Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Verbesserung der Konsistenz von Anforderungsdokumenten durch Natural Language Processing

Bachelorarbeit

im Studiengang Informatik

von

Thomas Winter

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Joel Greenyer
Betreuer: Dipl.-Math. Olga Liskin

Hannover, 02.11.2015

Zusammenfassung

In Software-Projekten werden häufig verschiedene Darstellungsformen für Anforderungen parallel verwendet, um die Vorteile beider Formen zu nutzen. Oft werden für eine Anforderung UseCases und Story Cards aufgrund ihrer ähnlichen Struktur parallel genutzt und sollten nach Möglichkeit miteinander verknüpft werden.
Diese Arbeit stellt eine Methode vor, um die Verknüpfung von UseCases und Story Cards im Projekt zu vereinfachen, indem Verknüpfungsmöglichkeiten berechnet und vorgeschlagen werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es gibt viele verschiedene Möglichkeiten, Anforderungen an ein System festzuhalten und zu dokumentieren. Oft werden verschiedene Formen parallel genutzt, um sich die Vorteile beider Darstellungsformen zu sichern und die Nachteile auszugleichen. Laut Berry [1] ist die überwältigende Mehrheit von Anforderungsdokumenten in natürlicher Sprache verfasst und da es bereits Verfahren zur automatischen Auswertung von Anforderungsdokumenten in formaler Sprache (z.B. Klassendiagramme) gibt, ist es vor allem interessant, die natürlichsprachlichen zu untersuchen. Zwei ähnliche natürlichsprachliche Darstellungsformen, die häufig innerhalb eines Projektes verknüpft werden, sind UseCase-Diagramme und Story Cards (UserStories). Bei hinreichend umfangreichen Projekten kann sich der manuelle Vergleich zwischen diesen zu einem sehr zeitaufwändigen Unterfangen entwickeln. Um dem Requirements Engineer oder auch Requirements Analyst dies zu erleichtern, ist ein automatischer Prozess denkbar, der die Anforderungsdokumente im Projekt analysiert und dem Requirements Engineer auf Wunsch verwandte Dokumente vorschlägt. Im Rahmen des Requirements Engineering ist dieser Prozess ungefähr im Bereich der Anforderungsanalyse (Requirements Analysis) einzuordnen. Ein solcher Prozess ist im Rahmen dieser Arbeit entwickelt und realisiert worden und schlägt beispielsweise zu einem UseCase-Schritt verwandte UserStories vor, für die eine hinreichend große Übereinstimmung errechnet wurde. Ziel der Arbeit ist es also, das Vergleichen von zahlreichen Anforderungsdokumenten zu vereinfachen, wodurch ermöglicht wird, schneller und effizienter Verknüpfungen zwischen ähnlichen/verwandten Dokumenten zu setzen. Bei Veränderungen der Dokumente müssen diese in der Regel neu abgeglichen und angepasst werden, wodurch Zusatzkosten entstehen. Durch das Verknüpfen von Anforderungen können diese Zusatzkosten minimiert werden, wenn darauf hingewiesen wird, dass bei Veränderung eines Dokumentes mit bestehender Verknüpfung auch die verknüpften Dokumente angepasst werden sollten. Daraus ergibt sich eine Verbesserung der Rückverfolgbarkeit (Traceability) und eine höhere Konsistenz zwischen den

Dokumenten.

Gliederung der Arbeit Die Arbeit ist in insgesamt sechs Kapitel gegliedert, wobei dieses die Einleitung darstellt:

- In **Kapitel 2** werden allgemeine Grundlagen besprochen, die für das Verständnis der weiteren Kapitel hilfreich sind.
- **Kapitel 3** beschreibt die Herleitung eines Konzeptes, um die Ähnlichkeit zwischen Anforderungsdokumenten zu bestimmen.
- In **Kapitel 4** wird die Implementierung des Konzeptes dargestellt und erläutert.
- **Kapitel 5** wertet die Ergebnisse der Methode aus und evaluiert sie.
- In **Kapitel 6** werden abschließend die gewonnenen Erkenntnisse zusammengefasst.

Kapitel 2

Grundlagen

In diesem Kapitel sollen einige für das Verständnis der folgenden Kapitel wichtigen Begriffe kurz erklärt werden.

AADs Im Rahmen dieser Arbeit werden Anforderungsdokumente unterschiedlicher Typen miteinander verglichen. Dabei handelt es sich konkret um UseCaseSteps in UseCases und UserStories in UserTasks bzw. in einer Story Map. Da diese Anforderungsdokumente hierarchisch gegliedert sind und wir uns auf den Vergleich der untersten Hierarchieebene, also die elementarste Form, beschränken, wollen wir diese im Folgenden als

atomare Anforderungsdokumente, kurz AADs

bezeichnen.

NLP-Konzepte Die Arbeit baut auf den Techniken des Natural Language Processing, kurz NLP, auf, wobei drei Begriffe besonders Beachtung finden. Diese drei Begriffe werden anhand des folgenden Beispiels erklärt:

'Falls das Datum weit in der Zukunft liegt, kann er auch das gewünschte Datum direkt über einen Datepicker wählen.'

Tokenization Tokenization bezeichnet den Prozess, aus einem natürlichsprachlichen Text einzelne Wörter, Konzepte oder Sätze zu extrahieren, die dann allgemein als Token bezeichnet werden. Dabei muss entschieden werden, wo ein Token beginnt und endet (üblicherweise Leerzeichen oder Satzzeichen). Gerne wird auch die Interpunktion (Kommata, Punkte, Klammern) verworfen und sämtliche Tokens grundsätzlich kleingeschrieben. Ein

weiterer Aspekt bei der Tokenization ist das *Compound-Breaking*, also das Auflösen von Wort-Verbunden. Das Beispiel sähe unter Beachtung dieser Punkte und mit Compound-Breaking von 'Datepicker' nach Tokenization so aus:

> 'falls das datum weit in der zukunft liegt kann er auch das gewünschte datum direkt über einen date picker wählen'

Stemming Durch das Stemmen von Tokens wird eine Art Wortstamm gebildet, den sich eng verwandte Wörter teilen. Die einfachste Anwendung von Stemming wäre es, Singular und Plural des gleichen Wortes auf den selben Wortstamm zu stemmen (beispielsweise 'datum' und 'daten' beide zu 'dat'). Einen passenden gemeinsamen Wortstamm zu finden, ist nicht immer einfach, denn es ist möglich, zu viel von den Token wegzuschneiden, sodass sie mit ganz anderen Token verwechselt werden (beispielsweise 'datum' und 'daten' nur zu 'da', was wiederum eine völlig andere Bedeutung hat). Dieses Problem wird als *Overstemming* bezeichnet. Was und wie weit gestemmt wird, hängt vom konkreten Stemming-Verfahren ab, das Beispiel sähe idealerweise so aus:

> 'fall das dat weit in der zukunft lieg kann er auch das gewünscht dat direkt über ein dat pick wähl'

Stop-Word-Removal Hiermit wird die Notwendigkeit beschrieben, Tokens, die keinen oder nur einen geringen Beitrag zur Beschreibung des Inhaltes liefern, aus einem natürlichsprachlichen Text herauszufiltern. Typische deutsche *Stop Words* sind z.B. 'ein', 'wieder', 'mit' und 'als', wobei sich eine recht vollständige Sammlung von *Stop Words* leicht finden lässt[1]. Von dem Beispiel würde nach Stop-Word-Removal nur noch das Folgende übrigbleiben:

> 'fall dat weit zukunft lieg gewünscht dat direkt dat pick wähl'

Linguistische Schwierigkeiten Bei der Analyse von natürlichsprachlichen Dokumenten ist zunächst die verwendete Sprache entscheidend. Viele Methoden im NLP bauen auf Wörterbüchern oder ähnlichem auf, um genau arbeiten zu können, weshalb Probleme auftreten können, wenn Worte mehrerer Sprachen im gleichen Dokument oder sogar Korpus (Sammlung

[1] `http://www.ranks.nl/stopwords/german`

aller Dokumente) auftauchen. Die genauesten Ergebnisse werden erreicht, wenn im gesamten Korpus ausschließlich eine Sprache verwendet wird, was in der Praxis jedoch extrem unwahrscheinlich ist. Oft werden Mischwörter aus Deutsch und Englisch gebildet und es sind viele ursprünglich englische Wörter in den allgemeinen deutschen Sprachgebrauch gerutscht. Im oben betrachteten Beispiel findet sich z.B. das englische Wort 'Datepicker', was zeitgleich auch einen Wortverbund darstellt. Die Sprache ist bei der Tokenization noch nicht entscheidend, bei Stemming und Stop-Word-Removal aber sehr wohl. Da aber dennoch üblicherweise der überwiegende Teil eines Korpus in der gleichen Sprache verfasst ist und verhältnismäßig wenig Wörter aus anderen Sprachen auftauchen, halten sich die negativen Auswirkungen in der Regel in Grenzen.
Eine weitere Schwierigkeit stellen pronomiale Anaphern dar. Damit sind Pronomen (er, sie, es) gemeint, die sich auf ein vorher genanntes Subjekt beziehen, welches aber im betrachteten Kontext nicht eindeutig zugeordnet werden kann. Betrachten wir als Beispiel den folgenden natürlichsprachlichen Text:

> 'Der Administrator trägt Nutzer, die Zugang zum System haben, ein. Er kann ihre Daten nachträglich ändern.'

Nimmt man die Sätze einzeln her, geht aus dem zweiten Satz allein nicht hervor, auf wen sich 'Er' bezieht.
Außerdem spielen besonders in der deutschen Sprache Verbunde aus Wörtern (Kompositionen) eine Rolle. Typische Verbunde sind z.B. 'Radfahrer' oder 'Dampfschiff'. Im Deutschen darf man Kompositionen auch mit Bindestrich schreiben, falls dadurch die Lesbarkeit verbessert wird, z.B. 'Heimkino-Installation'. Diese zwei Schreibweisen für Kompositionen werden in verschiedenen Methoden des NLP (z.B. Stemmer) unterschiedlich behandelt, wobei die Schreibweise ohne Bindestrich intuitiv schwerer aufgelöst werden kann.

Precision und Recall Damit werden im Information Retrieval zwei verbreitete Methoden bezeichnet, um die zu einer Anfrage (Query) ausgegebenen Dokumente zu bewerten. Dabei werden vier Fälle betrachtet, die in Tabelle 2.1 dargestellt sind: Die True Positives (TP) und True Negatives (TN) sind dabei die korrekt ausgegebenen bzw. nicht ausgegebenen Dokumente, während die False Positives (FP) und False Negatives (FN) entsprechend die Irrtümer in der Ausgabe repräsentieren. Man definiert nun die Precision als Maß dafür, wieviele der ausgegebenen Dokumente Treffer waren (also als korrekt gelten) und damit als folgendes Verhältnis:

$$Precision = \frac{\#korrektausgegebeneDokumente}{\#ausgegebeneDokumente} = \frac{TP}{TP + FP} \tag{2.1}$$

	Relevant	Irrelevant
Ausgegeben	True Positives (TP)	False Positives (FP)
Nicht ausgegeben	False Negatives (FN)	True Negatives (TN)

Tabelle 2.1: Die vier Fälle zur Einordnung von Ergebnis-Dokumenten

Der Recall ist hingegen als Maß dafür definiert, wieviele der insgesamt korrekten Dokumente auch ausgegeben wurden. Dies entspricht dem Verhältnis:

$$Recall = \frac{\#korrektausgegebeneDokumente}{\#korrekteDokumente} = \frac{TP}{TP + FN} \tag{2.2}$$

Kapitel 3

Konzeptuelles

Um ein Konzept zum Vergleichen zweier AADs zu entwerfen, um Gemeinsamkeiten zu erkennen und eine Verlinkung vorzuschlagen, sind mehrere Ansätze untersucht worden. Auf dem heutigen Stand der Forschung gibt es zwei Verfahren [6], um die semantische Übereinstimmung zwischen zwei natürlichsprachlichen Texten zu ermitteln: die korpusbasierten Verfahren, welche mittels Methoden des Natural Language Processing versuchen, sich dem Problem auf syntaktisch-lexikalischer Ebene zuzuwenden und die wissensbasierten Verfahren, welche eher die Idee verfolgen, auf semantischer Ebene zu arbeiten und den Inhalten, Konzepten und sogar Relationen eines Textes eine Bedeutung zuzuweisen.

In diesem Kapitel wird zunächst ein wissensbasierter Ansatz vorgestellt und später auf eine korpusbasierte Methode zurückgegriffen.

3.1 Wissensbasierter Ansatz

Wissensbasierte Verfahren zur semantischen Analyse umfassen meist die Benutzung von semantischen Netzen, Taxonomien oder Ontologien, um Konzepten eine Bedeutung zuzuweisen. In semantischen Netzen werden allgemein Konzepte über Kanten miteinander verknüpft, um entsprechende semantische Beziehungen darzustellen. Taxonomien klassifizieren Konzepte in einer Hierarchie (z.B. die Einordnung einer Tierart in Familie, Gattung, Ordnung) und Ontologien fügen Relationen zwischen den Konzepten hinzu.
Breitman et al. sehen Domänen-Ontologien als wertvolles Nebenprodukt des Software Engineering an [2]. Leonid Kof stellt einen Ansatz vor, um Inkonsistenzen zwischen verschiedenen Anforderungsdokumenten besser erkennen zu können [8]. Dazu kombiniert er ursprünglich unabhängig entstandene Verfahren und beschreibt die Erstellung einer Domänen-Ontologie als recht umfangreiches Unterfangen, bei dem die einzelnen Schritte den vormals unabhängigen Verfahren entsprechen, die dann aufeinander aufbauen und sich gegenseitig unterstützen können. Der gesamte Prozess ist dabei in Abbildung

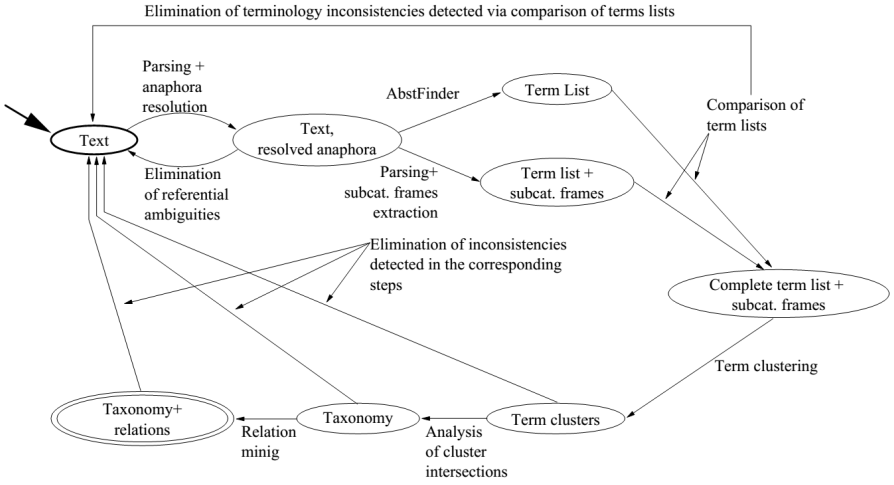

Abbildung 3.1: Konstruktion einer Domänen-Ontologie nach [8]

3.1 zu sehen und wird im Folgenden kurz erläutert.

