Jetty ist nach (Jetty, 2011) im Rahmen eines open-source Eclipse Projekts entstanden und repräsentiert sowohl einen Anwendungsserver, dessen Aufgabe es ist Programme auszuführen, als auch einen Webserver, der für Beantwortung von Anfragen eines Anwendungssystems über das Internet zuständig ist. In dieser Arbeit wird Jetty verwendet um das in dem *Kapitel 6* beschriebene Matching-System auszuführen und um Anfragen entgegen nehmen zu können.

2.4.4 Lift

Lift repräsentiert nach (Derek Chen-Becker, 2011) ein leistungsstarkes Webframework zur Entwicklung von skalierbaren Webanwendungen, welches vom Webframework Ruby on Rails inspiriert worden ist. Es ist in der Programmiersprache Scala implementiert und bietet daher vollständigen Zugriff auf alle Java-Bibliotheken. Lift bietet mehrere interessante Features, die den Einsatz von Lift begründen, wie REST-Support, First-Class-XML-Support[22] und Json-Support. Dadurch, dass das vorgestellte Webframework einen sehr leichtgewichtigen REST-Support bietet und in der Programmiersprache Scala implementiert worden ist, wird es für die Implementierung des Matching-Systems in *Kapitel 6* als RESTful Webservice verwendet.

2.4.5 Maven

Maven ist nach (Apache , 2011) ein Build-Management-Tool, um insbesondere Java-Programme zu erstellen und verwalten zu können. Es dient dazu größere Systeme mithilfe einer standardisierten Projektstruktur zu organisieren.

[22] XML ist in der Programmiersprache vollständig integriert

2.5 Metamodelle

Ein Metamodell definiert eine bestimmte Struktur zur Spezifikation eines Modells. Ein Modell in der Informatik bezeichnet eine abstrakte Abbildung der Realität. Nach (Schmidt, 2011) stehen Metamodell und Modell in einer Klasse-Instanz Beziehung zueinander. Das Metamodell beschreibt die Struktur des Modells und das Modell selbst stellt eine konkrete Instanz[23] des Metamodells dar. Das Prinzip der Metamodellierung kann eingesetzt werden, um für ein System eine möglichst generische Schnittstelle zu entwickeln. Im Folgenden werden zwei gängige Metamodelle kurz vorgestellt.

BNF

Die Backus-Naur-Form ist eine formale Metasprache zur Definition der Syntax von Programmiersprachen. Dabei bietet sie eine einfache Möglichkeit, Metamodelle zu definieren. So kann nach (Schmidt, 2011) beispielsweise eine in BNF vorliegende Beschreibung der Sprache C++ ein Metamodell dieser Sprache repräsentieren. Jedes C++- Programm wäre eine Instanz dieses Metamodells. In dieser Arbeit wird unter anderem die Erweiterte-BNF verwendet, um Metamodelle zu beschreiben. Eine solche EBNF- Grammatik besteht sowohl aus einer Menge von terminalen und nichtterminalen Symbolen, als auch aus einer Menge von Produktionsregeln, die definieren, aus welchen Symbolen die jeweils linke Seite einer Regel zusammengesetzt ist.

XML-Schema

Eine XSD[24] ist nach (2 Wikipedia, 2011) eine Empfehlung des W3C[25] zum Definieren von Strukturen für XML-Dokumente. Dabei stellen die korrespondierenden XML-Dokumente Modelle dieses Metamodells dar und können gegen diese Spezifikation validiert werden.

---

[23] Ist ein konkretes Objekt einer bestimmten Struktur

[24] XML Schema Definition

[25] Gremium zur Standardisierung von World Wide Web betreffende Techniken

3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen des generischen Matching-Modul spezifiziert. Dabei werden sowohl die funktionalen, als auch die nicht-funktionalen Anforderungen aus Entwicklersicht festgelegt.

3.1 Funktionale Anforderungen

Das Matching-Modul soll:

1. eine in Scala programmierte Schnittstelle bereitstellen, welche die Matching-Daten entgegennimmt.
2. die Struktur der Matching-Daten durch ein internes Metamodell spezifizieren
3. hinsichtlich des internen Metamodels erweiterbar sein
4. sich wie eine funktionale Bibliothek verhalten, was dazu führt, dass es keinen Zustand besitzt. Die Matching-Daten werden entgegengenommen, berechnet und zurückgegeben.
5. ein mathematisch basierendes Matching gewährleisten.
6. ein Matching der Nachfragetypen: numerisch, Intervall, Selektion, Aufzählung, Zeichenkette und Wahrheitswert gewährleisten.
7. ein Matching von Attributen gewährleisten, die zueinander identisch sein müssen
8. ein Matching von Attributen gewährleisten, deren Abweichung voneinander in der Berechnung des Matching-Ergebnisses berücksichtigt wird.
9. ein Matching berücksichtigen, bei dem alle Profile des gleichen Matching-Typs miteinander verglichen werden.
10. ein n : m Matching gewährleisten , bei dem n- Profile eines Matching-Typs mit m- Profilen eines anderen Matching-Typs verglichen werden. Dabei ist zu berücksichtigen, dass der Matching-Grad nur für diejenigen Profile berechnet wird, die sich im Typ voneinander unterscheiden.

3.2 Nichtfunktionale Anforderungen

Das Matching-Modul soll:

1. Skalierbar hinsichtlich der Matching-Anfrage sein.
2. Wiederverwendbar und integrierbar sein.
3. ein Online-Matching ermöglichen, so dass die Ergebnisse in Echtzeit dem Kunden im Browser angezeigt werden können.

4 Entwurf

In diesem Abschnitt wird der Entwurf eines Matching-Moduls vorgestellt. Im Folgenden wird gezeigt, mit welchen Architekturstilen und Konzepten man einen generischen, mathematisch basierenden und parallelen Matching-Algorithmus gemäß den Anforderungen aus *Kapitel 3* entwerfen kann. Um den generischen Aspekt des Matching-Moduls zu realisieren, wird ein Metamodell, siehe *Kapitel 2.5*, eingeführt, welches in einer EBNF die Struktur der Ein- und Ausgabedaten spezifiziert. Das Matching-Modul stellt eine zentrale Komponente dar, welche für die Berechnung der Matching-Ergebnisse verantwortlich ist. Für die Berechnung wird das mathematische Modell aus *Kapitel 2.1.3* als Grundlage verwendet und an die spezifischen Anforderungen aus *Kapitel 3* angepasst. Um einen Algorithmus zu implementieren, der ein paralleles Matching auf hohem Abstraktionsniveau ermöglicht, werden die parallelen Architekturstile Auftraggeber-Arbeiter und Tupelraum aus dem *Kapitel 2.2.1* zusammen mit dem Aktor-Modell aus *Kapitel 2.2.2* verwendet.

4.1 Schnittstelle

Dieses Modul bekommt als Eingabe die Matching-Daten in Form einer Collection[26], gemischt aus Nachfrage- und Angebots-Tupel und matcht[27] die zueinander passenden Profile mithilfe des in *Abbildung 18* spezifizierten Algorithmus. Die Schnittstelle des Matching-Moduls ist abhängig von einem internen Metamodell, da es die Struktur der Matching-Daten spezifiziert.

Die Schnittstelle des Matching-Moduls wird durch die Methode `matching(...)` repräsentiert und nimmt die Matching-Daten entgegen:

- matching(Liste[Matching-Kriterium]) → Liste[Matching-Ergebnis]

Welche Struktur sich hinter dem abstrakten Datentypen Matching-Kriterium und Matching-Ergebnis verbirgt, wird im internen Metamodell definiert.

[26] Repräsentiert eine Menge aus Elementen

[27] Ähnlichkeiten zwischen zwei Objekten ermitteln

Vorbedingungen

- Die Gewichtungen der einzelnen Nachfrage-Tupel müssen einen Wert zwischen 0.0 und 1.0 aufweisen. Die Summe der Attributgewichtungen eines Profils muss dem Wert 1.0 entsprechen.

- Die Struktur der Eingabedaten müssen dem internen Metamodell entsprechen.

Nachbedingungen

- Die zueinander passenden Profile werden mit dem berechneten Matching Modul als Matching-Ergebnisse zurückgegeben, welche der Struktur der Ausgabedaten vom internen Metamodell entsprechen müssen.

- Die berechneten Matching-Grade der Ergebnisse müssen größer als 0.0 sein, da alle Matching-Ergebnisse mit dem Matching-Grad 0.0 vorher herausgefiltert werden und nicht zurückgegeben werden.

4.2 Struktur des internen Metamodells

Das interne Metamodell spezifiziert anhand einer EBNF-Grammatik die Struktur der Ein-und Ausgabedaten des Matching-Moduls.

4.2.1 Eingehende Daten

Die folgende Tabelle beschreibt die Struktur der Matching-Daten, welche die Matching-Schnittstelle entgegen nimmt. Dabei werden in der *Tabelle 9* mehrere Produktionsregeln gemäß einer EBNF-Grammatik aufgelistet, um zu zeigen aus welchen nichtterminalen und terminalen Symbolen der jeweils links stehende Typ besteht. Hierbei wird die typische EBNF-Notation verwendet.

Linker EBNF-Teil	Rechter EBNF-Teil
Matching-Kriterien	{Matching-Kriterium}[28]
Matching-Kriterium	Nachfrage |[29] Angebot
Nachfrage	Matching-Typ Profilname ID Attributbezeichnung Gewichtung Nachfrageinhalt [Ähnlichkeitsrelation][30]

[28] beliebig viele

[29] oder

Nachfrageinhalt	Numerisch | Aufzählung | Selektion | Intervall | wahrheitswert | zeichenkette
Angebot	Matching-Typ Profilname ID Attributbezeichnung Angebotsinhalt
Angebotsinhalt	Numerisch | Aufzählung | wahrheitswert | zeichenkette
Ähnlichkeitsrelation	NumerischeÄhnlichkeitsRelation | IntervallÄhnlichkeitsRelation
Matching-Typ	zeichenkette
Numerisch	fließkommazahl | ganzzahl
Aufzählung	zeichenkette zeichenkette {zeichenkette}
Selektion	zeichenkette zeichenkette {zeichenkette}
Intervall	LinkeGrenze RechteGrenze
LinkeGrenze	Numerisch
RechteGrenze	Numerisch
Abttributbezeichnung	zeichenkette
Gewichtung	Numerisch
Profilname	zeichenkette
ID	ganzzahl
NumerischeÄhnlichkeis Relation	größerAls | größerGleich | kleinerAls | kleinerGleich | gleich
IntervallÄhnlichkeits Relation	offen | geschlossen | linksGeschlossenRechtsOffen | linksOffenRechtsGeschlossen | linksOffen | linksGeschlossen | rechtsOffen | rechtsGeschlossen

Tabelle 9: Struktur der eingehenden Matching-Daten in EBNF

Dabei ist darauf hinzuweisen, dass eine Ähnlichkeitsrelation nur bei numerischen Nachfrageattributen oder Intervallen angegeben werden muss. Nach diesem Metamodell bestehen sowohl Nachfrage, als auch Angebot aus einem Matching-Typ, der beschreibt, ob es sich beim jeweiligen Matching-Kriterium z.B. um eine Firma,

[30] optional

einen Bewerber oder um ein Produkt handelt. Diese Information ist wichtig, um die Anforderungen *9* und *10* aus *Kapitel 3.1* zu realisieren. Da die Attributtypen Selektion und Intervall bei einem Angebot keinen Sinn machen, unterscheidet das Metamodell zwischen Nachfrageinhalt und Angebotsinhalt.

Die folgende Abbildung zeigt ein Beispiel, wie eine solche Instanz der EBNF aus *Tabelle 9*, unabhängig von der Programmiersprache, aussehen kann. Sie zeigt eine Liste, die dem Matching-Modul über dessen Schnittstelle zur Berechnung übergeben werden kann. Die Validierung, ob eine Scala-Instanz dem internen Metamodell entspricht, kann durch einen Compiler erfolgen. Aus Gründen der Übersichtlichkeit werden Abkürzungen für die Matching-Typen *Bewerber: B* und *Firma: F* eingeführt.

Liste[Nachfrage(B,Gates,3,Gehalt,0.3,3500,>=),Nachfrage (B,Kay,4,Gehalt,1.0,2500,>=),

Nachfrage(B,Gates,3,Beruf,0.7,Programmierer,==), Nachfrage (B,Kay,4,Beruf,0.7,Designer,==),

Nachfrage(F,AKRA,1,Beruf,0.6,Programmierer,==),Nachfrage(F,Dynport,2,Beruf,0.6,Designer,==),

Nachfrage(F,AKRA,1,Alter,0.1,[18,35],offenesIntervall),Nachfrage(F,Dynport,2,Alter,0.1,30,>=),

Nachfrage(F,AKRA,1,Sprachen,0.3,{C#,Scala}),Nachfrage(F,Dynport,2,Sprachen,0.3,{Java,Scala,C},

Angebot(F,AKRA,1,Beruf,Programmierer),Angebot(F,AKRA,1,Gehahlt,3000),

Angebot(F,Dynport,2,Beruf,Designer),Angebot(F,Dynport,2,Gehalt,2800),

Angebot(B,Gates,3,Alter,56),Angebot(B,Kay,4,Alter,22)

Angebot(B,Gates,3,Beruf,Programmierer),Angebot(B,Kay,4,Beruf,Designer)

Angebot(B,Gates,3,Sprachen,{C#,F#}),Angebot(B,Kay,4,Sprachen,{Java,Scala})**]**

Abbildung 13: Eingabedaten – Beispiel

4.2.2 Ausgehende Daten

Die folgende EBNF beschreibt die Struktur der Matching-Ergebnisse, die das Matching-Modul zurückgibt.

