Karlsruher Institut für Technologie

Verknüpfung von Text- und Modellentitäten von Softwarearchitektur-Modellen mithilfe von Wortvektoren

Bachelor's Thesis von

Robin Richard Schulz

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Anne Koziolek
Zweitgutachter:	Prof. Ralf Reussner
Betreuender Mitarbeiter:	M.Sc. Jan Keim
Zweiter betreuender Mitarbeiter:	M.Sc. Yves Schneider

08. Juli 2019 – 07. November 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 07. November 2019

. . . . . .

(Robin Richard Schulz)

Zusammenfassung

Die Verknüpfung von Entitäten von Softwarearchitektur-Dokumentationen und -Modellen bietet Möglichkeiten, die Qualität eines Software-Entwicklungsprozesses zu verbessern. Zu diesen Möglichkeiten zählen beispielsweise die Analyse der Konsistenz und das Übertragen von Änderungen zwischen den Artefakten Dokumentation und Modell. Um solche Verknüpfungen automatisiert berechnen zu können, bedarf es Methoden der linguistischen Datenverarbeitung. In der vorliegenden Arbeit wird ein Verfahren präsentiert, welches dieses Problem mithilfe von Wortvektoren angeht. Diese werden für vielfältige Aufgaben der linguistischen Datenverarbeitung genutzt und übertreffen dabei häufig herkömmliche Verfahren. In dieser Arbeit werden sie dazu verwendet, Ähnlichkeiten zwischen Entitäten zu berechnen. Dabei wird zwischen Nominalphrasen und Verbphrasen unterschieden, wobei Erstere mit nicht-relationalen und Letztere mit relationalen Architekturmodell-Entitäten verknüpft werden können. Dazu wird mithilfe des PARSE-Rahmenwerks ein Graph aufgebaut, der den Dokumentationstext repräsentiert und eine Klassifikation von Phrasen und Wörtern enthält. Anhand dieses Graphen und einer Repräsentation des Architekturmodells kann das in dieser Arbeit entwickelte Verfahren Verknüpfungen berechnen und an den Graphen annotieren. Es wurden verschiedene Parametrisierungen des Verfahrens evaluiert, wobei allgemein festgehalten werden kann, dass das Referenzverfahren, ein String-Ähnlichkeitsvergleich mithilfe der Levenshtein-Distanz, im F_1-Maß überboten werden kann. Für Nominalphrasen wurde ein durchschnittlicher Vorsprung von 13.6 Prozentpunkten des Wortvektor-Verfahrens gegenüber dem Levenshtein-Verfahren gemessen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Für den Erfolg und die Langlebigkeit eines Softwaresystems ist die Erstellung und Pflege von Architektur-Dokumentationen von entscheidender Bedeutung. Erstellt man also ein Architekturmodell samt zugehöriger textueller Beschreibung der Entwurfsentscheidungen und der Funktion der Software, hat man bereits zwei Artefakte, welche dieselbe Sache beschreiben: Die zu entwickelnde oder bereits entwickelte Software. Es liegt also nahe, diese beiden Artefakte miteinander zu verknüpfen. Eine solche Verknüpfung von Dokumentationstext und Architekturmodell ermöglicht vielfältige Anwendungen mit dem Ziel, das Verständnis über das vorliegende Softwareprojekt zu verbessern um zeitliche und wartungstechnische Aufwände zu reduzieren und die Qualität des Entwicklungsprozesses zu verbessern. Beispielsweise können mit Hilfe solcher Verknüpfungen zu einer gegebenen Modell-Komponente die erklärenden Passagen im textuellen Teil der Dokumentation gefunden werden, was insbesondere bei komplexeren Architekturen den Verständnisprozess beschleunigt.

Ein weiterer Anwendungsfall ist die Konsistenzanalyse. Parnas nennt die Verschlechterung der Dokumentation durch fehlendes Anpassen bei neuen Änderungen als eine Ursache für die Verschlechterung von Software über die Zeit und damit auch für höhere Entwicklungs- und Wartungskosten [18]. Wohlrab et al. haben 2019 in einer Umfrage [31] festgestellt, dass Inkonsistenzen in Wortlaut und Sprache relativ häufig seien und auch einen negativen Einfluss haben. In [11, 12] wurden veraltete Dokumentation und Inkonsistenzen zwischen Artefakten (wie z.B. Architekturmodellen und Dokumentationstexten) als aktuelle Probleme bzw. Nachteile der Nutzung von Dokumentation ausgemacht. Es wird eine Vision vorgestellt, die dem Alterungsprozess von Software entgegenwirken soll. Dabei spielt die Konsistenzanalyse eine wichtige Rolle: Die Änderungen an einem Artefakt sollen auch auf andere Artefakte übertragen werden. Auch zu diesem Anwendungsfall kann das in dieser Arbeit entwickelte Verfahren im Spezialfall der Artefakte *Dokumentationstext* und *Architekturmodell* einen Teil beitragen: Für die Feststellung, was bei Änderungen in einem Artefakt im Anderen genau geändert werden soll, sind Verknüpfungen zwischen Entitäten der beiden Artefakte essenzielle Informationen.

Für die Berechnung der Verknüpfungen ist die Verwendung von *NLP*-Verfahren (*Natural Language Processing*) notwendig. Dazu werden für diese Arbeit Wortvektoren benutzt. Die Herausforderung dieser Verknüpfungsfindung ergibt sich aus der möglichen Komplexität und Informalität eines Dokumentationstextes. Es ist anzunehmen, dass Satzteile, die für den Menschen als Modellentitäten erkennbar sind, nicht immer ein Zeichenketten-gleiches oder -ähnliches Pendant im Modell haben. Auch Zeichenketten-ähnliche Entitäten können semantisch doch unterschiedlich sein. Hat man beispielsweise, wie in Abbildung 1.1 zu sehen ist, die Entitäten *HttpServletRequest*, *WebApiServlet* und *WebPageServlet*, so ist für den Menschen schnell ersichtlich, was in einem Dokumentationstext mit *servlet* gemeint ist, wie in diesem Fall *WebApiServlet* oder *WebPageServlet*. Ein Zeichenketten-Ahnlichkeitsvergleich

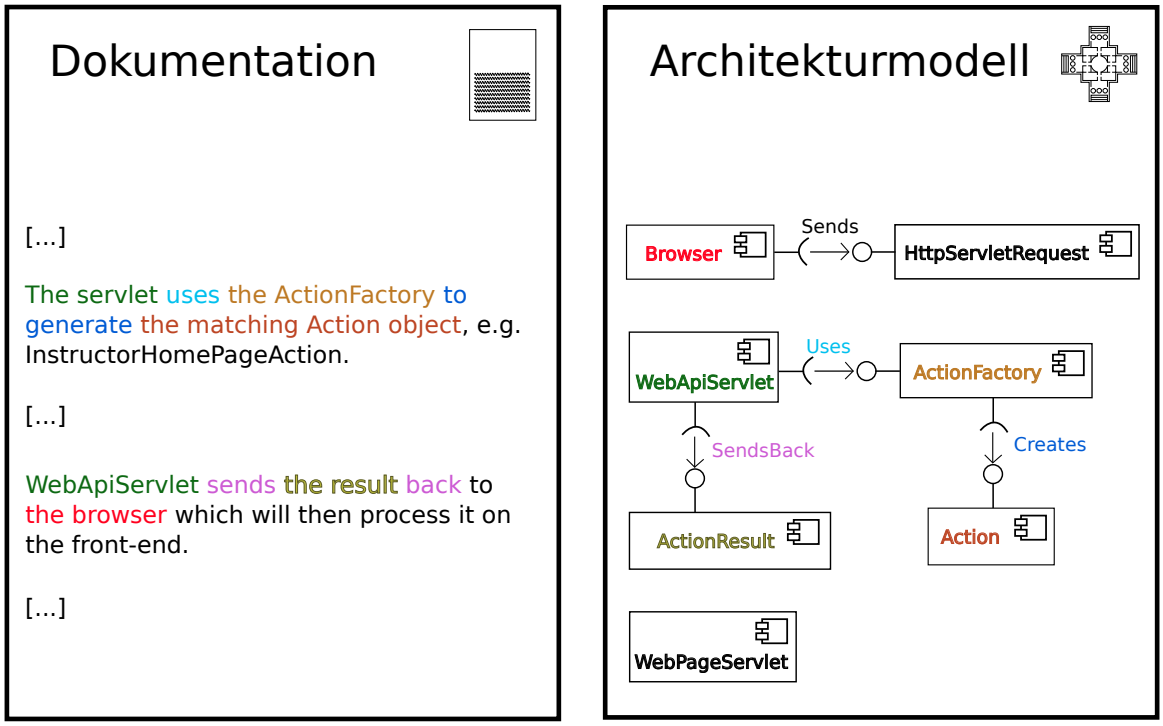

Abbildung 1.1.: Beispiel für Zusammenhänge zwischen Dokumentationstext und Architekturmodell.

würde diese Eindeutigkeit nicht abbilden und *HttpServletRequest* eine ähnlich hohe Ähnlichkeit wie den beiden anderen Entitäten zusprechen.

Um eine bessere Zuordnung erzielen zu können, braucht man also Vergleiche, die nicht nur die Struktur, sondern auch eine semantische Dimension des Wortes berücksichtigen. In dieser Arbeit soll erforscht werden, ob die Repräsentation dieser Dimension durch sogenannte *Wortvektoren* ermöglichbar ist. Wortvektoren erhalten semantische Informationen dadurch, dass in ihnen Kontextinformationen enthalten sind. Sie haben beeindruckende Eigenschaften: Es lassen sich Versätze zwischen Wörtern im Vektorraum finden, die darauf hinweisen, dass Vektoroperationen wie beispielsweise Addition, Subtraktion sowie Abstandsfunktionen mit diesen Vektoren sinnvoll einsetzbar sein könnten. Mikolov et al. zeigen mit ihrem Modell, dass mit gut trainierten Wortvektoren beispielsweise folgendes gilt: Zieht man vom Vektor des Wortes *biggest* den Vektor von *big* ab und addiert den Vektor von *small*, so erhält man einen Vektor, der Vektor des Wortes *smallest* am ähnlichsten ist [14]. Die erste Subtraktion beschreibt also eine Kodierung für einen Offset zwischen den Verb-Steigerungsstufen *Positiv* und *Komparativ*.

Der Beitrag dieser Arbeit ist es, dass Dokumentationstexte und Architekturmodelle insofern miteinander verbunden werden, dass Verknüpfungen zwischen Entitäten beider Artefakte erkannt und aus- bzw. weitergegeben werden. Hierzu wurde im Kontext der oben vorgestellten Vision von [12] ein Ansatz entwickelt, in welchem die Verknüpfungen berechnet werden.

Die Arbeit ist wie folgt strukturiert: In Kapitel 2 werden die für die Arbeit relevanten Verständnisgrundlagen dargeboten. Kapitel 3 widmet sich den verwandten Arbeiten. In Kapitel 4 wird der Entwurf der Herangehensweise an die Verknüpfungsberechnung beschrieben. Kapitel 5 beschreibt die Implementierung des im vorangehenden Kapitel beschriebenen Ansatzes. In Kapitel Kapitel 6 wird die Evaluation der Implementierung des Ansatzes beschrieben. Das geschieht mithilfe der *Goal-Question-Metric*-Evaluationsmethode. Das letzte Kapitel (Kapitel 7) fasst die Arbeit noch einmal kurz zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2. Grundlagen

Im Folgenden werden die für die Arbeit und deren Kontext relevanten Verfahren kurz dargelegt. Dabei wird zunächst auf Softwarearchitektur-Modelle (Abschnitt 2.1) eingegangen. Anschließend wird in Abschnitt 2.2 das Konzept von Wortvektoren und verschiedene Arten derselben erklärt. Darauf folgt Abschnitt 2.3, der sich mit grammatikalischen Grundlagen beschäftigt. Im nächsten Abschnitt (**??**) werden verwendete Rahmenwerke und im letzten Abschnitt (2.6) die Levenshtein-Distanz kurz erklärt.

2.1. Softwarearchitektur-Modelle

Eine Softwarearchitektur wird in [22] wie folgt definiert:

> A software architecture is the result of a set of design decisions relating to the structure of a system with components and their relationships as well as their mapping to execution environments.

Ein Softwarearchitekturmodell enthält demnach die Information über die Struktur eines Softwaresystems, also den Zusammenhang der existierenden Komponenten. Im Unterschied zum Programmcode, aus dem Architekturinformationen ebenfalls ersichtlich sind, ist ein Softwarearchitekturmodell dazu gedacht, explizit Entwurfsentscheidungen der Architektur zu dokumentieren.

Zur Erstellung solcher Softwarearchitekturmodelle gibt es sogenannte Metamodelle. Diese beschreiben die Beschaffenheit der Architekturmodelle. Bekannte Beispiele solcher Metamodelle sind die *Unified Modeling Language (UML)* [24] und das *Palladio Component Model (PCM)* [23]. Elemente der Architekturmodelle werden auch *Instanzen* von Elementen des Metamodells genannt, in Anlehnung an das Prinzip der Objektorientierung [22].

Das Palladio Component Model, welches für die Implementierung und Evaluation dieser Arbeit verwendet wird, zeichnet sich dadurch aus, dass es in verschiedene *view points* und *view types* unterteilt ist. View points beschreiben hier das Interesse, das man an einem System haben kann und sind bei Palladio in *structural*, *behavioral*, *deployment* und *decision* unterteilt. Jedem dieser view points lassen sich nun view types zuordnen, die einen Teil des Metamodells repräsentieren. Im Beispiel des view points *structural* wären dies der *repository view type* und der *assembly view type*. Repositories „beinhalten Datentypen, Interfaces und Komponenten“, während assembly views „die innere Struktur einer zusammengesetzten Einheit spezifizieren“ [22].

2.2. Wortvektoren

Zelling S. Harris stellte 1954 [9] die Hypothese auf, dass jede Sprache vollständig durch eine *Verteilungsstruktur* (engl. *distributional structure*) beschrieben werden kann. In dieser Hypothese wird jedem Element der Sprache eine *Verteilung* zugeordnet, die als die Summe der *Umgebungen* des Elements definiert wird. Eine Umgebung umfasst den konkreten Kontext eines Elements w, also die Menge der w umgebenden anderen Elemente inklusive ihrer Positionen (relativ zu w).

Diese Hypothese machen sich einige Ansätze zunutze, deren Gemeinsamkeit es ist, dass sie, vor allem mit Verfahren des maschinellen Lernens (*ML*), eine Abbildung von Wörtern zu Vektoren berechnen. Die entstehenden *Wortvektoren* stellen eine kontextuelle Nähe der Wörter in den Trainingstexten durch eine Nähe im entstehenden Vektorraum dar. Die verwendeten ML-Verfahren lernen die Abbildung von Wörtern zu Wortvektoren implizit durch das Training eines Neuronalen Netzes [14, 1], welches in seiner eigentlichen Aufgabe zur Kontext- oder Wort-Vorhersage gedacht ist. Mikolov et al. [13] verwenden zum Beispiel ein Modell, das zur Prädiktion des nächsten Wortes verwendet wird. Die Wortvektorberechnung ist dabei nur ein Zwischenschritt, welcher eine „One-Hot"-Abbildung, bei der jedem Wort eine eigene Dimension im Vektorraum zugeordnet wird, vermeidet, indem die Dimensionalität auf einen Wert reduziert wird, der wesentlich kleiner als das verwendete Vokabular ist. In diesem Beispiel wurden bei einem Vokabular mit 320 Millionen Wörtern Vektorräume der Dimension 80, 320 und 640 erzeugt.

Mit *word2vec* [14] wurden 2013 von Mikolov et al. zwei Modelle zur Erzeugung von Vektorrepräsentationen vorgestellt, welche ein schnelles Lernen „hochwertiger" Wortvektoren basierend auf sehr großen Datensätzen („Milliarden von Wörtern") und Vokabularen („Millionen von Wörtern") ermöglichen sollen. Die *CBOW-Architektur* prädiziert Worte aufgrund ihrer lokalen Umgebung und das *Skip-gram-Modell* prädiziert die Umgebung eines gegebenen Wortes. Dies geschieht jeweils anhand der Wortvektoren. Diese Verfahren ermöglichen es, verschiedene Arten linguistischer Wortähnlichkeit im Vektorraum wiederzufinden, wie in Abbildung 2.1 beispielhaft zu sehen ist. Auf der linken Seite sieht man, dass sich im Vektorraum übertragbare Offsets finden lassen. Im Beispiel kodiert der Offset die Geschlechtsbeziehung. Auf der rechten Seite der Abbildung wird deutlich, im Vektorraum mehrere Beziehungen kodiert werden können. In diesem Fall wäre das Plural–Singular und Männlich–Weiblich.

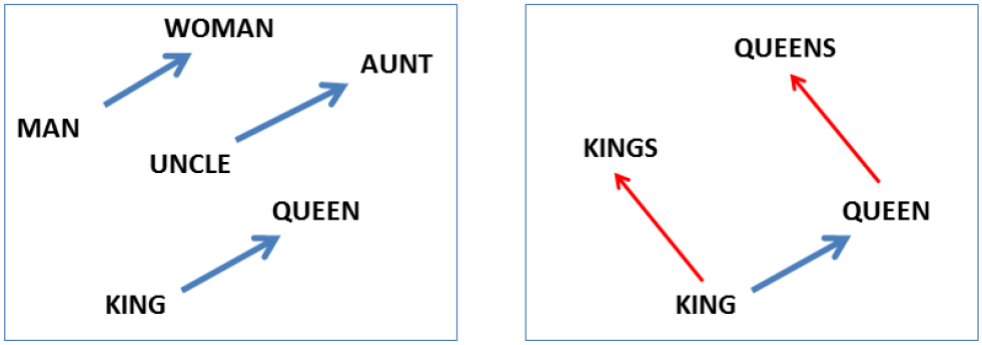

Abbildung 2.1.: Übertragbare Offsets von Wortvektoren. Quelle: [13]

Das Wortvektor-Modell von facebook, *fastText* [1], wurde 2017 um die Funktionalität erweitert, nicht nur das ganze Wort, sondern auch seine Zusammensetzung, in Form seiner *N-Gramm-Repräsentation*, zur Berechnung des Wortvektors zu nutzen. Die N-Gramm-Repräsentation ist eine Zerlegung eines Textes und in diesem Fall insbesondere eines Wortes in N Teile, die *Subwords* genannt werden.

Im Unterschied zu den Verfahren, die ein festes Mapping von Wort zu Vektor erzeugen, gibt es Wortvektoren, die auch bei der Anwendung des Modells einen Kontext erwarten und diesen mit in die Berechnung des Wortvektors einfließen lassen. Diese Modelle erzielen bessere Ergebnisse in NLP-Problemen wie Sprachverständnis-Aufgaben [4]. Allerdings werden die Anwendungsmöglichkeiten auf Fälle eingeschränkt, in denen ein sprachlicher Kontext vorliegt.

2.2.1. Kosinus-Ähnlichkeit bei Wortvektor-Vergleichen

Die im verwendeten *fastText*-Modell enthaltenen Vektoren sind reelle Zahlen, die durch Fließkommazahlen dargestellt werden. Das konkrete Modell hat dabei eine Vektorraum-Dimension von 300. Für Vektoren reeller Zahlen gibt es verschiedenste Metriken. Die bekannteste ist wohl der *euklidische Abstand*, der auf den quadrierten Differenzen der Vektorwerte in den einzelnen Dimensionen basiert. Es werden also hierbei Punkte verglichen. Die Kosinus-Ähnlichkeit ist eine Metrik, die im Gegensatz dazu die Richtung der Vektoren vergleicht. Das geschieht, indem der Kosinus des Winkels zwischen den Vektoren gebildet wird. Dabei wird auf das Standardskalarprodukt zurückgegriffen. Seien a, b Vektoren, $|\cdot|$ die euklidische Norm und ϕ der Winkel zwischen a und b. Dann beschreibt folgende Formel das Standardskalarprodukt:

$$a \cdot b = |a||b|cos(\phi)$$

Dies ergibt, nach $cos(\phi)$ umgestellt, die Formel für die Kosinus-Ähnlichkeit:

$$Kosinus\text{-}\ddot{A}hnlichkeit(a, b) = cos(\phi) = \frac{a \cdot b}{|a||b|}$$

Die Kosinus-Ähnlichkeit ist durch den Kosinus auf das Intervall $[-1, 1]$ beschränkt. Durch diese Beschränktheit kann ein Schwellenwert festgelegt werden, ab welchem ein Paar an Vektoren a, b als ähnlich gelten kann.