Zunächst wird aus dem Input-Text mithilfe eines Part-of-Speech-Taggers ein Baum erzeugt, wobei zusätzlich zu den PoS-Annotationen (Worttypus, z.B. Adjektiv, Nomen, Verb)[1][2] auch Subjekt und Objekte extrahiert werden, welche zum Auflösen von pronomialen Anaphern benutzt werden.
Nun muss der anapherfreie Text erneut geparsed werden, wobei mithilfe des Tools AbstFinder [5] mögliche Terme ermittelt und in einer Liste gesammelt werden (ähnlich den Termlisten in Kapitel 3.2), die dann noch mit *Subcategorization Frames* (Objekte/Subjekte als Argumente von Verben) abgeglichen werden, um eine kombinierte/komplette Termliste zu erzeugen.
Aus der kombinierten Termliste (auch diese findet sich in Kapitel 3.2 in ähnlicher Form wieder) werden die Terme mithilfe des Tools ASIUM [4] zu Clustern gruppiert, um eine Taxonomie zu erzeugen, wobei zur Klassifizierung eine Linearkombination der drei von Nenádic et al. vorgeschlagenen Übereinstimmungsarten (kontextuell, lexikalisch und syntaktisch) [12] benutzt wird.
Abschließend werden mittels *Relation Mining* noch Relationen zwischen Termen ermittelt, wobei eine potentielle Assoziation zwischen zwei Termen entsteht, wenn sie im gleichen Satz auftauchen [8].
Bemerkenswert ist, dass der Requirements Analyst nach jedem einzelnen Schritt das Zwischenergebnis auf Inkonsistenzen prüfen muss und seine manuelle Entscheidung den Input für einen weiteren Durchlauf des Prozesses liefert.

[1] Englische PoS-Tags: `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

[2] Deutsche PoS-Tags: `http://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html`

Bei der Benutzung von wissensbasierten Methoden ist Interaktivität daher von entscheidender Bedeutung. Um Inkonsistenzen zu erkennen, sollte an möglichst vielen Stellen des Prozesses der Output des automatischen Teils vom Requirements Analyst manuell gegengeprüft werden.

Eignung Der vorgestellte wissensbasierte Ansatz klingt zunächst vielversprechend und hat den Vorteil, dass linguistische Schwierigkeiten wie pronomiale Anaphern erkannt und beseitigt werden können. Die Übereinstimmungsberechnung aus dem *Term Clustering* (siehe oben) war im Rahmen dieser Arbeit ursprünglich als Anwendung auf den Vergleich von zwei Anforderungsdokumenten vorgesehen. Gan et al. berechnen in ihrer Arbeit ebenfalls eine *Semantic Similarity* zwischen Termen aus Ontologien im Bereich der Biomedizin [7].
Allerdings erfordert der vorgestellte Ansatz an vielen Stellen die Rücksprache mit dem Requirements Analyst. Die Erstellung einer Ontologie eignet sich grundsätzlich eher, wenn ein komplexes, zusammenhängendes Anforderungsdokument vorliegt. Im Rahmen dieser Arbeit sollen nur atomare Anforderungsdokumente (AADs) verglichen werden, die in der Regel aus 1-2 Sätzen bestehen (siehe Anhang A und B). In diesem Kontext erscheint der Aufwand zur Ontologie-Erstellung im Vergleich zum zusätzlichen Nutzen nicht gerechtfertigt. Es ergibt sich ein Tradeoff zwischen Automatisierbarkeit und Erkennung von Inkonsistenzen. Bei hinreichend vielen AADs in einem Projekt sollte daher eher Wert auf ein automatisches Vorgehen gelegt werden, um den manuellen Aufwand und die zeitliche Belastung des Requirements Analyst beim Suchen von ähnlichen AADs zu minimieren.
Es entstand noch die Idee, eine einzige Ontologie für ein gesamtes Projekt zu erzeugen, wobei dann sämtliche Anforderungsdokumente und zugehörige AADs im entsprechenden Projekt als Gesamtheit betrachtet werden könnten. Dazu müsste dann aber zusätzlich noch die Position der AADs im Projekt vermerkt werden, um später spezifisch Aussagen darüber treffen zu können, welche AADs nun konkret übereinstimmen. Zusätzlich sind oben beschriebene Tools wie AbstFinder nicht zur freien Verfügung standen, wodurch keine konkrete Umsetzung des oben beschriebenen Ansatzes zur Erstellung einer Ontologie möglich war. Wir wollen uns daher im Folgenden auf einen korpusbasierten Ansatz konzentrieren, der im nächsten Abschnitt diskutiert wird.

3.2 Korpusbasierter Ansatz

In Anbetracht der oben genannten Umstände musste eine alternative Herangehensweise gefunden werden, die größtenteils automatisiert ablaufen kann. Intuitiv sollen zwei AADs verlinkt werden, wenn sich ihre Beschreibung auf die gleiche Funktionalität bezieht und sie somit die gleiche Bedeutung haben, auch wenn sie möglicherweise sprachlich differenziert ausgedrückt wurden. Laut Natt och Dag et. al [3] ist keine funktionale Beschreibung von 'Bedeutung' möglich, ohne wie oben beschrieben manuell einzugreifen. Sie schlagen vor, das Problem auf eine terminologische Ebene zu verlagern und treffen die Annahme, dass sich zwei Anforderungsdokumente ähneln, wenn sie die gleiche Terminologie benutzen [3]. Um dennoch möglichen Inkonsistenzen Sorge zu tragen, werden mögliche Verknüpfungen zwischen AADs vorgeschlagen, welche dann aber noch vom Requirements Analyst manuell gesetzt werden müssen.

Um Kandidaten für eine mögliche Verknüpfung zu klassifizieren, wird daher eine Metrik berechnet, die kennzeichnet, bis zu welchem Grad sich die Terminologie zweier AADs ähnelt.

> Im Bezug auf obige Herleitung und in Anlehnung an die von Nenádic eingeführten Begriffe [12] soll diese Metrik im Folgenden *Terminological Similarity* heißen, oder kurz TS.

3.2.1 Information Retrieval

Um sich der Berechnung einer solchen Metrik zu nähern, werden Methoden aus dem Information Retrieval (IR) benutzt, die kurz dargestellt werden.

Im Information Retrieval geht es allgemein um die Beschaffung von Informationen aus in der Regel natürlichsprachlichen Dokumenten. Anwendung findet Information Retrieval vor allem in Online-Suchmaschinen.

Der grundlegendste Schritt ist es, zunächst eine sogenannte Inzidenz-Matrix aufzustellen, bei der für eine gewisse Eingabe (Query) und eine Sammlung verschiedener Dokumente für jeden Term in der Eingabe dargestellt wird, ob der Term in dem jeweiligen Dokument auftaucht oder nicht, was dementsprechend als 1 oder 0 kodiert wird.

Als Erweiterung lässt sich statt eines binären Wertes auch die Häufigkeit des Auftretens eines Terms im jeweiligen Dokument festhalten, die sogenannte *Term Frequency* oder kurz TF. Eine Matrix über sämtliche in der betrachteten Sammlung vorkommenden Terme und Dokumente beinhaltet meist sehr viele Nullen, da naturgemäß nur ein Bruchteil der Gesamtheit aller Terme überhaupt in dem betrachteten Dokument vorkommt. Das Speichern einer solchen Matrix und Anfragen daran sind daher hinreichend effizient durchführbar [3]. In dieser Matrix lässt sich nun für jedes Dokument ein

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
ANTHONY	1	1	0	0	0	1
BRUTUS	1	1	0	1	0	0
CAESAR	1	1	0	1	1	1
CALPURNIA	0	1	0	0	0	0
CLEOPATRA	1	0	0	0	0	0
MERCY	1	0	1	1	1	1
WORSER	1	0	1	1	1	0

ANTHONY	157	73	0	0	0	1
BRUTUS	4	157	0	2	0	0
CAESAR	232	227	0	2	1	0
CALPURNIA	0	10	0	0	0	0
CLEOPATRA	57	0	0	0	0	0
MERCY	2	0	3	8	5	8
WORSER	2	0	1	1	1	5

Abbildung 3.2: Darstellung der Dokumente als binäre Vektoren $\in \{0,1\}^v$ (oben) und als TF-Vektor $\in \mathbb{N}^v$ (unten) im Term-Dokument-Vektorraum, wobei v die Anzahl sämtlicher Terme darstellt. Hier wäre v = 7.

Vektor ermitteln, der die im Dokument auftauchenden Terme notiert. Dabei stellt jeder Term eine Dimension dar. Als Beispiel dazu und in Anlehnung an Abbildung 3.2 ein fiktives Dokument:

> '*Anthony mochte Cleopatra gern und fragte Brutus, ob Anthony eine Halskette von Cleopatras Mutter für Cleopatras Geburtstag bekommen könnte.*'

Dieser Beschreibungstext wird gefiltert und gestemmt, wie in Kapitel 3.2.2 beschrieben, was hier (zu Demonstrationszwecken) zu den Termen {Anthony, Brutus, Cleopatra} führt, was dem binären Vektor {1,1,0,0,1,0,0} und dem TF-Vektor {2,1,0,0,3,0,0} entspräche. Wegen der Dimension 7 wurde an dieser Stelle auf eine graphische Darstellung dieser Vektoren verzichtet, der Unterschied zwischen binären Vektoren und TF-Vektoren ist allerdings in Abbildung 3.2 zu sehen. Besondere Bedeutung in diesem Vektorraum-Modell (*Vector Space Model*, [10]) hat der Kosinus des Winkels zwischen zwei Vektoren. Er wird maximal oder 1, wenn die beiden Vektoren in die gleiche Richtung zeigen (parallel) und minimal oder -1, wenn sie in entgegengesetzte Richtungen zeigen. Dies eignet sich intuitiv gut dazu, sehr ähnlichen, bzw. sehr verschiedenen Dokumenten entsprechende Werte zuzuweisen, weshalb der Kosinus in Kapitel 3.2.4 zur Gewichtung benutzt wird.

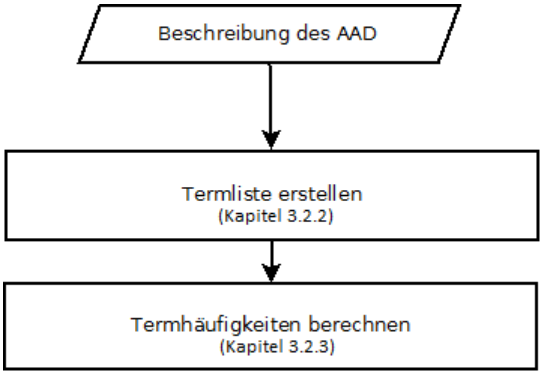

Abbildung 3.3: Der allgemeine Ablauf der vorbereitenden Maßnahmen für die Übereinstimmungs-Berechnung zweier Dokumente

Der grundlegende Ablauf der vorbereitenden Schritte für die Berechnung der TS zwischen zwei AADs ist in Abbildung 3.3 zu erkennen und wird im Folgenden näher erläutert.

3.2.2 Termliste

Zunächst muss der Algorithmus aus der jeweiligen Beschreibung des AADs eine sinnvolle Termliste generieren, in der nach Möglichkeit alle relevanten Konzepte enthalten und alle irrelevanten ausgeschlossen werden. Betrachtet man als Beispiel

> 'Der Administrator schaut sich die jeweiligen Termine an und weist ihnen neue Ressourcen zu.'

lassen sich intuitiv die Terme 'Administrator', 'Termin', 'Ressourcen', 'anschauen', 'zuweisen', 'neu' und 'jeweiligen' als terminologisch relevant erkennen.

In einer weiteren Beispiel-Beschreibung

> 'Ich kann einer Reservierung eine Farbe zuordnen. Mit dieser Farbe wird die Reservierung dann im Kalender angezeigt.'

lässt sich wiederum den Termen 'Reservierung', 'Farbe', 'zuordnen', 'Kalender' und 'anzeigen' intuitiv eine hohe terminologische Relevanz zuordnen.

Aus diesen Beispielen ist die Tendenz erkennbar, dass besonders Nomen, Verben und zum Teil Adjektive/Adverbien eine hohe terminologische Relevanz aufweisen, während Artikel, Konjunktionen und Pronomen schlichtweg

irrelevant sind, da sie keinen terminologischen Informationsgehalt tragen. An dieser Stelle sind zwei Vorgehensweisen denkbar:

1. Die eingehende Beschreibung mithilfe eines PoS-Taggers zu annotieren und dann die Nomen, Verben und Adjektive herausfiltern
2. Die Beschreibung mithilfe von Tokenization, Stop-Word-Removal und Stemming auf ihre elementaren Bestandteile reduzieren

PoS-Tagging: Zur Verwendung eines PoS-Taggers für die deutsche Sprache bietet sich das PoS-Tagging-Modul des OpenNLP-Toolkits[3] oder das entsprechende Modul des bekannten Stanford-Parsers[4] an. Beides sind statistisch-probabilistische *Machine-Learning*-Parser basierend auf *Maximum Entropy*, die zunächst mithilfe eines (meist mitgelieferten) Models trainiert werden müssen oder bereits vortrainiert sind. Diese lieferten im Test zwar beide akzeptable und vergleichbare Resultate, die aber im Rahmen des hier vorgestellten Verfahrens lediglich als Filterungsfunktion dienen würden, wobei die annotierten PoS-Tags nachträglich wieder entfernt werden müssten, um die Terme nicht zu verfälschen. An dieser Stelle sei noch angemerkt, dass sich PoS-Tagging in wissensbasierten Verfahren sinnvoller einsetzen lässt, um dort beispielsweise einen Tag-Baum zu erzeugen, der dann weiter ausgewertet wird. Das PoS-Tagging erweist sich in diesem Verfahren daher eher als überflüssig, weshalb hier die zweite Möglichkeit genutzt wird und wie in Kapitel 2 beschrieben, Tokenization und Stop-Word-Removal angewendet werden.

Stemming: Außerdem sollte die Termliste noch gemäß Kapitel 2 gestemmt werden, um die Terme besser zu abstrahieren und die im nächsten Absatz erklärten Term Frequencies nicht zu verfälschen. Auf die verschiedenen Methoden zum Stemmen der deutschen Sprache wird in Kapitel 4.1 näher eingegangen.

Wichtig: Die generierte Termliste ist redundant, sie kann also den gleichen Term mehrfach beinhalten, was dem mathematischen Konzept einer Multimenge entspricht. Diese Darstellung wird für den nächsten Schritt benötigt.

3.2.3 Term Frequencies

Als nächstes muss die Term Frequency für jeden Term einer AAD-Beschreibung bestimmt werden. Die generierte Termliste wird herangezogen und jeder

[3] `https://opennlp.apache.org/`
[4] `http://nlp.stanford.edu/software/tagger.shtml` [13]

Term in kleingeschriebener Notation zusammen mit der Anzahl seines Vorkommens paarweise in eine weitere Liste geschrieben. Der Algorithmus geht dabei folgendermaßen vor,

```
Data: Termliste
Result: Liste mit Tupeln aus (Term, Häufigkeit)

while nicht am Ende der Termliste do
    hole aktuellen Term;
    if aktueller Term bereits in der Ausgabe then
        hole die Häufigkeit des Terms;
        addiere 1 zu der Häufigkeit;
    else
        schreibe neues Tupel (Term, 1) in die Ausgabe;
    end
end
```

Algorithm 1: berechne Termhäufigkeiten

wobei sich im Folgenden für die Term Frequency eines Terms t in einem AAD d die Notation $TF_{t,d}$ findet.

