Ein Ansatz zur Traceability Link Recovery für natürlichsprachliche Software-Dokumentation und Quelltext

Bachelor's Thesis von

Dennis Steinbuch

an der Fakultät für Informatik
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

Erstgutachter:	Prof. Dr. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf Reussner
Betreuender Mitarbeiter:	M.Sc. Dominik Fuchß
Zweiter betreuender Mitarbeiter:	M.Sc. Sophie Corallo

19. Juni 2023 – 19. Oktober 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, und weder ganz oder in Teilen als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet.

Wimsheim, 19.10.2023

. . . . . .

(Dennis Steinbuch)

Zusammenfassung

Wartbarkeit spielt eine zentrale Rolle für die Langlebigkeit von Softwareprojekten. Ein wichtiger Teil der Wartbarkeit besteht darin, dass die natürlichsprachliche Dokumentation des Quelltextes einen guten Einblick in das Projekt und seinen dazugehörigen Quelltext liefert. Zur besseren Wartbarkeit dieser beiden Software-Artefakte besteht die Aufgabe dieser Arbeit darin, Verbindungen zwischen den Elementen dieser beiden Artefakte aufzubauen. Diese Verbindungen heißen Trace Links und können für verschiedene Zwecke der Wartbarkeit genutzt werden. Diese Trace Links ermöglichen zum Beispiel die Inkonsistenzerkennung zwischen den beiden Software-Artefakten oder können auch für verschiedene Analysen benutzt werden. Um diese Trace Links nachträglich aus den beiden Software-Artefakten natürlichsprachlicher Dokumentation und Quelltext zu gewinnen, wird das bereits bestehende ArDoCo Framework benutzt und auf das Software-Artefakt Quelltext erweitert. Ebenfalls werden ArDoCos bestehende Entscheidungskriterien auf den neuen Kontext angepasst. Der neuartige Kontext führt zu Herausforderungen bezüglich der Datenmenge, die durch neue Entscheidungskriterien adressiert werden. Dabei zeugen die Ergebnisse dieser Arbeit eindeutige von Potenzial, weswegen weiter darauf aufgebaut werden sollte.

Inhaltsverzeichnis

Zusammenfassung **i**

1 Einleitung **1**

2 Grundlagen **3**
2.1 Nachverfolgbarkeitsverbindungen . . . 3
2.2 Softwaremodelle . . . 4
2.2.1 Metamodell . . . 4
2.2.2 Codemodell . . . 4
2.3 Architecture Documentation Consistency Framework . . . 6
2.3.1 ArDoCo Heuristiken . . . 8
2.3.2 Wortähnlichkeitsmetriken . . . 10

3 Verwandte Arbeiten **13**

4 Architekturelle Heuristiken von ArDoCo **15**
4.1 Auswirkungen des neuen Kontexts auf ArDoCo . . . 15
4.2 ArDoCo Anpassungen . . . 17
4.3 Anpassbare Parameter in ArDoCo Heuristiken . . . 19
4.4 Durchgeführte Anpassungen an den Heuristiken . . . 22

5 Heuristiken für die Traceability Link Recovery von Quelltext **25**
5.1 Heuristiken aus verwandter Literatur . . . 25
5.2 Neue Informanten . . . 26
5.2.1 Hinzufügen nicht gefundener NounMappings . . . 26
5.2.2 Filtern der gefundenen Textelementen . . . 28
5.2.3 Filtern mittels Wortähnlichkeit der Namen . . . 29
5.2.4 Nach Ordnerstruktur filtern . . . 29
5.2.5 Filtern anhand Pre- und Suffix . . . 30
5.2.6 Filtern mittels inkorrekter Klassen . . . 30
5.2.7 Filtern durch allgemeine Informatik Begriffe . . . 31
5.2.8 Filtern durch Konfidenzwert . . . 32

6 Evaluation **33**
6.1 Goldstandard . . . 33
6.2 Evaluationsmetriken . . . 35
6.3 Evaluation bestehender ArDoCo Heuristiken . . . 36
6.4 Evaluation angepasster ArDoCo Heuristiken . . . 41
6.5 Evaluation mit neuen Heuristiken . . . 44

6.6 Evaluation ohne Goldstandard 48
6.7 Gefährdung der Validität 49

7 Zusammenfassung **51**

Literatur **53**

Abbildungsverzeichnis

2.1 Beispiel Artefakte NLSAD und Quelltext von Teastore [9] 3
2.2 QuelltextTransformation 5
2.3 Beispiel für die Umwandlung von Java-Quelltext in das Codemodell anhand der UML Notation am Projekt Teammates[9] 5
2.4 ArDoCo Pipeline, basierend auf Keim et al. 2023 [10] 7
2.5 Auflistung von Beispielsätzen einer NLSAD 8

4.1 Erweiterung von ArDoCo 15
4.2 Paket Diagramm der Anpassungen an ArDoCo 17
4.3 Vereinfachte Version der Codemodell-Architektur, basierend auf Telge 2023 [20] 18

5.1 ArDoCo Architektur mit neuen Agenten 26
5.2 CodeModelAgent mit seinen Informanten 27

Tabellenverzeichnis

2.1 Levenshtein-Distanz Beispiel 11

4.1 Sinvolle bestehende Heuristiken zum Anpassen 19
4.2 Wortsähnlichkeitsmetriken und ihre Paramter 21
4.3 Verbesserungen durch Jaro-Winkler und Levenshtein Schwellwert Erhöhung 22

6.1 Ergebnis mit bestehenden Heuristiken ohne gekürzten Pfad 37
6.2 Ergebnis mit bestehenden Heuristiken mit gekürzten Pfad 38
6.3 Anzahl der klassifizierten Trace Links mit bestehenden Heuristiken . . . 39
6.4 Ergebnis mit angepassten Heuristiken 42
6.5 Anzahl der klassifizierten Trace Links mit angepassten Heuristiken . . . 44
6.6 Verbesserungen der einzelnen neue Informanten 45
6.7 Verbesserungen durch neue Informanten 46
6.8 Ergebnis mit neuen Heuristiken 47
6.9 Anzahl der klassifizierten Trace Links mit neuen Heuristiken 47

Abkürzungsverzeichnis

TL Trace Link

TLR Trace Link Recovery

NLSAD natürlichsprachliche Softwarearchitektur Dokumentation

NLSD natürlichsprachliche Software Dokumentation

SAM Softwarearchitektur Modell

KDM Knowledge Discovery Metamodel

NLP Natural Language Processing

ArDoCo Architecture Documentation Consistency

VSM Vector Space Model

IR Information Retrieval

FET fixture element type

LSI Latent Semantic Indexing

TP true positive

FP false positive

FN false negative

TN true negative

1 Einleitung

In der Entwicklung von Softwareprojekten sind diverse Artefakte wie Architekturmodell, Quelltext oder auch die Dokumentation fester Bestandteil. Dabei stehen die Artefakte und deren Elemente in Verbindung. So werden Klassen, Schnittstellen oder auch Pakete aus dem Quelltext in natürlichsprachlichen Sätzen in der Dokumentation beschrieben und ihre Funktion erläutert. Um bei diesen Verbindungen nicht den Überblick zu verlieren, ist das Erstellen von Trace Links zwischen den Artefakten und deren Elementen sinnvoll. Dabei sind Trace Links spezifizierte Verbindung zwischen Paaren von Software-Artefakten oder seinen Elementen [5]. Außerdem haben diese Trace Links noch mehr Nutzen, wie zum Beispiel die Inkonsistenzerkennung oder die Auswirkungsanalyse [7].

An großen Softwareprojekten arbeiten typischerweise mehrere Entwickler gleichzeitig an den verschiedenen Artefakten. Dabei ist nicht zu verhindern, dass Inkonsistenzen jeglicher Art auftauchen. „Die meisten Entwickler betrachten Inkonsistenz als etwas Unerwünschtes, das man möglichst vermeiden sollte." [15]. Jedoch gibt es verschiedene Arten der Inkonsistenz, Inkonsistenzen in den Software-Artefakten selbst und Inkonsistenzen zwischen verschiedenen Artefakten. „Manchmal ist [außerdem] der Aufwand zur Behebung einer Inkonsistenz wesentlich größer als das Risiko, dass die Inkonsistenz nachteilige Folgen haben wird." [15]. Das ist deshalb möglich, da die Inkonsistenzen teilweise auf grundlegende Designentscheidungen zurückzuführen sind, welche getroffen werden müssen [15]. Jedoch müssen diese Inkonsistenzen erstmal gefunden werden, was manuell durchgeführt aufwendig, teuer, aber vor allem auch fehleranfällig ist. Somit ist es sinnvoll, die Gewinnung der Trace Links zu automatisieren.

Bestehende Literatur bezieht sich zum größten Teil auf Anforderungen und Quelltext als Artefakte [4] für die Gewinnung von Trace Links. Jedoch wurde noch wenig auf die Trace Links zwischen natürlichsprachliche Software Dokumentation (NLSD) und dem Quelltext eingegangen. Auch diese Trace Links bringen nutzbare Vorteile mit sich. Dadurch versteht man potenziell das Programm besser, kann es besser warten, eine Impact Analyse damit ausführen und Trace Links können auch als gutes Fundament zur Wiederverwendung bestehender Software dienen [4]. Wegen diesen Vorteilen ist das Ziel der Arbeit das automatisierte Gewinnen von Trace Links zwischen den beiden Artefakten Quelltext und der dazugehörigen NLSD.

Um das Vorhaben umzusetzen, wird das Projekt ArDoCo [8] als Grundlage genutzt. Architecture Documentation Consistency (ArDoCo) dient deshalb als Grundlage, da es bereits erfolgreich Trace Links zwischen den Artefakten Software Architekturmodell und der natürlichsprachlichen Software Dokumentation (NLSD) aufbaut. ArDoCo wird somit angepasst, um in dieser Arbeit Trace Links zu gewinnen. Damit ArDoCo möglichst effizient, ohne große Änderungen genutzt werden kann, muss der Quelltext jedoch erstmals in ein Modell umgewandelt werden. Dies ist deshalb relevant, da ArDoCo als Eingabe die natürlichsprachliche Softwarearchitektur Dokumentation (NLSAD) und ein Modell

entgegennimmt. Die Arbeit von Telge [20] hilft dabei den Quelltext in ein entsprechendes Codemodell zu transformieren, welche als Eingabe benutzt wird. Die Arbeit von Telge setzt sich dabei umfassend mit dem Thema auseinandergesetzt. Somit wird in dieser Bachelorarbeit ArDoCo um die Implementierung der Quelltexttransformation in ein Codemodell erweitert. Nach erfolgreichem Abschluss der Arbeit kann dadurch ArDoCo zwischen drei Software-Artefakten Trace Links gewinnen.

2 Grundlagen

Dieses Kapitel behandelt die benötigten Grundlagen zum Verstehen dieser Bachelorarbeit. Abschnitt 2.1 erörtert die Trace Links, welche in der Arbeit erstellt werden. Der darauffolgende Abschnitt 2.2 beschreibt die für die Arbeit relevanten Softwaremodelle, das Metamodell und das Codemodell [20]. Der letzte Abschnitt 2.3 befasst sich mit ArDoCo und seinem Vorgehen der TLR. Dazu werden ebenfalls seine Heuristiken erläutert und auf seine genutzten Wortähnlichkeitsmetriken eingegangen.

2.1 Nachverfolgbarkeitsverbindungen

Ein Trace Link ist eine spezifizierte Verbindung zwischen einem Quell- und einem Zielartefakt, diese Verbindung ist bidirektional [5]. Dabei kann der Trace Link nicht nur zwischen den Artefakten selbst bestehen, sondern auch zwischen seinen Elementen. Die in diesem Abschnitt beschriebenen Grundlagen basieren auf dem Buch von Cleland-Huang et al. [5].

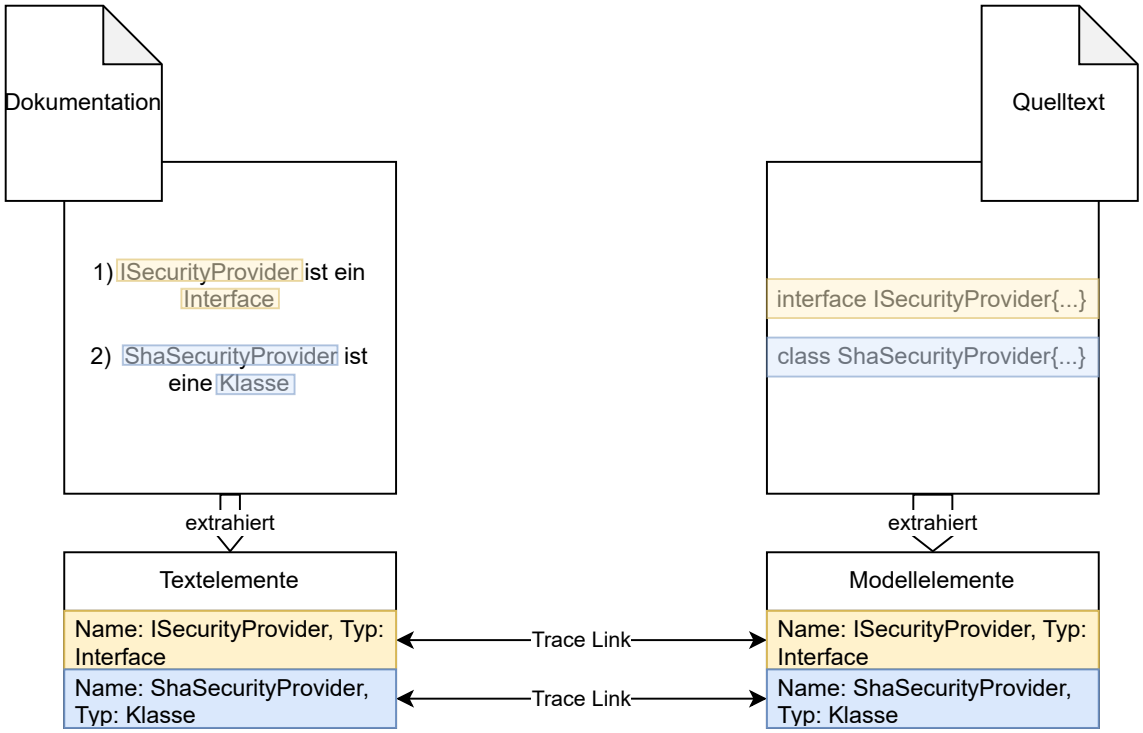

Abbildung 2.1: Beispiel Artefakte NLSAD und Quelltext von Teastore [9]

Mit Trace Links kann man verschiedene Analysen durchführen, unter anderem die Erkennung von Inkonsistenz. Die Trace Links, die in dieser Arbeit erstellt werden, sind

zwischen Elementen in der NLSD und dem Quelltext. Dabei werden die Trace Links nicht beim Erstellen der Artefakte erzeugt (engl.*trace link capture*) sondern nachträglich gewonnen (engl.*trace link recovery*). Die Trace Links können dabei auf drei unterschiedlichen Weisen erstellt werden: automatisiert, manuell oder semi-automatisiert, was die Kombination der beiden vorgehenden Varianten beschreibt [5, p. 10].

Abbildung 2.1 zeigt ein Beispiel von Quelltext mit seiner entsprechenden Dokumentation aus dem Benchmark Projekt Teastore. In der NLSD werden zwei Elemente gefunden, ISecurityProvider vom Typ Interface und ShaSecurityProvider vom Typ Klasse. Analog werden auch im Quelltext Elemente gefunden. Zwischen den Elementen der NLSD und ihren Gegenstücken im Quelltext werden dann Trace Links gewonnen. Dabei hat ein Trace Link immer genau ein Element in der NLSD und eins im Quelltext. Dabei kann die Klasse ShaSecurityProvider aus dem Quelltext auch Teil von mehrere Trace Links sein, wenn sie an mehreren Stellen in der NLSD genannt wird. Im Falle von Mehrfachnennung fügt ArDoCo im selben Satz der NLSD die Elemente zusammen, sodass nur ein Trace Link pro Element pro Satz entsteht, wie Abbildung 2.1 zeigt. Ein tieferer Einblick in ArDoCo liefert Abschnitt 2.3.

2.2 Softwaremodelle

Softwaremodelle dienen der Veranschaulichung von softwarebezogenen Themen. Ein Beispiel dafür ist die Softwarearchitektur, welche in solchen Modellen besser zu verstehen ist. Es gibt verschiedene Arten der Modelle, wobei diese Arbeit nur kurz auf das Metamodell und genauer auf das entwickelte Codemodell der Masterarbeit [20] eingeht. Diese Arbeit betrachtet deshalb das Codemodell genauer, da es benötigt wird, um den Quelltext in ein Modell zu transformieren, welches von ArDoCo verarbeitet werden kann. Da das Metamodell Grundlage des Codemodells ist, wird es in Unterabschnitt 2.2.2 genauer erläutert.

2.2.1 Metamodell

Ein Metamodell ist eine besondere Art von Modell, das die abstrakte Syntax einer Modellierungssprache spezifiziert. Die typische Rolle eines Metamodells ist es, die Semantik zu definieren, wie Modellelemente in einem Modell instanziiert werden. Ein Modell enthält typischerweise Modellelemente. Diese werden durch Instanziierung von Modellelementen aus einem Metamodell, d. h. Metamodellelementen, erstellt. Dieser Abschnitt ist eine übersetzte Definition aus dem Dokument der OMG Group [16].

2.2.2 Codemodell

Das Codemodell und die Transformation des Quelltextes in dieses ist eine der beiden Haupt Grundlagen dieser Bachelorarbeit. Damit befasst sich die Masterarbeit von Tobias Telge [20]. Diese Masterarbeit dient als Basis des folgenden Abschnittes und wird zum besseren Verständnis der Bachelorarbeit erläutert.

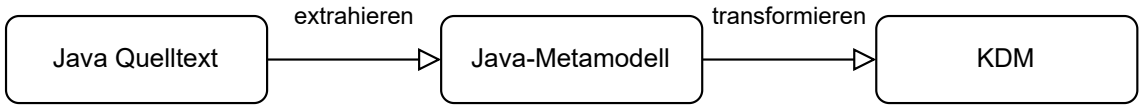

Abbildung 2.2: QuelltextTransformation

Als einheitliches Codemodell, wir das Knowledge Discovery Metamodel (KDM) [16] benutzt, da dieses unabhängig von der Programmiersprache ist. Um jedoch den Quelltext der entsprechenden Programmiersprachen in das KDM zu transformieren, benötigt man zwei Schritte, welche anhand der Abbildung 2.2 erläutert werden.

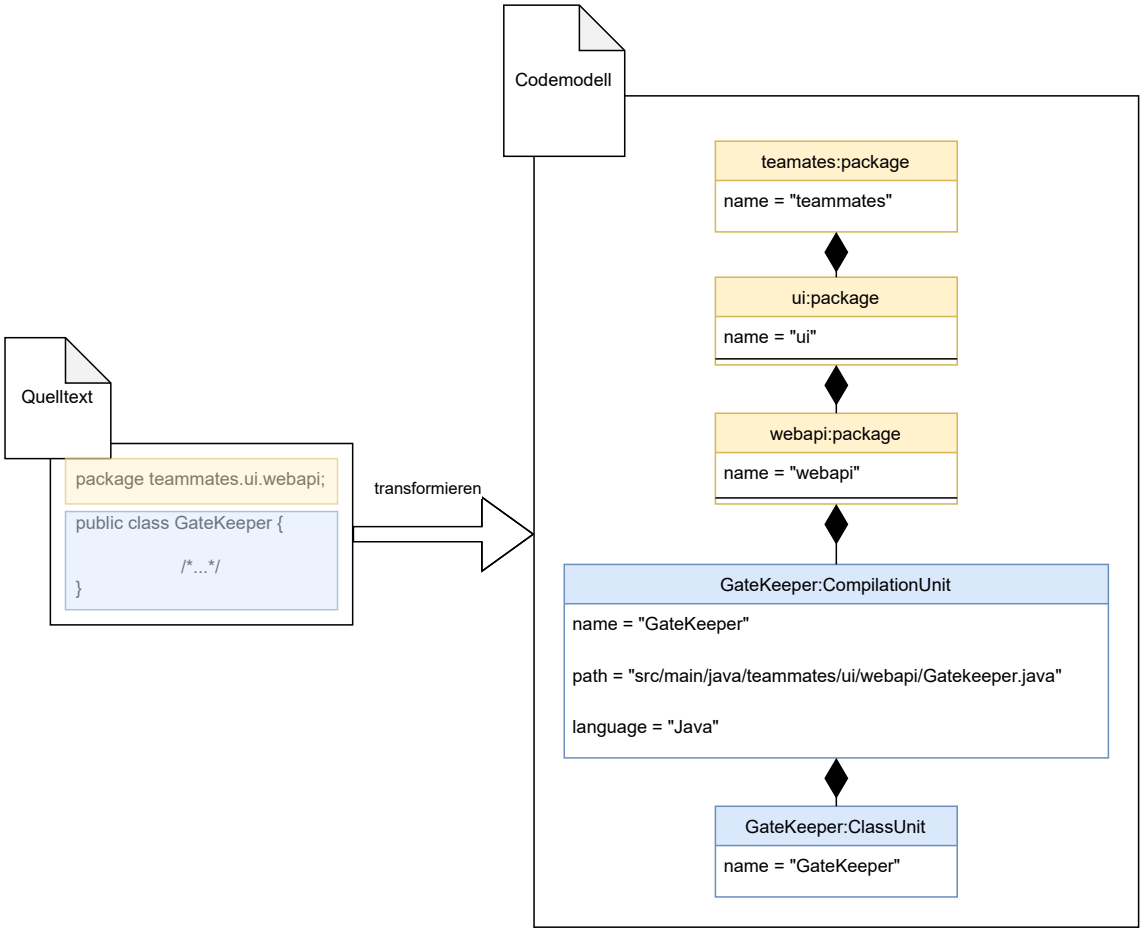

Abbildung 2.3: Beispiel für die Umwandlung von Java-Quelltext in das Codemodell anhand der UML Notation am Projekt Teammates[9]

Für die Erläuterung der Umwandlung nehmen wir das Beispiel aus dem Projekt Teammates, welches in Abbildung 2.3 zu sehen ist. Dieses Projekt sowie alle später genannten, sind Bestandteil der ArDoCo Benchmark [9], welche für Beispiel, sowie die Evaluation genutzt wird. In diesem Beispiel betrachten wir die Programmiersprache Java, in welcher der Quelltext vorliegt. Im ersten Schritt muss der Quelltext in ein Metamodel umgewandelt werden. Dazu braucht jede Programmiersprache seinen eigenen Extrahierer, in unserem

Fall ist MoDisco [3] die gewählte Option. Dieser extrahiert aus dem Java Quelltext, in unserem Fall die Klasse „GateKeeper“, welche in der Paketstruktur „teammates.ui.webapi“ liegt, ein Java-Metamodell. Dies ist eine Zwischenstufe, welche nicht explizit in Abbildung 2.3 dargestellt ist, jedoch ähnlich zu der auf der rechten Seiten dargestellten Version ist. Das Java-Metamodell kann dann weiter in das Standardmetamodell KDM transformiert werden, wie es auf der rechten Seite in Abbildung 2.3 zu sehen ist. Diese Aufgabe übernimmt ebenfalls MoDisco [3], da es das KDM integriert hat.

