Karlsruher Institut für Technologie

Automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext

Masterarbeit von

Tobias Telge

an der Fakultät für Informatik
Institut für Informationssicherheit und Verlässlichkeit (KASTEL)

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

17. September 2022 – 17. März 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Zusammenfassung

Nachverfolgbarkeitsverbindungen zwischen Architektur und Quelltext können das Wissen über ein System erweitern. Aufgrund des Erstellungsaufwands existieren in Softwareprojekten oft keine oder nur unvollständige Nachverfolgbarkeitsinformationen. Diese Arbeit untersucht einen Ansatz mit zwei Schritten, um automatisiert Nachverfolgbarkeitsverbindungen zwischen Architekturmodellelementen und Quelltext zu generieren. Damit die Erstellung von Nachverfolgbarkeitsverbindungen für verschiedene Programmiersprachen und Architektur-Metamodelle vereinheitlicht wird, werden im ersten Schritt aus den vorliegenden Artefakten Modelle erstellt. Der Quelltext wird dabei in ein von der konkreten Programmiersprache unabhängiges Modell überführt. Dafür wird ein Metamodell verwendet, das auf dem von der OMG spezifizierten KDM basiert. Für den zweiten Schritt werden auf den erstellten Modellen arbeitende Heuristiken und Aggregationen definiert. Diese werden genutzt, um die Nachverfolgbarkeitsverbindungen zu generieren. Die Heuristiken nutzen zum Beispiel Paket-, Pfad-, Namen- und Methoden-Informationen. Die Evaluation des Ansatzes nutzt einen dafür erstellten Goldstandard mit fünf Fallstudien. Es werden Nachverfolgbarkeitsverbindungen für PCM, UML, Java und Shell generiert. Für den Mikro-Durchschnitt des F_1-Maßes wird ein Wert von 99,11 % erreicht. Fließt jede Komponente und Schnittstelle in gleichem Maße in den Wert ein, beträgt das F_1-Maß 93,71 %. Insgesamt können mit dem Ansatz dieser Arbeit also sehr gute Ergebnisse erzielt werden. Für die *TEAMMATES*-Fallstudie wird mithilfe mehrerer Quelltextversionen der Einfluss der Konsistenz auf die Ergebnisse untersucht. Der Mikro-Durchschnitt des F_1-Maßes ist für die konsistentere Version um 6,05 Prozentpunkte höher. Die Konsistenz kann also die Qualität der Ergebnisse beeinflussen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Bei der Entwicklung eines Softwaresystems fallen viele verschiedene Artefakte an. Dazu gehören zum Beispiel Anforderungsdokumente, Softwarearchitekturmodelle oder Quelltext. Diese Artefakte stehen miteinander in Beziehung. So implementieren zum Beispiel bestimmte Quelltextabschnitte bestimmte Anforderungen oder, wie in Abbildung 1.1 dargestellt, bestimmte Elemente eines Architekturmodells. Um diese Beziehungen explizit zu machen, können Nachverfolgbarkeitsverbindungen (*engl.* Trace Links) zwischen den Artefakten erstellt werden. Die Verbindungen können das Wissen über ein System erweitern und in der Softwareentwicklung helfen, zum Beispiel bei der Auswirkungsanalyse [16].

Nachverfolgbarkeitsverbindungen können direkt gemeinsam mit den Artefakten erstellt werden. Aufgrund des damit verbundenen Aufwands existieren in Softwareprojekten jedoch oft keine oder nur unvollständige Nachverfolgbarkeitsinformationen [11]. Dann müssen die Verbindungen nachträglich aus den Artefakten gewonnen werden. Das kann manuell erfolgen, was jedoch aufwendig, fehlerbehaftet und teuer ist. Daher bietet es sich an, den Vorgang zu automatisieren.

Die automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen wird in der Literatur vor allem im Hinblick auf die Verbindung von Anforderungen mit Quelltext untersucht [8]. Doch auch die Verbindung von Softwarearchitekturmodellen und Quelltext ist von großer Bedeutung.

Eine Softwarearchitektur ist das Ergebnis von Entwurfsentscheidungen und beschreibt die grundlegenden Elemente und Beziehungen eines Systems [23, S. 37]. Sie wird bei der Entwicklung von Softwaresystemen häufig erstellt, da sie verschiedene Vorteile bietet. Dazu gehört eine erleichterte Kommunikation, da sich Entwickler und andere Personen bei der Diskussion auf die Architektur und ihre Konzepte beziehen können [4]. Zudem kann das System anhand der Architektur analysiert werden, um zu überprüfen, ob es bestimmte Qualitätsanforderungen erfüllt [4]. Durch eine Aufteilung des Systems in Komponenten können diese in verschiedenen Systemen wiederverwendet werden [4]. Auch bei der Projektplanung wie der Kostenschätzung kann die Architektur helfen [4].

Mithilfe von Nachverfolgbarkeitsverbindungen zwischen Architektur und Quelltext lässt sich zum Beispiel die Auswirkung einer Architekturänderung analysieren. Eine andere Nutzungsmöglichkeit ist die Prüfung, welche Architekturelemente noch nicht implementiert wurden. Auch zur Prüfung der Konsistenz zwischen Architektur und Quelltext sind Nachverfolgbarkeitsverbindungen hilfreich. Daher ist das Ziel dieser Arbeit die automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext.

In der bereits erwähnten Abbildung 1.1 ist ein Beispiel für Nachverfolgbarkeitsverbindungen zwischen Architektur und Quelltext dargestellt. Die Komponente *MediaManagement* wird hier durch die Klasse *MediaManagerImpl* realisiert. Der Name wurde bei der Implementierung also leicht verändert. Außerdem wird die Architekturschnittstelle

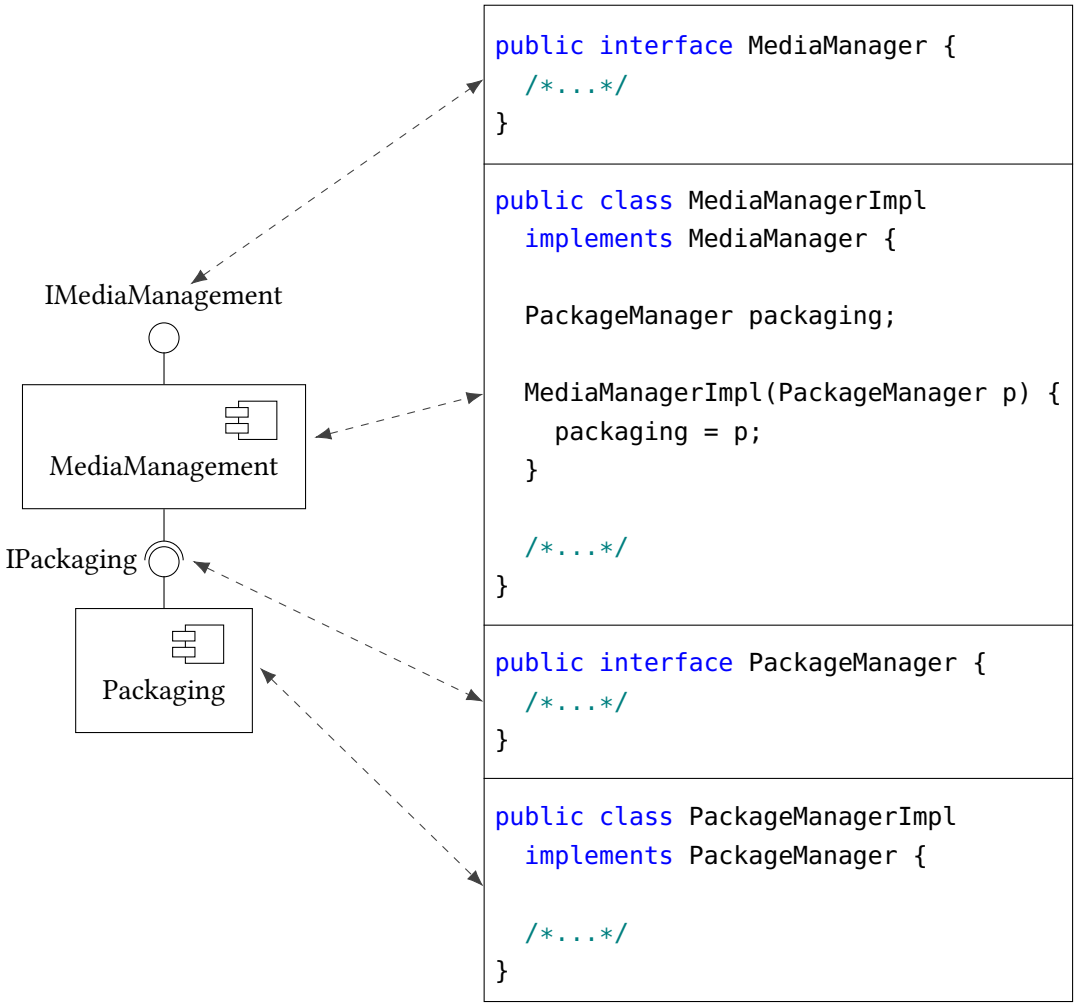

Abbildung 1.1.: Beispiel für Nachverfolgbarkeitsverbindungen zwischen Architektur und Quelltext

IMediaManagement im Quelltext durch die Schnittstelle *MediaManager* umgesetzt. In der Architektur bietet *MediaManagement* die Schnittstelle an, während analog im Quelltext *MediaManagerImpl* die Schnittstelle implementiert. Andere Nachverfolgbarkeitsverbindungen existieren zwischen den Schnittstellen *IPackaging* und *PackageManager* sowie zwischen der Komponente *Packaging* und der Klasse *PackageManagerImpl*.

Der Ansatz dieser Arbeit soll nicht auf konkrete Programmiersprachen oder Architektur-Metamodelle festgelegt sein. Es soll also unter anderem möglich sein, für den Quelltext verschiedene Programmiersprachen zu unterstützen. Im ersten Schritt des Ansatzes werden daher der vorhandene Quelltext und das vorhandene Architekturmodell in neue Modelle überführt. Diese Modelle sollen unabhängig von einer konkreten Programmiersprache oder einem konkreten Architektur-Metamodell sein. Damit soll die Erstellung von Nachverfolgbarkeitsverbindungen für verschiedene Programmiersprachen und Architektur-Metamodelle vereinheitlicht werden.

Im zweiten Schritt des Ansatzes werden Nachverfolgbarkeitsverbindungen zwischen den Elementen der Modelle aus dem ersten Schritt erstellt. Dafür werden verschiedene Heuristiken eingesetzt, die Kandidaten für mögliche Nachverfolgbarkeitsverbindungen bewerten. Dabei nutzen Heuristiken Informationen wie zum Beispiel ähnliche Namen oder Strukturen als Hinweise. Die Ergebnisse der verschiedenen Heuristiken werden aggregiert, um die Nachverfolgbarkeitsverbindungen zu erhalten.

Der eigene Beitrag dieser Arbeit beinhaltet die Definition von Modellen, die die Gewinnung von Nachverfolgbarkeitsverbindungen im nachfolgenden Schritt vereinheitlichen sollen. Für den zweiten Schritt werden verschiedene Heuristiken definiert, die Kandidaten für Nachverfolgbarkeitsverbindungen bewerten. Die Ergebnisse der Heuristiken werden geeignet aggregiert, um die Nachverfolgbarkeitsverbindungen zu erstellen. Die definierten Heuristiken und die durch ihre Aggregation entstandenen Nachverfolgbarkeitsverbindungen werden mithilfe eines dafür angefertigten Goldstandards evaluiert. Es wird also für einige Fallstudien unter anderem geprüft, wie gut die einzelnen Heuristiken funktionieren und wie sehr die am Ende gewonnenen Nachverfolgbarkeitsverbindungen mit dem Goldstandard übereinstimmen.

Diese Arbeit führt nun in Kapitel 2 für das Verständnis der Arbeit hilfreiche Grundlagen ein. Dann werden in Kapitel 3 verwandte Arbeiten vorgestellt. In Kapitel 4 wird der Ansatz dieser Arbeit näher erläutert. Anschließend wird in Kapitel 5 die gewählte Architektur zur Umsetzung des Ansatzes vorgestellt. Hier werden auch die erstellten Metamodelle beschrieben. In Kapitel 6 wird auf die Implementierung eingegangen. Hier findet sich unter anderem eine Beschreibung der definierten Heuristiken und ihrer Aggregation. Die Erläuterung der Evaluation folgt in Kapitel 7. Dabei wird unter anderem auf die Erstellung des Goldstandards, die gewählten Metriken und die Ergebnisse eingegangen. Zum Schluss erfolgt in Kapitel 8 eine Zusammenfassung und ein Ausblick.

2. Grundlagen

Dieses Kapitel erläutert grundlegende Informationen, die beim Verständnis dieser Arbeit helfen. Zuerst wird in Abschnitt 2.1 allgemein auf Nachverfolgbarkeitsverbindungen eingegangen. In Abschnitt 2.2 wird das Palladio-Komponentenmodell (PCM) eingeführt. Zum Schluss werden in Abschnitt 2.3 mögliche Metriken zur Evaluation von Nachverfolgbarkeitsverbindungen beschrieben.

2.1. Nachverfolgbarkeitsverbindungen

Nachverfolgbarkeit kann allgemein definiert werden als das Maß, zu dem eine Beziehung zwischen Produkten des Entwicklungsprozess festgestellt werden kann [17, S. 481]. Eine Nachverfolgbarkeitsverbindung ist eine einzelne spezifizierte Assoziation zwischen einem Quell- und einem Zielartefakt [10, S. 5]. Die im Folgenden beschriebenen Grundlagen von Nachverfolgbarkeitsverbindungen basieren auf dem Buch von Cleland-Huang et al. [10].

Die Granularität von Nachverfolgbarkeitsverbindungen kann sich unterscheiden. Sie hängt von der Größe der betrachteten Artefakte ab. Die Verbindungen können manuell, automatisiert oder halbautomatisiert, also durch Kombination von automatisierten und manuellen Tätigkeiten, erstellt werden [10, S. 10]. In dieser Arbeit wird die automatisierte Erstellung von Nachverfolgbarkeitsverbindungen betrachtet.

Nachverfolgbarkeitsverbindungen können direkt zusammen mit den Artefakten erstellt werden (*engl.* trace link capture). Sie können aber auch, wie in dieser Arbeit, nachträglich gewonnen werden (*engl.* trace link recovery).

Es kann zwischen Vorwärts- und Rückwärtsnachverfolgung (*engl.* forward and backward tracing) unterschieden werden [10, S. 18]. Bei der Vorwärtsnachverfolgung entspricht die Richtung der Reihenfolge im Entwicklungsprozess, also zum Beispiel von der Architektur zum Quelltext.

Zudem kann zwischen vertikaler und horizontaler Nachverfolgung (*engl.* vertical and horizontal tracing) unterschieden werden [10, S. 19]. Bei der vertikalen Nachverfolgung werden im Gegensatz zur horizontalen Nachverfolgung Artefakte mit unterschiedlichem Abstraktionsniveau betrachtet. Das ist in dieser Arbeit mit dem Architekturmodell und dem Quelltext als betrachtete Artefakte der Fall.

2.2. Palladio-Komponentenmodell

Im Allgemeinen ist eine Softwarearchitektur das Ergebnis von Entwurfsentscheidungen und beschreibt die grundlegenden Elemente und Beziehungen eines Systems [23, S. 37]. Softwarearchitekturen können mit Modellen beschrieben werden. Dafür gibt es verschiedene Metamodelle, deren Instanzen dann Architekturmodelle sind. Ein Metamodell ist

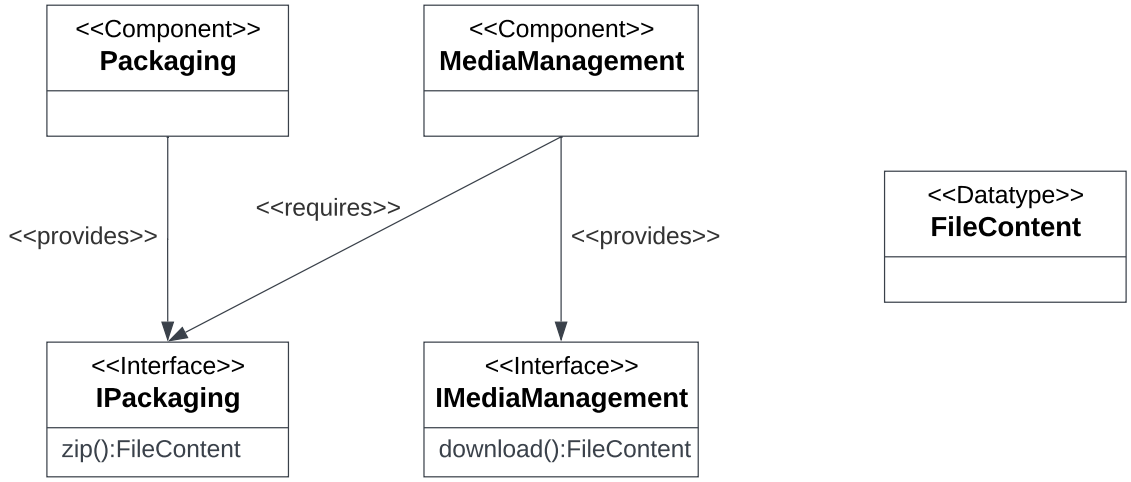

Abbildung 2.1.: Eine Visualisierung eines beispielhaften Komponenten-Repository in Palladio

ein Modell, das die mögliche Struktur von Modellen beschreibt und somit Modelle repräsentiert [26, S. 108]. Im Metamodell UML können zum Beispiel Komponentendiagramme Softwarearchitekturen beschreiben [24, S. 31]. Ein anderes Architektur-Metamodell ist das Palladio-Komponentenmodell. Es wurde entwickelt, um Leistungsvorhersagen für Softwarearchitekturen zu ermöglichen. Die in diesem Abschnitt beschriebenen Grundlagen von Palladio basieren auf dem Buch von Reussner et al. [23].

Das Komponenten-Repository in Palladio enthält verschiedene Komponenten, Schnittstellen und Datentypen. Die Datentypen werden zum Beispiel als Typen für Parameter in Schnittstellen verwendet. Schnittstellen definieren über ihre Signaturen Services und können von Komponenten angeboten oder benötigt werden. Dabei werden Schnittstellen unabhängig von ihrer Verwendung durch bestimmte Komponenten definiert. Komponenten sind Software-Bausteine, deren Inneres nicht verstanden werden muss, um sie zusammensetzen oder auf Rechnern verteilen zu können [23, S. 47]. Abbildung 2.1 zeigt beispielhaft eine Visualisierung eines Komponenten-Repository in Palladio. Es enthält die Komponenten *Packaging* und *MediaManagement*, die Schnittstellen *IPackaging* und *IMediaManagement* sowie den Datentyp *FileContent*. Die Komponente *Packaging* bietet die Schnittstelle *IPackaging* an. Die Komponente *MediaManagement* bietet die Schnittstelle *IMediaManagement* an und benötigt die Schnittstelle *IPackaging*. Der Datentyp *FileContent* wird in den Signaturen der Schnittstellen als Rückgabetyp verwendet.

In Palladio gibt es sowohl *Basic Components* als auch *Composite Components*. Eine *Composite Component* ist aus anderen Komponenten zusammengesetzt. Diese Teile können neben *Basic Components* auch wieder *Composite Components* sein. Für die Zusammensetzung von Komponenten in *Composite Components* werden in Palladio sogenannte *assembly contexts* genutzt. Ein *assembly context* bezieht sich immer auf eine bestimmte Komponente. Die Zusammensetzung wird auch als vertikale Komposition bezeichnet. Für die horizontale Komposition werden ebenfalls *assembly contexts* genutzt. Sie beschreibt, wie

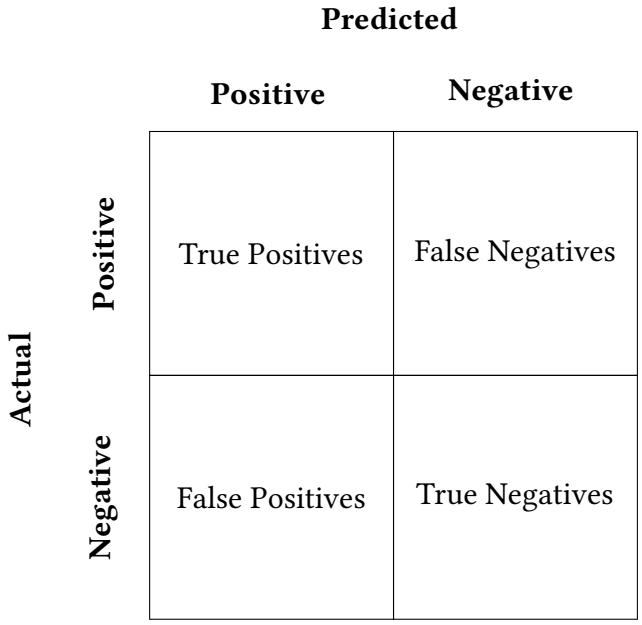

Abbildung 2.2.: Konfusionsmatrix

die Komponenten verdrahtet sind, also über ihre angebotenen und benötigten Schnittstellen miteinander verbunden sind. Daneben gibt es in Palladio noch *allocation contexts* und *usage contexts*. *Allocation contexts* beschreiben die Zuordnung der Komponenten zu Hardware- und Software-Ressourcen. *Usage contexts* beschreiben die Nutzung an den Grenzen des Systems, wie zum Beispiel die Anzahl der Benutzer. Eine Komponente kann in mehreren *assembly contexts* und *allocation contexts* auftreten.

2.3. Metriken

Um Nachverfolgbarkeitsverbindungen zu evaluieren, können verschiedene Metriken eingesetzt werden. Im Folgenden werden solche Metriken basierend auf der Arbeit von Hayes et al. [14] eingeführt.

Allgemein ordnet ein Klassifikator zu klassifizierende Objekte zuvor definierten Klassen zu. Die Ergebnisse einer binären Klassifikation können wie in Abbildung 2.2 als Konfusionsmatrix dargestellt werden. Es wird zwischen der vom Klassifikator vorhergesagten Klasse und der tatsächlichen Klasse unterschieden. Ist die tatsächliche Klasse „Positiv“ und die vorhergesagte Klasse ebenfalls „Positiv“, so zählt dieses Ergebnis zu den *True Positives*. Wird analog die tatsächliche Klasse „Negativ“ korrekt vorhergesagt, gehört das Ergebnis zur Menge der *True Negatives*. Die *False Positives* umfassen die Fälle, in denen der Klassifikator fälschlicherweise die Klasse „Positiv“ vorhersagt. Entsprechend umfasst die Menge der *False Negatives* die Fälle, in denen fälschlicherweise die Klasse „Negativ“ vorhersagt wird.

Mit der Zahl TP wird im Folgenden die Anzahl der *True Positives* bezeichnet. Analog ist TN die Anzahl der *True Negatives*, FP die Anzahl der *False Positives* und FN die Anzahl der

False Negatives. Mit TP, FP, TN und FN lassen sich nun verschiedene Metriken berechnen, die dabei helfen sollen, die Qualität einer Klassifikation zu messen.