3.2.4 Berechnen der Terminological Similarity

Für das terminologische Vergleichen zweier AADs sind nun weitere Schritte erforderlich, die in Abbildung 3.4 dargestellt sind. Es wird also ein Weg gesucht, einem Paar aus zwei AADs eine Bewertung darüber zuordnen zu können, wie stark sie terminologisch übereinstimmen. Dazu sollten für zwei AADs a und b folgende Kriterien erfüllt sein

- Terme, die entweder in a oder b, aber nicht in beiden vorkommen, tragen nicht zur TS bei und sollten ein Gewicht von 0 bekommen
- Je häufiger ein Term aus a in b vorkommt, desto höher sollte sein Gewicht sein

Dabei sieht das konkrete Vorgehen wie folgt aus:
Für zwei AADs a und b wird für jeden Term in einer gewissen Menge an Termen m ein Gewicht für a und b berechnet, welches über alle Terme aufaddiert wird.

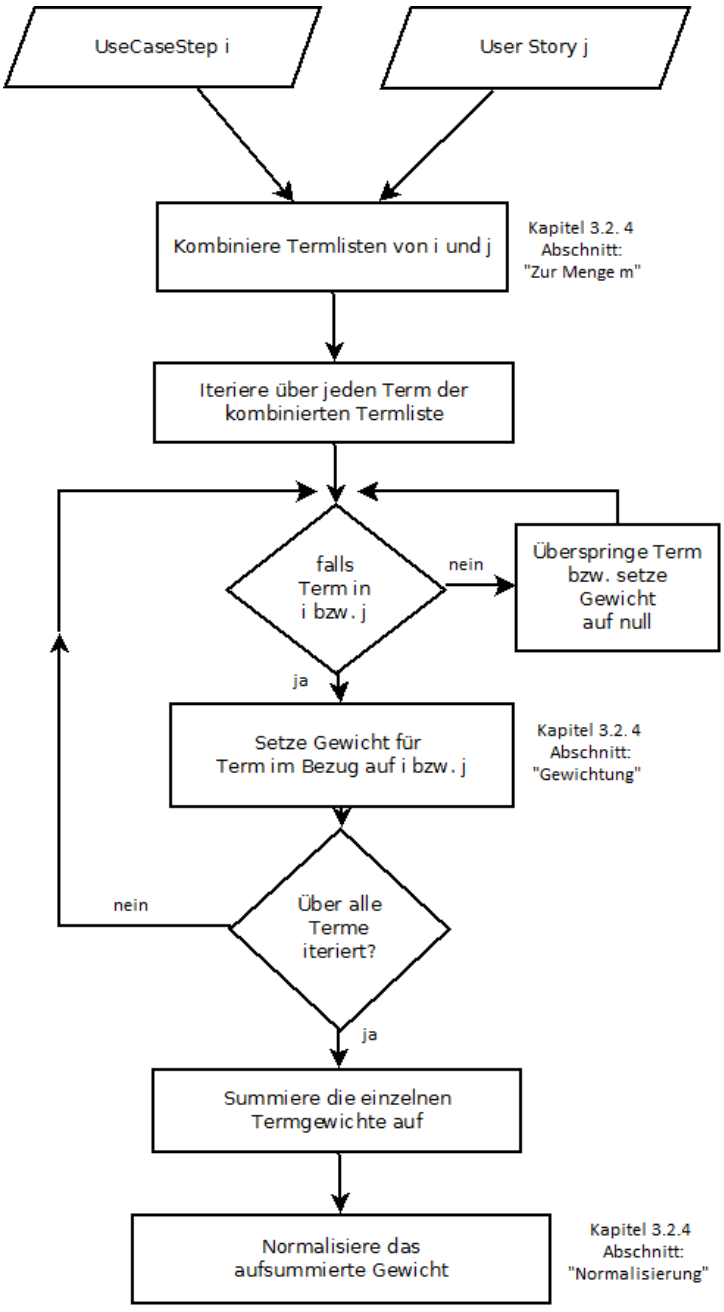

Abbildung 3.4: Allgemeines Vorgehen zur Berechnung der TS zwischen einem UseCaseStep und einer UserStory als Flussdiagramm

Zur Menge m Es wäre denkbar, dass m die Menge aller Terme im gesamten Projekt darstellt, wie im Abschnitt 3.2 beschrieben. Da eine im Projekt gespeicherte und gepflegte Matrix/Tabelle offensichtlich viele Nullen beinhaltet, die sich dementsprechend auch in den jeweiligen Term Frequency-Listen widerspiegeln und eine logarithmische Gewichtung der TF am sinnvollsten ist, wie im Folgenden erläutert wird, beinhaltet m lediglich die Terme, die in den verglichenen Dokumenten enthalten sind, um den Aufwand zu reduzieren: Es lässt sich vorhersagen, dass Terme, die zwar im Projekt, aber nicht im konkret verglichenen AAD-Paar auftauchen, stets eine Term Frequency von null aufweisen, sodass sie nie einen Beitrag zur TS liefern. Nur die im verglichenen AAD-Paar auftretenden Terme, sozusagen die lokale Terminologie, liefern also potentiell einen Beitrag zur TS. Auch hier kann es natürlich Terme in AAD a geben, die nicht in AAD b stehen und umgekehrt. Um zu vermeiden, dass in diesen Fällen (wenn die TF null ist) der Logarithmus an der Stelle null gebildet wird, wo er mathematisch nicht definiert ist, wird in diesen Fällen das Gewicht und damit der Beitrag des Terms zur TS auf null gesetzt. Um nun also lediglich die lokale Terminologie zu betrachten, werden die beiden Termlisten zu einer gemeinsamen kombiniert, in der jeder Term beider Listen genau einmal auftaucht.

Ein Beispiel dazu:

AAD a = { *kategorie, liste, ressource, user, dialog* }
kombiniert mit
AAD b = { *kategorie, kalender, termin, ressource, auslesen* }
wird dementsprechend zu
{ *kategorie, liste, ressource, user, dialog, kalender, termin, auslesen* }.

oder mathematisch für Liste X und Y: $(X, Y) \mapsto (X \cap Y) \cup (X \cup Y)$. Die Term Frequency von *'liste'* in AAD b ist null, weshalb sein Gewicht entsprechend auf null gesetzt wird.

Das allgemeine Vorgehen zur Berechnung der TS ist in Abbildung 3.4 dargestellt, wobei im Folgenden genauer auf die aus der Literatur bekannten Methoden zur Gewichtung und Normalisierung eingegangen werden soll.

Gewichtung Übliche Gewichtungsschemata sind in Abbildung 3.5 zu finden. Aufgrund der geforderten Kriterien klingt es vor allem logisch, die TF für diese Betrachtung heranzuziehen. Angenommen, der Term a taucht viermal und Term b einmal in Dokument x auf, d.h. $TF_{a,x} = 4$ und $TF_{b,x} = 1$, dann sollte a eine höhere terminologische Bedeutung aufweisen als b, aber auch keine viermal höhere [3]. Eine lineare oder proportionale Gewichtung

Term occurrence		Document frequency		Normalization
n (natural)	$\mathrm{tf}_{t,d}$	n (natural)	df_t	n (no normalization)
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)
a (augmented)	$0.5 + \frac{0.5 \times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$			$\frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_n^2}}$

Abbildung 3.5: Typische Schemata für die Gewichtung der einzelnen Komponenten. Quelle: [10]

der TF ist also weniger wünschenswert, weshalb hier auf die logarithmische Gewichtung gemäß

$$w_d(t) = \begin{cases} 1 + log_b(TF_{t,d}), & TF_{t,d} > 0 \\ 0, & sonst \end{cases} \tag{3.1}$$

zurückgegriffen wird. Die Notation $w_d(t)$ bezeichnet dabei das Gewicht des Terms t in AAD d, wobei d ein UseCaseStep oder eine UserStory sein kann.
Wie der Abbildung 3.6 zu entnehmen ist, hat die logarithmische Gewichtung der TF gegenüber der linearen Gewichtung für Toplisten mit einer Länge kleiner 100 auch einen positiven Effekt auf den Recall. Dazu mehr in Kapitel 5. In der Literatur finden sich außerdem verschiedene Angaben zur Basis b der in diesem Abschnitt genannten Gewichtungen: [3] benutzt den Logarithmus Dualis mit Basis 2, während [9] den dekadischen Logarithmus mit Basis 10 benutzt. Wir verwenden im Folgenden die Basis 2.

Die zweite Komponente stellt die sogenannte *Document Frequency*(DF) dar, die beschreibt, in wievielen Dokumenten ein Term t auftaucht. Die logarithmische Gewichtung der DF

$$w_d(t) = log_b(\frac{N}{DF_t}) \tag{3.2}$$

bezieht sich auf die Gesamtheit N aller Dokumente und liefert einen geringeren Wert, je mehr Dokumente es gibt, in denen der jeweilige Term auftaucht. Diese Gewichtung basiert auf einer Annahme aus der Informationstheorie, wonach häufig auftretende Terme/Worte einen geringeren Informationsgehalt tragen als selten auftretende und wird daher auch als *Inverse Document Frequency* (IDF) bezeichnet [9].

Bezug zum Konzept Dies erweist sich vor allem dann als vorteilhaft, wenn sich im Dokument noch *Stop Words* befinden. Da zu diesem Zeitpunkt allerdings die Termliste betrachtet wird, in der bereits alle *Stop Words* herausgefiltert wurden, ist die Annahme nicht mehr korrekt und eine logarith-

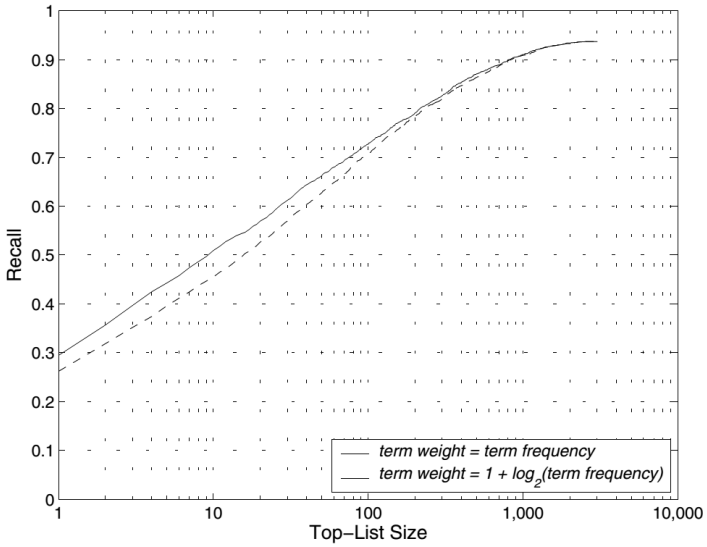

Abbildung 3.6: Die Auswirkung der TF-Gewichtung auf den Recall im Verhältnis zur Länge der Topliste. Quelle: [3]

mische Gewichtung der DF würde dazu führen, dass gerade den Konzepten/Termen, die in vielen AADs auftauchen und daher als terminologisch relevant gelten, eine geringere Bedeutung zugewiesen wird als den selten auftretenden und damit nicht so relevanten Konzepten. Die natürliche Gewichtung der DF hingegen wäre möglicherweise eine sinnvolle Komponente zur Ergänzung, hat es allerdings aufgrund zeitlicher und programmatischer Probleme nicht in die Implementierung geschafft.

Normalisierung Abschließend sollte das Gewicht noch normalisiert werden. Wir betrachten Termlisten zunächst als binäre Vektoren, die nur aus den Wörtern bestehen, welche auch in der Beschreibung des zugehörigen AADs auftauchen. Die in der einschlägigen Literatur üblichen Methoden zur Normalisierung finden sich exemplarisch in Abbildung 3.7. All diese Verfahren basieren auf der Schnittmenge zwischen den Termlisten, d.h. ein Term liefert immer nur dann einen Beitrag zur TS, wenn er in beiden Termlisten mindestens einmal auftaucht. Sie werden im Folgenden kurz dargestellt.

- **Matching-Koeffizient**: Die simpelste Methode entspricht genau der Schnittmenge zwischen zwei Termlisten und beachtet nicht die Länge des jeweiligen Vektors. Sie zählt lediglich, wieviele Dimensionen in beiden Vektoren ungleich null sind.
- Der **Dice-Koeffizient** normalisiert die Länge, indem durch die gesamte Anzahl von Dimensionen ungleich 0 geteilt wird, also der Summe der Beträge der Vektoren

Similarity measure	Definition
matching coefficient	$X \cap Y$
Dice coefficient	$\frac{2\|X \cap Y\|}{\|X\| + \|Y\|}$
Jaccard (or Tanimoto) coefficient	$\frac{\|X \cap Y\|}{\|X \cup Y\|}$
Overlap coefficient	$\frac{\|X \cap Y\|}{\min(\|X\|, \|Y\|)}$
cosine	$\frac{\|X \cap Y\|}{\sqrt{\|X\| \times \|Y\|}}$

Abbildung 3.7: Verfahren zur Normalisierung von Vektorgrößen. Quelle: [10]

- Der **Jaccard-Koeffizient** wird zwar im IR häufig benutzt, berücksichtigt aber leider weder die TF noch die DF
- Der **Overlap-Koeffizient** wird maximal, wenn die Vektoren sich gegenseitig inkludieren, also entweder $X \subseteq Y$ oder $Y \subseteq X$.
- Der **Kosinus** hingegen ist im Gegensatz zum Dice-Koeffizienten nicht so sehr vom Unterschied des Betrages der Vektoren abhängig.

Bezug zum Konzept Wir wollen ein Verfahren finden, das die Länge der UseCaseSteps und UserStories einbezieht, aber große Unterschiede zwischen der Länge beim Vergleich nicht zu stark in die TS einfließen lässt. Zu diesem Zweck lässt sich der Matching-Koeffizient direkt ausschließen, da er die Länge von Vektoren überhaupt nicht berücksichtigt. Betrachten wir das folgende Beispiel:

$|x| = 2$ und $|y| = 7$ mit $x = \{kategorie, ressource\}$ und
$y = \{dialog, ressource, kalender, user, eintragen, datum, kategorie\}$
und den Normalisierungen
Dice: $2 * 2/9 = 4/9 = 0.44$, Jaccard: $2/5 = 0.4$,
Overlap: $2/2 = 1$, Kosinus: $2/\sqrt{2 * 7} \approx 0.53$

Die beiden Termlisten x und y unterscheiden sich in ihrer Länge, wobei x komplett in der längeren enthalten ist. Zunächst einmal lässt sich speziell für diesen Fall erkennen, dass Overlap eine exakte Übereinstimmung von 1 ergibt, was dadurch begründet ist, dass x komplett in y enthalten ist. Trotz dieser Inklusion wird die Tatsache, dass fünf Terme aus y nicht in x vorkommen, nicht in die Berechnung einbezogen. Wir streben eher die Bewertung

von 100% Übereinstimmung an, wenn die Termlisten komplett identisch sind, weshalb auch Overlap ausscheidet.

Dice und Jaccard liefern ähnliche Werte, wobei das geringe Verhältnis von gemeinsamen Terme zu allen Termen von Jaccard stärker 'bestraft' wird [10]. Da sich zwei AAD-Beschreibungen ähneln sollen, wenn sie die gleiche Terminologie benutzen, ist es besser, UserStory x geringer zu bewerten als UserStory z (beide im Vergleich zu UseCaseStep y), wenn mehr als $|x|$ Terme von z in y enthalten sind (wobei $|z| > |x| = 2$).
Aus diesem Grund ist Jaccard minimal besser geeignet als Dice, wobei sich der Unterschied sinnvoller zeigen würde, wenn $|x| = |y|$ gilt.