Durch diese Umwandlung in das Codemodell, bekommen wir weit aus mehr Informationen aus dem Quelltext heraus. Jedes Paket bzw. Unterpaket, in dem die Klasse „GateKeeper“ zu finden ist, wird als separates Element dargestellt. Die Klasse „GateKeeper“ an sich wird als CompilationUnit erfasst, da diese Klasse vor der Ausführung des Quelltextes kompiliert werden muss. Zusätzlich wird der Typ der CompilationUnit als eigenständiges Element erfasst. In unserem Fall ist „GateKeeper“ eine Klasse und wird somit als ClassUnit typisiert. Falls „GateKeeper“ jedoch ein Interface statt einer Klasse wäre, würde sich der Typ auf InterfaceUnit ändern.

Dieses Vorgehen ist auf jede beliebige Programmiersprache anwendbar. Die einzige Voraussetzung dafür ist, dass man einen entsprechenden Extrahierer für die Programmiersprache bereitstellen muss.

In dieser Bachelorarbeit werden nicht nur Elemente aus Java extrahiert, sondern ebenfalls Shell Skripte. Der entsprechende Extrahierer sowie alle benötigten Implementierungsarbeiten zum Überführen des Quelltextes in das Codemodell sind bereits in ArDoCo integriert. Shell Skripte werden auch deshalb in Betracht gezogen, da diese Bestandteil der Projekte des ArDoCo Benchmarks sind, an welchen die Evaluation durchgeführt wird. Kapitel 6 geht genauer ins Detail.

Damit jedoch die Elemente des Codemodells im Rahmen dieser Arbeit richtig genutzt werden können, erhalten diese noch eine universell unterscheidbare ID. Die Umsetzung ist dabei genauer in Abschnitt 4.1 beschrieben.

2.3 Architecture Documentation Consistency Framework

Architecture Documentation Consistency (ArDoCo) ist ein Framework zur Inkonsistenzerkennung zwischen NLSAD und seinem zugehörigen Softwarearchitekturmodell [10]. Dies wird durch Trace Link Recovery (TLR) erreicht, welches in Abschnitt 2.1 erklärt worden ist. Der folgende Unterabschnitt basiert auf den drei aufeinander aufbauenden Arbeiten [18, 11, 10] und erläutert ArDoCo anhand dieser.

ArDoCo basiert auf einem Vector Space Model (VSM), welches Information Retrieval (IR) Methoden benutzt. Die benutzten IR Methoden sind die Heuristiken, welche in Unterabschnitt 2.3.1 beschrieben werden. Das VSM beschreibt das Modell, wie die gewonnen Informationen gespeichert werden. Dies passiert über Vektoren, in welchen die Informationen, wie zum Beispiel die Wahrscheinlichkeiten von Wörtern, gespeichert werden. Um Ähnlichkeiten zu finden und TLs zu gewinnen, vergleicht man diese Vektoren miteinander.

Abbildung 2.4 zeigt, dass ArDoCo zwei Artefakte als Eingaben entgegen nimmt. Zum einen die NLSAD und zum anderen das Architekturmodell des Software-Projektes. Aus die-

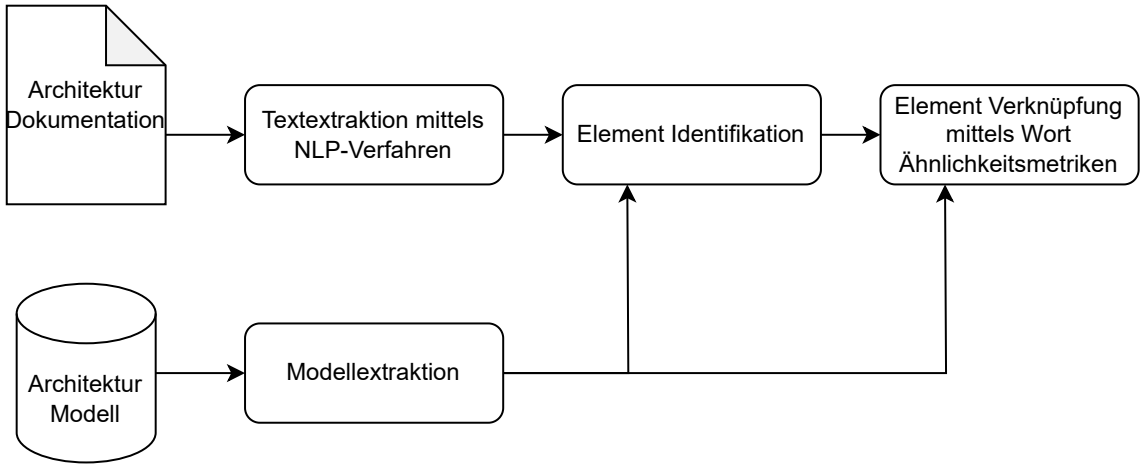

Abbildung 2.4: ArDoCo Pipeline, basierend auf Keim et al. 2023 [10]

sen Artefakten werden dann die Trace Links gewonnen. Das in Abbildung 2.4 dargestellte Vorgehen ist die TLR von ArDoCo. Dies wird in vier Schritten realisiert.

Im ersten Schritt werden die Elemente aus dem Architekturmodell extrahiert und in eine einheitliche, interne Darstellung gebracht. In unserem Beispiel von Abbildung 2.1 ist es ISecurityProvider und ShaSecurityProvider als Namen und Interface, Klasse als Typ.

Der zweite Schritt besteht in der Extrahierung von Informationen aus der NLSAD. Dabei werden Namen und Typen von potenziellen Modell Elementen durch Natural Language Processing (NLP) Verfahren und verschiedene Heuristiken gesucht. Das NLP-Verfahren bezeichnet hierbei die detaillierte Analyse der Sätze, um die vorkommenden Arten von Wörtern herauszufinden. Bekannte Vorgehen davon sind part-of-speech tagging, Satzspaltung, Lemmatisierung und Dependency Parsing. Heuristiken hingegen helfen der Entscheidungsfindung, so werden zum Beispiel durch die Ähnlichkeitsheuristik ähnliche oder gleiche Wörter von Namen bzw. Typen zu einem Cluster zusammengefügt, um so zu entscheiden, dass die Wörter dieselben Elemente beschreiben. Alle gefunden Informationen erhalten einen Konfidenzwert, welche ihre Wahrscheinlichkeit widerspiegelt, ebenfalls im Architekturmodell vorzukommen. Im Beispiel von Abbildung 2.1 sind es die zwei Namen ISecurityProvider, ShaSecurityProvider und als Typ Interface, Klasse.

Im nächsten Schritt der Element Identifikation werden aus den gewonnenen Informationen potenzielle Elemente identifiziert. Dabei wird versucht jedem Namen-Cluster ein Typ-Cluster zuzuweisen. Bei diesem Schritt wird sich an den bereits extrahierten Elementen des Architekturmodells orientiert und diese zum Vergleich hinzugezogen. Somit erhalten wir in unserem Beispiel zwei Elemente namens ISecurityProvider vom Typ Interface und ShaSecurityProvider vom Typ Klasse.

Im letzten Schritt werden die zusammengesetzten Elemente der Dokumentation mit denen des Architekturmodells durch Trace Links verknüpft. Dabei werden Wortähnlichkeitsmetriken benutzt. Im Fall von ArDoCo sind es die normalisierte Levenshtein-Distanz und die Jaro-Winkler-Ähnlichkeit. Die Levenshtein-Distanz sagt aus, wie viele Änderungen an den einzelnen Buchstaben der Wörter nötig sind, um sie in das jeweilige Andere zu ändern [12]. Dabei sind Änderungen als Einfügungen, Löschungen oder Ersetzungen der Buch-

staben zu verstehen. Die Jaro-Winkler-Ähnlichkeit verfolgt denselben Ansatz, berechnet aber ihren Wert anders und bekommt dann Werte zwischen null und eins, wobei bei eins beide Wörter identisch sind [21]. Auf die Wortähnlichkeitsmetriken wird detaillierter in Unterabschnitt 2.3.2 eingegangen.

Nach dem alle Schritte abgeschlossen sind, ist die TLR erfolgreich ausgeführt. Dadurch sind nun nachträglich aus den beiden bestehenden Software-Artefakten NLSAD und dem Software Architektur Modell Trace Links gewonnen worden. Durch diese Trace Links haben die Entitäten der beiden Artefakte Verbindungen zu ihren jeweiligen Gegenstücken im anderen Artefakt. ArDoCo geht nun noch einen Schritt weiter und behandelt die Inkonsistenzerkennung mittels Filter. Dies wird jedoch an dieser Stelle nicht näher erläutert, da primär die TLR in dieser Arbeit von Interesse ist.

2.3.1 ArDoCo Heuristiken

Wie in Abschnitt 2.3 schon beschrieben, hat ArDoCo vier Phasen, wobei nur in der Modellextraktion keine Heuristiken zum Einsatz kommen. Genauso wie Abschnitt 2.3 basiert der gesamte Abschnitt auf den Arbeiten von ArDoCo [18, 11, 10]. In der ersten Phase, der Text-Extraktion, werden folgende Heuristiken benutzt und anhand von Abbildung 2.5 erläutert:

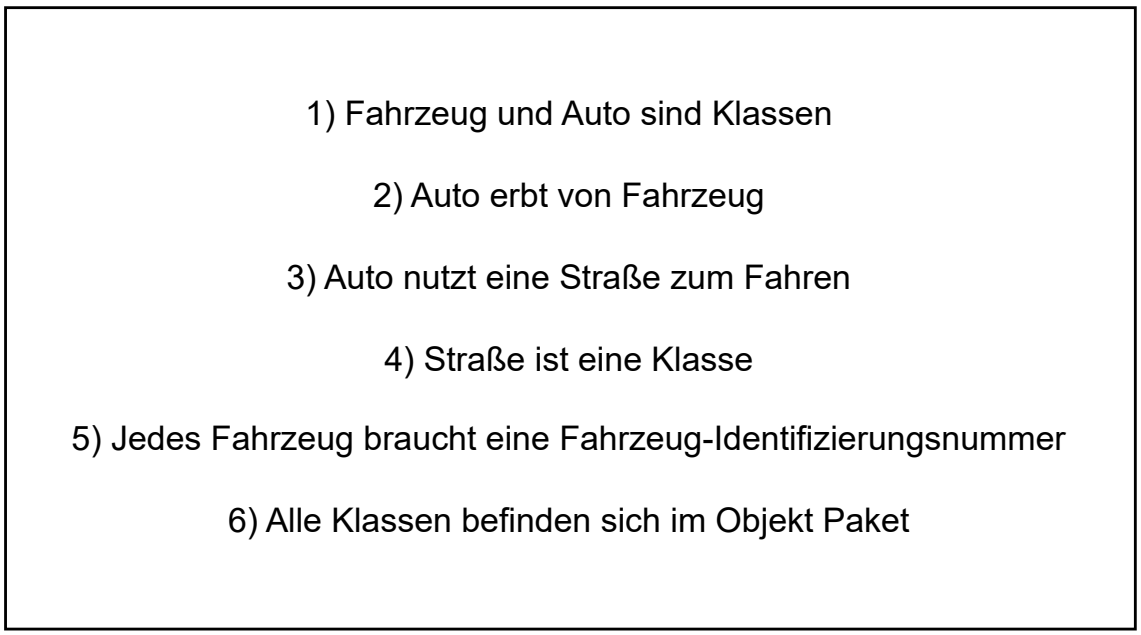

Abbildung 2.5: Auflistung von Beispielsätzen einer NLSAD

Nomen Alle Nomina werden aus dem Text extrahiert und erhalten zwei Konfidenzwerte. Diese zeigen an, wie wahrscheinlich es ist, dass das Nomen einen Namen bzw. einen Typen darstellt. Wenn das Nomen als Plural vorliegen, wird der Konfidenzwert Typ erhöht. Im Beispielsatz 1, haben die Nomen „Fahrzeug; Auto“ eine höhere Namen-Konfidenzwert als

„Klassen“. Andersherum verhält es sich mit den Typ-Konfidenzwerten. Folglich bekommt „Klassen“ einen höheren Typ-Konfidenzwert, da es im Plural vorkommt.

Eingehende Abhängigkeiten Es werden eingehende Abhängigkeiten eines Wortes untersucht und ihnen Namen und Typ Konfidenzwerte zugeteilt. Eingehende Abhängigkeiten sind dabei Abhängigkeiten, die aussagen, welche Elemente mit diesem Element in Zusammenhang stehen. Im Beispielsatz 3 befindet sich eine solche, da „Straße“ von „Auto“ zum Fahren benutzt wird. Somit besitzt „Straße“ eine eingehende Abhängigkeit, welche von „Auto“ ausgeht.

Ausgehende Abhängigkeiten Ausgehende Abhängigkeiten unterscheiden sich zu eingehenden Abhängigkeiten in ihrer entgegengesetzten Abhängigkeitsrichtung. Ein Beispiel dafür ist in Beispielsatz 2, „Fahrzeug“ hat eine ausgehende Abhängigkeit zu „Auto“. Diese ausgehenden Abhängigkeiten werden als speziellen Typ wie numerischer Modifikator oder vorausbestimmende Abhängigkeiten gespeichert.

Muster Suche Die Muster Suche untersucht die NLSD nach verschiedenen Muster, wie zum Beispiel dem Artikel-Typ-Name Muster. Wenn ein Wort nicht eindeutig auf Name oder Typ zugewiesen werden kann und auf ein Artikel-Name Muster folgt, wird dieses Wort als Typ klassifiziert. Wenn solch ein Wort zwischen Artikel und Typ vorkommt, wird es wiederum als Name klassifiziert. Im Beispielsatz 2 kommt solch ein Muster vor, bei den drei Wörtern „Die Auto Klasse“ handelt es sich um ein Artikel-Name Muster, auf welches das Wort „Klasse“ folgt. Deswegen wird „Klasse“ als Typ gespeichert.

Begrenzungszeichen Es werden nach Termen mit Begrenzungszeichen, wie zum Beispiel der Bindestrich, gesucht und jeder Term sowie das Gesamte abgespeichert. Im Beispielsatz 5 wird somit aus dem Wort „Fahrzeug-Identifizierungsnummer“ die Terme „Fahrzeug; Identifizierungsnummer; Fahrzeug Identifizierungsnummer“ abgespeichert.

Zusammengesetzte Terme In dieser Heuristik wird nach Vorkommen von Name und Typ gesucht. Falls ein Name auf einen Typ folgt, oder vice versa, wird ein neuer Zusammengesetzter Term erstellt. Somit wird aus „Objekt; Paket“ in Beispielsatz 6 ein zusammengesetzter Term „Objekt Paket“ gespeichert.

Ähnlichkeit In der Ähnlichkeitsheuristik wird nach gleichen oder ähnlichen Wörtern gesucht und diese in einem Cluster zusammengefügt. Dadurch werden alle Vorkommen von „Klasse; Klassen“ zu einem einzigen Typ-Cluster „Klasse“ zusammengefügt. Der Unterabschnitt 2.3.2 erläutert die dafür benutzten Metriken näher.

Danach folgt die Identifikation der Elemente. In diesem Schritt werden in der NLSD gefunden Informationen mit denen aus dem Modell verglichen. Im Beispiel von Abbildung 2.1 wird dadurch klar, dass „ISecurityProvider; ShaSecurityProvider“ vom Typ „Interface“ bzw. „Klasse“ sein müssen, da ISecurityProvider, ShaSecurityProvider im Modell ein Interface bzw. Klasse ist.

Der letzte Schritt der somit Heuristiken benutzt, ist die Verknüpfung von Elementen. Bei diesem gibt es verschiedene Agenten die immer jeweils zwei Elemente vergleichen und zwischen diesen ein Trace Link erstellt, falls ihre Konfidenz einen bestimmten Wert überschreitet. Die Vergleiche finden mit einer Kombination aus Levenshtein-Distanz und Jaro-Winkler-Ähnlichkeit statt. Der Unterabschnitt 2.3.2 erläutert die Funktionsweise der beiden Metriken genauer.

2.3.2 Wortähnlichkeitsmetriken

Die beiden Wortähnlichkeitsmetriken kommen nicht nur in ArDoCo vor, sondern sind auch Teil einer neuen Heuristik, welche später vorgestellt wird. Um diese besser verstehen zu können, werden die beiden Wort Ähnlichkeitsmetriken Jaro-Winkler-Ähnlichkeit und die Levensthein Distanz näher erläutert. Der erste folgenden Abschnitte basiert auf dem Paper von Winkler und William E. [21] und der zweite auf Levenshtein und Vladimir I [12].

Jaro-Winkler-Ähnlichkeit Die Jaro-Winkler-Ähnlichkeit ist ein Maß, welches angibt, wie ähnlich zwei Wörter zueinander sind. Um diese Ähnlichkeit zu berechnen benötigt man folgende zwei Formeln:

$$Jaro - Ähnlichkeit = \begin{cases} 0 & if \quad m = 0 \\ \frac{1}{3} * (\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}) & for \quad m! = 0 \end{cases} \tag{2.1}$$

$$Jaro - Winkler - Ähnlichkeit = Jaro - Ähnlichkeit + P * L * (1 - Jaro - Ähnlichkeit) \tag{2.2}$$

In der Jaro-Ähnlichkeitsformel beschreibt m die Anzahl an übereinstimmenden Zeichen, t die Hälfte aller benötigten Transpositionen, um von einem Wort zum Anderen zu kommen. |s1| und |s2| beschreiben die Länge der beiden entsprechenden Wörter. Um die Formel verständlicher zu machen, hilft folgendes Beispiel. In diesem werden die beiden Wörter „bus“ und „menus“ miteinander verglichen. Da es bei den beiden Wörter die übereinstimmenden Zeichen „u,s“ gibt, befinden wir uns im zweiten Fall der Formel. In unserem Fall ist m = 2, |s1| = 3, |s2| = 5 und t = 1,5. T ergibt sich daraus, dass man bei dem Wort „bus“ das b auf ein n ändern muss und dann noch die zwei Buchstaben m und e am Anfang hinzufügen muss um das Wort „menus“ zu bekommen. Dies sind drei Transpositionen und da t die Hälfte davon ist, ist t = 1,5. Wenn man alle Werte in die Formel einfügt, ergibt sich:
$Jaro - Ähnlichkeit = \frac{1}{3} * (\frac{2}{3} + \frac{2}{5} + \frac{2-1,5}{2}) = 0,4389$

Wenn man nun die Jaro-Winkler-Ähnlichkeit dieser beiden Wörter berechnen will, braucht man noch P, welches den Skalierungsfaktor beschreibt. Dieser gibt an, in welchem Maße der Wert nach oben korrigiert wird, falls gemeinsame Präfixe in den beiden Wörtern vorhanden sind. Der Standardwert beträgt 0,1 und wird hier auch verwendet. L gibt die Anzahl an übereinstimmenden Präfix Zeichen an, in unserem Fall ist L = 0. Somit lässt sich berechnen:

$$Jaro - Winkler - Ähnlichkeit = 0,4389 + 0,1 * 0 * (1 - 0,4389) = 0,4389$$

Levenshtein-Distanz Die Levensthein-Distanz gibt die minimale Anzahl an Operationen an, die gebraucht werden, um die erste Zeichenkette in die zweite umzuwandeln. Die dafür benötigten Operationen sind einfügen, löschen und ersetzen von Zeichen. Der Algorithmus zum Berechnen der Levensthein-Distanz ist wie folgt:

$$Levenshtein - Distanz_{i,j} = \begin{cases} Levenshtein - Distanz_{i-1,j-1} & +0 \; falls \; u_i = v_j \\ Levenshtein - Distanz_{i-1,j-1} & +1 \; (Ersetzung) \\ Levenshtein - Distanz_{i,j-1} & +1 \; (Einfügen) \\ Levenshtein - Distanz_{i-1,j} & +1 \; (Löschung) \end{cases} \tag{2.3}$$

Die Tabelle 2.1 veranschaulicht den Algorithmus mithilfe einer Matrix. Dabei nehmen wir die selben Beispielwörter wie bei der Jaro-Winkler-Ähnlichkeit, „bus“ und „menus“. Die Zahlen in den einzelnen Spalten und Zeilen besagen, wie viele Operationen benötigt werden, um von diesem ersten Teilwort zum zweiten zu kommen. Die Zahl zwei in Spalte „e“ und Zeile „b“ besagt, dass man zwei Operationen braucht, um vom Teilwort „b“ zum Teilwort „me“ zu gelangen. Die Gesamtzahl der Operationen, welche benötigt werden, um vom ersten Wort zum zweiten zu kommen, stehen in der Matrix an dem Eintrag in letzter Zeile und Spalte. In unserem Fall beträgt dieser drei. Somit braucht es drei Operationen, um vom Wort „bus“ zum Wort „menus“ zu gelangen. Die genauen dafür zu benutzenden Operationen werden nicht erläutert, da es mehrere Möglichkeiten zur Erfüllung der Transposition gibt.

null	ε	m	e	n	u	s
ε	0	1	2	3	4	5
b	1	1	2	3	4	5
u	2	2	2	3	3	4
s	3	3	3	3	3	3

Tabelle 2.1: Levenshtein-Distanz Beispiel

3 Verwandte Arbeiten

Dieses Kapitel zeigt eine Übersicht von verwandten Arbeiten im Bereich der Trace Link Recovery (TLR) zwischen verschiedenen Software-Artefakten. Dabei liegt der Fokus auf den beiden Software-Artefakten Quelltext und der natürlichsprachlichen Software Dokumentation (NLSD). Um den verwandten Arbeiten eine Struktur zu geben, sind diese in drei Gruppen gegliedert.