2.3. Grammatik

In diesem Abschnitt werden grammatikalische Grundlagen dieser Arbeit erklärt. Dabei wird das Konzept der Phrasen und Komposita erklärt.

Phrasentypen

In dieser Arbeit werden zwei Formen von Phrasen verwendet. Daher werden diese hier kurz erläutert. Eine Phrase ist ein „aus mehreren, eine Einheit bildenden Wörtern, auch aus einem einzelnen Wort bestehender Satzteil“ [5]. Zwei Spezialfälle davon sind die im

Folgenden erklärten Nominal- und Verbalphrasen. Daneben gibt es beispielsweise noch Präpositionalphrasen, Adjektivphrasen, Adverbphrasen, welche im Kontext dieser Arbeit allerdings keine Rolle spielen. In Abbildung 2.2 ist ein beispielhafter Satz mit markierten Nominal- und Verbalphrasen zu sehen. Auch enthält er eine Präpositionalphrase.

Nominalphrasen (NP) sind ein Überbegriff für alle Phrasen, die ein Substantiv als Kopf der Phrase besitzen. Alle Substantive sind in einer Phrase dieser Kategorie enthalten.

Verbalphrasen (VP) sind den Nominalphrasen ähnlich. Hierbei ist das Kernglied der Phrase ein Verb. Alle Verben lassen sich in Verbalphrasen einordnen.

Abbildung 2.2.: Ein beispielhafter Satz mit markierten Nominal-, Verbal und Präpositionalphrasen (NP, VP und PP).

Komposita
Ein Kompositum bezeichnet ein zusammengesetztes Wort. Dabei ist im Rahmen dieser Arbeit speziell das Nominalkompositum mit Benutzung von Binnenmajuskeln relevant. Hierbei werden Nomen zusammengesetzt, ohne dass die hinteren Nomen kleingeschrieben werden, was sich insbesondere in Softwaremodellen sehr häufig vorfindet. Diese Schreibweise wird auch *camel case* genannt. Ein nichttriviales Beispiel für ein Binnenmajuskel-Nominalkompositum mit mehreren Großbuchstaben hintereinander wäre *UserDBAdapter*.

2.4. Sprachverarbeitung mit PARSE

In diesem Abschnitt wird das für die Sprachverarbeitung verwendete Rahmenwert *PARSE* erklärt.

PARSE (Programming ARchitecture for Spoken Explanations) ist ein Projekt, welches für das Programmieren in natürlicher Sprache entwickelt wurde. Die Idee dahinter ist, dass viele Menschen Rechner benutzen, aber wenige sie auch programmieren können. Diese Benutzer wissen durchaus, was sie von Software erwarten, nur fehlt ihnen die Sprache, dies dem Rechner mitteilen zu können. Diese Lücke zu schließen, ist die Vision des Projekts *Programmieren in natürlicher Sprache* [20]. Im Rahmen dessen bietet PARSE einen agentenbasierten Ansatz zur Verarbeitung gesprochener natürlicher Sprache (vgl. Abbildung 2.3). Diese Sprach-Blöcke müssen zunächst in Text umgewandelt werden. Anschließend wird dieser Text zergliedert, wobei ein Graph aufgebaut wird. Darauf agierend können Agenten den erstellten Graphen modifizieren. In diesen grundsätzlichen Aufbau integriert sich das in dieser Arbeit entwickelte Verfahren, indem ein solcher Agent bereitgestellt wird.

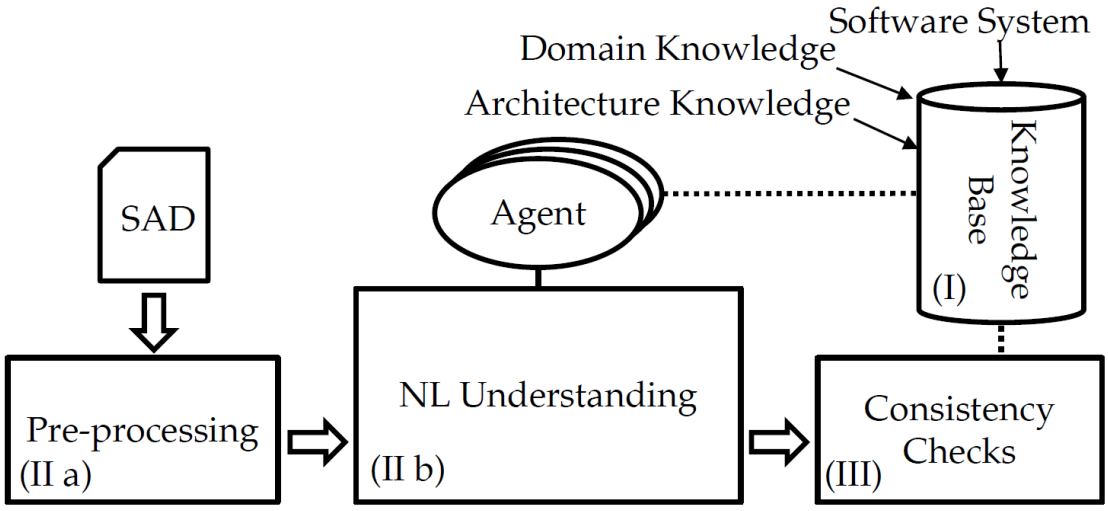

Abbildung 2.3.: Übersicht des Ansatzes aus [12]: Verwendung des PARSE-Ansatzes für Software-Architekturdokumente (*SAD*) und Software-Architekturmodelle (*Architecture Knowledge*)

Wortarten

PARSE enthält einen Zergliederer, welcher Worte und Phrasen klassifiziert. Davon treten in dieser Arbeit die folgenden Phrasentypen auf:

Abkürzung	**Erklärung**
NP	Nominalphrasen
VP	Verbalphrasen
NN	Nomen
NNP	Nomen im Singular
VB	Verben
VBZ	Verben im Singular, konjugiert in der dritten Person.
JJ	Adjektive
DT	Determinativa. Diese sind für die nähere Bestimmung von Nomen zuständig [3].
PP	Präpositionen
CC	Koordinierende Konjunktionen
TO	Das Wort „to“

2.5. Ontologien

Ontologien sind Repräsentationen von Informationen, welche mittels Relationen miteinander verbunden sind. Dargestellt werden die Informationen als Mengen von Konzepten, Relationen, Attributen und Datentypen und Regeln, welche diese Mengen miteinander verbinden. Damit sind Ontologien mächtige Werkzeuge für den Zugriff auf Wissensrepräsentationen unterschiedlichster Art. Diese relative Allgemeinheit in der Definition ermöglicht es, auch Modelle wie das Palladio-Komponentenmodell damit darzustellen.

Das in dieser Arbeit verwendete Rahmenwerk, *Apache Jena*, kann einen programmatischen Zugriff auf Ontologien darstellen, welche mit Hilfe der *Web Ontology Language (OWL)* definiert wurden.

2.6. Levenshtein-Distanz

Die Levenshtein-Distanz wird als Vergleichsverfahren für die Evaluation benutzt. Sie ist bestimmt durch die minimale Anzahl an Operationen, die benötigt wird, um eine Zeichenkette w_1 in eine andere Zeichenkette w_2 umzuwandeln. Operationen können dabei das Löschen, das Einfügen oder das Ersetzen eines einzelnen Zeichens sein. Da diese Distanz schlecht mit einem Schwellenwert verrechenbar ist, wurde sie für diese Arbeit so abgewandelt, dass sie immer im Intervall $[0, 1]$ liegt. Diese Abwandlung geschieht wie folgt:

$$\mathit{Levenshtein\text{-}Ähnlichkeit}(w_1, w_2) = 1 - \frac{\mathit{Levenshtein\text{-}Distanz}(w_1, w_2)}{|w_1|}$$

Als Beispiel seien $w_1 = \mathit{UserDatabaseAdapter}, w_2 = \mathit{UserDBAdapter}$. Um von UserDatabaseAdapter zu UserDBAdapter zu kommen, muss „atabase“ gelöscht, und ein „B“ dafür eingefügt werden. Das wären 8 Operationen. Stattdessen kann allerdings auch beispielsweise der erste Buchstabe von „atabase“ durch „B“ ersetzt werden. Damit wird eine Operation gespart. Die (absolute) Levenshtein-Distanz liegt bei 7. Dementsprechend liegt die (relative) Levenshtein-Ähnlichkeit bei $1 - \frac{7}{|w_1|} = 1 - \frac{7}{19} \approx 0.63$.

3. Verwandte Arbeiten

Im diesem Kapitel werden andere Arbeiten besprochen, die mit dieser Arbeit thematisch verwandt sind. Dabei geht es um Anwendungen von Wortvektoren und verwandte Arbeiten bei der Problemstellung, Wörter ähnlicher oder gleicher Bedeutung zu finden.

Wortvektoren werden für Aufgaben wie Klassifikation der Stimmung (engl. *sentiment*) von Tweets [25], personifizierte Empfehlungen [17] oder dem Zergliedern natürlichsprachlicher Texte [2] genutzt. Im softwaretechnischen Kontext wurde die Möglichkeit der Verwendung von Wortvektoren zur Erkennung semantischer Ähnlichkeit zwischen Code-Dokumenten (unter anderem zur Fehlererkennung) gezeigt [32]. Ye et al. verwenden hierbei das Prinzip der Wortvektoren, um eine semantische Beziehung zwischen natürlichsprachlichen Begriffen und „Codeausschnitten“ zu finden: Sie trainieren Wortvektoren mit einem *Skip-gram*-Modell, wobei die Datenbasis eine Mischung von Dokumentations- und Code-Texten ist. Das Skip-gram-Modell benutzt bei Wörtern oder Code-Tokens, die in ähnlicher Form (z.B. nur Unterschiede in Groß- und Kleinschreibung) auch im jeweils anderen Dokument vorkommen, auch den jeweils anderen Kontext zum Training des Wortvektors. So wird eine Ähnlichkeit der Wortvektoren herbeigeführt.

Tian et al. haben mit SEWordSim 2014 ein Verfahren vorgestellt, welches ein *Wortnetz* bereitstellt, das automatisch aus StackOverflow-Informationen generiert wurde [29] und beschreiben, dass sich das viel verwendete Wortnetz *WordNet* allgemeinen Texten bedient, worin sie ein Problem sehen, da Software-spezifische Wortbedeutungen oft nicht erfasst werden können.

Bhingardive et al. präsentierten 2014 einen Wortvektor-basierten Ansatz zur Vergröberung eines Wort-Netzes durch Vereinigung von sogenannten *senses*. Dabei konnten sie eine „signifikante Verbesserung gegenüber WordNet-basierten Ansätzen“ erzielen.

Ontologie-Verschmelzung

Im Bereich der Verschmelzung von Ontlogien (engl. *ontology merging*) geht es um das Vereinigen oder Verschmelzen unterschiedlicher Ontologien zum Zwecke der Kombination verschiedener Wissensrepräsentationen. Dabei bedarf es auch Methoden der Sprachverarbeitung, um linguistische Ähnlichkeit von Konzepten bestimmen zu können. Diese linguistische Ähnlichkeitsberechnung ist genau das Kernproblem, welches in dieser Arbeit bearbeitet wird. Die Autoren von PROMPT [16] halten sich die Verwendung eines konkreten Verfahrens offen, Stumme und Maedche [26, 27], die sich ebenfalls mit Ontologie-Verschmelzung beschäftigten, verwenden dafür das Verfahren SMES [15], welches unter anderem einen auf auf regulären Ausdrücken basierten Zerteiler und ein Lexikon für die Sprachverarbeitung benutzt. In beiden Publikationen wird mithilfe von SMES aus zwei zu verschmelzenden Ontologien und „relevanten“ natürlichsprachlichen Dokumenten *formale Kontexte* über die genannten Dokumente erstellt. Ein formaler Kontext beschreibt

in diesem Fall ein Tripel aus Dokumenten, Attributen und einer Relation, wobei die Menge der Attribute aus den Konzepten der betrachteten Ontologie besteht und ein Paar (*doc*, *concept*) dann in der Relation enthalten ist, wenn eine Instanz von *concept* in *doc* enthalten ist. Letztere Relation wird mithilfe von SMES [15] berechnet. 2011 stellten Chen et al. einen Ansatz zur Ontologie-Verschmelzung vor, welcher auf WordNet und *FFCA (Fuzzy Formal Concept Analysis)* basiert.

Anforderungs-Rückverfolgbarkeit

Anforderungs-Rückverfolgbarkeit (engl. *Requirements Traceability*) ist ein Bereich, der dem Problem der Zuordenbarkeit von Dokumentationstext-Entitäten zu Architekturmodell-Entitäten sehr nahe steht. Es geht dabei um die Möglichkeit, "das Leben einer Anforderung zu beschreiben, sowohl vorwärts als auch rückwärts"[7], also um die Betrachtung dieser Anforderung in jeder Phase oder Iteration eines Entwicklungsprozesses. Insbesondere geht es demnach auch darum, Anforderungen in beliebigen Artefakten im Entwicklungsprozess wiederzufinden. Rempel und Mäder stellten 2016 in einer empirischen Studie [21] über 24 „mittelgroßen bis große Open-Source-Projekte“ fest, dass drei der vier untersuchten „Anforderungs-Rückverfolgbarkeits-Aktivitäten“ die Softwarequalität positiv beeinflussen. Es wurde ein Zusammenhang zwischen der Vollständigkeit der Rückverfolgbarkeit und der Defekt-Rate gemessen.

In diesem Bereich gibt es auch Ansätze, in denen Wortvektoren genutzt werden. Guo, Cheng und Cleland-Huang stellten 2017 ein Modell für die Anforderungs-Verfolgbarkeit vor, welches auf den eigenen Daten (ä large industrial dataset") state-of-the-art-Methoden (das *Vector Space Model* und *Latent Semantic Indexing*) in den Evaluationsmetriken *MAP*, *precision* und *recall* übertrifft [8].

4. Entwurf

In dieser Arbeit wird ein Verfahren vorgestellt, welches mithilfe von Wortvektoren Verknüpfungen zwischen Entitäten von Softwarearchitektur-Modellen und Dokumentationstexten dieser Architekturen findet.

Da Software in vielen unterschiedlichen Bereichen entwickelt und eingesetzt wird, sind für das verwendete Wortvektormodell nicht nur softwarebezogene, sondern auch allgemeine Trainingsdaten wichtig. Weiterhin erleichtert es die Entwicklung des Verfahrens, wenn jedem zu verarbeitenden Wort ein Wortvektor zugewiesen werden kann. Dies ist bei Verfahren wie *word2vec* [14] und *GloVe* [19] nicht der Fall. Diese können nur Wörter verarbeiten, die schon im Trainingsdatensatz vorkamen. Das Verfahren *fastText* [1] erfüllt dank der Verarbeitung von Sub-Wort-Information (von engl. *subword information*) diese Anforderung. Da es von diesem Wortvektor-Verfahren vortrainierte Modelle gibt, wurde es zur Entwicklung und Evaluation des in dieser Arbeit entwickelten Verfahrens ausgewählt. Konkret wird ein auf der gesamten englischsprachigen Wikipedia trainiertes *fastText*-Modell verwendet.

Das in dieser Arbeit entwickelte Verfahren ist in den Entwurf von [12], und damit in das Konzept von PARSE [30] eingebettet. Dazu wurde ein Agent entwickelt, der auf den erstellten Graphen zugreift und in diesem Verknüpfungen zu Softwarearchitekturmodell-Entitäten annotiert. Das Architekturmodell wird vom Agenten selbst in Form einer OWL-Ontologie geladen.

Der Agent führt, sobald er mit diesen Informationen initialisiert ist und die Ausführung angestoßen wird, die Berechnung der Verknüpfungen durch, was *(II b) in Abbildung 2.3* entspricht. Dabei geht die Verknüpfungsberechnung vom Dokumentationstext aus. Im Folgenden näher beschriebene Elemente dieses Textes werden verarbeitet und mithilfe einer Wortvektor-Repräsentation mit Komponenten aus dem Softwarearchitekturmodell anhand eines Ähnlichkeitsmaßes verglichen. Falls der Wert für die nach diesem Maß ähnlichste Verknüpfung einen Schwellenwert überschreitet, wird diese zur Annotation ausgewählt. In Abbildung 4.2 wird ein mit Modellentitäten verknüpfter Satz gezeigt.

Es wird zwischen zwei Arten von Verknüpfungen unterschieden, die sich nach der grammatikalischen Klassifikation durch PARSE richten (vgl. Abbildung 4.3): *Nominalphrasen* sind diejenige grammatikalische Gruppe, die mit nicht-relationalen Modellkomponenten verknüpft werden können. *Verbalphrasen* werden im Gegensatz zu Nominalphrasen den relationalen Modellkomponenten zugeordnet, also den Verbindungen zwischen nicht-relationalen Modellkomponenten.

Der Ablauf der Verknüpfungsberechnung orientiert sich an diesen zwei Verknüpfungsarten. Zunächst werden einzelne Nomen innerhalb von Nominalphrasen und Nominalphrasen als Einheit verarbeitet. Das wären im Beispiel Abbildung 4.3 die mit *NNP* oder *NN* markierten Wörter. Anschließend werden Verknüpfungen für ganze Nominalphrasen, wie im Beispiel Abbildung 4.3 „the matching Action object“, berechnet. Die aus diesen beiden

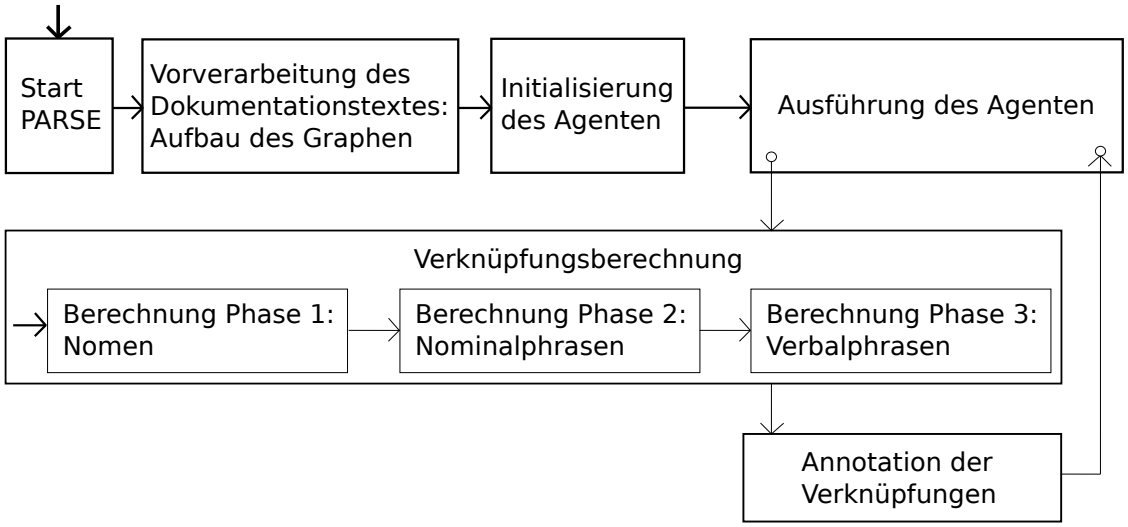

Abbildung 4.1.: Ablauf des Vorgangs zur Berechnung und Annotation von Verknüpfungen.