Der Kosinus liefert einen Wert zwischen Jaccard/Dice und Overlap, liegt aber deutlich näher bei ersteren. Er 'bestraft' den Längenunterschied zwischen x und y aufgrund der Wurzel nicht so hart wie Dice, wodurch sich der Unterschied erklären lässt, im Fall $|x| = |y|$ würden die beiden aber übereinstimmen [10].

Im Kontext dieses Konzeptes ist es nicht unwichtig, bei zwei AADs, die sich durch ihren Inhalt und damit durch die Anzahl ihrer relevanten Terme (die Länge der Termliste) stark unterscheiden, dafür zu sorgen, ihre terminologische Übereinstimmung deshalb nicht zu sehr abzuwerten. Ein UseCaseStep x mit deutlich längerer Beschreibung als eine UserStory y ist nicht allein aufgrund dieses Längenunterschiedes weniger oder stärker relevant.
Aufgrund dieser Längenunterschied-Irrelevanz-Eigenschaft und da wir nun die Termliste (oder treffender: die Liste mit den Term Frequencies) eines AADs durch ganzzahlige Vektoren repräsentieren wollen, in denen jede Dimension die Häufigkeit des jeweiligen Terms darstellt (siehe Kapitel 3.2), eignet sich das Kosinus-Verfahren am besten für die Normalisierung, da es das einzige Verfahren ist, welches nicht nur mit binären Vektoren im Vektorraum der Terme arbeitet, sondern aufgrund seiner Bedeutung im Vektorraum (siehe Kapitel 3.2) auch quantitative Informationen wie die TF oder Wahrscheinlichkeiten mit einbeziehen kann [10]. Das Kosinus-Verfahren benutzt die euklidische Länge der Vektoren und setzt daher einen euklidischen Vektorraum voraus, d.h. die Länge aller Vektoren muss auf 1 normiert sein. Da die TermFrequencies von UserStories und UseCases wie die meisten Repräsentationen zur Berechnung von semantischen Übereinstimmungen allerdings nicht normiert sind, bilden sie genau genommen gar keinen euklidischen Vektorraum. Trotzdem ist das Kosinus-Verfahren im Gegensatz zu (hier nicht vorgestellten) probabilistischen Methoden aber konzeptuell simpler und einfacher zu berechnen, weshalb es im Information Retrieval und auch in diesem Konzept verwendet wird.

Formel zur Berechnung Setzt man nun also Gleichung (3.1) und die Kosinus-Normalisierung als Komponenten zusammen, erhält man in der Notation von Abbildung 3.5 das Schema 'lc' (ohne Document Frequency) und in Anlehnung an [3] als Formel zur Berechnung der *Terminological Similarity* zwischen einem UseCaseStep ucs und einer UserStory us den Term

$$TS(ucs, us) = \frac{\Sigma_t w_{ucs}(t) * w_{us}(t)}{\sqrt{\Sigma_t (w_{ucs}(t))^2 * (w_{us}(t))^2}} \tag{3.3}$$

wobei $w_{ucs}(t)$ und $w_{us}(t)$ die Gewichte des Terms t in der jeweiligen Termliste des UseCaseSteps bzw. der UserStory sind.

3.3 Nutzung vorhandener Strukturen im Projekt

In diesem Abschnitt soll untersucht werden, inwiefern die im Projekt vorhandenen Strukturen einen Einfluss auf die Übereinstimmung zwischen einem UseCaseStep und einer UserStory haben. Es soll festgestellt werden, ob gewisse Gegebenheiten und Strukturen eine prozentuale Erhöhung des errechneten Wertes suggerieren, die dann beispielsweise in Form eines Faktors umgesetzt wird.

Zeitachse in der Story Map Wie in Abschnitt 2.2 erwähnt, hat eine User Story Map eine zeitliche Komponente, durch welche die User Tasks chronologisch eingeordnet werden können. Solch eine zeitliche Komponente fehlt aber bei Use Cases, sodass kaum ein zeitlicher Zusammenhang zwischen Use Cases und User Tasks hergestellt werden kann. Als Beispiel betrachten wir den UseCaseStep x, die UserStory a in User Task 2 und die UserStory b in User Task 5. X wird in Schritt 1 mit a und in Schritt 2 mit b verglichen. Offensichtlich ist b chronologisch nach a einzuordnen, aber da die Repräsentation von x als UseCaseStep keine Aussage über die zeitliche Einordnung zulässt, kann man die Übereinstimmung in Schritt 1 nicht höher oder niedriger bewerten als in Schritt 2.

Die zeitliche Position des zu einer verglichenen UserStory gehörenden User Tasks innerhalb der Story Map spielt also keine erkennbare Rolle für die TS.

Interne Strukturen Eine weitere sinnvolle Betrachtung ist der potentielle Zusammenhang zwischen der Position eines UseCaseSteps im Use Case und der Position einer UserStory im User Task. Wie in Kapitel 2.3 beschrieben,

stellt ersteres meist eine grobe Reihenfolge der Durchführung eines Use Cases dar, während letzteres eher als Priorität innerhalb des User Tasks betrachtet werden sollte. Die UserStories innerhalb eines User Tasks stellen vor allem Details, Variationen und Alternativen des User Tasks dar, keine Reihenfolge. Die Prioritäten der einzelnen UseCaseSteps unterscheiden sich hingegen nicht voneinander, weshalb auch hier gilt:

> **Die Position eines UseCaseSteps im Use Case korreliert nicht mit der Priorität einer UserStory im User Task, weshalb hier ebenfalls keine Anpassung der TS erforderlich ist.**

Vorhandene Verknüpfungen Interessant ist jedoch die Einbeziehung von bereits vorhandenen Verknüpfungen im Projekt, da diese stets manuell getroffen werden und somit eine hohe Relevanz für den semantischen Zusammenhang darstellen. Angenommen, der UseCaseStep x ist mit den UserStories a und b verknüpft und es soll die TS zwischen UseCaseStep y zu anderen UserStories im Projekt bestimmt werden, wobei als Randbedingung gilt, dass sich y im gleichen Use Case befindet wie x, die beiden also 'Nachbarn' sind. Betrachtet man nun alle UserStories im Projekt, ist es sinnvoll, denjenigen UserStories ein höheres Gewicht zu geben, die bereits mit den jeweiligen Nachbarn verknüpft sind. Im Beispiel bedeutet dies, dass a und b ein höheres Gewicht bekommen, da sie bereits mit x, dem Nachbarn von y, verknüpft sind. Folglich wird die Gruppierung nach Kohärenz der AADs im jeweiligen Anforderungsdokument (die UseCaseSteps des gleichen Use Cases beispielsweise hängen in gewisser Weise zusammen und beziehen sich auf den gleichen übergeordneten Kontext) ausgenutzt und es gilt:

> **Bereits vorhandene Verknüpfungen zwischen AADs im Projekt führen zu einer erhöhten TS.**

Um dieser strukturellen Eigenschaft, die verknüpfte Elemente in einen Kontext zu einander bringt, Beachtung zu schenken, wird daher ein zusätzlicher Faktor für die gesamte TS des jeweiligen AADs eingeführt, der im Folgenden in Anlehnung an den Begriff *Contextual Similarity* aus [12] als contexSim bezeichnet wird.

Domain Model Abschließend ist auch die Einbindung eines Domain-Models für das jeweilige Projekt denkbar. Darin finden sich für die Domäne besonders relevante Konzepte, welche einen höheren Beitrag zur TS liefern können als normale Terme. Ein typisches Domain-Model ist in Abbildung 3.8 zu sehen, wird in UML als Klassendiagramm dargestellt und beinhaltet neben

den Klassen bzw. Konzepten auch Relationen. Diese Relationen zwischen den Konzepten könnten zwar in einem wissensbasierten Verfahren ausgewertet (Relation Mining) und z.B. in einer Ontologie dargestellt werden (siehe Kapitel 3.1), können aber in diesem (korpusbasierten) Prozess bei der Berechnung der TS nicht einbezogen werden. Hier werden nur die Klassennamen als Terme in einer Termliste betrachtet und mit der Termliste der einzelnen AADs verglichen, wobei sichergestellt wird, dass der jeweilige Term bei Übereinstimmung mit einem Term aus dem Domain-Model herausragend priorisiert wird, nicht jedoch alle Terme des AADs. Für die Verwendung eines Domain-Models wird demnach ebenfalls ein zusätzlicher Faktor eingeführt, der mit dem Gewicht des entsprechenden Terms multiplikativ verrechnet wird und im Folgenden DMF heißt.

Das Vergleichen mit Termen aus einem vorgegebenen Domain Model führt bei übereinstimmenden Termen zu einer höheren Priorität.

Alternativen Anstatt die Einbeziehung von Domain-Model und benachbarten AADs in Form eines Faktors darzustellen, der multiplikativ mit der TS verrechnet wird, könnte man auch zwei neue Übereinstimmungs-Begriffe einführen, z.B. die Kontextuelle Übereinstimmung (CS) und die Domänen-Übereinstimmung (DS). Das Gewicht eines einzelnen Terms würde dann nicht nur aus der TS mit Gewichtung 1 und zusätzlichem Domain-Model-Faktor bestehen, sondern beispielsweise aus einer Linearkombination von TS und DS, die beide unterschiedlich gewichtet werden, wobei die Gewichte α und β summiert wieder 1 ergeben. Im folgenden Beispiel ist diese Situation dargestellt,

$\alpha = 0.8$, $\beta = 0.2$, $\gamma = 0.7$, $\delta = 0.3$
Besser $S_{interim}$ = 0.8 * TS + 0.2 * DS und
S_{final} = 0.7 * $S_{interim}$ + 0.3 * CS
statt $S_{interim}$ = 1 * TS * DMF und S_{final} = 1 * $S_{interim}$

wobei CS und DS noch logarithmisch gewichtet sein könnten und $S_{interim}$ einen Zwischenwert für die Übereinstimmung eines einzelnen Terms t bildet, wobei die TS wie im Kapitel 3 beschrieben aus $w_{ucs}(t)$ und $w_{us}(t)$ besteht. Die DS könnte binär berechnet werden (1, falls t im Domain-Model, 0 falls nicht), aber auch die $TF_{t,DomainModel}$ wäre bei hinreichend komplexem/umfangreichem Domain-Model denkbar. Die Werte für α und β könnten auch nach Belieben angepasst werden. Dieses Vorgehen hätte den Vorteil,

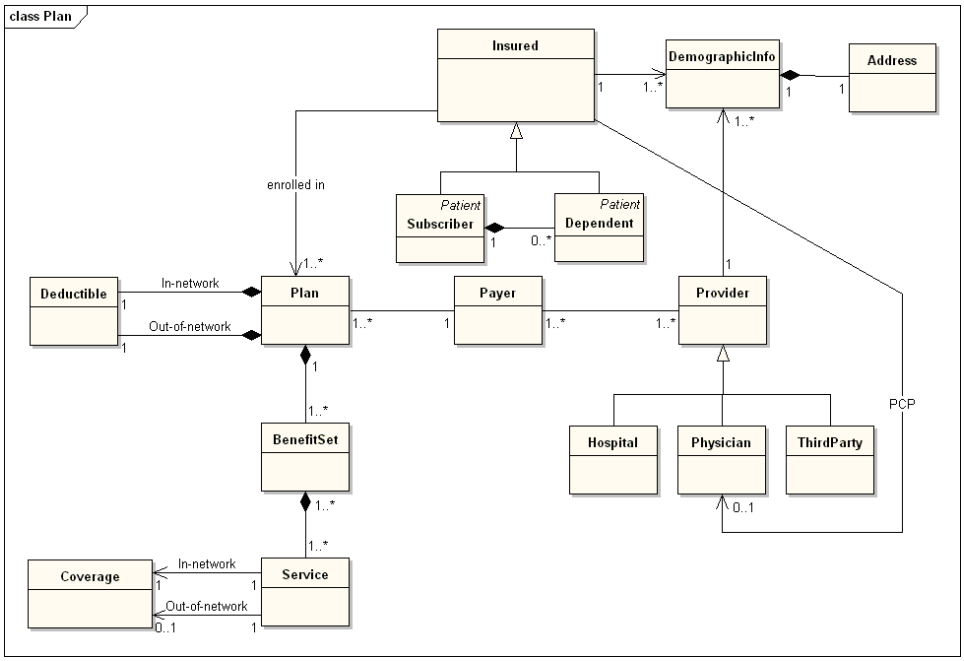

Abbildung 3.8: Ein typisches Domain-Model. Die Klassennamen werden als Terme betrachtet, die Relationen und Klassendetails ignoriert. [Quelle: Wikipedia]

dass die DS unabhängig von der TS in die Gesamt-Übereinstimmung einfließen kann, auch wenn die TS null beträgt.
Wenn über alle Terme iteriert wurde, könnte dann analog dazu die gesamte Übereinstimmung S_{final} als Linearkombination von $S_{interim}$ und CS dargestellt werden (entsprechend dem Beispiel wieder mit zwei weiteren Faktoren γ und δ, die unabhängig von α und β sein können), statt lediglich den Faktor contexSim dran zu multiplizieren. Die Untersuchung, ob eine solche Änderung der Übereinstimmungs-Berechnungen zu repräsentativeren Werten führt, könnte Gegenstand weiterer Arbeiten sein.

3.4 Anwendung auf UseCaseSteps und UserStories

Im Folgenden sollen nun die oben vorgestellten Grundlagen aus dem Information Retrieval konkret auf die atomaren Anforderungsdokumente angewendet werden. Wie bereits erwähnt, werden im Rahmen dieses Konzeptes UseCaseSteps und UserStories miteinander verglichen. Der Algorithmus in Pseudocode zur Berechnung der TS zwischen einem UseCaseStep und einer UserStory sieht dabei so aus wie in Algorithmus 3 beschrieben.

Für das Erzeugen einer Liste mit den als am relevantesten bewerteten UseCaseSteps zu einer UserStory und umgekehrt wird dann das Vorgehen in Algorithmus 2 benutzt.

Data: UseCaseStep: x
Result: Liste mit relevantesten UserStories zu x: similarStories
erzeuge Liste mit allen zu vergleichenden UserStories im Projekt (storiesToCompare);

while *es gibt noch unverglichene UserStories in storiesToCompare* **do**
    Data: aktuelle UserStory y
    rufe calculateSimilarity(x, y) auf;
    schreibe das Ergebnis als Tupel (y, TS zwischen x und y) in similarStories;
end

sortiere similarStories absteigend nach TS und blende dann die TS aus;

Algorithm 2: erzeuge Liste mit relevantesten UserStories zu einem UseCaseStep

Data: UseCaseStep i
Data: UserStory j
Result: Terminological Similarity zwischen i und j

hole die Termlisten von i und j;
berechne die TermFrequencies aus i und j;
kombiniere die Termlisten zu einer gemeinsamen Termliste t;
Data: DomainModelFactor DMF
Data: einzelnes Termgewicht $S_{interim}$
Data: kombiniertes Termgewicht S_{final}
setze $S_{interim} = 0$;
setze $S_{final} = 0$;

while *nicht am Ende von t* **do**
  Data: Gewicht des aktuellen Terms in i w_1
  Data: Gewicht des aktuellen Terms in j w_2
  Data: aktueller Term temp
  if *temp in Termliste von i enthalten* **then**
    berechne TermFrequency von temp in i;
    setze w_1 auf $1 + log_2(TF_{t,i})$
  else
    setze w_1 auf 0;
  end
  if *temp in Termliste von j enthalten* **then**
    berechne TermFrequency von temp in j;
    setze w_2 auf $1 + log_2(TF_{t,j})$
  else
    setze w_2 auf 0;
  end
  setze entsprechenden Wert für DMF;
  setze $S_{interim} = w_1 * w_2 * DMF$;
  setze $S_{final} = S_{final} + S_{interim}$;
end

führe Normierung von x mit Kosinus-Koeffizient durch;
gebe die TS zwischen i und j zurück;

Algorithm 3: calculateSimilarity(UseCaseStep i, UserStory j)

Diese Algorithmen liefern zunächst nur einen groben Überblick, wurden aber im Rahmen eines Prototypen implementiert und werden in Kapitel 4 näher beschrieben.