Linker Teil der EBNF	Rechter Teil der EBNF
Matching-Ergebnisse	{Matching-Ergebnis}
Matching-Ergebnis	Profil Profil Matching-Grad
Profil	Profilname ID
Profilname	zeichenkette
ID	ganzzahl
Matching-Grad	ganzzahl | fließkommazahl | zeichenkette

Tabelle 10: Struktur der Ausgabedaten in EBNF

Wie in der *Tabelle 10* dargestellt, besteht ein Matching-Ergebnis aus zwei Profilen mit dem dazugehörigen Matching-Grad. Das Matching-Modul berechnet diese Matching-Ergebnisse und liefert als Resultat eine Liste dieser Ergebnisse zurück.

Matching-Ergebnis(AKRA,ORACLE,0.66)
Matching-Ergebnis(AKRA,Lufthansa-Systems,1.0)
Matching-Ergebnis(Sabine,Peter,0.98)
Matching-Ergebnis(Hans,Fernseher1,0.24)

Abbildung 14: Beispiel Matching-Ergebnisse

4.3 Interne Sichtweise des Matching-Moduls

Das Modul besteht aus vier Klassen, die in ein Modul gekapselt sind, um gemeinsam die Funktionalität des Matching zu erfüllen.

- Auftraggeber (Eine Instanz)
- Arbeiter (n- Instanzen)
- Tupelraum (Eine Instanz)
- Mathematisches Modell (Utility Klasse[31])

Die interne Sichtweise des Matching-Moduls ist in *Abbildung 15* in Form eines Kompositionsstrukturdiagramms grafisch dargestellt. Sowohl der Auftraggeber, als auch der Arbeiter und der Tupelraum sind als eigenständige Ausführungseinheiten mittels Aktoren modelliert. Die Kommunikation zwischen ihnen geschieht nachrichtenbasiert gemäß dem Aktor-Modell aus dem *Kapitel 2.2.2*.

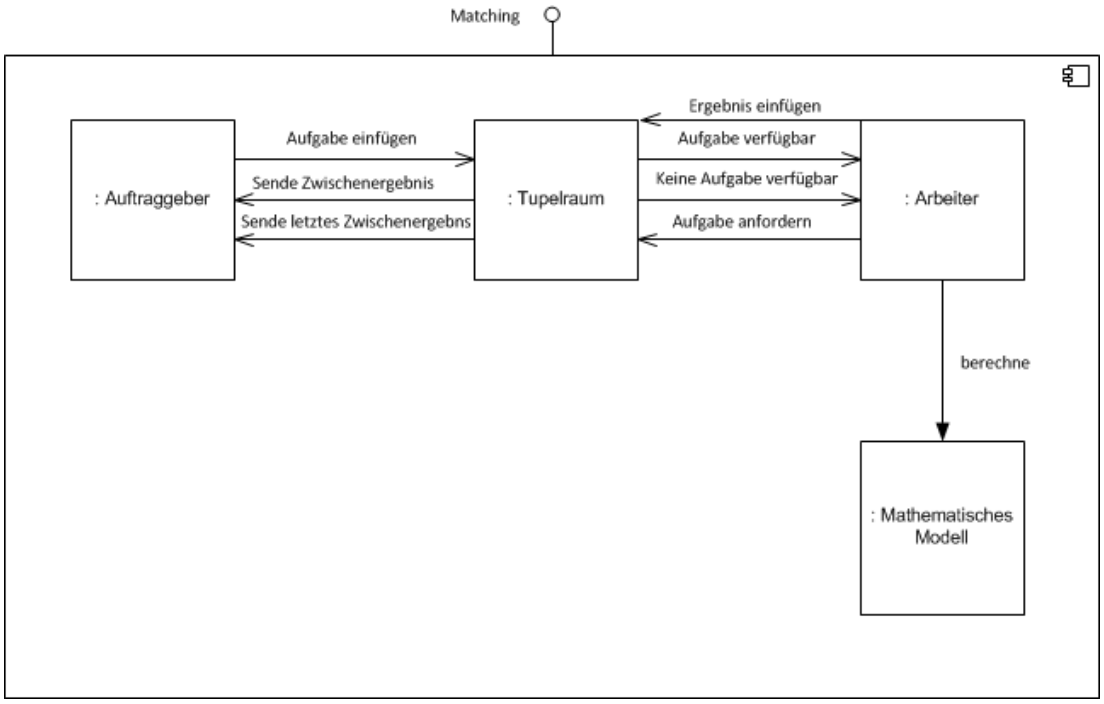

Abbildung 15: Interne Sichtweise des Matching-Moduls

[31] Können nicht instanziiert werden und beinhalten eine Menge von definierten Methoden

Wird die öffentliche Methode `matching(...)` des Matching-Moduls aufgerufen, werden Instanzen vom Auftraggeber, Arbeiter und Tupelraum erzeugt. Das Matching-Modul erzeugt vom Auftraggeber und vom Tupelraum genau eine Instanz, wobei von der Arbeiterklasse abhängig von der Anfragengröße mehrere Arbeiter instanziiert werden. Nachdem die Aktoren gestartet und mit Daten versorgt worden sind, wartet das Matching-Modul auf das Matching-Ergebnis. Der Tupelraum verwaltet sowohl die Aufgaben, als auch die Zwischenergebnisse. Da das Matching-Modul dem Auftraggeber und den Arbeitern bei der Instanziierung den Tupelraum übergibt, geschieht die Kommunikation zwischen diesen beiden Aktoren ausschließlich über den Datenspeicher. Dies hat den Vorteil, dass sich Auftraggeber und Arbeiter nicht kennen müssen und dadurch eine sehr geringe Kopplung erreicht wird.

Um die Kommunikation zwischen diesen Aktoren zu verdeutlichen, wird im nächsten Unterkapitel beschrieben, wie das Auftraggeber-Arbeiter-Muster und das Tupelraum-Muster zusammen mit dem Aktor-Modell umgesetzt worden ist.

4.3.1 Auftraggeber – Tupelraum - Arbeitnehmer

Der Auftraggeber hat die Aufgabe die Matching-Daten von der Schnittstelle des Matching-Moduls entgegenzunehmen. Dazu sendet das Matching-Modul dem Auftraggeber die Nachricht *Berechne* und übergibt diesem, die Matching-Anfrage. Des Weiteren ist der Auftraggeber für die Dekomposition der Matching-Anfrage verantwortlich, damit diese Anfrage parallel von den Arbeitern bearbeitet werden kann. Was sich genau hinter der Dekomposition der Matching-Anfrage verbirgt, wird im *Kapitel 4.3.4* beschrieben. Nach der Dekomposition legt der Auftraggeber die zerlegten Matching-Aufgaben in Form von Tupeln zur Weiterverarbeitung in den Datenspeicher, indem er die Nachricht *AufgabeHinzufügen* an den Tupelraum sendet. Danach wechselt der Auftraggeber in den Zustand *WarteAufErgebnisse*, um die Zwischenergebnisse zu empfangen. Das folgende Zustandsdiagramm zeigt das Verhalten des Auftraggebers. Dabei stellen die Zustandsübergänge die Nachrichten dar, die er empfangen kann. Da das Aktor-Modell die Modellierung eines Aktors als Zustandsautomat zulässt, wird dieses Entwurfsmuster bei der Realisierung des Auftraggebers berücksichtigt.

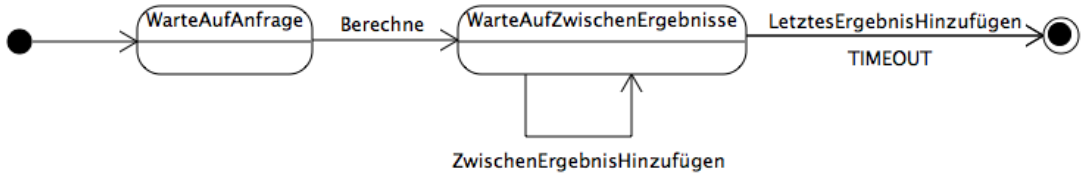

Abbildung 16: Zustandsdiagramm des Auftraggebers

Sobald die Arbeiter vom Matching-Modul gestartet worden sind, fordern sie vom Tupelraum in einem bestimmten Zeitintervall Aufgaben an. Falls der Tupelraum zu diesem Zeitpunkt Aufgaben in seinem Speicher hält, so wird dem jeweiligen Arbeiter die Aufgabe zur Bearbeitung übergeben. Falls nicht, wartet dieser eine bestimmte

Zeit bevor er die Aufgabenanforderung wiederholt. Sobald ein Arbeiter das Ergebnis ermittelt hat, legt er es in den Datenspeicher, indem er die Nachricht *ErgebnisHinzufügen* dem Tupelraum sendet. Nachdem ein Zwischenergebnis dem Tupelraum hinzugefügt worden ist, übergibt der Tupelraum dem Auftraggeber anhand der Nachricht *ZwischenergebnisHinzufügen* das Ergebnis-Tupel, was zu keinem Zustandswechsel des Auftraggebers führt. Nach dem Erhalt eines Zwischenergebnisses, ist der Arbeitgeber für die Komposition dieser Ergebnisse verantwortlich. Sobald die letzte Aufgabe bearbeitet worden ist, benachrichtigt der Tupelraum den Arbeitgeber anhand der gesonderten Nachricht *LetztesZwischenergebnisHinzufügen*. Dies führt dazu, dass der Auftraggeber in seinen Endzustand übergeht und das Endergebnis dem Matching-Modul zurückgibt. Falls nicht alle Ergebnisse eintreffen, wirft der Auftraggeber nach einer bestimmten Zeit ein Timeout und gibt alle bis dahin empfangenen Ergebnisse zurück.

4.3.2 Interne Repräsentation der Matching-Daten

Die interne Datenstruktur der Matching-Daten lehnt sich an die Struktur des internen Metamodells an. Dabei wird dieses Modell, welches in einer EBNF im *Kapitel 4.3* beschrieben worden ist mit der Programmiersprache Scala in Form von abstrakten Datentypen umgesetzt. Wie diese Abbildungen realisiert wurden, wird im *Kapitel 5.1* erläutert.

4.3.3 Mathematisches Modell

Die Klasse mathematisches-Modell aus *Abbildung 15* implementiert ein an *Kapitel 3.1.4* angelehntes Modell und kapselt das mathematische Know-how in Form von Bewertungsfunktionen und Ähnlichkeitsrelationen, die verwendet werden, um den Matching-Grad eines Angebot-Anfrage-Paares zu berechnen. Dieses Modell wird als Grundlage für die Implementierung des generischen Matching-Moduls verwendet. Dabei ist darauf hinzuweisen, dass das Modell nach (Vries, 2009) nicht komplett übernommen wird. Stattdessen wurde es an die spezifischen Anforderungen angepasst. Wie schon in den Matching-Grundlagen erwähnt, ist die Wahl der Bewertungsfunktion abhängig vom Attributtyp. Daher werden nun die jeweiligen Bewertungsfunktionen vorgestellt, die entweder aus dem Modell von (Vries, 2009) entnommen wurden oder selbst definiert worden sind.

Numerische Werte und Intervalle

Für die Abwicklung dieser Attributtypen wird die Bewertungsfunktion aus *Abbildung 3* verwendet, die im *Kapitel 2.3.1.1* definiert worden ist.

Selektionen

Hierbei wird die Funktion für Selektionen aus dem mathematischen Modell nach (Vries, 2009) hinzugezogen.

Aufzählungen

Um bei diesem Attributtypen einen Matching-Grad zu berechnen, kann folgender Ansatz verwendet werden:

$$\text{Matching} - \text{Grad} = \frac{\text{Treffer des Angebots}}{|\text{Aufzählung}|}$$

Abbildung 17: Bewertungsfunktion für Aufzählungen

Hierbei wird die Anzahl der übereinstimmenden Angebotswerte durch die Größe der Aufzählung dividiert.

Zeichenketten

Hier erfolgt eine recht einfache Bewertungsfunktion, die nur einen Matching-Grad von 1.0 zurückgibt, falls der Attributwert der Nachfrage identisch (==) mit dem Wert des Angebotsattribut ist.

Wahrheitswerte

Für diesen Attributtypen wird ebenfalls die Gleichheitsrelation (==) verwendet, um zwischen zwei booleschen Ausdrücken die Übereinstimmung zu ermitteln.