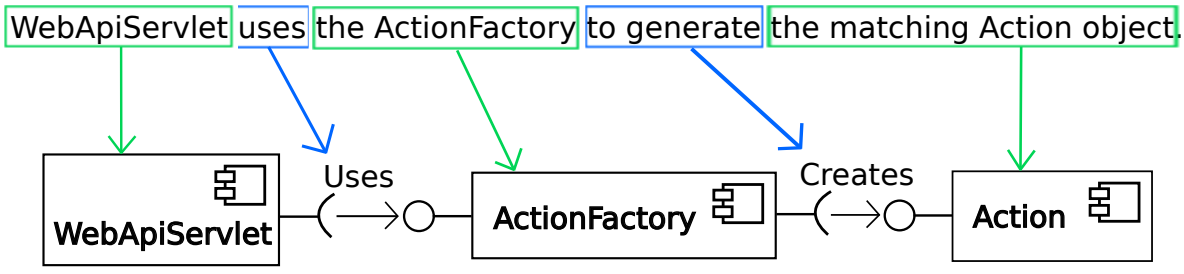

Abbildung 4.2.: Verknüpfung einiger Phrasen eines beispielhaften Satzes aus TEAMMATES mit entsprechenden Modellentitäten.

Phasen entstehenden Verknüpfungen werden für die Berechnung von Verbalphrasen-Verknüpfungen benutzt.

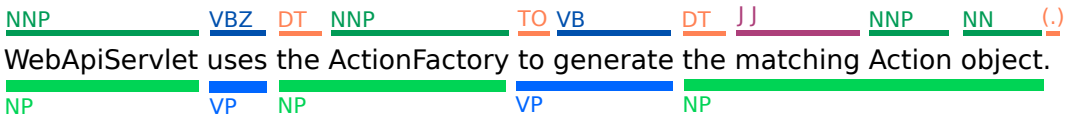

Abbildung 4.3.: Ein beispielhafter Satz aus TEAMMATES, zergliedert mit PARSE. Unten: Phrasentypen. Oben: Wort/ Zeichen-Typen

Nachdem die Verknüpfungsberechnung abgeschlossen ist, werden die gefundenen Verknüpfungen an den PARSE-Graphen annotiert.

Im weiteren Verlauf dieses Kapitels werden die einzelnen Phasen der Verknüpfungsberechnung, Nomen-, Nominalphrasen- und Verbalphrasen-Verarbeitung, im Detail erläutert. Dabei werden auch Konzepte erklärt, die evaluiert wurden, aber aufgrund schlechteren Abschneidens nicht im finalen Verfahren enthalten sind. Im letzten Abschnitt wird dieses anhand der Parametrisierungen der einzelnen Phasen beschrieben.

4.1. Phase 1: Verarbeitung einzelner Nomen

In der ersten Phase soll es um die Verarbeitung einzelner Nomen, genauer einzelner Wörter, gehen. Ein besonderes Augenmerk wird hierbei auf Software-typische Binnenmajuskel-Komposita wie *WebApiServlet* oder *UserDBAdapter* gelegt. Im ersten Unterabschnitt wird zunächst der Vergleich im Falle von Nicht-Komposita, also Wörtern, die nicht aufzuteilen sind, erklärt. Der zweite Unterabschnitt beschreibt, wie Komposita verarbeitet werden, um sie möglichst gut vergleichen zu können.

Um möglichst viele der korrekten Verknüpfungen zu finden, wird für jedes Nomen jeder Nominalphrase über alle nicht-relationalen Modellentitäten iteriert. Sei (*noun*, *component*) ein beliebiges Vergleichspaar. Dann wird für *noun* und *component* jeweils ein Wortvektor gebildet. Diese werden anhand der in Unterabschnitt 2.2.1 erklärten Kosinus-Ähnlichkeit miteinander verglichen. Liegt diese Kosinus-Ähnlichkeit unter dem Schwellenwert, wird das Paar verworfen. Andernfalls wird es zu den gefundenen Verknüpfungen gezählt. Nach Berechnung aller Verknüpfungen werden potenziell aufgetretene Zweideutigkeiten (*noun*, $component_0$), (*noun*, $component_1$), ... aufgelöst, indem das Paar mit der höchsten Kosinus-Ähnlichkeit ausgewählt wird. Diesem Vorgehen liegt die Annahme zugrunde, dass mit einem Nomen im Dokumentationstext höchstens eine Modellentität referenziert wird.

Bei der Berechnung der Wortvektoren wird zwischen Komposita und Nicht-Komposita, also Wörtern, die nicht aufzuteilen sind, unterschieden. Dabei ist es unerheblich, ob das betrachtete Wort nun ein Nomen aus dem Text, oder ein Bezeichner einer Modellentität ist.

4.1.1. Nicht-Zusammengesetzte Nomen

Wörter w, die nicht aufzuteilen sind, können einfach verarbeitet werden: Hier wird w an die Schnittstelle zu *fastText* übergeben, welche einen Wortvektor zurückliefert.

4.1.2. Zusammengesetzte Nomen

In Softwarearchitektur-Modellen tauchen oft Komposita auf, die den Zweck haben, verschiedene Eigenschaften der beschriebenen Entität in einem Wort zusammenzupacken. Das Wort HttpServletRequest beispielsweise beschreibt einen Request, der mittels HTTP (HyperText Transfer Protocol) an ein Servlet geschickt wird. Diese verschiedenen Eigenschaften gilt es zu separieren, um einen bezüglich der Güte der Verknüpfungsberechnung guten Wortvektor erstellen zu können. Dazu wird ein solches Kompositum zunächst anhand der folgenden Merkmale erkannt:

- Trennzeichen: _ oder -
- Binnenmajuskel (engl. *camel case*), z.B. HttpServletRequest
- Eine Mischform dieser Merkmale, z.B. HttpServlet_Request

Mit Hilfe dieser Merkmale wird ein als solches erkanntes zusammengesetztes Nomen in ein n-Tupel aufgespalten. Um aus diesen Tupeln einen einzelnen Wortvektor zu erzeugen, wurden vom Verfasser dieser Arbeit verschiedene *Vektorsummen* definiert:

Gewichtete Summe

Mithilfe einer gewichteten Summe kann der Einfluss einzelner Subworte auf den zu berechnenden Wortvektor gesteuert werden. Somit können für die Wortbedeutung wichtigere Subworte hervorgehoben und andere, weniger wichtige, zurückgestellt werden. Bei einer gewichteten Summe wird zunächst der Wortvektor der einzelnen Subworte gebildet. Diese werden dann mit Gewichten $w \in \mathbb{R}$ skalar multipliziert, addiert und normalisiert:

$$WeightedSum_{w_1,\ldots,w_n} = \frac{\sum_{i=1}^{n} w_i wordvector(subword_i)}{\| \sum_{i=1}^{n} w_i wordvector(subword_i) \|_2}$$

Durchschnitts-Summe

Die Durchschnits-Summe ist ein Spezialfall der gewichteten Summe. Hierbei wird den Subwörtern $(subword_1, \ldots, subword_n)$ jeweils das Gewicht $w = w_1 = \ldots = w_n = \frac{1}{n}$ zugewiesen.

Letztwort-Summe

Grundgedanke hinter dieser Summe ist, dass die Kernbedeutung eines Binnenmajuskel-Kompositums meist am Ende steht. Anhand von *HttpServletRequest* kann man das gut sehen. Dieses Kompositum beschreibt einen „Request“, welcher via *HTTP* zu einem *Servlet* weitergeleitet wird. Dieser Bedeutsamkeit des letzten Subwortes soll mithilfe der im Folgenden erklärten Letztwort-Summe ein entsprechendes Gewicht bei der Berechnung des Wortvektors des Kompositums beigemessen werden.

Diese Summe teilt die Subwörter $\{subword_1, ..., subword_n\}$ auf:
Für $\{subword_1, ..., subword_{n-1}\}$ wird die Durchschnitts-Summe berechnet. $subword_n$ wird, gewichtet mit Gewicht w hinzuaddiert. Das Resultat wird wieder normalisiert. So erhält man eine Summe, bei der der Wortvektor des letzten Subwortes $subword_n$, unabhängig von der Gesamtzahl der Subwörter, immer gleich viel Gewicht im resultierenden Vektor besitzt. Der Beitrag der einzelnen Subworte $subword_k \in \{subword_1, ..., subword_{n-1}\}$ zum resultierenden Vektor wird dementsprechend im Vergleich zu $subword_n$ bei gleichbleibendem w mit steigendem n kleiner.

Diese Summe konnte bei Tests die besten Ergebnisse erzielen und wird deshalb für die Verknüpfungsberechnung verwendet. Die Letztwort-Summe wird für die Verknüpfungsberechnung verwendet, da durch die Höher-Bewertung des letzten Subwortes ein besseres Abschneiden bezüglich der Verknüpfungsberechnung im Vergleich zur bloßen Durchschnittsberechnung zu erwarten ist.

4.1.3. Untervektorräume

Eine weitere Parametrisierungsmöglichkeit sind Untervektorräume. Hierbei wird die Berechnung der Kosinus-Ähnlichkeit nicht im gesamten, 300-dimensionalen Raum, sondern in einem Untervektorraum durchgeführt.In diesem sind $k > 0$ Dimensionen für alle Vektoren $\equiv 0$, was einer Abbildung auf einen $(300 - k)$-dimensionalen Raum gleichkommt. Die Idee dabei ist, dass in günstigen Untervektorräumen die semantische Ähnlichkeit oder Verschiedenheit bezüglich einer genauer bestimmten Menge an Eigenschaften besser bestimmt werden könnte.

Das in dieser Arbeit entworfene Verfahren zur Bestimmung eines solchen Untervektorraumes wird im Folgenden beschrieben. Seien $p, \epsilon \in [0, 1]$. Zunächst wird (von Hand) eine Menge an Wörtern W bestimmt, die bestimmte Gemeinsamkeiten haben. Im konkreten Fall wurden hierbei einige Begriffe der Softwaretechnik und andere Informatik-bezogene Begriffe verwendet. Diese sind im Anhang in Abschnitt A.1 zu finden. Danach wird für jede Dimension dim des Vektorraumes berechnet, ob diese im Untervektorraum landet, in diesem also nicht kongruent 0 ist. Dies ist genau dann der Fall, wenn für p Prozent der Wörter $word$ gilt, dass der Wert der dim-ten Dimension des Wortvektors von $word$ in einer ϵ-Umgebung um den Median liegt. So erhält man einen Untervektorraum U, in welchem vor allem die Dimensionen nicht null sind, in denen sich die Wortvektoren der Wörter aus W relativ ähnlich (abhängig von p und ϵ) sind. Im dazu komplementären Raum U^K, in dem genau die Dimensionen $\equiv 0$ sind, die in U nicht $\equiv 0$ sind, sind demnach die Dimensionen vertreten, in denen sich die gegebenen Wortvektoren relativ stark unterscheiden. Die Hypothese ist nun, dass Wortvektoren, deren Wörter semantisch etwas anderes meinen, in U^K besser unterscheidbar sein könnten, da die Gemeinsamkeiten der IT-Begrifflichkeiten in U^K weniger vertreten sind. Das würde eine Verbesserung der Verknüpfungsberechnung bedeuten.

4.2. Phase 2: Verarbeitung von Nominalphrasen

In der zweiten Phase werden ganze Nominalphrasen betrachtet. Der Unterschied zu den zusammengesetzten Nomen aus Phase 1 ist, dass innerhalb ganzer Nominalphrasen unterschiedlichere Wortarten auftreten, die den einzelnen Worten dank PARSE auch zugeordnet werden können. Es können somit also auch Syntaxinformationen verarbeitet werden. Es gibt in dieser Phase ein Verfahren, welches sich der Vektorsummen aus Abschnitt 4.1 bedient und ein neues Verfahren, das Punktwolkenverfahren. Diese beiden Verfahren werden am Ende dieses Kapitels dargestellt. Unabhängig davon gibt es allgemeine Parametrisierungsmöglichkeiten für die Berechnung der Wortvektoren: Eine Wortarten-Gewichtung, einen Wortarten-Filter und einen Wortfilter. Diese werden im Folgenden erklärt.

Wortarten-Gewichtung

Eine Gewichtung der unterschiedlichen Wortarten erlaubt es, manche Wortarten als mehr oder weniger wichtig bei der Berechnung des Wortvektors zu betrachten. Das ist insbesondere dann hilfreich, wenn man Wortarten (mithilfe des Filters) nicht gänzlich ausschließen will oder andere Wortarten als Kernelemente ansieht. Eine vielversprechende Gewichtung würde beispielsweise Nomen höher gewichten als andere Wortarten, da Nomen zum inhaltlichen Kern einer Nominalphrase gehören.

Filter

Der Wortarten- und der Wortfilter funktionieren relativ ähnlich. Hierbei kann eine Menge an Wörtern oder Wortarten angegeben werden, die nicht bei der Berechnung der Kosinus-Ähnlichkeit zwischen Nominalphrase und Entitätenbezeichner in Betracht gezogen werden sollen. Dies ist unabhängig davon, ob das Standard- oder das Punktwolkenverfahren gewählt wird. Der Zweck dieser Filter ist es, Worte oder Wortgruppen, die das Ergebnis des Vergleichs verfälschen bzw. verschlechtern würden, von vorneherein auszuschließen. In der Standardparametrisierung für den Wortartenfilter sind deshalb unter anderem Artikel, Pronomen, Satzzeichen und sonstige Wortarten enthalten, die nichts oder nur wenig zur Bedeutung der Nominalphrase beitragen und das Ergebnis der Berechnungen verschlechtern könnten. Der Wortfilter ist vor allem für Wörter gedacht, die zwar aufgrund ihrer Wortgruppe eine Relevanz zugeschrieben bekommen würden, womöglich sogar mit hohem Gewicht, die aber dennoch wenig zum Kerninhalt der Phrase beitragen. Ein Beispiel dafür wäre das Wort *component*, das häufig bei Palladio-Modellbeschreibungen vorkommt, aber meist nur die Art der Entität (eine Komponente, englisch *Component*, des Palladio-Component-Model) beschreibt, nicht jedoch die Komponente an sich. Bei solchen Wortfiltern besteht die Gefahr, damit auch Falsch-positive zu erzielen. Im oben genannten *component*-Beispiel wäre das eine Entität, die *component* oder ein inhaltlich ähnliches oder gleiches Wort im Namen trägt. Daher sollten diese, wenn überhaupt, sehr spärlich eingesetzt werden.

4.2.1. Vektorsummenverfahren

Ähnlich wie in Abschnitt 4.1, wird beim Vektorsummenverfahren dieser Phase für jede Nominalphrase jedes Satzes die Kosinus-Ähnlichkeit mit allen Nicht-relationalen

Architekturmodell-Entitäten berechnet. Falls es mindestens eine Verknüpfung gibt, deren berechnete Kosinus-Ähnlichkeit über dem Schwellenwert liegt, wird auch hier die beste ausgewählt. Für jedes Wort wird mithilfe einer Parametrisierung der ersten Phase ein Wortvektor erstellt. Allen Nicht-Nomen und Nicht-Komposita wird dabei direkt der Wortvektor zugewiesen. Die Wortvektoren der Komposita wie in Unterabschnitt 4.1.2 werden mit einer der Vektorsummen berechnet. Die entstehenden Wortvektoren werden nun mithilfe der gewichteten Summe auf einen einzigen Wortvektor abgebildet. Die Berechnung des Wortvektors der Modellentität erfolgt wie in Abschnitt 4.1.

4.2.2. Punktwolkenverfahren

Das Punktwolkenverfahren entstand aus dem Gedanken, dass bei (insbesondere längeren) Nominalphrasen eine Abbildung der einzelnen Wörter der Phrase auf einzelne Subwörter der Modellentität einen präziseren Vergleich ermöglichen könnte, als es mit reiner Summierung von jeweils Phrase und Modellentität möglich ist. Dass miteinander verknüpfbare Wörter von Nominalphrase und Modellentitäts-Bezeichner auch in unterschiedlicher Reihenfolge auftreten können, zeigt Abbildung 4.4.

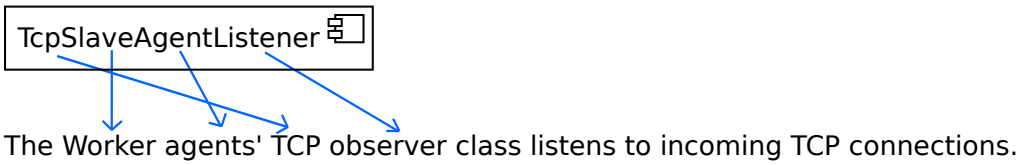

Abbildung 4.4.: Anwendungsbeispiel für das Punktwolkenverfahren: Miteinander verknüpfbare Einzelteile von Nominalphrase und Modellentitäts-Bezeichner. *TcpSlaveAgentListener* ist eine Klasse aus Jenkins [10].

Ähnlichkeitsmaß

Die Ähnlichkeit von Phrase und Modellentität wird anhand der Kosinus-Ähnlichkeiten der einzelnen Paare der gefundenen Abbildung von Phrasen-Wörtern auf Subwörter des Modellentitäts-Bezeichners berechnet.
Die Menge $M = \{(word_1, subword_1), \ldots, (word_n, subword_n)\}$ beschreibe diese Abbildung. $W = \{w_1, \ldots, w_n\}$ sei die Menge der Gewichte der Nominalphrasen-Wörter. Die Punktwolken-Ähnlichkeit berechnet sich dann als die gewichtete Summe der einzelnen Kosinus-Ähnlichkeiten der Paare aus M:

$$Punktwolken\text{-}Ähnlichkeit = \frac{\sum_{i=1}^{n} w_i * Kosinus\text{-}Ähnlichkeit(word_i, subword_i)}{\sum_{w \in W} w}$$

Vorbereitung

Um die Punktwolken-Ähnlichkeit, und damit die beste bijektive Abbildung zwischen zwei Punktwolken berechnen zu können, werden zwei gleich große Punktwolken benötigt. Das ist nicht immer der Fall. Im Beispiel Abbildung 4.4 hat die Modellkomponente *TcpSlaveAgentListener* vier Subwörter, während *„The worker agents' TCP observer class“* aus sechs,

nach Filterung der Artikel aus fünf Subwörtern besteht. Dieses Problem wird dadurch gelöst, dass der Algorithmus zur Berechnung der nach Punktwolkenähnlichkeit besten Abbildung mehrfach aufgerufen wird: Seien PW_1 und PW_2 Punktwolken mit $|PW_1| \geq |PW_2|$. Dann wird für jede Teilmenge $T \subseteq PW_1$ mit $|T| = |PW_2|$ der Algorithmus aufgerufen. Das unter den Teilmengen beste Ergebnis wird zurückgegeben.

Bei dieser Vorgehensweise können potenziell wichtige Informationen verloren gehen. Daher gibt es auch die (Parameter-abhängige) Möglichkeit, bei ungleich großen Punktwolken auf das in Unterabschnitt 4.2.1 beschriebene Vektorsummenverfahren zurückzufallen, sodass das Punktwolkenverfahren nur für gleich große Punktwolken durchgeführt wird.

Berechnung der Abbildung

Die nach dem oben definierten Punktwolken-Ähnlichkeitsmaß beste Abbildung der Phrasenpunktwolke auf die Modellentitäts-Subwort-Punktwolke wird anhand eines rekursiven Algorithmus ermittelt. Dieser wird im Folgenden anhand des in Abbildung 4.5 zu sehenden Pseudocodes erklärt.