3.5 Ausgabeformat

Die Wahl eines geeigneten Formats zur Ausgabe der Ergebnisse kann zur Verbesserung des Konzeptes beitragen. In diesem letzten Teil des Konzeptes werden daher die Möglichkeiten dazu diskutiert.
Für die Ausgabe der zu einem AAD relevanten AADs des anderen Typs bieten sich grundsätzlich zwei Möglichkeiten an: Zum einen könnte man einen Schwellwert festlegen, wobei nur AADs ausgegeben werden, deren TS über diesem Schwellwert liegt, und zum anderen eine sogenannte Topliste erzeugen, indem immer genau gleich viele potentiell relevante AADs ausgegeben werden.
Um einen sinnvollen Schwellwert festzulegen, könnte man überlegen, ab welcher errechneten TS der Requirements Engineer im Durchschnitt tatsächlich den Link setzt. Dieses Vorgehen würde allerdings eine empirische Betrachtung voraussetzen, um ein repräsentatives Ergebnis zu bekommen und es könnte dennoch sein, dass es False Negatives gibt, die Schwelle also zu hoch gesetzt wurde.
Des weiteren entsteht durch die dynamische Länge der Ausgabeliste bei entsprechend niedriger Wahl der Schwelle die Möglichkeit, dass zu viele Ergebnisse ausgegeben werden und der Nutzer den Überblick und damit Zeit verliert.
Um ein wenig mehr Determinismus ins Spiel zu bringen, betrachten wir im Vergleich dazu die fixe Ausgabelänge der Topliste. Hier bietet sich die Möglichkeit, die initial nicht auftauchenden, aber als korrekt identifizierten Ergebnisse (FNs) nachrücken zu lassen, sollten anhand des initialen Ergebnisses hinreichend viele Links gesetzt worden sein und die entsprechenden FNs nicht zu weit außerhalb der Topliste liegen. Ein Beispiel dazu findet sich in Abbildung 3.9: Initial werden die zu einem fiktiven UseCaseStep relevantesten UserStories 1-7 in einer Topliste ausgegeben. Ebenfalls korrekt, jedoch nicht in der Topliste enthalten, sind die Stories 9 und 11. Wenn der Requirements Engineer nun beschließt, Story 1 und 2 zu verlinken, verschwinden diese aus der Topliste und machen Platz für weitere Stories. Dadurch rutscht Story 9 im nächsten Schritt tatsächlich in die Topliste hinein, während Story 11 leider auch nach Verlinkung und Entfernung von Story 9 nicht aufrücken kann, auch weil ihre TS einen zu geringen Wert aufweist.

Einen Schwellwert an anderer Stelle zu benutzen, erfüllt nämlich einen gewissen Nutzen: Mit Verwendung einer Topliste werden immer gleich viele Ergebnisse ausgegeben, außer der äußerst unwahrscheinliche Fall tritt ein, dass ein solch großer Anteil aller beispielsweise UserStories verlinkt wurde, dass nicht mehr genug übrig bleiben, um die Topliste zu füllen. Die Topliste wird also stets mit verbliebenen Stories aufgefüllt, ungeachtet ihrer TS, was im schlimmsten Fall dazu führt, dass nur Ergebnisse mit einer TS von 0 in der Topliste auftauchen, einfach weil es keine anderen Stories mehr gibt. Deshalb ist es interessant, einen Schwellwert als unterste Grenze zu bestim-

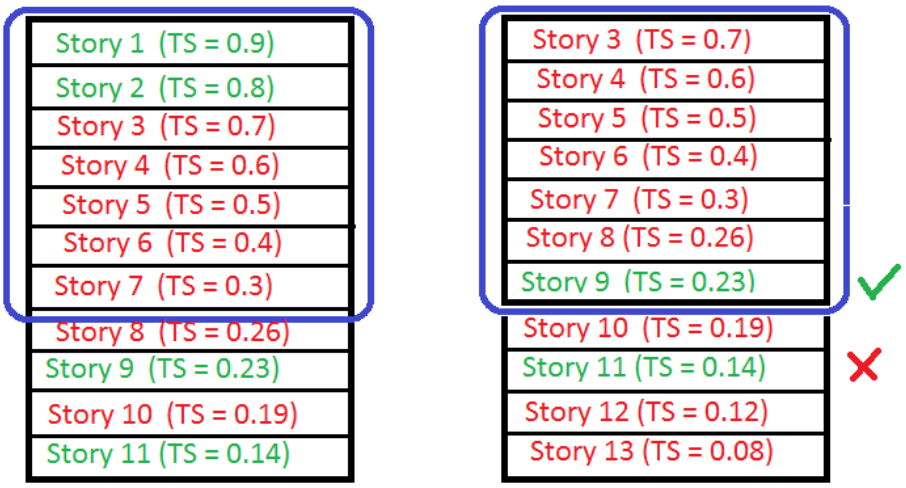

Abbildung 3.9: Die Nachrück-Problematik: Als korrekt identifizierte UserStories (TP+FN) zu einem fiktiven UseCaseStep sind in grün, alle inkorrekten UserStories (FP+TN) in rot dargestellt. Die blaue Umrandung kennzeichnet die Topliste.

men, wonach für ein Ergebnis zunächst überprüft wird, ob die errechnete TS hoch genug ist, um überhaupt betrachtet zu werden und erst danach, ob sie sich zur Aufnahme in die Topliste eignet. Das führt zwar gegebenenfalls zu weniger Ergebnissen als in der Topliste dargestellt werden sollen, schließt aber sowohl die extrem unwahrscheinlichen Ergebnisse und auch die definitiv komplett irrelevanten mit einer TS von 0 kategorisch aus. Das Risiko, durch Setzen einer Minimalschwelle False Negatives zu erzeugen, ist außerdem insgesamt deutlich geringer als bei einer wie oben beschriebenen Durchschnittswert-Schwelle. Intuitiv soll an dieser Stelle ohne empirische Betrachtung vermerkt werden, dass die Wahrscheinlichkeit, dass der Nutzer ein Ergebnis verlinkt, dessen errechnete TS unterhalb von 15% liegt, vernachlässigbar gering sein sollte, weshalb sich ein Wert von 0.15 für diese Schwelle anbietet.

Kapitel 4

Implementierung

Im folgenden Kapitel soll die Implementierung des Konzeptes aus Kapitel 3 besprochen werden, wobei eher die technische und programmatische Seite betrachtet wird. Die Implementierung erfolgte durch Einbettung in ein vorhandenes Tool zum Erheben von Anforderungen.

4.1 Prototyp: SimReqFinder

Zur Demonstration des Konzeptes ist ein Prototyp entstanden, der in Java geschrieben und in das vom Fachgebiet Software Engineering bereitgestellte Integrated Requirements Environment (IRE) eingebettet und SimReqFinder getauft wurde. Das IRE wurde mithilfe des Play!-Frameworks entworfen und stellt UseCases und zugehörige UseCaseSteps einer UserStoryMap aus UserTasks und zugehörigen UserStories gegenüber. Außerdem lassen sich die UseCaseSteps und UserStories bereits miteinander verknüpfen, weshalb es sich gut für die Demonstration dieses Konzeptes eignet.

An der Implementierung sind konkret 3 Klassen beteiligt, die in Abbildung 4.1 als Klassendiagramm dargestellt sind und deren Zusammenspiel im Folgenden genauer beschrieben wird.

Um zu einem UseCaseStep bzw. einer UserStory ähnliche AADs des jeweils anderen Typs anzuzeigen, wird die Methode *getLinkedStoryRecommendations()* bzw. *getLinkedStepRecommendations()* in der entsprechenden Model-Klasse aufgerufen. In der View geschieht dies durch Klicken auf den kleinen blauen Pfeil neben der Beschreibung eines UseCaseSteps bzw. durch Klicken auf den Titel einer UserStory.
Betrachten wir als Beispiel die Methode *getLinkedStoryRecommendations()*, um ähnliche UserStories zu einem UseCaseStep zu finden: Zunächst werden sämtliche AADs des jeweils anderen Typs im gesamten Projekt in einer Liste gesammelt, wobei die bereits mit dem betrachteten UseCaseStep verknüpften UserStories rausgefiltert werden. Anschließend wird die genaue Berechnung an die Hilfsklasse SimReqFinder weitergegeben und dort die Methode *get-*

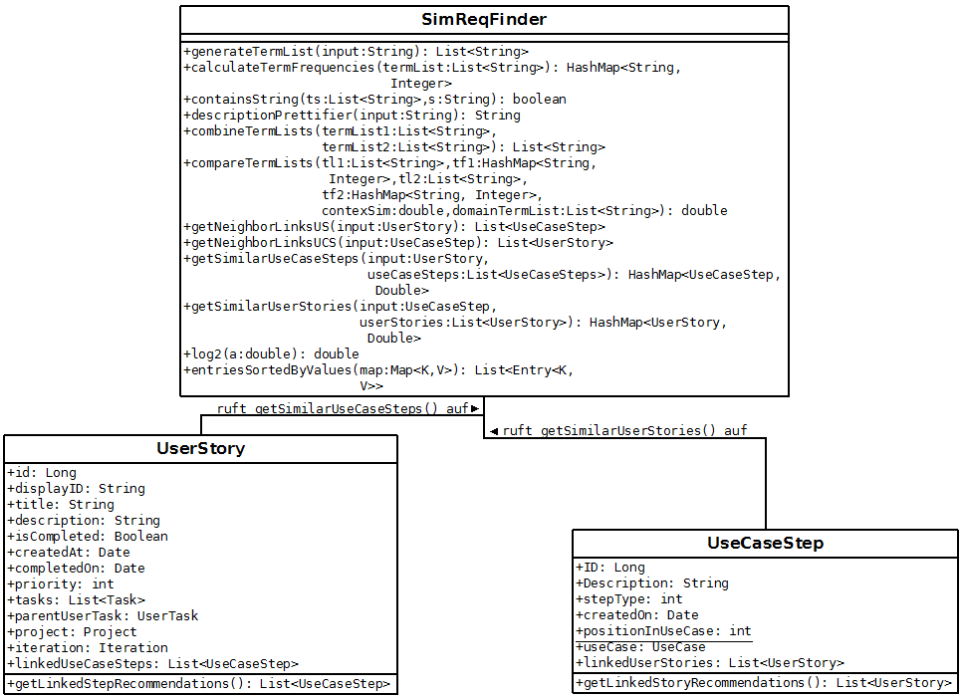

Abbildung 4.1: Die Model-Klassen *UserStory* und *UseCaseStep* rufen in *getLinkedStepRecommendations()* bzw. *getLinkedStoryRecommendations()* die Methode *getSimilarUseCaseSteps()* bzw. *getSimilarUserStories()* in der Hilfsklasse *SimReqFinder* auf.

SimilarUserStories() mit dem betrachteten UseCaseStep und der Liste aller zu vergleichenden UserStories aufgerufen. Das Ergebnis wird dabei in eine HashMap geschrieben, wobei wie in Algorithmus 2 in Kapitel 3.4 angedeutet, Key-Value-Paare (Tupel) aus UserStory und zugehöriger TS erzeugt werden. Das Resultat wird dann absteigend nach der TS sortiert, sodass die relevantesten Ergebnisse an erster Stelle in der HashMap stehen. Diese Sortierung machen wir uns zunutze, indem danach nur die ersten n Elemente dieser HashMap ausgegeben werden, wobei n (im Code topListSize) die Länge der Topliste beschreibt. Außerdem werden nur diejenigen Stories ausgegeben, deren TS oberhalb der Minimalschwelle minThreshold liegt, sodass ggf. auch weniger als n Stories angezeigt werden.

Der Aufruf von *getSimilarUserStories()* in der SimReqFinder-Klasse gliedert sich dabei in Anlehnung an den äußeren Teil des Algorithmus 3 aus Kapitel 3.4 durch die folgenden Methodenaufrufe:

Data: UseCaseStep
Data: StoriesToCompare
Result: HashMap<UserStory, Double> similarities

Termliste und TermFrequencies
TL1 = generateTermList(UseCaseStep);
TF1 = calculateTermFrequencies(TL1);

Links von Nachbar-Steps ermitteln
neighborLinks = getNeighborLinksUCS(UseCaseStep);

Iteration über alle UserStories *in* StoriesToCompare
TL2 = generateTermList(UserStory);
TF2 = calculateTermFrequencies(TL2);
TS = compareTermLists(TL1, TF1, TL2, TF2, conSim, DomMod);
similarities.put(UserStory, TS);

Algorithm 4: getSimilarUserStories(UseCaseStep, StoriesToCompare)

Wobei conSim (im Code contexSim) dem im Kapitel 3.3 vorgeschlagenen Faktor zur Einbeziehung der kontextuellen Übereinstimmung entspricht und entsprechend verrechnet wird, falls die aktuell verglichene UserStory bereits mit einem Nachbarn vom betrachteten Step verlinkt ist. DomMod bezeichnet die Termliste des Domain-Models, welches als String im Model 'Project' notiert wird und woraus zu Beginn der Methode *getSimilarUserStories()* bzw.

getSimilarUseCaseSteps() genau wie bei Beschreibungstexten von AADs eine Termliste generiert wird. Zudem wird im Code die entsprechende Topliste mit den relevanten AADs inklusive errechneter TS für bessere Transparenz auf der Konsole ausgegeben, da diese in der View aus Gründen der Übersichtlichkeit ausgeblendet wird.
Im Folgenden werden die in Algorithmus 4 auftauchenden Methoden noch etwas genauer betrachtet.

Methode: *generateTermList(Input)* In dieser Methode wird aus einem Input vom Typ String die Termliste entsprechend Kapitel 3.2.2 erzeugt, wobei zunächst mithilfe der Methode *descriptionPrettifier()* etwaige doppelte Anführungszeichen (") zu einfachen Apostrophen (') umgewandelt werden, um Fehlern vorzubeugen, da in Java bereits Strings durch (") begrenzt werden. Für den weiteren Verlauf wurde die Bibliothek 'Apache Lucene'[1] verwendet, um die erforderlichen Schritte Tokenization, Stop Word-Removal und Stemming umzusetzen. In Lucene wird die Grundlage für Textanalysen stets durch einen TokenStream gelegt, auf den dann verschiedenste Filter angewendet werden. In diesem Fall benutzen wir den *GermanAnalyzer*, um einen gültigen TokenStream des Input-Strings zu erzeugen und wenden dann einen *HunspellStemFilter* darauf an.