Eine solche Metrik ist die Präzision. Diese misst den Anteil der TP an allen Fällen, in denen der Klassifikator die Klasse „Positiv“ vorhersagt. Die Metrik gibt somit an, mit welchem Anteil eine positive Vorhersage tatsächlich positiv ist.

$$\text{Präzision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Die Ausbeute (*engl.* Recall) misst den Anteil der TP an allen Fällen, in denen die tatsächliche Klasse „Positiv“ ist. Die Metrik gibt somit an, mit welchem Anteil tatsächlich positive Fälle korrekt vorhergesagt wurden.

$$\text{Ausbeute} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Die Präzision oder die Ausbeute allein ist unter Umständen nicht besonders aussagekräftig. Zum Beispiel kann eine Ausbeute von 100 % leicht erreicht werden, indem der Klassifikator immer die Klasse „Positiv“ vorhersagt. Um die Präzision und die Ausbeute zu kombinieren, gibt es daher unter anderem das F_1-Maß. Dieses bildet das harmonische Mittel zwischen Präzision und Ausbeute.

$$F_1\text{-Maß} = 2 \times \frac{\text{Präzision} \times \text{Ausbeute}}{\text{Präzision} + \text{Ausbeute}}$$

Eine weitere Metrik ist die Genauigkeit. Diese Metrik hängt im Gegensatz zu den anderen erwähnten Metriken auch von der Zahl der *True Negatives* ab. Sie misst den Anteil der Fälle, in denen der Klassifikator die korrekte Klasse vorhersagt, an allen Vorhersagen. Sie ist besonders bei einem ausgewogenen Datensatz, in dem die verschiedenen Klassen ähnlich oft vorkommen, aussagekräftig.

$$\text{Genauigkeit} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

Bei einer Klassifikation in mehr als zwei Klassen können die Zahlen TP, FP, TN und FN für jede Klasse einzeln berechnet werden. Dabei werden bei Betrachtung einer bestimmten Klasse jeweils alle anderen Klassen als „Negativ“ gezählt. Für die Evaluation wird das Ergebnis also für jede der Klassen als binäre Klassifikation betrachtet. Nun können die obigen Metriken für jede Klasse einzeln berechnet werden. Um das Gesamtergebnis über alle Klassen zu bewerten, kann der Durchschnitt dieser klassenspezifischen Metriken gebildet werden. Dies wird als Makro-Durchschnitt bezeichnet. Dabei fließen die Ergebnisse der einzelnen Klassen somit zum gleichen Maße in das Endergebnis ein. Das gilt auch, wenn der Datensatz unausgewogen ist, die verschiedenen Klassen also unterschiedlich oft darin vorkommen. Besonders gute oder schlechte Ergebnisse in einer selten vorkommenden Klasse können das Endergebnis somit stark beeinflussen. Um die Ausgewogenheit des Datensatzes zu berücksichtigen, können der gewichtete Makro-Durchschnitt oder der

Mikro-Durchschnitt gebildet werden. Beim gewichteten Makro-Durchschnitt wird die für eine Klasse berechnete Metrik mit der Anzahl der Vorkommen dieser Klasse im Datensatz gewichtet. Beim Mikro-Durchschnitt fließt jede einzelne Vorhersage des Klassifikators in gleichem Maße ins Endergebnis ein. Es werden also TP, FP, TN und FN jeweils über alle Klassen aufsummiert und anhand der Ergebnisse der Summen die obigen Metriken berechnet.

3. Verwandte Arbeiten

Es gibt verschiedene Arten von verwandten Arbeiten. So gibt es Arbeiten, die sich mit der Verarbeitung von Quelltext und der Extraktion von Modellen daraus beschäftigen. In Abschnitt 3.1 werden solche Arbeiten vorgestellt. Andere Arbeiten befassen sich mit der Konsistenzhaltung von Modellen und der Gewinnung von Nachverfolgbarkeitsverbindungen. Solche Arbeiten finden sich in Abschnitt 3.2.

3.1. Modellextraktion aus Quelltext

Der „Java Model Parser and Printer“ (JaMoPP) von Heidenreich et al. [15] soll die Lücke zwischen Programmier- und Modellierungssprachen schließen. Dafür definiert JaMoPP ein komplettes Metamodell für Java, das auch die statische Semantik erfasst. Es enthält alle Java-Elemente wie Pakete, Klassen, Importe und Variablen bis Java 5. Zudem kann JaMoPP Instanzen dieses Metamodells in Java-Quelltext überführen sowie aus Java-Quelltext eine Instanz des Metamodells erzeugen. Für letzteres erstellt ein Parser zuerst einen abstrakten Syntaxbaum (AST). Dieser enthält noch unaufgelöste Referenzen, die im nächsten Schritt aufgelöst werden. So werden symbolische Namen durch explizite Verbindungen zu den entsprechenden Elementen ersetzt.

MoDisco von Bruneliere et al. [7] ist ein weiteres Projekt, das die Erzeugung von Modellen aus Quelltext beinhaltet. MoDisco bietet ein generisches und erweiterbares Rahmenwerk für modellgetriebenes Reverse-Engineering. Beim modellgetriebenen Reverse-Engineering wird in einer einem ersten Schritt ein Modell aus Quelltext, Dokumentation oder anderen Artefakten gewonnen, das eine Sicht auf das System bietet. Im zweiten Schritt werden Modelle für ein spezifisches Ziel genutzt, zum Beispiel das Refactoring. Für Java bietet MoDisco ein vollständiges Java-Metamodell und eine entsprechende Software-Komponente, die Instanzen dieses Metamodells aus Java-Quelltext erzeugen kann. Zudem stellt MoDisco Implementierungen von Standard-Metamodellen der Object Management Group (OMG) bereit. Dazu gehört zum Beispiel das Knowledge Discovery Metamodel (KDM), das unabhängig von konkreten Programmiersprachen ist. Für Java bietet MoDisco auch eine entsprechende Transformation vom Java-Metamodell zum KDM.

Auch für andere Programmiersprachen gibt es Arbeiten, die sich mit der Erzeugung von KDM-Modellen befassen. Barbier et al. [3] transformieren Cobol-Quelltext in KDM-Modelle. Aus PHP-Quelltext erzeugen Trias et al. [28] KDM-Modelle. Wulf et al. [30] transformieren C#-Quelltext in KDM-Modelle.

Becker et al. [5] stellen vor, wie sie im Rahmen des Q-ImPrESS-Projektes aus Quelltext ein Komponentenmodell extrahieren, das dann für Qualitätsvorhersagen genutzt werden kann. Im vorgestellten Reverse-Engineering-Prozess wird zuerst das Werkzeug SISSy verwendet, um aus dem Quelltext einen generalisierten abstrakten Syntaxbaum,

das GAST-Modell, zu extrahieren. SISSy unterstützt dabei die Programmiersprachen Java, C, C++ und Delphi und kann zur statischen Quelltextanalyse verwendet werden. Aus dem GAST-Modell werden dann mithilfe von SoMoX die Komponenten wiedergewonnen. Dazu werden verschiedene Quelltextmetriken evaluiert und kombiniert. Zum Beispiel wird die Abstraktheit, also die Anzahl der Schnittstellen, abstrakten Klassen und anderen abstrakten Entitäten, als Metrik verwendet. Weitere Metriken sind zum Beispiel die Kopplung oder Schnittstellenverletzungen. Die kombinierten Metriken werden in einem Clustering-Algorithmus verwendet, um die Klassen des GAST-Modells zu aggregieren. Mehrere Iterationen führen so zu immer abstrakteren Komponenten. Zusammen mit den extrahierten Schnittstellen ergibt sich so ein Architekturmodell. Für die Qualitätsanalysen wird dieses Modell mit weiteren Informationen angereichert und das Verhalten der Komponenten erfasst und abstrahiert.

3.2. Konsistenzhaltung und Nachverfolgbarkeitsverbindungen

Der Vitruvius-Ansatz von Klare et al. [21] befasst sich mit der Konsistenzhaltung verschiedener Modelle bei der sichtenbasierten Systementwicklung. Dabei werden verschiedene Modelle, die zusammen ein virtuelles Single-Underlying-Model (V-SUM) bilden, mithilfe von Modelltransformationen konsistent gehalten. Das V-SUM repräsentiert dabei das System. Über verschiedene Sichten, die Informationen des V-SUMs darstellen, kann das System betrachtet und angepasst werden. Für die Konsistenzhaltung führen Klare et al. die Reactions-Sprache und die Mappings-Sprache ein. Mit der Reactions-Sprache kann imperativ festgelegt werden, wie auf Änderungen an einem Modell reagiert werden muss, um die Konsistenz zu einem Modell, das eine Instanz eines anderen Metamodells ist, wiederherzustellen. Es können also unidirektionale Transformationen abgeleitet werden. Mit der Mappings-Sprache kann deklarativ festgelegt werden, wie zwei Metamodelle miteinander in Beziehung stehen. Daraus können dann bidirektionale Transformationen abgeleitet werden. Beim Vitruvius-Ansatz werden also Modelle betrachtet, die bestimmte Konsistenzeigenschaften erfüllen. Beim Ansatz dieser Masterarbeit werden hingegen existierender Quelltext und existierende Architekturmodelle betrachtet, die keine besonderen Konsistenzeigenschaften erfüllen müssen.

Der Commonalities-Ansatz von Klare und Gleitze [20] beschreibt, wie mehrere Modelle konsistent gehalten werden können. Dafür macht der Ansatz die gemeinsamen Konzepte, die in den verschiedenen Modellen repräsentiert werden, explizit. Die Konzepte werden in einem neuen Konzept-Metamodell erfasst und die Beziehungen zwischen diesem und den existierenden Metamodellen spezifiziert. Es können mehrere Konzept-Metamodelle hierarchisch zusammengesetzt werden, um mehr Modelle konsistent zu halten. Für die Beschreibung des Konzept-Metamodells und der Beziehungen zu den existierenden Metamodellen führen Klare und Gleitze die Commonalities-Sprache ein. Durch das Explizitmachen der gemeinsamen Konzepte sollen die Verbindungen zwischen den Metamodellen leichter verständlich werden. Um die verschiedenen Modelle dann konsistent zu halten, können Transformationen abgeleitet werden.

Grammel et al. [13] stellen ihren Ansatz vor, Nachverfolgbarkeitsverbindungen zwischen beliebigen Quell- und Zielmodellen zu erzeugen. Im ersten Schritt des Ansatzes werden Quell- und Zielmodell importiert und in ein gemeinsames Datenmodell überführt. Dann werden verschiedene Algorithmen angewandt, um die Modellelemente in Quell- und Zielmodell zu identifizieren, die dasselbe Konzept repräsentieren. Die Algorithmen geben für jede Kombination von Elementen des Quell- und Zielmodells einen Ähnlichkeitswert an. Um Algorithmen, die Ähnlichkeiten zwischen Graphen finden, nutzen zu können, muss das gemeinsame Datenmodell eine Graphstruktur aufweisen. Als Ähnlichkeitsmaße, auf denen die Berechnungen der Algorithmen basieren, werden Attributsähnlichkeitsmaße, Verbindungsähnlichkeitsmaße und Ähnlichkeitsmaße, die die Metamodelle berücksichtigen, verwendet. Attributsähnlichkeitsmaße messen die Ähnlichkeit zwischen einzelnen Knoten. Dazu gehört zum Beispiel die Ähnlichkeit des Namens. Verbindungsähnlichkeitsmaße messen die Ähnlichkeit zwischen Knotenmengen. So repräsentieren Elternknoten mit ähnlichen Kindknoten wahrscheinlich dasselbe Konzept und werden dadurch als ähnlich angesehen. Neben der Betrachtung der Kindknoten gehört hierzu zum Beispiel auch die Betrachtung der Grapheditierdistanz. Andere Ähnlichkeitsmaße berücksichtigen die Metamodelle von Quell- und Zielmodell. Für diese Maße werden Ähnlichkeiten zwischen Metamodellelementen berechnet. Zwei Modellelemente werden dann als ähnlich angesehen, wenn sie Instanzen von ähnlichen Metamodellelementen sind. Die Ergebnisse eines Algorithmus werden als Matrix betrachtet, und die Matrizen der verschiedenen Algorithmen werden in einem Würfel zusammengeführt. Der Würfel wird konfiguriert, zum Beispiel durch Bildung des Durchschnitts der Ähnlichkeitswerte für jede Modellelementkombination, und eine Abbildung von Quellmodellelementen auf Zielmodellelemente wird erzeugt. Diese Abbildung wird analysiert und anhand von Heuristiken werden Nachverfolgbarkeitsverbindungen gebildet.

Florean et al. [12] verwenden verschiedene auf maschinellem Lernen basierende Textklassifizierer, um Quelldateien Architekturmodulen zuzuordnen und vergleichen die Ergebnisse. Das Ziel des Einsatzes solcher Klassifizierer ist, die Zuordnung von Quelltexteinheiten zu den entsprechenden Architekturmodulen teilweise zu automatisieren. Anstatt dass für ein System alle Zuordnungen manuell durchgeführt werden müssen, reicht eine initiale manuell durchgeführte Zuordnung eines gewissen Anteils der Quelltexteinheiten aus. Auf dieser initialen Zuordnung wird ein Textklassifizierer trainiert. Die restlichen Quelltexteinheiten, die noch nicht zugeordnet wurden, werden dann automatisiert durch den trainierten Klassifizierer ihren Architekturmodulen zugeordnet. Als Textklassifizierer untersuchen Florean et al. Naive Bayes, Stützvektormaschinen (SVM) und logistische Regression. Ihre Experimente führen die Autoren auf fünf verschiedenen Systemen aus. Dabei betrachten sie nur Java-Quelltext. Im ersten Experiment werden verschiedene Varianten der Datenextraktion aus den Quelldateien und der Vorverarbeitung dieser Daten untersucht. Beispiele für extrahierte Daten sind Paketdeklarationen und öffentliche Methoden. Beispiele für die Vorverarbeitung sind die Stammformreduktion (*engl.* Stemming), das Kleinschreiben und das Entfernen von Java-Schlüsselwörtern. Zur Evaluierung wird eine Kreuzvalidierung durchgeführt und als Metrik wird die Genauigkeit verwendet. Die Ergebnisse lassen darauf schließen, dass Paketdeklarationen besonders wichtig für die Zuordnung sind, vor allem, wenn sich die Struktur der Architektur und der Pakete ähneln. Die besten Ergebnisse werden erzielt, wenn nur Paket- und Klassendeklarationen

extrahiert werden. In diesem Fall erzielen SVM und logistische Regression mit je 0,93 bessere Ergebnisse als Naive Bayes mit 0,86. Die Vorverarbeitung scheint im Vergleich zur Auswahl der extrahierten Daten weniger wichtig zu sein. Als weiteres Experiment untersuchen die Autoren, wie sich der Anteil der initial zugeordneten Quelldateien auf die Ergebnisse auswirkt. Die Ergebnisse lassen darauf schließen, dass ab 15 % keine großen Verbesserungen mehr erzielt werden. Im Gegensatz zum Ansatz dieser Masterarbeit wird hier also für ein betrachtetes System eine initiale Zuordnung eines gewissen Anteils der Quelltexteinheiten benötigt. Zudem betrachten Florean et al. nur Java-Quelltext und nur Architekturmodule, ohne Schnittstellen speziell zuzuordnen.

Einen ähnlichen auf maschinellem Lernen basierenden Ansatz verwenden Olsson et al. [22]. Auch sie trainieren einen Naive Bayes-Klassifikator auf einer initialen Zuordnung eines Anteils der Quelltexteinheiten. Dabei nutzen sie unter anderem Paket- und Datei-Namen. Zusammengesetzte Wörter werden getrennt und auf ihre Stammform reduziert. Mit ihrem Ansatz erzielen Olsson et al. bessere Ergebnisse als HuGMe.

HuGMe von Christl et al. [9] ist eine weitere Technik, die zur Zuordnung von Quelltext zu Architekturmodulen genutzt werden kann. Wie die Ansätze von Florean et al. [12] und Olsson et al. [22] benötigt HuGMe eine initiale Zuordnung von Quelltexteinheiten. HuGMe soll die manuelle Zuordnung nicht ersetzen, sondern unterstützen. Daher werden nur einfach zuzuordnende Quelltexteinheiten automatisiert ihren Architekturmodulen zugeordnet. Die restlichen Quelltexteinheiten werden dem Benutzer vorgelegt, der diese dann manuell zuordnen kann. Die Zuordnungsaufgabe wird bei HuGMe als Clustering-Aufgabe betrachtet. Konkret bedeutet das, dass Quelltexteinheiten, die demselben Architekturmodul zugeordnet werden, ein Cluster bilden. HuGMe nutzt einen inkrementellen Clustering-Algorithmus mit mehreren Schritten. Zuerst filtert der Algorithmus diejenigen Quelltexteinheiten, die nur wenig bis gar nicht mit den initial zugeordneten Quelltexteinheiten verbunden sind. Für diese Einheiten kann HuGMe keine sinnvolle Zuordnungsentscheidung treffen. Dann bildet der Algorithmus eine Anziehungsmatrix, die für jedes Paar von Quelltexteinheit und Architekturmodul einen Anziehungswert beinhaltet. Dieser Wert repräsentiert die Wahrscheinlichkeit, dass für dieses Paar eine Nachverfolgbarkeitsverbindung besteht. Er wird aufgrund von Abhängigkeiten zwischen der zuzuordnen Quelltexteinheit und den schon initial zugeordneten Einheiten gebildet. Christl et al. haben zwei Anziehungsfunktionen definiert, die auf existierenden Clustering-Methoden aufbauen. Auf Grundlage der Anziehungsmatrix werden die potentiellen Architekturmodule für jede der Quelltexteinheiten entdeckt. Der Algorithmus ordnet die Quelltexteinheiten, für die nur ein einziger Kandidat zur Zuordnung entdeckt wurde, automatisiert zu. Alle anderen werden dem Benutzer zur manuellen Zuordnung vorgelegt.

InMap von Sinkala und Herold [25] dient dazu, interaktiv Quelltexteinheiten ihren entsprechenden Architekturmodulen zuzuordnen. Dabei wird keine initiale Zuordnung eines gewissen Anteils der Quelltexteinheiten benötigt. Als Eingabe wird allerdings eine Beschreibung der Architekturmodule in natürlicher Sprache benötigt, die, falls sie nicht schon vorhanden ist, vom Benutzer angefertigt werden muss. Der Ansatz basiert auf Information Retrieval und hat sieben Schritte. Im ersten Schritt werden Entitäten, die der Benutzer nicht zuordnen will, herausgefiltert. In den nächsten Schritten werden die Quelldateien vorverarbeitet und indexiert. In Schritt vier wird für jedes Architekturmodul eine Suchabfrage gebildet, die zur Suche in den indexierten Dateien genutzt wird. Diese

enthält den Namen des Moduls, die natürlichsprachige Beschreibung des Moduls, die Namen der Klassen, die dem Modul schon zugeordnet sind, sowie die Namen der Methoden dieser Klassen. Im fünften Schritt wird die Ähnlichkeit zwischen Modulen und Klassen bewertet. Dafür werden die einzelnen Terme in der Suchabfrage mit ihrer term frequency-inverse document frequency gewichtet. In Schritt sechs wird für jede noch nicht zugeordnete Klasse das Architekturmodul mit der höchsten Ähnlichkeitsbewertung ausgewählt und die Ergebnisse gefiltert. Die nach ihrer Ähnlichkeitsbewertung sortierten Zuordnungen werden dem Benutzer angezeigt. Im siebten Schritt werden die Zuordnungen vom Benutzer angenommen oder abgelehnt. Auf Grundlage der neuen Informationen dieser Bewertung durch den Benutzer kehrt InMap zum vierten Schritt zurück. Dort werden für jede Suchabfrage die dem entsprechenden Modul neu zugeordneten Klassen der Suchabfrage hinzugefügt. Der Algorithmus stoppt, wenn alle Klassen zugeordnet wurden oder entweder der Benutzer oder InMap keine Entscheidungen mehr treffen können. Ebenso wie beim Ansatz dieser Masterarbeit wird bei InMap keine initiale Zuordnung eines gewissen Anteils der Quelltexteinheiten benötigt. Allerdings benötigt InMap eine Beschreibung der Architekturmodule in natürlicher Sprache. Darüber hinaus betrachtet InMap nur Architekturmodule, ohne Schnittstellen speziell zuzuordnen. In dieser Masterarbeit werden Nachverfolgbarkeitsverbindungen automatisiert gewonnen. Der Ansatz von InMap ist halbautomatisiert, da der Benutzer die empfohlenen Zuordnungen bewerten muss.

Keim et al. [19] befassen sich mit der Gewinnung von Nachverfolgbarkeitsverbindungen zwischen textuellen Softwarearchitekturdokumentationen und Architekturmodellen. Dafür stellen sie das Framework SWATTR vor, das verschiedene Ausführungsschritte beinhaltet. Zuerst wird die textuelle Dokumentation analysiert. Dabei kommen verschiedene Verfahren der Verarbeitung natürlicher Sprache wie die Lemmatisierung zum Einsatz. Dann werden relevante Informationen aus dem Text extrahiert. Ebenso werden aus dem Modell der Softwarearchitektur Modellelemente extrahiert. Die extrahierten Informationen werden genutzt, um mögliche Elemente in der textuellen Dokumentation zu finden. Zum Schluss werden die Modellelemente mit den gefundenen Textelementen verglichen. Ist die Konfidenz eines Vergleichs hoch genug, wird eine Nachverfolgbarkeitsverbindung zwischen den entsprechenden Elementen gebildet.

4. Ansatz

Im Folgenden wird der Ansatz dieser Arbeit, um automatisiert Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext zu gewinnen, vorgestellt. In Abschnitt 4.1 wird eine Übersicht über den Ansatz gegeben. Darauf folgend werden die einzelnen Schritte des Ansatzes in Abschnitt 4.2 und Abschnitt 4.3 je in einem eigenen Abschnitt näher erläutert.

4.1. Übersicht über den Ansatz

Im Rahmen dieser Arbeit wird ein Ansatz umgesetzt, der die automatisierte Generierung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext ermöglicht. Die Ausgangssituation dabei ist, dass für ein bestimmtes System der Quelltext sowie ein komponentenbasiertes Architekturmodell zur Verfügung stehen. Dabei müssen der Quelltext und das Architekturmodell keine besonderen Konsistenzeigenschaften erfüllen. Zwischen den Bestandteilen dieser beiden Artefakte sollen dann automatisiert die Nachverfolgbarkeitsverbindungen erstellt werden. Ein Eingreifen des Benutzers ist dabei nicht vorgesehen.

Der Ansatz dieser Arbeit soll nicht auf konkrete Programmiersprachen oder Architektur-Metamodelle festgelegt sein. Es soll also unter anderem möglich sein, für den Quelltext verschiedene Programmiersprachen zu unterstützen. Im ersten Schritt des Ansatzes namens Modellgenerierung werden daher der vorhandene Quelltext und das vorhandene Architekturmodell in neue Modelle überführt. Diese Modelle sind unabhängig von einer konkreten Programmiersprache oder einem konkreten Architektur-Metamodell. Damit soll die Erstellung von Nachverfolgbarkeitsverbindungen im folgenden Schritt erleichtert und für verschiedene Programmiersprachen und Architektur-Metamodelle vereinheitlicht werden. Die dafür erstellten Metamodelle heißen „Codemodell für Trace-Links“ (CMTL) bzw. „Architekturmodell für Trace-Links“ (AMTL). Das sogenannte „Trace-Link-Modell“ (TLM) beschreibt, wie Nachverfolgbarkeitsverbindungen zwischen dem durch das CMTL beschriebenen Quelltextmodell und dem durch das AMTL beschriebenen Architekturmodell aufgebaut sind.