Totale Berwertungsfunktion

Hierzu wird die Funktion aus *Abbildung 7* verwendet, welche die Gewichtungen der Attribute berücksichtigt.

4.3.4 Matching-Algorithmus

Dieser Algorithmus wird im Matching-Modul durch die Aktoren *Auftraggeber*, *Arbeiter* und *Tupelraum* implementiert. Jede Ausführungseinheit realisiert für sich seinen Teilalgorithmus, der zur Gesamtfunktionalität des Matchings beiträgt.

Eingabedaten: Liste aus Matching-Kriterien

Ausgabedaten: Liste von Matching-Ergebnissen

Der Algorithmus ist in drei Schritten organisiert: Dekomposition – Matching – Merging, wobei die Dekomposition und das Merging vom Auftraggeber übernommen werden und das Matching von den Arbeitern. Die Kommunikation zwischen ihnen geschieht wie im *Kapitel 4.3.1* beschrieben allein über den Tupelraum. Die folgende *Abbildung 18* skizziert diesen Algorithmus grafisch.

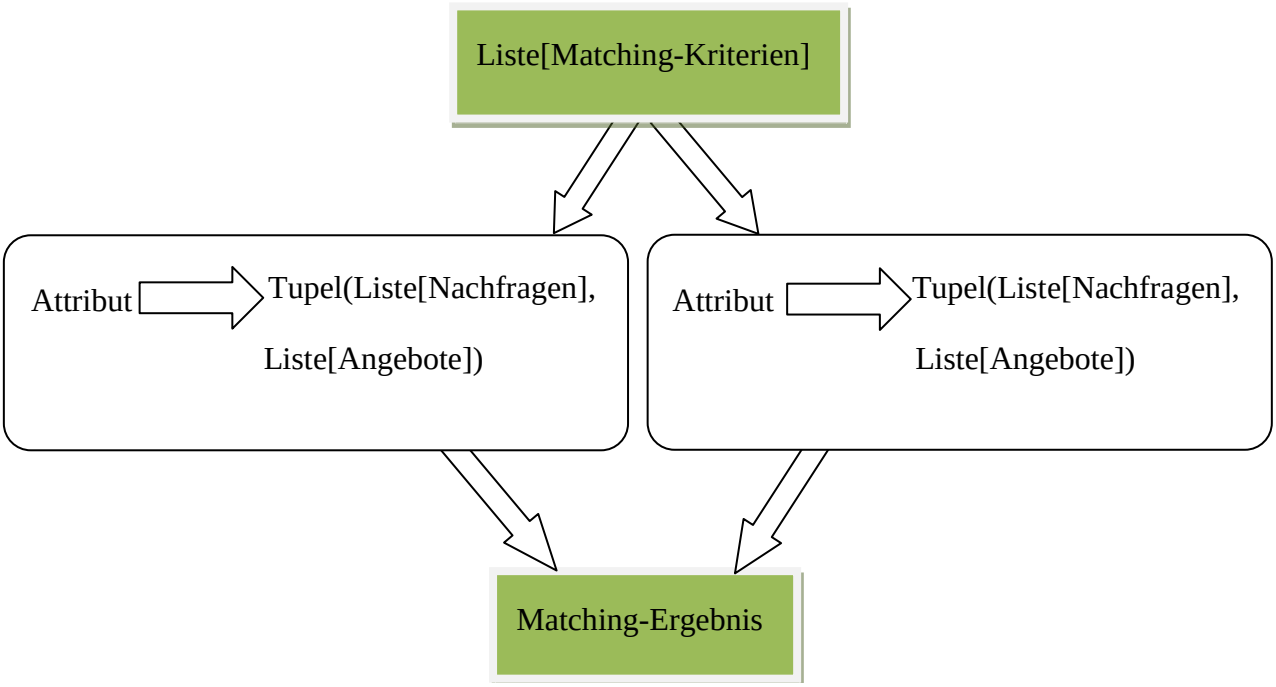

Abbildung 18: Matching-Algorithmus

Dekomposition

Bei diesem Schritt zerlegt der Auftraggeber wie in *Abbildung 18* ersichtlich die eingehenden Daten nach gleichartigen Attributen. Dies bedeutet, dass nach der Dekomposition die Attribute auf die entsprechenden Nachfragen und Angebote mit gleicher Attributbezeichnung abgebildet werden. Damit erreicht man einen höheren Parallelisierungsgrad. Des Weiteren wird die eingehende Liste von Matching-Kriterien in Nachfragen und Angebote partitioniert, um im Matching-Schritt des Arbeiters eine einfachere Matching-Struktur zu gewährleisten.

Durch die Dekomposition ist es möglich, dass die Arbeiter Teilaufgaben unabhängig voneinander und parallel bearbeiten können. Nach der Zerlegung der Matching-Anfrage in kleinere Aufgaben, werden diese dem Tupelraum wie im *Kapitel 4.3.1* beschrieben, hinzugefügt. Im Folgenden wird der Pseudocode mit dazugehöriger Notationsübersicht zur Umsetzung der Dekomposition vorgestellt.

Variable	Datentyp
`anfrage`	Liste: Matching-Kriterium
`gruppierte-Anfrage`	Abbildung: String → Liste: Matching-Kriterium
`partitionierte-Anfrage`	Abbildung: String → Tupel(Liste: Nachfrage, Liste: Angebot)

Tabelle 11: Notationsübersicht der Abbildung 19

Der Auftraggeber zerlegt die Matching-Anfrage in Teilaufgaben. Die Anzahl der Teilaufgaben ist abhängig von der Anzahl der verschiedenen Attribute. Je mehr verschiedene Attribute es gibt, desto höher ist der Parallelisierungsgrad des Algorithmus. Durch die Dekomposition wird zusätzlich der Matching-Raum verringert, da die Angebots-Nachfrageattribute unterschiedlicher Bezeichnung nicht miteinander verglichen werden.

```
1. dekomposition(anfrage) = {
2.     gruppierteAnfrage = guppiereNachGleichenAttributen(anfrage)
3.     partitionierteAnfrage =
       partitioniereNachfragenUndAngebote(gruppierteAnfrage)
4.     return partitionierteAnfrage

   }
```

Abbildung 19: Pseudocode - Dekomposition

Matching

Das Matching übernehmen die Arbeiter, die unabhängig voneinander die Teilaufgaben aus dem Tupelraum zur Bearbeitung herausnehmen. Für die Berechnung des Matching-Grades rufen die Arbeiter in *Zeile 5* der *Abbildung 21* die entsprechende Bewertungsfunktion des mathematischen Modells aus *Kapitel 4.3.3* auf und multiplizieren ihn mit der Gewichtung des Nachfrageattributs, um eine Priorisierung zu berücksichtigen. In den nächsten Abbildungen wird der Pseudocode mit dazugehöriger Notationsübersicht vorgestellt, der die Matching-Tätigkeit des Arbeiters spezifiziert.

Variable	Datentyp
aufgabe	Tupel(Liste: Nachfrage, Liste: Angebote)
n	Nachfrage
a	Angebot
mg	Double
p1	Profil
p1	Profil
E	Liste: Matching-Ergebnis

Tabelle 12: Notationsübersicht der Abbildungen 20 und 21

Im Matching-Algorithmus muss berücksichtig werden, ob nur Profile gleichen Matching-Typs (*siehe Anforderung 9*) miteinander verglichen werden sollen, oder Profile unterschiedlichen Typs (*siehe Anforderung 10).* Im Algorithmus selber unterscheiden sich diese zwei Anforderungen in einer Zeile, was folgende Abbildung verdeutlicht.

```
1. If(n.id != a.id) → Berechne(n,a)
2. If(n.matchingTyp != a.matchingTyp) → Berechne (n,a)
```

Abbildung 20: Pseudocode - Matching 1

Die Anweisung in *Zeile 1* beschreibt den Fall, bei dem Profile gleichen Matching-Typs miteinander verglichen werden, wie z.B. (Firmen : Firmen). Durch die Anweisung wird verhindert, dass der Matching-Grad zwischen Angeboten und Nachfragen des gleichen Profils berechnet wird. *Zeile 2* dagegen berücksichtigt, gemäß der *Anforderung 10*, dass nur Profile unterschiedlicheren Typs miteinander verglichen werden. Je nachdem, welche Matching-Art berücksichtigt werden soll, wird die entsprechende Bedingung aus *Zeile 4* der *Abbildung 21* dies gewährleisten.

```
1. matching(aufgabe) {
2.     ∀ n ∈ aufgabe do {
3.       ∀ a ∈ aufgabe do {
4.         If(n.id!=a.id) oder If(n.matchingTyp != a.matchingTyp){
5.           mg = r.gewichtung * mathematischesModell.berechne(n,a)
6.           p1 = erzeugeProfile(n.name,n.id)
7.           p1 = erzeugeProfile(a.name,a.id)
8.           E ← erzeugeMatchingErgebnis(p1,p2,mg)
9.          }}}
10.  return E
     }
```

Abbildung 21: Pseudocode - Matching 2

Merging

Sobald der Auftraggeber Zwischenergebnisse vom Tupelraum erhält, vereinigt er diese zu einem Gesamtresultat. Dabei bildet das Matching-Ergebnis zwei Profile auf den dazugehörigen Matching-Grad ab. Wenn ein Zwischenergebnis vom Tupelraum eintrifft, wird zuerst geprüft, ob es im Gesamtresultat bereits einen Eintrag dieser zwei Profile gibt. Ist dies der Fall, wird der neu übergebene Matching-Grad in das bereits vorhandene Ergebnis eingerechnet. Falls nicht, wird ein neuer Eintrag im Gesamtresultat angelegt. Welche Anweisungsschritte das Merging beinhaltet, wird im Folgenden anhand des Pseudocodes mit dazugehöriger Notationsübersicht dargestellt.

Variable	Datentyp
ergebnisse	Liste: Matching-Resultat
e	Matching-Resultat
p1	Profil
p2	Profil
gesamtErgebnis	Liste: Matching-Resultat

Tabelle 13: Notationsübersicht der Abbildung 22

```
1. merging(ergebnisse) = {
2.     ∀ e ∋ ergebnisse do {
3.         p1 = e.p1
4.         p2 = e.p2
5.         If((p1,p2)∈ gesamtErgebnis)
6.             gesamtErgebnis.update(e)
7.         Else
8.             gesamtErgebnis ← e

           }
       }
```

Abbildung 22: Pseudocode - Merging

Für den Matching-Fall, dass die Ähnlichkeit zwischen zwei verschiedene Profiltypen *A* und *B* berechnet werde soll, kann die Anzahl der zu vergleichenden Profile über das Kreuzprodukt zweier Mengen ermittelt werden. Das Kreuzprodukt stellt wie in *Abbildung 1* beschrieben jedes Element aus der einen Menge mit jedem Element aus der zweiten Menge in Beziehung:

$$A \times B = \{(p1 , p2) \mid p1 \in A \wedge p2 \in B\}$$

Abbildung 23: Matching-Raum der Profile

Da die Intention der Matching-Art aus Anforderung *10* es ist, alle Profile des einen Typs *A* mit allen Profilen des Typs *B* miteinander zu vergleichen, ergibt sich die Anzahl der zu vergleichenden Profile aus der Vereinigung der Kreuzprodukte:

$|A \times B \cup B \times A|$.

Beispiel

Das folgende Beispiel demonstriert die Vorgehensweise des Matchings, indem die Matching-Daten aus *Tabelle 14* dem Algorithmus übergeben werden und die Schritte *Dekomposition – Matching – Merging* durchgeführt werden. Dabei handelt es sich um ein Matching bei dem nur Profile unterschiedlichen Typs miteinander verglichen werden. Das Beispiel beschreibt den Anwendungsfall, dass Firmen mit Bewerber verglichen werden und deren Ähnlichkeit ermittelt wird. Aus Gründen der Übersichtlichkeit umfassen die zwei Firmenprofile: *F* = {*AKRA*, *Dynport*} drei Nachfragen und zwei Angebote und die Bewerber: *B* = {*Gates*, *Kay*} jeweils zwei Nachfragen und drei Angebote. Um die Matching-Daten noch einmal vor Augen zu führen, werden diese mit entsprechender Variablenbenennung im Folgenden aufgelistet:

Nachfrage		**Angebot**	
Variable	**Nachfrage-Tupel**	**Variable**	**Angebots-Tupel**
n1	Gates,3,Beruf,0.7,Programmierer	**a1**	AKRA,1,Beruf,Programmierer
n2	Gates,3,Gehalt,0.3,3500,>=	**a2**	AKRA,1,Gehahlt,3000
n3	Kay,4,Beruf,0.7,Designer	**a3**	Dynport,2,Beruf,Designer
n4	Kay,4,Gehalt,0.3,2500,>=	**a4**	Dynport,2,Gehalt,2800
n5	AKRA,1,Alter,0.1,[18,35],offenes-Intervall	**a5**	Gates,3,Alter,56
n6	AKRA,1,Beruf,0.5,Programmierer	**a6**	Gates,3,Beruf,Programmierer

n7	AKRA,1,Sprachen,0.4,{C#,Scala}	**a7**	Gates,3,Sprachen,{C#,F#}
n8	Dynport,2,Alter,0.1,30,>=	**a8**	Kay,4,Alter,36
n9	Dynport,2,Beruf,0.5,Designer	**a9**	Kay,4,Beruf,Designer
n10	Dynport,2,Sprachen,0.4,{Java,Scala,C}	**a10**	Kay,4,Sprachen,{Java,Scala}

Tabelle 14: Matching-Daten – Beispiel

Wie in *Tabelle 14* ersichtlich, müssen nur Attribute mit Ähnlichkeitsrelationen versorgt werden, die entweder vom numerischen Typ sind oder ein Intervall repräsentieren. Die folgende Tabelle zeigt die Ergebnisse der Dekomposition. Die *Tabelle 15* demonstriert, wie die Matching-Daten aus *Tabelle 14* in vier unabhängige Teilaufgaben zerlegt wurden.