Stemming-Methoden Dieser ist eine relativ neue Kombination von einzelnen Filtern und erledigt praktischerweise alle drei Schritte auf einmal. Es wurde auch der *GermanStemFilter* getestet, wobei dort tendenziell mehr Overstemming auftrat. Leider hat der *HunspellStemFilter* in manchen Fällen aus bisher ungeklärten Gründen gewisse Terme doppelt auftauchen lassen, was wiederum die TermFrequency verfälscht. Das folgende Beispiel demonstriert diese Situation:

> **Input**: 'Falls das Datum weit in der Zukunft liegt, kann er auch das gewünschte Datum direkt über einen Datepicker wählen.'
> **German**: {fall, datum, weit, zukunf, lieg, gewunsch, datum, direk, datepick, wahl}
> **Hunspell**: {fall, datum, weit, zukunft, liegen, liegen, gewunscht, datum, direkt, direkt, datepick, wahl}

Zu erkennen ist, dass die Komposition 'Datepicker' bei beiden Stemmern nicht aufgelöst wird. Zu diesem Zweck wurde die Benutzung des *HyphenationCompoundWordTokenFilter* in Betracht gezogen, der mittels eines Wörterbuchs und klarer Silbentrennung (als .xml-Datei einzubinden) diesen Verbundtyp (siehe Kapitel 2) auflösen sollte, welcher aber in Ermangelung

[1] `http://lucene.apache.org/`

eines passenden Wörterbuches wieder verworfen wurde. Der *HunspellStemFilter* ist hingegen ein sogenannter Affix-Stemmer (der Präfixe und Suffixe entfernt), wobei eine Affix-Deklaration zusammen mit einem passenden Wörterbuch von OpenOffice frei verfügbar ist[2], weshalb diese Implementierung den *HunspellStemFilter* benutzt.

Methode: *compareTermLists(...)* In der zweiten Hauptmethode der Klasse SimReqFinder werden nun konkret zwei Termlisten miteinander verglichen, um einen Wert für die Terminological Similarity zwischen den zugehörigen AADs zu liefern, was dem inneren Teil von Algorithmus 3 entspricht. Dazu werden die beiden übergebenen Termlisten zunächst mithilfe der Hilfsmethode *combineTermLists()* gemäß Kapitel 3.2.4 zu einer gemeinsamen Termliste verbunden, über die dann iteriert wird. Für jeden einzelnen Term wird dabei ein eigenes Gewicht in Bezug auf die beiden Termlisten berechnet, dann aufaddiert und normiert. Das gesamte Gewicht eines Terms wird an dieser Stelle gemäß Kapitel 3.3 mit dem Faktor für das Domain Model multiplikativ verrechnet, falls der Term im Domain Model auftaucht. Dazu wird die Termliste des Domain Models ebenfalls an diese Methode übergeben. Wurde über alle Terme der gemeinsamen Termliste iteriert, wird dann noch der entsprechende Faktor für die kontextuelle Übereinstimmung multiplikativ mit dem Gesamtergebnis verrechnet und danach der Wert für die TS zwischen den Termlisten zurückgegeben.

Hilfsmethoden Die restlichen Methoden der SimReqFinder-Klasse sind Hilfsmethoden, um die Arbeit der Hauptmethoden zu entlasten. In *calculateTermFrequencies(Termliste)* wird lediglich die Vorgehensweise aus Algorithmus 1 in Kapitel 3.2.3 umgesetzt, wobei intern eine HashMap mit Tupeln aus String (Term) und Integer (Häufigkeit des Terms) gebildet wird. In *getNeighborLinksUCS(UseCaseStep)* (und analog *getNeighborLinksUS(UserStory)*) werden sämtliche mit den benachbarten Steps von UseCaseStep verlinkten UserStories zu einer Gesamtliste hinzugefügt, wobei darauf geachtet wird, dass UseCaseStep kein Nachbar von sich selbst sein kann und dass jede UserStory nur einmal auftritt, da mehrere Steps mit der gleichen Story verknüpft sein können. Ansonsten ist noch die Methode *entriesSortedByValue(HashMap)* erwähnenswert, die eine HashMap absteigend nach Value sortiert.

4.2 Erweiterbarkeit

Grundsätzlich ist die hier vorgestellte Methode auch auf Anforderungsdokumente anderen Typs anwendbar, solange ein natürlichsprachlicher Beschrei-

[2] `http://netix.dl.sourceforge.net/project/aoo-extensions/1075/13/dict-de_de-frami_2013-12-06.oxt`

bungstext erkennbar ist. Daraus würde genau wie bei UseCaseSteps und UserStories entsprechend Kapitel 3.2 mit der Methode *generateTermList()* eine Termliste generiert werden, was theoretisch mit beliebigen Strings funktioniert. Dazu kann zunächst die gesamte Klasse SimReqFinder.java auch in anderen Projekten verwendet werden, lediglich die Methoden *getNeighborLinksUS() bzw. -UCS* (zur Einbeziehung von Nachbar-Verknüpfungen, die z.B. bei Mockups gar nicht auftreten können) und die Hauptmethoden *getSimilarUserStories() bzw. -UseCaseSteps* sind speziell für den Vergleich von UserStories und UseCaseSteps entstanden und müssten entsprechend angepasst werden, wobei die grundlegende Struktur der Methoden zum Iterieren und paarweise Vergleichen erhalten bleiben kann.
Im Kontext dieser Implementierung ist die Einführung von zusätzlichen Models im IRE ungleich umfangreicher (auch, was die Darstellung/View in Play! betrifft) und man müsste sich aus einem Pool von verschiedenen Anforderungstypen für zwei entscheiden, die man vergleichen möchte, um die Methoden aus SimReqFinder.java weiterhin in der vorliegenden Form verwenden zu können.
In Kapitel 3.2.4 wird erwähnt, dass zusätzlich zur Term Frequency auch die Document Frequency zur Berechnung der TS geeignet sein könnte. Um diese nachträglich in die Implementierung einzubauen, sind zusätzliche Schritte nötig. Man könnte zum Beispiel Terme als neues Model betrachten und die zugehörige Document Frequency dort als Attribut abspeichern, auf das dann zur Laufzeit mit geringem Aufwand zugegriffen werden kann. Sobald im Projekt Beschreibungstexte von Dokumenten geändert werden, müsste zudem die DF aller Terme in diesen geänderten Dokumenten aktualisiert und beim Löschen eines Dokumentes die DF aller Terme im Dokument um 1 reduziert werden. Es böte sich alternativ noch die Möglichkeit an, sämtliche Terme als Liste im Projekt-Model zu halten und entsprechend zu verwalten. Dazu könnte man (ähnlich wie für Term Frequencies) HashMaps benutzen, um Tupel aus Term und DF abzuspeichern. Um die DF dann mit der TF zu verknüpfen (in Anlehnung an das sogenannte TF-IDF-Modell), würde man in der *compareTermLists()*-Methode beim Setzen des Termgewichtes die DF des Terms multiplikativ verrechnen (idealerweise als DF, nicht IDF, siehe Kapitel 3.2.4).

4.3 Probleme und Effizienz

Momentan wird bei jedem Aufruf von *getLinkedStoryRecommendations() bzw. getLinkedStepRecommendations()* die Termliste jedes verglichenen AADs erst neu gesetzt, was zu spürbaren Laufzeiteinbußen führt (circa. 13s Wartezeit, bis alles analysiert wurde und die Verlinkungs-Vorschläge zu sehen sind). Angedacht war daher ursprünglich, die Termliste eines AADs als Attribut an die jeweilige Model-Klasse zu binden, um zur Laufzeit mit z.B. *UseCase-*

Step.termList darauf zugreifen zu können. Das würde außerdem erfordern, dass die Termliste nur dann erneut generiert wird, wenn sich die Beschreibung des AADs ändert. Aufgrund von Problemen mit dem Konstruktor des Models lag bei diesem Vorgehen die Beschreibung eines AADs bei seiner Erstellung nicht korrekt im Model vor, sondern entsprach (vermutlich aufgrund des Play!-Formularsystems) dem null-Objekt, was folglich stets zu *NullPointerExceptions* führte. Idealerweise wird der Konstruktor des jeweiligen Models mit dem aus dem Eingabeformular entnommenen Beschreibungs-String aufgerufen und entsprechend die Termliste des AADs auf diesen Wert gesetzt. Um *NullPointerExceptions* zu vermeiden, wurde daher versucht, die Termliste im Konstruktor zunächst auf 'todo' zu setzen und dann die Update-Methoden in Model und Controller entsprechend abzuändern, damit wenigstens dort eine sinnvolle Termliste generiert wird (aus dem Beschreibungs-String des Formulars beim Erstellen eines neuen AADs im IRE), bislang jedoch ohne Erfolg. Es wäre außerdem denkbar, die Analyse (Erzeugen der Termlisten) der AADs im Projekt auszulagern und immer dann neu aufzurufen, wenn ein neues AAD hinzugekommen ist. Das würde im Endeffekt den Analyse-Aufwand aus der *getLinkedStoryRecommendations()*-Methode verschieben und dort die Wartezeit verkürzen, wobei man dann ggf. beim Hinzufügen vieler neuer AADs auf die Analyse warten müsste. Ein Vorteil des hier benutzten Verfahrens ist freilich, dass es keine Probleme mit Updates der AAD-Beschreibungen gibt, da bei jedem Aufruf von *getLinkedStoryRecommendations()* stets alle Termlisten neu generiert werden. Ob die Laufzeit im Endeffekt signifikant oder nur geringfügig verbessert wird, wäre eine interessante Betrachtung, die es aber aufgrund von Zeitmangel nicht mehr in die Implementierung geschafft hat.

Kapitel 5

Auswertung

In diesem Kapitel wird zunächst diskutiert, welche Methode sich am besten zur Evaluation eignet. Im Folgenden wird dann die in Kapitel 3 beschriebene Methode anhand eines ausgewählten Beispiel-Projektes evaluiert.

5.1 Methode zur Evaluation

Da das in Kapitel 3 vorgestellte Konzept auf den Methoden des Information Retrieval aufbaut, liegt es daher nahe, auch die typischen Evaluationsmethoden zur Bewertung der Relevanz von ausgegebenen Informationen bzw. Dokumenten zu verwenden. Wir betrachten daher die Precision und den Recall, gemäß Kapitel 2. Wenn wir eine hohe Precision anstreben, wird der Anteil der korrekten Ergebnisse maximiert. Das klingt zunächst durchaus erstrebenswert, aber wenn der Großteil der Ergebnisse korrekt ist, wird dem Nutzer die Entscheidung, zwischen zwei oder drei sehr ähnlichen Dokumenten eines zur Verlinkung auszuwählen, möglicherweise erschwert, was dann wiederum einen erhöhten Zeitaufwand zur Folge hat. Bei niedriger Precision ist nur ein geringer Anteil der Ergebnisse auch korrekt, sodass dem aufmerksamen Nutzer schneller klar werden dürfte, was man verlinken möchte und was direkt ausgeschlossen werden kann. Eine hohe Precision ist daher kein besonders gutes Maß für die Güte dieses Konzeptes.
Interessanter ist der Recall, bei dem errechnet wird, wieviele der korrekten Ergebnisse auch ausgegeben werden. Wenn von 5 korrekten Ergebnissen auch alle 5 in der Ausgabe auftauchen, ist das für uns wertvoller, als wenn eine hohe Precision erzeugt, aber nur ein Bruchteil aller korrekten Ergebnisse ausgegeben wird.
Aufgrund dieser Tatsache wäre es noch interessant, beide Methoden zu kombinieren und dabei Gewichte zuzuweisen. Dies ist mit der sogenannten F-Measure möglich, die ebenfalls häufig im IR angewendet wird, um ein harmonisches Mittel zwischen Precision und Recall zu bilden (mit $\beta = 1$).

In unserem Fall würde in der Gleichung

$$F = \frac{(\beta^2 + 1) * P * R}{\beta^2 * P + R} \tag{5.1}$$

für β ein Wert kleiner 1 gesetzt werden, um dem Recall ein höheres Gewicht als der Precision zuzuweisen (bspw. β=0.5 $\Rightarrow$ Recall fließt viermal stärker in die F-Measure ein). Da die Precision zwar nicht so bedeutend wie der Recall ist, aber eben auch nicht unbedeutend, würde die F-Measure sicherlich ein besser ausbalanciertes Ergebnis liefern. Aus Zeitmangel ist im Folgenden aber stattdessen nur der Recall berechnet und ausgewertet worden.

5.2 Die Länge der Topliste

Die gewählte Länge/Anzahl der ausgegebenen relevantesten Ergebnisse, der Topliste, liefert für die Präzision einen erwartungsgemäß nachteiligen Effekt. Je länger die Topliste, desto schlechter die Präzision, da die Wahrscheinlichkeit erhöht wird, FPs (False Positives, siehe Kapitel 2) in der Ausgabe zu haben. Um die Präzision zu maximieren, müsste man die Länge der Topliste an die Anzahl der TPs anpassen. Dies ist ein weiterer Nachteil der Precision, da so die FNs gar nicht beachtet werden. Auf den Recall hat die Länge der Topliste einen gegensätzlichen Einfluss. Ist die Topliste länger, erhöht sich die Wahrscheinlichkeit, dass alle korrekten Ergebnisse auch in der Ausgabe auftauchen, während eine zu kurze Topliste zu FNs führt, was den Recall negativ beeinträchtigt. Ein gutes Maß für die Länge der Topliste liefert George Miller in einer psychologisch motivierten Untersuchung [11], in der er unter Beachtung der Aufmerksamkeitsspanne und des Kurzzeitgedächtnisses zu dem Schluss kommt, dass ein Mensch im Durchschnitt ungefähr 7 Objekte auf einen Blick erkennen und einordnen kann. Deshalb wird in diesem Ansatz als Standardlänge für die Topliste der Wert 7 gewählt. Die Länge der Topliste erzeugt also einen Tradeoff zwischen Precision und Recall, bei dem wir uns, wie im vorigen Abschnitt beschrieben, dem Recall zuwenden wollen.

5.3 Berechnung des Recall

Um nun den Recall konkret berechnen zu können, wird zunächst festgelegt, was überhaupt als 'korrekt' gelten soll. Dazu nehmen wir ein Beispielprojekt aus dem in Kapitel 4 vorgestellten Integrated Requirements Environment her, in dem sich 31 UseCaseSteps in insgesamt 6 UseCases und 53 UserStories, verteilt auf 7 UserTasks, befinden. Hier werden zunächst sämtliche Links manuell gesetzt und notiert, was im Folgenden als 'korrekt' angesehen wird. Die Liste der UseCaseSteps und UserStories im Beispielprojekt ist im Anhang A respektive B zu finden und die manuell identifizierten Links sind in Tabelle 5.1 zu sehen. Da verlinkte AADs aber wie oben beschrieben aus

dem Pool von zu vergleichenden AADs verschwinden, könnten die manuell festgelegten Links gar nicht von der Methode erkannt werden. Daher sind die manuell festgelegten Links separat notiert worden, um die Ausgabe der Methode nicht zu verfälschen. Von 27 UseCaseSteps, zu denen überhaupt eine passende UserStory ermittelt wurde, sind 20 nur mit genau einer UserStory verbunden. Die restlichen 7 Steps haben eine Verbindung zu 2 Stories, weshalb die häufigste Verbindung eine 1-zu-1-Verbindung ist. Der durchschnittliche Recall wird somit berechnet, indem man für jedes einzelne AAD-Paar den Recall bestimmt und dann das arithmetische Mittel dieser Werte bildet. An dieser Stelle wird aus Gründen der Umsetzbarkeit nur die Richtung UseCaseStep $\longrightarrow$ UserStories betrachtet. Als Faktor für die Beeinflussung durch das Domain Model wurde 1.4 und für die Kontextuelle Übereinstimmung der Faktor 1.3 festgelegt. Das Domain Model besteht exemplarisch aus drei Termen und wurde auf 'kalender ressource benutzer' festgelegt. Die Länge der Topliste wurde auf 7 und die Minimalschwelle auf 0.15 gesetzt. Die Ergebnisse der Auswertung finden sich in Tabelle 5.2.