Die erste Gruppe sind Arbeiten, welche die Trace Link Recovery zwischen der NLSD und einem anderen Artefakt ausführt, welches nicht der Quelltext ist. In dieser Gruppe ist ArDoCo [8], die Grundlage dieser Arbeit, zu erwähnen. Diese führt die TLR zwischen der NLSD und dem Architekturmodell aus. In Abschnitt 2.3 wurde bereits die Funktionsweise von ArDoCo erläutert und in Unterabschnitt 2.3.1 wird genauer auf die benutzten Heuristiken eingegangen. Diese Gruppe grenzt sich eindeutig über die unterschiedlichen Software-Artefakte als Eingabe ab.

Die zweite Gruppe befasst sich mit der TLR zwischen Quelltext und einem anderen Artefakt, welches nicht die NLSD ist. Dabei beschäftigen sich die meisten Arbeiten wie auch die von Zhang et al. [23] mit der TLR zwischen Quelltext und Anforderungen [4]. Dabei benutzt Zhang et al. [23] hauptsächlich drei Heuristiken zum Gewinnen der Trace Links. Diese sind Synonyme, Verb-Objekt Sätze und strukturelle Informationen. Die Arbeit von Zou et al. [24] geht dabei noch weiter und nutzt weitere Informationen für die TLR. Dazu werden zum einen Schlüsselsätze aus den Anforderungen extrahiert und zum anderen Gewichtungen von potenziellen Elementen vergeben. Beide Ansätze lassen sich auch auf die TLR zwischen Quelltext und der NLSD anwenden. Die Gewichtung der potenziellen Elemente ist bereits in ArDoCo vorhanden und wird deswegen in dieser Arbeit verwendet. Der Unterschied zu dieser Arbeit besteht in der unterschiedlichen Eingabe von Software-Artefakten.

Es gibt aber auch Arbeiten wie die von Parizi, Lee und Dabbagh [17], die sich mit der Traceability zwischen Quelltext und entsprechenden Tests beschäftigt. In dieser Arbeit wird die TLR mit Hilfe von fixture element types (FETs) durchgeführt. Dabei dienen die FETs als Anhaltspunkt für Trace Links (TLs). Diese werden gesucht und sollten bestenfalls Instanzvariablen in den Testfälle sein. Die FETs werden dann im zweiten Schritt reduziert. Ein Problem bei diesem Vorgehen ergibt sich, wenn keines der vorkommenden Objekten als FET deklariert ist. Somit können dann auch keine TLs gewonnen werden. Das beschriebene Vorgehen sowie andere dieses Bereiches der TLR lässt sich nur schwer auf die TLR zwischen Quelltext und NLSD übertragen.

Eine vielversprechende Gruppe für diese Arbeit besteht aus der TLR zwischen dem Quelltext und seiner NLSD. Hierbei gibt es allerdings noch wenig Arbeiten, sodass diese Ausarbeitung den Forschungsbereich gut ergänzt. Die Arbeit von Antonio et al. [2] basiert auf einem Vector Space Model (VSM) und nutzt Information Retrieval (IR) Methoden. Dabei erläutert die Arbeit den Prozess der Verarbeitung von der NLSD und dem Quelltext.

Der Text wird dabei in nur Kleinbuchstaben und ohne Stoppwörter transformiert. Aus dem Quelltext werden hingegen Identifikatoren extrahiert und separiert, falls der Identifikator aus mehreren Wörtern besteht. Zum Schluss werden auch hier die Schritte der Textverarbeitung angewandt. Die TLR an sich wird jedoch nur semi-automatisiert durchgeführt, welche sich von diesem voll automatisiertem Ansatz unterscheidet.

In der Arbeit von Marcus und Maletic [13] wird ein weiteres Konzept, das Latent Semantic Indexing (LSI), eingefügt. Diesem Konzept liegt das VSM zugrunde und ist Teil der IR Methoden. Mithilfe von LSI lassen sich Zusammenhänge zwischen Wörtern darstellen, sowie die Bedeutung von Sätzen identifizieren, welche in der TLR berücksichtigt werden. Durch dieses Konzept wurden Ähnlichkeitsmetriken eingeführt, welche das Problem von ähnlichen statt denselben Wörtern in den verschiedenen Software Artefakten behebt. Die ähnlichen Wörter werden dadurch mit TLs verbunden. Dieser Ansatz ermöglicht es, bereits teilweise die TLR zu automatisieren, bringt jedoch noch viele falsche TLs mit sich. Das Konzept dient als gute Grundlage, auf welcher auch ArDoCo aufbaut und nutzt, um die TLR vollständig zu automatisieren.

Die Arbeit von Nagano, Ichikawa und Kobayashi [14] befasst sich mit einer semi-automatisierte Methode der TLR, die auch funktioniert, wenn Quelltext und Dokumentation in verschiedenen Sprachen geschrieben wurden. Diese verarbeitet die NLSD zuerst mit einer morphologischen Analyse und anschließend mit dem Entfernen von Stoppwörtern. Für den Quelltext wird hingehen zuerst ein Syntaxbaum aufgebaut. Der Syntax-Parser, der den Syntaxbaum erstellt, ist dabei individuell für jede Programmiersprache. Im nächsten Schritt werden die Identifikator-Schlüsselwörter und Kommentar-Schlüsselwörter, die jedem Knoten entsprechen, aus den Identifikatoren und Kommentaren des Quelltextes extrahiert. Dadurch entsteht ein Konvertierungswörterbuch, welches Paare von Identifikator-Schlüsselwörtern und die konvertierten Kommentar-Schlüsselwörter enthält und zur Ausgabe eines Schlüsselwortindexes des Quelltextes verwendet wird. Zum Schluss gilt es die TLs mittels Bayes-Training und -Vorhersage aus den beiden verarbeiteten Software-Artefakten zu gewinnen. Die erläuterte Vorgehensweise zum Erstellen eines Syntaxbaumes, findet in dieser Arbeit keine Anwendung. Die Bachelorarbeit verfolgt das Ziel, die Trace Links vollständig automatisiert zu gewinnen.

Automatisierte TLR sind in den Arbeiten von Antoniol et al. [1] und Chen und Grundy [22] beschrieben. Die Arbeit von Antoniol et al. [1] basiert auf ihrer vorhergehenden Arbeit [2] und schafft die automatisierte TLR für C++ und Java Quelltext mit nur mittelmäßigen Ergebnissen. Die Arbeit von Chen und Grundy [22] erreicht hingegen bessere Ergebnisse in Bezug auf Präzision und Ausbeute, da sie ihren Fokus auf das Überschreiten der Limits des VSM legt. Allerdings beziehen sie sich hierbei nur auf Java-Quelltext.

Die Bachelorarbeit ermöglicht die automatisierte Trace Link Recovery für potenziell jede Programmiersprache, solange sie einen Extraktor besitzt. Das wird genauer in Abschnitt 2.3 erläutert. Um gute Ergebnisse zu erzielen, werden die besten Heuristiken der vorgestellten verwandten Arbeiten in diese Bachelorarbeit eingebaut. Auf die verschiedenen neu eingebauten Heuristiken wird genauer in Kapitel 5 eingegangen.

4 Architekturelle Heuristiken von ArDoCo

In diesem Kapitel wird auf die ersten Schritte eingegangen, in denen lediglich mit den bestehenden Heuristiken Ergebnisse in der TLR erzielt werden. Dazu wird als erstes ArDoCo angepasst, um auch Quelltext verarbeiten zu können, was in Abschnitt 4.2 näher erläutert ist.

Außerdem werden sinnvolle Heuristiken mit ihren Parametern in Abschnitt 4.3 beschrieben. Die an diesen Parametern durchgeführten Änderungen sowie deren Auswirkungen werden dann in Abschnitt 4.4 erläutert.

4.1 Auswirkungen des neuen Kontexts auf ArDoCo

ArDoCo nutzt bisher als Eingaben die NLSAD und das Software-Architekturmodell. Im Kontext dieser Bachelorarbeit wird ArDoCo jedoch mit der Eingabe des Quelltextes anstatt des Architekturmodells konfrontiert, wie es in Abbildung 4.1 zu sehen ist. Dies bringt ein paar Herausforderungen mit sich, welche in diesem Abschnitt mitsamt Lösungsansätzen erläutert werden.

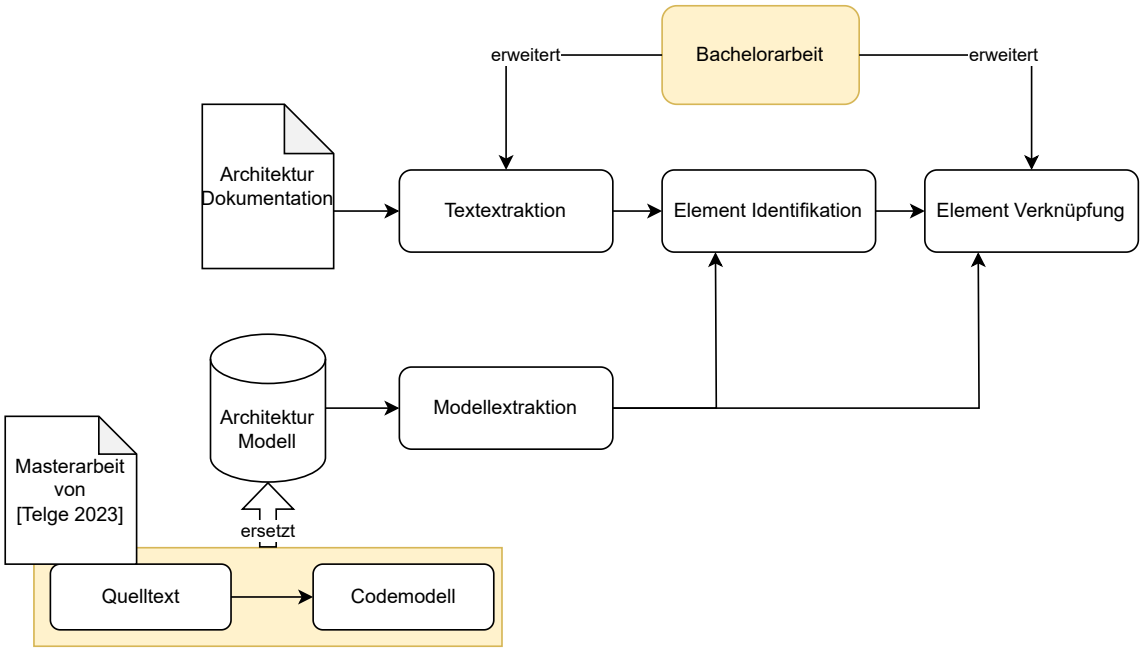

Abbildung 4.1: Erweiterung von ArDoCo

Der auffälligste Unterschied besteht darin, dass ArDoCo nun ein anderes Modell als Eingabe verarbeiten muss. Dieses Codemodell ist voraussichtlich deutlich größer als die

bisherigen Architekturmodelle. Dies ist der Fall, da das Architekturmodell nur den Zweck hat, die Architektur des Quelltextes widerzuspiegeln, anstatt eine detaillierte Übersicht zu liefern, wie es das Codemodell macht. Das Codemodell besteht aus sämtlichen Elementen des Quelltextes, wie zum Beispiel seinen Klassen, Interfaces oder auch Paketen. Dies führt unausweichlich zu einer längeren Laufzeit, da mehr Elemente verarbeitet werden müssen. Um bei großen Datenmengen die Laufzeit von ArDoCo zu regulieren, ist eine potenzielle Lösung, das Filtern des Quelltextes bei der Transformation in das Codemodell. Somit werden Elemente entfernt, die nicht direkt Code enthalten. Ein Beispiel dafür ist das Ausfiltern von Dateien mit Namen wie „module-info.java“, da diese nur Informationen enthalten und ebenfalls keinen Bezug zur NLSD haben. Dadurch wird das Codemodell kleiner und ebenso die Laufzeit kürzer.

Im Quelltext kann es vorkommen, dass sich in einem Paket ein Klasse mit identischem Namen befindet. Ein Beispiel dafür ist „BibtexExtractor“ im Projekt JabRef [9], welches als Paket und Klasse vorliegt. Der einzige Unterschied befindet sich in der Groß- und Kleinschreibung dieser beiden Namen. Das Paket wird klein geschrieben, während die Klasse, wie vorhin genannt, groß geschrieben wird. In diesem Fall erstellt ArDoCo einen Namen-Cluster „BibtexExtractor“ und kann diesem nur einen Typ-Cluster zuweisen, obwohl „BibtexExtractor“ sowohl vom Typ Klasse als auch Paket ist. Dies führt zu einer fehlerhaften Verknüpfung. Um das zu verhindern ist es denkbar, den Kontext der Textelemente miteinzubeziehen. Dies ermöglicht die exaktere Zuweisung von Namen und Typen. Den Bezug des Kontextes von Textelementen in ihrer Typfindung stellt für ArDoCo eine Herausforderung dar, was eine Unterscheidung erschwert. Dieses Problem ist jedoch auch im Rahmen dieser Arbeit im Verhältnis zu seinen Nutzen zu zeitaufwendig. Deswegen wird dieser Sachverhalt nicht weiter betrachtet und andere Heuristiken in den Fokus gesetzt.

Eine weitere Herausforderung besteht darin, dass im Quelltext Situationen auftreten können, in denen Elemente ähnliche Namen besitzen. Ein konkretes Beispiel hierfür ist das Auftreten von Namen, wie „Test, Rest, Jest“ in der NLSD des Projektes Teammates. Hier haben diese Wörter jeweils unterschiedliche Bedeutung. „Test“ hat viele Bedeutungen, wie zum Beispiel Klassen zum Testen. Hingegen meint „Jest“ das Testframework mit eben diesem Namen, wie aus folgendem Satz zu erkennen ist: „[...] are tested separately with Jest.“. Aus dem Satz „[...] back end logic is designed as a REST-ful controller.“ lässt sich erkennen, dass „REST“ eine Designentscheidung für die back-end-Logik darstellt. Somit meinen alle drei Wörter unterschiedliche Dinge, werden jedoch durch die Cluster Heuristik zu einem NounMapping zusammengefügt, welche näher in Unterabschnitt 2.3.1 erläutert worden sind. Um diesem Problem entgegenzuwirken, könnte eine mögliche Lösung darin bestehen, die Cluster Heuristik anzupassen, sodass diese strenger vergleicht, welche Wörter zu einem NounMapping zusammengeführt werden sollten. Dies wäre im dargestellten Beispiel jedoch nicht hilfreich, da die Wörter sich nur um einen Buchstaben unterscheiden. Im Gegensatz dazu würde es NounMappings wie zum Beispiel „webapi, WebApiServlet“ nicht mehr geben.

Eine weitere Herausforderung ist ein Spezialfall des Problems der ähnlichen Namen. In dem Projekt JabRef gibt es zum Beispiel die beiden Klassen „BibDatabase“ und „BibDatabases“, welche sich nur durch Singular bzw. Plural unterscheiden. Hier ist eine Unterscheidung, welche von beiden Klassen in der NLSD nun gemeint ist, nicht klar durchführbar. Es lässt sich nur schwer unterscheiden, sodass man davon ausgehen muss, dass in der

NLSD alle Textelemente den exakt selben Namen haben wie ihr Gegenstück des Modells. Eine potenzielle Lösung könnte dennoch sein, den Kontext des Satzes mit einzubeziehen. Das erweist sich jedoch, wie vorhin bereits erläutert, als schwierig. Folglich muss die Entscheidungsfindung zum Hinzufügen von TLs in den Goldstandard klar in Abschnitt 6.1 erläutern sein.

Die genannten Herausforderungen sind nur ein Ausschnitt der potenziellen Schwierigkeiten, die bei der Verwendung von ArDoCo in einem neuen Kontext auftreten können.

4.2 ArDoCo Anpassungen

Um im ersten Schritt überhaupt die bestehenden Heuristiken zu nutzen und anzupassen, müssen erstmals Anpassungen an ArDoCo stattfinden, sodass ArDoCo auch mit dem Quelltext als Eingabe arbeiten kann. Dazu wird eine neue Pipeline erstellt, namentlich „ArDoCoForSadCodeTraceabilityLinkRecovery“, welche sich aus den einzelnen Pipelineschritten zusammensetzt, die für die Umsetzung benötigt werden. Für die Erstellung der neuen Pipeline wird sich an die Variante zwischen Software-Architekturmodell und der natürlichsprachlichen Software-Dokumentation gehalten.

Die neue Pipeline besteht nahezu aus denselben Schritten, wie auch die Variante zwischen Softwarearchitektur Modell (SAM) und NLSAD, mit der Ausnahme, dass die bestehende Modellextraktion entfernt und eine neue eingebaut wird. Dazu wird ein CodeModelConnector, sowie CodeModelProviderAgent und -Informant erstellt, wie Abbildung 4.2 es zeigt.

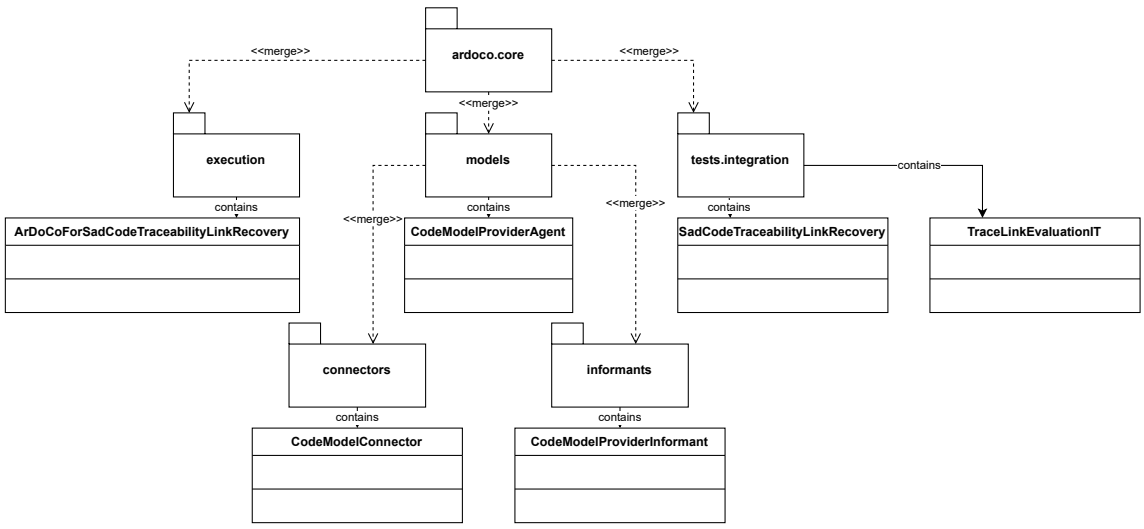

Abbildung 4.2: Paket Diagramm der Anpassungen an ArDoCo

Der CodeModelProviderAgent nutzt den AllLanuagesExtractor, um das Codemodell aus dem Quelltext zu extrahieren. Der AllLangugesExtractor ist aus der Masterarbeit von Telge [20] und neben weiteren Aspekten bereits in ArDoCo implementiert, damit das Codemodell erstellt werden kann. Der AllLanguages Extractor kann für jede beliebige Programmiersprache erweitert werden.

Die Extrahierung selbst, sowie die Speicherung der extrahierten Elemente, findet in dem CodeModelProviderInformant statt. Man kann jedoch auch ein bereits erstelltes Codemodel zur TLR nutzen.

Um nur die benötigten Entitäten zu erhalten und diesen ihre globale Identifikation zuzuweisen, wird der CodeModelConnector verwendet. Dort werden die Entitäten in Klassen, Interfaces, Pakete und Shell Skripte unterteilt und ihr entsprechende,r relativer Pfad im Quelltext als eindeutige Identifikation ermittelt und zugewiesen. Dabei werden im momentanen Stand der Arbeit noch keine Unterklassen einer Klasse gefunden und als Element gespeichert.

Im CodeModelConnector ist ein zusätzlicher Modus hinzugefügt, der die Pfade kürzt, indem er den „src/(main|text)/java“, falls dieser vorhanden ist, aus dem Pfad entfernt. Das erweist sich in der TLR als nützlich, wenn in dem Projekt, auf dem die TLR ausgeführt wird, die gleiche Paketstruktur für ihre Klassen und dazugehörigen Testklassen verwendet wird. Wenn in der NLSD also ein Paket erwähnt wird, werden immer zwei TL erstellt, zum einen für das Paket in „main“ und zum anderen für das Paket in „test“. Es kann auch passieren, dass nur einer der beiden TL entsteht, welcher sogar der ungewollte sein kann, was die Ausbeute reduziert. Wenn man nun jedoch diesen Teil im Pfad entfernt, wird immer nur ein TL erstellt, der auch auf das richtige Paket zeigt. Dies ist deshalb der Fall, da die beiden Pakete in „main“ und „test“ im Grunde das selbe Paket darstellen.

Durch das Nutzen des gekürzten Pfadmodus kann bereits eine höhere Ausbeute in den Projekten Teammates und JabRef erzielt werden. In Teammates wird die Ausbeute um neun Prozentpunkte gesteigert und in JabRef um 20 Prozentpunkte. Ebenfalls erhöht sich die Präzision von JabRef um fünf Prozentpunkte. Auf die anderen Projekte hat der Modus keinen Einfluss.