Dekomposition der Matching-Daten

Attribut	Tupel(Liste: Nachfrage, Liste: Angebot)
Beruf	([n1,n3,n6,n9] , [a1,a3,a6,a9])
Gehalt	([n2,n4], [a2,a4])
Alter	([n5,n8], [a5,a8])
Sprachen	([n7,n10], [n7,n10])

Tabelle 15: Dekomposition – Beispiel

In *Tabelle 16* werden die Matching-Grade der einzelnen Angebot-Nachfrage Paare aufgelistet. Dazu wurden die entsprechenden Bewertungsfunktionen des mathematischen Modells aus *Kapitel 4.3.3* aufgerufen.

Matching

(Profil,Profil)	Gesamt-Matching-Grad
(AKRA,Gates)	0.75
(Gates,AKRA)	0.78
(Dynport,Gates)	0.1
(Gates,Dynport)	0.0
(AKRA,Kay)	0.29

(Kay,AKRA)	0.3
(Dynport,Kay)	0.86
(Kay,Dynport)	1.0

Tabelle 16: Matching - Beispiel

Die letztendlichen Matching-Ergebnisse aus *Tabelle 17* werden durch den Auftraggeber ermittelt, indem dieser alle Zwischenergebnisse eines Profilpaares zu einem Gesamtresultat vereint. Die Anzahl der Profilpaare kann wie bereits erwähnt folgendermaßen ermittelt werden: $|F \times B \cup B \times F|$ → 8 Profilpaare

Merging

(Profil,Profil)	Gesamt-Matching-Grad
(AKRA,Gates)	0.75
(Gates,AKRA)	0.78
(Dynport,Gates)	0.1
(Gates,Dynport)	0.0
(AKRA,Kay)	0.29
(Kay,AKRA)	0.3
(Dynport,Kay)	0.86
(Kay,Dynport)	1.0

Tabelle 17: Merging - Beispiel

5 Realisierung

Im Folgenden wird beschrieben, wie das Matching-Modul im Einzelnen in der Programmiersprache Scala implementiert wurde. Des Weiteren wird erläutert, wie die aus dem Entwurf erwähnten Architekturstile angewendet wurden, um das Matching-Modul zu realisieren. Anschließend wird sowohl auf die Umsetzung des mathematischen Modells, als auch auf die Realisierung des im *Kapitel 4.3.4* spezifizierten Matching-Algorithmus eingegangen. Aus Gründen der Übersichtlichkeit, werden nur die wichtigsten Code-Ausschnitte dargestellt und erläutert. Um Scala im Detail kennen zu lernen, wird auf (Martin Odersky, 2010) verwiesen.

Wie aus dem Entwurf ersichtlich, besteht das Modul hauptsächlich aus 4 Klassen, die verwendet wurden, um die im Entwurf erwähnten Architekturstile zu realisieren und die Funktionalität des Matchings zu implementieren. Um diese Klassen als eigenständige Ausführungseinheiten implementieren zu können, wurden wie bereits im *Kapitel 4.3.1* beschrieben, diese Objekte als Aktoren, gemäß dem Aktor-Modell aus *Kapitel 2.2.2*, realisiert.

5.1 Schnittstelle

Die EBNF des internen Metamodells aus *Kapitel 4.2* wurde in Scala umgesetzt, indem die jeweils linksstehenden Elemente der EBNF als abstrakte Datentypen realisiert werden und aus den rechtsstehenden Elementen bestehen. Dem Matching-Modul wird über seine öffentliche Schnittstelle aus *Abbildung 24* die Matching-Anfrage übergeben.

Dazu implementiert das Matching-Modul das Trait[32] *Matching*, welche die abstrakte Methode `matching`, gemäß der im Entwurf festgelegten Methodensignatur, spezifiziert. Nachdem die beteiligten Aktoren Auftraggeber, Arbeiter und Tupelraum vom Matching-Modul erzeugt und gestartet worden sind, wird dem Auftraggeber die Matching-Anfrage, über die synchrone Methode *`!?`* aus dem Scala-Actor-Framework, übergeben. Dabei repräsentiert `!?` nach (Esser, 2010) eine blockierende Methode und kehrt erst nach einem `reply()`–Aufruf des Auftraggebers mit dem Matching-Ergebnis zurück.

[32] Abstrakte Klasse, in der Methoden spezifiziert und definiert werden können

```
1. trait Matching {
2.  def matching(anfrage: List[MatchingKriterium])
                              : List[MatchingErgebnis]
   }
3. object MatchingModul extends Matching {

4.   def matching(aufgaben: List[MatchingKriterium]) = {
      // Erzeugen und Starten der beteiligten Aktoren
5.    val matchingErgebnis = auftraggeber !? berechne(aufgaben)
     }
   }
```

Abbildung 24: Implementierung der Matching-Schnittstelle

5.2 Auftraggeber

Die nächsten Abbildungen zeigen die Realisierung des Auftraggebers, in dem insbesondere auf seine Zustände und Zustandsübergänge, welche im *Kapitel 4.3.1* in Form eines Zustandsdiagramms spezifiziert worden sind, eingegangen wird.

Der Auftraggeber speichert Informationen über die eingehenden Matching-Ergebnisse in der Instanzvariable *gesamtErgebnis*. Die `act`-Methode des Auftraggebers repräsentiert in *Abbildung 25* den Startzustand des Auftraggebers und definiert den definiert den Lebenszyklus eines Aktors. Wenn der Auftraggeber gestartet wird, wird er fließend in den `warteAufAnfrage`-Zustand überführt.

```
1. class Auftraggeber(tupelRaum: TupelRaum) extends Actor {
2.    var gesamtErgebnis = List[MatchingResultat]()

3.    def act = {
4.        warteAufAnfrage()        // Zustandsübergang
       }

   }
```

Abbildung 25: Auftraggeber - Startzustand

Die folgende Abbildung zeigt denjenigen Zustand, in dem der Auftraggeber, auf eine Matching-Anfrage wartet. Sobald das Matching-Modul dem Auftraggeber die Nachricht *Berechne* schickt, führt der Auftraggeber die Dekomposition der Anfrage durch, fügt die Teilaufgaben gemäß dem Linda-Architekturstil dem Tupelraum hinzu und geht in den Zustand *WarteAufErgebnisse* über.

```
1.  def warteAufAnfrage = {
2.      link(speicher)
3.      loop{
4.      react{
5.      case Berechne(anfrage) => {
6.         val ausgaben = dekomposition(anfrage)
7.         aufgaben.foreach(elem => {
8.            val aufgabe = Tuple2(elem._2._1,elem._2._2)
9.            speicher ! aufgabeHinzufügen(aufgabe)
           })
10.        WarteAufErgebnisse( )      // Zustandsübergang
11.        reply(gesamtErgebnis)
12.        exit("Arbeit beenden!")
        }

      }}}
```

Abbildung 26: Auftraggeber - WarteAufAnfrage-Zustand

In diesem Zustand führt der Auftraggeber in *Zeile 6* die Zerlegung der Matching-Anfrage gemäß dem Algorithmus aus *Kapitel 4.3.4* in kleinere Teilaufgaben durch und fügt sie in *Zeile 9* dem Tupelraum mithilfe der asynchronen Nachricht *aufgabeHinzufügen* hinzu. Dabei werden Aktor-Nachrichten in Scala, siehe *Abbildung 27*, als Case-Objekte realisiert. Case-Klassen finden insbesondere ihre Verwendung im Bereich des Pattern-Matchings, bei dem nicht nur wie in Java Ganzzahlen im Sinne vom Switch-Case Konstrukt verarbeitet werden, sondern auch komplexe Objekte. Damit ist es möglich, eine empfangene Nachricht als Case-Objekt auf das Nachrichtenprotokoll des Aktors mittels Pattern-Matching abzubilden.

```
1. case class Berechne(anfrage: List[MatchingKriterium])
2. case class
3.    ZwischenergebnisHinzufuegen(ergebnis: List[MatchingErgebnis]
4. case class
5.    LetztesErgebnisHinzufuegen(ergebnis: List[MatchingErgebnis])
```

Abbildung 27: Nachrichtenprotokoll des Auftraggebers

Die Methode `dekompostion()` wurde bereits im *Kapitel 4.3.4* erläutert und wird daher nicht weiter betrachtet. Nachdem der Auftraggeber alle Teilaufgaben in den Datenspeicher gelegt hat, wird dieser in den Zustand *warteAufErgebnisse* überführt, welcher in *Abbildung 28* dargestellt ist. Jedes Mal, wenn der Tupelraum dem Auftraggeber ein Zwischenresultat schickt, nimmt dieser das Zwischenergebnis und vereinigt es mit den bisher empfangenen Ergebnissen.

```
1. def warteAufErgebnisse = {
2.     var flag = true
3.     while(flag){
4.       receiveWithin(aufgaben.size * 60){
5.         case EmpfangeZwischenergebnis(zwischenergebnis) =>
6.                     merging(zwischenergebnis)
7.         case EmpfangeLetztesErgebnis(letztesErgebnis) =>  {
8.               merging(letztesErgebnis)
9.               flag = false
           }
10.        case TIMEOUT => flag = false
   }}}
```

Abbildung 28: Auftraggeber - WarteAufErgebnisse-Zustand

Dabei ist zu unterscheiden, ob es sich bei dem Ergebnis um das letzte Zwischenergebnis handelt. Falls ja, wird der Zustand nach dem *Merging* verlassen und in den vorherigen Zustand gewechselt. Falls nicht, vereinigt der Auftraggeber das Zwischenergebnis mit den bereits empfangenen Ergebnissen und verharrt im Zustand *warteAufErgebnisse*.

Dabei ist darauf hinzuweisen, dass der Sender der Matching-Anfrage, nämlich das Matching-Modul, nur im Zustand *warteAufAnfrage* bekannt ist. Um nach dem Erhalt des letzten Ergebnisses das Gesamtresultat dem Sender mithilfe der Methode `reply()` zurückzugeben, ist der Rücksprung in diesen Zustand notwendig. Um dies zu realisieren, wird hierfür ein *Flag*[33] verwendet. Sobald diese boolesche Variable auf **`false`** gesetzt wird, wird der Auftraggeber in den vorherigen Zustand überführt und übermittelt dem Matching-Modul das Gesamtergebnis.

Durch die `link()` – Methode aus *Abbildung 26*, verbindet sich der Auftraggeber mit dem Tupelraum. Dieses Feature hat die Eigenschaft, dass sobald sich der Auftraggeber mittels der `exit()` – Methode beendet, sich automatisch auch alle verbundenen Aktoren beenden. Damit wird bewerkstelligt, dass sobald der Auftraggeber dem Matching-Modul das Ergebnis zurückgibt, sich alle anderen

[33] Repräsentiert eine boolesche Variable, die die Werte wahr oder falsch annehmen können

beteiligen Aktoren ebenfalls beenden. Wie in den *Grundlagen 2.3.2* erwähnt, bietet die blockierende Methode `receiveWithin()` die Möglichkeit, nur eine bestimmte Anzahl von Millisekunden auf eine Nachricht zu warten. Wenn zu diesem Zeitpunkt noch keine Nachricht eingegangen ist, wird dem Aktor die Nachricht TIMEOUT gesendet. Dadurch kann der Auftraggeber entscheiden, was bei einem TIMEOUT geschehen soll. Wie in der *Abbildung 28* ersichtlich, wird bei dem Fall eines TIMEOUT´s ebenfalls in den vorherigen Zustand zurückgesprungen, um alle bis dahin eingegangenen Ergebnisse dem Matching-Modul zurückzugeben. Für die Wahl der Länge, bis ein TIMEOUT ausgelöst wird, wird hierbei die Aufgabenmenge mit 60 Millisekunden multipliziert. Damit wird verhindert, dass beim Ausfall eines Arbeiter-Aktors, der Auftraggeber ewig auf den Empfang der Ergebnisse wartet.