Warum nur eine Richtung? Im Projekt werden die Links immer in beide Richtungen gesetzt, d.h. ein UseCaseStep wird immer genau dann mit einer UserStory verknüpft, wenn diese UserStory mit ihm verknüpft wird. Daher spielt es intuitiv keine Rolle, welche Richtung man betrachtet. Es ist zu erwarten, dass nicht jedes AAD mit einem AAD des anderen Typs verlinkt werden kann (und dadurch isoliert bleibt). Die so betroffenen UseCaseSteps bzw. UserStories können nicht in die Berechnung für den Recall einbezogen werden, da die Anzahl der als korrekt identifizierten Antworten null betragen und man an den entsprechenden Stellen durch null teilen würde. Im betrachteten Beispiel konnte für 4 von 31 UseCaseSteps keine Verbindung zu einer UserStory identifiziert werden, das entspricht einem Anteil von ungefähr 13%. Umgekehrt sind ganze 28 von 53 UserStories isoliert, was bereits 53% aller Stories ausmacht. Deshalb ist eine minimale Anzahl von isolierten AADs anzustreben, was im betrachteten Beispiel eben genau der Richtung UseCaseStep $\longrightarrow$ UserStories entspricht.

Alles korrekt verlinkt? Angenommen, zu einem UseCaseStep x wurden 2 UserStories als korrekte Links identifiziert. Das Programm gibt 7 Ergebnisse aus, in denen beide Stories vorkommen. Der Recall wäre 1 und die Precision $2/7 \approx 0.28$. Danach setzt der User den Link von x zu den beiden korrekt erkannten Stories und beide verschwinden aus dem Pool von zu vergleichenden UserStories für diesen Step x. Fügt der User zwischenzeitlich keine weiteren UserStories zum Projekt hinzu, sucht aber trotzdem ein weiteres Mal nach potentiell verlinkbaren UserStories für Step x, werden mit sehr hoher Wahrscheinlichkeit nur noch irrelevante False Positives in der Topliste auftauchen, was einer Präzision von 0% entspräche. Diese Situati-

on erscheint zunächst recht unrealistisch, da aber nicht garantiert werden kann, dass die ursprüngliche Festlegung von korrekten Links fehlerfrei war und dadurch doch noch False Negatives existieren, ist es daher auch nicht empfehlenswert, an dieser Stelle bereits keine Ausgabe mehr zu liefern, da dem Nutzer so möglicherweise Links entgehen. Es wird daher eine Topliste generiert, solange es Ergebnisse gibt, die über der Minimal-Schwelle liegen.

Ergebnisse und Diskussion Von insgesamt 34 manuell erkannten Links wurden 11 nicht erkannt (zu erkennen an dem NA in Tabelle 5.2), bzw. tauchten nicht in der Topliste auf. In drei Fällen (UC1, Step 5; UC3, Step 2; UC4, Step 8) ist es allerdings möglich, dass durch das Verlinken einer korrekt ausgegebenen UserStory (S6 bzw. S35 bzw. S10) diese False Negatives nachrücken können (wie in Kapitel 3.5 beschrieben), wodurch der Recall verbessert werden würde.
Außerdem ist zu bemerken, dass insgesamt sechsmal eine TS größer 1.0 ausgerechnet wurde. In all diesen Fällen (zu erkennen am ! in Tabelle 5.1 und 5.2) stimmt die Beschreibung des UseCaseSteps exakt mit der Beschreibung der UserStory überein, was auf Copy & Paste schließen lässt. Die Übereinstimmung wurde dann durch die zusätzliche Gewichtung von Domain Model (40% Zuwachs) und/oder Kontextueller Übereinstimmung (30% Zuwachs) verfälscht, was den Wert größer 1.0 erklärt.
Im Großen und Ganzen erweist sich die Wahl der Zusatzgewichte für das Domain Model und die Kontextuelle Übereinstimmung aber als durchaus sinnvoll. Zwar sorgt die zusätzliche Gewichtung des Domain Models beim Paar (UC2, Step 1 $\longleftrightarrow$ Story 30) dafür, dass die Story nicht ausgegeben wird (False Negative) und beim Paar (UC2, Step 3 $\longleftrightarrow$ Story 31) für eine Verfälschung der TS und Positionierung auf Platz 2 in der Topliste, doch beim Paar (UC4, Step 2 $\longleftrightarrow$ Story 23) führt sie zu einer besseren Positionierung der Story in der Topliste: Dort stimmen die Terme 'kalender' und 'zeig' mit Story 23 und die Terme 'woche' und 'zeig' mit Story 27 überein. Da 'kalender' aber ein Term des Domain Models ist, wird hier korrekterweise ein höheres Gewicht festgelegt, obwohl in beiden Fällen zwei Terme übereinstimmen. Dadurch ergibt sich eine bessere Abgrenzung zwischen Story 23 auf Platz 1 mit einer TS von 0.6 und Story 27 auf Platz 2 mit einer TS von 0.45. Natürlich hängen die Auswirkungen der Domain Model-Gewichtung auch stark von der Wahl des Domain Models selbst ab, vermutlich wäre hier ein passenderes Domain Model hilfreicher.
Zu Auswirkungen der Wahl des Faktors contexSim für die Kontextuelle Übereinstimmung von 1.3 konnte leider keine Aussage getroffen werden, da im untersuchten Beispielprojekt aus oben genannten Gründen keine Verknüpfungen existieren durften. Es ist aber zu erwarten, dass in den Fällen, wo beide Faktoren, also sowohl der Domain-Model-Faktor als auch contexSim, einfließen, der ausgegebene TS-Wert etwas zu hoch liegt und beide Faktoren

UseCase-Links		
UseCase 1	Step 1	S1
	Step 2	S1
	Step 3	S51
	Step 4	S2, S3
	Step 5	S2, S6!
	Step 6	S53!
	Step 7	S1!
	Extension 1	S28!
UseCase 2	Step 1	S30
	Step 2	S33, S30
	Step 3	S31
	Step 4	S32, S34!
UseCase 3	Step 1	S35
	Step 2	S36, S38
UseCase 4	Step 1	NA
	Step 2	S23
	Step 3	S18
	Step 4	S19
	Step 5	S11
	Step 6	NA
	Step 7	S7
	Step 8	S10, S8
	Step 9	NA
	Extension 1	S20!
UseCase 5	Step 1	NA
	Step 2	S22
	Step 3	S52!
UseCase 6	Step 1	S43
	Step 2	S7, S11
	Step 3	S15
	Step 4	S24

Tabelle 5.1: Identifizierte Links zwischen UseCaseSteps und UserStories. NA bedeutet, dass kein Link identifiziert werden konnte und ein ! bedeutet, dass der Text identisch war.

UseCase	Step	Story 1 (ID, Pos, Sim)	Story 2 (ID, Pos, Sim)	Recall
UseCase 1	Step 1	(S1, 1, 0.4)	-	1
	Step 2	(S1, 1, 0.74)	-	1
	Step 3	(S51, NA, NA)	-	0
	Step 4	(S2, 1, 0.64)	(S3, 4, 0.37)	1
	Step 5	(S2, NA, NA)	(S6, 1, 1.02)!	0.5
	Step 6	(S53, 1, 1.03)!	-	1
	Step 7	(S1, 1, 1.13)!	-	1
	Extension 1	(S28, 1, 1.08)!	-	1
UseCase 2	Step 1	(S30, NA, NA)	-	0
	Step 2	(S33, 1, 0.89)	(S30, 3, 0.61)	1
	Step 3	(S31, 2, 0.64)	-	1
	Step 4	(S32, 2, 0.85)	(S34, 1, 1.05)!	1
UseCase 3	Step 1	(S35, NA, NA)	-	0
	Step 2	(S35, 1, 0.25)	(S38, NA, NA)	0.5
UseCase 4	Step 1	-	-	-
	Step 2	(S23, 1, 0.6)	-	1
	Step 3	(S18, NA, NA)	-	0
	Step 4	(S19, 4, 0.16)	-	1
	Step 5	(S11, 1, 0.54)	-	1
	Step 6	-	-	-
	Step 7	(S7, 1, 0.43)	-	1
	Step 8	(S10, 1, 0.37)	(S8, NA, NA)	0.5
	Extension 1	(S20, 1, 1.0)!	-	1
UseCase 5	Step 1	-	-	-
	Step 2	(S22, NA, NA)	-	0
	Step 3	(S52, 1, 1.05)!	-	1
UseCase 6	Step 1	(S43, NA, NA)	-	0
	Step 2	(S7, NA, NA)	(S11, NA, NA)	0
	Step 3	(S15, 3, 0.28)	-	1
	Step 4	(S24, 1, 0.54)	-	1

Tabelle 5.2: UseCaseSteps und verwandte UserStories der Ausgabe. Jede UserStory ist durch ihre ID, ihre Position in der Topliste und ihre berechnete TS dargestellt. NA bedeutet, dass die Story nicht ausgegeben wurde (FN) und ein ! deutet auf identische Beschreibungstexte hin.

entsprechend leicht reduziert werden müssten.
Auch wenn die Positionierung des Ergebnisses in der Topliste keine Rolle für den Recall spielt, fällt auf, dass sich das korrekte Ergebnis ingesamt 16-mal an oberster Stelle in der Topliste wiederfindet. In lediglich vier Fällen taucht ein korrekt erkannter Link unterhalb von Position 2 in der Topliste auf, was hauptsächlich auf die Verwendung von inkonsistenter Terminologie zurückzuführen ist.
Wie in Tabelle 5.2 erkennbar ist, liegt die geringste Terminological Similarity eines True Positives bei 0.16 (UC4, Step 4 $\longleftrightarrow$ Story 19), wobei der geringe Wert und die niedrige Position in der Topliste auf Inkonsistenzen bei der Terminologie hindeuten: Die (gestemmten) Termlisten der besagten AADs wurden zu

UC4, Step 4 = {fall, datum, weit, zukunf, lieg, gewunsch, datum, direk, datepick, wahl}
und **Story 19** = {bestimm, datum, dat, pick, eingeb}

ermittelt, wobei auffällt, dass der Verbund 'Date-Picker' korrekterweise am Bindestrich gebrochen und auf die Terme 'date' und 'picker' reduziert wurde, während 'Datepicker' als einzelner Term betrachtet wird. Somit wird mit dieser Methode leider nur einer von zwei Verbundtypen (gemäß Kapitel 2) korrekt behandelt, die Benutzung beider Verbundtypen für das gleiche Konzept wird daher als terminologisch inkonsistent betrachtet und wirkt sich entsprechend negativ auf die TS aus. Zudem wurden aufgrund der Sprachabhängigkeit des benutzten Stemmers (siehe Kapitel 2) die Terme 'datum' und 'date' nicht auf den gleichen Wortstamm reduziert. Der gewählte minimale TS-Schwellwert von 0.15 erweist sich angesichts dieses Beispiels allerdings als durchaus brauchbar.

Bildet man nun das arithmetische Mittel über die Werte der Spalte 'Recall' in Tabelle 5.2, erhält man für den durchschnittlichen Recall einen Wert von

$$Recall_{avg} = \frac{17 * 1 + 7 * 0 + 3 * 0.5}{27} = \frac{18.5}{27} \approx 0.685 \tag{5.2}$$

Auffällig ist an dieser Stelle auch, dass der Recall für ein einzelnes Paar immer nur die Werte 0, 1 oder 0.5 annehmen kann, da zu einem UseCaseStep maximal zwei verwandte UserStories identifiziert werden konnten.

Bewertung und Einordnung Im Vergleich zu Natt och Dag et al. [3] muss zunächst festgehalten werden, dass diesen eine Experten-Analyse der möglichen Links zur Verfügung stand und dort Customer Wishes (Kundenseite) mit Product Requirements (Entwicklerseite) verglichen werden, die

deutlich umfangreicher ausfallen als die hier verglichenen AADs. Im Rahmen ihrer Arbeit ergab sich ein Recall zwischen 0.41 und 0.51 und im besten Fall ein Wert von 0.65. Trotz Unterschieden im untersuchten Szenario (unter anderem wurden in [3] keine Strukturfaktoren wie die Kontextuelle Übereinstimmung und das Domain-Model mit einbezogen) wird in Anwendung des Szenarios dieser Arbeit ein ähnlicher Wert für den durchschnittlichen Recall erreicht. Daraus lässt sich eine gewisse Flexibilität der vorgestellten Methode zur Berechnung der Terminological Similarity im Hinblick auf die Struktur von Beschreibungstexten ableiten, was bereits in Kapitel 4.2 angedeutet wurde.

Verbesserung des Recalls Wie bereits erwähnt, könnten auch bisher nicht erkannte False Negatives im Ergebnis auftauchen, falls nach erfolgreicher Verknüpfung das verknüpfte AAD aus dem Vergleichspool gelöscht und das Nachrücken gemäß Kapitel 3.5 ermöglicht wird, wodurch der Recall verbessert werden kann, sollte sich das entsprechende FN nicht zu weit außerhalb der Topliste befinden (siehe dazu Abbildung 3.9). Eine minimale Verlängerung der Topliste bis zu einem Wert von etwa 9 [11] kann auch zu einem höheren Recall beitragen, allerdings sollte man beachten, dass ein durchschnittlicher Recall von 100% praktisch nicht zu erreichen ist, da dann stets alle korrekten Ergebnisse auftauchen müssten, was bei hinreichend vielen identifizierten Links den Umfang der Topliste sprengen würde. Um einen genaueren Wert für den durchschnittlichen Recall zu bekommen, wäre ein umfangreicheres Projekt mit deutlich mehr UserStories und UseCaseSteps denkbar besser geeignet. Es wäre dann eine genauere Einteilung im Wertebereich des Recalls für einzelne Paare zu erwarten (nicht nur 0, 1 oder 0.5), was gegebenenfalls zu einem Durchschnittswert führen würde, der aus einer breiteren Fächerung von einzelnen Werten zusammengesetzt ist. Die Verwendung von Copy & Paste für Beschreibungen von AADs ist hinderlich und führt zu unerwünschten Werten für die TS, je nach Wahl der zusätzlichen Gewichte für Domain Model und Kontextuelle Übereinstimmung im Bereich größer 1. Außerdem wäre die Identifikation von Links durch einen Domänen-Experten (z.B. einen erfahrenen Requirements Engineer) sinnvoller, da die Fehlerquote erwartungsgemäß geringer ausfallen würde.