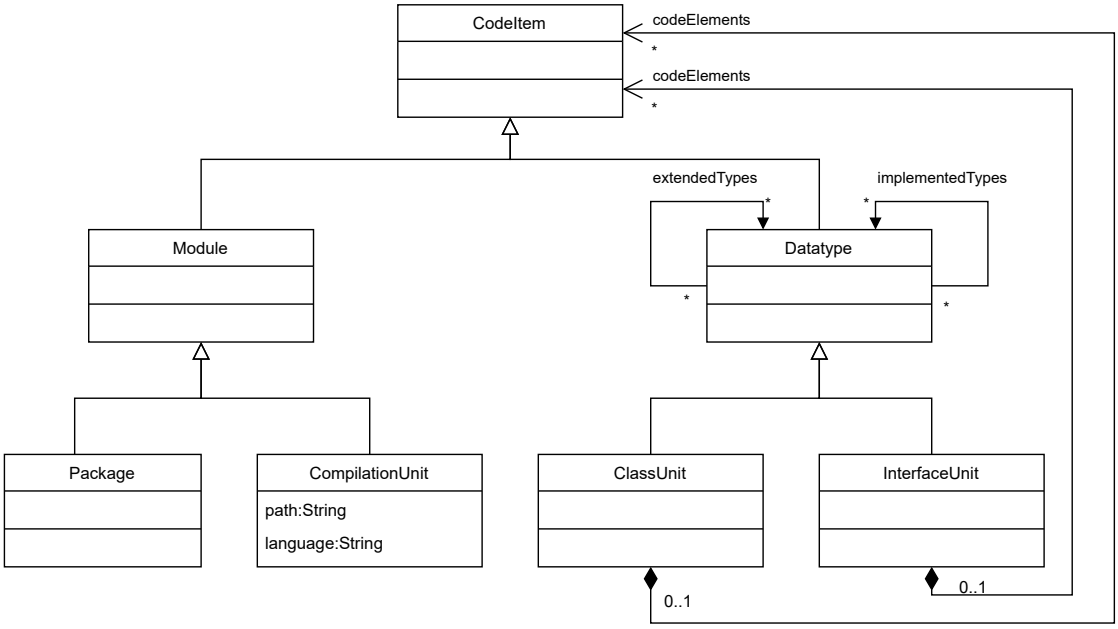

Abbildung 4.3: Vereinfachte Version der Codemodell-Architektur, basierend auf Telge 2023 [20]

Die Fehleranfälligkeit des gekürzten Pfadmodus besteht darin, wenn Klassen in diesen Paketen ebenfalls gleich benannt werden. Diese Klassen werden dann als ein Element gesehen, obwohl sie unterschiedlich sind. Dies ist jedoch in den betrachteten ArDoCo Benchmark Projekten nicht aufgetreten. Außerdem ist es in der Informatik nicht zu empfehlen, Klassen in unterschiedlichen Paketen gleich zu benennen, dies führt zu mehr Komplexität des Quelltextes. Dadurch kann die Übersichtlichkeit für den Programmierer potenziell beeinträchtigt werden. Aus diesem Grund führt dieses Vorgehen zu schlechtem Quelltext. Daher kann man die potenziell erzeugten Fehler vernachlässigen.

4.3 Anpassbare Parameter in ArDoCo Heuristiken

ArDoCo [8] selbst hat bereits viele Agenten und Informanten mit anpassbaren Parametern. Jedoch sind nicht alle von den Agenten und Informanten relevant für die TLR zwischen der NLSD und Quelltext. In den Agenten kann man nur die in ihm instanziierten Informanten herausnehmen oder Neue hinzufügen. Das Hinzufügen neuer Heuristiken bzw. ihrer Informanten wird in Kapitel 5 behandelt. Ebenfalls nicht von Interesse im Rahmen dieser Bachelorarbeit sind die Agenten und Informanten bezüglich der Inkonsistenzerkennung, die ArDoCo nach der TLR ausführt.

Informant	Parameter	bestehender Wert
ExtractionDependentOccurrenceInformant	probability	1.000000
InDepArcsInformant	nameOrTypeWeight	0.500000
InDepArcsInformant	probability	1.000000
InstantConnectionInformant	probability	1.000000
InstantConnectionInformant	probabilityWithoutType	0.800000
MappingCombinerInformant	minCosineSimilarity	0.400000
NameTypeConnectionInformant	probability	1.000000
NameTypeInformant	probability	1.000000
NounInformant	nameOrTypeWeight	0.500000
NounInformant	probability	0.200000
OutDepArcsInformant	nameOrTypeWeight	0.500000
OutDepArcsInformant	probability	0.800000
ReferenceInformant	probability	0.750000
SeparatedNamesInformant	probability	0.800000

Tabelle 4.1: Sinvolle bestehende Heuristiken zum Anpassen

Alle Agenten und zusätzlich die Informanten zur Inkonsitenzerkennung sind aus den potenziell anpassbaren Heuristiken entfernt. Dadurch bleiben die in Tabelle 4.1 aufgelisteten Informanten sowie ihre anpassbaren Parameter übrig.

Im Folgenden werden noch die Funktionen der einzelnen Informanten erläutert, sowie der Einfluss seiner Parameter auf die TLR.

ExtractionDependentOccurrenceInformant Dieser Informant ist Teil der Text-Extraktion. Er sucht nach potenzielle Namen und Typen von Instanzen in der NLSD. Wenn er welche findet, speichert er diese als NounMapping ab. Dabei ist das NounMapping als Name oder als Typ gespeichert. Der Parameter „probability“ sagt aus, wie wahrscheinlich die gefunden Namen oder Typen sind und wird als Teil des NounMappings gespeichert.

InDepArcsInformant Dieser Informant ist Teil der Text-Extraktion und analysiert die eingehenden Abhängigkeiten von Wörtern. Die einengenden Abhängigkeiten sind Wörter, die von dem zu analysierenden Wort abhängen, um zum Beispiel ihre Bedeutung zu verstehen. Diese Abhängigkeiten beeinflussen, inwiefern das zu analysierende Wort ein Name oder ein Typ darstellt und modifiziert die Wahrscheinlichkeit seines NounMappings anhand des Produktes der beiden Parameter „probability“ und „nameOrTypeWeight“.

InstantConnectionInformant Dieser Informant ist Teil der Elementverknüpfung. Er ist der Informant, der die ersten InstanceMappings erstellt, welche später zu TLs werden. Er geht dabei alle von der NLSD identifzierten Elemente durch und vergleicht sie mit den aus dem Quelltext extrahierten Elementen. Dieser Vergleich basiert auf deren Namen und als Basis wird die Jaro-Winkler-Ähnlichkeit benutzt. Dabei bestimmt die „probability“ bzw. die „probabilityWithoutType“, je nachdem ob das Textelement einen zugewiesenen Typ hat, wie wahrscheinlich dieses InstanceMapping ist.

MappingCombinerInformant Dieser Informant ist Teil der Textextraktion und kombiniert ähnliche PhraseMappings zu einem Einzigen. Ein PhraseMapping ist dabei eine Liste von Sätzen mit allen Infos, die in diesem Satz vorkommen. Attribute davon sind zum Beispiel die Satznummer, der Text an sich, die enthaltenen Wörter oder auch potenzielle Nebensätze. Dabei müssen die kombinierten PhraseMappings einen Ähnlichkeitswert größer dem Parameter „minCosineSimilarity“ haben.

NameTypeConnectionInformant Dieser Informant ist Teil der Elementverknüpfung. Er sucht im vorgefiltertem Text nach Vorkommen von verschiedenen Mustern, die Namen und Typ beinhalten. Ein Beispiel hierfür ist die Situation, in der einem Namen unmittelbar ein Typ folgt. Falls dies der Fall ist, wird ein neues Textelement erstellt und gespeichert. Die Wahrscheinlichkeit, dass das neu gefundene Textelement korrekt ist, wird mithilfe des Wertes des Parameter „probability“ definiert.

NounInformant Dieser Informant ist Teil der Textextraktion und klassifiziert Wörter als Nomen. Diese Nomen werden weiter analysiert und als Name oder Typ gespeichert. Die Wahrscheinlichkeit, dass es sich bei dem entsprechendem Nomen um einen Namen oder Typen handelt, definiert das Produkt aus den beiden Parametern „probability“ und „nameOrTypeWeight“.

OutDepArcsInformant Dieser Informant ist Teil der Textextraktion und analysiert die ausgehenden Abhängigkeiten von Wörtern. Er funktioniert dabei wie der InDepArcsInformant, jedoch mit den ausgehenden statt eingehenden Abhängigkeiten. Das Produkt der beiden Parameter „probability“ und „nameOrTypeWeight“ beschreibt ebenfalls die Wahrscheinlichkeit, dass das Wort ein Name oder Typ ist.

ReferenceInformant Dieser Informant ist Teil der Elementverknüpfung. Dieser geht dabei alle extrahierten Modellelemente durch und überprüft, ob in der Text-Extraktion Namen vorkommen, die ähnlich zu den Modellelementnamen sind. Falls diese ähnlich sind, speichert er diese neuen Textelemente ab. Die Textelemente erhalten den Parameter „probability“ als ihren Konfidenzwert. Die Ähnlichkeitsvergleiche basieren auch hier auf Jaro-Winkler und Levenshtein.

SeparatedNamesInformant Dieser Informant ist Teil der Textextraktion und überprüft alle Wörter nach Trennzeichen. Falls sie diese besitzen, wird das Wort an dem Trennzeichen unterteilt und als zwei verschiedene Namen gespeichert. Die Wahrscheinlichkeit ihrer Korrektheit wird durch den Parameter „probability“ widergespiegelt.

Metrik	Paramter	bestehender Wert
Levenshtein	levenshteinMinLength	2
Levenshtein	levenshteinMaxDistance	1
Levenshtein	levenshteinThreshold	0.90
Jaro-Winkler	jaroWinklerSimilarityThreshold	0.90

Tabelle 4.2: Wortsähnlichkeitsmetriken und ihre Paramter

Zudem sind auch noch Heuristiken zu erwähnen, welche nicht direkt Teil von Informanten sind, sondern sich um Wortähnlichkeitsvergleiche kümmern. Diese werden ebenfalls in verschiedenen Informanten benutzt und haben demnach auch einen nicht zu vernachlässigenden Einfluss auf die TLR. Bei den Wortähnlichkeitsvergleichen wird die Jaro-Winkler-Wortähnlichkeit und die Levenshtein-Distanz genutzt, welche in Unterabschnitt 2.3.2 näher erläutert worden sind. Parameter die man modifizieren kann, sind aus Tabelle 4.2 zu entnehmen. Bis auf den Threshold (dt. Schwellenwert) der beiden Wortähnlichkeitenmetriken sind die übrigen Parameter bereits in Unterabschnitt 2.3.2 erläutert worden. Der Schwellenwert in diesem Kontext sagt aus, ab welchem Ergebniswert der beiden Wortähnlichkeitsmetriken zwei Wörter als ähnlich definiert werden sollen. Bei der Jaro-Winkler-Wortähnlichkeit ist dieser Schwellwert und seine Funktion schnell ersichtlich, da dort bei der Wortähnlichkeitsberechnung Werte zwischen null und eins erzeugt und dann mit dem Schwellenwert verglichen werden. Bei der Levenshtein-Distanz ist es weniger klar erkennbar, da man als Ergebnis einen Wert erhält, der die Anzahl an Operationen beinhaltet. Dieser Wert liegt also nicht im Bereich von null bis eins. Der Schwellenwert wird hier zur Bestimmung der maximalen Distanz benutzt, wie man im folgendem Quelltextausschnitt sieht.

$$int maxDynamicDistance = (int) Math.min(this.maxDistance, this.threshold * Math.min(firstWord.length(), secondWord.length()));$$

Diese dynamisch für die jeweiligen zwei Wörter erzeugte maximale Distanz wird dann mit der tatsächlichen Distanz verglichen. Wenn die tatsächlichen Distanz größer als die maximale Distanz ist, werden die beiden Wörter als nicht ähnlich angesehen.

4.4 Durchgeführte Anpassungen an den Heuristiken

Die Parameter der Levenshtein-Distanz mit der Mindestlänge von zwei sowie die maximale Distanz von eins sind bereits sinnvoll gewählt. Es sind bei diesen beiden Parametern keine weitere Anpassungen nötig, da diese Werte aussagen, dass Wörter erst ab einer Wortlänge von mindestens zwei verglichen werden und diese sich um maximal eine Operation unterscheiden. Dadurch gewährleistet man sinnvolle Vergleiche. Es werden somit Wörter, die nur aus einem Buchstaben bestehen, nicht verglichen. Dieser Vergleich ist bereits präzise eingestellt, da sonst zu viele Wörter als ähnlich deklariert werden.

Die Schwellenwerte der beiden Wortähnlichkeitsmetriken werden angepasst, indem der Schwellwert jeweils von 0.9 auf 0.95 erhöht wird. Diese Erhöhung dient dazu, den Wortvergleich zu verschärfen, da in der NLSD meist die exakten Namen der Quelltextelemente genannt werden. Somit verknüpft man weniger ähnliche Worte mit diesem Element und erhält dadurch weniger inkorrekte TLs.

Der Einfluss der Schwellenwerterhöhung ist Tabelle 4.3 zu entnehmen. Im Projekt Teastore wird die Präzision und die Ausbeute, um vier bzw. einem Prozentpunkt, leicht erhöht. In den Projekten Mediastore und JabRef sind die Auswirkungen weit aus größer und bewirken einen Anstieg der Präzision um 15 Prozentpunkte in Mediastore bzw. 14 Prozentpunkte in JabRef. Der Ausbeutungswert dieser beiden Projekte bleibt unverändert.

Die Anpassung des Schwellenwertes bringt jedoch auch eine Einbuße von einem Prozentpunkt in der Ausbeute des Projektes Teammates. Dies geht aber einher mit einer Erhöhung der Präzision von ebenfalls einem Prozentpunkt. Insgesamt sieht man deutlich, dass diese Änderung mehr Positives als Negatives mit sich bringt, weswegen diese Anpassung der Heuristik sinnvoll ist, Vergleich Tabelle 4.3.

Projekt	Metrik	Einfluss
Teastore	Präzision	+0.04
Teastore	Ausbeute	+0.01
Teammates	Präzision	+0.01
Teammates	Ausbeute	-0.01
Mediastore	Präzision	+0.15
Mediastore	Ausbeute	+0.00
JabRef	Präzision	+0.14
JabRef	Ausbeute	+0.00

Tabelle 4.3: Verbesserungen durch Jaro-Winkler und Levenshtein Schwellwert Erhöhung

Außerhalb der Anpassung der Schwellenwerte haben die anderen Informanten bzw. Heuristiken, die in Tabelle 4.1 zu sehen sind, keinen Einfluss auf die Ergebnisse. Die einzige Ausnahme ist der NounInformant, der veränderte Ergebnisse liefert. Jedoch sind diese Veränderungen minimal im Bereich von einem bis drei Prozentpunkte, und nicht immer positiv. Dies führt dazu, dass keine weiteren Anpassungen an den bisher bestehenden Heuristiken durchgeführt werden.

Man sollte die bestehenden Heuristiken jedoch nicht außer Acht lassen, da sie eventuell im Zusammenspiel mit neu implementierten Heuristiken einen größeren Einfluss bekommen könnten. Dies könnte mit den Heuristiken dieser Bachelorarbeit sowie eventuell weiteren Heuristiken in der Zukunft evaluiert werden. Im zeitlichen Rahmen dieser Bachelorarbeit konnten diese weiterführenden Modifikationen nicht evaluiert werden.

5 Heuristiken für die Traceability Link Recovery von Quelltext

In diesem Kapitel werden die neu implementierten Heuristiken vorgestellt. Dazu wird ihre Funktionsweise, Implementierung im Projekt und was sie bewirken sollen erläutert.

5.1 Heuristiken aus verwandter Literatur

Die in Unterabschnitt 2.3.1 erläuterten Heuristiken werden durch die in diesem Kapitel vorgestellten neuen Heuristiken ergänzt. Diese neuen Heuristiken stammen aus anderen Arbeiten zur Trace Link Recovery (TLR) zwischen Quelltext und der natürlichsprachliche Software Dokumentation (NLSD) sowie eigen erdachte Heuristiken. Das Ziel dieser Heuristiken ist es, die TLR von ArDoCo für den Rahmen dieser Arbeit zu verbessern. Um das Ziel zu verifizieren, werden die neuen Heuristiken evaluiert. Das Vorgehen der Evaluation wird in Kapitel 6 erläutert.

Chen und Grundy [22] befassen sich mit dem Vector Space Model (VSM) und erweitert dieses mit regulären Ausdrücken, Schlüsselphrasen und Clustering. Dabei sind hier primär die Heuristiken interessant, da Chen und Grundy mit diesen gute Ergebnisse erzielen. Die regulären Ausdrücke finden Passagen in der Dokumentation, welche die extrahierten Elemente direkt erwähnen. Die Schlüsselphrasen hingegen werden aus den Kommentaren des Quelltextes erzeugt und dienen dazu die Trace Links zu gewinnen. Diese werden benutzt, wenn es keine Namenskonventionen gibt und somit dasselbe Element unterschiedlich in den jeweiligen Artefakten benannt ist. Die letzte Heuristik dieses Ansatzes, Clustering, ist bereits in ArDoCo implementiert, wie in Unterabschnitt 2.3.1 beschrieben ist.

Das Vorgehen von Antoniol et al. [1] ist ähnlich zu ArDoCo, verfügt aber ebenfalls über drei neue Heuristiken. Diese beziehen sich auf die Vorverarbeitung der NLSAD. Als erstes werden alle Großbuchstaben in Kleinbuchstaben umgewandelt. Danach werden Stoppwörter wie Artikel, Nummern, Satzzeichen etc. entfernt. Im letzten Schritt der Vorverarbeitung wird eine morphologische Analyse durchgeführt und damit alle Plurale in Singulare umgewandelt, sowie alle konjungierten Verben in ihren Infinitiv, also deren Grundform. Mithilfe dieser Heuristiken kann ArDoCo voraussichtlich in der Phase der Textextraktion verbessert werden.

In der Arbeit von Settimi et al. [19] wird ein Thesaurus benutzt, um das Vokabular der potenziellen TL Elementen einzuschränken. Ein Thesaurus ist dabei eine Ansammlung an Wörtern mit seinen Synonymen. Die Einschränkung des Vokabulars bezieht sich darauf, welche Wörter präferiert werden und welche nicht. Diese Einschränkung kann in ähnlicher Form in der Phase der Elementverknüpfung zum Filtern der TLs benutzt werden.

5.2 Neue Informanten

In diesem Abschnitt werden die neu hinzugefügten Informanten und ihre Funktion erläutert sowie ihre Implementierung näher gebracht. Die Informanten sind dabei in der Reihenfolge ihrer Ausführung geordnet. Die neuen Informanten werden in drei neuen Agenten sowie einer neuen Modellextraktion untergebracht. Diese sind in Abbildung 5.1 farbig markiert.

Dabei bestehen die beiden Agenten CompoundModelElementsNamesAgent und FilterNounMappingAgent jeweils nur aus einem gleichnamigen Informanten. Der CodeModelAgent besteht jedoch aus mehreren Informanten, wie es in Abbildung 5.2 zu sehen ist. Die Codeextraction mit seinen Bestandteilen ist bereits in Abschnitt 4.2 näher erläutert worden.

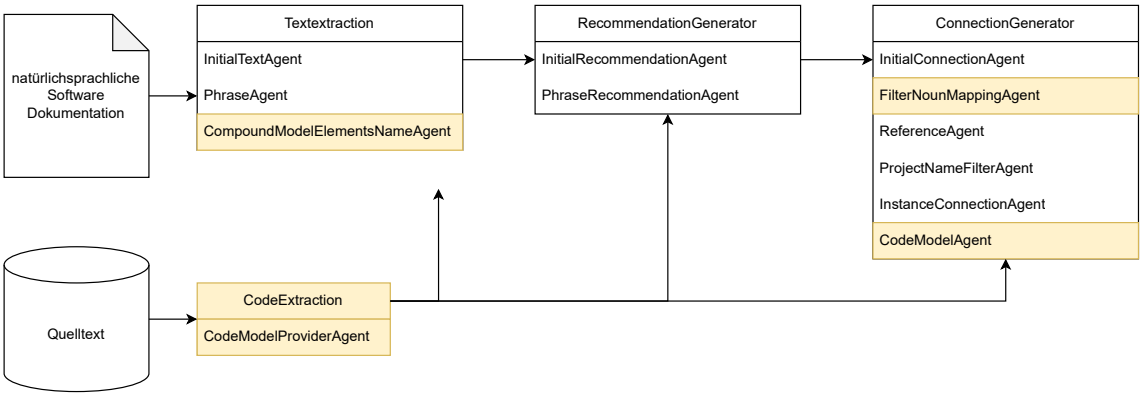

Abbildung 5.1: ArDoCo Architektur mit neuen Agenten

5.2.1 Hinzufügen nicht gefundener NounMappings

Im Projekt Mediastore ist beobachtet worden, dass ein entscheidendes NounMapping fehlt, das als Textelement für viele TLs benutzt wird. Genau solche nicht gefundenen NounMappings werden mithilfe dieses Informanten gefunden und abgespeichert, sodass daraus RecommendedInstances erstellt werden können. Diese RecommendedInstances werden dann weiter für das Erstellen von Instance links benutzt, aus denen schlussendlich TLs gewonnen werden.