5.3 Arbeiter

Wie beim Auftraggeber, wird dem Aufragnehmer ebenfalls der Tupelraum bei der Instanziierung übergeben. Um benachrichtigt zu werden, wenn sich die Aktoren Auftraggeber und Tupelraum beenden, verbinden sich die Arbeiter über die bereits

```
1. case class KeineAufgabeVerfügbar
2. case class AufgabeVerfügbar(n:List[Nachfrage],a:List[Angebot])
```

Abbildung 29: Nachrichtenprotokoll des Arbeiters

Sobald ein Arbeiter gestartet worden ist, fordert dieser in einem Intervall von 50 Millisekunden eine Aufgabe vom Tupelraum an. Falls der Tupelraum eine Aufgabe für den Arbeiter bereithält, übermittelter diese mittels der Nachricht *Aufgabe-Verfügbar*. Falls nicht, sendet der Tupelraum die Nachricht *KeineAufgabeVerfügbar* an den Arbeiter, der die Aufgabenanforderung nach 50 Millisekunden wiederholt.

```
1. def act = {
2.   link(speicher)
3.    loop{
4.     speicher ! AufgabeAnfordern
5.      react{
6.       case AufgabeVerfügbar(nachfragen,angebote) => {
7.         nachfragen.foreach(r => {
8.         angebote.foreach(o => {
9.         val matchingGrad = r.gewichtung * Evaluation.eval(r,o)
10.         if(matchingGrad!=0.0) {
11.          ergebnis.+:(MatchingErgebnis(p1,p2,matchingDegree))
          }})})
12.      sender ! ErgebnisHinzufuegen(ergebnis)
        }
13.      case KeineAufgabeVerfügbar => {Thread.sleep(50)}

    }}}
```

Abbildung 30: Arbeiter - Scala Code

Wenn dem Arbeiter eine Aufgabe übertragen worden ist, berechnet dieser in der *Zeile 9* aus *Abbildung 30* den Matching-Grad für jedes Nachfrage-Angebotspaar, indem er die `eval()`- Methode des mathematischen Modells aufruft und mit der Attributgewichtung multipliziert. Nachdem der Matching-Grad ermittelt worden ist, wird ein Matching-Ergebnis gemäß der Struktur des internen Metamodells erzeugt. Abschließend fügt der Arbeiter dem Tupelraum seine berechneten Ergebnisse hinzu. Durch das Aktor-Element *sender* ist es möglich nach (Esser, 2010) auf eine Nachricht asynchron zu antworten. Dabei repräsentiert dieses Element den Sender der empfangenen Nachricht.

5.4 Tupelraum

Wie im Entwurf aus *Kapitel 4* beschrieben, übernimmt der Tupelraum die Verwaltung der Aufgaben und Zwischenergebnisse. Die folgende Abbildung stellt das Nachrichtenprotokoll des Tupelraumes dar.

```
1. case class AufgabeHinzufuegen
   (aufgabe: Tuple2[List[Nachfrage],List[Angebot]])
2. case class ErgebnisHinzufuegen(e: List[MatchingErgebnis])
3. case class AufgabeAnfordern
```

Abbildung 31: Nachrichtenprotokoll des Tupelraumes

Dem Tupelraum können mittels der Nachrichten *AufgabeHinzufuegen* und *ErgebnisHinzufuegen* Tupel hinzugefügt werden. Des Weiteren meldet sich der Auftraggeber am Tupelraum an, um von ihm benachrichtigt zu werden, sobald Zwischenergebnisse von den Arbeitern dem Speicher hinzugefügt worden sind. Wenn ein Arbeiter ein Matching-Ergebnis in den Tupelraum gelegt hat, prüft der Speicher in den *Zeilen 11-15* aus *Abbildung 32*, ob alle Aufgaben bearbeitet worden sind und benachrichtigt dementsprechend den Auftraggeber. Um zu ermitteln, ob alle Ergebnisse eingetroffen sind, wird ein Zähler verwendet, der die eingehenden Matching-Ergebnisse zählt. Sobald dieser Zähler den Wert der Aufgabenanzahl erreicht hat, benachrichtigt dieser den Auftraggeber anhand der gesonderten Nachricht *LetztesErgebnisHinzufügen*.

Die nächste Abbildung zeigt, wie der Tupelraum als Aktor implementiert wurde. Durch die **case**-Anweisungen werden die Nachrichten deklariert, die der Aktor empfangen kann und auf Code-Blöcke abgebildet, die ausgeführt werden, sobald ihm eine bekannte Nachricht gesendet wurde.

```
1. class TupelRaum extends Actor {
2.  def act = {
3.   loop{
4.    react{
5.     case AufgabeHinzufügen(aufgabe) => {aufgabenPool+:(aufgabe)}
6.     case AufgabeAnfordern => {
7.      if(!aufgabenPool.isEmpty){
8.       val aufgabe = aufgabenPool.head
9.       sender ! AufgabeVerfügbar(aufgabe.nachfrage,aufgabe.angebot)
        }
10.     else {sender ! KeineAufgabeVerfügbar}
       }
11.    case ErgebnisHinzufuegen(ergebnis) => {
12.     empfangeneErgebnisse += 1
13.     if(empfangeneErgebnisse==anzahlAufgaben){
14.      auftraggeber ! LetztesErgebnisHinzufügen(ergebnis)
        }
15.     else{auftraggeber ! ZwischenergebnisHinzufügen(ergebnis)}
       }

}}}}
```

Abbildung 32: Tupelraum - Scala Code

5.5 Mathematisches Modell

Das mathematische Modell beinhaltet zum einen eine Menge von Berwertungsfunktionen und zum anderen mehrere Ähnlichkeitsrelationen, die von den Berwertungsfunktionen zur Berechnung des Matching-Grades verwendet werden. Wie bereits mehrmals erwähnt, ist die Wahl der Bewertungsfunktion abhängig vom Datentyp des Nachfrageattributs. Die folgende `eval()`- Methode bildet den Typ des Nachfrageattributs auf die entsprechende Bewertungsfunktion ab. In diesem Abschnitt wird der Kürze halber nur auf die Bewertungsfunktionen eingegangen, die für numerische Attribute und Intervalle den Matching-Grad berechnen.

```
1. def eval(n: Nachfrage,a: Angebot): Double = {
2.  val inhalt = n.attributWert
3.  inhalt match {
4.  case inhalt if(inhalt.isInstanceOf[Intervall]) =>

5.    {…siehe Abbildung 46…}

6.  case inhalt if(inhalt.isInstanceOf[Numerisch]) =>

7.    {…siehe Abbildung 47…}
    }
      … }
```

Abbildung 33: Methode eval( ) des mathematischen Modells

Je nachdem, von welchem Typ das Nachfrageattribut ist, wird das Angebot-Nachfrage-Paar auf die entsprechende Bewertungsfunktion abgebildet.
Für die weitere Beschreibung des mathematischen Modells, werden in den nächsten zwei Abbildungen eine Auswahl von Ähnlichkeitsrelationen für numerische Werte und Intervalle in Scala definiert. Sie werden als Funktionen definiert und der Bewertungsfunktion für Intervalle zusammen mit den Attributwerten des Angebot-Nachfrage Paares als Argumente übergeben.

```
1. val greaterEqual: (Double,Double) => Boolean = (a,b) => a>=b
2. val lessEqual: (Double,Double) => Boolean = (a,b) => a<=b
3. val equal: (Double,Double) => Boolean = (a,b) => a==b
```

Abbildung 34: Ähnlichkeitsrelationen für Numerische Werte in Scala

```
1. val greaterEqualLessEqual: (Double,Double,Double) => Boolean =
2.    (x,a,b) => (x >= a) && (x <= b)
3. val greaterThanLessThan: (Double,Double,Double) => Boolean =
4.    (x,a,b) => (x > a) &&(x < b)
```

Abbildung 35: Ähnlichkeitsrelationen für Intervalle in Scala

Die *Abbildung 36* zeigt, wie die Ähnlichkeitsrelation *relation* eines numerischen Nachfrageattributs mittels des Scala Features Pattern-Matching ermittelt wird. Hierbei werden die numerischen Werte wie im *Kapitel 4.3.3* auf Intervalle abgebildet. Dabei ist die Intervallart abhängig von der Ähnlichkeitsrelation. Je nachdem, welche Ähnlichkeitsrelation es ist, wird die jeweilige Funktion aus *Abbildung 34* als Vergleichsoperation verwendet. In *Zeile 3* aus *Abbildung 36* wird ein Intervall erzeugt, indem der numerische Wert des Nachfrageattributs als linke Intervallgrenze und $+\infty$ als rechte Intervallgrenze verwendet wird und in *Zeile 5* zusammen mit der Vergleichsoperation (>=) der Bewertungsfunktion `evalIntervall()` übergeben wird. Wenn die Ähnlichkeitsrelation der (==)-Funktion entspricht, wird der numerische Nachfragewert sowohl als linke, als auch rechte Intervallgrenze abgebildet und zusammen mit der *equal*-Funktion aus *Abbildung 34* der Berwertungsfunktion übergeben. *SR* stellt hierbei eine *Enumeration*[34] dar, welche die Menge der Ähnlichkeitsrelationen enthält.

```
1. case relation if(relation.equals(SR.greaterEqual)) => {
2.    val linkeGrenze = inhalt
3.    val intervall = new Intervall(linkeGrenze,PositiveInfinity)
4.    n.updateInhalt(intervall)
5.    val matchingGrad = evalIntervall(n,a,greaterEqual)
   }
6. case relation if(relation.equals(SR.equal)) => {
7.    val linkeGrenze = inhalt
8.    val intervall = new Intervall(linkeGrenze,linkeGrenze)
9.    n.updateInhalt(intervall)
10.   val matchingGrad = evalIntervall(n,a,equal)
   }
   …
```

Abbildung 36: Numerische Werte werde auf Intervalle abgebildet

[34] Aufzählung

In der nächsten Abbildung wird die Ähnlichkeitsrelation eines Nachfrageattribut vom Typ Intervall auf eine der Scala-Funktionen für Intervalle aus *Abbildung 35* abgebildet und zusammen mit den Attributwerten des Angebot-Nachfrage Paares der Bewertungsfunktion übergeben.

```
1. n.relation match{
2.   case relation if(relation.equals(SR.open)) =>
3.          evalIntervall(n,a,greaterEqualLessEqual)
4.   case relation if(relation.equals(SR.closed)) =>
5.          evalIntervall(n,a,greaterThanLessThan)
6. }
```

Abbildung 37: Abbildung von Ähnlichkeitsrelationen der Intervalle auf Scala-Funktionen

Die nächste in Scala implementierte Funktion `evalIntervall()` realisiert die Bewertungsfunktion für Intervalle, die wie beschrieben sowohl für numerische Nachfrageattribute, als auch für Nachfrageintervalle verwendet wird, um den Matching-Grad eines Angebot-Nachfrage Paares zu berechnen. Dabei wird hier das Prinzip der High-order-Functions verwendet, indem man einer Funktion eine weitere Funktion als Argument übergibt. Diese Bewertungsfunktion erwartet als Argumente ein Nachfrageattribut, ein Angebotsattribut und eine Ähnlichkeitsrelation, die wie folgt definiert ist. Aus Gründen der Übersichtlichkeit werden für die Variablen *linkeGrenze* und *rechteGrenze* in den Zeilen *6-9* die Abkürzungen *from* und *to* verwendet.

```
1. val evalIntervall: (N,A,(Double,Double,Double)
     => Boolean) => Double = (n,a,f) => {
2.   val linkeGrenze = n.inhalt.linkeGrenze
3.   val rechteGrenze = n.inhalt.rechteGrenze
4.   val x: Double = a.inhalt
5.   val e = 0.2                          // Fuzzy Level
6.   x match {
7.   case x if(x < from && x > (1-e)*from)=>(x/(from*e))-((1-e)/e)
8.   case x if(h(x,from,to)) => 1.0
9.   case x if(x > to && x < (1+e)*to)=>((1+e)/e)-(x/(to*e))
10.  case _ => 0.0

  }}
```

Abbildung 38: Bewertungsfunktion für numerische Werte und Intervalle in Scala

`EvalIntervall()` berechnet den Matching-Grad, indem sie die im *Kapitel 2.1.3.1* formal spezifizierte Bewertungsfunktion aus *Abbildung 3* in Scala realisiert.

6 Matching-System

In diesem Kapitel wird eine Systemarchitektur im Sinne eines Matching-Systems vorgestellt. Sie beschreibt eine Möglichkeit das in der Programmiersprache Scala programmierte Matching-Modul so aufzubereiten, dass die Funktionalität des Matchings von einem beliebigen Anwendungssystem genutzt werden kann. Hierbei ist darauf hinzuweisen, dass nicht alle Komponenten des Matching-System realisiert wurden. Diese Komponenten werden dennoch in den Anforderungen und in der Architektur erwähnt um die Möglichkeiten des Matching-Moduls aufzuzeigen.