Kapitel 6

Fazit

In diesem abschließenden Kapitel sollen die gewonnenen Erkenntnisse gesammelt und Verbesserungsmöglichkeiten für weitere Arbeiten angeführt werden.
Zunächst wurde in Kapitel 3.1 die Erkenntnis gewonnen, dass sich korpusbasierte Verfahren tendenziell besser eignen als wissensbasierte, um eine große Menge AADs von geringem Umfang zu vergleichen, da der Aufwand für den Requirements Engineer im Einzelnen deutlich geringer ausfällt.
Es hat sich gezeigt, dass die vom IRE (siehe Kapitel 4) bereitgestellte Funktion zum Erzeugen einer UserStory aus einem UseCaseStep (mit entsprechend identischem Beschreibungstext) zu TS-Werten größer 1 führt, falls zusätzliche Gewichte für ein vorgegebenes Domain Model und/oder vorhandene Verknüpfungen in die Berechnung einbezogen werden (siehe Kapitel 5.3).
Zudem führt die Benutzung von inkonsistenter Terminologie (z.B. Synonymen oder der gleiche Begriff in verschiedenen Sprachen) erwartungsgemäß zu einer Verfälschung der TS.
Im Vergleich zu der direkt verwandten Arbeit von Natt och Dag et al. [3], deren Methode die Grundlage für das in Kapitel 3.2 erarbeitete Konzept geliefert hat, führt die vorliegende Methode trotz Unterschieden im Anwendungsszenario zu ähnlichen Werten, wobei ein durchschnittlicher Recall-Wert von 0.685, oder 68.5%, erreicht wird. Damit werden fast 70% aller korrekten Verknüpfungen von der Methode korrekt berechnet und vorgeschlagen, was die Arbeit und den zeitlichen Aufwand des Requirements Engineers beim Suchen nach möglichen Verknüpfungen, um die Traceability und damit die Konsistenz der Anforderungsdokumente zu verbessern, signifikant reduzieren dürfte.

Verbesserungsvorschläge Die hier genannten Möglichkeiten zur Erweiterung haben es hauptsächlich aus Zeitgründen nicht in die Implementierung geschafft, könnten die Methode aber insgesamt verbessern.
Wie in Kapitel 3.2 erläutert, könnte zusätzlich die Document Frequency zur Berechnung herangezogen werden, um möglicherweise ein passenderes Er-

gebnis für die Terminological Similarity zu erhalten. Eine Möglichkeit, dies zu realisieren, wird in Kapitel 4.2 aufgezeigt.
Eine alternative Bewertung der zusätzlichen Einflüsse durch Domain-Model und vorhandene Verknüpfungen wird in Kapitel 3.3 vorgeschlagen, wobei diese dann nicht faktoriell mit der TS multipliziert werden, sondern eine Linearkombination dieser Einflüsse und der TS berechnet wird, wodurch diese einzeln gewichtet werden können.
Ein weiterer Kritikpunkt ist die aktuelle Implementierung der Termlisten, welche bei jedem Aufruf der Vergleichsmethode für sämtliche zu vergleichenden AADs neu erzeugt werden, was die Ausgabe der verwandten AADs verzögert. Ein Vorschlag zur Lösung dieses Problems wird in Kapitel 4.3 dargestellt.
Wie weiterhin in Kapitel 5 ersichtlich ist, könnte die Methode zusätzlich mit einer geeigneten F-Measure ausgewertet werden, um auch die Precision mit einzubeziehen, was potentiell eine besser ausbalancierte Auswertung ermöglicht.
Zuletzt könnten noch Synonyme von Termen in einem Synonym-Wörterbuch nachgeschlagen und entsprechend in die Berechnungen für die TS einbezogen werden, um Inkonsistenzen in der Terminologie zu beseitigen und die ausgegebenen TS-Werte zu verbessern.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] D. Berry. Natural language and requirements engineering - nu?, 2001.

[2] K. Breitman and J. Sampaio do Prado Leite. Ontology as a requirements engineering product. In *Proceedings of the 11th IEEE International Requirements Engineering Conference, IEEE Computer Society Press (2003)*, pages 309–319, 2003.

[3] J. N. o. Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, RE '04, pages 283–294, Washington, DC, USA, 2004. IEEE Computer Society.

[4] D. Faure and C. Nedellec. Asium: Learning subcategorization frames and restrictions of selection. In *Proceedings of the 10th Conference on Machine Learning (ECML)*, 1998.

[5] L. Goldin and D. M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Autom. Softw. Eng.*, 4(4):375–412, 1997.

[6] S. Harispe, S. Ranwez, S. Janaqi, and J. Montmain. *Semantic Similarity from Natural Language and Ontology Analysis*. Morgan and Claypool, 2015.

[7] R. Jiang, M. Gan, and X. Dou. From Ontology to Semantic Similarity: Calculation of Ontology-Based Semantic Similarity. *The Scientific World Journal*, 2013.

[8] L. Kof. Natural language processing: Mature enough for requirements documents analysis? In A. Montoyo, R. Muñoz, and E. Métais, editors, *NLDB*, Lecture Notes in Computer Science, pages 91–102. Springer, 2005.

[9] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[10] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[11] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information, 1956.

[12] G. Nenadić, I. Spasić, and S. Ananiadou. Automatic discovery of term similarities using pattern mining. In *COLING-02 on COMPUTERM 2002: second international workshop on computational terminology - Volume 14*, COMPUTERM '02, pages 1–7, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[13] K. Toutanova, D. Klein, C. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL*, pages 252–259, 2003.

Anhang A

UseCases im Beispielprojekt

Use Case UC1: Ressourcen verwalten

1. 'Wenn eine neue Ressource (Raum, Beamer, Laptop) ängeschafft"wurde, möchte der Administrator diese eintragen'
2. 'Dazu öffnet er die Ressourcenverwaltung im System und wählt neue Ressource anlegen". Er trägt den Namen und Beschreibung der Ressource ein und bestätigt.'
3. 'Daraufhin zeigt ihm das System die aktualisierte Ressourcenliste an.'
4. 'Genauso kann der Administrator bestehende Ressourcen ändern oder löschen.
5. 'Dazu geht er bei einer der Ressourcen in der Liste 'ändern' bzw. 'löschen' aus. Zum ändern kann er die Daten in einem Formular ändern.'
6. 'Wenn der Administrator eine Ressource löscht, die in einem zukünftigen Termin noch reserviert ist, warnt ihn das System, dass er diese Termin prüfen sollte.'
7. 'Der Administrator schaut sich die jeweiligen Termine an und weist ihnen neue Ressourcen zu.'
8. 'Eine neue Ressource anlegen'
9. Extension: 'Ich kann nach Ressourcen-Kategorien filtern (Räume, Rechner)'

Use Case UC2: Kategorien verwalten

1. 'Um die Ressourcen besser überblicken zu können und um den Kalender übersichtlicher darstellen zu können, kann der Administrator Kategorien für Ressourcen anlegen.'
2. 'In der Ressourcenverwaltung kann er dazu Kategorien anlegen und den Ressourcen eine Kategorie zuordnen'
3. 'Wenn der Administrator eine Kategorie löscht, bleiben die Ressourcen zu dieser Kategorie erhalten. Sie werden aber als öhne Kategorieängegeben.'
4. 'Die Ressourcen werden entsprechend ihrer Kategorien sortiert angezeigt'

Use Case UC3: Benutzer verwalten

1. 'Der Administrator trägt Nutzer, die Zugang zum System haben, ein.'
2. 'Er kann ihre Daten nachträglich ändern. Das Passwort kann er nur neu setzen, er kann es jedoch nicht einsehen.'

Use Case UC4: Eine neue Reservierung eintragen

1. 'Der Mitarbeiter möchte gern z.B. für eine Besprechung einen Raum und ein Notebook reservieren. Dazu öffnet er den Kalender.'
2. 'Der Kalender wird in der Wochenansicht angezeigt und zeigt die aktuelle Woche an.'
3. 'Der Nutzer navigiert mit 'Woche vor' zum gewünschten Datum.'
4. 'Falls das Datum weit in der Zukunft liegt, kann er auch das gewünschte Datum direkt über einen Datepicker wählen.'
5. 'Er klickt auf die entsprechende Uhrzeit im Kalender und trägt seinen Termin dort ein.'
6. 'Wenn er dabei bemerkt, dass die Ressource, die er reservieren will, noch gar nicht im System ist, kann er diese anlegen.'

7. 'Nachdem der Nutzer alle Daten eingegeben hat, wählt er 'Termin anlegen'.'
8. 'Falls eine seiner Ressourcen zur gewünschten Zeit belegt ist, warnt ihn das System und lässt ihn die Termindaten ändern.'
9. 'Der Mitarbeiter kann nur die Daten korrigieren oder die Warnung ignorieren und die Reservierung trotzdem anlegen. Das System legt den Termin entsprechend an.'
10. Extension: 'Statt 7-Tage-Woche sollte mir eine Woche nur von Montag bis Freitag angezeigt werden. So wird nicht so viel Platz verschwendet.'

Use Case UC5: Reservierungen einsehen

1. 'Der Nutzer wählt 'Reservierungen einsehen' und bekommt eine Liste aller Ressourcen zur Auswahl'
2. 'Dort wählt der Nutzer eine Ressource aus und bekommt eine Kalenderansicht zu sehen, in der alle Reservierungen dieser Ressource eingetragen sind.'
3. 'Der Nutzer kann nun diesen Kalender drucken oder auch als HTML bzw. XML-Datei exportieren.'

Use Case UC6: Mehrere Reservierungen zum selben Thema eintragen

1. 'Der Mitarbeiter möchte mehrere Reservierungen anlegen (z.B. wenn in einer Woche zu drei Terminen Gruppenübungen stattfinden)'
2. 'Dazu trägt er eine Reservierung wie gewohnt ein.'
3. 'Wenn er jetzt die nächste Reservierung anlegen will, ist das Formular mit den Daten aus der letzten Reservierung vor-ausgefüllt. Das Datum ist angepasst auf das neue Datum, das er im Kalender angeklickt hat.'
4. 'Der Mitarbeiter gibt allen Reservierungen zu dieser Gruppenübung bzw. Lehrveranstaltung dieselbe Farbe, damit das im Kalender erkennbar ist.'

Anhang B

UserTasks im Beispielprojekt

UT1: Ressourcen verwalten

S1: Eine neue Ressource anlegen

S4: Ressourcen-Dialog sollte so aussehen wie der Dialog zur Userverwaltung

S6: Wenn der Administrator eine Ressource löscht, die in einem zukünftigen Termin noch reserviert ist, warnt ihn das System, dass er diese Termin prüfen sollte.

S33: Einer Ressource eine Kategorie zuordnen.

S34: Die Ressourcen werden entsprechend ihrer Kategorien sortiert angezeigt

S51: Der Nutzer wählt Reservierungen einsehenünd bekommt eine Liste aller Ressourcen zur Auswahl

S2: Eine Ressource löschen

S3: Ressourcen editieren

S55: Der Administrator schaut sich die jeweiligen Termine an und weist ihnen neue Ressourcen zu.

UT2: Aufträge

S5: Doppelte Ressourcen im Production-System entfernen

S48: Bug: Tests laufen nicht ohne DB

S49: Datenquelle konfigurierbar machen

S50: DB-Zugriff zentralisieren (alles in eine Klasse)

S46: Alle Funktionen auch über Menü bereitstellen (Reservierungen eintragen, Ressourcen verwalten, ...)

UT3: Termine verwalten

S10: Das System erkennt, wenn es einen Terminkonflikt gibt, d.h. wenn eine Ressource zeitgleich mehrfach reserviert werden soll. Das System warnt dann den User

S24: Ich kann einer Reservierung eine Farbe zuordnen. Mit dieser Farbe wird die Reservierung dann im Kalender angezeigt.

S7: Einen neuen Termin anlegen

S8: Einen Termin ändern

S11: Ich kann einen neuen Termin eintragen, indem ich direkt im Kalender auf die entsprechende Stelle klicke

S12: Ich kann einen Termin editieren, indem ich ihn direkt im Kalender anklicke

S17: Winter- und Sommerzeit nicht berücksichtigen

S9: Einen Termin löschen

S13: Wenn ich einen neuen Termin über das Menü anlege, wird das Datum standardmäßig auf das heutige Datum gesetzt hat

S15: Komfortfunktion: Das System merkt sich die Daten (Terminname, Abkürzung, Ressourcen), wenn ich gerade einen Termin angelegt habe. Wenn ich jetzt einen neuen Termin anlege, wird das Formular mit diesen Daten vorausgefüllt. Nur die Terminzeit wird angepasst.

S16: Beim Anlegen eines Termins darf die Endzeit nicht vor der Startzeit liegen.

S14: Fehler im Termin-Dialog differenziert anzeigen (leerer Terminname vs. keine Ressourcen angegeben, etc.)

UT4: Alle belegten Ressourcen im Kalender anzeigen

S18: Kalendernavigation: Ich kann im Kalender um Wochen nach vorn und hinten springen. Es gibt ein extra-Feld für 'Heute'

S20: Statt 7-Tage-Woche sollte mir eine Woche nur von Montag bis Freitag angezeigt werden. So wird nicht so viel Platz verschwendet.

S23: Kalender zeigt zum Systemstart immer das heutige Datum an

S25: Kontraste für die Termin-Farben erhöhen.

S28: Ich kann nach Ressourcen-Kategorien filtern (Räume, Rechner)

S29: Kalender soll immer an die Fenstergröße angepasst werden. Wenn das Fenster zu klein wird (aber nur dann), soll ein Scrollbalken erscheinen

S27: Zeige alle Ressourcenkonflikte für eine Woche separat explizit an

S21: Ich kann mir eine Monatsübersicht anzeigen lassen. Dazu wird jede Reservierung, die an einem Tag vorhanden ist, mit der Abkürzung in der entsprechenden Farbe angezeigt.

S47: Ich kann den Kalender in iCal exportieren

S52: Der Nutzer kann nun diesen Kalender drucken oder auch als HTML bzw. XML-Datei exportieren.

S26: Markiere Reservierungen, die einen Ressourcenkonflikt besitzen

S22: Ich kann die Ansicht nach Reservierungen, die bestimmte Ressourcen beinhalten, filtern lassen

S19: Ich kann ein bestimmtes Datum über einen Date-Picker eingeben.

UT5: Kategorien verwalten

S30: Eine neue Kategorie anlegen

S31: Eine Kategorie löschen. Ressourcen, die der zu löschenden Kategorie zugeordnet waren, werden dann einfach als 'ohne Kategorie' dargestellt.

S32: Zeige Ressourcen sortiert nach Kategorien an

UT6: Benutzer verwalten

S35: Einen neuen Benutzer anlegen

S39: Benutzer müssen sich in das System einloggen

S36: Einen Benutzer ändern

S38: Passwort eines Benutzers zurücksetzen

S40: User bleibt eingeloggt

S41: Rollen werden unterschieden (Admin, User)

S42: Nur Admins dürfen Nutzer verwalten

S37: Benutzer löschen

UT7: Wiederholbare Termine verwalten

S43: Ich kann Termin-Serien erstellen (wöchentlich wiederkehrend)

S44: Lösche/Editiere einzelne Termine aus Serien

S45: Lösche eine ganze Serie

Anhang C

Inhalt der CD

Die CD beinhaltet:

- das IRE inklusive SimReqFinder als Eclipse-Projekt, im .zip-Archiv verpackt
- diese Ausarbeitung als .pdf-Datei und als .tex-Datei im Unterordner Ausarbeitung
- das aus dem Code generierte JavaDoc im entpackten Unterordner /doc
- und das zur Auswertung verwendete Beispielprojekt, eingetragen in die mitgelieferte Datenbank (ebenfalls im .zip-Archiv)

Betriebsanleitung (für Play!):

1. Das Archiv (SimReqFinder_final.zip) in Verzeichnis x entpacken
2. Über die Systemkonsole zu x navigieren
3. 'activator run' eingeben
4. Im Browser 'localhost:9000' aufrufen
5. Auf Kompilierung warten
6. Loslegen

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 01.11.2015

_____

Thomas Winter