Dieser Informant ist Teil der Textextraktion und iteriert über alle extrahierten Modellelementen. Für jedes dieser Modellelemente wird überprüft, ob der Name in CamelCase auftritt und, ob der Name bereits in den NounMappings enthalten ist. Der CamelCase ist dabei eine Schreibweise, bei der Wörter die laut Rechtschreibung nicht zusammengeschrieben werden doch zusammengesetzt werden und das folgende Wort mit einem Großbuchstaben beginnt. Beispiel dafür ist das Wort CamelCase, welches genau in seiner eigenen Schreibweise dargestellt ist. Es wird deshalb nach dem CamelCase des Namens überprüft, da aufgefallen ist, das diese Modellelemente häufig im Text auseinandergeschrieben werden, was es ArDoCo schwerer macht sie zu finden. Das beste Beispiel dafür

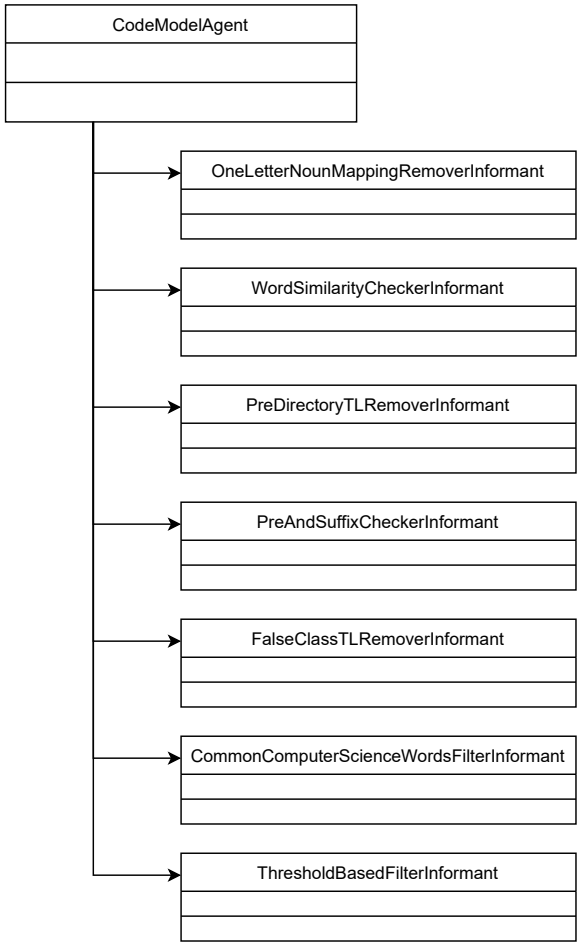

Abbildung 5.2: CodeModelAgent mit seinen Informanten

ist das Modellelement „AudioFile“ vom Typ Klasse. Diese Klasse kommt im Text nur als „audio file“ oder „audio files“ vor, meint jedoch immer die eben genannte Klasse.

Falls ein Modellelement im CamelCase gefunden wird, welches nicht im NounMapping vorhanden ist, werden alle Sätze nach diesem Modellelementnamen abgesucht. Bei der Suche wird jedoch der CamelCase des Namens entfernt, also wir aus dem Namen „AudioFile“ die zwei Wörter „Audio File“. Da die NLSD alle in der englischen Sprache geschrieben sind, wird der aufgeteilte Name noch kleingeschrieben. Wenn dann ein Satz gefunden wird, in welchem die zwei Wörter des Namens hintereinander vorkommen, werden diese Wörter als Typ „Word“ gespeichert und weiter die Sätze iteriert, bis alle Vorkommen gefunden werden. Alle gefunden Wörter die als „Word“ abgespeichert sind, werden nun benutzt, um ein NounMapping zu erstellen und denen der bestehenden Liste an NounMappings hinzuzufügen.

5.2.2 Filtern der gefundenen Textelementen

Bei den ersten Beobachtungen der TLs ist aufgefallen, dass einige der inkorrekten TLs durch ihre entsprechenden NameMappings entstehen. Diese NameMappings sind Ansammlungen von in der NLSD gefundenen Namen. Wenn diese Namen ähnlich sind, werden diese zu einem NameMapping zusammengefügt.

In den NameMappings ist aufgefallen, dass es Wörter enthält, die nicht zu den anderen enthaltenen Wörter ähnlich sind. Dies lässt sich gut anhand zwei NameMappings aus dem Projekt Teammates erkennen.

Das erste NameMapping besteht aus allen Vorkommen der Wörter „Test, tests, REST, Jest“ in der NLSD. Die ersten beiden Wörter „Test“ und „tests“ kommen dabei deutlich häufiger als die anderen beiden Wörter „REST“ und „Jest“ im NameMapping vor. Außerdem wird dieses NameMapping mit dem Paket „test“ verknüpft, welches eindeutig zu den ersten beiden Wörtern zugewiesen werden kann. Hingegen meint „Jest“ das Testframework mit eben diesem Namen, wie aus folgendem Satz zu erkennen ist: „[...] are tested separately with Jest.“. Aus dem Satz „[...] back end logic is designed as a REST-ful controller.“ lässt sich erkennen, dass „REST“ eine Designentscheidung für die back-end-Logik darstellt.

Dieses fehlerhafte NameMapping ist bereits in Abschnitt 4.1 aufgegriffen und als fehlerhaft betitelt worden. Durch den neuen Informanten FilterNounMappingInformant wird dieses Problem gelöst und die Vorkommen der beiden Wörter „REST, Jest“ aus dem NameMapping entfernt. Der Informant ist dabei nach dem letzten Hinzufügen von NounMappings am Anfang des ConnectionGenerator angesiedelt.

Das zweite inkorrekte NameMapping besteht aus allen Vorkommen der Wörter „common, Common, common.util, common.exceptions, common.datatransfer“. Diese Wörter haben das Teilwort „common“ gemeinsam, jedoch beschreiben nur die beiden ersten Wörter keine Paketstruktur. In dieser Paketstruktur, ist jedoch nur das zweite Teilwort hinter dem Punkt entscheidend. Das zweite Teilwort unterscheidet sich jedoch eindeutig von allen anderen Wörtern des NameMappings, sodass dieses Wort inkorrekterweise im NameMapping vorhanden ist.

Der Informant entfernt nicht nur Wörter aus dem NounMapping nach ihrer Ähnlichkeit, sondern auch anhand des Vorkommen von Paketstrukturen. Im zweiten Fall entfernt er alle Paketstrukturen aus de NameMapping „common, Common, common.util, common.exceptions, common.datatransfer“, sodass nur noch „common, Common“ übrig bleibt.

Die gefunden Paketstrukturen sind dennoch wichtig für die TLR, da diese Textelemente ebenfalls Endpunkte von TLs sind, jedoch nicht vom Paket „common“, sondern von ihrem jeweiligen Unterpaket. Diese Unterpakete sind teilweise auch in anderen NameMappings vorhanden, sodass diese daraus gewonnenen TLs nicht vollständig verloren gehen.

In Zukunft lässt sich dieser Informant erweitern, um alle Vorkommen der Wörter beizubehalten, jedoch in anderen NameMappings. Dazu kann man die entfernten Wörter in allen bestehenden NameMappings suchen und falls dieses Wort nicht in bestehenden NounMappings vorhanden ist, ein neues NounMapping erstellen. Voraussetzung ist dafür jedoch, dass der Informant weiter nach vorne in der Pipeline platziert wird, sodass danach noch Textelemente aus den NounMappings erstellt werden können.

5.2.3 Filtern mittels Wortähnlichkeit der Namen

Bei genaueren Analysen der Instance Mappings, welche dazu genutzt werden die finalen TLs zu erzeugen, sind Unstimmigkeiten aufgefallen. Dort werden Text und Model Entität verknüpft, die keinen ähnlichen Namen besitzen. Ein Instance Mapping besteht dabei aus Informationen des Modellelementes und des Textelementes. Das Modellelement ist dabei mit seiner UID, seinem Namen und seinem Typ abgebildet. Das Textelement ist mit seinem Namen, Vorkommen in den entsprechenden Sätzen, sowie seinem Typ abgebildet.

Beispielsweise ist in den Instance Mappings des Projektes JabRef aufgefallen, dass verschiedene Modellelemente wie etwa „customimporter, customentrytypes“ mit dem Textelement „bus“ verknüpft oder „documentviewer“ auf „We“, welches jeweils inkorrekt ist, da die Namen der beiden Elemente nicht ähnlich sind. Ähnliches Verhalten wird auch in den anderen Projekten beobachtet. Alle diese genannten TLs werden von dem InstanceConnectionInformant generiert und mit einem Konfidenzwert von 1.0 oder 0.8 ausgestattet. TLs ohne Typ bekommen den Konfidenzwert 0.8 zugewiesen, wohingegen TLs mit einem Typen einen Konfidenzwert von 1.0 bekommen. Manche von den im InstanceConnectionInformant gewonnenen TLs sind zudem noch in ihrem Konfidenzwert von den eigenen Informanten des CodeModelAgents modifiziert worden.

Damit diese inkorrekten Instance Mappings nicht zu TLs werden, wird dieser Informant erstellt. Dieser Informant betrachtet nochmals die textuelle Repräsentation der eben erläuterten Instance Mappings und fügt bei zu großer Abweichung eine negative Konfidenz hinzu. Dies reduziert die momentan Konfidenz dieses Instance Mappings. Der Informant holt sich zuerst alle Instance Mappings und iteriert über diese und überprüft, ob die Namen des Modellelementes und des Textelementes ähnlich sind. Diese Überprüfung wird von der Helferklasse „SimilarityUtils“ von ArDoCo übernommen, welche auch in ArDoCo selbst benutzt wird. Diese Methode arbeitet mit dem Jaro-Winkler-Ähnlichkeit und der Levensthein-Distanz, welche bereits in Unterabschnitt 2.3.2 erläutert worden sind. Um die Spezialfälle von mit Leerzeichen getrennten Wörter im Text zu erfassen, wird das Leerzeichen entfernt und somit die beiden Wörter zusammengesetzt.

5.2.4 Nach Ordnerstruktur filtern

In den ersten Experimenten hat sich herausgestellt, dass ArDoCo zum Beispiel im Projekt Mediastore TLs gewonnen hat zu Paketen, die nicht zum Projekt gehören. In der Ordnerstruktur der meisten Projekten, ist die Institution, welche an dem Projekt arbeiten, vorhanden. Diese TLs werden von dem InstanceConnectionInformant erstellt und mit einem Konfidenzwert von 0.8 oder 1.0 versehen, wie es bereits in Unterabschnitt 5.2.3 beschrieben worden ist.

Im Beispiel von Mediastore ist es das KIT, wie aus folgender UID zu entnehmen ist „Implementation/mediastore.ejb.mediamanagement/src/edu/kit/ipd/sdq/mediastore“. Dort bezeichnet erst das zweite Auftreten des Projektnames den Anfang des Projektes, in welchem die relevanten Ordner mit entsprechenden Klassen liegen. Somit sind TLs zu den Paketen kit „Implementation/mediastore.ejb.mediamanagement/src/edu/kit“ oder ipd „Implementation/mediastore.ejb.mediamanagement/src/edu/kit/ipd“ nicht korrekt und

sollten nicht mit Textelementen verknüpft werden. Das Verhindern solcher TLs übernimmt der PreProjectDirectoryTLRemoverInformant.

In den betrachteten Projekten gab es Unterschiede in der Ordnerstruktur, wobei nur das Projekt Teammates auf zweimaliges nennen des Projektnamens verzichtet und stattdessen UIDs wie folgt aussehen: „src/client/java/teammates/client“. Die Gemeinsamkeit aller Projekte besteht jedoch darin, dass in der Ordnerstruktur der Projektname immer als einzelner auftritt, wenn danach die relevanten Pakete und Klassen kommen. Also iteriert der Informant über alle TLs und überprüft, ob die Sequenz „Projektname~/“ in der UID vorkommt. Wenn dies nicht der Fall ist, wird dem TL ein negativer Konfidenzwert hinzugefügt. Man kann deshalb nach dieser Sequenz filtern, da das erste Vorkommen des Projektnamen, mit „.“ statt „/“ getrennt ist, wie es in dem gezeigtem Pfad von Mediastore der Fall ist.

5.2.5 Filtern anhand Pre- und Suffix

Dieser Informant vergleichen die Pre- und Suffixe der Modellelemente und der gefundenen Entitäten der NLSD, die mit einem TL verknüpft sind. Dies ist deshalb relevant, da zum Beispiel im Projekt Mediastore viele TLs fälschlicherweise auf Interfaces statt mit den in der NLSD beschriebene Pakete verknüpft werden. Dieses Verhalten trifft deshalb auf, da Endpunkte in der NLSD oder dem Quelltext in mehreren verschiedenen TLs vorkommen dürfen. Im Projekt Mediastore ist es jedoch wichtig, da manche Klassen, Interfaces und Pakete sich nur durch Pre- und Suffixe unterscheiden, in der NLSD jedoch nur eins davon gemeint ist. Ein anschauliches Beispiel dafür ist das Paket „mediamanagement“, welches die Klasse „MediaManagementImpl“ enthält und es zusätzlich in dem Paket „basic.interfaces“ ein Interface mit dem Namen „IMediaManagement“ gibt. Es ist klar zu erkennen, dass sich diese Namen nur durch in der Informatik gängige Pre- und Suffixe unterscheidet wie das „I“ für Interfaces, oder das „Impl“ für Klassen.

Es wird über alle TLs iteriert und falls der Name des Modellelementes das entsprechende Pre- oder Suffix besitzt, wird es mit dem Namen des Element aus der NLSD verglichen und bei keiner Übereinstimmung des Pre- bzw. Suffixes, wird der TL aus der Liste der TLs entfernt. Die zu überprüfenden Pre- und Suffixe sind konfigurierbar, sowie weitere Parameter fast aller Agenten und Informanten von ArDoCo.

Es besteht dennoch die Gefahr, dass in der NLSD eben solche Interfaces bzw. Klassen gemeint sind, jedoch ohne das entsprechende Pre- bzw. Suffix. Dieser Fall ist aber nicht in den betrachteten ArDoCo Benchmark Projekten aufgetreten, sodass diese Implementierungsentscheidung getroffen und das Risiko für andere Projekte eingegangen wird.

5.2.6 Filtern mittels inkorrekter Klassen

Das Problem, welches dieser Informant angeht, ist dass nicht korrekte Klassen mit Textelementen verknüpft werden. Man kann davon ausgehen, dass der Name einer Klasse in der NLSD exakt so geschrieben vorkommt wie die Klasse im Quelltext benannt ist. Wenn der Klassenname nicht in der exakten Schreibweise vorkommt, kann man davon ausgehen, dass eine andere Klasse gemeint ist. Zuständig für das gewinnen dieser TLs ist

der InstanceConnectionInformant, welche die TLs mit einem Konfidenzwert von 0.8 bzw. 1.0 verseht, wie es bereits in Unterabschnitt 5.2.3 beschrieben worden ist.

Ein Beispiel welche TLs inkorrekt sind, sieht man im Projekt Teastore in diesem Satz: „Data is retrieved from the PersistenceProvider and product recommendations from the Recommender service.“. In diesem Satz wird dem Namen „Recommender“ den Typen Klasse zugeordnet und zu einem Textelement vereint. Dieses Textelement wird dann mit dem Modellelement „RecommenderSelector“ für einen TL verknüpft. Dieser TL ist inkorrekt, was man an den nicht übereinstimmenden Namen der beiden Elementen sehen kann.

Um die inkorrekt verknüpften Klassen der TLs zu entfernen, werden den TLs negative Konfidenzwerte hinzugefügt. Im Detail werden sich alle NameMappings der Textelemente aus der NLSD angeschaut und mit dem Modellelement Namen verglichen. Um dies korrekt durchzuführen, muss der Spezialfall der zusammengesetzten Wörter beachtet werden. Falls es sich um ein zusammengesetztes NameMapping handelt, werden die ersten beiden Wörter des NameMappings zusammengesetzt, bevor sie mit dem Modellelementnamen verglichen werden. Falls einer der beiden Vergleiche zutrifft, wird dem TL ein negativer Konfidenzwert hinzugefügt.

5.2.7 Filtern durch allgemeine Informatik Begriffe

ArDoCo besitzt in ihrem Bereich der Inkonsistenzerkennung eine Filterliste, in der allgemeine Informatik Begriffe stehen, die nicht in der Inkonsistenzerkennung beachtet werden sollen. Diese Filterliste hat jedoch keinen Einfluss auf die TLR, obwohl dort ebenfalls solche allgemeinen Wörter TLs zugewiesen werden.

Im Projekt JabRef werden TLs gewonnen, die auf das Paket „architecture“ zeigen, weil das Wort in zwei Sätzen vorkommt. Zum Einen „We have been successfully transitioning from a spaghetti to a more structured architecture [...]“ und zum Anderen „This allows us to keep the architecture [...]“. In beiden Sätzen ist jedoch nicht das Paket gemeint, sondern die Architektur des Software-Projektes.

Um TLs auf solche Textelemente zu verhindern, wird der Informant erstellt. Jedoch muss man vorsichtig in der Wahl der Wörter sein, welche man filtern möchte. Durch diese Filterliste kann man für jedes Projekt hohe und eventuell sogar perfekte Präzisionswerte bekommen, jedoch muss man dann die Liste von Projekt zu Projekt neu anpassen, was den Sinn dieser Arbeit der automatischen TLR verfehlt.

Die List der Begriff zum Filtern ist konfigurierbar, sodass man in Spezialfällen nachträglich noch Wörter hinzufügen kann. Die aktuell gewählt Konfiguration besteht aus den vier Wörtern „architecture, file, data, instance“. Diese sind allgemeine Informatik Begriffe, welche in den betrachteten ArDoCo Benchmark Projekten auftreten. Dennoch treten nicht alle Wörter in allen Projekten auf, sodass in etwa ein oder zwei Wörter pro Projekt gewählt werden. Diese geringe Anzahl an Wörtern erlaubt es Einfluss, jedoch keinen großen, auf die Ergebnisse zu nehmen.

Um eine perfekt passende Konfiguration an Wörter zu finden, muss eine eigenständige Evaluation dafür stattfinden. Dies kann in Zukunft gemacht werden, da dieser Informant durchaus wichtig ist. Im Rahmen dieser Bachelorarbeit war die Zeit dafür nicht ausreichend, sodass nur Wörter mit wenig Einfluss auf die Ergebnisse ausgewählt werden.

5.2.8 Filtern durch Konfidenzwert

Bei den ersten Experimenten ist aufgefallen, dass der Projektname als Modellelement vom Typ Paket in den TLs vorkommt. Diese Pakete sollten eigentlich in der Grundform von ArDoCo bereits ausgefiltert werden. Die Konfidenz der zugehörigen RecommendedInstance werde bereits von ArDoCo auf minus unendlich gesetzt. Jedoch werden die daraus entstandenen TLs mit negativer Konfidenz bisher nicht entfernt. Dieser neue Informant, namens ThresholdBasedFilterInformant entfernt all diejenigen TLs, welche eine Konfidenz unter einem konfigurierbaren Konfidenzwert besitzen. Somit fallen die erwähnten TLs zu den Paketen des Projektnames raus. Dies sind jedoch nicht die einzigen TLs, die von diesem Informanten entfernt werden. Alle anderen Informanten des CodeModelAgent reduzieren nur die Konfidenzen der untersuchten TLs anstatt sie direkt zu entfernen. Diese Konfidenzwerte können beim Durchlaufen der verschiedenen Informanten ebenfalls unter den konfigurierbaren Konfidenzwert fallen, sodass sie von diesem Informanten entfernt werden.

6 Evaluation

In diesem Kapitel werden die aus den verschiedenen Arbeitspaketen erhaltenen Ergebnisse evaluiert. Um das umzusetzen werden zunächst Goldstandards für die benutzten Projekte erstellt werden. Dies wird in Abschnitt 6.1 genauer erläutert. Die Evaluation der Ergebnisse erfolgt anhand den Metriken, welche in Abschnitt 6.2 erläutert werden. Nachdem man die Goldstandards erstellt und die Metriken definiert hat, folgt die Ausführung von ArDoCo in seinem neuen Kontext, sodass man die ersten Ergebnisse erhält, welche in Abschnitt 6.3 evaluiert werden. Diese Ergebnisse werden in Abschnitt 6.4 durch Anpassungen an den bestehenden Heuristiken optimiert und anschließend ebenfalls evaluiert. Da die Heuristiken jedoch nicht für die Trace Link Gewinnung zwischen der natürlichsprachlichen Software Dokumentation (NLSD) und Quelltext ausgelegt sind, werden neue passende Heuristiken gesucht und implementiert. Die dadurch entstandenen Ergebnisse werden in Abschnitt 6.5 evaluiert.

6.1 Goldstandard

Ein Goldstandard ist in unserem Kontext eine Liste an manuell erstellten Trace Links (TLs) zwischen den Software-Artefakten NLSD und dem Quelltext. Eine Erklärung was Trace Link (TL) sind, ist in Abschnitt 2.1 zu finden.

Nicht jedes Projekt bietet sich zur Erstellung eines Goldstandards an. Das Projekt benötigt eine natürlichsprachliche Software Dokumentation. Ohne diese können offensichtlich keine TLs erstellt werden. Bestenfalls sollte die NLSD aktuell sein und somit wenig Inkonsistenzen zum Quelltext aufweist. Dies kann jedoch nicht garantiert werden, außerdem richtet sich diese Arbeit an genau solche Projekte. Das Ziel dieser Arbeit ist dennoch nicht der Umgang mit Inkonsistenzen und somit sind Projekte ohne diese zu bevorzugen. Außerdem sollte das Projekt nicht zu groß sein, sowie eine übersichtliche NLSD besitzen. Bei großen Projekten ist der Aufwand zum Erstellen eines Goldstandards zu zeitaufwendig, sowie fehleranfälliger. Alle diese Kriterien treffen auf die Projekte zu, welche sich bereits im ArDoCo Benchmark [9] befinden. Enthaltende Projekte dort sind: Teammates, Teastore, Mediastore, JabRef und BigBlueButton [9]. Was diese Projekte für jeweilige Funktionen erfüllen, ist im Kontext dieser Arbeit nicht relevant und wird somit auch nicht näher erläutert.

Das Erstellen von Goldstandards ist keinesfalls trivial, sondern birgt verschiedene Fehlerquellen. Einige davon werden im Laufe dieser Sektion erläutert, sowie ihre Lösung in diesem Kontext näher gebracht.