Bei diesem Algorithmus wird für jeden Vektor der ersten Punktwolke ($vectors_1$) zunächst der (nach Kosinus-Ähnlichkeit) nächste Vektor aus der anderen Punktwolke ($vectors_2$) bestimmt. Unter Annahme dieser Paarung werden rekursiv die anderen bestmöglichen Paare ausgewählt. Es wird also jeder Vektor „festgehalten“ und dafür das bestmögliche Resultat berechnet. Nach der Schleife wird diejenige Menge an Paarungen zurückgegeben, welche die höchste Punktwolken-Ähnlichkeit aufweist. Im Basisfall, in dem beide Punktwolken eine Kardinalität von 1 aufweisen, wird das einzigmögliche Paar zurückgegeben.

4.3. Phase 3: Verarbeitung von Verbalphrasen

Bei der Berechnung der Verbalphrasen-Verknüpfungen unterscheidet sich das grundsätzliche Vorgehen von den zwei vorherigen Phasen. Da Relationsnamen wie beispielsweise *Creates*, *Uses* oder *SendsBack* nicht zwingend eindeutig sind, wird zur Berechnung der Verknüpfungen zwischen relationalen Entitäten und Verben der Kontext, in dem das Verb steht, benötigt. Genauer gesagt kann eine Verknüpfung eines Verbs und einer relationalen Entität nur dann gefunden werden, wenn für das dem Verb in seiner Funktion als Prädikat zuzuordnende Subjekt und für das entsprechende Objekt Verknüpfungen zu nicht-relationalen Modellentitäten existieren. Für die erfolgreiche Berechnung einer Verb–Relationalentität-Verknüpfung müssen diese bereits korrekt erkannt sein. An diesen Vorgaben orientiert sich das Vorgehen bei der Verknüpfungsberechnung. Im Folgenden (Unterabschnitt 4.3.1) wird zunächst das generelle Vorgehen erklärt. Anschließend wird in Unterabschnitt 4.3.2 auf die Parametrisierungen eingegangen.

4.3.1. Generelle Vorgehensweise

Zur Berechnung der Verbalphrasen-Verknüpfungen wird über alle Sätze iteriert. Innerhalb dessen wird über alle Nominalphrasen, für welche eine vorberechnete Verknüpfung existiert, iteriert. Für jede dieser Nominalphrasen-Verknüpfung ($np_1, comp_1$) wird im

```
calculateBestPairs(vectors1 : List<WordVec>, vectors2: List<WordVec>)
                    : List<(WordVec, WordVec)>
    invariant : |vectors1| == |vectors2|
    size : N := |vectors1|
    if (size > 1)
        bestPairs : List
        for (currentVector ∈ vectors1)
            currentBestPair := calculateBestPair(currentVector, vectors2)
            currentBestPairs := calculateBestPairs(vectors1 \ {currentBestPair[0]},
                                                   vectors2 \ {currentBestPair[1]})
            currentBestPairs := currentBestPairs ∪ {currentBestPair}
            if (pairListSum(currentBestPairs) > pairListSum(bestPairs))
                bestPairs := currentBestPairs
            endif
        endfor
        return bestPairs
    endif
    // basecase
    return {calculateBestPair(vectors1[0], vectors2)}

calculateBestPair(vec: WordVector, vectors : List<WordVec>) : (WordVec, WordVec)
    otherVec:= argmax_{other∈vectors}(cosineSimilarity(vec, other))
    return (vec, otherVec)

pairListSum(vectors: List<(WordVec, WordVec)>) : ℝ
    return Punktwolken−Ähnlichkeit(vectors)
```

Abbildung 4.5.: Pseudocode zur Berechnung der nach Punktwolken-Ähnlichkeitsmaß besten Abbildung zweier Punktwolken. *calculateBestPairs* stößt die Berechnung an.

auf np_1 folgenden übrigen Rest des Satzes nach der nächstmöglichen Nominalphrasen-Verknüpfung (np_2, $comp_2$) gesucht, für welche gilt:

- Zwischen $comp_1$ und $comp_2$ besteht im Architekturmodell mindestens eine direkte Relation.
- Zwischen np_1 und np_2 existiert mindestens eine Verbalphrase.

Wird ein solches Paar gefunden, so wird die Verbalphrase, welche am nächsten an np_2 ist und noch zwischen np_1 und np_2 liegt, für eine Verknüpfung mit den Relationalentitäts-Kandidaten in Betracht gezogen. Ob diese Verknüpfung verworfen wird oder nicht, hängt von den im Folgenden erklärten Parametrisierungen ab.

4.3.2. Parametrisierungen

Die Parametrisierungsmöglichkeiten dieser Phase beschränken sich darauf, dass entschieden werden kann, ob Vergleiche (mit Wortvektoren) stattfinden oder nicht.

Keine Vergleiche

In manchen Fällen gibt es im Architekturmodell keine aussagekräftigen Bezeichner für Relationen. Daher kann dort entweder auf die Berechnung von Verbalphrasen-Verknüpfungen verzichtet werden, oder diese Parametrisierung genutzt werden, die keine der in Unterabschnitt 4.3.1 gefundenen Verknüpfungen verwirft. Aufgrund dessen, dass es zwischen zwei Architekturmodell-Entitäten oftmals nur eine Relation gibt, ist es durchaus möglich, dass diese Vorgehensweise nicht ganz so unpräzise ist, wie man vermuten könnte.

Vergleiche mit Wortvektoren

Sollte das Architekturmodell Relationen mit aussagekräftigen Bezeichner beinhalten, können Wortvektor-basierte Vergleiche sinnvoll genutzt werden. Für jede infrage kommende Relationalentität wird das folgende getan: Für jedes Wort der Verbalphrase wird die Kosinus-Ähnlichkeit zum Bezeichner der Relationalentität berechnet. Der höchste dabei auftretende Wert wird mit dem Schwellenwert-Parameter verglichen. Ist dieser Wert geringer als der Schwellenwert, wird die Verknüpfung verworfen.

5. Implementierung

Nachdem im vorigen Kapitel die in dieser Arbeit entwickelten Konzepte erklärt wurden, widmet sich dieses Kapitel der Umsetzung dieser Konzepte. Zunächst soll ein Überblick über die Architektur gegeben werden. Darin wird unter anderem das Agentenmodell und die Client-Server-Architektur beschrieben. Anschließend wird der Programmablauf anhand des Agenten genauer erklärt. Dies beinhaltet die Kommunikation mit den Rahmenwerken, insbesondere auch die Annotation der Verknüpfungen, und das Anstoßen der Verknüpfungsberechnung. Die darauf folgenden Abschnitte widmen sich dem Server und den Paketen *vectorrepresentation* und *mappingcalc*.

5.1. Gesamtarchitektur

Die Architektur des entwickelten Verfahrens teilt sich in zwei Pakete und den PARSE-Agenten auf. Der PARSE-Agent sorgt für die Kommunikation mit dem PARSE-Rahmenwerk und der Ontologiebibliothek Apache Jena. Von PARSE angestoßen, führt der Agent die Verknüpfungsberechnung durch und annotiert das Ergebnis im PARSE-Graphen. Für die Durchführung der Verknüpfungsberechnung sorgt das Paket *mappingcalc*. Die Repräsentation der Wortvektoren befindet sich in einem eigenen Paket, genannt *vectorrepresentation*. In Abbildung 5.1 ist eine Übersicht über die Architektur gegeben.

Die externen Pakete bestehen aus dem PARSE-Rahmenwerk (*edu.kit.ipd.parse.luna*), dem Ontologie-Rahmenwerk (*org.apache.jena.ontology*) und zwei Schnittstellen-Paketen, die einen vereinfachten Zugriff auf die bereits genannten Pakete ermöglichen. In Abbildung 5.2 ist ein Überblick über die externen Pakete und daraus verwendete Klassen dargeboten. Diese externen Pakete wurden nicht im Rahmen dieser Arbeit erstellt.

Die verwendete *WordEmbedding*-Implementierung *FastTextClientWordEmbedding* stellt den Client-Teil einer Server-Client Lösung für den Zugriff auf Wortvektoren dar. Der Server-Teil wurde in Python implementiert und stellt, entkoppelt von der Laufzeit der PARSE-Applikation, Wortvektoren über HTTP bereit. Der dadurch entstehende laufzeittechnische Mehraufwand wird durch einen Puffer-Speicher im Client, also im Programmkontext der PARSE-Applikation, aufgefangen.

5.2. Programmablauf

Für den Programmablauf ist der PARSE-Agent, im Modell *WordVectorMappingAgent* genannt, von zentraler Bedeutung, wie in Abbildung 5.3 zu sehen ist.

Zunächst wird PARSE gestartet und aus dem Dokumentationstext ein Graph aufgebaut, in welchem jedem Wort ein *INode*, also ein Knoten zugewiesen wird. Diese Knoten haben

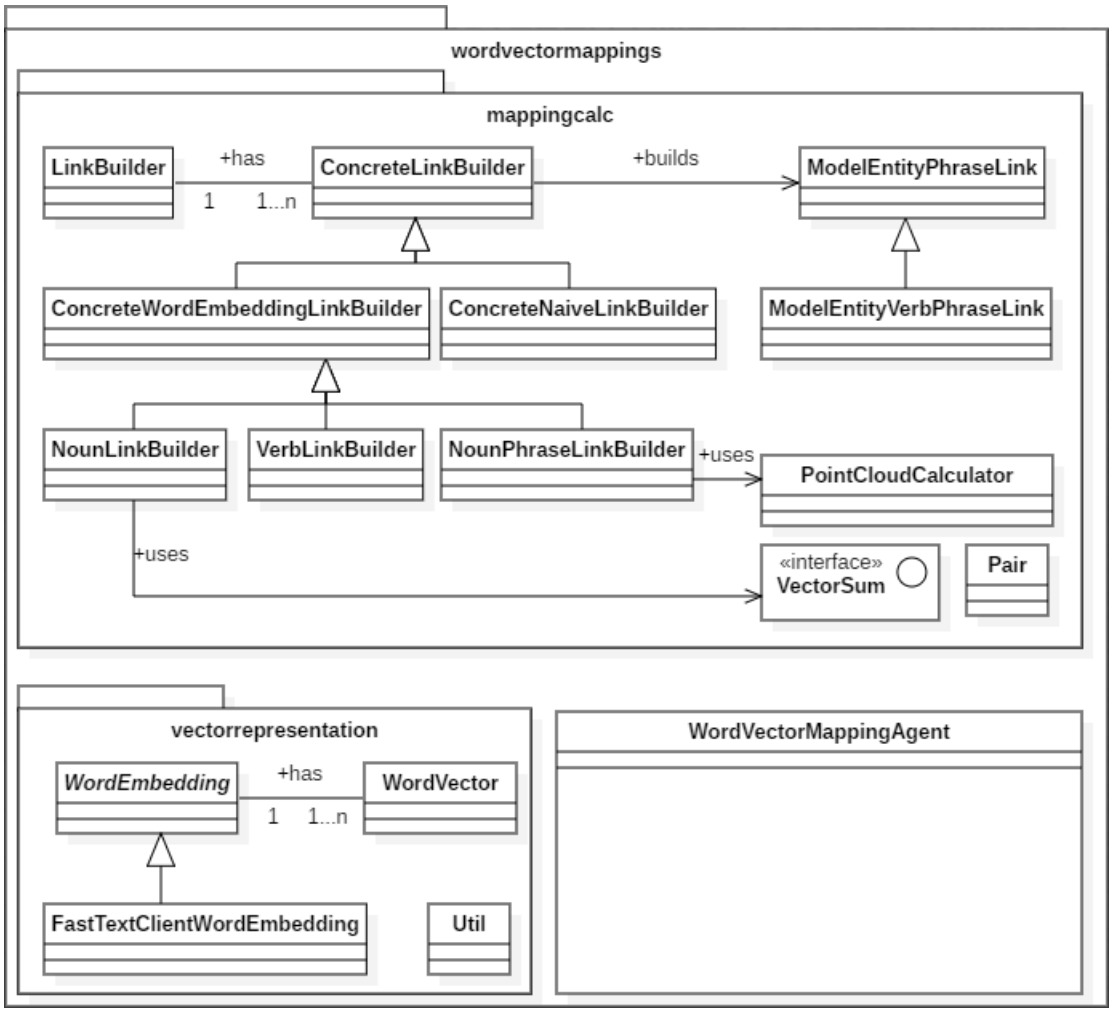

Abbildung 5.1.: Architekturdiagramm der Java-Seite ohne Details.

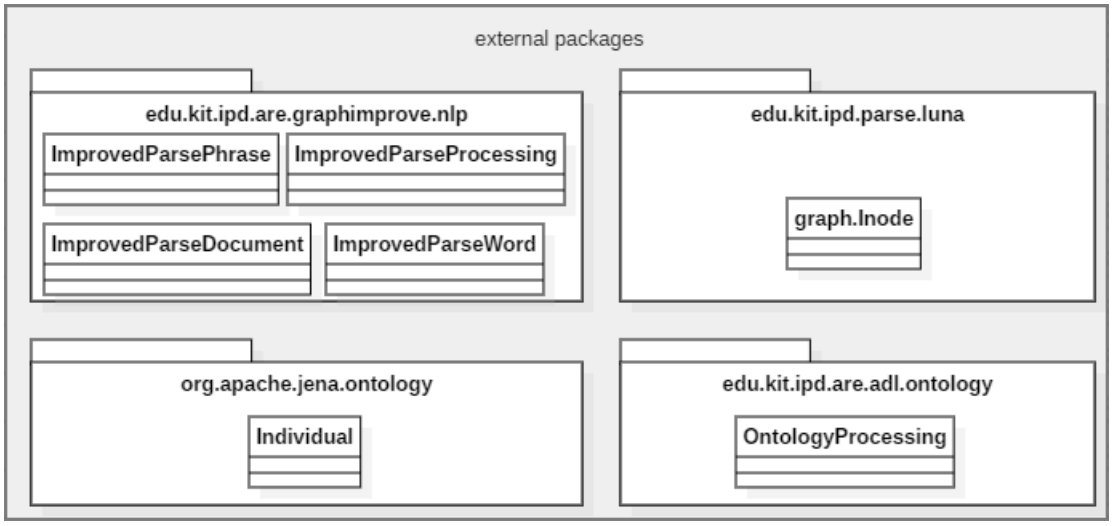

Abbildung 5.2.: Externe Pakete und daraus verwendete Klassen.

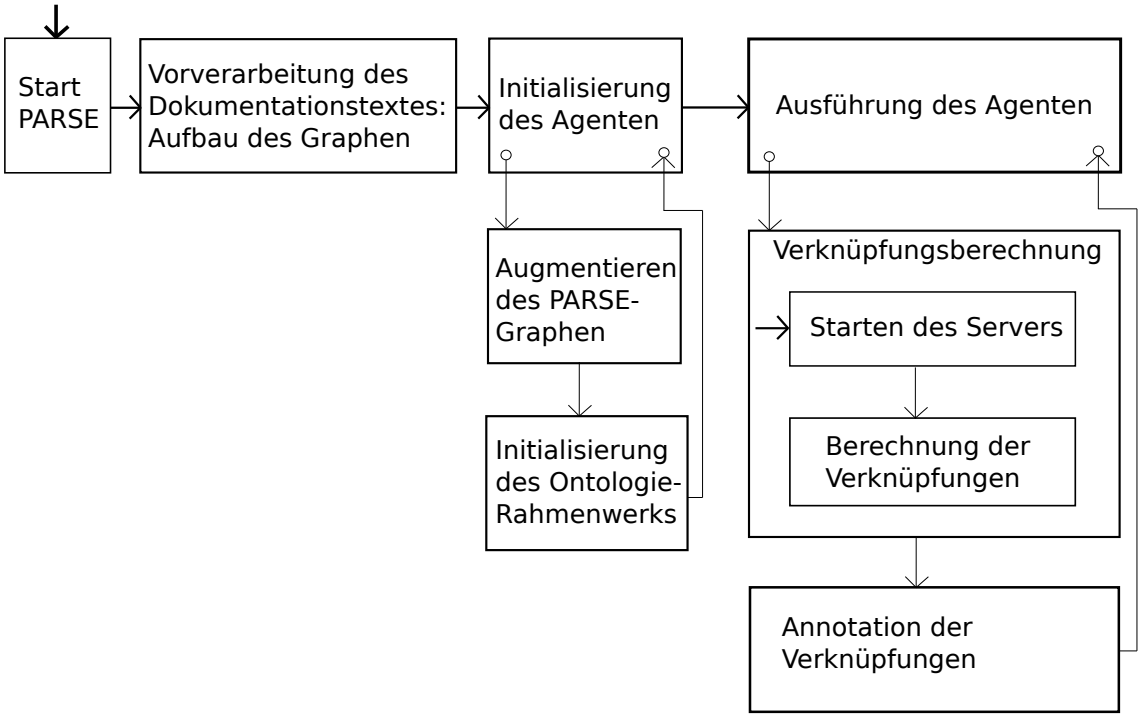

Abbildung 5.3.: Schematischer Programmablauf. Die vertikale Flussrichtung zeigt vom Agenten angestoßene Abläufe.

einige Attribute, welche unter anderem die Position im Satz, und im Dokument, sowie die Wort- bzw. Phrasen-Klassifikation beschreiben.

Danach wird der Agent initialisiert. Dabei wird der PARSE-Graph mit Hilfe von *GraphImprove* augmentiert, indem für jede Phrase ein eigener Knoten angelegt wird. Diese Knoten sind untereinander, und mit den jeweiligen präexistenten Wort-Knoten verbunden. Dies erleichtert die Handhabung und Annotation. Anschließend wird das Ontologie-Rahmenwerk initialisiert, indem das Architekturmodell geladen wird.

Zuletzt wird der Agent ausgeführt. Dabei wird zunächst die Verknüpfungsberechnung angestoßen. Sollte der Wortvektor-Server nicht bereits gestartet sein, wird dieser nun gestartet. Für mehrfache Ausführungen zeigt sich die Entkopplung von Server und PARSE-Umgebung als durchaus vorteilhaft, da beim Start des Wortvektor-Servers mehrere Gigabyte an Daten verarbeitet werden müssen. Nun werden, in mehreren Schritten, die Verknüpfungen berechnet. Ist dies geschehen, werden die Verknüpfungen an die von der *GraphImprove*-Bibliothek hinzugefügten Phrasen-Knoten annotiert. In Abbildung Abbildung 5.4 wird die Annotation der Verknüpfungen verdeutlicht: Die unterste Zeile besteht aus den originalen, von PARSE generierten Knoten und Kanten. In den einzelnen Knoten sind die Wörter des Satzes und einige weitere Werte, die das Wort im Satz und im Dokument indizieren und es klassifizieren, enthalten. Darauf aufbauend ist in der mittleren Zeile die von *GraphImprove* generierte Abstraktion auf Phrasen-Ebene zu sehen. Hier sind die jeweiligen Phrasen-Knoten miteinander verbunden und formen so wieder einen Satz. In der obersten Zeile sind die von *WordVectorMappingAgent* generierten Knoten zu sehen, die die URI, also den eindeutigen Identifikations-Bezeichner eines *Individuals* im *OWL*-Modell

der Architektur, und den Namen der entsprechenden Architekturmodell-Entität enthält. Diese Knoten zeigen jeweils auf einen Knoten einer mit der Architekturmodell-Entität verknüpften Nominal- oder Verbalphrase.

Die erzeugten und annotierten Verknüpfungen zum Architekturmodell können nun weiterverwendet werden.

5.3. Wortvektor-Server

Der vom sonstigen Programmablauf entkoppelte Wortvektor-Server ist dafür zuständig, Wortvektor-Anfragen zu beantworten. Hierbei wird ein Subwort-Modell von Fasttext genutzt. Der Server kann entweder vom Wortvektor-Agenten, also aus der PARSE-Laufzeit heraus, oder manuell gestartet werden. Das Initialisieren des Servers läuft wie folgt ab: Zunächst werden alle Start-Argumente verarbeitet: *sim_task_db_size* gibt die Anzahl an Wörtern an, die für Ähnlichkeitsvergleiche verwendet werden sollen. *port* gibt den Port an, unter dem der Server laufen soll. Die Adresse ist fest *localhost. model_location* gibt den Speicherort des Fasttext-Modells an.