Der zweite Schritt namens Trace-Link-Generierung umfasst die Generierung von Nachverfolgbarkeitsverbindungen. Dabei werden die neu erstellten Modelle aus dem Modellgenerierungsschritt genutzt. Es werden verschiedene Heuristiken eingesetzt, die Kandidaten für mögliche Nachverfolgbarkeitsverbindungen bewerten, also eine Aussage darüber treffen, wie wahrscheinlich ein Kandidat tatsächlich eine Nachverfolgbarkeitsverbindung ist. Dabei nutzen Heuristiken Informationen der Modellelemente wie ähnliche Namen als Hinweise. Auch Beziehungen zwischen verschiedenen Modellelementen werden durch

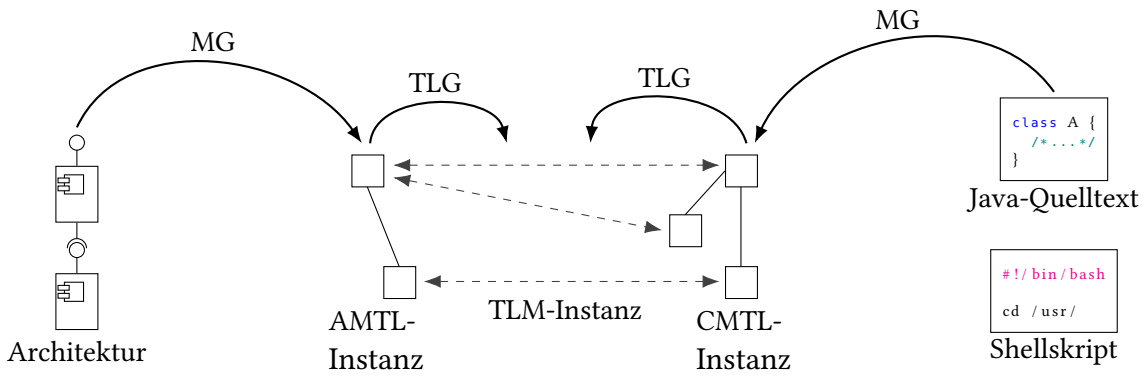

Abbildung 4.1.: Die zentralen Schritte des Ansatzes und ihre Ergebnisse

Heuristiken berücksichtigt. Die Ergebnisse der verschiedenen Heuristiken werden aggregiert, um die Nachverfolgbarkeitsverbindungen zu erhalten.

Die Generierung der Nachverfolgbarkeitsverbindungen arbeitet auf den im Modellgenerierungsschritt erstellten, von konkreten Programmiersprachen unabhängigen Modellen. Somit können in folgenden Arbeiten neue Programmiersprachen unterstützt werden, ohne dass die Heuristiken oder Aggregationsmethoden verändert werden müssen. Die den Heuristiken und Aggregationsmethoden zugrunde liegenden Datenstrukturen bleiben schließlich weiter bestehen. Werden das CMTL oder das AMTL in folgenden Arbeiten erweitert, um neue Aspekte des Quelltextes oder der Architektur zu erfassen, so bietet es sich an, diese Erweiterungen in die bestehenden Metamodelle zu integrieren, ohne die existierenden Strukturen grundlegend zu verändern. Dies kann zum Beispiel durch die Bildung neuer Unterklassen erfolgen. Wird dies so umgesetzt, müssen die Heuristiken oder Aggregationsmethoden nicht an die Erweiterungen angepasst werden. Es müssen aber neue Heuristiken erstellt werden, um die durch die Modellerweiterungen neu hinzugefügten Informationen bei der Generierung der Nachverfolgbarkeitsverbindungen zu berücksichtigen.

Die zentralen Schritte des Ansatzes und ihre Ergebnisse sind in Abbildung 4.1 dargestellt. Im Modellgenerierungsschritt (MG) werden aus einem vorhandenen Architekturmodell und Quelltext AMTL- und CMTL-Instanzen extrahiert. Dabei werden verschiedene Architektur-Metamodelle und Programmiersprachen unterstützt. In der Abbildung sind daher beispielhaft eine Java-Datei und ein Shellskript dargestellt. Im Trace-Link-Generierungsschritt (TLG) werden als TLM-Instanz Nachverfolgbarkeitsverbindungen zwischen den Elementen der extrahierten Modelle erstellt.

Weitere Details zum Modellgenerierungsschritt werden in Abschnitt 4.2 und zum Trace-Link-Generierungsschritt in Abschnitt 4.3 erläutert.

4.2. Modellgenerierung

Für den Modellgenerierungsschritt werden Modelle sowie Extrahierer für die Modellerstellung spezifiziert. Das betrifft zum einen Modelle des Quelltexts und zum anderen Modelle

der Architektur. Diese neuen Modelle sollen die Erstellung von Nachverfolgbarkeitsverbindungen erleichtern und für verschiedene Programmiersprachen und Architektur-Metamodelle vereinheitlichen. Auch die Nachverfolgbarkeitsverbindungen selbst werden modelliert.

Aus dem vorliegenden Quelltext des Systems wird ein für das weitere Vorgehen im TLG-Schritt geeignetes Modell erstellt. Das Modell soll für die Erstellung von Nachverfolgbarkeitsverbindungen zur Architektur relevante Informationen enthalten. Dabei sollen verschiedene Programmiersprachen unterstützt werden. Dazu werden übergreifende Konzepte verschiedener Programmiersprachen genutzt. Zur Beschreibung der Modelle wird das CMTL erstellt.

Das von OMG spezifizierte KDM enthält im Quelltextpaket Metamodellelemente, die allgemeine, sprachunabhängige Programmelemente repräsentieren [1, S. 83]. Die Sprachunabhängigkeit wird auch dadurch demonstriert, dass es schon für einige Programmiersprachen Arbeiten zur Erstellung von KDM-Modellen gibt. Solche Arbeiten gibt es zum Beispiel für Java [7], C# [30], Cobol [3] und PHP [28]. Für das CMTL werden Ausschnitte aus dem KDM übernommen, die für den TLG-Schritt relevant sind. Zum Beispiel enthält das CMTL Konzepte wie Module und Datentypen. Für die Erstellung der Nachverfolgbarkeitsverbindungen nicht unmittelbar benötigte Details sind hingegen nicht im Modell enthalten. Dazu gehören zum Beispiel Anweisungen in Methoden. Eine genaue Beschreibung des CMTL findet sich in Unterabschnitt 5.2.2.

Um das Modell des Quelltexts, also eine CMTL-Instanz, zu generieren, werden Modellextrahierer eingesetzt. Zur Unterstützung verschiedener Programmiersprachen wird für diese je ein eigener Extrahierer entwickelt. Dabei werden schon existierende Werkzeuge für die Quelltextverarbeitung genutzt.

Aus der Architektur des Systems wird ebenfalls ein für das weitere Vorgehen geeignetes Modell erstellt. Die Architektur liegt dabei bereits als Modell vor. Die für die Erstellung der Nachverfolgbarkeitsverbindungen relevanten Informationen werden in die in dieser Arbeit spezifizierte Modellform überführt. Es können dabei verschiedene Metamodelle für die vorliegende Architektur unterstützt werden. Vorliegende Architekturmodelle werden somit in eine gemeinsame Form gebracht. Das Modell, in das die Architektur überführt wird, enthält unter anderem die verschiedenen Komponenten der Architektur mitsamt der Schnittstellen, die sie anbieten bzw. benötigen. Es wird entsprechend das AMTL erstellt, das sich an Ausschnitten des PCM orientiert. Eine genaue Beschreibung des AMTL findet sich in Unterabschnitt 5.2.3.

Um das vorhandene Architekturmodell in eine AMTL-Instanz zu überführen, wird ein Extrahierer verwendet. Dabei ist der Extrahierer spezifisch für das Metamodell des vorhandenen Architekturmodells. Für PCM und UML werden entsprechende Extrahierer erstellt.

Auch das TLM wird erstellt. Es beschreibt, wie Nachverfolgbarkeitsverbindungen zwischen Quelltext- und Architekturmodell aufgebaut sind. Die Menge der konkreten Nachverfolgbarkeitsverbindungen zwischen dem Quelltextmodell und dem Architekturmodell eines konkreten Systems ist also eine TLM-Instanz. Eine Nachverfolgbarkeitsverbindung hat genau zwei Endpunkte, einen in der AMTL-Instanz und einen in der CMTL-Instanz. Sie beschreibt, dass der Quelltext-Endpunkt ein Teil der Implementierung des Architektur-Endpunktes ist. Der Architektur-Endpunkt kann zum Beispiel eine Komponente und der

Quelltext-Endpunkt eine Kompilierungseinheit sein. Aus einem Architektur-, bzw. Quelltext-Endpunkt lässt sich leicht eindeutig auf das entsprechende Element im originalen Architekturmodell bzw. Quelltext schließen. Somit können aus einer Nachverfolgbarkeitsverbindung zwischen Modellelementen von AMTL- und CMTL-Instanzen leicht die entsprechenden Elemente im ursprünglichen Architekturmodell und Quelltext identifiziert werden. Eine genaue Beschreibung des TLM findet sich in Unterabschnitt 5.2.4.

4.3. Generierung von Nachverfolgbarkeitsverbindungen

Zur Generierung der Nachverfolgbarkeitsverbindungen werden im TLG-Schritt verschiedene Heuristiken angewendet. Eine Heuristik bewertet Kandidaten für mögliche Nachverfolgbarkeitsverbindungen. Sie trifft also eine Aussage darüber, ob sie einen Kandidaten als eine tatsächliche Nachverfolgbarkeitsverbindung betrachtet. Dies ist dann der Fall, wenn sie in den Modellen einen Hinweis darauf findet. Wenn eine Heuristik einen Hinweis findet, kann sie auch bewerten, wie sicher sie sich ist, dass es sich bei dem Kandidaten tatsächlich um eine Nachverfolgbarkeitsverbindung handelt. Sie kann also mit einer Zahl darstellen, wie vielversprechend der gefundene Hinweis ist.

Eine Heuristik ordnet den Kandidaten also je eine Konfidenz zu. Ein Kandidat für eine Nachverfolgbarkeitsverbindung ist ein Tupel, bestehend aus einem Architektur- und einem Quelltext-Endpunkt. Die Konfidenz sagt aus, ob die Heuristik einen Hinweis gefunden hat, dass zwischen diesen Endpunkten tatsächlich eine Nachverfolgbarkeitsverbindung besteht, und wenn ja, wie sicher sie sich ist. Eine Heuristik betrachtet jedes mögliche Tupel von Endpunkten eines Architektur- und eines Quelltextmodells und ordnet ihm eine Konfidenz zu. Dabei kann die Konfidenz keinen spezifischen Wert haben, nämlich dann, wenn die Heuristik keinen Hinweis auf eine Nachverfolgbarkeitsverbindung findet. Findet die Heuristik einen Hinweis, so wählt sie für die Konfidenz einen Wert zwischen null und eins. Ein hoher Wert zeigt an, dass laut der Heuristik eine generierte Nachverfolgbarkeitsverbindung wahrscheinlich korrekt wäre.

Es gibt auch Heuristiken, die nach Hinweisen suchen, dass ein Kandidat keine tatsächliche Nachverfolgbarkeitsverbindung darstellt. Diese Heuristiken weisen den Kandidaten, für die sie solche Hinweise finden, ebenfalls je eine Konfidenz mit einem Wert zu. Hier zeigt ein hoher Wert an, dass laut der Heuristik eine generierte Nachverfolgbarkeitsverbindung wahrscheinlich nicht korrekt wäre.

Heuristiken können auch nur bestimmte Arten von Kandidaten betrachten, zum Beispiel nur Endpunkttupel mit bestimmten Architekturelementen wie Schnittstellen. Alle nicht betrachteten Kandidaten erhalten dann eine Konfidenz ohne Wert. Somit ordnen Heuristiken nur den Endpunkttupeln, die sie betrachten und für die sie einen Hinweis auf eine Nachverfolgbarkeitsverbindung finden, Konfidenzen mit Werten zu.

Heuristiken können Informationen der Modellelemente wie ähnliche Namen nutzen. Wenn zum Beispiel ein Architekturelement *MediaStore* heißt, und ein Quelltextelement *MediaStoreImpl*, ist das ein Indiz für eine Nachverfolgbarkeitsverbindung. Hier kann eine Heuristik zum Beispiel überprüfen, ob ein Name im anderen Namen enthalten ist. Andere Heuristiken betrachten Ähnlichkeiten in der Struktur von Modellelementen des Architektur- und des Quelltextmodells.

Um geeignete Heuristiken zu finden, können Ansätze aus verwandten Arbeiten genutzt werden. Zum Beispiel wird in Vitruvius [21] zur Konsistenzhaltung definiert, dass es für eine Komponente eine entsprechende Klasse gibt, die denselben Namen mit dem Zusatz *Impl* hat. Daraus lässt sich eine Heuristik ableiten, die Namen auf allgemeine Begriffe wie *Impl* untersucht.

Um die Nachverfolgbarkeitsverbindungen zu erstellen, werden die Ergebnisse der Heuristiken aggregiert. Dabei werden Aggregationsmethoden wie zum Beispiel das Maximum der Konfidenzen oder Schwellwerte verwendet. Der Gedanke hinter den Schwellwerten ist, dass zum einen nicht jede Heuristik gleich gute Ergebnisse erzielt und zum anderen Heuristiken Konfidenzen auf unterschiedliche Art und Weise verteilen können. So kann eine Heuristik Endpunkttupel eher konservativ, das heißt eher niedrig bewerten, während eine andere oft hohe Konfidenzen vergibt. Um diese Unterschiede zwischen den Heuristiken auszugleichen, können die Schwellwerte entsprechend gewählt werden.

Andere Aggregationsmethoden weisen zum Beispiel basierend auf dem Ergebnis einer Heuristik jedem Architektur-Endpunkt die Quelltext-Endpunkte mit der höchsten Konfidenz zu. Für solche Aggregationsmethoden müssen keine konkreten Parameterwerte festgelegt werden.

Heuristiken, die bewerten, ob ein Kandidat keine tatsächliche Nachverfolgbarkeitsverbindung darstellt, müssen ebenfalls geeignet aggregiert werden. Das heißt, die Kandidaten, für die von diesen Heuristiken Hinweise gefunden wurden, müssen aus den Ergebnissen der anderen Heuristiken gefiltert anstatt hinzugefügt werden.

5. Architektur

In diesem Kapitel wird die gewählte Architektur zur Umsetzung des Ansatzes aus Kapitel 4 erläutert. Zuerst wird in Abschnitt 5.1 ein grober Überblick über die Architektur gegeben. In Abschnitt 5.2 wird dann detaillierter auf einzelne Architekturausschnitte eingegangen.

5.1. Übersicht über die Architektur

Das Projekt zur automatisierten Generierung von Nachverfolgbarkeitsverbindungen zwischen Quelltext und Architektur wird ARCOTL (Architecture-Code-Trace-Links) genannt. Das Projekt ist in vier Modulen implementiert: *models*, *modelgeneration*, *tracelinkgeneration* und *evaluation*. Jedes der vier Module enthält ein Paket mit gegebenenfalls mehreren Unterpaketen. Die Paketstruktur des Projekts wird in Abbildung 5.1 als UML-Paketdiagramm dargestellt. Dabei stellt die *use*-Beziehung die Abhängigkeiten der Module dar. Es gibt keine zyklischen Abhängigkeiten zwischen den Modulen. Durch die Aufteilung des Projekts in mehrere Module wird die Komplexität der Abhängigkeiten reduziert. Damit kann das System leichter verstanden und gewartet werden.

Das Modul *models* enthält das Paket *models* und hat keine Abhängigkeiten zu den anderen Modulen. Es enthält die Implementierung der Modelle CMTL, AMTL und TLM für Quelltext, Architektur und Nachverfolgbarkeitsverbindungen. Die Implementierung der Modelle ist somit unabhängig von der konkreten Modellextraktion und der konkreten Generierung der Nachverfolgbarkeitsverbindungen. Die Unterpakete *amtl*, *cmtl* und *tlm* enthalten die Klassen und Schnittstellen, die spezifisch für das entsprechende Metamodell sind.

Das Modul *modelgeneration* enthält das Paket *mg*. Es setzt den Modellgenerierungsschritt des Ansatzes aus Kapitel 4 um. Das Modul enthält die Implementierung von Extrahierern, die aus gegebenen Architekturmodellen bzw. Quelltext AMTL- bzw. CMTL-Instanzen extrahieren. Daher hat das Modul auch eine Abhängigkeit zum Modul *models*. Das Unterpaket *architecture* enthält die Extrahierer, die AMTL-Instanzen erstellen, während das Unterpaket *code* die Extrahierer enthält, die CMTL-Instanzen extrahieren. Für die bei der Modellgenerierung unterstützten Programmiersprachen und Architektur-Metamodelle existiert je ein eigenes Unterpaket. Das sind im Rahmen dieser Arbeit Java und Shell bzw. PCM und UML.

Das Modul *tracelinkgeneration* enthält das Paket *tlg*. Es setzt den Trace-Link-Generierungsschritt des Ansatzes aus Kapitel 4 um. Das Modul ist dafür verantwortlich, automatisiert die Nachverfolgbarkeitsverbindungen zwischen einer AMTL- und einer CMTL-Instanz zu generieren. Dafür benötigt das Modul die Implementierung der Modelle im Modul *models* und hat daher eine entsprechende Abhängigkeit. Das *functions*-Paket enthält in seinen Unterpaketen die Heuristiken und Aggregationen, die zur Generierung der Nachverfolg-

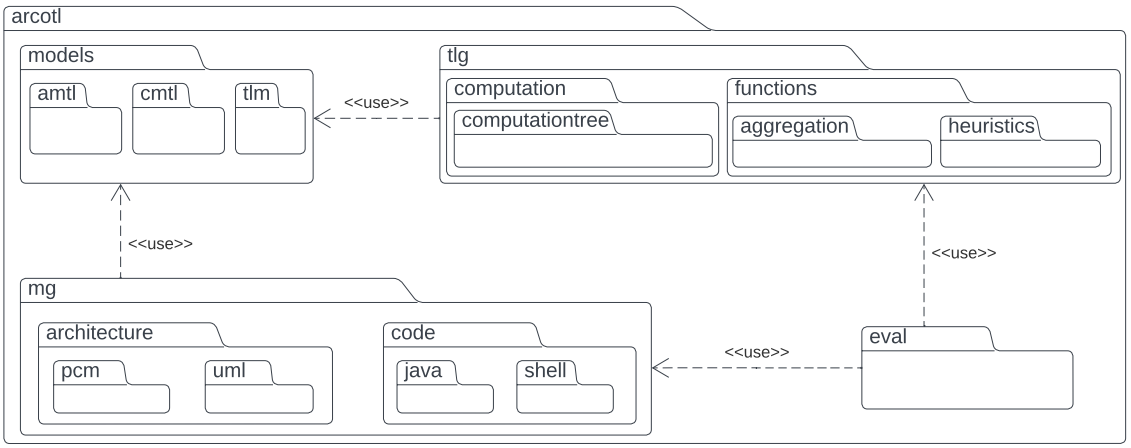

Abbildung 5.1.: Die Paketstruktur des ARCOTL-Projekts

barkeitsverbindungen verwendet werden. Im *computation*-Paket wird die Berechnung der Nachverfolgbarkeitsverbindungen für gegebene Modelle implementiert. Dafür enthält das Paket zum Beispiel Klassen, die die Berechnungsergebnisse repräsentieren. Zudem ist das *computationtree*-Unterpaket enthalten. Dieses enthält die Implementierung des Berechnungsbaumes. Dieser wird in Unterabschnitt 5.2.6 erläutert.

Das Modul *evaluation* enthält das Paket *eval*. Es wird in der in Kapitel 7 beschriebenen Evaluation verwendet. Das Modul enthält die verschiedenen Fallstudien, die zur Evaluation genutzt werden sollen. Mithilfe der Module *modelgeneration* und *tracelinkgeneration* werden automatisiert Modelle aus den Fallstudien extrahiert, Heuristiken angewandt und Nachverfolgbarkeitsverbindungen zwischen den Modellen erzeugt. Daher hat das Modul *evaluation* auch Abhängigkeiten zu den Modulen *modelgeneration* und *tracelinkgeneration*, sowie indirekt auch zum *models*-Modul. Für die erzeugten Ergebnisse werden in Skripten verschiedene Metriken berechnet.

In Abbildung 5.2 wird ein typischer Ablauf zur automatisierten Generierung von Nachverfolgbarkeitsverbindungen dargestellt. Damit illustriert die Abbildung auch das Zusammenspiel der verschiedenen Module. Die *Evaluation* aus dem *evaluation*-Modul generiert hier für ein existierendes Architekturmodell und den Quelltext eines Systems Nachverfolgbarkeitsverbindungen. Dazu ruft sie zuerst im *ArchitectureExtractor* die `extractModel`-Methode mit dem Pfad zum existierenden Architekturmodell auf. Diese Methode extrahiert dann eine AMTL-Instanz und gibt sie zurück. Analog wird ein Quelltextmodell mithilfe des *CodeExtractors* erzeugt. Dabei zeigt der Pfad auf den Ordner mit dem Quelltext. Der *ArchitectureExtractor* und der *CodeExtractor* befinden sich im *modelgeneration*-Modul. Anschließend wird im *TraceLinkGenerator* die `generateTraceLinks`-Methode mit der AMTL- sowie der CMTL-Instanz aufgerufen. Diese Methode generiert automatisiert die Nachverfolgbarkeitsverbindungen zwischen den Modellen und gibt sie zurück. Der *TraceLinkGenerator* befindet sich im *tracelinkgeneration*-Modul. Die generierten Nachverfolgbarkeitsverbindungen können dann im *evaluation*-Modul zur Berechnung von Metriken weiterverwendet werden.

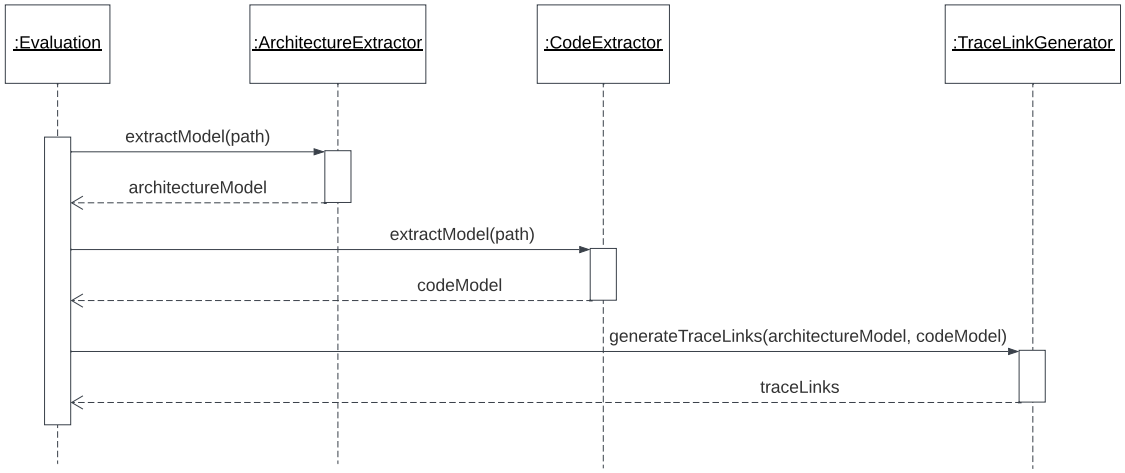

Abbildung 5.2.: Typischer Ablauf der automatisierten Generierung von Nachverfolgbarkeitsverbindungen

5.2. Detaillierte Architektur

In den folgenden Abschnitten wird detaillierter auf einzelne Architekturausschnitte eingegangen. Zuerst wird in Unterabschnitt 5.2.1 die allgemeine Struktur der Quelltext- und Architekturmodelle erläutert. In Unterabschnitt 5.2.2 und Unterabschnitt 5.2.3 werden das Quelltext-Metamodell CMTL und das Architektur-Metamodell AMTL beschrieben. Das Metamodell der Nachverfolgbarkeitsverbindungen zwischen AMTL- und CMTL-Instanzen wird in Unterabschnitt 5.2.4 erläutert. In Unterabschnitt 5.2.5 wird auf die Architektur der Berechnung der Nachverfolgbarkeitsverbindungen eingegangen. Zum Schluss wird in Unterabschnitt 5.2.6 die Architektur des bei der Berechnung verwendeten Berechnungsbaumes beschrieben. Die genaue Funktionsweise von Heuristiken und Aggregationen wird erst in Kapitel 6 erläutert.