6.1 Anforderungen

Die in diesem Kapitel beschriebene Systemarchitektur zeigt die Möglichkeiten auf, die es mit der Verwendung des Matching-Moduls gibt. Um das Matching-Modul von außen verwenden zu können, werden mehrere Hilfsmodule und Hilfstechnologien genutzt. Bei der Auswahl dieser Hilfsmittel wurde darauf geachtet, folgende Anforderungen zu erfüllen:

1. Der Dienst des Matching-Moduls soll jederzeit aufrufbar sein.

   → Hierzu wird der Matching-Dienst als RESTful Webservice implementiert. Damit wird erreicht, dass jeder Zeit der Dienst des Matching-Moduls von außen über die HTTP-basierenden REST-Methoden aus *Kapitel 2.4.1* angesprochen werden kann. Um das REST-Prinzip verwenden zu können wird das Webframework Lift aus *Kapitel 2.4.4* verwendet, welches einen einfachen und leichtgewichtigen REST-Support bietet.

2. Systeme heterogener Technologien sollen in der Lage sein dieses Matching-System ansprechen zu können.

   → Um dies zu gewährleisten wird das standardisierte Transportprotokoll HTTP verwendet werden, um einen Datenaustausch zwischen einem Anwendungssystem beliebiger Technologie und dem Matching-System zu ermöglichen. Das Anwendungssystem muss lediglich HTTP unterstützen.

3. Dem Matching-Modul sollen die Matching-Daten in einem XML- und Json Datenformat übergeben werden können.

→ Es werden bestimmte Konvertierungsbibliotheken von Scala verwendet, um ein Parsing der Daten zu gewährleisten.

4. Die Struktur der Ein- und Ausgabe soll spezifiziert sein.

   → Diese Struktur wird durch ein externes Metamodell in Form einer EBNF-Grammatik definiert. Diese Grammatik wird sowohl für XML, als auch für Json mittels der jeweiligen Schemasprache umgesetzt.

5. Es soll einen Validierungsmechanismus geben, der die eingehenden und herausgehenden Daten gegen die jeweilige Spezifikation prüft.

   → Für die Validierung wird auf entsprechende Bibliotheken der Java SE API zugegriffen. Hierbei wird die eingehende XML- oder Json Datei gegen das jeweilige externe Metamodell geprüft.

6. Die Anzahl der Matching-Anfragen müssen skalierbar sein.

   → Dazu wird das Prinzip des parallelen Servers verwendet, welches vom Lift-Framework unterstützt wird. Bei jeder HTTP-Anfrage wird ein Worker-Prozess erzeugt, der für die Bearbeitung der Anfrage zuständig ist.

7. Das Anwendungssystem soll die Möglichkeit bekommen die Matching-Daten im Webservice persistent zu speichern.

   → Zur persistenten Verwaltung der Matching-Daten kann ein relationales Datenbanksystem verwendet werden. Das Datenbankschema ist hierbei durch das interne Metamodell aus *Kapitel 4.2* definiert.

6.2 Architektur

In diesem Abschnitt wird eine Systemarchitektur entworfen, die ein Matching-System, gemäß den spezifizierten Anforderungen aus dem vorherigen Kapitel, modelliert. Die folgende Abbildung zeigt anhand eines Verteilungsdiagramms die Systemarchitektur des Matching-Systems. Dabei ist darauf hinzuweisen, dass nur die farblich markierten Komponenten und Schnittstellen technisch realisiert worden sind, um einmal den vertikalen Durchstich von der Matching-Anfrage bis hin zum Modul zu demonstrieren.

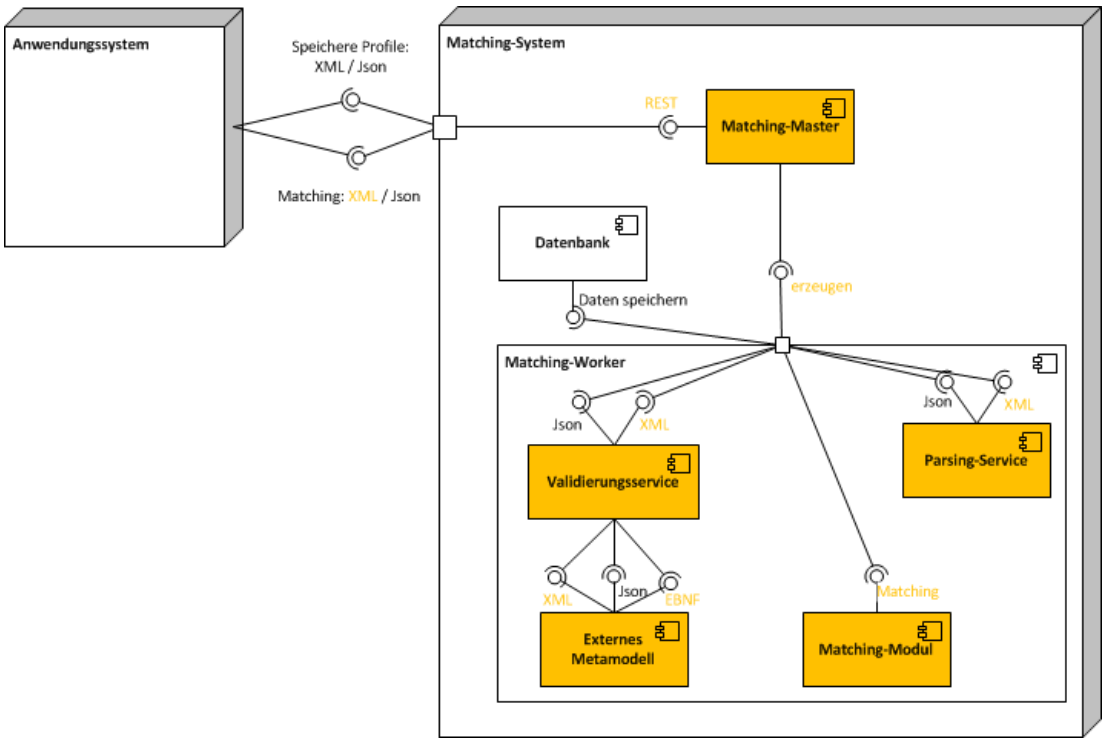

Abbildung 39: Systemarchitektur des Matching-Systems

Das Matching-System wird als RESTful-Webservice realisiert und bietet somit dem Anwendungssystem eine Möglichkeit, über Standardtechnologien wie HTTP, XML und Json, den Matching-Dienst anzufordern. Dadurch, dass das Matching-System als Webservice implementiert ist, wird eine lose Kopplung zwischen Anwendungssystem und Matching-System gewährleistet und ermöglicht die Kommunikation zwischen heterogenen Systemen.

Um das Matching-Modul als RESTful Webservice zu implementieren, wird ein Framework benötigt, welches das REST-Prinzip unterstützt. Das in Scala implementierte Webframework Lift aus *Kapitel 2.4.4* unterstützt diesen Architekturstil für serviceorientierte Anwendungen und bietet einfache und effiziente Möglichkeiten um Matching-Anfragen eines Anwendungssystems über das standardisiertes Protokoll HTTP entgegen zu nehmen.

Dabei kann das Anwendungssystem den Dienst des Matching-Systems über die HTTP-basierenden REST-Methoden GET, POST, PUT und DELETE, wie im *Kapitel 2.4.1* beschrieben, zugreifen. Das Anwendungssystem greift auf das Matching-Modul ausschließlich mit dem POST-Aufruf zu, um die Matching-Daten entweder im XML-Format oder im Json-Format über das HTTP-Protokoll dem Matching-System zu übergeben. Der Webservice nimmt in Form eines Matching-Masters die Anfragen entgegen.

Hierbei ist darauf hinzuweisen, dass das Lift-Framework für jede HTTP-Anfrage einen Worker-Thread erzeugt, der im Folgenden als Matching-Worker bezeichnet wird. Diese Ausführungseinheit ist für die Bearbeitung der jeweiligen Anfrage zuständig. Dieser Prozess beinhaltet die Validierung, Parsing und Matching der Eingabedaten. Der Validation-Service prüft abhängig vom Dateityp (XML oder Json) die eingehenden Daten gegen das jeweilige externe Metamodell, welches die Struktur der Eingabedaten des Matching-Systems spezifiziert. Bei erfolgreicher Validierung werden die Daten zum Parsing-Service weitergeleitet, der die eingehenden Daten in eine interne, Scala-abhängige Datenstruktur konvertiert, die dem internen Metamodell entspricht. Nach der Konvertierung des externen Metamodells in die Struktur des internen Metamodells, werden die Daten dem Matching-Modul übergeben. Die berechneten Matching-Ergebnisse werden dann dem Worker-Prozess zurückgegeben, in das eingehende Datenformat zurück konvertiert und über HTTP an das anfragende System zurückgesendet.

6.2.1 Schnittstelle

Wie schon erwähnt, erreicht man durch einen RESTful Webservice eine einfache HTTP-basierende Datenübertragung zwischen heterogenen Systemen. Die Schnittstelle des Matching-System wird durch einen POST-Aufruf des HTTP-Protokolls realisiert, der die Matching-Daten im XML oder Json Format entgegennimmt. Hierbei ist zu erwähnen, dass die eingehenden Matching-Daten der Struktur des externen Metamodells entsprechen müssen. Um die Struktur der eingehenden und ausgehenden Daten des Matching-Systems zu spezifizieren, werden zwei externe Metamodelle verwendet, die im Folgenden vorgestellt werden. Dabei ist darauf hinzuweisen, dass das Metamodell als EBNF nur zur Veranschaulichung dient und daher nicht bei der praktischen Umsetzung des Matching-Systems verwendet wird.

6.2.1.1. Externes Metamodell als EBNF

Dieses Metamodell erweitert das im *Kapitel 4.2* beschriebene interne Metamodell um benutzerfreundliche Eigenschaften, die dem Anwendungssystem erlauben, hierarchisch verschachtelte Angebote und Nachfragen zu definieren. Aus Effizienzgründen geschieht bei der Konvertierung des externen Metamodells in das interne Metamodell eine Denormalisierung der Matching-Daten, indem die

hierarchisch strukturierten Angebote und Nachfragen in eine flache Tupelstruktur gebracht werden. Diese Struktur der Matching-Daten werden ebenfalls von den beiden weiteren Metamodellen beschrieben. Der Unterschied ist, dass es für die Schemasprachen XSD und Json-Schema Validierungsbibliotheken in mehreren Programmiersprachen gibt, um einkommende XML- oder Json-Dokumente gegen deren Spezifikation zu validieren. Im Folgenden werden die abstrakten Datentypen vorgestellt, die die EBNF der Eingabedaten aus dem *Kapitel 4.2.1* des internen Metamodells erweitern.

Linker Teil der EBNF	Rechter Teil der EBNF
Matching-Typen	Matching-Typ {Matching-Typ}
Matching-Typ	Name {Profil}
Profil	Name ID Matching-Kriterien
Matching-Kriterien	{Angebot}{Nachfrage}
Nachfrage	Attributbezeichnung Gewichtung Nachfrageinhalt [Ähnlichkeitsrelation]
Angebot	Attributbezeichnung Angebotsinhalt

Tabelle 18: Externes Metamodell in EBNF

Matching-Typen

Dieser abstrakte Datentyp stellt die Wurzel der Grammatik dar und besteht aus mindestens einem und höchstens zwei Matching-Typen. Damit wird dem Anwendungssystem die Möglichkeit gegeben, anzugeben, von welchem Matching-Typ die jeweiligen Profile sind. Dabei kann ein Profil vom Matching-Typ Firma, Bewerber, Produkt oder auch Auto sein.

Matching-Typ

Ein Matching-Typ kapselt alle Matching-Profile eines gemeinsamen Typs.

Profil

Der abstrakte Datentyp Profil besteht wie in *Tabelle 18* dargestellt aus einem Namen, einer ID und aus Matching-Kriterien, die beschreiben, was das jeweilige Profil anbietet bzw. sucht.

Matching-Kriterien

Der Typ Matching-Kriterien wird entweder durch eine Menge von Angeboten oder Nachfragen repräsentiert.

Nachfrage

Eine Suchanfrage besteht im externen Metamodell aus einem Attribut, einer Attributgewichtung und einer Ähnlichkeitsrelation.

Angebot

Ein Angebot wird einfach durch ein Attribut repräsentiert, welches wie in den Matching-Grundlagen eine Attributbezeichnung auf einen Inhalt abbildet.