Nachdem diese Argumente verarbeitet wurden, wird das als Datei bereitstehende Fasttext-Modell in den Hauptspeicher geladen. Dies geschieht mithilfe einer von Facebook, den Entwicklern von Fasttext, bereitgestellten Python-Bibliothek.

Anschließend wird der Server initialisiert und gestartet. Der Server ist mit Hilfe der Python-Bibliothek *Flask* [6] realisiert und stellt mehrere Funktionen zur Verfügung, die über eine an das REST-Paradigma angelehnte HTTP-Schnittstelle angesprochen werden können. Diese werden im Folgenden erklärt. Tatsächlich verwendet wird davon allerdings nur die einfache Funktion, einen Wortvektor zu einem gegebenen Wort zu liefern. Alle Anfragetypen geben eine Antwort im JSON-Format zurück.

Wortvektor-Anfrage

Der Pfad `/fastTextSubwordWordEmbedding/vectors/<word>` liefert zu einem gegebenen Wort `<word>` einen Wortvektor zurück. Dies wird mittels *HTTP-GET*-Anfragen realisiert.

Nächster-Wortvektor-Anfrage

Der Pfad `/fastTextSubwordWordEmbedding/vectors/getNearestForVector` liefert zu einem gegebenen Wortvektor den nächsten Wortvektor zurück, der in der Datenbank enthalten ist. Hier kommt der Parameter *sim_task_db_size* ins Spiel. Dieser beschreibt, wie viele Wörter für diesen Vergleich verwendet werden sollen. Dabei werden für $sim\_task\_db\_size = n$ die n am Häufigsten im Trainingsdatensatz vorgekommenen Wörter ausgewählt. Der Wortvektor aus diesen n Wort–Wortvektor-Paaren, welcher die höchste Kosinus-Ähnlichkeit zu dem Wortvektor aus der Anfrage erzielt, wird zurückgegeben. Der Grund für die Beschränkung auf n Wort–Wortvektor-Paaren ist die Verarbeitungsdauer. Sowohl die Anfrage, als auch die Antwort werden im bereits erwähnten JSON-Format dargestellt. Dieser Anfragetyp ist mithilfe von *HTTP-POST* realisiert.

Nächste-K-Wortvektoren-Anfrage

Der Pfad `/fastTextSubwordWordEmbedding/vectors/getNearestKForVector` liefert zu ei-

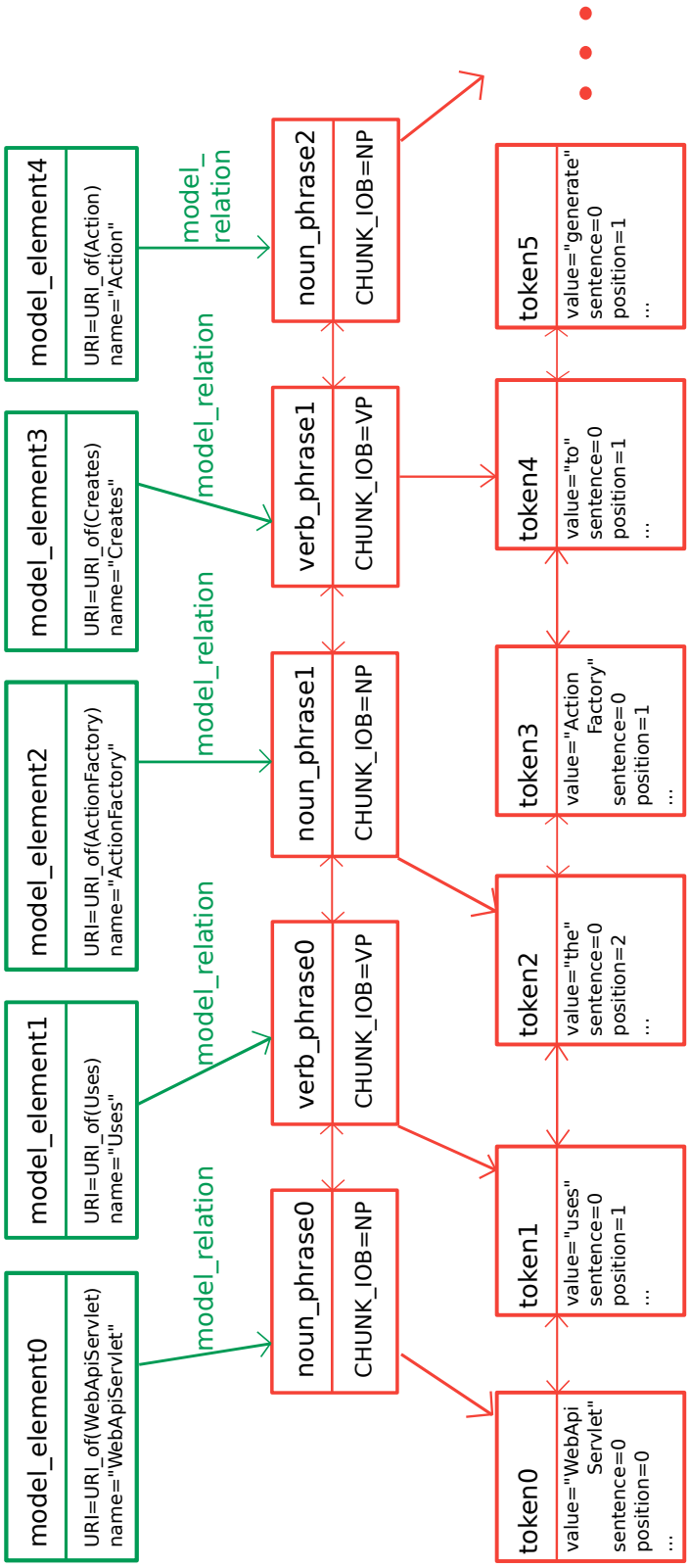

Abbildung 5.4.: Ausschnitt eines mit Modellentitäten-Verknüpfungen annotierten PARSE-Graphen.

nem gegebenen Wortvektor die nächsten k Wortvektoren zurück. Dabei wird, wie auch bei der Nächster-Wortvektor-Anfrage der Parameter *sim_task_db_size* benutzt, um die Wortmenge zu beschränken. Hierbei werden die bezüglich Kosinus-Ähnlichkeit ähnlichsten *k* Wort-Wortvektor-Paare zurückgegeben. Auch hier wird wieder das JSON-Format sowohl für die Anfrage, als auch für die Antwort verwendet. Der Anfragetyp ist ebenfalls in *HTTP-POST* umgesetzt.

5.4. Wortvektor-Repräsentations-Paket

Das Paket *vectorrepresentation* kapselt Wortvektoren und deren Quelle. Dabei stellen Kindklassen der abstrakten Klasse *WordEmbedding* eine Modellinstanz eines bestimmten Wortvektor-Verfahrens dar. *FastTextClientWordEmbedding* ist hierbei dasjenige, welches für die Entwicklung und Evaluation dieser Arbeit genutzt wurde. Über die Elternklasse WordEmbedding wird vom genauen Modell abstrahiert. Als Schnittstelle für den Informationsaustausch mit dem Modell bietet WordEmbedding Methoden, um Wortvektoren zu gegebenen Strings oder auch zu gegebenen Wortvektoren ähnliche Wortvektoren zu erhalten. Auch die Logik zur Berechnung der in Unterabschnitt 4.1.3 erklärten Untervektorräume und eine entsprechende Methode befindet sich hier.

Die Klasse *WordVector* kapselt einen Wortvektor. Dazu bietet die Klasse Berechnungsmethoden für Addition, Subtraktion, skalare Multiplikation, Normierung und Kosinus-Ähnlichkeit.

5.5. Verknüpfungsberechnungs-Paket

Im Verknüpfungsberechnungs-Paket *mappingcalc* geschieht die Hauptarbeit; hier werden mithilfe verschiedener Klassen Verknüpfungen berechnet und zusammengeführt.

Die Klasse *LinkBuilder* ist dazu gedacht, Verknüpfungen zusammenzutragen. Dazu werden Instanzen von Kindklassen von *ConcreteLinkBuilder* einer Instanz von LinkBuilder übergeben. Letztere stößt dann die Verknüpfungsberechnung der einzelnen ConcreteLinkBuilder an und trägt die Ergebnisse zusammen.

Die ConcreteLinkBuilder-Kindklassen *NounLinkBuilder*, *NounPhraseLinkBuilder* und *VerbLinkBuilder* spiegeln die drei in Abschnitt 4.1, Abschnitt 4.2 und Abschnitt 4.3 eingeführten Phasen bzw. Arten von Verknüpfungsberechnungen wider.

Die Klasse *ModelEntityPhraseLink* kapselt eine Verknüpfung zwischen Phrase und Ontologie-Individualentität. Letztere steht stellvertretend für eine Architekturmodell-Entität.

In den folgenden Unterabschnitten werden einige Details der wichtigsten Klassen erklärt.

5.5.1. LinkBuilder

Die *LinkBuilder*-Klasse dient dazu, die verschiedenen Instanzen von Kindklassen von *ConcreteLinkBuilder* zu bündeln. Dazu werden diese nacheinander dem LinkBuilder hinzugefügt, was eine *FIFO*-Warteschlange erzeugt. Sobald der „Build-Prozess“ angestoßen wird,

werden die in dieser Warteschlange enthaltenen ConcreteLinkBuilder-Instanzen nacheinander ausgeführt. Nach jeder Ausführung einer ConcreteLinkBuilder-Instanz werden die von diesem generierten Verknüpfungen zur Gesamtmenge hinzugefügt. Dies geschieht parametrisiert: Duplikate können entweder ignoriert, oder eliminiert werden. Die resultierende Gesamtmenge wird der nächsten ConcreteLinkBuilder-Instanz übergeben. Diese kann diese *vorberechneten Verknüpfungen* für die eigene Verknüpfungsberechnung nutzen, was insbesondere für die *VerbLinkBuilder*-Klasse notwendig ist. Sobald die Warteschlange leer ist, wird die resultierende Gesamtmenge der Verknüpfungen gefiltert. Auch dies geschieht wieder parametrisiert. Es kann ausgewählt werden, dass für alle vorkommenden Fälle von (*Phrase*Entität)-Verknüpfungsduplikaten $(p, e_i), (p, e_j)$ dasjenige Paar mit der niedrigeren Ähnlichkeit verworfen wird. In Abbildung 5.5 ist ein Aktivitätsdiagramm dieses Ablaufes zu sehen.

5.5.2. NounLinkBuilder

Die *NounLinkBuilder*-Klasse setzt die in Abschnitt 4.1 beschriebene erste Phase des Entwurfs um. Initialisiert wird eine NounLinkBuilder-Instanz mit einem *WordEmbedding*, sowie dem PARSE-Graphen (via *ImprovedParseDocument*) und der das Architekturmodell enthaltenden Ontologie (via *OntologyProcessing*). Diese stammen aus den externen Paketen. Der Schwellenwert für die Ähnlichkeitsvergleiche wird ebenfalls benötigt.

An der initialisierten Instanz können nun noch weitere Parametrisierungen vorgenommen werden. Die zu verwendende Vektorsumme und ein eventueller Untervektorraum können angegeben werden. Anschließend wird mit dem Aufruf von `build()` die Verknüpfungsberechnung angestoßen. Es wird über alle Nominalphrasen iteriert. Für jede dieser Nominalphrasen wird über alle Wörter iteriert. Für jedes dieser Wörter wird nach dem nach Kosinus-Ähnlichkeit besten Paar (*Phrase*, Entität) gesucht. Liegt dieses über dem Schwellenwert, wird dieses Paar nach der Berechnung der restlichen Paare zusammen mit diesen zurückgegeben.

Die Suche nach dem besten Paar läuft wie folgt ab: Für jedes *Individual*, welches eine Modellentität repräsentiert, wird ein Wortvektor berechnet. Dieser wird mit dem Wortvektor des aktuellen Wortes mithilfe der Kosinus-Ähnlichkeit verglichen. Das Paar mit der höchsten Kosinus-Ähnlichkeit wird zurückgegeben.

Berechnung der Wortvektoren

Die Wortvektorberechnung erfolgt mithilfe der Vektorsummen. Dabei wird *VectorSum* als funktionale Schnittstelle realisiert. Zunächst wird jedoch das jeweilige Wort und der Entitätenbezeichner wie in Unterabschnitt 4.1.2 beschrieben aufgespalten. Das geschieht für die Trennung von Binnenmajuskel-Komposita anhand des folgenden regulären Ausdrucks:

```
(?<!(^|[A-Z]))(?=[A-Z])|(?<!^)(?=[A-Z][a-z])
```

Dieser findet Stellen der Länge null, also Stellen *zwischen* den Zeichen, in Zeichenketten, welche eine der folgenden Bedingungen erfüllen:

- Vor der Stelle kommt weder ein Großbuchstabe noch ein Zeichenketten-Anfang vor. Nach der Stelle kommt ein Großbuchstabe vor.

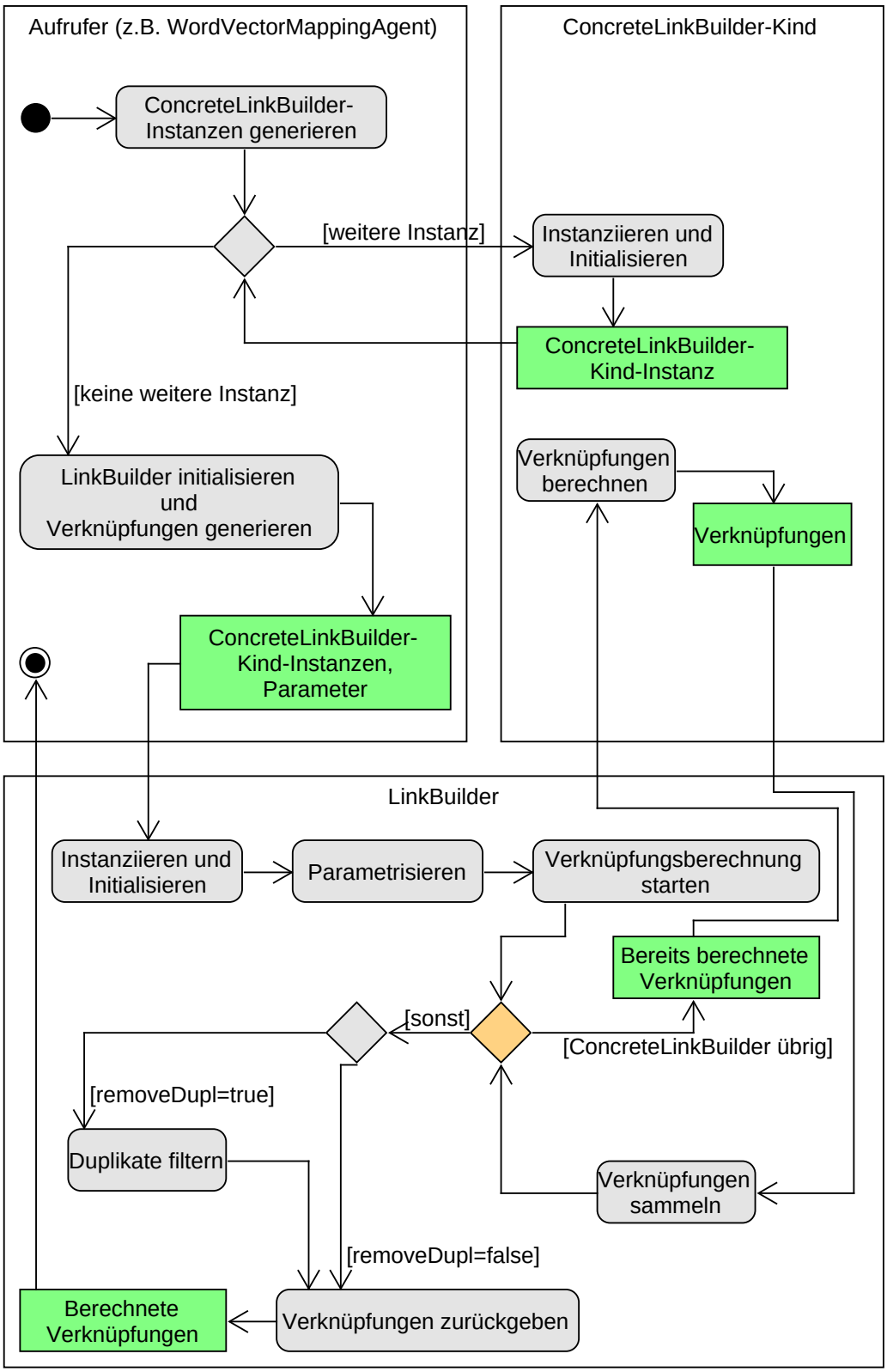

Abbildung 5.5.: Aktivitätsdiagramm der Benutzung eines LinkBuilders. Grüne Kästen beschreiben Daten, orange steht für Schleifen-Verzweigungen.

- Vor der stelle kommt kein Zeichenketten-Anfang vor. Nach der Stelle tritt ein Großbuchstabe, gefolgt von einem Kleinbuchstabe auf.

Diese beiden Bedingungen sind notwendig, um auch Wörter wie *DefaultSCMCheckoutStrategyImpl* verarbeiten zu können, welche auch mehrere aufeinander folgende Großbuchstaben beinhalten. Die erste Bedinung würde hierbei zwischen *Default* und *SCMCheckoutStrategyImpl* schneiden. Die zweite Bedingung würde zwischen *DefaultSCM* und *CheckoutStrategyImpl* schneiden. Damit ist *SCM* erfolgreich als Subwort ausgeschnitten. Auf die anderen Stellen treffen beide Bedingungen zu. Anschließend wird jedes Subwort noch anhand der anderen Trennzeichen aus Unterabschnitt 4.1.2 aufgeteilt.

Die resultierenden Subwörter werden dann mithilfe der ausgewählten *VectorSum* zu einem Wortvektor verrechnet. Falls eine Aufspaltung nicht möglich ist, wird direkt der Wortvektor gebildet.

5.5.3. NounPhraseLinkBuilder

Die *NounPhraseLinkBuilder*-Klasse setzt die in Abschnitt 4.2 beschriebenePhase des Entwurfs um. Der Ablauf wird im Folgenden anhand des in Abbildung 5.6 gezeigten Diagrammes erklärt.

Die Initialisierung und Parametrisierung funktioniert ähnlich wie bei der in Unterabschnitt 5.5.2 beschriebenen *NounLinkBuilder*-Klasse: Auch hier gibt es wieder einen Schwellenwert. Es kann zudem ausgewählt werden, ob das Punktwolkenverfahren verwendet werden soll und ob bei ungleich großen Punktwolken auf das Vektorsummenverfahren zurückgegriffen werden soll. Die Wort- und Wortarten-Filter werden als `Collection<String>` realisiert, die einem instanziierten NounPhraseLinkBuilder übergeben werden können. Für die Wortarten-Gewichtung kann eine `Map<String, Double>` übergeben werden. Die Filter und Gewichte werden sowohl vom Punktwolkenverfahren als auch vom Vektorsummenverfahren verwendet.

Die Verknüpfungsberechnung, von `build()` angestoßen, läuft prinzipiell analog zu Unterabschnitt 5.5.2 ab: Für alle im PARSE-Graph vertretenen Nominalphrasen wird die ähnlichste nicht-relationale Modellentität gesucht. Dafür wird über alle solchen Entitäten iteriert und die Ähnlichkeit zur aktuellen Nominalphrase berechnet. Das Paar (*Nominalphrase*, *Entität*) mit der höchsten Ähnlichkeit wird zurückgegeben und, falls es den Ähnlichkeits-Schwellenwert überschreitet, zu den gefundenen Verknüpfungen hinzugefügt.