5.2.1. Modellstruktur

In Abbildung 5.3 ist die allgemeine Struktur der Modelle in einem UML-Klassendiagramm dargestellt. Diese Struktur basiert auf dem KDM [1]. Jedes Modell (*Model*) ist ein Modellelement (*ModelElement*). Die Modell-Klasse basiert auf der Klasse *KDMModel* im KDM [1, S. 39]. Ein Modell kann beliebig viele Entitäten enthalten. Eine Entität (*Entity*) ist ebenfalls ein Modellelement. Die Entität-Klasse basiert auf der Klasse *KDMEntity* [1, S. 41]. Eine Entität hat einen Namen. In dieser Arbeit werden Quelltext- und Architekturmodelle betrachtet. Wie in den folgenden Abschnitten beschrieben, bestehen für Quelltext- bzw. Architekturmodelle Einschränkungen, welche Arten von Entitäten sie direkt enthalten können.

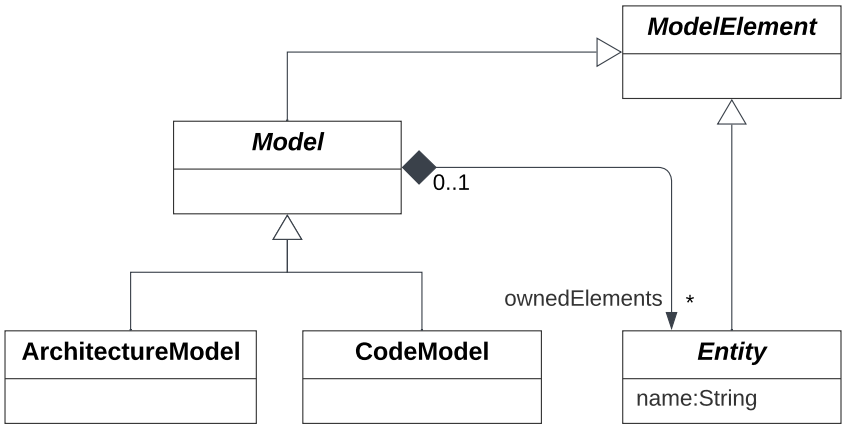

Abbildung 5.3.: Die allgemeine Struktur der Modelle

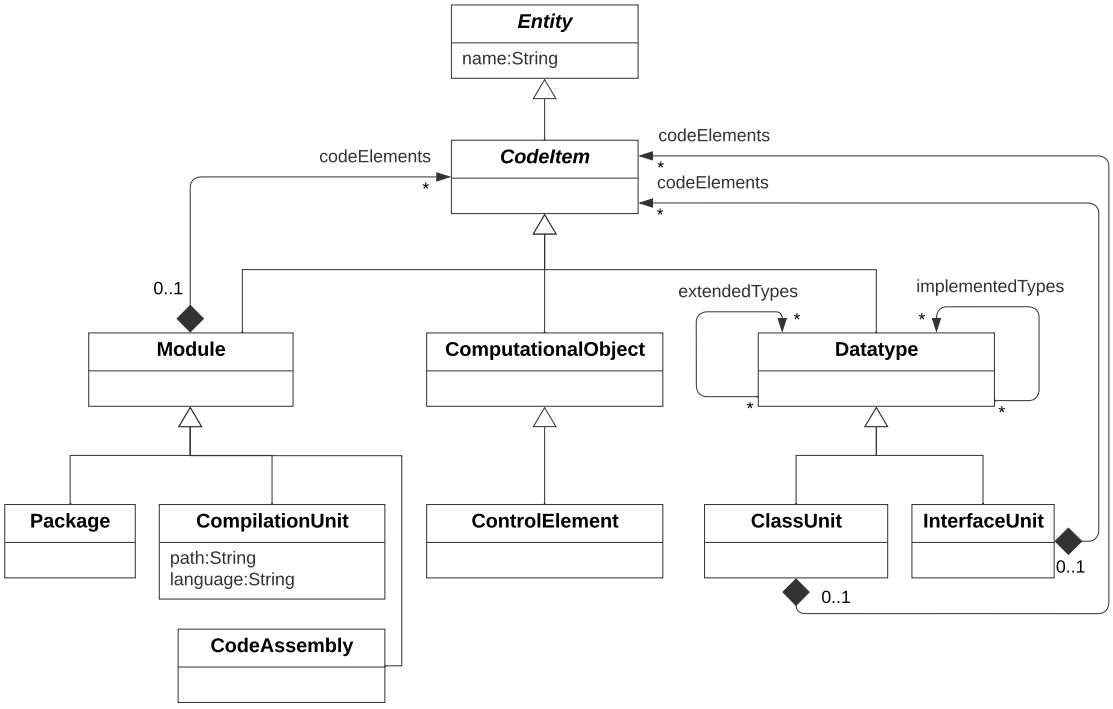

Abbildung 5.4.: Das Quelltext-Metamodell CMTL

5.2.2. Quelltextmodell

In Abbildung 5.4 ist ein das CMTL beschreibendes UML-Klassendiagramm dargestellt. Die Elemente basieren auf dem Quelltextpaket des KDM [1]. Ein allgemeines Quelltextelement (*CodeItem*) ist eine Entität und hat damit einen Namen. Es gibt drei Arten von Quelltextelementen. Sie werden durch die Unterklassen Modul (*Module*), Berechnungsobjekt (*ComputationalObject*) und Datentyp (*Datatype*) repräsentiert.

Ein Modul ist eine Programmeinheit, die andere Quelltextelemente enthalten kann. Es werden verschiedene spezielle Arten von Modulen betrachtet. Eine Kompilierungseinheit (*CompilationUnit*) ist die einer Quelldatei entsprechende Programmeinheit. Sie kann Datentypen und Berechnungsobjekte enthalten. Im KDM können Dateien separat mit einem *InventoryModel* modelliert werden. Für diese Arbeit werden aus dem *InventoryModel* der Dateipfad und die Programmiersprache von Quelldateien genutzt. Sonstige Informationen aus dem *InventoryModel* werden in dieser Arbeit nicht benötigt. Daher wird in dieser Arbeit kein zusätzliches *InventoryModel* erstellt, sondern die Kompilierungseinheit erhält ihren Dateipfad und ihre Programmiersprache als Attribute. Der Dateipfad ist dabei relativ zum Pfad des Verzeichnisses, in dem sich der Quelltext des Systems befindet. Kompilierungseinheiten können in anderen Modulen wie zum Beispiel Paketen liegen.

Ein Paket (*Package*) ist eine logische Sammlung von Quelltextelementen. Pakete können verschachtelt sein, also Unterpakete enthalten. Sie werden zum Beispiel von Java unterstützt.

Ein *CodeAssembly* beinhaltet Quelltextelemente, die zusammen lauffähig gemacht werden, indem sie beispielsweise kompiliert und gelinkt werden. In C sind zum Beispiel Quelldateien und in sie eingefügte Header-Dateien in einer *CodeAssembly*.

Module werden normalerweise als logische Komponenten des Systems verwendet, die die Kapselung fördern [1, S. 88]. Damit bietet die Modulstruktur vielversprechende Hinweise für die Generierung von Nachverfolgbarkeitsverbindungen zu den Komponenten des Architekturmodells.

Im CMTL sind zwei spezielle Arten von Datentypen enthalten. Eine Klasse (*ClassUnit*) kann in objektorientierten Programmiersprachen wie Java vorkommen. Sie kann Quelltextelemente wie Methoden oder innere Klassen enthalten. Eine Schnittstelle (*InterfaceUnit*) kann ebenfalls Quelltextelemente wie Methoden enthalten. Auch die *implements*-Beziehung [1, S. 129] sowie die *extends*-Beziehung [1, S. 139] aus dem KDM sind im Quelltext-Metamodell enthalten. So kann mit den *implementedTypes* die Implementierung einer Schnittstelle im Modell repräsentiert werden. Ebenso kann mit den *extendedTypes* die Erweiterung eines Datentypen im Modell repräsentiert werden. In objektorientierten Programmiersprachen können zum Beispiel Klassen von anderen Klassen erben. Dabei ist Mehrfachvererbung möglich, das heißt eine Klasse kann von mehreren anderen Klassen erben. In Java ist das zwar nicht möglich, aber andere Programmiersprachen wie C++ unterstützen diese Form der Vererbung. Ebenso kann zum Beispiel in Java eine Schnittstelle andere Schnittstellen erweitern. In manchen Programmiersprachen wie TypeScript können Schnittstellen auch Klassen erweitern. Weitere bestimmte Arten von Datentypen aus dem KDM sind nicht im Quelltext-Metamodell enthalten. Dazu gehören unter anderem die vielen verschiedenen Arten von primitiven Typen wie *Boolean*. Sollten zukünftige Arbeiten

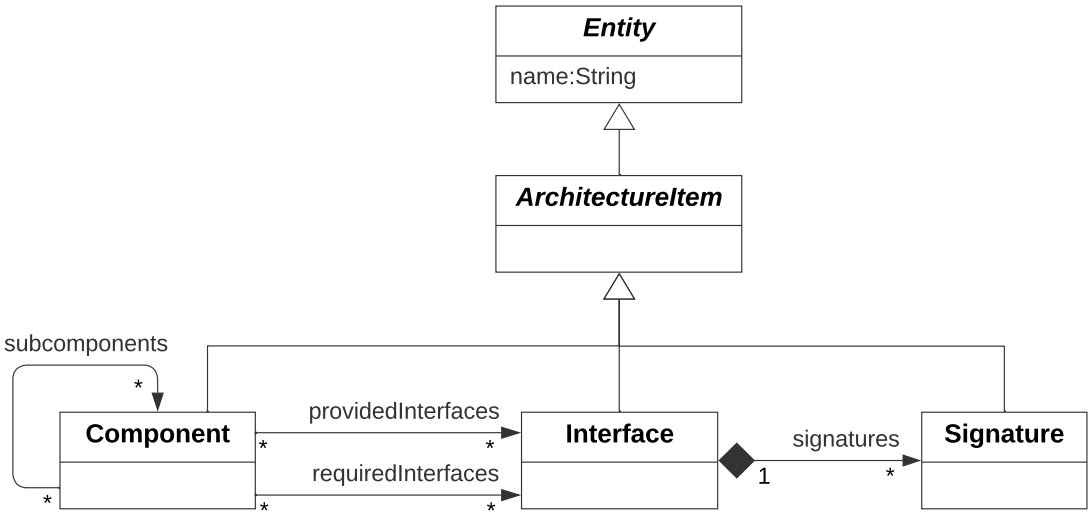

Abbildung 5.5.: Das Architektur-Metamodell AMTL

also Datentypen aus dem Architekturmodell und dem Quelltext umfassend vergleichen wollen, müssten weitere Unterklassen aus dem KDM übernommen werden.

Im CMTL ist mit dem Kontrollelement (*ControlElement*) nur eine Art von Berechnungsobjekten enthalten. Ein Kontrollelement repräsentiert aufrufbare Elemente mit einem bestimmten Verhalten. Das beinhaltet Funktionen, Prozeduren und Methoden. Im KDM wird dabei noch zusätzlich zwischen *CallableUnits* und *MethodUnits* unterschieden. *MethodUnits* sind dabei im Gegensatz zu *CallableUnits* in Klassen enthalten. Diese Unterscheidung wird in dieser Arbeit nicht genutzt. Außer dem Namen sind keine weiteren Teile der Signatur im CMTL enthalten. Parameter und Rückgabetypen werden in dieser Arbeit nicht als Hinweise zur Generierung von Nachverfolgbarkeitsverbindungen verwendet. Sollten zukünftige Arbeiten diese Elemente verwenden wollen, können die entsprechenden KDM-Klassen verwendet werden.

Ein Quelltextmodell darf nur Module direkt enthalten. Alle sonstigen Quelltextelemente dürfen nur indirekt im Quelltextmodell enthalten sein. Also kann eine Klasse zum Beispiel in einer Kompilierungseinheit enthalten sein. Die Kompilierungseinheit kann dann direkt im Quelltextmodell enthalten sein oder zum Beispiel in einem Paket liegen.

5.2.3. Architekturmodell

Abbildung 5.5 zeigt ein UML-Klassendiagramm, das das AMTL beschreibt. Analog zum Quelltextelement im CMTL gibt es im AMTL das Architekturelement (*ArchitectureItem*). Die verschiedenen Arten von Architekturelementen basieren auf dem PCM. Eine Komponente (*Component*) ist ein Baustein einer Softwarearchitektur. Sie kann optional Subkomponenten haben. Eine *Basic Component* im PCM entspricht im AMTL einer Komponente ohne Subkomponenten. Währenddessen entspricht eine *Composite Component* im PCM im

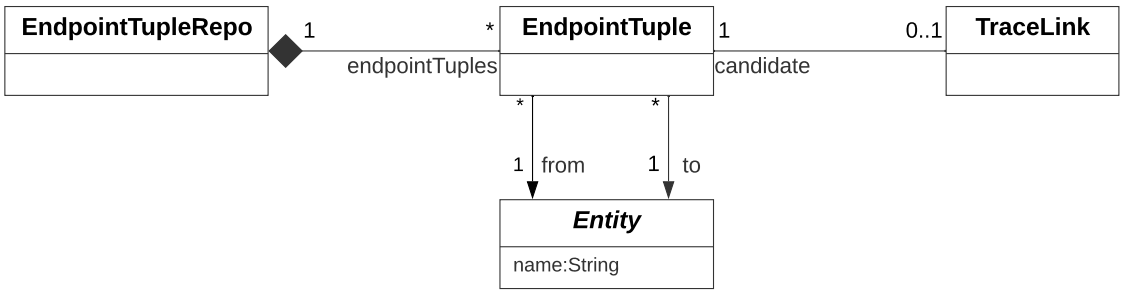

Abbildung 5.6.: Das Metamodell der Nachverfolgbarkeitsverbindungen TLM

AMTL einer Komponente mit Subkomponenten. Eine Komponente kann Schnittstellen (*Interfaces*) anbieten oder benötigen. Angebotene Schnittstellen werden von der Komponente implementiert. Eine Komponente bietet also die Funktionalität an, die in ihren angebotenen Schnittstellen spezifiziert wird. Benötigte Schnittstellen spezifizieren die Funktionalitäten, die von der Komponente benötigt werden. Eine Schnittstelle enthält eine beliebig große Menge von Methodensignaturen (*Signatures*). Wie im CMTL sind auch im AMTL außer dem Namen keine weiteren Teile der Signatur enthalten. Ein Architekturmodell darf nur Komponenten und Schnittstellen direkt enthalten.

Die genaue Verdrahtung der Komponenten, also wie genau die Komponenten in einem System über ihre angebotenen und benötigten Schnittstellen miteinander verbunden sind, wird in dieser Arbeit nicht verwendet. Aus dem PCM ist somit nur die vertikale Komposition, also die Kapselung von Komponenten in *Composite Components*, durch die Subkomponenten-Assoziation im AMTL enthalten. Die horizontale Komposition, also die systemspezifische Verdrahtung der Komponenten, ist im AMTL nicht enthalten. Für Palladio basieren AMTL-Instanzen somit nur auf dem Komponenten-Repository.

5.2.4. Modell der Nachverfolgbarkeitsverbindungen

Ein das TLM beschreibende UML-Klassendiagramm ist in Abbildung 5.6 dargestellt. Die möglichen Enden von Nachverfolgbarkeitsverbindungen werden als Endpunkte bezeichnet. Ein Endpunkttupel (*EndpointTuple*) hat genau zwei Endpunkte. Ein Endpunkt ist dabei die Quelle und einer das Ziel. Ein Endpunkttupel ist ein Kandidat für eine Nachverfolgbarkeitsverbindung (*TraceLink*). Zwischen den Endpunkten eines Endpunkttupels kann also optional eine Nachverfolgbarkeitsverbindung bestehen. Eine Nachverfolgbarkeitsverbindung hat immer genau ein Endpunkttupel, verbindet also genau zwei Endpunkte in einer bestimmten Richtung. Es kann keine zwei Nachverfolgbarkeitsverbindungen mit denselben Endpunkten in derselben Richtung geben.

Allgemein kann jede Entität ein Endpunkt sein. In dieser Arbeit hat ein Endpunkttupel genau einen Endpunkt im Architekturmodell und genau einen im Quelltextmodell. Der Architektur-Endpunkt ist dabei immer die Quelle und der Quelltext-Endpunkt immer das Ziel. Als Architektur-Endpunkte werden in dieser Arbeit Komponenten und Schnittstellen verwendet. Als Quelltext-Endpunkte werden Kompilierungseinheiten verwendet. Signaturen,

Berechnungsobjekte und Datentypen sind also keine Endpunkte. Der Grund dafür ist, dass im Rahmen dieser Arbeit keine solch feingranularen Nachverfolgbarkeitsverbindungen erzeugt werden. Von den Modulen im Quelltextmodell sind nur Kompilierungseinheiten Endpunkte. Die anderen grobgranulareren Arten von Modulen wie Pakete sind keine Endpunkte. Wird eines dieser Module einer Komponente vollständig zugeordnet, werden Nachverfolgbarkeitsverbindungen zwischen der Komponente und allen direkt oder indirekt im Modul enthaltenen Kompilierungseinheiten erstellt.

Das Endpunkttupel-Repository (*EndpointTupleRepo*) enthält alle möglichen Endpunkttupel für die Modelle, zwischen denen Nachverfolgbarkeitsverbindungen erzeugt werden sollen. In dieser Arbeit sind diese Modelle immer ein Architektur- und ein Quelltextmodell.

5.2.5. Berechnung der Nachverfolgbarkeitsverbindungen

In Abbildung 5.7 wird das UML-Klassendiagramm der Berechnung im Trace-Link-Generierungsschritt dargestellt. Eine Berechnung (*Computation*) dient in dieser Arbeit dazu, Nachverfolgbarkeitsverbindungen zwischen den Elementen eines Architektur- und eines Quelltextmodells zu generieren. Eine Berechnung hat somit je genau ein solches Modell und berechnet Nachverfolgbarkeitsverbindungen zwischen ihren Endpunkten. Dazu verwendet die Berechnung verschiedene Heuristiken. Wie in Unterabschnitt 5.2.4 beschrieben, enthält das Endpunkttupel-Repository für das Architektur- und das Quelltextmodell der Berechnung alle möglichen Endpunkttupel. Jedes dieser Endpunkttupel ist ein möglicher Kandidat für eine Nachverfolgbarkeitsverbindung. Somit werden letztendlich aus dem Endpunkttupel-Repository die Nachverfolgbarkeitsverbindungen ausgewählt. Eine Heuristik ordnet jedem der Endpunkttupel aus dem Endpunkttupel-Repository je eine Konfidenz zu. Wenn eine Heuristik für ein Endpunkttupel einen Hinweis darauf findet, dass hier eine Nachverfolgbarkeitsverbindung existiert, so ordnet sie diesem Endpunkttupel einen Konfidenzwert zwischen null und eins zu. Der Wert sagt aus, wie vielversprechend der Hinweis ist. Allen anderen Endpunkttupeln weist die Heuristik eine Konfidenz ohne Wert zu. Wie in Abschnitt 4.3 beschrieben gibt es auch Heuristiken, die nach Hinweisen suchen, dass ein Endpunkttupel keine tatsächliche Nachverfolgbarkeitsverbindung darstellt. Hier zeigt ein hoher Konfidenzwert an, dass für das Endpunkttupel eine generierte Nachverfolgbarkeitsverbindung wahrscheinlich nicht korrekt wäre.

Die Heuristiken werden in einem Berechnungsbaum organisiert und aggregiert. Dieser wird in Unterabschnitt 5.2.6 näher erläutert. Eine Berechnung hat somit einen Berechnungsbaum, dargestellt durch seinen Wurzelknoten. Anhand dieses Berechnungsbaumes und den zwei Modellen generiert die Berechnung die Nachverfolgbarkeitsverbindungen.

Eine Berechnung hat genau ein Berechnungsergebnis (*ComputationResult*). In diesem werden alle von dieser Berechnung erzeugten Konfidenzen gespeichert. Sobald die Berechnung abgeschlossen ist, weist das Berechnungsergebnis jeder Kombination bestehend aus einem Knoten aus dem Berechnungsbaum und einem Endpunkttupel aus dem Endpunkttupel-Repository genau eine Konfidenz zu. Dies wird durch eine qualifizierte Assoziation dargestellt. Somit kann zum Beispiel für einen bestimmten Knoten auf die von diesem Knoten berechneten Konfidenzen aller Endpunkttupel aus dem Endpunkttupel-Repository zugegriffen werden. Andersherum kann auch für ein bestimmtes Endpunkttupel auf die von jedem Knoten des Berechnungsbaumes berechneten Konfidenzen dieses Tupels

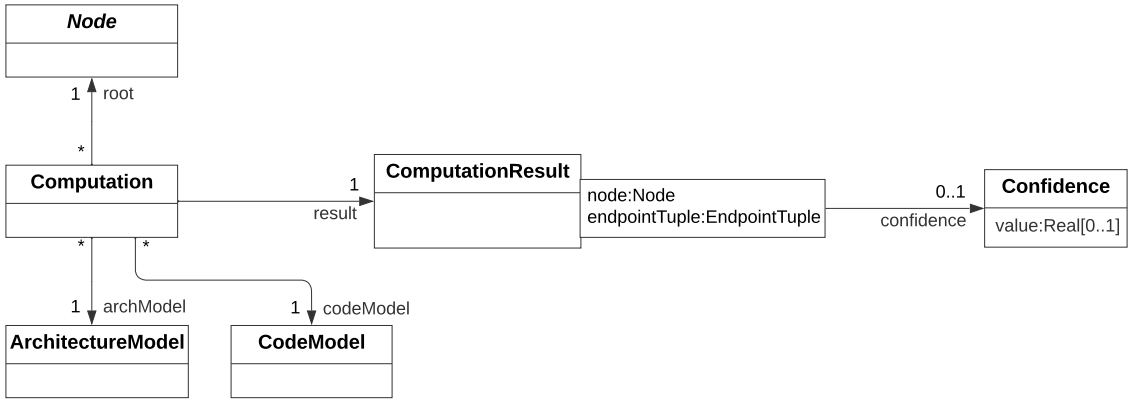

Abbildung 5.7.: Klassendiagramm der Berechnung

zugegriffen werden. Die vom Wurzelknoten berechneten Konfidenzen stellen die finalen Konfidenzen dar, auf deren Grundlage die Nachverfolgbarkeitsverbindungen erstellt werden.

5.2.6. Berechnungsbaum

Der Berechnungsbaum organisiert und kombiniert verschiedene Heuristiken. Abbildung 5.8 zeigt das UML-Klassendiagramm des Berechnungsbaumes.