Eine allgemeine Gefahr besteht beim Erstellen der Goldstandards darin, sich zu sehr an den erzeugten TL von den Projekt zu orientieren. Wenn die hinzugefügten TLs sich an den bisherigen Heuristiken anpassen, entsteht eine Überoptimierung an das Problem. Um

dies zu verhindern und die finale Version der Arbeit zu überprüfen, wird ein Projekt der ArDoCo Benchmark vorerst nicht in die Evaluation aufgenommen. An diesem Projekt wird am Ende der Prototyp getestet und somit die TLR ausgeführt, ohne die gewonnenen Trace Links mit einem Goldstandard zu evaluieren. Die gewonnenen TLs werden mit Augenmerk angeschaut und überprüft, ob plausible Ergebnisse erstellt werden. Das Projekt, dass dabei außen vor genommen wird, ist BigBlueButton.

Es sind jedoch noch weitere Probleme beim Erstellen des Goldstandards aufgekommen. Die NLSD der verschiedenen Benchmark Projekten beinhaltet einige Stellen, die unklar sind, sodass eine klare Aussage, welche Entität gemeint ist und im welchem Satz sie vorkommt schwierig zu treffen ist. Faktoren, die dabei eine Rolle gespielt haben, sind Sonderzeichen wie Doppelpunkt und Semikolon. Zum Vorteil dieser Arbeit lagen bereits vorverarbeitete NLSD vor, anhand dieser zumindest solche Unklarheiten entfernt werden konnten. Diese vorverarbeiteten NLSD stammen von ArDoCo, welche vor dem selben Problem standen.

Ein Nachteil der bestehenden vor verarbeiteten NLSD von ArDoCo ist, dass die Vorverarbeitung nicht von der aktuellsten Version der NLSD stammt und es somit zu Inkonsistenzen zum Quelltext kommt. Es wird also über Klassen oder Pakete geschrieben, welche nicht mehr im Quelltext vorkommen. Dies stellt aber kein Problem dar, da die Pipeline somit auch keine entsprechende Entität im Quelltext findet und dadurch auch kein Trace Link erstellen sollte. Durch die Vorverarbeitung wird mehr Klarheit beim Erstellen der manuellen Trace Link kreiert und somit präzisere Goldstandards erstellt. Die Pipeline des Projektes arbeitet ebenfalls mit der vorverarbeiteten Version der NLSD, um Gleichheit zu ermöglichen.

Es treten jedoch noch weitere Probleme auf, die nicht durch eine Vorverarbeitung gelöst werden konnten. Es werden in wenigen NLSDs über Klassen von Bibliotheken geschrieben, welche nicht im Quelltext des Projektes vorkommen und somit auch nicht im Goldstandard übernommen werden. Es gibt auch Schreibfehler in der NLSD von zum Beispiel Teammates in dem ein Paket „common.exceptions“ beschrieben wird, welches aber im Quelltext unter dem Namen „common.exception“ vorkommt.

Ein weiteres Beispiel für Schreibfehler bzw. Interpretationen ist in der NLSD von JabRef im Satz: „The model represents the most important data structures (BibDatases, BibEntries, Events, and related aspects) and has only a little bit of logic attached.“ vorhanden. Hier ist die Klasse „BibDatabases“ falsch geschrieben, wird dennoch zusammen mit „BibDatabase“ in den Goldstandard aufgenommen. Ebenfalls wird eine Klasse BibEntries erwähnt, welche jedoch nicht im Quelltext vorkommen. Mit diesem Textelement ist die Klasse „BibEntry“ gemeint, was eine Interpretation des Textes erfordert.

Ein weiteres Problem ist der Umgang mit dem Kontext des Satzes. In der NLSD von Mediastore wird das Textelement „user“ häufig genannt, jedoch in unterschiedlichen Kontexten. Anhand von folgenden Ausschnitten, soll der Entscheidungsprozess beim Erstellen des Goldstandard nähergebracht werden. Im ersten Beispielsatz „When a user logs into the system [...]“ wird das Modellelement User vom Typ Klasse gemeint, da in diesem Projekt der User als Klasse dargestellt ist. Jedoch im Kontext vom Satz „To meet the user authentication requirement [...]“, ist nicht direkt der User an sich gemeint, sondern seine Authentifikation, welches einen TL nicht rechtfertigt. Im letzten Beispiel „By contrast, the UserDBAdapter component provides all functions required in order to encapsulate

database access for the user data.“ könnte man auch meinen, dass nur die Daten des Users gemeint sind und deswegen kein TL in den Goldstandard hinzugefügt werden sollte. Jedoch sind die Daten eines Users in der Klasse User gespeichert, sodass dieser TLs relevant ist und dementsprechend dem Goldstandard hinzugefügt wird.

Des weiteren werden vereinzelt auch Abkürzungen von Klassen in der NLSD benutzt. Nur solch die ebenfalls in ihrer Abkürzung als Modellelement bestehen, werden in den Goldstandard aufgenommen. Die Abkürzung UI steht für User Interface, jedoch sind die Klassen im Quelltext ebenfalls nur mit UI benannt. Ein Beispiel für nicht hinzugefügte Abkürzungen befindet sich in der NLSD von Teammates im Satz „Represented by the Db classes.“, wobei „Db“ hier für die Klassen, welche „DataBundle“ im Namen beinhalten, steht. Jedoch kann ArDoCo solche Abkürzungen nicht korrekt interpretieren. Da solche Abkürzungen nur vereinzelt in den NLSDs vorkommen, wird dies im Rahmen dieser Arbeit nicht weiter betrachtet. Der Nutzen für von solch einer neuen Heuristik wäre wesentlich geringer, als ihr Zeitaufwand beim Erstellen. Jedoch ist solch eine neue Heuristik durchaus möglich und wäre eine potenziell Idee für weitere Ausarbeitungen.

Es kommt auch oft vor, dass mehrere gleichnamige Entitäten im Quelltext vorkommen. In Teastore gibt es zum Beispiel das Paket „Service“ und die Klasse „Service.java“. Dadurch weiß man nicht zuverlässig von welcher der beiden Entität gerade die Rede ist, was zu potenziell fehlerhaften Trace Links führt. Als Richtlinie wird dafür der Kontext hinzugezogen. Falls von einer Komponente bezüglich diesem Namen geredet wird, bezieht es sich auf das entsprechende Paket. Im Beispiel von Mediastore „MediaManagement component“, ist das Paket MediaMangagement gemeint. Wenn im Kontext dieses Namen von einer Instanz geredet wird, ist die Klasse des Textelementes gemeint. Im Fall von Teastore „service instances“, ist also die Klasse gemeint. Dieses Unterscheidungskriterium lässt sich auf diese Projekte anwenden, kann aber bei anderen Projekten mit weniger präzisen NLSD zu Unstimmigkeiten führen. Dieses Problem wird noch erweitert, indem es auch noch mehrere gleichnamige Pakete gibt. In Teammates gibt es zum Beispiel zu jedem Paket ein dazugehöriges gleichnamiges Paket, in dem die Tests dafür enthalten sind. Da die Pipeline nicht zwischen diesen unterscheiden kann, müssen neue Heuristiken erstellt werden. In den Goldstandard werden nur die jeweils gemeinten Entitäten übernommen, was eine gewisse Interpretation benötigt.

Abschließend werden zudem noch die Namen der einzelnen Projekte in der NLSD nicht in den Goldstandard übernommen. Die Projektnamen sind im Quelltext Paketen im Projekt selbst, erfüllen jedoch keine Funktionen, sondern liefern nur Auskunft über den Namen des Projektes.

6.2 Evaluationsmetriken

Die Ergebnisse der Arbeit sollen mithilfe des in Abschnitt 6.1 erstellten Goldstandards evaluiert werden. Häufig genutzte Metriken zur Evaluierung von TLR sind Präzision, Ausbeute und das F1-Maß. Um diese zu nutzen, werden die verschiedenen gewonnen Trace Links näher klassifiziert. Trace Links (TLs), die automatisch erzeugt und auch im Goldstandard enthalten sind, nennt man true positive (TP). Ein Beispiel dafür ist im Projekt Teastore der Satz 8 „The UI provides [...]“, hier wird „UI“ gefunden und mit dem

Modellelement „ui“ vom Typ Paket verknüpft. Dieser TL ist ebenfalls im Goldstandard vorhanden.

Wenn kein Trace Link erzeugt wird, obwohl einer im Goldstandard ist, ist es ein false negative (FN). Ein Beispiel dafür ist im Projekt Teammates zu finden, dort wird im Satz 87 „Logic API is represented by the classes Logic, GateKeeper, EmailGenerator, EmailSender, TaskQueuer.“ nicht das Texelement TaskQueuer mit seiner namens gleichen Klasse im Modell verknüpft, steht jedoch im Goldstandard.

Wenn wiederum ein Trace Link erstellt wird, der nicht im Goldstandard enthalten ist, wird dieser als false positive (FP) bezeichnet. Ein Beispiel dafür ist im Projekt JabRef im Satz 5 „[...] and has only a little bit of logic attached.“. Dort wird inkorrekterweise dem Textelement „bit“ das Modellelement „git“ vom Typ Paket zugeordnet, obwohl dieser TL nicht im Goldstandard steht.

Im Fall, wenn einem Textelement kein Modellelement zugeordnet werden kann, also kein Trace Link gewonnen wird und zu diesem Textelement auch kein TL im Goldstandard definiert ist, werden diese true negative (TN) genannt.

Mithilfe dieser Unterteilung können die verschiedenen Metriken mit folgenden Formeln berechnet werden. Alle Ergebnisse stellen einen Prozentwert da, wenn der Präzision zum Beispiel eins beträgt, wird kein TL gefunden, der nicht auch im Goldstandard vorhanden ist.

Präzision bezeichnet die Anzahl der TPs in Relation zur Anzahl aller gewonnen Trace Links.

$$Präzision = \frac{TP}{TP + FP} \tag{6.1}$$

Ausbeute bezeichnet das Verhältnis zwischen den TPs und dem Ergebnis und wird mit folgender Formel berechnet:

$$Ausbeute = \frac{TP}{TP + FN} \tag{6.2}$$

Das F1-Maß ist die Zusammensetzung aus Präzision und Ausbeute und wird mit folgender Formel berechnet:

$$F1 - Maß = 2 \cdot \frac{Präzision \cdot Ausbeute}{Präzision + Ausbeute} \tag{6.3}$$

6.3 Evaluation bestehender ArDoCo Heuristiken

Nachdem die Goldstandards und die Metriken definiert sind, beleuchtet dieser Abschnitt die Ergebnisse der Evaluation. Um die Evaluation durchzuführen werden die bisherigen Tests von ArDoCo mit einem neuen Test bezüglich diesem Ansatz der TLR erweitert. Die Grundlage zum Testen ist bereits in ArDoCo vorhanden. Außerdem orientiert sich der Test an den Tests und entsprechenden Klassen und von ArDoCo selbst. Der erstellte Test liefert die Ergebnisse in den einzelnen Metriken in leicht verständlicher Form aus.

Innerhalb der ersten Evaluationen der vorläufigen Ergebnisse sind Fehler im Goldstandard entdeckt und behoben worden. Diese sowie weitere Probleme werden in Abschnitt 6.7 näher erläutert. Die endgültigen Ergebnisse dieses Arbeitspaketes sind in Tabelle 6.2 zu sehen. Außerdem sind auch noch die Ergebnisse ohne den gekürzten Pfad gewonnen worden, wie die Tabelle 6.1 zeigt.

Projekt	Präzision	Ausbeute	F1-Maß
Teastore	0.27	0.97	0.42
Teammates	0.17	0.70	0.27
Mediastore	0.29	0.69	0.40
JabRef	0.19	0.70	0.30
Gesamtübersicht	0.23	0.77	0.35
Gesamtübersicht (gewichtet)	0.21	0.76	0.33

Tabelle 6.1: Ergebnis mit bestehenden Heuristiken ohne gekürzten Pfad

Durch das Nutzen des gekürzten Pfadmodus wird bereits eine höhere Präzision und Ausbeute in den Projekten Teammates und JabRef erzielt. Dies zeigen die Tabelle 6.1 und Tabelle 6.2.

In Teammates steigt die Präzision um zwei und in JabRef um fünf Prozentpunkte. Dies liegt daran, dass mit dem gekürzten Pfad weniger FPs gefunden werden, da nun weniger Ordner mit ähnlichem Pfad vorkommen, die sich nur um die Komponente „main“ bzw. „test“ unterscheiden. Ebenfalls steigt die Ausbeute um neun in Teammates und in JabRef um ganze 20 Prozentpunkte. Das liegt daran, dass viele Ordner aus „src/test/java“ die TL auf die Textelemente bekommen und keine zu der „src/main/java“ gewonnen werden, obwohl die Pakete aus „src/main/java“ im Goldstandard stehen. Dies führt zu weniger TPs, also auch zu einem kleineren Ausbeutungswert. Da durch den gekürzten Pfad diese beiden Ordner zum selben zusammengeführt sind, werden nun mehr von den nicht gefunden TLs entdeckt. Der beschriebene Effekt trifft in dem Projekt JabRef im großen Maße auf, da sich dort die Ausbeute um zwei Prozentpunkte verbessert.

In den Projekten Teastore und Mediastore hat der gekürzte Pfadmodus kaum Einfluss. Dies liegt an der Beschaffenheit der Ordnerstruktur bzw. den im Goldstandard vorkommenden TLs. Das Projekt Teastore besitzt nur wenige Ordner die in „src/main/java“ und „src/test/java“ vorkommen, nämlich pro Modul immer nur ein einziges, sofern in diesem Modul überhaupt ein Testordner vorhanden ist. Zusätzlich dazu besitzt der Goldstandard von Teastore keinen einzigen TL auf einen Ordner, der in „src/main/java“ und „src/test/java“ vorkommt. In Mediastore enthalten die Pfade des Projektes nicht den Teil „src/(main|test)/java“. Somit hat der gekürzte Pfadmodus keinen Einfluss auf die Projekte Teastore und Mediastore.

Projekt	Präzision	Ausbeute	F1-Maß
Teastore	0.27	0.97	0.42
Teammates	0.19	0.79	0.30
Mediastore	0.29	0.69	0.40
JabRef	0.24	0.90	0.37
Gesamtübersicht	0.24	0.84	0.38
Gesamtübersicht (gewichtet)	0.23	0.83	0.35

Tabelle 6.2: Ergebnis mit bestehenden Heuristiken mit gekürzten Pfad

Die Tabelle 6.2 zeigt, dass ArDoCo im neuen Kontext bereits annehmbare Ergebnisse erzielt. Die verschiedenen Werte der Projekte sind relativ ähnlich, wobei Teastore positiv im Fall der Ausbeute und Teammates negativ im Fall der Präzision auffällt.

Die höchsten Präzisionen besitzen die beiden Projekte Teastore mit 27% und Mediastore mit 29%. Im Gegensatz dazu liegt JabRef bei 24% und Teammates bei 19%. Die Ausbeute der Projekte sind unterschiedlich, von Teastore mit 97% über Teammates mit 79% bis hin zu Mediastore mit 69%. Die TLR von ArDoCo ohne weitere Anpassungen ist unterschiedlich von Projekt zu Projekt.

ArDoCo erreicht bereits in dem Projekt Teastore eine Ausbeute von 97%. Die Präzision von Teastore ist nahe an den 100%, weswegen kein weiterer Aufwand darin investiert wird diesen zu verbessern. Selbst die Ausbeute der übrigen Projekten Teammates und JabRef, mit 79% bzw. 90% sind mit Ausnahme von Mediastore mit 69% im Vergleich zu der Präzision hoch. Deswegen fokussiert sich die Bachelorarbeit im weiteren Verlauf auf die Verbesserung der Präzision.

Die Präzision aller Projekte von 19% in Teammates bis zu 29% in Mediastore sind im Vergleich zur Ausbeute gering. Diese Werte zeigen, dass ArDoCo noch zu viele TLs findet, welche keine TPs sind.

Um einen genaueren Einblick in die Entstehung der Werte der einzelnen Metriken zu geben, sind in Tabelle 6.3 die jeweilige Anzahl der klassifizierten TLs pro Projekt enthalten. Diese bieten sich an, um genauere Rückschlüsse der Ergebnisse zu ziehen. Das wird erreicht, indem man die klassifizierten TLs analysiert, um die Quellen der Fehler zu identifizieren. Dieses Wissen bildet die Grundlage für die Umsetzung der Modifikationen und neuen Heuristiken, die speziell auf das erstellen von TLs zwischen Quelltext und NLSD angepasst sind.

Mithilfe der TPs und FPs der verschiedenen Projekte, können außerdem noch Rückschlüsse auf die Größe der jeweiligen Dokumentationen der Projekte gezogen werden. Diese Werte müssen jedoch immer in Relation zu den bereits in Tabelle 6.2 gezeigten Metriken betrachtet werden.

Wie aus den beiden Tabellen: Tabelle 6.2, Tabelle 6.3 zu sehen ist, handelt es sich bei dem Projekt Teammates um das größte Projekt bzw. das Projekt mit der längsten natürlichsprachlichen Dokumentation. Die Dokumentation vom Projekt Teammates besteht aus 198 Sätzen in der vorverarbeiteten Version der Dokumentation. Die dadurch entstandene Menge an Text- sowie Modellelementen ermöglicht viele Verknüpfungsoptionen. Es wer-

den mehr ähnlich Wörter gefunden, die zu einem NounMapping zusammengefügt werden, ohne das sie das selbe Textelement meinen. Außerdem gibt es mehr Modellelemente, die für Verknüpfungen zu einzelnen Textelementen in Frage kommen. Diese Gründe sowie weitere gestaltet die TLR in diesem Projekt am herausfordernsten. Mit 116 TPs und 507 FPs werden schon viele TLs gefunden, im Verhältnis sind es jedoch zu viele inkorrekte TLs, wie die Präzision von 19% deutlich macht. Alle vorverarbeiteten Dokumentationen sind aus dem ArDoCo Benchmark [9] entnommen.

Projekt	True Positives	False Positives	False Negatives	True Negatives
Teastore	60	161	2	10.742
Teammates	116	507	30	163.884
Mediastore	34	85	15	4.417
JabRef	26	84	3	28.708

Tabelle 6.3: Anzahl der klassifizierten Trace Links mit bestehenden Heuristiken

Ein Beispiel für ein FP von dem Projekt Teammates ist die Klasse „ArchitectureTest“, welche häufig vertreten ist. Mit über 30 TLs taucht sie jedoch nie in der Dokumentation auf. Diese inkorrekten TLs werden gewonnen anhand des NounMappings des Textelemente „Architecture“. Dieses Wort wird einmal im Text im ersten Satz gefunden, weshalb das Wort als NounMapping gespeichert wird. Jedoch wird diesem NounMapping noch viele verschiedene Wortvarianten hinzugefügt, welche ArDoCo als ähnlich betrachtet und in der NLSD gefunden hat. Dies liegt an ArDoCos Ähnlichkeits-Heuristik, welche in diesem Fall zu großzügig Wörter in ein NounMapping zusammenfügt. Deswegen gelangen auch Wörter wie „Architecture“ ins NounMapping zusammen mit „Test, tests, REST, Jest, [...]“, obwohl dies zu keinem anderen enthaltenen Wort ähnlich ist. Alle Namensvarianten, die gefunden werden, sind „Architecture, Test, REST, TestNG, Jest, tests, test, test.driver, test.cases“. Jedes dieser Wortvarianten ist in verschiedenen Sätzen gefunden worden und alle werden mit demselben Modellelement zu einem TL verbunden. Es ist anhand der ersten beiden Wörter „Architecture“ und „Test“ erkennbar, warum ArDoCo das Modellement „ArchitectureTest“ ausgewählt hat, da sie zusammen genau den Namen des Modellelementes wiederspiegeln. Jedoch ist klar erkennbar, dass das erste Wort nicht ähnlich zu den Anderen ist. Dennoch sind alle anderen Wortvarainten ähnlich zueinander. Da alle Wörter zu einem NounMapping gehören, werden zu allen Vorkommnissen dieser Wörter, jeweils ein TL gewonnen, obwohl im Einzelfall diese Wörter nicht ähnlich zu dem des Modellelementes ist. Dies reduziert die Präzision erheblich. Jedoch bleibt dieses Beispiel kein Einzelfall, sondern kommt häufiger vor. Zum Beispiel mit der Klasse „Instructor“, die nur einmal in der NLSD vorkommt.

Ein weiteres Beispiel für ein FP ist das Paket „attributes“, welches mit 13 TLs häufig vorkommt, aber nur zweimal in der Dokumentation erwähnt wird. Wie der Satz „Update is done using UpdateOptions inside every Attributes.“ zeigt, bezieht sich die Erwähnung von „attributes“ nicht auf das Paket selbst, sondern auf die Attribute von bestimmten Klassen.

Die hohe Ausbeute der jeweiligen Projekte zeigt, dass ArDoCo bereits schon einige der TLs aus dem Goldstandard findet. Das Projekt Teastore führt die anderen Projekte in diesem Bereich an, mit einem Ausbeutungswert von 97% und 60 TPs im Verhältnis zu seinen zwei FNs. Der Satz „Service instances register themselves at the registry on startup.“ veranschaulicht dieses Verhältnis ganz gut. Hier, sowie in anderen Sätzen, wird die „Service instance“ bereits richtig mit der Service Klasse aus dem Quelltext verknüpft. Andererseits werden noch viele TLs zu Sätzen gefunden, die inkorrekt sind. Das Shell Skript „start.sh“ ist beispielsweise solch ein inkorrekter TL aus dem bereits genannten Satz.

Die einzigen zwei TLs die ArDoCo in Teastore noch nicht findet ist einmal eine spezifische Klasse namens „SlopeOneRecommender“ und das Paket „persistence“ im Satz „The Persistence service provides access to the data persisted in the relational database backend.“. Dies liegt daran, dass ArDoCo nicht den Zusammenhang zwischen „persistence“ und „service“ findet und stattdessen nur die Klasse Service verknüpft. Die hohe Ausbeute sowie die niedrige Präzision im Fall von dem Projekt Teastore ist darauf zurückzuführen, dass ArDoCo bereits schon viele TLs findet auch wenn diese nicht immer korrekt sind. Dieses Verhalten ist auch anhand der Tabelle 6.2 bei den anderen Projekten in unterschiedlichen Ausmaßen zu sehen.