Für die Ähnlichkeitsberechnung werden zunächst der Wort- und der Wortartenfilter angewendet. Anschließend gibt es mehrere mögliche Abläufe, die von der Parametrisierung abhängen. In den Folgenden Unterabschnitten werden diese erläutert:

5.5.3.1. Vektorsummenverfahren

Das Vektorsummenverfahren läuft ähnlich zu Unterabschnitt 5.5.2 ab. Hierbei wird der Wortvektor für den Entitätenbezeichner genau wie in Unterabschnitt 5.5.2 berechnet. Für die Berechnung des Wortvektors der Nominalphrase wird die gewichtete Summe verwendet. Dazu wird jedem nach dem Filtern übrig gebliebenen Wort der Phrase ein Gewicht

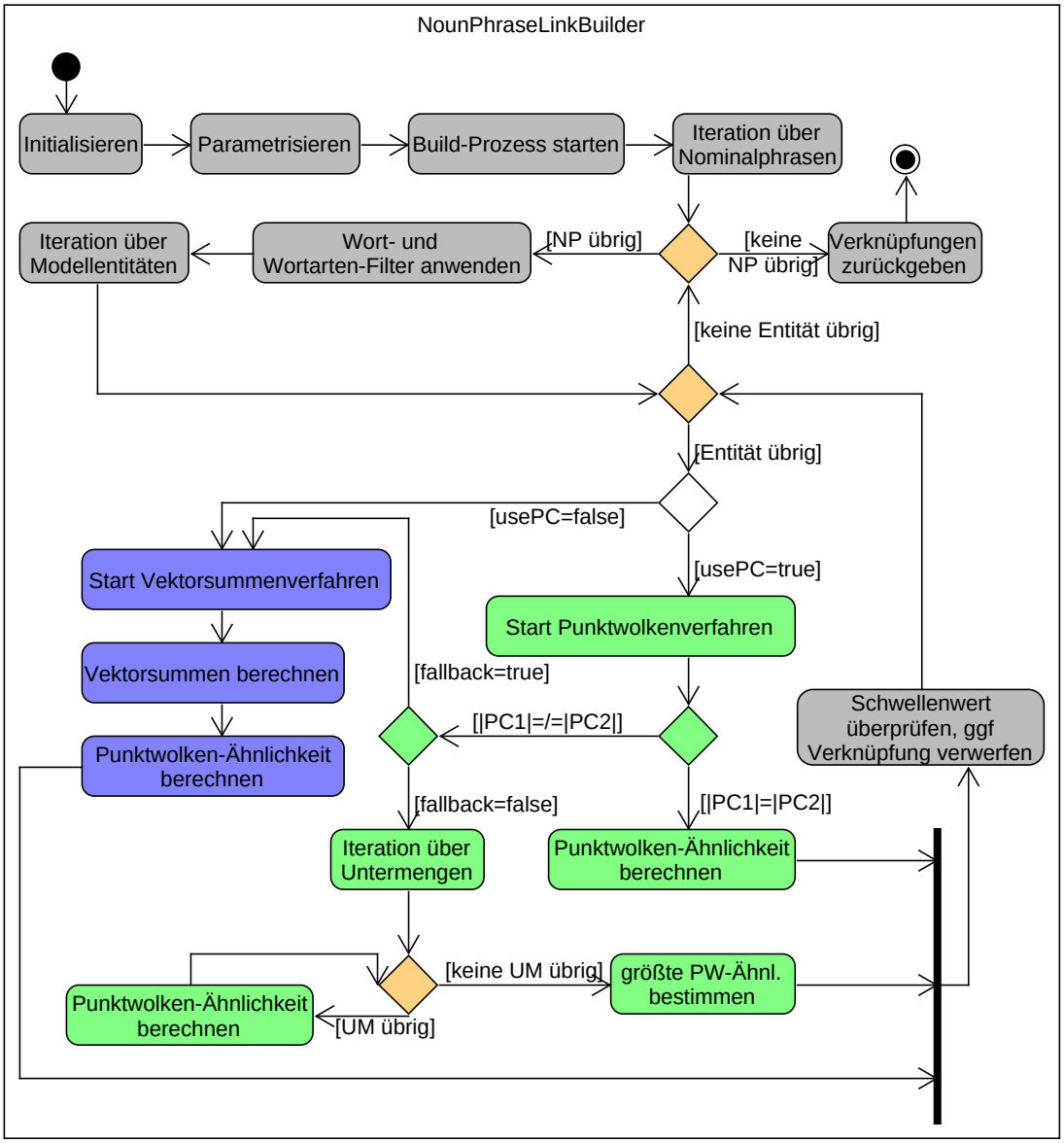

Abbildung 5.6.: Aktivitätsdiagramm der Benutzung einer NounPhraseLinkBuilder-Instanz. Die grün markierten Abläufe gehören zum Punktwolken- und die Blauen zum Vektorsummenverfahren. Orange markiert Verzweigungen für Iterationen.

entsprechend der übergebenen Wortarten-Gewichte zugeordnet. Der Standardwert ist dabei 1.0.

Zuletzt wird die Kosinus-Ähnlichkeit zwischen den beiden berechneten Wortvektoren berechnet und zurückgegeben.

5.5.3.2. Punktwolkenverfahren

Für das Punktwolkenverfahren wird, falls der *fallback*-Parameter aktiv ist, bei ungleich großen (Sub-)Wortmengen der Phrase einerseits und des Entitätenbezeichners andererseits, auf das Vektorsummenverfahren zurückgegriffen. Falls dieser Parameter nicht aktiv ist, müssen die beiden Wortmengen auf dieselbe Kardinalität gebracht werden. Falls nicht, wird direkt die Punktwolken-Ähnlichkeit berechnet. Sei n die Kardinalität der kleineren Wortmenge. Falls dieser Parameter nicht aktiv ist, wird anhand eines rekursiven Algorithmus jede n-elementige Teilmenge der größeren Wortmenge berechnet. Dies geschieht in einer gekapselten Unterklasse, genannt *FixedSizedSubsetProcessing*. Für jede dieser Teilmengen wird die Punktwolken-Ähnlichkeit berechnet. Nach Ausführung dieses Algorithmus wird der höchste Ähnlichkeitswert zurückgegeben.

Berechnung der Punktwolken-Ähnlichkeit

Zur Berechnung der Punktwolken-Ähnlichkeit wird der in Abschnitt 4.2 erklärte Ähnlichkeitsalgorithmus verwendet. Dieser ist in der Klasse *PointCloudCalculator* implementiert.

5.5.4. VerbLinkBuilder

Die *VerbLinkBuilder*-Klasse setzt die in Abschnitt 4.3 beschriebenePhase des Entwurfs um. Die Initialisierung gleicht den anderen beiden *ConcreteWordEmbeddingLinkBuildern*, auch hier wird wieder ein Schwellenwert mitgegeben. Ein Schwellenwert von unter 0 steht für die Nicht-Benutzung der Vergleiche, wie in Unterabschnitt 4.3.2 beschrieben. Zusätzlich kann eingestellt werden, ob statt Wortvektor-Vergleichen anhand der *Levenshtein*-Distanz verglichen werden sollen. Das ist vor allem für die Evaluation vonnöten.

Ansonsten wird wie in Unterabschnitt 4.3.1 beschrieben vorgegangen. Die Relationalentitäten werden dabei anhand einer inneren Klasse *Connector* dargestellt, welche Referenzen auf die beiden *Individual*-Instanzen enthält, die jeweils eine nicht-relationale Modell-Komponente beschreiben. Des weiteren ist eine Referenz auf die relationale Modell-Komponente enthalten. So kann bei der Berechnung der Verknüpfungen für zwei gegebene Nominalphrasen-Verknüpfungen $(np_1, comp_1)$, $(np_2, comp_2)$ leichter nach einer passenden Relation gesucht werden.

6. Evaluation

In diesem Kapitel wird die Evaluation der in Kapitel 4 beschriebenen Phasen dargelegt. Da diese Phasen aufeinander aufbauen, kann eine Phase nicht unabhängig von den vorherigen Phasen evaluiert werden. Es wird also immer der Gesamtstand zum Ende der jeweiligen Phase evaluiert. Die Verbalphrasen bilden hierbei nur insofern eine Ausnahme, als dass bei der Evaluation der Verbalphrasen-Phase nur die Verknüpfungen von Verbalphrasen zu relationalen Modellentitäten betrachtet werden. Die Mengen der möglichen findbaren Verknüpfungen von Phase 1 und 2 zum einen und Phase 3 zum anderen sind nämlich disjunkt. In den ersten Phasen werden nur Nominalphrasen (und Einzelnomen) betrachtet, während die letzte Phase sich ausschließlich den Verben widmet.

Dieses Kapitel ist wie folgt strukturiert: Zunächst wird die Evaluationsmethodik, nach der sich die Evaluation richtet, beschrieben. Darin wird genauer auf Ziele, Fragen und Metriken eingegangen. Anschließend folgt ein Abschnitt über die verschiedenen zur Evaluation verwendeten Datensätze. Die nächsten drei Abschnitte zeigen die Ergebnisse der jeweiligen Phasen mitsamt einer kurzen Interpretation. Zuletzt folgt ein Abschnitt, in welchem eine insgesamte Interpretation stattfindet. Hier werden auch insbesondere die Bedrohungen für die Validität der Ergebnisse diskutiert.

6.1. Methodik

Das Evaluationskonzept richtet sich nach der *GQM*-Methode (Goal Question Metric). Hierbei werden zunächst Ziele definiert, die die Evaluation erreichen soll. Dann werden dazu passende Fragen bezüglich konkreter Eigenschaften gestellt, welche durch Metriken beantwortet werden sollen.

6.1.1. Ziele

Güte der Verknüpfungen von Text- zu Modell-Entitäten bestimmen

Es soll Ziel des zu entwickelnden Modells sein, Entitäten von Dokumentationstexten die durch sie beschriebenen Entitäten in den Softwarearchitekturmodellen möglichst vollständig und korrekt zuzuordnen. Diese Beschreibung gilt im Speziellen auch für die zwei anderen Ziele.

Güte der Leistung des Modells im Bereich der Nomen und Nominalphrasen bestimmen

Für dieses Ziel werden die zwei diesem Ziel zugeordneten Phasen, Abschnitt 4.1 und Abschnitt 4.2 einzeln evaluiert. Die Ergebnisse der Nominalphrasen-Phase enthalten dabei stets die der Einzelnomen-Phase. Es wird also bei der Bewertung dieser Phase vor allem die Veränderung in der Leistungsgüte im Vergleich zur Einzelnomen-Phase betrachet.

Güte der Leistung des Modells im Bereich der Relationserkennung über Verben bestimmen

Die Güte der Relationserkennung über Verben wird, wie bereits beschrieben, separat betrachtet.

6.1.2. Fragen

Die im Folgenden definierten Fragen gelten für alle drei Ziele gleichermaßen. Es ändert sich nur die verwendete Menge der vorgegebenen Verknüpfungen, also die Verknüpfungsklasse. Die Fragen werden je nach Phase hinsichtlich der Verknüpfungsklassen *Nomen/ Nominalphrasen* oder *Verbalphrasen* beantwortet.

Die Datenbasis für die Beantwortung der Fragen bezüglich der Güte des zu evaluierenden Modells wird durch mehrere Datensätze gegeben. Diese bestehen jeweils aus Text und einem Architekturmodell, sowie einem Goldstandard, der festlegt, welche Verknüpfungen es zu finden gilt.

Es soll die absolute Qualität innerhalb der jeweils relevanten Verknüpfungsklasse gemessen werden (Vergleich mit den manuell festgelegten Verknüpfungen des Goldstandards). Auf Basis dessen soll ein Vergleich zum naiven Ansatz und zu den vorherigen Inkrementen gezogen werden.

Wie gut ist die Erkennung von Zusammenhängen von Text- mit Modell-Entitäten?

Hier gilt es, die Gesamtleistung des zu evaluierenden Modells zu messen.

- Wie viele der manuell erstellten Verknüpfungen werden vom zu evaluierenden Modell gefunden?
- Wie viele der gefundenen Verknüpfungen sind korrekt (in der Menge der manuell erstellten Verknüpfungen enthalten)?

Wie gut ist das Modell im Vergleich zum naiven Ansatz?

Der naive Ansatz ist es, einfach alle Textentitäten mit allen Modellentitäten auf String-Gleichheit zu vergleichen und alle Funde als Verknüpfungen auszugeben. Dieser Ansatz soll hierfür ebenso nach den im vorigen Abschnitt definierten Fragen evaluiert werden. Auch die Metriken (*Präzision*, *Ausbeute* und *F-Maße*) bleiben dieselben.

6.1.3. Metriken

Die hier vorgestellten Metriken bilden eine Basis für die Beantwortung der Fragen aus Unterabschnitt 6.1.2. Sei V für diesen Abschnitt die Menge der vom manuell erstellten Goldstandard vorgegebenen, als *korrekt* bezeichneten Verknüpfungen und V_M die Menge der vom Modell gefundenen Verknüpfungen.

Präzision (engl. *precision*)

Die Präzision ist der Anteil der berechneten und nach Goldstandard korrekten Verknüpfungen an allen berechneten Verknüpfungen, sie misst also, wie viele der berechneten

Verknüpfungen korrekt sind Die Präzision ist definiert durch

$$Präzision = \frac{|V_M \cap V|}{|V_M|} \tag{6.1}$$

Ausbeute (engl. *recall*)

Die Ausbeute ist der Anteil der berechneten und nach Goldstandard korrekten Verknüpfungen an allen korrekten Verknüpfungen. Sie misst also, wie viele der manuell erstellten Verknüpfungen vom zu evaluierenden Modell gefunden werden. Die Ausbeute ist definiert durch

$$Ausbeute = \frac{|V_M \cap V|}{|V|} \tag{6.2}$$

F_β-Maß

Das F_1-Maß, ein Spezialfall des F_β-Maßes gewichtet Ausbeute und Präzision gleichmäßig. Sollte man Genauigkeit oder Präzision höher gewichten wollen, was je nach Anwendungsfall beides sinnvoll sein kann, kann man dies mit dem F_β-Maß tun. Dabei gewichten Werte $\beta > 1$ die Ausbeute höher (β^2-mal so hoch). Werte $\beta < 1$ gewichten dementsprechend die Präzision höher ($\frac{1}{\beta^2}$-mal so hoch).

$$F_\beta = (1 + \beta^2) * \frac{Ausbeute * Präzision}{Ausbeute + \beta^2 * Präzision} \tag{6.3}$$

Für die Evaluation dieser Arbeit werden das $F_{\sqrt{2}}$- und das $F_{\sqrt{\frac{1}{2}}}$-Maß betrachtet.

F_1-Maß

Das F_1-Maß ist das harmonische Mittel der Maße Ausbeute und Präzision und ein Spezialfall des F_β-Maßes ($\beta = 1$) Es ist der Versuch, beide Maße so zu kombinieren, dass beide insofern als „wichtig“ betrachtet werden, dass z.B. ein sehr hoher Ausbeute- und ein sehr niedriger Präzisionswert einen niedrigen F_1-Wert ergeben. Dies ist notwendig, da die beiden Maße für sich alleine keine verlässliche qualitative Aussage über das Modell treffen. Eine hohe Ausbeute kann durch das Modell, welches alle möglichen Verknüpfungen ausgibt, erreicht werden, während eine hohe Präzision durch die Ausgabe der leeren Menge erreicht werden kann. Diese Extrembeispiele zeigen das Dilemma, welches es mit diesem Maß aufzulösen gilt.

Das F_1-Maß ist definiert durch

$$F_1 = 2 * \frac{Ausbeute * Präzision}{Ausbeute + Präzision} \tag{6.4}$$

6.2. Datensätze

Die Evaluation wird anhand verschiedener Datensätze durchgeführt. Dabei gibt es zwei verschiedene in PCM modellierte Architekturmodelle, *MediaStore* und *TEAMMATES* [22, 28]. Für diese wurden jeweils unterschiedliche Texte verfasst, bzw. gegebene Texte modifiziert.

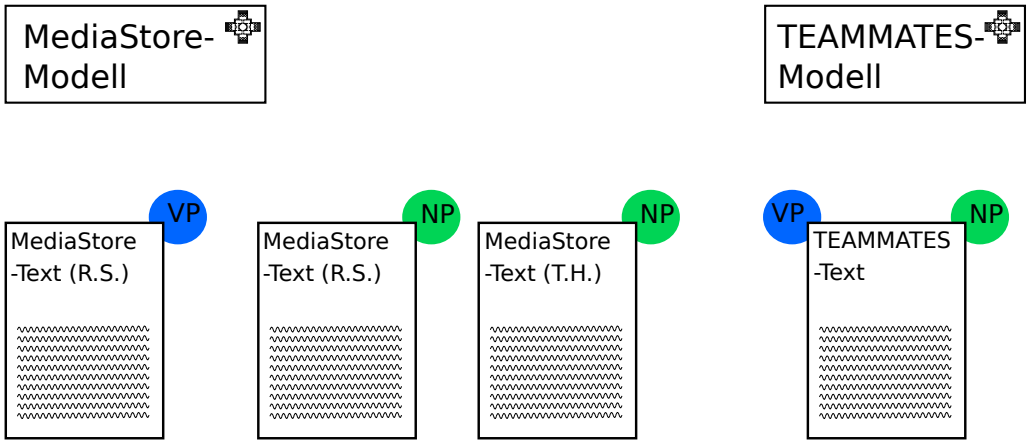

Abbildung 6.1.: Übersicht der Datensätze. Für MediaStore existieren mehrere Texte. Die Kreise zeigen die Verknüpfungsklassen an.

6.2.1. MediaStore

Anhand des MediaStore-Modells wird in *Modelling And Simulating Software Arcitectures* [22] das Palladio-Komponentenmodell erklärt. Für die Verknüpfungsklasse der Nomen und Nominalphrasen wurden hierbei vom Autor dieser Arbeit und von einer anderen Studentin jeweils ein Beschreibungstext von MediaStore erstellt. Letzterer war das in dieser Arbeit beschriebene Verfahren zum Erstellungszeitpunkt nicht bekannt. Das Ziel dieser Modifikation ist ein höherer Anteil an nichttrivialen Wort- bzw. Phrasenverknüpfungen, was dazu dienen soll, die Grenzen des Ansatzes besser finden zu können. Die jeweiligen Goldstandards, also die Referenzpunkte, welche festlegen, welche Verknüpfungen korrekt sind, wurden von den jeweiligen Text-Verfassern mitangelegt. Der den vom Autor dieser Arbeit verfassten Text enthaltende Datensatz wird im Folgenden $MediaStore_1$, und der andere $MediaStore_2$ genannt.

Für die Verknüpfungsklasse der Verben wurde der vom Autor dieser Arbeit für die andere Klasse modifizierte Text um einige weitere Sätze erweitert, um eine größere Datenbasis und damit Aussagekraft zu erreichen. Dieser Datensatz wird mit $MediaStore_{1V}$ benannt.

6.2.2. TEAMMATES

Das TEAMMATES-Modell wurde im Rahmen dieser Arbeit anhand der Dokumentation von TEAMMATES im Palladio-Komponentenmodell (PCM) nachmodelliert. Der Beschreibungstext stammt ebenfalls aus der Dokumentation. Es wurden lediglich Umformatierungen vorgenommen. Für diesen Datensatz unterscheidet sich der Text der beiden Verknüpfungsklassen nicht. Der Goldstandard wurde hierbei auch vom Autor dieser Arbeit erstellt.

6.3. Ergebnisse Phase 1: Nomen

Diese Phase arbeitet bereits auf Nominalphrasen, weshalb die Evaluation mit derselben Verknüpfungsklasse wie Abschnitt 6.4 bewerkstelligt werden kann.