Eine Heuristik ordnet Endpunkttupeln Konfidenzen zu. Es gibt zwei verschiedene Arten von Heuristiken. Unabhängige Heuristiken berechnen die Konfidenzen, ohne die Ergebnisse anderer Heuristiken zu berücksichtigen. Ein Beispiel hierfür ist die *PackageHeuristic*, die die Paketstruktur im Quelltextmodell mit der Komponentenaufteilung im Architekturmodell vergleicht. Abhängige Heuristiken nutzen schon berechnete Konfidenzen für ihre eigenen Berechnungen. Die *HintInheritanceHeuristic* zum Beispiel nutzt schon gefundene Hinweise, um sie entlang der *extends*- und *implements*-Beziehungen zu vererben.

Die zwei konkreten Heuristiken *PackageHeuristic* und *HintInheritanceHeuristic* sind in Abbildung 5.8 nur beispielhaft enthalten. Durch Unterklassenbildung können beliebig viele andere Heuristiken implementiert werden. Eine Übersicht über die in dieser Arbeit verwendeten Heuristiken und Details zu ihrer Funktionsweise findet sich in Unterabschnitt 6.2.1.

Um die Ergebnisse von verschiedenen Heuristiken zu kombinieren, werden Aggregationsmethoden verwendet. Alle Aggregationsmethoden erben von der abstrakten Klasse *Aggregation*. Aggregationsmethoden ordnen genau wie Heuristiken jedem Endpunkttupel aus einem Endpunkttupel-Repository genau eine Konfidenz zu. Ein *ConfidenceAggregator* betrachtet für jedes Endpunkttupel einzeln die für es berechneten Konfidenzen und aggregiert sie. Die Aggregationsmethode *Maximum* zum Beispiel weist einem bestimmten Endpunkttupel die höchste Konfidenz zu, die von den zu kombinierenden Heuristiken für dieses Endpunkttupel berechnet wurde. Aggregationsmethoden können auch wieder die Ergebnisse andere Aggregationen kombinieren. Im Gegensatz zu Heuristiken hängt das Er-

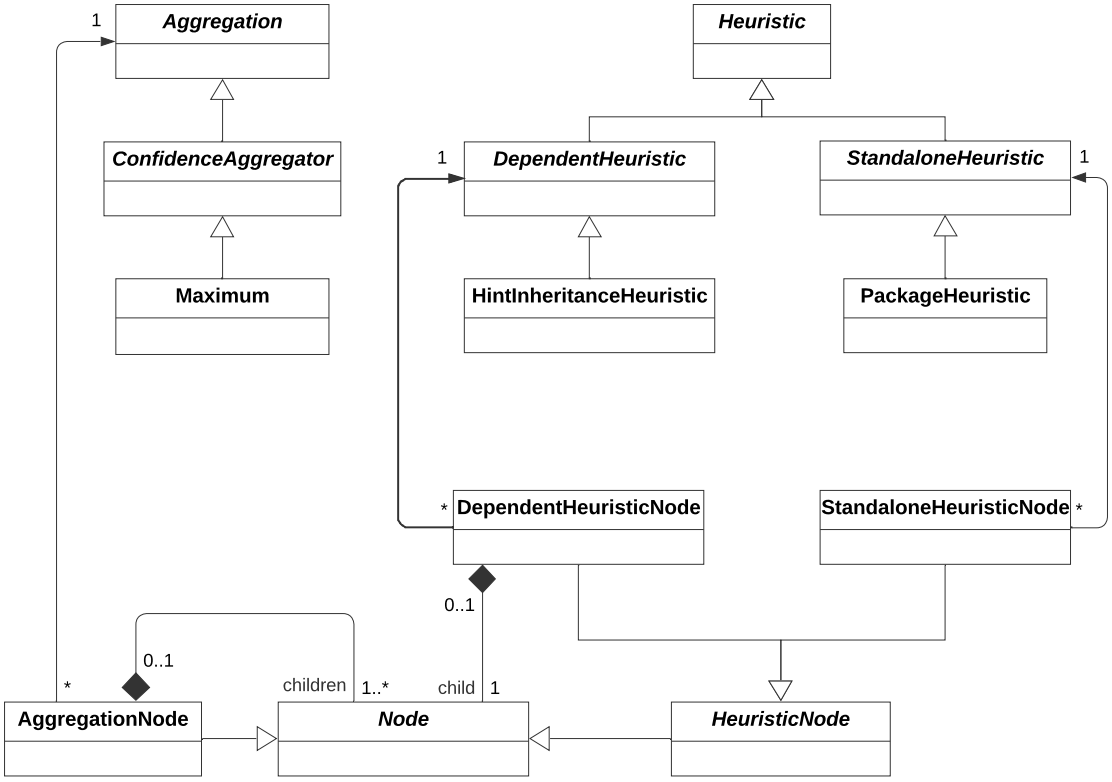

Abbildung 5.8.: Klassendiagramm des Berechnungsbaums

gebnis einer Aggregation nicht von den konkreten Architektur- und Quelltextmodellen ab, sondern nur von den Ergebnissen der zu aggregierenden Heuristiken oder Aggregationen.

Der *ConfidenceAggregator* und ihre Unterklasse *Maximum* sind in Abbildung 5.8 nur beispielhaft enthalten. Durch Unterklassenbildung können beliebig viele Aggregationsmethoden implementiert werden. Diese Aggregationsmethoden können wie *Maximum* von *ConfidenceAggregator* erben oder auch eine andere Art der Aggregation verwenden, die nicht jedes Endpunkttupel einzeln betrachtet. Die Funktionsweise der im Rahmen dieser Arbeit implementierten Aggregationsmethoden wird in Unterabschnitt 6.2.2 näher erläutert.

Ein Berechnungsbaum besteht aus Knoten mit ihren Kind-Beziehungen. Er hat genau einen Wurzelknoten. Dabei kann ein Knoten im Berechnungsbaum die Ergebnisse seiner Kindknoten für seine eigene Berechnung nutzen. Für die Klassen *StandaloneHeuristic*, *DependentHeuristic* und *Aggregation* gibt es jeweils eine entsprechende Knoten-Klasse. Ein *StandaloneHeuristicNode* hat dabei genau eine unabhängige Heuristik, ein *DependentHeuristicNode* genau eine abhängige Heuristik und ein *AggregationNode* genau eine Aggregation als Berechnungsmethode. Eine Heuristik oder Aggregation kann in einem Berechnungsbaum durch entsprechende Knoten mehrmals vorkommen. Ein Berechnungsbaum gibt eine konkrete Anordnung von Heuristiken und Aggregationen an.

In einem Berechnungsbaum müssen die Kinder eines Knotens vor diesem berechnet werden, da das Ergebnis eines Knotens von denen seiner Kinder abhängt. Ein Knoten einer unabhängigen Heuristik hat keine Kinder, da unabhängige Heuristiken die Ergebnisse anderer Heuristiken nicht berücksichtigen und alleinstehend ausgeführt werden können. Somit stellt jeder *StandaloneHeuristicNode* ein Blatt des Berechnungsbaumes dar. Ein Knoten einer abhängigen Heuristik hat genau einen Kindknoten. Die abhängige Heuristik nutzt die berechneten Konfidenzen des Kindes für ihre eigenen Berechnungen. Der Berechnungsbaum legt somit fest, wovon eine abhängige Heuristik konkret abhängt. Der Kindknoten kann dabei jede beliebige Art von Knoten sein und hat, wenn es kein *StandaloneHeuristicNode* ist, auch selbst wieder Kinder. Ein Aggregationsknoten hat mindestens einen Kindknoten. Er aggregiert die berechneten Konfidenzen seiner Kinder entsprechend seiner Aggregationsmethode. Die Kindknoten können wieder jede beliebige Art von Knoten sein. Das heißt, ein Aggregationsknoten kann jede beliebige Kombination aus Knoten von unabhängigen Heuristiken, abhängigen Heuristiken und Aggregationen kombinieren. Das Ergebnis eines Aggregationsknotens hängt nur von den Ergebnissen der Kindknoten ab. Ein Knoten einer unabhängigen Heuristik hingegen hängt ausschließlich von den konkreten Architektur- und Quelltextmodellen ab, und ein Knoten einer abhängigen Heuristik hängt sowohl von den konkreten Modellen als auch von den Ergebnissen seines Kindknotens ab.

Durch diese Aufteilung in Heuristiken bzw. Aggregationen und entsprechende Knotenklassen sind die Heuristiken und Aggregationen unabhängig von ihrer konkreten Anordnung in Berechnungsbäumen und können in allen möglichen Strukturen verwendet werden. Die Anordnung, die in dieser Arbeit für die Berechnung der Nachverfolgbarkeitsverbindungen verwendet wird, wird in Unterabschnitt 6.2.3 erläutert.

6. Implementierung

In diesem Kapitel wird die Implementierung des Ansatzes beschrieben. Dabei wird jeweils in einem eigenen Abschnitt auf den Modellgenerierungsschritt und den Trace-Link-Generierungsschritt eingegangen. Zuerst wird in Abschnitt 6.1 auf die Umsetzung der verschiedenen Extrahierer eingegangen, die Architektur- und Quelltextmodelle generieren. In Abschnitt 6.2 wird dann erläutert, wie die Generierung der Nachverfolgbarkeitsverbindungen zwischen den Elementen der extrahierten Modelle genau umgesetzt wurde. Als Programmiersprachen werden Java sowie für die Evaluation Python verwendet. Die in Abschnitt 5.1 beschriebenen Module *models*, *modelgeneration*, *tracelinkgeneration* und *evaluation* werden in entsprechenden Maven-Modulen implementiert.

6.1. Modellgenerierung

In diesem Abschnitt werden die implementierten Extrahierer beschrieben, die die Architektur- und Quelltextmodelle generieren. In dieser Arbeit werden bei der Modellgenerierung die Architektur-Metamodelle PCM und UML sowie die Programmiersprachen Java und Shell unterstützt. Für jedes der Architektur-Metamodelle und jede der Programmiersprachen werden Extrahierer implementiert. In Unterabschnitt 6.1.1 wird die Implementierung der Architektur-Extrahierer erläutert. In Unterabschnitt 6.1.2 wird der Java-Extrahierer und in Unterabschnitt 6.1.3 der Shell-Extrahierer beschrieben.

6.1.1. Architektur-Extrahierer

Für die Extraktion von AMTL-Instanzen aus existierenden PCM- und UML-Modellen werden ein PCM- und ein UML-Extrahierer benötigt. Die Extrahierer werden im *pcm* bzw. *uml*-Paket implementiert. Im Rahmen des SWATTR-Projekts [19] existieren schon Parser für PCM und UML, die ein PCM- bzw. UML-Modell aus XML-Dateien lesen können. Diese Parser werden in leicht erweiterter Form verwendet. Die Datenstruktur, die der PCM bzw. UML-Parser ausgibt, wird vom PCM bzw. UML-Extrahierer in eine AMTL-Instanz überführt.

6.1.2. Java-Extrahierer

Für die Unterstützung von Java bei der Modellgenerierung wird ein entsprechender Extrahierer benötigt. Der Java-Extrahierer nutzt die Eclipse Java Development Tools (JDT), um abstrakte Syntaxbäume aus Java-Quelltext zu generieren. Dafür bieten die JDT einen entsprechenden Parser. Die JDT werden gut gepflegt und unterstützen so zum Beispiel im Gegensatz zu JaMoPP [15] oder MoDisco [7] auch neuere Java-Versionen.

Java-Element	CMTL-Element
Paket	Package
Kompilierungseinheit	CompilationUnit
Klasse	ClassUnit
Enum	ClassUnit
Record	ClassUnit
Schnittstelle	InterfaceUnit
Methode	ControlElement
Erben von einer Klasse	extends-Beziehung
Erweiterung einer Schnittstelle	extends-Beziehung
Implementierung einer Schnittstelle	implements-Beziehung

Tabelle 6.1.: Beziehungen zwischen Java-Elementen und CMTL-Elementen

Jede Kompilierungseinheit, also jede Java-Quelldatei, wird dem Parser einzeln übergeben, um einen entsprechenden AST zu generieren. Aus den generierten ASTs wird dann eine CMTL-Instanz erzeugt. Um Beziehungen wie die *extends*-Beziehung für das Quelltextmodell generieren zu können, werden mit dem JDT-Parser auch Bindings aufgelöst. Die Bindings ziehen Verbindungen zwischen verschiedenen Programmteilen. So ist es zum Beispiel möglich, von `class Base extends Superclass` aus herauszufinden, welche `Superclass` im Programm als Oberklasse von `Base` gemeint ist. Das funktioniert auch, wenn es mehrere Klassen mit dem Namen `Superclass` gibt.

Die Beziehungen zwischen Java-Elementen und CMTL-Elementen sind in Tabelle 6.1 aufgelistet. Neben Klassen werden auch Enums und Records durch *ClassUnits* repräsentiert. Das wurde so vom KDM übernommen [1, S. 109]. Sowohl die Vererbungs-Beziehungen zwischen Klassen als auch die Erweiterungs-Beziehungen zwischen Schnittstellen werden durch *extends*-Beziehungen repräsentiert.

In Abbildung 6.1 ist für einen kurzen beispielhaften Java-Quelltextausschnitt das entsprechende Quelltextmodell als UML-Objektdiagramm dargestellt. Für das *edu.kit.kastel*-Paket gibt es im Quelltextmodell drei Paket-Objekte. Das *kastel*-Paket-Objekt enthält ein Kompilierungseinheit-Objekt. Das Kompilierungseinheit-Objekt hat den Dateinamen „Person“ als Namen. Zudem hat es den relativen Pfad der Datei innerhalb des Projektes als Pfad und „Java“ als Programmiersprache. Andere Kompilierungseinheiten im selben Java-Paket würden genauso im *kastel*-Paket-Objekt enthalten sein. Das Kompilierungseinheit-Objekt enthält ein Klassen-Objekt, das die Java-Klasse *Person* repräsentiert. Prinzipiell können auch mehrere Klassen, Enums und Records in derselben Kompilierungseinheit enthalten sein. Die Java-Klasse implementiert die `jump`-Methode. Die Methode wird im Quelltextmodell als Kontrollelement repräsentiert. Der Methodeninhalt, also zum Beispiel auszuführende Befehle, ist nicht im Quelltextmodell enthalten.

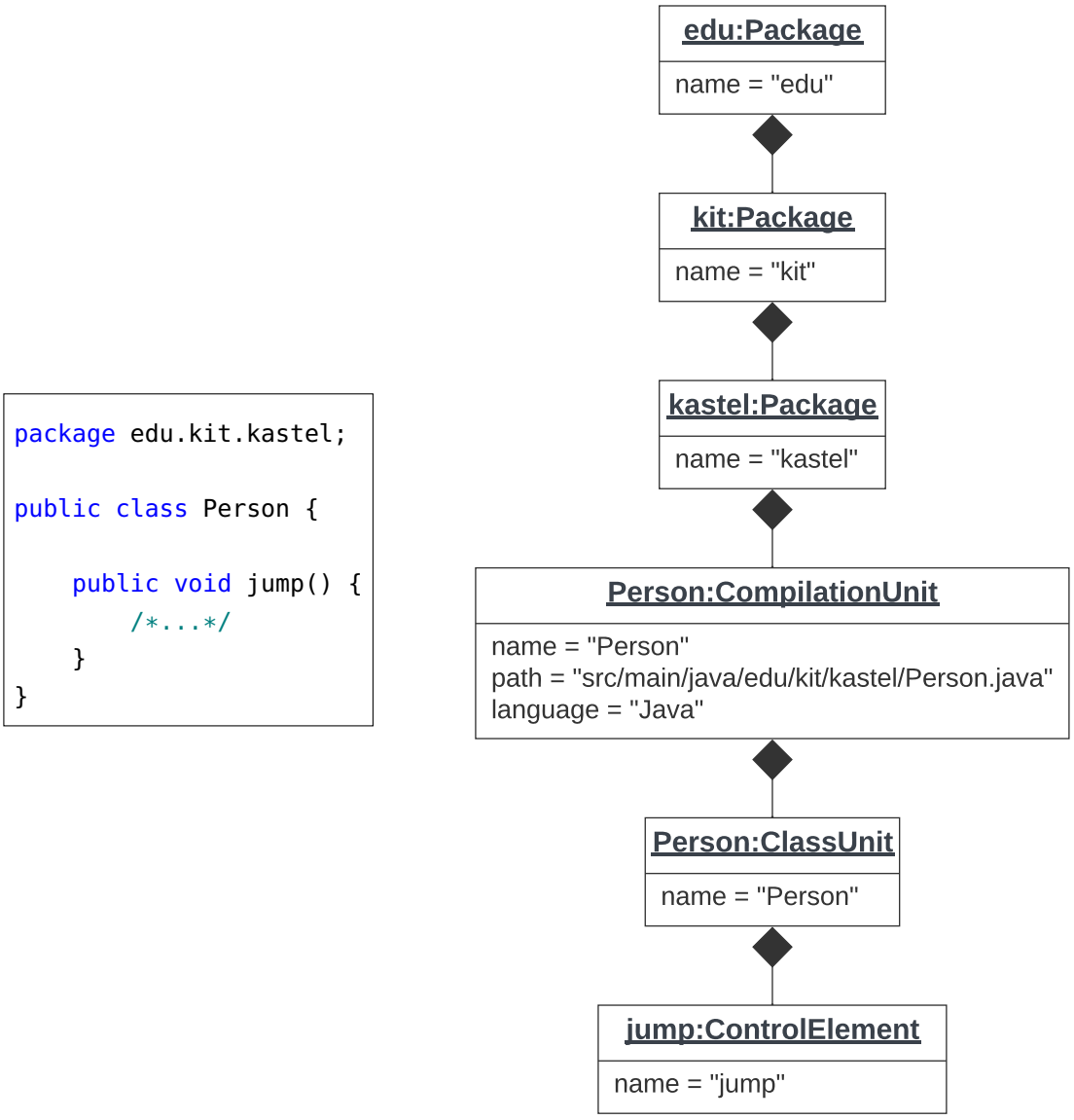

Abbildung 6.1.: Beispielhafter Java-Quelltext und UML-Objektdiagramm des entsprechenden Quelltextmodells

Shell-Element	CMTL-Element
Shellskript	CompilationUnit
Funktion	ControlElement
Shellskript bindet anderes Shellskript ein (Source-Befehl)	CompilationUnits sind in selber CodeAssembly

Tabelle 6.2.: Beziehungen zwischen Shell-Elementen und CMTL-Elementen

6.1.3. Shell-Extrahierer

Für die Unterstützung von Shell bei der Modellgenerierung wird ein entsprechender Extrahierer implementiert. Die Beziehungen zwischen Shell-Elementen und CMTL-Elementen sind in Tabelle 6.2 aufgelistet. Für jedes Shellskript wird eine Kompilierungseinheit für das Quelltextmodell erzeugt. Der Name der Kompilierungseinheit entspricht dabei dem Namen der Shellskript-Datei. Funktionen im Shellskript werden durch Kontrollelemente repräsentiert. Bindet ein Shellskript mit dem Source-Befehl ein anderes Shellskript ein, sind die beiden entsprechenden Kompilierungseinheiten in derselben *CodeAssembly* enthalten. Andere CMTL-Elemente wie Pakete, Klassen oder Schnittstellen werden für Shell nicht verwendet.

6.2. Generierung der Nachverfolgbarkeitsverbindungen

In diesem Abschnitt wird darauf eingegangen, wie die Generierung der Nachverfolgbarkeitsverbindungen zwischen den Elementen der extrahierten Modelle genau umgesetzt wurde. In Unterabschnitt 6.2.1 werden die verwendeten Heuristiken mit ihrer Funktionsweise vorgestellt. In Unterabschnitt 6.2.2 werden dann verschiedene Aggregationsmethoden erläutert, die die Ergebnisse der Heuristiken kombinieren können. Zum Schluss wird in Unterabschnitt 6.2.3 auf den in dieser Arbeit verwendeten Berechnungsbaum eingegangen, also wie die Heuristiken und Aggregationen genau organisiert sind.

6.2.1. Heuristiken

Das UML-Klassendiagramm für die in dieser Arbeit verwendeten Heuristiken ist in Abbildung 6.2 dargestellt. Zur Übersicht wird in Tabelle 6.3 die Funktionsweise der einzelnen Heuristiken kurz beschrieben. Als Blätter des Berechnungsbaums dienen die unabhängigen Heuristiken. Diese nutzen Pakete, Verzeichnisse, Methodensignaturen und Namen als Hinweise. Die Paket- und die Pfad-Heuristik betrachten Architekturkomponenten. Die Methoden-Heuristik betrachtet Architekturschnittstellen und die Namen-Heuristik betrachtet beides. Die abhängigen Heuristiken berechnen nur für Komponenten Konfidenzen. Die Schnittstellen-Bereitstellung-Heuristik nutzt dabei auch Hinweise für Architekturschnittstellen.

Die abhängigen Heuristiken funktionieren so, dass sie nie Hinweise ihres Kindknotens übernehmen. Das erleichtert ihre Evaluation, da klar ist, dass alle durch sie gefundenen

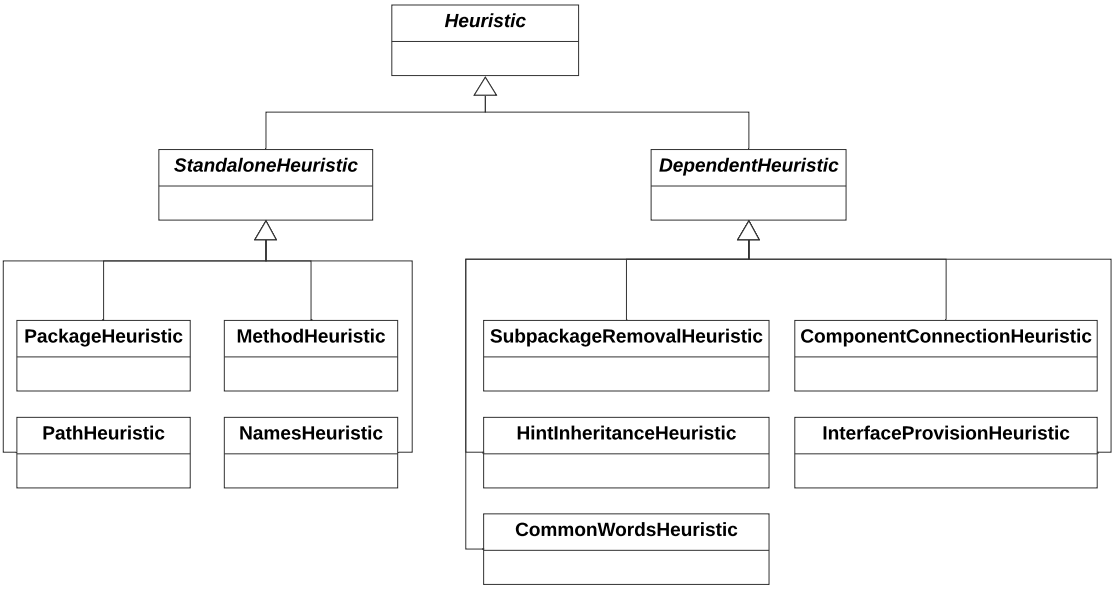

Abbildung 6.2.: UML-Klassendiagramm der verwendeten Heuristiken

Name der Heuristik	**Funktionsweise der Heuristik**
Paket-Heuristik (*PackageHeuristic*)	Vergleicht Pakete mit Komponenten
Pfad-Heuristik (*PathHeuristic*)	Vergleicht Verzeichnisse mit Komponenten
Methoden-Heuristik (*MethodHeuristic*)	Vergleicht Kontrollelemente mit Signaturen
Namen-Heuristik (*NamesHeuristic*)	Vergleicht Namen von Architekturelementen mit Namen von Kompilierungseinheiten und Datentypen
Unterpaket-Entfernung-Heuristik (*SubpackageRemovalHeuristic*)	Entfernt uneindeutig zugeordnete Unterpakete
Hinweis-Vererbung-Heuristik (*HintInheritanceHeuristic*)	Gibt Hinweise entlang der *extends*- und *implements*-Beziehungen weiter
Allgemeine-Wörter-Heuristik (*CommonWordsHeuristic*)	Prüft, ob Namen sich nur um festgelegte, allgemeine Wörter unterscheiden
Komponenten-Verbindungen-Heuristik (*ComponentConnectionHeuristic*)	Nutzt Verbindungen zwischen Komponenten, um Uneindeutigkeiten zu beseitigen
Schnittstellen-Bereitstellung-Heuristik (*InterfaceProvisionHeuristic*)	Prüft, ob sich die *provides*-Beziehung aus der Architektur im Quelltext wiederfindet

Tabelle 6.3.: Übersicht über die Funktionsweisen der Heuristiken

Hinweise von ihnen selbst stammen und nicht einfach vom Kindknoten übernommen wurden. Als Folge davon müssen im Berechnungsbaum das Ergebnis einer abhängigen Heuristik und das Ergebnis seines Kindknotens aggregiert werden.