Basierend auf den 15 FPs und einem Ausbeutungswert von 69% in Mediastore könnte die Annahme entstehen, dass ArDoCo in diesem Bereich die größten Schwierigkeiten aufweist. Diese Annahme ist jedoch nicht korrekt, da alle 15 FPs derselben Fehlerursache zuzuordnen sind. Alle 15 FPs bestehen aus der Klasse „AudioFile“, welche in allen TLs des Goldstandards nicht gefunden wird. Dies lässt sich darauf zurückführen, dass kein NounMapping zu dem Wort „AudioFile“ besteht und somit daraus auch keine RecommendedInstance oder ein TL entstehen kann. Das Wort „AudioFile“ tritt nie genauso in der NLSD auf, sondern nur in den beiden Varianten „audio file“ und „audio files“, welche im Kontext jedoch beides mal die „AudioFile“ Klasse meinen. Auf Basis dieser Feststellung versucht der Informant CompoundModelElementsNamesInformant, siehe Unterabschnitt 5.2.1, für solche Fälle NounMappings zu erstellen. Dadurch werden im Projekt Mediastore alle FPs eliminiert und 100% Ausbeute erreicht.

Betrachtet man die Präzision samt seinen FPs im Projekt Mediastore, fallen interessante FPs auf. Bei 85 FPs sind 22 davon Interfaces im Quelltext, die fälschlicherweise mit Klassen der Dokumentation verknüpft werden. Ein Beispiel dafür ist der Satz: „The MediaManagement component coordinates the communication of other components.“. Hier wird statt dem Paket „mediamanagement“ das Interface „IMediaManagement“ verknüpft. Diese fehlerhaften Verknüpfungen werden mithilfe des neuen Agenten PreAndSuffixCheckerInformant, siehe Unterabschnitt 5.2.5, behoben.

Im Projekt JabRef gibt es bei einer Ausbeute von 90% nur noch drei FNs, die sich alle auf den Satz: „The model represents the most important data structures (BibDatases, BibEntries, Events, and related aspects) and has only a little bit of logic attached.“ beziehen. Auffällig für diesen Satz ist der Schreibfehler im Klassennamen „BibDatases“, welches eigentlich die Klasse „BibDatabases“ meint. Dies macht die TLR für dieses Element schwieriger, jedoch sollte ArDoCo es trotzdem möglich sein ein TL zu gewinnen, da die TLR auf Ähnlicheitsvergleichen basiert und nicht auf exakten Namen. Die TLs im Goldstandard für diesen Satz sind die Klassen: BibDatabase, BibDatabases und BibEntry. Diese Textele-

mente werden erkannt und als Recommended Instances abgespeichert, aber mit anderen Modellelementen verknüpft. Die Textelemente BibEntries und BibDatabases werden mit vielen verschiedenen Modellelementen verknüpft, jedoch nicht mit den im Goldstandard definierten Modellelementen. Zum Beispiel wird BibEntries mit dem Modellelement BibEntryTest, oder BibEntryWriterTest verknüpft. Das Textelement BibDatabases wird zum Beispiel mit BibDatabasesFilesTest, BibDatabasesDiffTest, BibDatabaseMode, oder BibDatabaseTest verknüpft. In allen Fällen, werden die Textelemente mit Modellelementen verknüpft, welche nicht nur den Namen des Textelementes beinhalten, sondern diesen weiter ergänzen. Dies führt dazu, dass die Verknüpfung mit den Modellelemente in ihrer reinen Form nicht gefunden werden.

Die als FPs klassifizierten NounMappings in JabRef passen in das selbe Muster wie die in dem Projekt Teammates. Hier werden NounMappings für Textelemente erstellt und mit neuen Wörter erweitert, welche mit dem menschlichen Auge betrachtet nicht ähnlich zueinander sind.

Wie man in Tabelle 6.2 sehen kann, liegen die Gesamtwerte über alle Projekte hinweg, bei einer Präzision von 24% und bei einer Ausbeute von 84%. Dies sind für den ersten Schritt ohne Anpassungen an der TLR von ArDoCo bereits mit Antoniol et al. [2, 1] vergleichbare Ergebnisse. Wie an dem Ausbeutungswert von 84% zu sehen ist, findet ArDoCo bereits viele TLs auch in dem Kontext zwischen den Software-Dokumenten NLSD und Quelltext. Jedoch sind noch zu viele inkorrekte TL vorhanden, wie der Präzisionswert von 24% zeigt, welche anhand des Größenunterschiedes der Eingabe Modelle zu erklären ist. Das eingegebene Codemodell ist mit seinen vielen Klassen, Interfaces und Paketen natürlich größer als das Architekturmodell, welches nur einen Teil der Klassen, Interfaces etc. abbildet. Um die Präzision zu steigern, müssen bestehende Heuristiken angepasst, oder neue Heuristiken eingebaut werden, um die gewonnen TLs stärker zu filtern.

Die letzte Zeile in Tabelle 6.2 zeigt die durchschnittlichen Ergebnisse der Projekten gewichtet nach der Anzahl ihrer TPs und FNs. Dort ist zu sehen, dass die Werte der Präzision und Ausbeute jeweils um ein Prozentpunkt schlechter ausfallen. Dies liegt an dem im Vergleich zu den anderen Projekten schlechterem Ergebnis des Projektes Teammates, welches anhand seiner Größe mehr Gewichtung erhält.

6.4 Evaluation angepasster ArDoCo Heuristiken

Im zweiten Arbeitsschritt werden die verschieden Heuristiken von ArDoCo angeschaut und angepasst. Die durchgeführten Änderungen der bestehenden Heuristiken belaufen sich ausschließlich auf die Erhöhung des Schwellenwertes der Jaro-Winkler-Ähnlichkeit und der Levenshtein-Distanz.

Wie Tabelle 6.4 und Tabelle 4.3 zeigt, wird dadurch bereits größtenteils bessere Ergebnisse erzielt. Die größten Verbesserungen erreichen die Projekte Mediastore und JabRef. In diesen Projekten erhöhte sich die Präzision um 15 in Mediastore und um 14 Prozentpunkte in JabRef. Ihre jeweilige Ausbeute bleiben unverändert. Jedoch zeigen die Ergebnisse auch, dass die Anpassungen nicht in allen Fällen positiver Natur sind. Wie anhand von Tabelle 6.4 zu sehen ist, sind die Anpassungen im Fall von Teammates eine kleine Verschlechterung von einem Prozentpunkt im Bereich der Ausbeute. Dennoch verzeichnet Teammates in der

Präzision einen Anstieg von einem Prozentpunkt. Ebenfalls verbesserte sich die Präzision von Teastore um vier Prozentpunkte und seine Ausbeute um ein Prozentpunkt.

Die Verbesserung der Präzision in jedem Projekt lässt sich einfach erklären. Jedes Projekt enthält viele TLs, bei denen die Namen der Text- und Modellelemente nur ähnlich zueinander sind. Viele von diesen TLs sind jedoch nicht im Goldstandard vorhanden, sondern meistens nur exakt übereinstimmende Namen, wie zum Beispiel die Klasse „OriginCheckFilter“ im Satz „Second, custom filters are applied according to the order specified in web.xml, e.g. OriginCheckFilter.“ vom Projekt Teammates.

Wenn man die Ähnlichkeitsvergleich jedoch strenger macht, was durch die Anhebung des Schwellenwertes von 0.9 auf 0.95 passiert, fallen mehr nur ähnliche Namen bzw. deren TLs raus. Dies bewirkt eine höhere Präzision. Im Projekt Teastore und Teammates gibt es weniger TLs, die auf ähnlichen Namen basieren, weswegen dort die Präzision nur wenig ansteigt.

Im Projekt Mediastore wird jedoch ein Textelement besonders häufig mit einem inkorrekten Modellelement verknüpft. Dort wird das Textelement „file“ mit dem Modellelement namens „filters“ vom Typ Paket verknüpft. Dies führt zu 21 FP TLs. Dies beträgt die Hälfte der reduzierten FP in diesem Schritt, wie es anhand von Tabelle 6.3 und Tabelle 6.5 zu sehen ist. Die Namen der beiden Elemente sind in der Tat ähnlich, jedoch ist dieser TL inkorrekt. Genau diese FP und Weitere fallen durch den strengeren Vergleich raus, was die Präzision um 15 Prozentpunkte erhöht auf einen Wert von 44%, wie es in Tabelle 6.4 zu sehen ist.

Im Projekt JabRef gibt es ebenfalls viele TLs bei welchen die Namen nur ähnlich sind. Dort kann man als Beispiel die TLs nehmen mit dem Textelement „JabRef“, welches inkorrekterweise mit verschiedene Klassen des Modells verknüpft wird. Beispiele dafür sind „JabRefGUI, JabRefCLI, JabRefFrame, oder JabRefAction“. Alle diese Klassen haben ähnliche Namen zu dem Textelement „JabRef“, jedoch ist mit diesem Textelement lediglich das Projekt selbst gemeint. Dies geht aus folgendem Satz hervor: „The model should have no dependencies to other classes of JabRef and the logic should only depend on model classes.“. Durch diese wegfallenden FPs, sowie Weiteren, steigt die Präzision vom Projekt JabRef um 14 Prozentpunkte auf einen Wert von 38%, wie es in Tabelle 6.4 zu sehen ist.

Projekt	**Präzision**	**Ausbeute**	**F1-Maß**
Teastore	0.31	0.98	0.47
Teammates	0.20	0.78	0.32
Mediastore	0.44	0.69	0.54
JabRef	0.38	0.90	0.53
Gesamtübersicht	0.33	0.84	0.47
Gesamtübersicht (gewichtet)	0.28	0.82	0.41

Tabelle 6.4: Ergebnis mit angepassten Heuristiken

Das Problem mit dem Verknüpfen des Projektnames konnte im Fall von JabRef durch die Anpassung des Schwellenwertes nur teilweise behoben werden. Denn das Textelement „JabRef“ wird auch zu seinem gleichnamigen Paket Modellelement verknüpft, welches nicht entfernt wird. In allen Projekten gibt es TLs, die mit dem Textelement ihres jeweiligen Projektnamen verknüpft sind. Einzige Ausnahme ist hierbei das Projekt Mediastore. In seiner NLSD kommt der Projektname nur auseinandergeschrieben vor, wie im Satz: „One of the main components of Media Store is a server-side web front end, namely the Facade component, which delivers websites to the users and provides session management.“. Durch diese Schreibweise konnte ArDoCo korrekterweise keine TLs gewinnen. In den anderen Projekten wird jedoch das Projekt an sich gemeint und nicht potenziell ähnlich heißende Klassen oder Pakete mit exakt demselben Namen. Um genau diese TLs zu entfernen, besitzt ArDoCo bereits einen Informanten, der nach dem Projektnamen in den Instance Links sucht und denen eine Wahrscheinlichkeit von minus unendlich zuordnet. Jedoch existiert noch kein Informant, der genau solche Instance Links entfernt. Deswegen werden diese Instance Links zu TLs. Ein Informant der Instance Links unter einer gewissen Wahrscheinlichkeit entfernt, wird im nächsten Arbeitsschritt implementiert.

Die Verschlechterung der Ausbeute im Projekt Teammates ist schwieriger nachzuvollziehen, da eigentlich nur FPs wegfallen sollten. Indem man den Schwellenwert erhöht, fallen drei TLs raus, jedoch kommt auch ein neuer dazu. Diese wegfallenden TLs sind allesamt „util“ Pakete in den folgenden drei Sätzen: „The Common component contains common utilities used across TEAMMATES.“, „common.util contains utility classes.“, „x.util contains component test cases for testing the utility classes from the Common component.“. In allen drei Sätzen kommt jeweils das exakte Wort „util“ vor. Deswegen sollte man meinen, dass diese TLs durch strengere Ähnlichkeitsvergleichen nicht betroffen sind. Jedoch wird in den TLs nicht das Textelement „util“ verknüpften, sondern das Wort „utility“, welches ebenfalls in allen drei Sätzen vorkommt. Das Wort „utility“ steht jedoch im Bezug zu dem Wort „classes“ und meint somit die Klassen im Paket „util“, welches eigentlich verknüpft werden sollte.

Durch die Erhöhung des Schwellenwertes kommt der TL „servlet“ im Satz 14 „HttpUnit is used to set up a simulated web server in servlet-level tests, where an actual web server is not required.“ hinzu. Dieses Verhalten ist nicht nachzuvollziehen, da in beiden Varianten, mit und ohne Schwellenwerterhöhung ein Instance Link erzeugt wird. Dies ist deshalb der Fall, da das Textelement „servlet“ ebenfalls in Satz 33 vorkommt. Der einzige Unterschied in den Instance Links ist der Name des Textelementes, welcher ohne Anpassung „Servlet“ und mit Anpassung „servlet“ ist. Sie unterscheiden sich nur im Namen, da es im 14. Satz klein und im 33. groß geschrieben ist.

Durch die Anpassungen der Heuristiken wird die Ausbeute im Projekt Teammates um zwei Prozentpunkte schlechter. Diese Verschlechterung ist jedoch nicht repräsentativ, da ein neuer TP TL gewonnen wird und die entfallenen TLs nur inkorrekterweise gefunden werden. Demnach hatte die Anpassung des Schwellenwertes von 0.9 auf 0.95 in der Jaro-Winkler-Ähnlichkeit und der Levenshtein-Distanz nur positive Auswirkungen.

Wie die Tabelle 6.4 zeigt, werden die Gesamtwerte über alle Projekte hinweg, auf eine Präzision von 33% erhöht. Dieser Wert entspricht einer insgesamten Steigerung von neun Prozentpunkten im Bereich der Präzision. Der Ausbeutungswert von 84% bleibt

unverändert. Die Steigerung des F1-Maß beträgt demnach neun Prozentpunkte, von 38% auf 47%.

Im Anbetracht der gewichteten Ergebnisse, fallen die Veränderungen geringer aus, da Teammates nur geringfügig besser wird, mit einem Anstieg von einem Prozentpunkt in Präzision und Ausbeute.

Projekt	True Positives	False Positives	False Negatives	True Negatives
Teastore	61	135	1	10.768
Teammates	114	448	32	163.943
Mediastore	34	43	15	4.459
JabRef	26	43	3	28.749

Tabelle 6.5: Anzahl der klassifizierten Trace Links mit angepassten Heuristiken

6.5 Evaluation mit neuen Heuristiken

In der Evaluation der bisherigen angepassten Heuristiken Abschnitt 6.4 sind Gemeinsamkeiten zwischen den FPs und FNs der verschiedenen Benchmark Projekten aufgefallen. Eine Beispiel für diese Gemeinsamkeiten sind die inkorrekten TLs, die auf das Paket mit dem Projektnamen zeigen. Diese inkorrekten TLs sind in fast jeden Projekt aufgetaucht und werden mithilfe des ThresholdBasedFilterInformanten entfernt, wie es in Unterabschnitt 5.2.8 beschrieben ist. Alle weiteren Informanten sind ebenfalls aufgrund solcher beobachteten Gemeinsamkeiten, oder weiteren allgemeingültigen Inkorrektheiten innerhalb einem oder wenigen Projekten, entstanden.

Die durch die neuen Heuristiken entstandenen Verbesserungen zum vorherigem Arbeitsschritt, sind in Tabelle 6.7 dargestellt. Die Verbesserungen der Informanten alleinstehend betrachtet ist in Tabelle 6.6 zu sehen.

In Tabelle 6.6 ist zu sehen, dass die größte Verbesserung von dem CompounModelElementsNameInformanten hervorgeht. Dieser steigert die Ausbeute, sowie die Präzision des Projektes Mediastore. Die daraus folgende Präzision stieg um zwei Prozentpunkte, die Ausbeute hingegen um 31 Prozentpunkte und erreicht damit einen Wert von 100%, wie es in Tabelle 6.8 zu sehen ist. In der vorhergehenden Evaluation Abschnitt 6.4 ist bereits gemerkt worden, dass alle FNs im Projekt Mediastore sich auf ein Modellelement beziehen, nämlich die Klasse „AudioFile“. Diese TLs werden nicht gefunden, da in keinem Satz das entsprechende Textelement „audio file“ oder „audio files“ gefunden und entsprechend kein NounMapping hinzugefügt wird. Dies behebt der CompoundModelElementsNamesInformant, wie es in Unterabschnitt 5.2.1 erläutert worden ist. Die Präzision steigt ebenfalls, da vor dem neuen Informanten nur teilweise das Textelement „audio file“ erkannt wird. Somit werden TLs vom Textelement „audio“ nicht mehr mit der namensgleichen Klasse „Audio“ verknüpft. Durch das Wegfallen dieser vorher gefunden TLs ergibt sich eine Steigerung von zwei Prozentpunkten in der Präzision.

Informant	Projekt	Metrik	Einfluss
CompoundModelElementsNameInformant	Mediastore	Präzision	+0.02
CompoundModelElementsNameInformant	Mediastore	Ausbeute	+0.31
FilterNounMappingInformant	Teammates	Präzision	+0.02
OneLetterNounMappingRemoverInformant	Teastore	Präzision	+0.17
WordSimilarityCheckerInformant	Teastore	Präzision	+0.17
WordSimilarityCheckerInformant	Mediastore	Präzision	+0.01
WordSimilarityCheckerInformant	JabRef	Präzision	+0.05
PreProjectDirectoryTLRemoverInformant	Teastore	Präzision	+0.02
PreProjectDirectoryTLRemoverInformant	Teammates	Präzision	+0.01
PreProjectDirectoryTLRemoverInformant	Mediastore	Präzision	+0.03
PreProjectDirectoryTLRemoverInformant	JabRef	Präzision	+0.01
PreAndSuffixCheckerInformant	Mediastore	Präzision	+0.24
FalseClassTLRemoverInformant	Teastore	Präzision	+0.04
FalseClassTLRemoverInformant	Teammates	Präzision	+0.02
FalseClassTLRemoverInformant	Mediastore	Präzision	+0.08
FalseClassTLRemoverInformant	JabRef	Präzision	+0.03
CommonComputerScienceWordsFilterInformant	Teammates	Präzision	+0.01
CommonComputerScienceWordsFilterInformant	Mediastore	Präzision	+0.04
CommonComputerScienceWordsFilterInformant	JabRef	Präzision	+0.01
Alle CodeModelAgent Informanten	Teastore	Präzision	+0.20
Alle CodeModelAgent Informanten	Teammates	Präzision	+0.04
Alle CodeModelAgent Informanten	Mediastore	Präzision	+0.45
Alle CodeModelAgent Informanten	JabRef	Präzision	+0.11

Tabelle 6.6: Verbesserungen der einzelnen neue Informanten

Der FilterNounMappingInformant sorgt für passendere NounMappings von Textelementen, sowie es in Unterabschnitt 5.2.2 beschrieben worden ist. Dieser Informant beeinflusst jedoch durch die Schreibweise der NLSD nur das Projekt Teammates. In diesem bewirkt er eine Verbesserung der Präzision um zwei Prozentpunkte.

Der FilterNounMappingInformant ist genauso wie der CompoundModelElementsNameInformant unabhängig von dem Filtern des ThresholdBasedFilterInformant. Das Filtern trifft nur auf die Informanten des CodeModelAgents zu, wie es in Unterabschnitt 5.2.8 beschrieben worden ist. Alle folgenden Informanten können also nicht komplett alleine betrachtet werden, sondern immer im Zusammenspiel mit dem ThresholdBasedFilterInformant. Dieser Informant bringt jedoch alleinstehend keine Verbesserungen, sodass dieser die Ergebnisse der anderen Informanten nicht beeinflusst.

Des weiteren sind die Verbesserungen der folgenden Informanten nicht summierbar. Diese Informanten überschneiden sich in manchen Stellen teilweise oder vollständig. Im Fall von OneLetterNounMappingRemoverInformant, bringt der WordSimilarityCheckerIn-

formant genau dieselbe Funktionalität mit. Diese besondere Überschneidung ist jedoch nicht auf jedes beliebige Projekt übertragbar, sodass der OneLetterNounMappingRemoverInformant seine Daseinsberechtigung behält. Ein Beispiel dafür wäre ein TL, der ein einelementiges Textelement „i“ mit dem Modellelement „ui“ verknüpft. Diesen inkorrekten TL erkennt der WordSimilarityCheckerInformant nicht, da die beiden Namen zu ähnlich sind.

Im Projekt Teastore sind NounMappings aufgefallen, bei denen das Textelement nur aus einem Buchstaben besteht. Diese werden durch den OneLetterNounMappingRemoverInformant entfernt. Dies bringt eine Verbesserung von 17 Prozentpunkten im Projekt Teastore.

Der WordSimilarityCheckerInformant reduziert die Konfidenz von TLs, bei denen die Namen des Textelementes und Modellelementes nicht ähnlich genug zueinander sind, wie in Unterabschnitt 5.2.3 beschrieben. Dabei erweitert er die Verbesserung des OneLetterNounMappingRemoverInformanten, um eine Steigerung der Präzisionswerte von Mediastore um einen und von JabRef um fünf Prozentpunkte.

Der PreProjectDirectoryTLRemoverInformant ist einer von zwei Informanten, welche eine Verbesserung in jedem Projekt bewirkt. Die Funktion von ihm ist in Unterabschnitt 5.2.4 beschrieben. Er steigert die Präzision von Teastore um zwei, Mediastore um drei, sowie Teammates und JabRef um jeweils einen Prozentpunkt.

Der PreAndSuffixCheckerInformant bringt in seiner aktuellen Konfiguration nur eine Verbesserung im Projekt Mediastore. Seine Funktionalität, sowie aktuelle Konfiguration sind in Unterabschnitt 5.2.5 beschrieben. Er steigert die Präzision im Projekt Mediastore um 24 Prozentpunkte.