6.2.1.2. Externes Metamodell als XML-Schema

Dieses externe Metamodell beschreibt die Struktur der eingehenden XML-Daten indem das externe Metamodell aus *Tabelle 18* als Grundlage verwendet wird. Hierbei sei darauf hinzuweisen, dass das Json-Schema nicht weiter betrachtet wird, da es nicht realisiert worden ist. Um die Struktur der eingehenden XML-Dokumente zu definieren, werden dafür die Sprachelemente der Schemasprache XSD verwendet. Dabei werden die abstrakten Datentypen aus *Tabelle 18* als komplexe XSD-Elemente definiert. In den folgenden Abbildungen werden dabei die wichtigsten XSD-Elemente aufgelistet. Die Folgende Abbildung zeigt, wie der abstrakte Datentyp *Profil* aus *Tabelle 18* als komplexes XSD-Element spezifiziert werden kann.

```
1. <complexType name="Profil">
2.    <sequence>
3.       <element name="name" type="string"></element>
4.       <element name="id" type="int"></element>
5.       <element name="matchingKriterien"type=
6.       "tns:MatchingKriterien"></element>
7.    </sequence>
8. </complexType>
```

Abbildung 40: XSD-Element Profil

Die folgende Abbildung zeigt, wie der abstrakten Datentyp Nachfrage als XSD-Elemente umgesetzt worden ist.

```
1. <complexType name="Nachfrage">
2.    <sequence>
3.      <element name="attribut" type="string"></element>
4.      <element name="gewichtung" type="string"></element>
5.      <element name="nachfrageInhalt"
6.               type="tns:NachfrageInhalt"></element>
7.      <element name="ordnungsRelation"
8.               type="tns:OrdnungsRelation"
9.         maxOccurs="1" minOccurs="0"></element>
10.        <element name="nachfrage" type="tns:Nachfrage"
11.      maxOccurs="unbounded" minOccurs="0"></element>
12.   </sequence>
13.   </complexType>
```

Abbildung 41: XSD-Element Nachfrage

Im Folgenden wird ein XML-Dokument vorgestellt, welches die Matching-Daten beschreibt.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2.   <matchingTypen xmlns="http://www.example.org/matching">
3.    <matchingType>
4.     <name>Firma</name>
5.     <profil>
6.      <name>Akra</name>
7.      <id>1</id>
8.      <matchingKriterien>
9.        <angebot>
10.         <attribut>Gehalt</attribut>
11.          <angebotsInhalt>
12.            <numeric>3000</numeric>
13.          </angebotsInhalt>
14.        </angebot>
15.       </matchingKriterien>
16.     </profil>
17.    </matchingType>
18.  </matchingTypen>
```

Abbildung 42: Matching-Profil als XML-Dokument

Durch den Einsatz von XSD ist die Voraussetzung geschaffen, dass die eingehenden XML-Dokumente vom Anwendungssystem gegen die Spezifikation des externen Metamodells validiert werden können.

Durch die generische Schnittstelle gewinnt man folgende Aspekte:

- Flexibilität der Matching-Daten
  Durch das externe Metamodell ist die Art des Matching-Problems irrelevant.
- Technologie des Anwendungssystem ist unerheblich
  Durch HTTP und XML ermöglicht man eine Interaktion zwischen heterogenen Systemen.

6.2.2 Komponenten

6.2.2.1. Matching-Master

Diese Komponente realisiert das REST-Prinzip mithilfe der vom Lift-Framework bereitgestellten Bibliotheken. Dazu wird, wie in den Grundlagen des Lift-Frameworks aus *Kapitel 2.4.4* beschrieben, das Scala-Trait *RestHelper* verwendet, welches die REST-Methoden unterstützt. Dadurch, dass dieses Trait zusätzlich das Feature *respondAsync* zur asynchronen Kommunikation anbietet, wird erreicht, dass bei jeder Anfrage des Anwendungssystems ein Worker-Thread gestartet wird, um die Matching-Anfrage asynchron zu bearbeiten. Mit Hilfe dieses Features ist es leicht das Prinzip des parallelen Servers umzusetzen, um damit gemäß der Anforderungsspezifikation aus *Kapitel 6.1* ein skalierbares Matching-System zu entwickeln. Der folgende Scala-Code zeigt die Realisierung des Matching-Masters als ein Singleton-Objekt, welches von der Lift-Bibliothek *RestHelper* abgeleitet ist.

```
1. object MatchingMaster extends RestHelper {
2. serve {
3.       case "matching" :: _ XmlPost xml -> _  => {
4.         S.respondAsync {
5.         val ergenis: Elem = bearbeiteAnfrage(xml)
6.         Full(XmlResponse(<t>{ergebnis}</t>)) }}
7.       case "matching" :: _ JsonPost json -> _ => {...}
     }
   }
```

Abbildung 43: Matching-Master

Die *Abbildung 43* zeigt, dass bei den HTTP-Anfragen zwischen den Datenformaten XML und Json unterschieden wird. Je nachdem, welches Datenformat übergeben wird, wird der entsprechende Scala-Code ausgeführt. Hierbei ist darauf hinzuweisen, dass im praktischen Teil dieser Arbeit nur die XML-Schnittstelle implementiert wurde. Um auch Daten im Json-Format verarbeiten zu können, muss das System

lediglich um einen Validierungsmechanismus und eine Konvertierungskomponente erweitert werden.

Da der Dienst des Matchings unter einer speziellen URL angeboten wird, muss das Anwendungssystem seine Matching-Anfrage an die URL www.matching-Modul.de/matching senden, damit der Matching-Master die Anfrage auf den dazugehörigen Scala-Code abbilden kann. Sobald das Anwendungssystem dem Matching-System über das HTTP- Transportprotokoll eine POST-Anfrage an die richtige URL mit der Endung *\matching* sendet, empfängt der Matching-Master diese und erzeugt durch die Anweisung *in Zeile 4* einen Worker-Thread, der sich mit der Bearbeitung der Anfrage beschäftigt.

6.2.2.2. Matching-Worker

Diese Ausführungseinheit ruft abhängig vom eingehenden Datenformat die Dienste des Matching-Systems in der in *Tabelle19* definierten Reihenfolge auf. Sobald das Ergebnis ermittelt worden ist, verpackt der Worker das Resultat in eine *XmlResponse* und schickt es dem Anwendungssystem zurück.

Dienst	Scala-Code zur Realisierung des Aufrufs
Validierungsservice	`validerungsService.validate(xmlFile,xmlSchema)`
Parsing-Service	`parsingService.parseXML(xmlFile)`
Matching-Modul	`matchingModul.matching(matchingDaten)`
Parsing-Service	`parsingService.parseToXML(matchingErgebnis)`

Tabelle 19: Aufrufreihenfolge der Dienste des Matching-Systems

6.2.2.3. Validierungs-Service

Die Validierungskomponente ist als ein Singleton-Objekt realisiert, welches die öffentliche Methode `validiere()` mit folgender Signatur implementiert:

```
validiere(xmlFile: File,xmlSchema: File): File
```

Abbildung 44: Schnittstelle des Validierungsservices

Um ein XML-Dokument gegen ein XML-Schema zu valideren, wird intern in der Methode aus *Abbildung 44* auf Java Bibliotheken, wie *SchemaFactory* und *Validator* aus dem *javax.xml.validation* Package zugegriffen um den Validierungsmechanismus zu implementieren. Wenn die Prüfung erfolgreich war, wird das XML-Dokument zurückgegeben. Andernfalls werden entsprechende Fehlermeldungen vom Validierungs-Package geworfen.

6.2.2.4. Parsing-Service

Diese Komponente ist zum einen für das Konvertieren zwischen XML und Scala verantwortlich und zum anderen für die Denormalisierung des externen Metamodells. Dabei wird wie im *Kapitel 6.2.1.1* beschrieben das externe Metamodell aus Effizienzgründen in das interne Metamodell aus *Kapitel 4.2* überführt. Der Parsing-Service bietet wie in der nächsten Abbildung ersichtlich, zwei öffentliche Methoden an, die zum einen ein XML-Dokument in eine Scala-Struktur bringt und zum anderen ein Matching-Ergebnis von Scala nach XML konvertiert.

```
1. parseXML(xmlFile: File): List[MatchingKriterium]
2. parseToXML(matchingErgebnis: List[MatchingErgebnis]) : Elem
```

Abbildung 45: Schnittstelle des Parsing-Services

6.2.2.5. Datenbanksystem

Im Folgenden wird eine Möglichkeit vorgestellt, die Daten innerhalb des Matching-Moduls persistent verwalten zu können. Diese Komponente wurde nicht im praktischen Teil umgesetzt. Jedoch soll zumindest erwähnt werden, wie man eine persistente Speicherung der Matching-Daten erreicht.

Dazu wird ein Datenbankschema für ein relationales Datenbanksystem, wie MySQL oder Oracle, in der Form des internen Metamodells angelegt, um die Matching-Daten und Matching-Ergebnisse zu speichern. Der Matching-Worker muss über eine Datenbankschnittstelle Zugriff auf diese Datenbank haben, um die eingehenden Matching-Daten nach dem XML-Scala Konvertierungsvorgang persistent zu speichern. Mit diesem Konzept bietet es dem Anwendungssystem die Möglichkeit, die Matching-Daten im Webservice zu speichern und zu einem späteren Zeitpunkt erst über eine weitere Anfrage den Matching-Prozess anzustoßen. Dazu müsste lediglich die Schnittstelle des Matching-Masters angepasst werden, indem eine URL auf den Matching-Prozess abgebildet wird und eine weitere um die Matching-Daten persistent zu speichern. Es müsste lediglich eine Case-Anweisung hinzugefügt werden, um die Matching-Daten im Webservice zu speichern, was folgende Abbildung demonstriert.

```
case "matching" :: _ XmlPost xml -> _  => {…}
```

Abbildung 46: Erweiterung der Matching-Schnittstelle

6.3 Evaluierung

6.3.1.1. Starten des Webservers

Um das Matching-System auf einem beliebigen Rechner zu starten, muss lediglich das Build-Management-Tool Maven installiert sein. Sobald dies geschehen ist, kann der Lift-Webservice mithilfe des vom Webframework mitgelieferten Anwendungsservers Jetty ausgeführt werden. Dazu sind folgende Konsolen-Befehle notwendig:

Anweisung	Beschreibung
cd MatchingSystem	Gehe in das Projektverzeichnis des Matching-Systems
mvn jetty:run	Starte Jetty auf Port 8080

Tabelle 20: Deploy[35]-Vorgang des Matching-Systems

Diese Befehle führen zu folgender Konsolenausgabe:

```
2011-08-11 17:52:03.818:INFO::Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 5 seconds.
```

Abbildung 47: Konsolenausgabe - Start des Matching-Systems

Durch das Maven-Jetty-Plugin ist es möglich den Jetty-Anwendungs- und Webserver über Maven zu starten.

[35] Auslieferung eines Software Systems

6.3.1.2. Beispiel einer Matching-Anfrage

Ein Anwendungssystem, welches den Matching-Dienst des Webservice beanspruchen will, muss nur folgende zwei Aspekte beachten:

- Matching-Daten müssen dem externen Metamodell entsprechen
- Übertragung der Daten muss über das HTTP-Protokoll erfolgen

Dabei ist noch einmal darauf hinzuweisen, dass es unerheblich ist, in welcher Programmier-sprache das Anwendungssystem implementiert worden ist.

Das folgende Anwendungssystem baut eine Verbindung mit dem Webservice auf. Die Kommunikation geschieht über das HTTP-Protokoll, über dem die XML-Daten übertragen werden. Die folgende Abbildung zeigt, wie ein Java-Client mithilfe der HTTP-Bibliotheken von *Apache* mit dem Webservice kommunizieren kann.

```
1. import java.io.*;
2. import org.apache.http.*;

3. public class JavaClient {

4.  public static void main(String[] args){
5.      File xml = new File("matchingDaten.xml");
6.      HttpClient client = new DefaultHttpClient();
7.      HttpPost post = new HttpPost("http://Adresse/matching");
8.      StringEntity entity =
            new StringEntity(xml,"application/xml",HTTP.UTF_8);
9.      post.setEntity(entity);
10.     HttpResponse response = client.execute(post);
    }

}
```

Abbildung 48: Java-Client

Diese Abbildung zeigt, wie man mit einer Java-Anwendung das Matching-System ansprechen kann. Dazu wird eine HTTP-Verbindung mit dem Webservice aufgebaut. Da der Matching-Master, wie in *Abbildung 43* dargestellt, den Matching-Dienst unter einer bestimmten URL mit der Endung */matching* bereitstellt, ist es hierbei wichtig, dass der Java-Client den Service mit der richtigen URL anspricht. Die Matching-Anfrage wird in *Zeile 10* mittels der Methode `execute(…)` an das Matching-System gesendet und liefert das Matching-Ergebnis als *HttpResponse*-Datentyp zurück.

Die folgende Abbildung zeigt ein Auszug aus dem Matching-Ergebnis von der Matching-Anfrage aus *Abbildung 48*.