6.2.1.1. Pakete als Hinweise

Die Paket-Heuristik überprüft die Übereinstimmung zwischen den Namen von Architekturkomponenten und Paketen im Quelltextmodell. Sie nutzt also das Paket einer Kompilierungseinheit als Hinweis auf Nachverfolgbarkeitsverbindungen. Pakete im CMTL stellen logische Sammlungen von Quelltextelementen dar und strukturieren analog zu Komponenten das System. Damit bietet es sich an, für jede Komponente zu prüfen, ob ihr Pakete zugeordnet werden können. Florean et al. [12] verwenden auf maschinellem Lernen basierende Textklassifizierer, um Quelldateien Architekturmodulen zuzuordnen. Aus ihren Experimenten schließen sie, dass Paketdeklarationen besonders wichtig für die Zuordnung seien. Das gelte vor allem, wenn sich die Struktur der Architektur und der Pakete stark ähneln.

Im TLM sind keine Nachverfolgbarkeitsverbindungen von Paketen zu Architektur-Endpunkten vorgesehen. Nur Kompilierungseinheiten sind Quelltext-Endpunkte. Eine Zuordnung von einem Paket zu einer Komponente entspricht somit letztendlich Nachverfolgbarkeitsverbindungen zwischen allen direkt und indirekt im Paket enthaltenen Quelltext-Endpunkten und der Komponente.

Die Heuristik prüft bei Betrachtung einer Kompilierungseinheit und einer Komponente, wie sehr die Namen des Paketes der Kompilierungseinheit und all seiner Elternpakete sowie der Name der Komponente übereinstimmen. Dazu wird zuerst der Name der Komponente in seine einzelnen Wörter aufgeteilt. Ist mindestens eines der Wörter in den Paketnamen enthalten, wird das als Hinweis auf eine Nachverfolgbarkeitsverbindung gewertet. Florean et al. [12] und Olsson et al. [22] teilen in ihren auf maschinellem Lernen basierenden Ansätzen ebenfalls Namen in ihre einzelnen Wörter auf. Die Groß- und Kleinschreibung wird beim Namensvergleich ignoriert. Wortgrenzen werden jedoch beachtet. So wird zum Beispiel die Komponente *UI* nicht dem Paket *builder* zugeordnet, obwohl dessen Name *ui* enthält.

Wurde zwischen einer Komponente und einer Kompilierungseinheit ein Hinweis auf eine Nachverfolgbarkeitsverbindung gefunden, muss dieser Hinweis noch mit einem konkreten Konfidenzwert bewertet werden. Bei der Paket-Heuristik wird hierfür die Anzahl der Wörter im Komponentennamen, die in den Paketnamen enthalten sind, durch die Anzahl der Wörter im Komponentennamen insgesamt geteilt. Der Konfidenzwert beschreibt somit den Anteil des Architekturnamens, der durch Paketnamen abgedeckt wird.

Anstatt eine genaue Übereinstimmung von Wörtern zu verlangen, kann die Heuristik auch andere weniger strenge Vergleichsmethoden verwenden. Damit kann unter Umständen eine höhere Ausbeute erzielt werden, da vielleicht mehr korrekte Hinweise auf Nachverfolgbarkeitsverbindungen gefunden werden. Allerdings besteht die Gefahr, dass die Präzision der generierten Nachverfolgbarkeitsverbindungen sinkt. Schließlich könnten auch mehr falsche Hinweise gefunden werden. Eine mögliche Vergleichsmethode ist es, die Wörter erst zu lemmatisieren und dann auf eine genaue Übereinstimmung zu untersuchen. So kann zum Beispiel das Paket *servlets* der Komponente *Servlet* zugeordnet werden, da die

Wörter dasselbe Lemma haben. Da bei der Lemmatisierung unterschiedliche Wortarten im Allgemeinen nicht auf dasselbe Lemma zurückgeführt werden, können Wörter, die sich in ihrer Wortart unterscheiden, so nicht zuverlässig erkannt werden. Zum Beispiel würde das Paket *reencoder* nicht der Komponente *Reencoding* zugeordnet werden. Um auch unterschiedliche Wortarten auf dieselbe Form zurückzuführen, kann zum Beispiel die Stammformreduktion verwendet werden. Dabei werden Wörter auf ihren jeweiligen Wortstamm zurückgeführt. Bei der Stammformreduktion werden im Gegensatz zur Lemmatisierung manche Besonderheiten wie zum Beispiel bei irregulären Nomen nicht berücksichtigt. Beim Wortstamm handelt im Gegensatz zum Lemma nicht unbedingt um ein sinnvolles Wort. Da das Resultat von der Heuristik aber nur mit anderen Wortstämmen verglichen und sonst nicht weiter verarbeitet wird, stellt das hier keinen Nachteil dar. Florean et al. [12] sowie Olsson et al. [22] verwenden in ihren Ansätzen ebenfalls die Stammformreduktion zur Vorverarbeitung.

Es kann vorkommen, dass die Paket-Heuristik eine Kompilierungseinheit mehreren Komponenten zuordnet. Liegen zum Beispiel das Paket *project.servlet.preferences* und die Architekturkomponenten *Servlet* und *Preferences* vor, so wird das *servlet*-Paket der *Servlet*-Komponente zugeordnet. Das *preferences*-Paket wird hingegen der *Preferences*-Komponente zugeordnet. Also erhalten alle Quelltext-Endpunkte im *preferences*-Paket Hinweise auf Nachverfolgbarkeitsverbindungen sowohl zur *Servlet*- als auch zur *Preferences*-Komponente. Dabei kann es sein, dass die *Preferences*-Komponente eigentlich woanders implementiert ist, zum Beispiel im *project.preferences*-Paket. Die Unterpaket-Entfernung-Heuristik kann genutzt werden, um in Fällen wie dem beschriebenen nur den Hinweis des äußeren Pakets zu beachten. Dadurch werden die Kompilierungseinheiten im Unterpaket nur noch einer Komponente zugeordnet.

6.2.1.2. Verzeichnisse als Hinweise

Der Dateipfad einer Kompilierungseinheit kann als Hinweis auf Nachverfolgbarkeitsverbindungen genutzt werden. Durch die Verzeichnisstruktur des Systems werden logische Sammlungen von Dateien definiert. Verzeichnisse strukturieren analog zu Komponenten das System. Damit bietet es sich an, für jede Komponente zu prüfen, ob ihr Verzeichnisse zugeordnet werden können. Die Pfad-Heuristik überprüft die Übereinstimmung zwischen den Namen von Architekturkomponenten und Verzeichnissen.

Der Dateiname einer Kompilierungseinheit wird von der Pfad-Heuristik nicht berücksichtigt. Er wird stattdessen in der Namen-Heuristik genutzt, die in Unterunterabschnitt 6.2.1.3 beschrieben wird. Auch die Dateiendung der Kompilierungseinheit wird ignoriert.

Die Pfad-Heuristik nutzt dieselbe Art und Weise, Wörter zu vergleichen und darauf basierend Konfidenzwerte festzulegen, wie die Paket-Heuristik. Stimmt also mindestens ein Wort im Komponentennamen mit einem Wort im Pfad überein, dient das als Hinweis auf eine Nachverfolgbarkeitsverbindung. Der Anteil der Wörter des Komponentennamen, die im Pfad enthalten sind, bildet dann, wie in Unterunterabschnitt 6.2.1.1 beschrieben, den Konfidenzwert.

Nach manchen Konventionen werden Pakete in der Verzeichnisstruktur widergespiegelt. Da Paketnamen schon von der Paket-Heuristik genutzt werden, werden für in Paketen ent-

haltene Kompilierungseinheiten die entsprechenden Verzeichnisse von der Pfad-Heuristik herausgefiltert. Für eine Kompilierungseinheit mit dem Pfad *ms-database/src/main/java/mediastore/persistence/AudioFile.java*, die im Paket *mediastore.persistence* enthalten ist, nutzt die Pfad-Heuristik also nur den Pfadbeginn *ms-database/src/main/java*. Für eine Architekturkomponente *Database* kann dieser Pfadbeginn als Hinweis dienen, der durch die Paket-Heuristik nicht erkannt wird.

6.2.1.3. Namen in Kompilierungseinheiten als Hinweise

Auch die Namen einer Kompilierungseinheit und der in ihr enthaltenen Datentypen können als Hinweise für Nachverfolgbarkeitsverbindungen genutzt werden. Das wird zum Beispiel in der Namen-Heuristik umgesetzt.

Diese Heuristik eignet sich nicht nur für Komponenten, sondern auch für Architekturschnittstellen. Der Grund dafür liegt darin, dass sie jede Kompilierungseinheit einzeln betrachtet. Im Gegensatz dazu betrachtet zum Beispiel die Paket-Heuristik immer alle Inhalte eines größeren Moduls.

Die Namen-Heuristik betrachtet für eine Kompilierungseinheit ihren Namen, sowohl die Namen aller direkt oder indirekt in ihr definierten Datentypen. Es werden also zum Beispiel alle enthaltenen Klassen und Quelltextschnittstellen miteinbezogen. Dabei kann der Name der Kompilierungseinheit zum Beispiel auch mit dem Namen einer enthaltenen Klasse übereinstimmen. Die Namen-Heuristik überprüft, ob irgendeiner der betrachteten Namen mit dem Namen des Architektur-Endpunktes übereinstimmt. Ist dies der Fall, wird dies als ein Hinweis auf eine Nachverfolgbarkeitsverbindung gewertet.

Die Heuristik verwendet dabei als Übereinstimmungskriterium, dass der Name des Architektur-Endpunktes vollständig im Namen des *CodeItems* enthalten ist. Wie in Unterunterabschnitt 6.2.1.1 beschrieben, werden dabei Wortgrenzen beachtet. Somit stimmen *DB* und *LoadBalancer* nicht überein, da *DB* nur enthalten ist, wenn Wortgrenzen ignoriert werden. Bei Architekturschnittstellen wie *IUser*, deren erstes Wort ein „I“ ist, müssen nur die restlichen Wörter im Quelltextelement-Namen enthaltenen sein. Die Heuristik kann auch, wie in Unterunterabschnitt 6.2.1.1 für Pakete beschrieben, die Lemmatisierung oder die Stammformreduktion als Vorverarbeitungsmethoden verwenden.

Hat die Heuristik einen Hinweis gefunden, teilt sie für jedes betrachtete Quelltextelement die Wortanzahl des Namens des Architektur-Endpunktes durch die Wortanzahl des Namens des Quelltextelements. Für den Konfidenzwert bildet die Heuristik dann das Maximum dieser berechneten Werte. Dadurch erhalten für ein bestimmtes Quelltextelement Architektur-Endpunkte mit längerem vollständig enthaltenen Namen größere Werte. Zudem fließen Wörter, die der Quelltextelement-Name im Vergleich zum Namen des Architektur-Endpunktes zusätzlich hat, negativ in den Wert ein. In Tabelle 6.4 sind ein paar Beispielwerte aufgelistet.

Ein Hinweis auf eine Nachverfolgbarkeitsverbindung zwischen einer Architekturschnittstelle und einer *InterfaceUnit* soll besonders gut bewertet werden. Daher wird solchen durch die Namen-Heuristik gefundenen Hinweisen immer ein Konfidenzwert von eins zugewiesen.

In Vitruvius [21] wird zur Konsistenzhaltung definiert, dass es für eine Komponente eine entsprechende Klasse gibt, die denselben Namen mit dem Zusatz *Impl* hat. Dieses Konzept,

Name des Architektur-Endpunkts	Name des Quelltextelements	Wert
DB	LoadBalancer	kein Wert
DB	Db	1.0
DB	UserDbAdapter	1/3
UserDB	Db	kein Wert
IUser	UserDbAdapter	1/3
UserDBAdapter	UserDbAdapter	1.0

Tabelle 6.4.: Konfidenzwerte für einzelne Paare von Architektur-Endpunkt- und Quelltextelement-Namen

dass sich Namen nur durch ein bestimmtes hinzugefügtes Wort unterscheiden, kann auf mehrere allgemeine Begriffe erweitert werden. Anslow et al. [2] haben in einer Analyse von Java-Klassen-Namen festgestellt, dass die Begriffe *Test*, *Action*, *Impl*, *Factory*, *Exception* und *Data* am häufigsten auftreten. Diese Wörter werden von der Allgemeine-Wörter-Heuristik verwendet. Die Heuristik prüft wie die Namen-Heuristik, ob ein Komponenten-Name vollständig im Quelltextelement-Namen enthalten ist. Zusätzlich prüft die Heuristik, ob alle zusätzlichen Wörter im Quelltextelement-Namen einem der festgelegten allgemeinen Begriffe entsprechen. Ist beides erfüllt, ordnet die Heuristik die Endpunkte einander mit einem Konfidenzwert von eins zu.

Diese Art des Namensvergleichs lässt sich auch nutzen, um zwei Quelltextelement-Namen zu vergleichen. Wurde zum Beispiel die Klasse *UserDbAdapter* einer bestimmten Komponente zugeordnet, können so auch die Klassen *UserDbAdapterException* und *UserDbAdapterTest* der Komponente zugeordnet werden. Die Allgemeine-Wörter-Heuristik wendet die definierte Art des Namensvergleichs also nicht nur auf die Namen von Architektur-Endpunkt und Quelltextelement an, sondern vergleicht auch verschiedene Quelltextelemente. Sie ist eine abhängige Heuristik, die das Ergebnis des Kindknotens nutzt, um durch die Namensvergleiche neue Hinweise zu finden. Ordnet der Kindknoten also zum Beispiel die Klasse *UserDbAdapter* einer bestimmten Komponente zu, findet die Heuristik wie beschrieben neue Hinweise. Den Konfidenzwert übernimmt die Heuristik vom Ergebnis des Kindknotens. Schließt die Heuristik von mehreren Quelltextelementen auf einen Hinweis, so übernimmt sie das Maximum der Konfidenzwerte. Um die Präzision der Heuristik zu erhöhen, werden nur Quelltextelemente berücksichtigt, die nicht in unterschiedlichen Modulen liegen.

Es ist möglich, dass die festgelegten allgemeinen Begriffe in den Architekturkomponenten verwendet werden und damit zur Unterscheidung verschiedener Komponenten benötigt werden. Zum Beispiel kann ein Modell die Komponenten *Action* und *ActionFactory* beinhalten. Um diesem Umstand Rechnung zu tragen, wird für jedes Architekturmodell geprüft, ob einer der festgelegten allgemeinen Begriffe in den Namen verwendet wird. Ist das der Fall, wird dieses Wort für das Modell aus den allgemeinen Begriffen entfernt.

Die Allgemeine-Wörter-Heuristik betrachtet im Gegensatz zur Namen-Heuristik nur Komponenten und nicht Architekturschnittstellen. Die allgemeinen Wörter beziehen sich eher auf die Implementierung der Funktionalität. Am deutlichsten wird das beim Begriff *Impl*. Im Ansatz dieser Arbeit werden Architekturschnittstellen jedoch nur die Quelltext-Endpunkte zugeordnet, die dieselbe Funktionalität wie die Architekturschnittstelle spezifizieren. Eine konkrete Implementierung der Funktionalität im Quelltext wird hingegen der entsprechenden Architekturkomponente zugeordnet. Dabei können verschiedene Implementierungen der Funktionalität verschiedenen die Architekturschnittstelle anbietenden Komponenten zugeordnet werden.

6.2.1.4. Vererbung von Hinweisen

Die im Quelltextmodell enthaltenen *extends*- und *implements*-Beziehungen können verwendet werden, um von anderen Heuristiken gefundene Hinweise auf Nachverfolgbarkeitsverbindungen zu vererben. So kann zum Beispiel eine Klasse einen Hinweis erben, der ihre Oberklasse einer Architekturkomponente zuordnet. Die Klasse erhält damit den Hinweis, dass sie genau wie ihre Oberklasse zu dieser Komponente gehört. Hier wird ausgenutzt, dass eine Klasse im Allgemeinen eng zur Oberklasse gekoppelt ist. Das nutzt die Heuristik als Hinweis, dass beide Klassen zur selben Architekturkomponente gehören. Ebenso können zum Beispiel Quelltextschnittstellen andere Quelltextschnittstellen erweitern. Auch die Implementierung einer Schnittstelle wird genutzt, um die Hinweise der Schnittstelle an die sie implementierenden Datentypen zu vererben.

Die Hinweis-Vererbung-Heuristik gibt also Hinweise auf Nachverfolgbarkeitsverbindungen entlang der *extends*- und *implements*-Beziehungen weiter. Sind ein Datentyp und ein durch ihn erweiterter bzw. implementierter Datentyp in derselben Kompilierungseinheit enthalten, müssen Hinweise nicht vererbt werden. Schließlich dienen Kompilierungseinheiten als Quelltext-Endpunkte, und somit werden alle in einer Kompilierungseinheit enthaltenen Quelltextelemente denselben Architektur-Endpunkten zugeordnet. Die Hinweis-Vererbung-Heuristik ist eine abhängige Heuristik. Das heißt, ein Knoten im Berechnungsbaum mit der Hinweis-Vererbung-Heuristik hat einen Kindknoten. Die Heuristik nutzt das Ergebnis dieses Kindknotens für die Hinweise auf Nachverfolgbarkeitsverbindungen, die sie vererbt.

Die Heuristik gibt nur Hinweise auf Nachverfolgbarkeitsverbindungen zu Architekturkomponenten und nicht zu Architekturschnittstellen weiter. Architekturschnittstellen werden den Quelltext-Endpunkten zugeordnet, die dieselbe Funktionalität wie die Architekturschnittstelle spezifizieren. Die Quelltext-Endpunkte, die die Funktionalität implementieren, werden hingegen Komponenten zugeordnet. Entspricht eine Architekturschnittstelle somit zum Beispiel einer Quelltextschnittstelle, wird sie nicht den Datentypen, die die Schnittstelle implementieren, zugeordnet. Die Hinweis-Vererbung-Heuristik gibt dementsprechend gefundene Hinweise auf Nachverfolgbarkeitsverbindungen zu einer Architekturschnittstelle nicht weiter.

Gibt es für einen Datentypen schon einen Hinweis auf eine Nachverfolgbarkeitsverbindung, erbt er keine weiteren Hinweise. Die Heuristik betrachtet also nur Datentypen, für die im verwendeten Ergebnis des Kindknotens noch kein Hinweis auf Nachverfolgbarkeitsverbindungen gefunden wurde.

Um die Präzision der Heuristik weiter zu erhöhen, werden nur erweiterte und implementierte Datentypen berücksichtigt, die nicht in anderen Paketen liegen, als der Datentyp, an den Hinweise vererbt werden. Die Heuristik nutzt somit auch die Paketstruktur als Hinweis darauf, ob zwei Datentypen zur selben Architekturkomponente gehören. Es werden auch indirekte *extends*- und *implements*-Beziehungen berücksichtigt. Also erhält zum Beispiel eine Klasse auch die Hinweise der Oberklasse der Oberklasse, solange sie sich im selben Paket befindet.

Als Hinweis auf eine Nachverfolgbarkeitsverbindung nutzt die Heuristik also die Hinweise der erweiterten und implementierten Datentypen im selben Paket. Als Konfidenzwert wird derselbe Wert wie beim vererbten Hinweis genutzt. Werden für ein Endpunkttupel mehrere Hinweise vererbt, weist die Heuristik den größten der entsprechenden Konfidenzwerte zu.

6.2.1.5. Verbindungen zwischen Komponenten als Hinweise

Wird ein Quelltext-Endpunkt mehreren Komponenten zugeordnet, können Verbindungen zwischen Komponenten genutzt werden, um die zugeordneten Komponenten zu filtern. Hier werden Verbindungen von Komponenten über Architekturschnittstellen betrachtet. Damit ist gemeint, dass eine Komponente eine Schnittstelle benötigt, die von der anderen bereitgestellt wird. Die erste Komponente wird hier als „benötigend“ und die zweite als „bereitstellend“ bezeichnet.

Die Komponenten-Verbindungen-Heuristik ist eine abhängige Heuristik und nutzt das Ergebnis des Kindknotens, um die Präzision der Hinweise zu erhöhen. Die Heuristik nutzt die Verbindungen zwischen den Komponenten, um die Anzahl der verschiedenen Komponenten, denen ein Quelltext-Endpunkt im Ergebnis des Kindknotens zugeordnet wird, zu reduzieren. Sind unter den dem Quelltext-Endpunkt zugeordneten Komponenten zwei Komponenten, zwischen denen es die genannte Verbindungen gibt, wird von den zwei nur die bereitstellende Komponente beibehalten. Die Heuristik deckt damit den Fall ab, dass die Quelltextmodule der benötigenden Komponente die Quelltextmodule der bereitstellenden Komponente enthalten. Dabei spiegelt sich also die Verbindung der Komponenten in der Struktur des Quelltexts wider.

Ein Beispiel für eine Ausgangssituation einer erfolgreichen Anwendung der Komponenten-Verbindungen-Heuristik ist im UML-Objektdiagramm in Abbildung 6.3 dargestellt. Dabei besteht eine Verbindung zwischen der *Recommender*- und der *OrderBasedStrategy*-Komponente über die *Strategy*-Schnittstelle. Aus früheren Heuristiken sind drei Hinweise auf Nachverfolgbarkeitsverbindungen vorhanden. Davon sind zwei korrekt. Der Hinweis auf eine Nachverfolgbarkeitsverbindung von der *Recommender*-Komponente zur *OrderBasedStrategy*-Kompilierungseinheit ist hingegen nicht korrekt. Die inkorrekte Zuordnung könnte zum Beispiel von der Paket-Heuristik stammen, da die *OrderBasedStrategy*-Kompilierungseinheit im *recommender*-Paket enthalten ist. Mit der Komponenten-Verbindungen-Heuristik kann der inkorrekte Hinweis entfernt werden.