Parametrisierungen

Es wurden für diese Phase drei verschiedene Parametrisierungen mit dem naiven Ansatz verglichen. Die ersten zwei Parametrisierungen (UVR) arbeitet auf dem in Unterabschnitt 4.1.3 beschriebenen Untervektorraum, die drei anderen (Std) nutzen ihn nicht. Für die beiden UVR-Parametrisierungen wird die im Anhang in Abschnitt A.1 beschriebene Wortmenge zur Berechnung des Untervektorraums benutzt. Die Parametrisierungen $Std_{2.0}$ und $Std_{1.2}$ unterscheiden sich durch die Gewichtung des letzten Teilwortes entsprechend ihres Namens. Hier wurden die beiden ausgewählt, da sie auf den Testdaten die beiden lokalen Maxima bezüglich des F1-Maßes darstellen. $Std_{0.85}$ dient nur als Vergleichsobjekt für $UVR_{0.85}$. *Levenshtein* bezeichnet den (naiven) Vergleichsansatz für die Wortvektoren, welcher durch eine Verknüpfungsberechnung anhand der *Levenshtein-Ähnlichkeit* realisiert ist. Für einen bezüglich der Ergebnisse der Verknüpfungsberechnung möglichst optimalen Schwellenwert wurden verschiedene Werte getestet. Der Einfachkeit halber wird sich in der Evaluation auf den unten angegebenen Wert festgelegt.

	$UVR_{0.85}$	$UVR_{0.75}$	$Std_{2.0}$	$Std_{1.2}$	$Std_{0.85}$	Levenshtein
Epsilon	0,2	0,2				
Anteil_Wörter	0,65	0,65				
lastWordWeight	1,2	1,2	2,0	1,2	1,2	
Schwellenwert	0,85	0,75	0,75	0,75	0,85	0,71

Tabelle 6.1.: Parametrisierungen Phase 1

Ergebnisse: $MediaStore_1$

In diesem Datensatz wurde das nach F_1- und $F(_{\sqrt{2}})$-Maß beste Ergebnis mit der Parametrisierung $Std_{2.0}$ erreicht. Aufgrund der hohen Präzision konnte die Parametrisierung UVR das beste $F(_{\sqrt{0.5}})$-Maß-Ergebnis erzielen. Das Levenshtein-Verfahren fällt durch eine hohe Präzision und vergleichsweise niedrige Ausbeute auf.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	*0,888*	*0,4*	*0,552*	*0,632*	*0,49*
$Std_{1.2}$	0,704	**0,475**	0,567	0,606	0,533
$Std_{2.0}$	0,76	**0,475**	**0,585**	0,633	**0,543**
$UVR_{0.75}$	0,667	0,45	0,537	0,574	0,505
$UVR_{0.85}$	**0,941**	0,4	0,561	**0,649**	0,495
$Std_{0.85}$	1	0,325	0,491	0,591	0,42

Tabelle 6.2.: Ergebnisse Phase 1 mit Datensatz *$MediaStore_1$*

Ergebnisse: $MediaStore_2$

Mit der Parametrisierung $Std_{2.0}$ wurde unter den Wortvektor-Ansätzen das nach allen Metriken beste Ergebnis erzielt. In diesem Datensatz ist allerdings der naive Ansatz in allen Metriken führend. Auch hier erreicht das Levenshtein-Verfahren eine relativ hohe

Präzision, in der Ausbeute liegt es mit den Parametrisierungen $\text{Std}_{1.2}$, $\text{Std}_{2.0}$ und $\text{UVR}_{0.75}$ gleichauf. :

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	***0,5***	***0,152***	***0,233***	***0,284***	***0,198***
$\text{Std}_{1.2}$	0,389	**0,152**	0,219	0,256	0,191
$\text{Std}_{2.0}$	0,467	**0,152**	0,23	0,276	0,196
$\text{UVR}_{0.75}$	0,368	**0,152**	0,215	0,25	0,189
$\text{UVR}_{0.85}$	0,429	0,065	0,113	0,15	0,091
$\text{Std}_{0.85}$	1,0	0,043	0,083	0,12	0,064

Tabelle 6.3.: Ergebnisse Phase 1 mit Datensatz *MediaStore*$_2$

Ergebnisse: TEAMMATES

In diesem Datensatz erreicht die Parametrisierung $\text{Std}_{1.2}$ in allen Metriken das höchste Ergebnis, allerdings sehr knapp vor der Parametrisierung Std2.0. Der Unterschied in F1- und $F(\sqrt{2})$-Maß liegt unter einem Prozentpunkt. Das Levenshtein-Verfahren weist in diesem Datensatz eine vergleichsweise niedrige Präzision auf, vor allem im Hinblick auf die beiden anderen Datensätze.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	*0,75*	*0,482*	*0,587*	*0,633*	*0,547*
$\text{Std}_{1.2}$	**0,86**	**0,661**	**0,747**	**0,782**	**0,716**
$\text{Std}_{2.0}$	0,841	**0,661**	0,74	0,771	0,712
$\text{UVR}_{0.75}$	0,841	**0,661**	0,74	0,771	0,712
$\text{UVR}_{0.85}$	1,0	0,441	0,582	0,676	0,511
$\text{Std}_{0.85}$	1,0	0,357	0,526	0,625	0,455

Tabelle 6.4.: Ergebnisse Phase 1 mit Datensatz *TEAMMATES*

Interpretation

In zwei von drei Datensätzen (*MediaStore*$_1$ und *TEAMMATES*) sind Wortvektor-Ansätze in allen Metriken führend. Einzig bei *MediaStore*$_2$ schneiden Wortvektor-Ansätze schlechter ab. Dabei liegt allerdings im Vergleich *Levenshtein* zu $Std_{2.0}$ im F_1-Maß nur ein Unterschied von 0.3 Prozentpunkten vor, was auf eine um 3.3 Prozentpunkte niedrigere Präzision zurückzuführen ist. In den Datensätzen, in denen Wortvektor-Ansätze führen, sind größere Unterschiede zwischen $Std_{2.0}$ und *Levenshtein* im F_1-Maß zu verzeichnen: Bei *MediaStore*$_1$ sind es 3.3 und bei *TEAMMATES* gar 15.3 Prozentpunkte. Zwischen $Std_{1.2}$ und *Levenshtein* besteht im letzteren Datensatz sogar noch ein etwas größerer Unterschied (16 Prozentpunkte). Daher liegt die Vermutung nahe, dass die Wortvektoransätze aufgrund eines Charakteristikums, das *MediaStore*$_2$ von den anderen Datensätzen unterscheidet, in diesem Datensatz schlechter abschneiden. In Unterabschnitt 6.6.1 werden diese Unterschiede zwischen den Datensätzen näher beleuchtet.

Wie bereits angedeutet, ist $Std_{2.0}$ der $Std_{1.2}$-Parametrisierung leicht überlegen. Nur beim *TEAMMATES*-Datensatz sorgt eine leicht höhere Präzision (+1.9 Prozentpunkte) für bessere F-Maß-Ergebnisse.

Interessant ist, dass die *UVR*-Parametrisierungen im F_1-Maß durchweg leicht schlechter abschneiden als die *Std*-Parametrisierungen. Dabei ist in $MediaStore_2$ und *TEAMMATES* ein deutlich schlechteres Abschneiden von $UVR_{0.85}$ gegenüber $UVR_{0.75}$ zu sehen. Nur bei $MediaStore_1$ wirkt der höhere Schwellenwert so stark auf die Präzision ein (+27.4 Prozentpunkte), bei mäßig sinkender Ausbeute (−5 Prozentpunkte), dass das F_1-Maß um 2.4 Prozentpunkte steigt.

Aufgrund dieser insgesamt eher schlechten Ergebnisse wurde das Untervektorraum-Verfahren in den folgenden Phasen nicht weiterverwendet. Die Ergebnisse lassen allerdings nicht zwingend darauf schließen, dass Berechnungen in Untervektorräumen per se schlechter abschneiden. Möglicherweise war lediglich die Wahl der zur Bildung des Untervektorraumes verwendeten Wörter oder das Berechnungsverfahren an sich ungeeignet.

6.4. Ergebnisse Phase 2: Nominalphrasen

In Phase 2 wurden Nominalphrasen insgesamt betrachtet, welche aus mehr als einem Wort bestehen können.

Parametrisierungen

Für diese Phase wurden ebenfalls wieder drei Parametrisierungen mit dem naiven Ansatz (*Levenshtein*) verglichen. Diese haben folgende gemeinsame Grundlage: Der Schwellenwert liegt bei 0,75. Die Parametrisierung des NounLinkBuilders wurde von $Std_{2.0}$ aus Abschnitt 6.3 übernommen. Der Wortfilter besteht aus dem Wort *component*. Die Gewichte für die Einzelwort-Typen (*posWeights*) weisen allen Varianten von Nomen das Gewicht 2.0 zu.

Die PC-Parametrisierungen verwenden das in Unterabschnitt 4.2.2 erklärte Punktwolkenverfahren, wobei $PC_{no\_fallback}$ bei ungleich großen zu vergleichenden Wortmengen (zum Beispiel *HttpServletRequest* und *HttpRequest*) auf das Standardverfahren zurückfällt.

	$PC_{fallback}$	$PC_{no\_fallback}$	Std	Levenshtein
withPointCloud	true	true	false	
fallBackOnUnequal	true	false		
posWeights	w(NN*)=2.0	w(NN*)=2.0	w(NN*)=2.0	
WordFilter	{component"}	{component"}	{component"}	
lastWordWeight_P1	2,0	2,0	2,0	
Schwellenwert_P1	0,75	0,75	0,75	
Schwellenwert	0,75	0,75	0,75	0,71

Tabelle 6.5.: Parametrisierungen Phase 2

Ergebnisse: $MediaStore_1$

In diesem Datensatz ist die Parametrisierung *Std* mit Ausnahme der Präzision in allen Metri-

ken führend. Die Werte des Levenshtein-Verfahrens sind dieselben wie in Phase 1. Das gilt auch für die anderen Datensätze. Auch hier übertrifft es die Wortvektor-Parametrisierungen noch in der Präzision, wobei die Werte des Wortvektor-Verfahrens sowohl in Präzision (v.a. Std), als auch in Ausbeute verbessert werden konnten.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	***0,888***	*0,4*	*0,552*	*0,632*	*0,49*
Std	0,821	**0,575**	**0,676**	**0,719**	**0,64**
$PC_{fallback}$	0,777	0,525	0,627	0,67	0,589
$PC_{no\_fallback}$	0,488	0,5	0,494	0,492	0,496

Tabelle 6.6.: Ergebnisse Phase 2 mit Datensatz *MediaStore*$_1$

Ergebnisse: MediaStore$_2$

Auch in diesem Datensatz führt *Std* in allen Metriken mit Ausnahme der Präzision. Das Levenshtein-Verfahren liegt auch hier noch in der Präzision vorne, wenngleich auch hier eine Verbesserung der Wortvektor-Verfahren zu erkennen ist.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	***0,5***	*0,152*	*0,233*	*0,284*	*0,198*
Std	0,471	**0,174**	**0,254**	**0,3**	**0,22**
$PC_{fallback}$	0,412	0,152	0,222	0,262	0,193
$PC_{no\_fallback}$	0,318	0,152	0,206	0,233	0,184

Tabelle 6.7.: Ergebnisse Phase 2 mit Datensatz *MediaStore*$_2$

Ergebnisse: TEAMMATES

In diesem Datensatz führt *Std* in allen Metriken. $PC_{fallback}$ liegt bei der Ausbeute mit *Std* gleichauf. Das nach den Messwerten von MediaStore$_1$ und MediaStore$_2$ unerwartet schlechte Abschneiden des Levenshtein-Verfahrens in der Präzision aus Phase 1 setzt sich auch hier fort.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
Levenshtein	*0,75*	*0,482*	*0,587*	*0,633*	*0,547*
Std	**0,9**	**0,804**	**0,85**	**0,865**	**0,833**
$PC_{fallback}$	0,882	**0,804**	0,841	0,854	0,828
$PC_{no\_fallback}$	0,735	0,643	0,686	0,701	0,671

Tabelle 6.8.: Ergebnisse Phase 2 mit Datensatz *TEAMMATES*

Interpretation

In dieser Phase führen Wortvektor-Ansätze zumeist deutlich. Im Vergleich *Levenshtein* zu *Std* wurden folgende Unterschiede im F_1-Maß erreicht: Für *MediaStore*$_1$ liegt der Vorsprung von *Std* bei 12.4, für *TEAMMATES* bei 26.3 Prozentpunkten. Bei *MediaStore*$_2$ liegt er

nur bei 2.1 Prozentpunkten. Hierbei lässt sich eine Parallelität zu den Ergebnissen in Abschnitt 6.3 erkennen, in dieser Phase waren die Unterschiede unter den Datensätzen in ähnlicher Relation. Die Punktwolken-Parametrisierungen liegen immer hinter der *Std*-Parametrisierungen, mit Ausnahme des Datensatzes *TEAMMATES*. Hierbei liegen $PC_{fallback}$ und *Std* in der Ausbeute gleichauf. Besonders fällt auf, dass $PC_{fallback}$ nicht einmal *Levenshtein* im F_1-Maß überholen kann, was an einer deutlich niedrigeren Präzision bei gleicher oder, im Falle von $MediaStore_1$, leicht höheren Ausbeute liegt.

Ein Grund für diese teilweise sehr viel niedrigere Präzision könnte darin liegen, dass beim Vergleich von ungleich großen Punktwolken zu viele Informationen verloren gehen. Dadurch werden Falsch-Positive wahrscheinlicher. Hier einige Beispiele solcher Falsch-Positive aus $MediaStore_1$:

- (the user – UserManagement): Hier wird, da *the* als Artikel herausgefiltert wird, nur *user* mit *User* verknüpft.
- (a download – DownloadLoadBalancer): Hier geschieht dasselbe mit *download* und *Download*.

Falsch-Positive Verknüpfungen dieser Art können durch den *Fallback*-Parameter eliminiert werden, da hier das Punktwolkenverfahren nur dann angewendet wird, wenn die Punktwolken im Voraus schon gleich groß sind.

6.5. Ergebnisse Phase 3: Relationserkennung über Verbalphrasen

Parametrisierungen

Zur Berechnung der Verbalphrasen-Verknüpfungen wird auf vorberechnete Nominalphrasen-Verknüpfungen zurückgegriffen. Hierfür wird für *NoWV*, *Levenshtein* und *WV* die *Std*-Parametrisierung aus Abschnitt 6.4 verwendet, da diese in ihrer Phase am besten abgeschnitten hat. In dieser Phase wird der Levenshtein-Vergleichsansatz anders berechnet als zuvor: Sowohl die Parametrisierung *Levenshtein* als auch *WV* basieren auf einer Filterung der Ergebnisse der *NoWV*-Parametrisierung anhand des jeweiligen Ähnlichkeitsmaßes und eines Schwellenwerts. Daher können diese Parametrisierungen im Vergleich zu *NoWV* nur eine Verbesserung der Präzision, nicht jedoch der Ausbeute erzielen. Dass dies erfolgreich geschieht, ist im $TEAMMATES_V$-Datensatz zu sehen.

	WV	Levenshtein	NoWV
Phase2-Parametrisierung	Std	Std	Std
Schwellenwert	0,4	0,15	

Tabelle 6.9.: Parametrisierungen Phase 3

Ergebnisse: $MediaStore_{1V}$

In diesem Datensatz sind aufgrund fehlender Relationsbezeichnungen keine Ähnlichkeitsberechnungen möglich. Daher ist die Parametrisierung *NoWV* führend.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
NoWV	0,75	0,545	0,632	0,667	0,6
Levenshtein	*0,0*	*0,0*	*0,0*	*0,0*	*0,0*
WV	0,0	0,0	0,0	0,0	0,0

Tabelle 6.10.: Ergebnisse Phase 3 mit Datensatz $MediaStore_{1V}$

Ergebnisse: $TEAMMATES_V$

In diesem Datensatz sind die Relationsbezeichnungen vorhanden, weshalb die Parametrisierungen *WV* und *Levenshtein* anwendbar sind. Diese erzielen in allen Metriken dasselbe Ergebnis. Gleichauf sind alle die beiden Parametrisierungen bei der Ausbeute.

	Präzision	Ausbeute	F_1-Maß	$F_{\sqrt{0.5}}$-Maß	$F_{\sqrt{2}}$-Maß
NoWV	0,667	**0,8**	0,747	0,706	0,75
Levenshtein	**0,889**	**0,8**	**0,842**	**0,857**	**0,828**
WV	**0,889**	**0,8**	**0,842**	**0,857**	**0,828**

Tabelle 6.11.: Ergebnisse Phase 3 mit Datensatz $TEAMMATES_{1V}$

Interpretation

Für den Datensatz $MediaStore_1$ ist kein Vergleich zu einem Referenzverfahren möglich, da, wie bereits gesagt, wegen der fehlenden Relationsbezeichnungen im Modell keine Ähnlichkeitsberechnungen möglich sind. Hier kann daher nur das Verfahren, gemessen am Goldstandard, interpretiert werden. Die Ursache für die auffällig niedrige Ausbeute liegt größtenteils darin, dass Verknüpfungen für nichtrelationale Entitäten in den vorigen Phasen nicht gefunden wurden. In vier der fünf Falsch-Negativen ist die Modellkomponente *DB* durch *database* im Satz vertreten. Diese Verknüpfung konnte von den vorherigen Verfahren nicht gefunden werden. Daher können alle Relationen, die diese Komponente enthalten, nicht gefunden werden. Auch Einbuße in der Präzision lassen sich in ähnlicher Weise auf die Vorgängerphasen zurückführen.

Falsch-Positive, die nicht als „Folgefehler" einschätzbar sind, können daran liegen, dass ein Verb einer Relation zugeordnet wird, welches die Relation offensichtlich nicht beschreibt. Diese sind die für diese Phase interessanteren Fehler. In den Ergebnissen des Datensatzes $TEAMMATES_V$ kann man im Vergleich *NoWV* zu *WV* und *Levenshtein* sehen, dass die letzteren Verfahren einige Fehler dieser Kategorie beseitigen. Diese Fehler können nur aus der Klasse *Falsch-Positiv* kommen, da die beiden Vergleichsverfahren *WV* und *Levenshtein* konzeptionell gesehen lediglich die Ergebnisse von *NoWV* filtern, womit sie potenzielle Falsch-Positive eliminieren. Allerdings können dadurch auch Wahr-Positive herausgefiltert werden, was hier allerdings nicht der Fall war.

Interessant ist, dass *Levenshtein* und *WV* gleichauf liegen. Dies kann man auf den vergleichsweise kleinen Datensatz zurückführen. So gab es bei *NoWV* nur vier Falsch-positive und bei *Levenshtein* und *WV* nur ein einziges. Damit liegen für einen Vergleich zwischen *Levenshtein* und *WV* zu wenig Daten vor, um daraus einen Schluss ziehen zu können.

6.6. Insgesamte Interpretation

Generell lässt sich sagen, dass das entwickelte Wortvektor-Verfahren meistenfalls besser als das Referenzverfahren, die Ähnlichkeitsberechnung mittels Levenshtein-Distanz, ist. Anhand von Phase 1 kann man das Levenshtein- und das Wortvektor-Verfahren am besten vergleichen, da hier beide auf Wort-Ebene arbeiten. In Phase 2 (Abschnitt 6.3) ist dies nicht mehr der Fall und in Phase 3 (Abschnitt 6.5) ist die Datenbasis, wie bereits beschrieben, zu klein, um in dieser Hinsicht Schlüsse ziehen zu können. In Phase 1 sind, wie bereits beschrieben, mäßig große Vorsprünge des Wortvektorverfahrens (Parametrisierung $Std_{2.0}$) im Vergleich zu *Levenshtein* zu erkennen: −0.3, +3.3 und +15.3 Prozentpunkte bei den drei Datensätzen im F_1-Maß, im Durchschnitt also +6.1 Prozentpunkte.