```
1.    <matchingErgebnis>
2.          <profil>
3.                <name>AKRA</name>
4.                <id>1</id>
5.          </profil>
6.          <profil>
7.                <name>Max Mustermann</name>
8.                <id>2</id>
9.          </profil>
10.         <matchingGrad>0.8863636363636362</matchingGrad>
11.   </matchingResult>
```

Abbildung 49: Matching-Ergebnis des Java-Clients

Dabei werden zwei Profile mit dem dazugehörigen Matching-Grad dem Java-Client zurückgegeben. Sobald die Matching-Ergebnisse eingetroffen sind, kann das jeweilige Anwendungssystem die Ergebnisse auf dessen Benutzeroberfläche darstellen.

6.3.1.3. Performanz

Um die Performanz des Matching-Systems zu evaluieren, werden zwei Anwendungsszenarien vorgestellt, bei denen die Matching-Arten aus Anforderungen *9* und *10* aus *Kapitel* 3 berücksichtig wurden.

Testumgebung

In den folgenden zwei Anwendungsfällen wird der Webservice auf einem MacBook, welcher mit einem 2 GHz Intel Core 2 Duo und mit 2 GB Arbeitsspeicher ausgestattet ist, gestartet.

Szenario 1

Dieser Anwendungsfall richtet sich an die *Anforderung 9*, die ein Matching von Profilen des gleichen Typs verlangt. Hierbei werden Firmen miteinander verglichen und deren Ähnlichkeit durch das Matching-System berechnet. Dabei wird das Anwendungssystem in Java realisiert, welches den Webservice über HTTP die Matching-Daten im XML-Format übermittelt.

Profile	Nachfragen	Angebote	Profilpaare	Antwortzeiten
50	3	3	2450	2379 ms
50	7	7	2450	2120 ms
100	3	3	9700	8263 ms
100	7	7	9700	13496 ms

Tabelle 21: Evaluierung Szenario 1

Man erkennt in der *Tabelle 21*, wie stark die Anzahl der Profilpaare steigt und damit die Antwortzeiten erhöhen. Hierbei wird die Anzahl der Profilpaare durch das Matching-System selbst ermittelt, indem die Anzahl der Matching-Ergebnisse gezählt werden. Man erkennt ebenfalls, dass durch die Erhöhung der Attribute eines Profils die Antwortzeiten nicht stark erhöhen. Dies liegt an der Beschaffenheit des Algorithmus, da die Anzahl der verschiedenen Attribute seinen Parallelisierungsgrad bestimmt.

Szenario 2

Hierbei werden anhand des Anwendungsfalls einer Partnervermittlung Profile unterschiedlichen Matching-Typs miteinander verglichen und deren Übereinstimmungen ermittelt.

Profile	Angebote pro Profil	Nachfragen pro Profil	Profilpaare	Antwortzeiten
34 : 34	2 : 4	4 : 2	2312	3704 ms
1 : 34	2 : 4	4 : 2	68	952 ms
34 : 1	2 : 4	4 : 2	68	932 ms
1 : 100	2 : 4	4 : 2	200	1144 ms

Tabelle 22: Evaluierung Szenario 2

Die Tabelle zeigt, wie ein 1 : n Matching mithilfe dieser Matching-Art bewerkstelligt werde kann. Die Länge der Antwortzeiten ist abhängig von der Menge der Profilpaare, wie die *Tabelle 22* zeigt. Dadurch, dass die Menge der Profilpaare bei einem 1 : n Matching nur linear ansteigt, erzielt man akzeptable Ergebnisse. Bei einem n : m Matching steigt sich die Anzahl der Profilpaare quadratisch und erzielt deshalb wesentlich längere Antwortzeiten als bei einem 1 : n Matching.

7 Schlussbetrachtung

In diesem Abschnitt werden noch einmal die Kernaussagen dieser Arbeit zusammengefasst und anhand eines Ausblicks die Möglichkeiten zur Weiterentwicklung erläutert.

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde sich intensiv mit dem Themengebiet Matching in Marketplace-Systemen beschäftigt, da sich mehr und mehr Geschäftsprozesse online abbilden lassen. Zu einem sehr wichtigen Prozess gehört der Matching-Mechanismus, der Angebots- und Nachfrageprofile miteinander vergleicht und deren Ähnlichkeit berechnet. Diese Arbeit hat gezeigt, wie sich das Matching-Problem für Marketplace-Systeme anhand eines mathematischen Modells mithilfe von Bewertungsfunktionen formal spezifizieren lässt. Die Intention dieser Arbeit war es ein Modul zu entwickeln, welches die Funktionalität des Matchings unabhängig vom Anwendungsproblem umsetzt. Um die generische Eigenschaft des Matching-Moduls zu realisieren, wurde das Prinzip der Metamodelle eingesetzt, um eine universelle Struktur der Matching-Daten zu spezifizieren. Damit ist es möglich, dass Marketplace-Systeme verschiedener Matching-Probleme wie Job-Börsen oder Partnervermittlungen diesen Dienst über eine generische Schnittstelle nutzen können.

Da heutzutage versucht wird, die Geschwindigkeit einer Anwendung dadurch zu erhöhen, indem mehr Prozessorkerne eingesetzt werden, müssen die Programmierparadigmen entsprechend angepasst und erweitert werden. Das Konzept der parallelen Programmierung wurde eingesetzt, um den Matching-Algorithmus mithilfe des Aktor-Modells zu implementieren. Scala ist eine moderne JVM-basierende Programmiersprache, die die Konzepte der objektorientierten und funktionalen Programmierung vereint. Sie bietet die Möglichkeit mithilfe des Aktor-Modells, einen parallelen Matching-Algorithmus auf hohem Abstraktionsniveau zu implementieren. Durch diese Eigenschaften hat sich die Programmiersprache Scala gegen andere Kandidaten wie Java, C# oder Ruby durchgesetzt.

Des Weiteren wurde eine Systemarchitektur vorgestellt, die das Matching-Modul integriert und die Voraussetzungen anhand von Hilfskomponenten schafft, um es von einem beliebigen Anwendungssystem über die Standardtechnologien des Internets HTTP und XML ansprechen zu können. Durch die Realisierung des Matching-Systems als Webservice, kann diese Webanwendung auf einem beliebigen System gestartet werden, um Matching-Anfragen zu empfangen. Letztendlich repräsentiert die Umsetzung dieses Matching-Moduls keine typische Lösung, um ein Matching-System zu gewährleisten. Diese Arbeit zeigt nicht wie man mittels

Datenbankoptimierungen das effizienteste Matching erzielt. Aus den Ergebnissen aus *Kapitel 6.3.1.3* ist ersichtlich geworden, dass das Profil-Matching des gleichen Matching-Typs erheblich lange Antwortzeiten benötigt und ein Online-Matching nicht gewährleisten kann. Dagegen bietet die zweit Art des Matchings nach *Anforderung 10* schon die Möglichkeit des Online-Matchings, da insbesondere die Latenzzeiten beim 1 : n Matching nur linear ansteigen. Die Arbeit zeigt eine Lösung, die mithilfe einer sehr modernen Programmiersprache das Matching-Problem in Marketplace-Systemen unabhängig vom Anwendungsproblem auf einer höheren Systemebene realisiert.

7.2 Ausblick

Wie schon bereits erwähnt, wurden nicht alle Komponenten der Systemarchitektur aus *Abbildung 39* technisch realisiert. Diese nicht implementierten Komponenten wurden dennoch der Architektur hinzugefügt, um die Interaktion des Matching-Moduls mit anderen Komponenten zu demonstrieren. Im Folgenden wird noch einmal kurz darauf eingegangen, welche Möglichkeiten der Weiterentwicklung es gibt.

Semantisches Matching

Die Einführung eines semantischen Matchings mittels auf Ontologie basierter Verfahren könnte dazu verwendet werden, um Texte miteinander zu vergleichen und deren Ähnlichkeitsgrad zu berechnen.

Persistentes Matching

Durch die Einrichtung eines Datenbanksystems kann eine persistente Verwaltung der Matching-Daten gewährleistet werden. Dadurch besteht die Möglichkeit, dass ein Anwendungssystem die Matching-Daten im Webservice persistent speichern kann und Daten hinzufügen kann. Mit entsprechender Erweiterung der REST-Schnittstelle des Matching-Systems könnte auch die Änderbarkeit der Daten berücksichtigt werden.

Google App Engine

Installierung und Starten des Matching-Systems als Webservice auf der Google-App-Engine. Sie beschreibt nach (Google, 2011) eine Infrastruktur, um Webapplikationen zu starten und um den permanent Zugriff zu gewährleisten.

Erweiterung der Datenaustauschformate

Da wie bereits erwähnt, unterstützt das Matching-System zurzeit nur XML als Format der Matching-Daten. Um die Schnittstelle des Matching-Systems noch flexibler zu gestalten, könnten die Formate Json und YAML ebenfalls berücksichtigt werden. Dazu müsste man dem Matching-System um entsprechende Parsing-Services und Validierungsmechanismen für die jeweiligen Formate erweitern.

Webservice für mobile Applikationen nutzen

Dadurch, dass die etablierten Internetstandards wie HTTP und XML zur Realisierung des Webservice eingesetzt worden sind, ist die Technologie des Anwendungssystems unerheblich. Dies bietet die Möglichkeit, dass mobile Anwendungssysteme den Dienst des Matching-Systems anfordern können.

Erweiterung des Metamodells

Des Weiteren besteht die Möglichkeit das Metamodell so zu erweitern, dass mehr Attributtypen, wie zum Beispiel eine Umkreissuche, berücksichtigt wird.

Einsetzung eines Generators

Die Realisierung eines Generators kann die Aufbereitung der Matching-Daten für ein Anwendungssystem unterstützen. Dazu kann der Generator aus dem externen Metamodell eine HTML-Schablone generieren, in der das Anwendungssystem dessen spezifischen Metadaten eintragen kann.

Die eben genannten Möglichkeiten zur Weiterentwicklung bietet diese Arbeit um die Funktionalität des Matching-Systems zu erhöhen.

Literaturverzeichnis

1 Wikipedia . 2011. Wikipedia. [Online] 16. August 2011. [Zitat vom: 17. August 2011.] http://de.wikipedia.org/wiki/JavaScript_Object_Notation.

2 Wikipedia. 2011. XML Schema. [Online] 2011. [Zitat vom: 17. August 2011.] http://de.wikipedia.org/wiki/XML_Schema.

Apache . 2011. Apache. [Online] 2011. [Zitat vom: 17. August 2011.] http://maven.apache.org/.

Arno Haase, Markus Völter. 2010. itermis. [Online] 3. Mai 2010. [Zitat vom: 11. August 2011.] http://www.itemis.de/itemis-ag/veranstaltungen/2010/language=de/taps=646/29541/durchstarten-mit-scala.

Derek Chen-Becker, Marius Danciu, Tyler Weir. 2011. *Exploring Lift.* 2011.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1997. *Design Patterns Elements of Reusable Object-Oriented Software.* Oxford : s.n., 1997.

Esser, Friedrich. 2010. *Living Actos.* Hamburg : HAW, 2010.

Fielding, Roy Thomas. 2000. *Architectural Styles and the Design Of Network-based Software Architectures.* Irvine : s.n., 2000.

Google. 2011. Google App Engine. [Online] 2011. [Zitat vom: 20. August 2011.] http://code.google.com/intl/de-DE/appengine/.

Heidt, M.B. *Lind und Tuple Spaces.* Marbug : s.n.

Heidt, Michael Benjamin. 2005. *Linda und Tuple Spaces.* Marburg : s.n., 2005.

J.M.Joller. *Java Spaces - Konzepte und Beispiele.*

Jetty. 2011. [Online] 2011. [Zitat vom: 17. August 2011.] http://jetty.codehaus.org/jetty/.

Liam Quin. 2011. [Online] W3C, 23. April 2011. [Zitat vom: 17. August 2011.] http://www.w3/.org/XML.

Martin Odersky, Lex Soon, Bill Venners. 2010. *Programming in Scala Second Edition.* 2010.

Michael Scherer. [Online] [Zitat vom: 15. 08 2011.] http://ki.informatik.uni-wuerzburg.de/teach/ws-2004-2005/vki-coord/MichaelScherer.pdf.

Odersky, Martin. 2010. *JAX TV.* 15. April 2010.

—. 2009. *Scala by Example.* Schweiz : Programming Methods Laboratory, 2009.

Prankratius. *Software Engineering für moderne, parallele Plattformen.* Karlsruhe : s.n.

Sarstedt, Stefan. 2010. *Architektur von Informationssysteme - Architekurstile 2.* Hamburg : HAW Hamburg, 2010.

—. 2010. *Architektur von Informationssystemen Archtekturstile und Design Methodologien.* Hamburg : HAW, 2010.

Schmidt, Andreas. 2011. *Modell/Metamodelle.* Kalsruhe : s.n., 2011.

Simon Peyton Jones, Satnam Singh. 2008. *A Tutorial on Parallel and Concurrent Programming in Haskell.* Cambridge : s.n., 2008.

Timothy G. Mattson, Berverly A. Sanders, Berna L. Massingill. 2008. *Patterns for parallel Programming.* s.l. : Addsion-Wesley Professional, 2008.

Vries, Andreas de. 2009. *Mathematical model of interest matchmaking in electronic social networks.* Hagen : s.n., 2009.

Wikipedia. 2011. Wikipedia. [Online] 2011. http://de.wikipedia.org/wiki/Intervall_%28Mathematik%29.

Zimmermann, Bern. 2010. E-Business/E-Commerce. [Online] 1. Januar 2010. [Zitat vom: 11. August 2011.] http://www.www-kurs.de/e-business.htm.

Anhang

Auf der mitgelieferten CD befindet sich das in *Kapitel 6* beschriebene Matching-System. Das Wurzelverzeichnis *MatchingSystem* beinhaltet alle Scala-spezifischen Elemente zur Umsetzung des praktischen Teils dieser Arbeit.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____