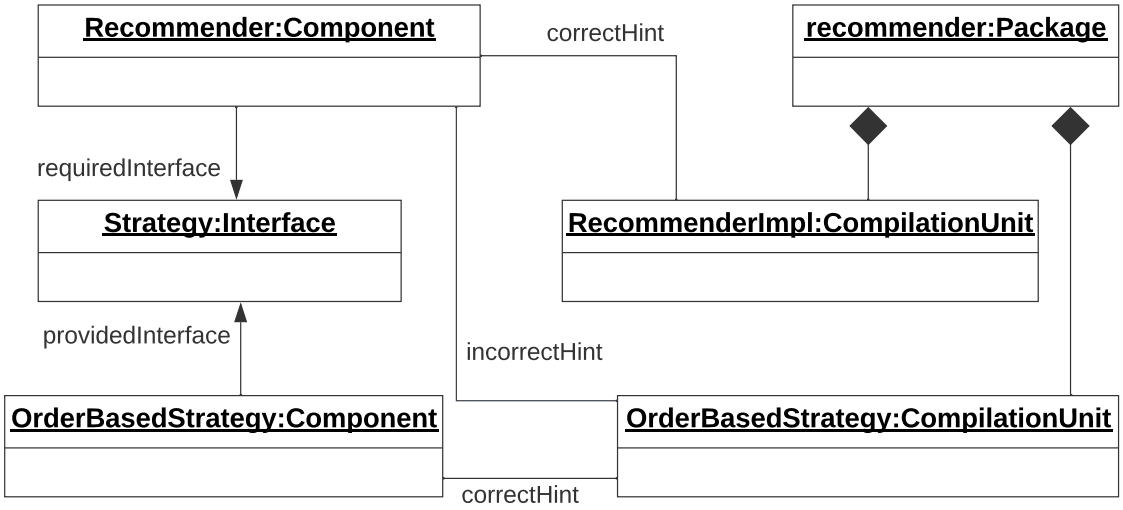

Abbildung 6.3.: UML-Objektdiagramm der Ausgangssituation einer erfolgreiche Anwendung der Komponenten-Verbindungen-Heuristik

6.2.1.6. Methoden als Hinweise

Um Nachverfolgbarkeitsverbindungen für Architekturschnittstellen zu finden, können Methodensignaturen als Hinweise genutzt werden. Wenn eine Architekturschnittstelle Signaturen enthält, kann die Übereinstimmung dieser Signaturen mit den Kontrollelementen einer Kompilierungseinheit überprüft werden. Die Methoden-Heuristik setzt das um. Methoden werden zum Beispiel auch in den Ansätzen von Florean et al. [12], Olsson et al. [22] und InMap [25] verwendet.

Die Heuristik betrachtet für eine Kompilierungseinheit alle Kontrollelemente, die direkt oder indirekt in ihr enthalten sind. Also werden zum Beispiel auch die Kontrollelemente einer inneren Klasse betrachtet. Allerdings werden Kontrollelemente eines Datentypen ignoriert, die schon in einem vom Datentypen erweiterten oder implementierten Datentypen deklariert wurden. Für jedes Kontrollelement wird also nur seine erste Deklaration betrachtet. Implementiert zum Beispiel eine Klasse eine Quelltextschnittstelle, soll eine Architekturschnittstelle mit übereinstimmenden Methoden schließlich der Quelltextschnittstelle und nicht der sie implementierenden Klasse zugeordnet werden. Analoges gilt zum Beispiel, wenn eine Klasse eine andere Klasse erweitert.

Die Methoden-Heuristik überprüft, ob irgendeine in der Architekturschnittstelle enthaltene Signatur mit einem Kontrollelement der Kompilierungseinheit übereinstimmt. Ist dies der Fall, wird es als ein Hinweis auf eine Nachverfolgbarkeitsverbindung gewertet. Zwei Methoden stimmen dabei genau dann überein, wenn sie denselben Namen haben. Eine Übereinstimmung in den Parametern oder dem Rückgabetypen ist nicht erforderlich, damit die Heuristik einen Hinweis auf eine Nachverfolgbarkeitsverbindung findet.

Für den Konfidenzwert bestimmt die Heuristik die Anzahl der Signaturen der Architekturschnittstelle, die wie oben beschrieben mit einem Kontrollelement der Kompilierungseinheit übereinstimmen. Diese Anzahl wird mit zwei multipliziert und durch die

Anzahl der Signaturen in der Architekturschnittstelle und der Kontrollelemente in der Kompilierungseinheit geteilt.

$$\text{Konfidenz} = \frac{2 \times \#\ \text{Übereinstimmungen}}{\#\ \text{Signaturen} + \#\ \text{Kontrollelemente}}$$

Durch diese Berechnung fließen Übereinstimmungen positiv in den Konfidenzwert ein. Signaturen in der Architekturschnittstelle, die keine Entsprechung in der Kompilierungseinheit haben, fließen negativ in den Wert ein. Das gleiche gilt für Kontrollelemente der Kompilierungseinheit ohne Entsprechung in der Architekturschnittstelle.

Als Beispiel kann das UML-Objektdiagramm in Abbildung 6.4 betrachtet werden. Die *IUtilOperations*-Schnittstelle enthält hier die Signaturen *toJson* und *toYaml*. Die *JsonUtils*-Kompilierungseinheit enthält das Kontrollelement *toJson*. Es gibt also eine Übereinstimmung. Für die Konfidenz ergibt sich ein Wert von 2/3.

6.2.1.7. Verbindung von Hinweisen für Komponenten und Schnittstellen

Um gefundene Hinweise für Komponenten und für Architekturschnittstellen zu verbinden, kann die *provides*-Beziehung zwischen Komponenten und Architekturschnittstellen ausgenutzt werden. Sowohl der betrachteten Komponente als auch den von ihnen bereitgestellten Schnittstellen wurden Mengen von Quelltext-Endpunkte zugeordnet. Es kann geprüft werden, ob und wo sich die *provides*-Beziehung auch zwischen den entsprechenden Mengen von Quelltext-Endpunkten wiederfindet.

Die Schnittstellen-Bereitstellung-Heuristik betrachtet für jede Komponente die einzelnen ihr zugeordneten Quelltextmodule. Für jedes Modul der Komponente wird überprüft, ob sich eine Entsprechung der *provides*-Beziehung finden lässt. Ist nur für manche der Module die Prüfung erfolgreich, wird angenommen, dass nur diese Module die Komponente tatsächlich implementieren. Für die anderen Module, für die die Prüfung fehlschlägt, wird angenommen, dass sie die Komponente nicht implementieren. Die Heuristik stellt also fest, dass für diese Module keine Nachverfolgbarkeitsverbindungen existieren sollten.

Für die Prüfung des Moduls einer Komponente werden alle Kompilierungseinheiten betrachtet, die einer von der Komponente bereitgestellten Schnittstelle zugeordnet wurden. Diese Kompilierungseinheiten werden Schnittstellen-Kompilierungseinheiten genannt. Zum einen wird geprüft, ob eine der Schnittstellen-Kompilierungseinheiten im Modul liegt. Zum anderen wird geprüft, ob ein Datentyp des Moduls einen Datentypen einer Schnittstellen-Kompilierungseinheit erweitert oder implementiert. Für die Prüfungen werden also sowohl die Modulstruktur als auch die *extends*- und *implements*-Beziehungen herangezogen. Ist mindestens eine der beiden Prüfungen erfolgreich, wurde eine Entsprechung der *provides*-Beziehung gefunden. Dann wird also angenommen, dass das Modul die Komponente tatsächlich implementiert.

Ein Beispiel für eine Ausgangssituation einer erfolgreichen Anwendung der Schnittstellen-Bereitstellung-Heuristik ist im UML-Objektdiagramm in Abbildung 6.4 dargestellt. Die *Util*-Komponente biete die *IUtilOperations*-Schnittstelle an. Von der Paket-Heuristik wurden für die *Util*-Komponente zwei Hinweise zu Paketen gefunden. Davon ist nur der Hinweis zum *data.util*-Paket korrekt. Der Hinweis zum *logic.util*-Paket ist inkorrekt.

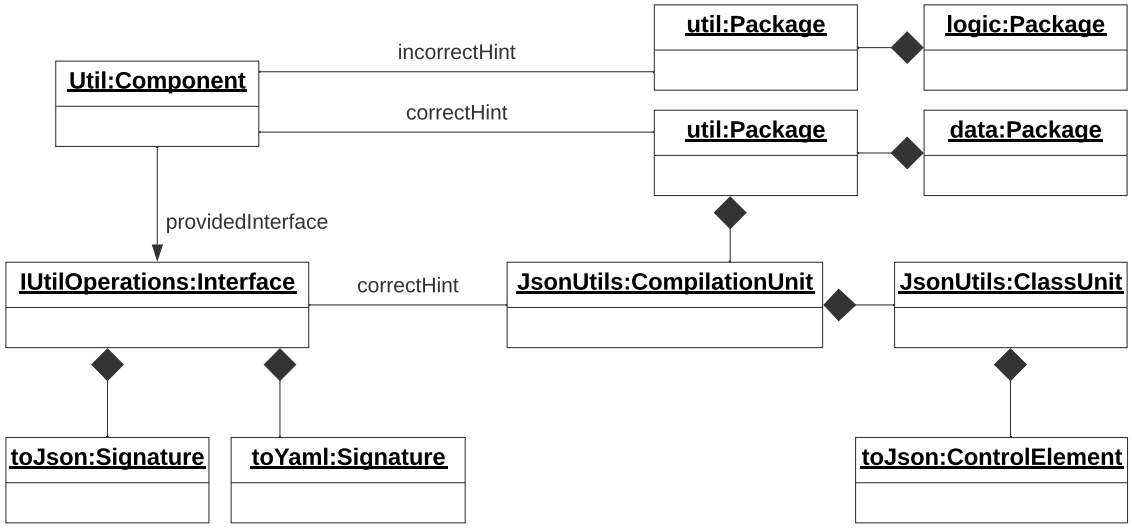

Abbildung 6.4.: UML-Objektdiagramm der Ausgangssituation einer erfolgreiche Anwendung der Schnittstellen-Bereitstellung-Heuristik

Von der Methoden-Heuristik wurde für die *IUtilOperations*-Schnittstelle ein korrekter Hinweis gefunden. Mithilfe dieses Hinweises zur *JsonUtils*-Kompilierungseinheit kann der inkorrekte Hinweis identifiziert werden. Das *data.util*-Paket enthält die *JsonUtils*-Kompilierungseinheit. Das *data.logic*-Paket hingegen hat keine Beziehung zur *JsonUtils*-Kompilierungseinheit. Auf dieser Grundlage stellt die Schnittstellen-Bereitstellung-Heuristik fest, dass der Hinweis zum *data.logic*-Paket nicht korrekt ist.

6.2.2. Aggregation von Heuristiken

Das UML-Klassendiagramm für die in dieser Arbeit verwendeten Aggregationen ist in Abbildung 6.5 dargestellt. Es werden zwei verschiedene Arten von Aggregationen unterschieden. Die erste Art von Aggregationen betrachtet für jedes Endpunkttupel einzeln die für dieses Endpunkttupel berechneten Konfidenzen und aggregiert diese. Dafür wird die Klasse *ConfidenceAggregator* implementiert, von der alle Aggregationen dieser Art erben. In Unterunterabschnitt 6.2.2.1 werden die Konfidenz-Aggregatoren Maximum und Schwellwert näher beschrieben. Die zweite Art von Aggregationen betrachtet jeden Endpunkt entweder des Quelltextmodells oder des Architekturmodells einzeln. Dafür wird die Klasse *Matcher* implementiert, von der alle Aggregationen dieser Art erben. In Unterunterabschnitt 6.2.2.2 werden die Zuordner *MatchBest*, *MatchSequentially* und *Filter* näher beschrieben.

6.2.2.1. Konfidenz-Aggregatoren

Die Maximum-Aggregation bildet das Maximum aller Konfidenzen eines Endpunkttupels. Das Maximum entspricht der Vereinigung aller von den Kindknoten gefundenen Hinweise.

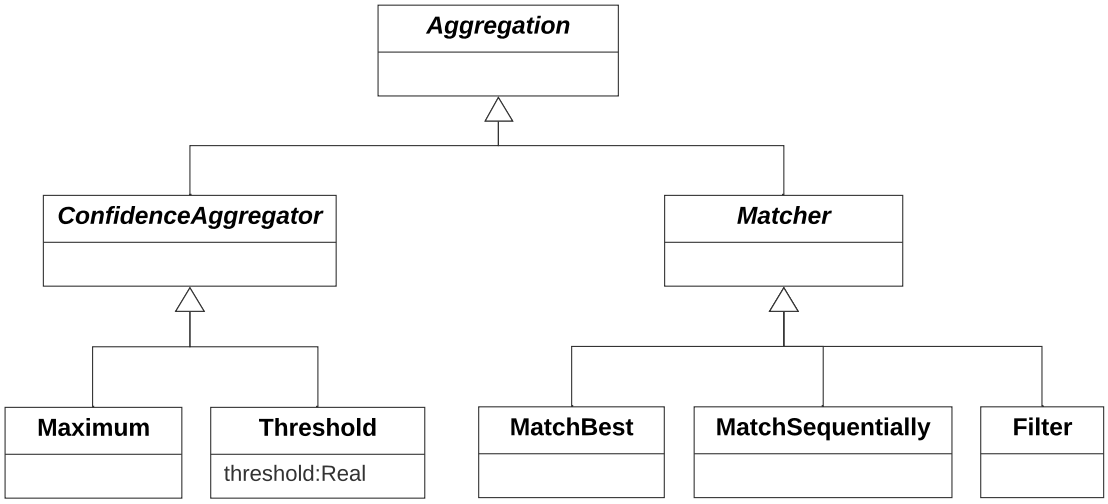

Abbildung 6.5.: UML-Klassendiagramm der verwendeten Aggregationen

Ein auf diese Weise aggregierter Konfidenzwert kann mit den Werten von Kindknoten verglichen werden, da der aggregierte Wert nie kleiner ist als ein Wert eines Kindknotens.

Eine *Threshold*-Aggregation hat einen Schwellwert zwischen null und eins. Sie überprüft, ob die Konfidenz eines Endpunkttupels mindestens so groß wie der Schwellwert ist. Ist dies nicht der Fall, wird der Hinweis entfernt. Heuristiken können ihre Konfidenzen auf unterschiedliche Art und Weise verteilen. So können manche Heuristiken gefundene Hinweise eher niedrig bewerten, während andere oft hohe Konfidenzen vergeben. Das hängt von der betrachteten Modelleigenschaft sowie der verwendeten Metrik ab. Um diese Unterschiede zu berücksichtigen und auszugleichen, können Schwellwerte entsprechend gewählt werden. Allerdings können sich die optimalen Schwellwerte zwischen einzelnen Systemen unterscheiden. Feste Werte für die Schwellwerte können nicht mit Unterschieden zwischen Systemen oder auch mit Unterschieden innerhalb eines Systems umgehen.

6.2.2.2. Zuordner

Die zweite Art von Aggregationen kann jeden Endpunkt eines Modells einzeln betrachten. Die Klasse *Matcher* erbt genau wie der Konfidenz-Aggregator von der Aggregation-Klasse. Von jedem Zuordner gibt es eine Variante, die jeden Quelltext-Endpunkt und eine Variante, die jeden Architektur-Endpunkt einzeln betrachtet. Für jeden betrachteten Endpunkt wird die `matchEndpoint`-Methode aufgerufen. Dabei hängen die Ergebnisse für einen betrachteten Endpunkt nur von den zu aggregierenden Ergebnissen, also den `childrenResults`, ab. Die Ergebnisse für die einzeln betrachteten Endpunkte bilden zusammen das Gesamtergebnis der Aggregation. Die konkreten Aggregationsmethoden *MatchBest*, *MatchSequentially* und *Filter* erben von der Zuordner-Klasse.

Die Aggregationsmethode *MatchBest* ordnet jedem der Endpunkte eines Modells die gemäß der zu aggregierenden Ergebnisse am besten passenden Endpunkte des anderen

Modells zu. Bei InMap [25] werden Klassen ebenfalls dem Architekturmodul mit dem höchsten Ähnlichkeitswert zugeordnet. Auch HuGMe [9] ordnet nur einfach zuzuordnende Quelltexteinheiten automatisiert ihren Architekturmodulen zu. Die *MatchBest*-Aggregation kann nicht nur Quelltextelemente ihrem besten Architektur-Endpunkt zuordnen, sondern auch andersrum Architektur-Endpunkten ihr bestes Quelltextelement.

Der genaue Ablauf der `matchEndpoint`-Methode der *MatchBest*-Klasse ist in Algorithmus 1 beschrieben. Zuerst wird für den betrachteten Endpunkt die beste Konfidenz unter den zu aggregierenden Ergebnissen berechnet. Hat die beste Konfidenz keinen Wert, so bedeutet dies, dass gemäß der zu aggregierenden Ergebnisse für den betrachteten Endpunkt kein Hinweis auf eine Nachverfolgbarkeitsverbindung existiert. Wird die *MatchBest*-Aggregationsmethode zum Beispiel auf eine einzelne Heuristik angewandt und die beste Konfidenz hat keinen Wert, dann hat diese Heuristik für den betrachteten Endpunkt keinen Hinweis gefunden. Die *MatchBest*-Aggregation ordnet dem betrachteten Endpunkt dann ebenfalls keinen Endpunkt zu. Existiert hingegen ein Hinweis, werden alle Endpunkte gesammelt, die gemäß der zu aggregierenden Ergebnisse mit der berechneten besten Konfidenz dem betrachteten Endpunkt zugeordnet wird. Die *MatchBest*-Aggregationsmethode ordnet die so gesammelten Endpunkte dem betrachteten Endpunkt mit der berechneten besten Konfidenz zu.

Algorithm 1 Die matchEndpoint-Methode der MatchBest-Klasse

```
function matchEndpoint(endpointToMatch, childrenResults)
    for each childResult in childrenResults do
        childBestConfidence ← the best confidence of the endpointToMatch to any
                              other endpoint in the childResult
        add the childBestConfidence to the childrenBestConfidences
    end for
    bestConfidence ← maximum of the childrenBestConfidences
    if bestConfidence has no value then
        return empty result
    end if

    for each childResult in childrenResults do
        childBestEndpointTuples ← all endpoint tuples in the childResult that
                                  contain the endpointToMatch and are mapped to
                                  the bestConfidence
        add the childBestEndpointTuples to the bestEndpointTuples
    end for
    partialMatchResult ← the bestEndpointTuples and the bestConfidence
    return partialMatchResult
end function
```

Als Beispiel wird nun betrachtet, wie die *MatchBest*-Aggregationsmethode auf eine einzelne Heuristik angewandt wird. Die Aggregationsmethode betrachtet dabei die Endpunkte des Architekturmodells einzeln. Als Heuristik wird für dieses Beispiel die in Unterabschnitt 6.2.1 beschriebene Paket-Heuristik betrachtet. Die Heuristik ordnet der

Architekturkomponente *WebApiServlet* die Inhalte des Paketes *project.web* mit Konfidenz 1/3 zu. Zudem ordnet sie der Komponente die Inhalte des Pakets *project.webapi.servlet* mit Konfidenz eins zu. Betrachtet die *MatchBest*-Aggregationsmethode nun die *WebApiServlet*-Komponente, so ordnet die Aggregation der Komponente die Inhalte des *project.webapi.servlet*-Paketes mit Konfidenz eins zu. Die Inhalte des *project.web*-Paketes ordnet die Aggregation, anders als die Heuristik, nicht der Komponente zu.

Im Gegensatz zur Schwellwert-Aggregation muss für die *MatchBest*-Aggregation kein Parameter festgelegt werden. Damit besteht bei der *MatchBest*-Aggregation nicht die Herausforderung, einen guten konkreten Schwellwert auszuwählen. Zudem kann die *MatchBest*-Aggregation im Vergleich zur Schwellwert-Aggregation besser mit Unterschieden zwischen Systemen und mit Unterschieden innerhalb eines Systems umgehen. Der optimale Schwellwert kann je nach betrachtetem System höher oder niedriger liegen. Für die *MatchBest*-Aggregation spielt es keine Rolle, ob die Konfidenzen eines betrachteten Systems allgemein höher oder niedriger liegen. Nur auf die relative Ordnung der Konfidenzen für den betrachteten Endpunkt kommt es an.

Die Aggregationsmethode *MatchSequentially* wählt für jeden der Endpunkte eines Modells das erste der zu aggregierenden Ergebnisse aus, das diesem Endpunkt einen beliebigen anderen Endpunkt zuordnet. Hier ist also die Ordnung der zu aggregierenden Ergebnisse, die durch die Anordnung der Kindknoten im Berechnungsbaum vorgegeben ist, von Bedeutung. Der genaue Ablauf der `matchEndpoint`-Methode der *MatchSequentially*-Klasse ist in Algorithmus 2 beschrieben. Das erste der zu aggregierenden Ergebnisse, das dem betrachteten Endpunkt einen beliebigen anderen Endpunkt zuordnet, wird ausgewählt. Von dem so ausgewählten Ergebnis wird der dem betrachteten Endpunkt zugehörige Teil übernommen. Das heißt, aus dem ausgewählten Ergebnis werden die Zuordnungen des betrachteten Endpunktes zu anderen Endpunkten mit den entsprechenden Konfidenzen übernommen. Die *MatchSequentially*-Methode führt dies für jeden der Endpunkte eines Modells einzeln aus.

Algorithm 2 Die matchEndpoint-Methode der MatchSequentially-Klasse

```
function matchEndpoint(endpointToMatch, childrenResults)
    for each childResult in childrenResults do
        partialChildResult ← all endpoint tuples in the childResult that contain the
                             endpointToMatch and their mapped confidences
        if partialChildResult contains a hint then
            return partialChildResult
        end if
    end for
    return empty result
end function
```

Der Hintergrundgedanke der *MatchSequentially*-Aggregationsmethode ist, dass im Berechnungsbaum die Kindknoten dieser Aggregation so angeordnet sind, dass früher vorkommende Kindknoten vielversprechendere Hinweise finden als später vorkommende Kindknoten. Das heißt genauer, dass davon ausgegangen wird, dass die Ergebnisse früherer Kindknoten eine höhere Präzision als spätere Kindknoten haben. Von der Aggre-

gation werden die früher vorkommenden Kindknoten dann bevorzugt behandelt. Findet zum Beispiel für einen Architektur-Endpunkt der erste Kindknoten eine Zuordnung zu Quelltext-Endpunkten, so wird diese Zuordnung von der Aggregation übernommen. Die folgenden Kindknoten werden für diesen Architektur-Endpunkt dann nicht mehr betrachtet. Würden für den Architektur-Endpunkt einfach alle Zuordnungen der Kindknoten zusammengefasst werden, könnte dies zwar zu einer höheren Ausbeute führen, aber die Präzision senken. Das Ziel der *MatchSequentially*-Aggregation ist es also, ein Ergebnis mit einer höheren Präzision zu erzeugen. Gleichzeitig gilt für jeden der betrachteten Endpunkte, dass wenn durch mindestens einen der Kindknoten eine Zuordnung gefunden wird, auch eine Zuordnung ausgegeben.

Auch die zuvor beschriebene *MatchBest*-Aggregation hat zum Ziel, die Präzision des Ergebnisses zu steigern, obwohl dadurch gegebenenfalls die Ausbeute gesenkt wird. Bei der *MatchBest*-Aggregation ist es allerdings wichtig, dass die Konfidenzen der einzelnen zu aggregierenden Ergebnisse gut vergleichbar sind. Vergibt eine Heuristik generell höhere Konfidenzen als eine andere, so wird sie von der *MatchBest*-Aggregation bevorzugt, obwohl sie vielleicht gar nicht bessere Ergebnisse findet. Die *MatchBest*-Aggregation eignet sich daher besonders für die Verwendung mit nur einem Kindknoten. Wenn zum Beispiel nur ein Kindknoten mit der Paket-Heuristik gewählt wird, dann werden die Konfidenzen immer nach derselben Metrik vergeben und sind somit gut vergleichbar. Die *MatchSequentially*-Aggregation hingegen benötigt keine gut vergleichbaren Konfidenzen von den Ergebnissen der Kindknoten. Es wird für einen betrachteten Endpunkt einfach nur überprüft, ob eine der zu aggregierenden Heuristiken bzw. Aggregationen einen Hinweis gefunden hat oder nicht. Damit eignet sich die *MatchSequentially*-Aggregation im Gegensatz zur *MatchBest*-Aggregation dazu, Heuristiken mit unterschiedlichen Metriken zu aggregieren.