In absoluter Betrachtung (Vergleich zum Goldstandard) sind die Ergebnisse je nach Datensatz mäßig bis gut (*MediaStore*$_1$ und *TEAMMATES*) und relativ schlecht (*MediaStore*$_2$). Bei *MediaStore*$_2$ beeinflusst die sehr geringe Ausbeute (Maximum: 0.174, aus *Std*, Phase 2) die F-Maße stark negativ.

Für den praktischen Einsatz des in dieser Arbeit erarbeiteten Verfahren lässt sich sagen, dass im Hinblick auf das F_1-Maß aus Phase 1 die $Std_{2.0}$-, aus Phase 2 die *Std*- und aus Phase 3 je nachdem, ob aussagekräftige Relationsbezeichnungen vorliegen oder nicht, die *WV*- oder die *NoWV*-Parametrisierung benutzt werden sollte.

6.6.1. Bedrohungen für die Validität

In diesem Unterabschnitt sollen die Evaluationsdurchführung kritisch reflektiert und somit die möglichen Bedrohungen für die Validität der Ergebnisse ausgeführt werden.

Datensätze

Einen großer Risikofaktor stellen die verwendeten Datensätze dar. Allein an den großen Unterschieden in den Absolutwerten der Parametrisierungen zwischen den einzelnen Datensätzen (zwischen *MediaStore*$_2$ und *TEAMMATES* meist fast mehr als dreimal so hohe Werte im F_1-Maß) lässt sich erahnen, dass gewisse Risiken nicht auszuschließen sind. Dieser große Unterschied in den absoluten Zahlen rührt in diesem Fall daher, dass der *TEAMMATES*-Datensatz einen Originaltext enthält, der nicht verändert wurde, und daher vergleichsweise weniger schwer zu findende Verknüpfungen enthält. Für *MediaStore*$_2$ wurde explizit ein Text so geschrieben/ angepasst, dass er möglichst vielfältige, nichttriviale Verknüpfungen darbietet, wodurch das Ergebnis naturgemäß schlechter wird. *MediaStore*$_1$ ist in dieser Hinsicht gewissermaßen ein Hybrid, da bei der Modifikation manche Teile im Original belassen und andere Teile im Sinne der Erschwerung der Verknüpfungserkennung verändert wurden.

Ein weiterer Risikofaktor besteht darin, dass die in einem Text auftauchenden Nominal- und Verbalphrasen, die es zu verknüpfen gilt nicht gleichverteilt vorkommen. Einige, zentralere Entitäten des Architekturmodells, wie *HttpServletRequest* bei *TEAMMATES*, werden naturgemäß öfter referenziert als andere. Wenn dieses im Text oft mit derselben Phrase referenziert wird, steigt dessen Gewicht im Verhältnis zu den anderen vorkommenden Phrase-Entität-Verknüpfungen, was sich Evaluationsergebnis positiv oder negativ niederschlägt, je nach dem, ob die Verknüpfung erkannt wurde oder nicht.

Dem Risiko, dass durch die Selbst-Erstellung des Evaluationsdatensatzes eine Verzerrung zugunsten eines positiveren Ergebnisses stattfinden kann, wurde durch den Austausch der *MediaStore*-Datensätze zwischen dem Autor dieser Arbeit und der Autorin einer parallelen Arbeit und durch die Verwendung eines Originaltextes im Falle von *TEAMMATES* entgegengewirkt. Ein Restrisiko, beispielsweise was die Erstellung des Goldstandards betrifft, bleibt dennoch, und ist schwer zu vermeiden.

Ein weiterer Punkt ist die Größe der Datensätze. Je kleiner ein Datensatz ist, desto un-repräsentativer wird er. Das ist vor allem im *TEAMMATES*-Verbalphrasen-Datensatz (TEAMMATES_V) zu sehen, welcher aus nur zehn Verknüpfungen besteht und sollte, wie auch schon in der Interpretation erwähnt, bei Sicht dieser Ergebnisse beachtet werden.

Rahmenwerk

Neben den Datensätzen stellt das verwendete Rahmenwerk einen kleineren Störfaktor dar. In Abbildung 6.2 ist zu sehen, wie die Verbalphrase *processes* fälschlicherweise als Nominalphrase klassifiziert wird. Aufgrund dessen kann für diese Verbalphrase keine Verknüpfung gefunden werden, was sich im Evaluationsergebnis niederschlägt.

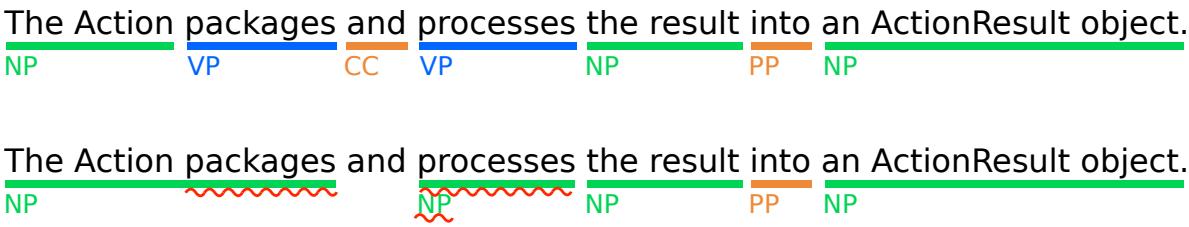

Abbildung 6.2.: Beispiel einer fehlerhaften Text-Klassifikation durch PARSE. Oben: der Soll-Zustand. Unten: Das PARSE-Ergebnis

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Verfahren entwickelt, mit dem sich unter Verwendung von Wortvektoren Verknüpfungen von Text- und Softwarearchitekturmodell-Entitäten finden lassen. Mithilfe solcher Verknüpfungen können beispielsweise Änderungen an einem der Artefakte besser auf das andere übertragen werden. Auch kann eine mithilfe von Verknüpfungen verbesserte Darstellung einer Architekturdokumentation das Verständnis potenziell erleichtern. Wohlrab et al. haben 2019 in einer Umfrage [31] festgestellt, dass Inkonsistenzen in Wortlaut und Sprache relativ häufig seien und auch einen negativen Einfluss haben. Inkonsistenzen dieses Typs können mit dem in dieser Arbeit entwickelten Verfahren gefunden werden. In [11, 12] wurde eine Vision vorgestellt, die dem Alterungsprozess von Software und damit den entstehenden Inkonsistenzen zwischen Artefakten entgegenwirken soll. Auch zu dieser Problemstellung kann diese Arbeit mit der Bereitstellung von Verknüpfungen zwischen dem Artefakt Dokumentationstext und dem Artefakt Softwarearchitektur-Modell einen Beitrag leisten.

Die Entwicklung des Verknüpfungsberechnungs-Verfahrens lief wie folgt ab: Zunächst wurden die zu findenden Verknüpfungen in zwei Gruppen unterteilt: Nominalphrasen und Verbalphrasen. Dabei wurden Nomen zunächst für sich stehend, und später im Kontext ihrer Nominalphrase betrachtet. Für Nomen, Nominalphrasen und Verbalphrasen wurde anschließend jeweils ein Verfahren entwickelt, um aus diesen Arten von Satzteilen und den Architekturmodell-Entitäten-Bezeichnern möglichst aussagekräftige Wortvektoren zu generieren. Mithilfe dieser Wortvektoren können Architekturmodell- und Text-Entitäten verglichen und verknüpft werden.

Anhand einer Fallstudie eines künstlichen und eines realen Falles konnte festgestellt werden, dass das entwickelte Wortvektor-Verfahren zur Verknüpfungsberechnung das Referenzverfahren, einen String-Ähnlichkeitsvergleich mithilfe der Levenshtein-Distanz in Präzision, Ausbeute und F_1-Maß überbieten kann. So konnte für Nominalphrasen ein durchschnittlicher Vorsprung von 13.6 Prozentpunkten im F_1-Maß erzielt werden. Bei dem realen Fall war der Vorsprung regelmäßig größer als bei den künstlichen Fallstudien, da bei Letzteren darauf geachtet wurde, nichttriviale Inkonsistenzen einzubauen, um das Verfahren und dessen Grenzen besser testen zu können.

Es sollte weiterhin beachtet werden, dass im Rahmen dieser Arbeit ein Wortvektor-Modell verwendet wurde, welches auf allgemeinen Daten trainiert wurde. Somit kann für zukünftige Arbeiten durchaus Potenzial darin gesehen werden, das entwickelte Verfahren durch das Training eines Wortvektor-Modells mit spezifischer auf den softwaretechnischen Bereich zugeschnittenen Trainingsdaten zu verbessern. Auch kann durch die Verwendung von verschiedenen Instanzen verschiedener Wortvektor-Verfahren das Resultat möglicherweise verbessert und ein Vergleich verschiedener Modellparameter wie Trainingsdatensätze oder Dimensionalität des Vektorraumes hergestellt werden.

In dieser Arbeit wurde bezüglich Untervektorräumen bzw. Hervorhebung einzelner Dimension nur ein einzelnes Verfahren mit einer Parametrisierung (eine Menge an Wörtern) evaluiert. Eine nähere Untersuchung solcher Räume, auch mit möglichen anderen Verfahren, kann das Verständnis über Wortvektor-Räume, und bei Erfolg möglicherweise auch andere Wortvektor-basierte Verfahren verbessern.

Auch wurden kleinere Limitationen durch den im PARSE-Rahmenwerk enthaltenen Zergliederer festgestellt. So wurden teilweise, wenn auch eher selten, grammatikalisch komplexere Sätze falsch klassifiziert, was eine Fehlerquelle für eine weitere Verarbeitung darstellt. Eine Verbesserung dieses Rahmenwerks käme somit auch dem in dieser Arbeit entwickelten Verfahren zugute.

Literatur

[1] Piotr Bojanowski u. a. „Enriching Word Vectors with Subword Information“. In: *arXiv:1607.04606 [cs]* (Juli 2016). arXiv: 1607.04606.

[2] Danqi Chen und Christopher Manning. „A Fast and Accurate Dependency Parser using Neural Networks“. en. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, S. 740–750. DOI: 10.3115/v1/D14-1082.

[3] *Determinativ*. https://www.wortbedeutung.info/Determinativ/. Aufgerufen am 07.11.2019.

[4] Jacob Devlin u. a. „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding“. In: *arXiv:1810.04805 [cs]* (Okt. 2018). arXiv: 1810.04805.

[5] *Duden: Phrase*. https://www.duden.de/rechtschreibung/Phrase#Bedeutung-2a. Aufgerufen am 07.11.2019.

[6] *Flask - a lightweight WSGI web application framework*. https://palletsprojects.com/p/flask/. Aufgerufen am 07.11.2019.

[7] O. C. Z. Gotel und C. W. Finkelstein. „An analysis of the requirements traceability problem“. In: *Proceedings of IEEE International Conference on Requirements Engineering*. Apr. 1994, S. 94–101.

[8] J. Guo, J. Cheng und J. Cleland-Huang. „Semantically Enhanced Software Traceability Using Deep Learning Techniques“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Mai 2017, S. 3–14. DOI: 10.1109/ICSE.2017.9.

[9] Zellig S. Harris. „Distributional Structure“. en. In: *WORD* 10.2-3 (Aug. 1954), S. 146–162. ISSN: 0043-7956, 2373-5112. DOI: 10.1080/00437956.1954.11659520.

[10] *Jenkins core: TcpSlaveAgentListener*. https://github.com/jenkinsci/jenkins/blob/master/core/src/main/java/hudson/TcpSlaveAgentListener.java. Aufgerufen am 07.11.2019.

[11] Jan Keim und Anne Koziolek. „Towards Consistency Checking between Software Architecture and Informal Documentation“. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. März 2019, S. 250–253. DOI: 10.1109/ICSA-C.2019.00052.

[12] Jan Keim, Yves Schneider und Anne Koziolek. „Towards Consistency Analysis Between Formal and Informal Software Architecture Artefacts“. In: *Proceedings of the 2nd International Workshop on Establishing a Community-Wide Infrastructure for Architecture-Based Software Engineering*. ECASE '19. Montreal, Quebec, Canada: IEEE Press, Mai 2019, S. 6–12. DOI: 10.1109/ECASE.2019.00010. URL: https://doi.org/10.1109/ECASE.2019.00010.

[13] Tomas Mikolov, Wen-tau Yih und Geoffrey Zweig. „Linguistic Regularities in Continuous Space Word Representations.“ In: *HLT-NAACL*. 2013, S. 746–751.

[14] Tomas Mikolov u. a. „Efficient Estimation of Word Representations in Vector Space“. In: *arXiv:1301.3781 [cs]* (Jan. 2013). arXiv: 1301.3781.

[15] G. Neumann u. a. „An Information Extraction Core System for Real World German Text Processing“. en. In: *arXiv:cmp-lg/9706023* (Juni 1997). arXiv: cmp-lg/9706023. URL: http://arxiv.org/abs/cmp-lg/9706023 (besucht am 31. 10. 2019).

[16] Natalya Fridman Noy und Mark Musen. „Algorithm and Tool for Automated Ontology Merging and Alignment“. en. In: (), S. 6.

[17] Makbule Gulcin Ozsoy. „From Word Embeddings to Item Recommendation“. en. In: *arXiv:1601.01356 [cs]* (Jan. 2016). arXiv: 1601.01356.

[18] D. L. Parnas. „Software aging“. In: *Proceedings of 16th International Conference on Software Engineering*. Mai 1994, S. 279–287. DOI: 10.1109/ICSE.1994.296790.

[19] Jeffrey Pennington, Richard Socher und Christopher Manning. „Glove: Global Vectors for Word Representation“. en. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, S. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: http://aclweb.org/anthology/D14-1162 (besucht am 06. 06. 2019).

[20] *Programmieren in natürlicher Sprache*. https://ps.ipd.kit.edu/176_453.php. Aufgerufen am 07.11.2019.

[21] Patrick Rempel und Parick Mäder. „Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality“. In: *IEEE Transactions on Software Engineering* 43.8 (Aug. 2016), S. 777–797. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2016.2622264.

[22] Ralf H. Reussner u. a. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016. ISBN: 026203476X, 9780262034760.

[23] Ralf Reussner u. a. *The Palladio Component Model*. Englisch. Techn. Ber. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 S. DOI: 10.5445/IR/1000022503.

[24] James Rumbaugh, Ivar Jacobson und Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[25] Aliaksei Severyn und Alessandro Moschitti. „UNITN: Training Deep Convolutional Neural Network for Twitter Sentiment Classification“. In: *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. Denver, Colorado: Association for Computational Linguistics, 2015, S. 464–469. DOI: 10.18653/v1/S15-2079.

[26] Gerd Stumme und Alexander Maedche. „FCA-MERGE: Bottom-Up Merging of Ontologies“. en. In: (), S. 6.

[27] Gerd Stumme und Alexander Maedche. „Ontology Merging for Federated Ontologies on the Semantic Web“. en. In: (), S. 9.

[28] *TEAMMATES feedback management tool for education - website*. original-date: 2014-05-02T07:43:00Z. Juni 2019. URL: https://github.com/TEAMMATES/teammates/blob/master/docs/design.md (besucht am 16. 06. 2019).

[29] Yuan Tian, David Lo und Julia Lawall. „SEWordSim: software-specific word similarity database". en. In: *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. Hyderabad, India: ACM Press, 2014, S. 568–571. ISBN: 978-1-4503-2768-8. DOI: 10.1145/2591062.2591071. (Besucht am 07. 06. 2019).

[30] Sebastian Weigelt und Walter F. Tichy. „ProNat: An Agent-based System Design for Programming in Spoken Natural Language". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, S. 819–820. URL: http://dl.acm.org/citation.cfm?id=2819009.2819183.

[31] Rebekka Wohlrab u. a. „Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects". en. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. Hamburg, Germany: IEEE, März 2019, S. 151–160. ISBN: 978-1-72810-528-4. DOI: 10.1109/ICSA.2019.00024. URL: https://ieeexplore.ieee.org/document/8703919/ (besucht am 31. 10. 2019).

[32] Xin Ye u. a. „From word embeddings to document similarities for improved information retrieval in software engineering". In: *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, 2016, S. 404–415. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884862.

A. Anhang

A.1. Parametrisierungen: Ergänzung

Die Untervektorraum-Parametrisierungen der Phase 1 enthalten eine für die Berechnung des Untervektorraumes benutzte Wortmenge W. Diese besteht aus den folgenden Wörtern:

database, file, facade, parse, sql, software, server, client, package, application, program, action, reader, xml, processing, strategy, debug, refactor, algorithm, palladio, uml, String, int, float, java, checkout, api, authenticator, processor, scheduler, cache, callable, link, core, hash, hashmap, http, agent, proxy, computer, cloud, connect, list, listener, pointer, descriptor, adapter, container, decorator, singleton, multiton, factory, abstract, state, null, noop, recursive, iterative, template

A.2. Evaluationsergebnisse

In diesem Abschnitt sind die der Berechnung von Ausbeute und Präzision zugrundeliegenden Messwerte zu sehen. Dazu gehören Wahr-Positive, Falsch-Positive und Falsch-Negative. Die Aufteilung erfolgt hier nach Datensätzen.

A.2.1. MediaStore$_1$

	Parametrisierung	**Wahr-Positive**	**Falsch-Positive**	**Falsch-Negative**
Phase 1				
	Levenshtein	16	2	24
	$\text{Std}_{1.2}$	19	8	21
	$\text{Std}_{2.0}$	19	6	21
	$\text{UVR}_{0.75}$	18	9	22
	$\text{UVR}_{0.85}$	16	1	24
	$\text{Std}_{0.85}$	13	0	27
Phase 2				
	Levenshtein	16	2	24
	Std	23	5	17
	$\text{PC}_{fallback}$	21	6	19
	$\text{PC}_{no\_fallback}$	20	21	20

Tabelle A.1.: Wahr-Positive, Falsch-Positive und Falsch-Negative von MediaStore$_1$

A.2.2. MediaStore$_2$

	Parametrisierung	Wahr-Positive	Falsch-Positive	Falsch-Negative
Phase 1				
	Levenshtein	7	7	39
	$\text{Std}_{1.2}$	7	11	39
	$\text{Std}_{2.0}$	7	8	39
	$\text{UVR}_{0.75}$	7	12	39
	$\text{UVR}_{0.85}$	3	4	43
	$\text{Std}_{0.85}$	2	0	44
Phase 2				
	Levenshtein	7	7	39
	Std	8	9	38
	$\text{PC}_{fallback}$	7	10	39
	$\text{PC}_{no\_fallback}$	7	15	39

Tabelle A.2.: Wahr-Positive, Falsch-Positive und Falsch-Negative von MediaStore$_2$

A.2.3. TEAMMATES

	Parametrisierung	Wahr-Positive	Falsch-Positive	Falsch-Negative
Phase 1				
	Levenshtein	27	9	29
	$\text{Std}_{1.2}$	37	6	19
	$\text{Std}_{2.0}$	37	7	19
	$\text{UVR}_{0.75}$	37	7	19
	$\text{UVR}_{0.85}$	23	0	33
	$\text{Std}_{0.85}$	20	0	36
Phase 2				
	Levenshtein	27	9	29
	Std	45	5	11
	$\text{PC}_{fallback}$	45	6	11
	$\text{PC}_{no\_fallback}$	36	13	20

Tabelle A.3.: Wahr-Positive, Falsch-Positive und Falsch-Negative von TEAMMATES

A.2.4. MediaStore$_{1V}$

	Parametrisierung	Wahr-Positive	Falsch-Positive	Falsch-Negative
Phase 3				
	NoWV	6	2	5
	Levenshtein	0	0	11
	WV	0	0	11

Tabelle A.4.: Wahr-Positive, Falsch-Positive und Falsch-Negative von MediaStore$_{1V}$

A.2.5. TEAMMATES$_V$

	Parametrisierung	Wahr-Positive	Falsch-Positive	Falsch-Negative
Phase 3				
	NoWV	8	4	2
	Levenshtein	8	1	2
	WV	8	1	2

Tabelle A.5.: Wahr-Positive, Falsch-Positive und Falsch-Negative von TEAMMATES$_V$