Der FalseClassTLRemoverInformant ist der zweite Informant, der eine Verbesserung von allen Projekten bewirkt. Seine Funktion ist in Unterabschnitt 5.2.6 beschrieben. Die Verbesserungen belaufen sich auf vier in Teastore, zwei in Teammates, acht in Mediastore und drei Prozentpunkte Steigerung der Präzision in JabRef.

Projekt	Metrik	Einfluss
Teastore	Präzision	+0.20
Teastore	Ausbeute	+0.00
Teammates	Präzision	+0.06
Teammates	Ausbeute	+0.00
Mediastore	Präzision	+0.48
Mediastore	Ausbeute	+0.31
JabRef	Präzision	+0.11
JabRef	Ausbeute	+0.00

Tabelle 6.7: Verbesserungen durch neue Informanten

Der CommonComputerScienceWordsFilterInformant ist nur mit Vorsicht zu betrachten, da er viel Einfluss auf die TLs haben kann, wie in Unterabschnitt 5.2.7 beschrieben. In der

aktuellen Konfiguration sind nur wenige Wörter enthalten, weswegen dieser Informant nur wenig Einfluss auf die Ergebnisse besitzt. Je nach NLSD kann dieser aber beliebig erweitert werden, um möglichst gute Ergebnisse zu erzielen. Die aktuellen Verbesserungen belaufen sich auf eine Steigerung von vier Prozentpunkte der Präzision in Mediastore, sowie jeweils ein Prozentpunkt in Teammates und JabRef.

Wenn man nur die Informanten des CodeModelAgent betrachtet, werden dadurch bereits gute Verbesserungen erzielt. Die im CodeModelAgent enthaltene Informanten sind aus Abbildung 5.2 zu entnehmen. In allen Projekten wird dadurch der Präzisionswert erhöht. Mit einer maximalen Steigerung von 45 Prozentpunkten in Mediastore und einer minimalen Steigerung von vier Prozentpunkten in Teammates.

Projekt	Präzision	Ausbeute	F1-Maß
Teastore	0.51	0.98	0.67
Teammates	0.26	0.78	0.36
Mediastore	0.92	1.00	0.96
JabRef	0.49	0.90	0.63
Gesamtübersicht	0.55	0.92	0.66
Gesamtübersicht (gewichtet)	0.45	0.87	0.57

Tabelle 6.8: Ergebnis mit neuen Heuristiken

Betrachtet man alle neuen Informanten, ergeben sich Verbesserungen, wie sie in Tabelle 6.7 zu sehen sind. Die größten Verbesserungen sind im Projekt Mediastore zu beobachten. Dort wird die Präzision um 48 Prozentpunkt auf 92% und die Ausbeute um 31 Prozentpunkte auf 100% erhöht. Diese Werte zeigen ganz gut, dass die TLR zwischen den beiden Software-Artefakten NLSD und Quelltext bereits funktionieren kann und großes Potenzial besitzt. Jedoch zeigen auch die schlechteren Werte im Projekt Teammates auf, dass es noch mehr Arbeit im Bereich dieser TLR benötigt, damit sie allgemein für alle Projekte mit einer NLSD gut funktioniert. Im Projekt Teammates wird nur eine Präzision von 26% und eine Ausbeute von 78% erreicht, wie es in Tabelle 6.8 zu sehen ist.

Projekt	True Positives	False Positives	False Negatives	True Negatives
Teastore	61	59	1	10.844
Teammates	114	330	33	164.023
Mediastore	49	4	0	4.498
JabRef	26	27	3	28.765

Tabelle 6.9: Anzahl der klassifizierten Trace Links mit neuen Heuristiken

6.6 Evaluation ohne Goldstandard

Um den Prototypen unabhängig von den definierten Goldstandards zu überprüfen, wurde das Projekt BigBlueButton aus der ArDoCo Benchmark [9] in der bisherigen Evaluation nicht genutzt. Diese unabhängige Überprüfung soll einen kleinen Einblick liefern, inwiefern diese Bachelorarbeit auf Projekte außerhalb der Evaluation anwendbar ist. Dies soll Aufschluss über die Allgemeingültigkeit der Arbeit liefern.

Der Prototyp generiert am Projekt BigBlueButton insgesamt 145 TLs. Bei genauerer Analyse sind viele TPs zu sehen, jedoch auch FPs, sowie FNs.

Der Großteil der gefunden TPs beziehen sich auf Pakete. Es werden Pakete wie zum Beispiel „redis“, „pubsub“ im Satz: „Redis PubSub provides a communication channel [...]“ gefunden. Diese TLs zu Paketen gehen auch so weit, dass mehrere gleichnamige Pakete gefunden werden. Ein Beispiel dafür ist das Paket „messages“, das in vier verschiedenen Unterpaketen vorkommt. Hier ist jedoch nicht ganz klar, welches der vier Pakete gemeint ist, oder ob alle relevant sind. Denn im Quelltext von Projekten ist der Code verteilter als in anderen Software-Artefakten, was an dem Vorkommen von gleichnamigen Paketen in verschiedenen Ordner erkennbar ist. Das Projekt JabRef liefert mit seinen sechs „event“ Paketen ein gutes Beispiel dafür. Nur das jeweils in der Dokumentation beabsichtigte Paket sollte durch einen TL verknüpft werden, jedoch kann durch den Kontext auch alle Vorkommen des Paketes gemeint sein. Deswegen ist es wichtig alle Vorkommen des Paketes zu finden, auch wenn nicht immer jedes einen TL bekommen soll.

Zu den TPs gehören auch verknüpfte Klassen, wie zum Beispiel die Klassen „User“ und „Meeting“, welche bereits zuverlässig gefunden werden.

Die Klasse „Meeting“ ist jedoch auch Teil von FPs, wie es anhand von Satz: „The meeting business logic is in the MeetingActor.“ zu erkennen ist. Hier bezieht sich „meeting“ auf seine Logik, welche in „MeetingActor“ vorhanden ist. Ein weiteres Problem besteht darin, dass das Paket „events“, welches in den Paketen „freeswitch/voice/events“ liegt inkorrekterweise verknüpft wird. Bei diesem Paket handelt es sich um Events des „freeswitch“ Objekt und nicht um Events allgemein. Dennoch werden viele TLs zu diesem Paket gefunden, wie zum Beispiel im Satz „Frontends still require MeetingStarted and MeetingEnded events [...]“, wobei die Events sich auf die beiden Klassen „MeetingStarted“ und „MeetingEnded“ beziehen. Hier müssen die Events mehr spezifiziert werden, was durch den Kontext des Satzes erreicht werden kann. Diese beiden Problem, lassen sich durch mehr Einbeziehung des Satz-Kontextes behoben werden.

Eine weitere Fehlerquelle, die zu FPs führt, sind allgemeine Informatik Begriffe, welche jedoch auch als Modellelement existieren. In der NLSD werden in verschiedenen Sätzen von CPU Kernen (engl. *CPU cores*) geredet, wobei „core“ bzw. „cores“ mit dem Paket „bigbluebutton/core“ verknüpft wird. Außerdem existiert im Quelltext ein Paket namens „list“, welches Interfaces von speziellen Listentypen enthält. Im Satz „It provides the list of users [...]“ ist jedoch kein Listentyp gemeint, sondern eine Liste von Objekten in diesem Fall von Usern. Beide Fälle könnten mit neuen Einträgen in der Filterliste des in Unterabschnitt 5.2.7 erläuterten Informanten behoben werden.

Es existieren auch FPs, bei welchen die Fehlerquelle das NounMapping ist. Dies wurde auch an den bisher evaluierten Benchmark Projekten beobachtet und konnte nicht vollständig verhindert werden. Ein Beispiel davon ist das Paket „list“, welches zum Beispiel in

diesem Satz: „Users are able to join the voice conference through the headset.“ mit „conference“ verknüpft wurde. Dieses Verhalten liegt an der Tatsache, dass der InstanceLink auf das Modellelement „list“ zwei NounMappings enthalt. Das Erste enthält das Vorkommen von dem Wort „list“, das zweite jedoch alle Vorkommen des Wortes „conference“.

Schließlich gibt es Fälle, in denen bestimmte Textelemente, wie zum Beispiel „Html5“, nicht erkannt werden. Dies liegt am Namen des im Quelltextes zugehörige Paket, welches den Namen „bigbluebutton-html5“ trägt. Diese beiden Namen sind zu unterschiedlich zueinander, sodass dieser TL nicht gefunden wird. Falls er doch gefunden werden sollte, würde der in Unterabschnitt 5.2.3 beschriebene Informant diesen spätestens ausfiltert. Bei solchen TLs ist der Kontext entscheidend, wie der Satz „The HTML5 server [...] keeping the state of each BigBlueButton client consistent with the BigBlueButton server.“ zeigt. Im aktuellen Zustand wird jedoch der Kontext nicht genug berücksichtigt.

Eine weitere Auffälligkeit ist das Textelement „bbb-html5“, welches exakt dem Paketnamen entspricht, jedoch nicht gefunden wird. Dies liegt daran, dass dieses Paket nicht im Codemodell vorliegt, der Codeextraktor es somit nicht gefunden oder extrahiert hat. Deswegen sollte in weiter aufbauenden Arbeiten sich ebenfalls auf den Codeextraktor fokussiert werden.

Somit lässt sich beobachten, dass im Allgemeinem die TLR zwischen Quelltext und der NLSD bereits funktionsfähig ist und seine Stärke in der hohe Ausbeute liegt. Im Gegensatz dazu ist die Einbeziehung des Kontextes der Sätze in der NLSD die größte Schwachstelle. Dies führt zu der niedrigen Präzision in den meisten Projekten.

6.7 Gefährdung der Validität

Um die Grenzen der durchgeführten Arbeit aufzuzeigen, werden potenzielle Limitationen und Gefährdungen der Validität der vorliegenden Bachelorarbeit in diesem Abschnitt adressiert.

Direkt im ersten Schritt der Evaluation, der Auswahl an Benchmark Projekten, kann man auf Gefahren treffen. Die gewählten Projekte können nicht repräsentativ für die Allgemeinheit von Software-Projekten mitsamt seiner NLSD sein. Um ein großes Spektrum an Software-Projekten abzubilden, sind die vier Projekte aus der ArDoCo Benchmark [9] gewählt worden. Diese stellen eine heterogene Auswahl von Software-Projekten, sodass sie anhand der Arbeit zum etablieren von Benchmark Datensätzen von Fuchß et al. [6] gewählt wurden.

Ein weiterer Punkt ist die Erstellung des Goldstandards, welcher viele Gefahren mit sich bringt. Der Goldstandard wurde dabei für die TLR zwischen der NLSD und dem Quelltext im Rahmen dieser Arbeit eigens erstellt. In der Erstellung des Goldstandards ist auf eine Nutzerstudie aus Zeitgründen verzichtet worden. Die in den Goldstandard aufgenommen TLs könnten somit inkorrekt oder unvollständig sein. Deswegen sind die getroffenen Entscheidungen, wann ein TLs in den Goldstandard aufgenommen wird, ausführlich in Abschnitt 6.1 erläutert.

Eine weitere Gefährdung beim Erstellen des Goldstandards ist ebenfalls eine potenzielle Überanpassung an die von ArDoCo bereits gewonnenen TLs. Um dies möglichst gut zu verhindern, sind nicht alle Projekte der ArDoCo Benchmark [9] evaluiert worden. Es

sind vier Projekten gewählt worden und somit ist das fünfte und letzte Projekt in der eigentlichen Evaluation nicht genutzt worden. Mithilfe diesem fünften Projekt namens „BigBlueButton“, ist die TLR ohne vorher definiertem Goldstandard ausgeführt worden. Die dadurch gewonnen TLs sind dann analysiert und evaluiert worden, wie es der Abschnitt 6.6 beschreibt.

7 Zusammenfassung

Diese Bachelorarbeit stellt einen Ansatz zur Trace Link Recovery (TLR) für natürlichsprachliche Software-Dokumentation und Quelltext dar. Um diesen Ansatz zu realisieren, wird ArDoCo [8] als Grundlage zur TLR genommen und durch das Codemodells von Telege [20] erweitert.

Dabei zeigt die TLR zwischen der natürlichsprachlichen Software Dokumentation (NLSD) und dem Quelltext Potenzial. Die Ergebnisse variieren dabei aber noch zwischen den evaluierten Projekten. Die Tabelle Tabelle 6.8 zeigt dabei die finalen Ergebnisse der Arbeit. Das beste Ergebnis liefert das Projekt Mediastore mit 92% Präzision, 100% Ausbeute und somit einem F1-Maß von 96%. Wohingegen Teammates mit 26% Präzision und 78% Ausbeute nur ein F1-Maß von 36% erreicht.

Der gewonnen Ergebnisse nach zu urteilen, kann die TLR zwischen NLSD und dem Quelltext viel erreichen. Dies kann aber allgemein für alle Software-Projekte nur funktionieren, wenn dieser Ansatz weiterentwickelt wird. Dazu zählt einerseits die Optimierung der bereits implementierten neuen Heuristiken. Andererseits sind diese nicht ausreichend, sodass weitere neue Heuristiken implementiert werden müssen. In dieser Arbeit sind bereits Ideen für solche neuen Heuristiken in Kapitel 5 integriert.

Es gibt jedoch auch Limitationen dieser Arbeit, welche sich auf den Schreibstil der NLSD beziehen. In den geschriebenen Sätzen ist der Kontext von Wörter wichtig, um sie richtig einzuordnen und mit den entsprechenden Modellelementen zu verknüpfen. Der Kontext in dem potenzielle Modellelemente vorkommen, lässt sich mit dem menschlichem Verstand einfach verstehen. Dem entwickelten Prototyp ist es jedoch nicht möglich den Kontext über mehr als zwei Wörter hinweg zu erfassen, sodass in einigen Situationen inkorrekte bzw. keine TLs gewonnen werden. Diese Limitation bezüglich dem Kontext von Wörter erweist sich als größte Herausforderung in diesem Ansatz der TLR und benötigt deswegen in Zukunft besonderer Aufmerksamkeit. Ebenfalls kann das Projekt in der Zukunft noch um das Extrahieren von Unterklassen der Codemodell Elemente erweitert werden. Dies ist noch nicht in der aktuellen Version vorhanden, jedoch ist es nicht selten der Fall, dass Projekte Unterklassen enthalten und diese potenziell in der NLSD referenziert werden.

Zusammenfassend lässt sich feststellen, dass der Ansatz der TLR zwischen der natürlichsprachlichen Software Dokumentation (NLSD) und Quelltext eine effektive Möglichkeit darstellt, die Wartbarkeit von Software-Projekten zu erhöhen. Dieser Ansatz der TLR bietet bereits Potenzial. Dennoch erfordert es weitere Entwicklung, um ihn in der Praxis nutzen zu können.

Literatur

[1] G. Antoniol u. a. „Recovering traceability links between code and documentation“. en. In: *IEEE Transactions on Software Engineering* 28.10 (Okt. 2002), S. 970–983. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1041053. URL: http://ieeexplore.ieee.org/document/1041053/ (besucht am 26. 05. 2023).

[2] Antoniol u. a. „Information retrieval models for recovering traceability links between code and documentation“. In: *Proceedings International Conference on Software Maintenance ICSM-94*. Victoria, BC, Canada: IEEE Comput. Soc. Press, 2000, S. 40–49. ISBN: 978-0-8186-6330-7. DOI: 10.1109/ICSM.2000.883003. URL: http://ieeexplore.ieee.org/document/883003/ (besucht am 25. 09. 2023).

[3] Hugo Bruneliere u. a. „MoDisco: a generic and extensible framework for model driven reverse engineering“. en. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. Antwerp Belgium: ACM, Sep. 2010, S. 173–174. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859032. URL: https://dl.acm.org/doi/10.1145/1858996.1859032 (besucht am 26. 05. 2023).

[4] Sofia Charalampidou u. a. „Empirical studies on software traceability: A mapping study“. en. In: *Journal of Software: Evolution and Process* 33.2 (Feb. 2021). ISSN: 2047-7473, 2047-7481. DOI: 10.1002/smr.2294. URL: https://onlinelibrary.wiley.com/doi/10.1002/smr.2294 (besucht am 26. 05. 2023).

[5] Jane Cleland-Huang, Orlena Gotel und Andrea Zisman, Hrsg. *Software and Systems Traceability*. en. London: Springer London, 2012. ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5. URL: https://link.springer.com/10.1007/978-1-4471-2239-5 (besucht am 26. 05. 2023).

[6] Dominik Fuchß u. a. „Establishing a Benchmark Dataset for Traceability Link Recovery Between Software Architecture Documentation and Models“. en. In: *Software Architecture. ECSA 2022 Tracks and Workshops*. Hrsg. von Thais Batista u. a. Bd. 13928. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2023, S. 455–464. ISBN: 978-3-031-36888-2 978-3-031-36889-9. DOI: 10.1007/978-3-031-36889-9_30. URL: https://link.springer.com/10.1007/978-3-031-36889-9_30 (besucht am 18. 10. 2023).

[7] Suhaimi Ibrahim u. a. „A requirements traceability to support change impact analysis“. en. In: *Asian Journal of Information Technology 4*. Jan. 2005, S. 345–355.

[8] Jan, Keim u. a. *ArDoCo*. Letzter Zugriff 18.10.2023. 2023. URL: https://github.com/ArDoCo.

[9] Jan, Keim u. a. *ArDoCo Benchmark*. Letzter Zugriff 18.10.2023. 2023. URL: https://github.com/ArDoCo/Benchmark.

[10] Jan, Keim u. a. „Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery“. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. L'Aquila, Italy: IEEE, März 2023, S. 141–152. ISBN: 979-8-3503-9750-5. DOI: 10.1109/ICSA56044.2023.00021. URL: https://ieeexplore.ieee.org/document/10092702/ (besucht am 26. 05. 2023).

[11] Jan Keim u. a. „Trace Link Recovery for Software Architecture Documentation“. en. In: *Springer Verlag* Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings. Ed.: S. Biffl (2021). Medium: PDF Publisher: Springer Verlag, S. 101–116. ISSN: 0302-9743. DOI: 10.5445/IR/1000138399. URL: https://publikationen.bibliothek.kit.edu/1000138399 (besucht am 26. 05. 2023).

[12] Levenshtein und Vladimir I. „Binary codes capable of correcting deletions, insertions, and reversals“. In: *Soviet physics doklady*. Bd. 8. 10. Soviet Union, 1966, S. 707–710.

[13] A. Marcus und J.I. Maletic. „Recovering documentation-to-source-code traceability links using latent semantic indexing“. In: *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, S. 125–135. ISBN: 978-0-7695-1877-0. DOI: 10.1109/ICSE.2003.1201194. URL: http://ieeexplore.ieee.org/document/1201194/ (besucht am 25. 09. 2023).

[14] Shouichi Nagano, Yusuke Ichikawa und Toru Kobayashi. „Recovering Traceability Links between Code and Documentation for Enterprise Project Artifacts“. In: *2012 IEEE 36th Annual Computer Software and Applications Conference*. Izmir, Turkey: IEEE, Juli 2012, S. 11–18. ISBN: 978-0-7695-4736-7. DOI: 10.1109/COMPSAC.2012.10. URL: http://ieeexplore.ieee.org/document/6340249/ (besucht am 26. 05. 2023).

[15] B. Nuseibeh, S. Easterbrook und A. Russo. „Leveraging inconsistency in software development“. In: *Computer* 33.4 (Apr. 2000), S. 24–29. ISSN: 00189162. DOI: 10.1109/2.839317. URL: http://ieeexplore.ieee.org/document/839317/ (besucht am 26. 05. 2023).

[16] OMG Group. *Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM)*. Sep. 2016.

[17] Reza Meimandi Parizi, Sai Peck Lee und Mohammad Dabbagh. „Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts“. In: *IEEE Transactions on Reliability* 63.4 (Dez. 2014), S. 913–926. ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2014.2338254. URL: http://ieeexplore.ieee.org/document/6862933/ (besucht am 30. 05. 2023).

[18] Sophie Schulz. *Linking Software Architecture Documentation and Models*. en. Medium: PDF Publisher: Karlsruher Institut für Technologie (KIT). 2020. DOI: 10.5445/IR/1000126194. URL: https://publikationen.bibliothek.kit.edu/1000126194 (besucht am 26. 05. 2023).

[19] R. Settimi u. a. „Supporting software evolution through dynamically retrieving traces to UML artifacts“. In: *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* Kyoto, Japan: IEEE, 2004, S. 49–54. ISBN: 978-0-7695-2211-1. DOI: 10.1109/IWPSE.2004.1334768. URL: http://ieeexplore.ieee.org/document/1334768/ (besucht am 25. 09. 2023).

[20] Tobias Telge. *Automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext.* de. Publisher: Karlsruher Institut für Technologie (KIT). 2023. DOI: 10.5445/IR/1000157372. URL: https://publikationen.bibliothek.kit.edu/1000157372 (besucht am 26. 05. 2023).

[21] William E Winkler. *String comparator metrics and enhanced decision rules in the Fellegi-Sunter model of record linkage.* en. 1990.

[22] Xiaofan Chen und John Grundy. „Improving automated documentation to code traceability by combining retrieval techniques“. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* Lawrence, KS, USA: IEEE, Nov. 2011, S. 223–232. ISBN: 978-1-4577-1639-3. DOI: 10.1109/ASE.2011.6100057. URL: http://ieeexplore.ieee.org/document/6100057/ (besucht am 26. 05. 2023).

[23] Yuchen Zhang, Chengcheng Wan und Bo Jin. „An empirical study on recovering requirement-to-code links“. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* Shanghai: IEEE, Mai 2016, S. 121–126. ISBN: 978-1-5090-2239-7. DOI: 10.1109/SNPD.2016.7515889. URL: https://ieeexplore.ieee.org/document/7515889/ (besucht am 30. 05. 2023).

[24] Xuchang Zou, Raffaella Settimi und Jane Cleland-Huang. „Improving automated requirements trace retrieval: a study of term-based enhancement methods“. en. In: *Empirical Software Engineering* 15.2 (Apr. 2010), S. 119–146. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-009-9114-z. URL: http://link.springer.com/10.1007/s10664-009-9114-z (besucht am 25. 09. 2023).