Die *MatchSequentially*-Aggregation eignet sich besonders gut zur Aggregation von Ergebnissen, bei denen es wahrscheinlich ist, dass eine für einen Endpunkt gefundene Zuordnung die in darauffolgenden Ergebnissen für diesen Endpunkt korrekt erkannten Nachverfolgbarkeitsverbindungen schon enthält. Schließlich werden für einen Endpunkt bei einer gefundenen Zuordnung alle darauf folgenden Ergebnisse ignoriert. Als Beispiel kann ein Knoten mit einer *MatchSequentially*-Aggregation betrachtet werden, dessen erster Kindknoten die Paket-Heuristik hat und dessen zweiter Kindknoten die Namen-Heuristik hat. Es kann dann angenommen werden, dass, wenn für eine Architekturkomponente durch die erste Heuristik ein passendes Quelltextpaket gefunden wird, dieses Paket oft die von der zweiten Heuristik gefundenen Kompilierungseinheiten enthält. Das tritt auch in den bereits betrachteten konkreten Modellausschnitten in Abbildung 6.3 und Abbildung 6.4 auf. In Abbildung 6.3 enthält das *recommender*-Paket die *RecommenderImpl*-Kompilierungseinheit. In Abbildung 6.4 enthält das *data.util*-Paket die *JsonUtils*-Kompilierungseinheit. In beiden Fällen kann die Verbindung von der Kompilierungseinheit zur *Recommender*- bzw. zur *Util*-Komponente sowohl von der Paket-Heuristik als auch von der Namen-Heuristik erkannt werden.

Die *MatchSequentially*-Aggregation kann auch als iteratives Verfahren interpretiert werden, wobei jeder der Kindknoten eine Iteration darstellt. In einer Iteration können neue Zuordnungen gefunden werden. Dabei werden in jeder Iteration nur für die Architektur- bzw. Quelltext-Endpunkte ohne existierende Zuordnung neue Zuordnungen gesucht. Das

ähnelt dem Verfahren bei InMap [25]. Dort werden in jeder Iteration nur die noch nicht zugeordneten Klassen betrachtet.

Neben Heuristiken, die Hinweise auf Nachverfolgbarkeitsverbindungen suchen, gibt es auch Heuristiken, die Hinweise darauf suchen, dass ein Kandidat keine tatsächliche Nachverfolgbarkeitsverbindungen darstellt. Bei positiven Heuristiken zeigt eine hohe Konfidenz für ein Endpunkttupel an, dass hier eine generierte Nachverfolgbarkeitsverbindung wahrscheinlich korrekt wäre. Währenddessen zeigt bei negativen Heuristiken eine hohe Konfidenz für ein Endpunkttupel an, dass hier eine generierte Nachverfolgbarkeitsverbindung wahrscheinlich nicht korrekt wäre. Die Unterpaket-Entfernung-Heuristik, die Komponenten-Verbindungen-Heuristik und die Schnittstellen-Bereitstellung-Heuristik sind negative Heuristiken. Um positive und negative Heuristiken miteinander zu verbinden und zu aggregieren, kann die Filter-Aggregationsmethode genutzt werden. Die Filter-Aggregation wird so verwendet, dass ihr erster Kindknoten im Berechnungsbaum immer eine positive Heuristik verwendet. Alle darauffolgenden Kindknoten der Filter-Aggregation verwenden hingegen negative Heuristiken. Die Filter-Aggregation filtert dann die Ergebnisse ab dem zweiten Kindknoten vom Ergebnis des ersten Kindknotens. Wenn der erste Kindknoten für ein Endpunkttupel einen Hinweis auf eine Nachverfolgbarkeitsverbindung gefunden hat, wird dieser Hinweis mit der entsprechenden Konfidenz nur dann beibehalten, wenn keiner der anderen Kindknoten einen Hinweis darauf gefunden hat, dass dieses Endpunkttupel keine tatsächliche Nachverfolgbarkeitsverbindung darstellt.

Von der Filter-Aggregation gibt es drei Varianten. *FilterAlways* geht wie beschrieben vor und filtert die Ergebnisse ab dem zweiten Kindknoten auf jeden Fall. Die zwei anderen Varianten *FilterArch* und *FilterCode* betrachten jeden Architektur- bzw. Quelltext-Endpunkt einzeln. Nur falls für den betrachteten Endpunkt nach dem Filtern noch mindestens ein Hinweis existiert, wird sein Teil des zu filternden Ergebnisses auch wirklich gefiltert. Ansonsten bleibt sein Teil des Ergebnisses ungefiltert.

6.2.3. Berechnungsbaum

Der in der Evaluation verwendete Berechnungsbaum ist in Abbildung 6.6 dargestellt. Dabei handelt es sich um eine mögliche Kombination der Heuristiken, andere Kombinationen werden damit nicht ausgeschlossen. Um die Darstellung zu vereinfachen, werden duplizierte Teilbäume durch gestrichelte Pfeile dargestellt. Die Pfeilrichtung repräsentiert die Reihenfolge der Berechnung. Der Knoten, auf den gezeigt, wird hängt somit in seiner Berechnung vom Ursprung des Pfeils ab. Die Wurzel des Baumes ist der einzige Knoten ohne ausgehenden Pfeil.

Im Berechnungsbaum werden Zuordnungen für Komponenten und Architekturschnittstellen zuerst getrennt in einzelnen Teilbäumen berechnet und erst ganz am Ende kombiniert. Im Folgenden wird zuerst der Teilbaum für die Komponenten erläutert.

Die Grundlage des Komponenten-Teilbaums ist die Paket-Heuristik. Sie wird von beiden *MatchBest*-Aggregationen weiterverarbeitet. So wird jede Komponente nur auf das beste Paket und jedes Paket nur auf die beste Komponente abgebildet. Auf diesem Ergebnis wird dann eine Unterpaket-Entfernung durchgeführt. Für den Filter wird dabei die *FilterArch*-Variante verwendet. Für eine Komponente wird ein Unterpaket also nur entfernt, wenn die

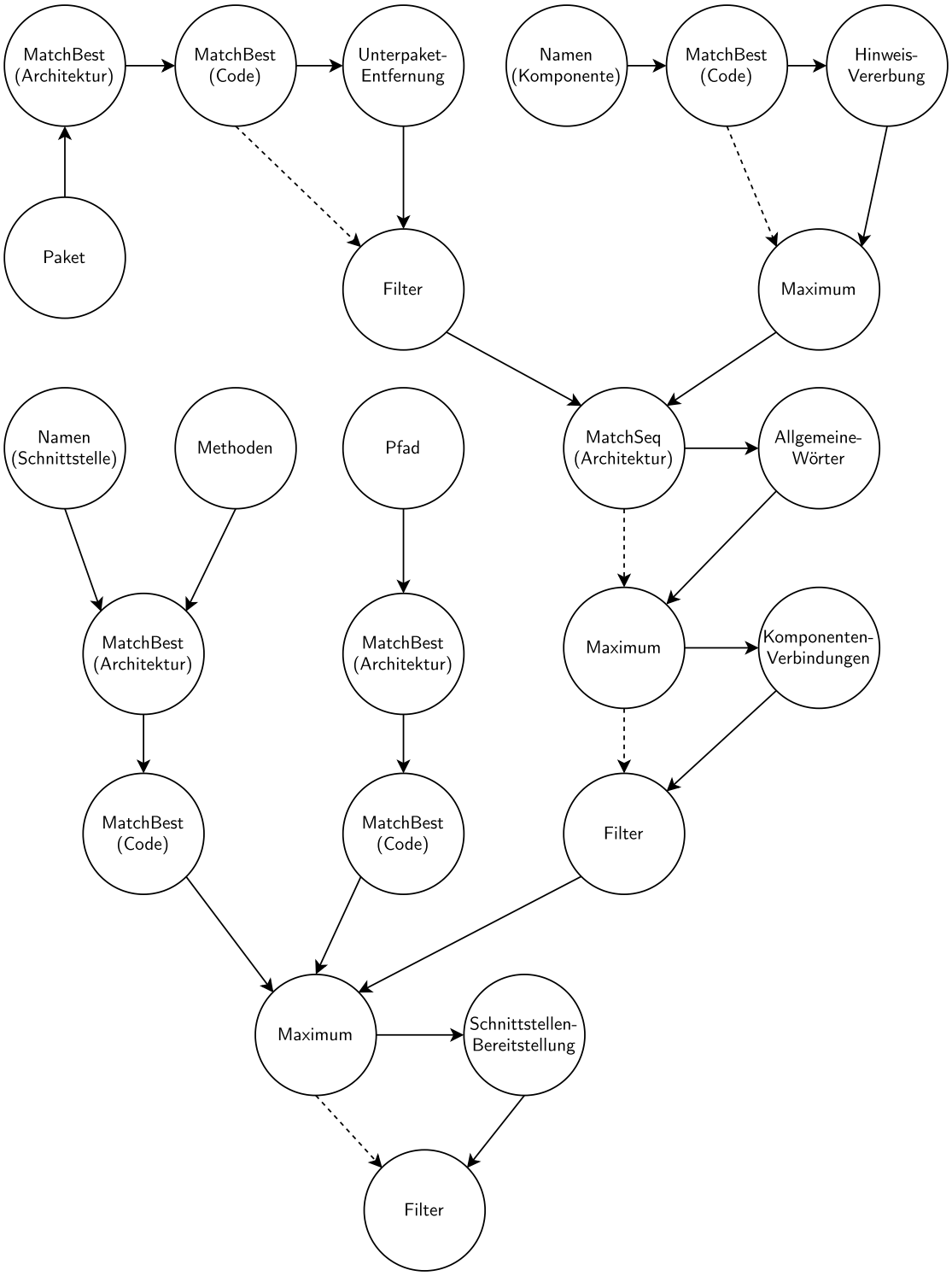

Abbildung 6.6.: Diagramm des Berechnungsbaums

durch *MatchBest* verarbeitete Paket-Heuristik ein alternatives Paket für die Komponente gefunden hat.

Die Namen-Heuristik wird im betrachteten Teilbaum ausschließlich auf Architekturkomponenten angewandt. Mit einer *MatchBest*-Aggregation wird jeder Quelltext-Endpunkt der besten Komponente zugeordnet, um die Präzision zu steigern. Ein Quelltext-Endpunkt wird also der Komponente, die den höchsten Anteil des Quelltext-Endpunkt-Namens abdeckt, zugeordnet. Eine *MatchBest*-Aggregation, die jedem Architektur-Endpunkt nur den besten Quelltext-Endpunkt zuordnet, ist hier weniger sinnvoll. Schließlich werden Komponenten meist von einer ganzen Sammlung von Quelltextelementen implementiert. Bei der Paket-Heuristik hingegen wird eine solche *MatchBest*-Aggregation verwendet. Dabei werden Komponenten schließlich ganze Pakete, also Sammlungen von Quelltext-Endpunkten, zugeordnet. Basierend auf dem Ergebnis der *MatchBest*-Aggregation werden noch Hinweise vererbt, um eine größere Ausbeute zu erreichen.

Die gefundenen Paket- und Namenhinweise werden durch eine *MatchSequentially*-Aggregation kombiniert. Die Namenhinweise werden also nur für Architekturkomponenten, für die kein Paket gefunden wurde, berücksichtigt. Es wird davon ausgegangen, dass die Struktur der Komponenten und Pakete grob übereinstimmt. Die Namenhinweise einer Komponente können aber über viele verschiedene Pakete hinweg verteilt sein. Zudem kann davon ausgegangen werden, dass in einem gefundenen Paket schon viele der korrekten Namenhinweise enthalten sind. Daher wird für eine Komponente bevorzugt nach Pakethinweisen gesucht. Nur wenn kein Paket gefunden wird werden Namenhinweise berücksichtigt. Zusätzlich werden Namenhinweise für Quelltext-Endpunkte, die in keinem Paket liegen, auf jeden Fall berücksichtigt. Für diese Quelltext-Endpunkte können schließlich prinzipiell keine Pakethinweise gefunden werden.

Zu den so erhaltenen Ergebnissen werden noch die Hinweise der Allgemeine-Wörter-Heuristik hinzugefügt. Anschließend wird die Komponenten-Verbindungen-Heuristik angewandt. Als Filter wird wieder die *FilterArch*-Variante verwendet. Auf die Pfad-Heuristik werden genau wie auf die Paket-Heuristik beide *MatchBest*-Aggregationen angewandt. Die so erhaltenen Pfadhinweise werden dem Ergebnis hinzugefügt.

Der Teilbaum für die Architekturschnittstellen nutzt die Namen-Heuristik und die Methoden-Heuristik als Blätter. Diese werden mit beiden *MatchBest*-Aggregationen kombiniert. Im Gegensatz zur Namen-Heuristik ist es hier gewollt, dass die Architekturschnittstelle nur wenigen besonders guten Quelltext-Endpunkten zugeordnet wird.

Die Ergebnisse von Komponenten und Architekturschnittstellen werden mit einer Maximum-Aggregation vereinigt. Zum Schluss wird noch die Schnittstellen-Bereitstellung-Heuristik angewandt, um die gesammelten Hinweise für Komponenten und Schnittstellen zu kombinieren und die Präzision zu erhöhen. Dabei wird wieder die *FilterArch*-Aggregation verwendet. Für alle Endpunkttupel, für die im Ergebnis des Wurzelknotens ein Hinweis existiert, werden Nachverfolgbarkeitsverbindungen generiert.

7. Evaluation

In diesem Kapitel wird die Evaluation des Ansatzes dieser Arbeit beschrieben. In Abschnitt 7.1 wird das gewählte Vorgehen bei der Evaluation in einem GQM-Plan kurz zusammengefasst.

Es wird für mehrere Fallstudien ein Goldstandard erstellt. Die Fallstudien und die Erstellung des Goldstandards werden in Abschnitt 7.2 beschrieben.

Die Qualität der einzelnen Heuristiken und Aggregationen für die Erstellung von Nachverfolgbarkeitsverbindungen wird untersucht. Dafür werden aus den Fallstudien mit den Extrahierern die entsprechenden CMTL- und AMTL-Instanzen gebildet und die Heuristiken und Aggregationen einzeln angewandt. Die von einer Heuristik bzw. Aggregation gefundenen Hinweise auf Nachverfolgbarkeitsverbindungen werden dann anhand des Goldstandards evaluiert. Dabei werden die Hinweise mit dem Goldstandard verglichen und verschiedene Metriken wie Präzision, Ausbeute und F_1-Maß erhoben. Details zu den betrachteten Metriken werden in Abschnitt 7.3 beschrieben. In Abschnitt 7.4 werden die Resultate der Heuristiken und Aggregationen erläutert. In Abschnitt A.1 im Anhang finden sich als Zusatz noch weitere Evaluationsergebnisse.

Auch die Ergebnisse des gesamten Berechnungsbaumes, also die durch Aggregation der Ergebnisse der Heuristiken entstandenen Nachverfolgbarkeitsverbindungen, werden anhand des Goldstandards evaluiert. Es werden wieder die in Abschnitt 7.3 beschriebenen Metriken verwendet. Die Ergebnisse werden in Abschnitt 7.5 beschrieben.

In Abschnitt 7.6 wird an einem Beispiel untersucht, welchen Einfluss die Konsistenz zwischen Architekturmodell und Quelltext auf die Nachverfolgbarkeitsverbindungen hat. Dafür werden verschiedene Quelltextversionen, die unterschiedlich konsistent zum Architekturmodell sind, untersucht. Für die verschiedenen Versionen werden Nachverfolgbarkeitsverbindungen für den Goldstandard erstellt. Die für die jeweilige Version generierten Nachverfolgbarkeitsverbindungen werden mithilfe der in Abschnitt 7.3 beschriebenen Metriken untersucht.

7.1. GQM-Plan

Ziel: Automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitekturmodell und Quelltext.
Frage 1: Finden die Heuristiken und Aggregationen geeignete Hinweise auf Nachverfolgbarkeitsverbindungen?
Metrik 1.1: Für die Heuristiken und Aggregationen Vergleich der durch sie erstellten Hinweise mit Goldstandard. Erhebung von Präzision, Ausbeute und F_1-Maß.
Frage 2: Wie gut werden durch die komplette Berechnung Nachverfolgbarkeitsverbindungen gefunden?

Fallstudie	Architektur	Komp.	Schnittst.	Quelltext	Java	Shell
TeaStore	V1	13	14		200	5
	V2	11	8			
TEAMMATES	grob	8	8	alt	699	1
	detailliert	17	25	neu	826	3
MediaStore		14	9		97	0
JabRef		6	0		1959	15
BigBlueButton		12	12		369	165
$\sum$		81	76		4150	189

Tabelle 7.1.: Die Anzahl der Komponenten, Schnittstellen sowie Java- und Shell-Dateien für die Fallstudien

Metrik 2.1: Vergleich der erstellten Nachverfolgbarkeitsverbindungen mit Goldstandard. Erhebung von Präzision, Ausbeute und F_1-Maß.
Frage 3: Wie gut funktioniert der Ansatz für verschiedene Programmiersprachen?
Metrik 3.1: Für die Sprachen Java und Shell Vergleich von Präzision, Ausbeute und F_1-Maß.
Frage 4: Welchen Einfluss hat die Konsistenz zwischen Architekturmodell und Quelltext auf die Ergebnisse?
Metrik 4.1: Vergleich von Präzision, Ausbeute und F_1-Maß zwischen Quelltextversionen mit unterschiedlicher Konsistenz zum Architekturmodell.

7.2. Goldstandard

Für die Evaluation wird ein Goldstandard erstellt. Dafür werden mehrere Fallstudien verwendet, die jeweils mehrere Architekturmodelle und Quelltextversionen enthalten können. Als Fallstudien werden *TeaStore*, *TEAMMATES*, *MediaStore*, *JabRef* und *BigBlueButton* verwendet. Die vorhandenen Architekturmodelle verwenden PCM und UML und stellen für die verschiedenen Systeme jeweils das Repository mit Komponenten und Schnittstellen dar. Alle Java- und Shell-Dateien der Fallstudien fließen in das Quelltextmodell ein. In Tabelle 7.1 werden für jede Fallstudie die Anzahl der Komponenten, der Architekturschnittstellen, der Java-Dateien und der Shell-Dateien aufgelistet.

TeaStore ist eine wissenschaftliche Anwendung, die auf Microservices basiert und entwickelt wurde, um als Test- und Referenzanwendung zu dienen [29]. Für *TeaStore* stehen zwei verschiedene Architekturmodelle zur Verfügung. Diese werden im Folgenden *V1* und *V2* genannt. Sie unterscheiden sich besonders bei den Architekturschnittstellen, von denen *V1* 14 hat und *V2* nur acht. Von den Komponenten sind manche nur in einem der Modelle enthalten oder anders benannt.

TEAMMATES ist ein Cloud-basierter Service, der unter anderem zur Verwaltung von Peer-Evaluationen genutzt werden kann [27]. Auch für *TEAMMATES* stehen zwei verschiedene Architekturmodelle zur Verfügung. Dabei ist eines eher grob und enthält nur acht Komponenten und genauso vielen Architekturschnittstellen. Das andere ist eher detailliert und feingranular mit 17 Architekturkomponenten und 25 Architekturschnittstellen. Im Vergleich zu einer früheren Quelltextversion ist das detaillierte Modell mit der zum Zeitpunkt dieser Arbeit aktuellen Quelltextversion in manchen Bereichen weniger konsistent. Zum Beispiel sind einige Architekturkomponenten und -schnittstellen in der neueren Quelltextversion nicht mehr implementiert. In der früheren Quelltextversion stimmen zudem die Namen mancher Architekturkomponenten und -schnittstellen stärker mit Namen im Quelltext überein. Um in den Einfluss der Konsistenz zu untersuchen, werden für beide Quelltextversionen Nachverfolgbarkeitsverbindungen für den Goldstandard erstellt.

MediaStore ist ein Beispielprojekt für Palladio, das unter anderem das Herunterladen von Audio-Dateien ermöglicht [23, S. 18]. Das Projekt ist mit nur 97 Java-Dateien eher klein.

JabRef ist eine plattformübergreifende graphische Anwendung zur Verwaltung von Literaturreferenzen [18]. Das Architekturmodell von *JabRef* ist mit sechs Komponenten eher grob, vor allem im Vergleich zu den 2098 Java-Dateien. Schnittstellen sind im Architekturmodell nicht enthalten.

BigBlueButton ist ein fürs Online-Lernen entwickeltes Konferenzsystem [6]. Im Vergleich zu den anderen Fallstudien enthält *BigBlueButton* deutlich mehr Shell-Dateien, nämlich 165.

Für jede Fallstudie werden von Hand Nachverfolgbarkeitsverbindungen erstellt, die den Goldstandard bilden. Für *TeaStore* werden dabei für beide Architekturmodelle Nachverfolgbarkeitsverbindungen erstellt. Ebenso werden für *TEAMMATES* für das grobe und das detaillierte Architekturmodell jeweils Nachverfolgbarkeitsverbindungen erstellt. Wie beschrieben werden für das detaillierte Architekturmodell für zwei verschiedene Quelltextversionen Nachverfolgbarkeitsverbindungen erstellt. Da das grobe Architekturmodell zu beiden Versionen in gleichem Maße konsistent ist, wird für dieses Modell nur die neuere Quelltextversion betrachtet.

Die Anzahl der Referenz-Nachverfolgbarkeitsverbindungen für die Fallstudien ist in Tabelle 7.2 dargestellt. Die Zahlen sind dabei nach Architekturkomponenten und Architekturschnittstellen aufgeschlüsselt. Zwischen den einzelnen Fallstudien unterscheidet sich die Zahl der Nachverfolgbarkeitsverbindungen stark. So gibt es für *MediaStore* insgesamt nur 60 Nachverfolgbarkeitsverbindungen. Für *JabRef* gibt es hingegen 1931 Nachverfolgbarkeitsverbindungen.

Im Goldstandard kann manchen Architektur-Endpunkten kein Quelltext-Endpunkt zugeordnet werden. Das kann wie bereits beschrieben auftreten, wenn der Quelltext und das Architekturmodell nicht mehr konsistent sind. Ebenso wird nicht jeder Quelltext-Endpunkt einem Architektur-Endpunkt zugeordnet. Die im Architekturmodell enthaltenen Komponenten und Schnittstellen decken also nicht unbedingt alle Kompilierungseinheiten ab.

Das grobe Architekturmodell von *TEAMMATES* sowie das Architekturmodell von *BigBlueButton* enthalten für jede Komponente jeweils eine gleichlautende Schnittstelle, die von der Komponente angeboten wird. Diese Architekturschnittstellen haben, im Gegensatz zu

Fallstudie	Version	Komp.	Schnittst.	∑
TeaStore	V1	189	20	209
	V2	155	9	164
TEAMMATES	grob	801	801	1602
	detailliert-kons.	401	25	426
	detailliert-inkons.	484	22	506
MediaStore		21	39	60
JabRef		1931	0	1931
BigBlueButton		368	368	736
∑		4350	1284	5634

Tabelle 7.2.: Die Anzahl der Referenz-Nachverfolgbarkeitsverbindungen für die Fallstudien aufgeschlüsselt nach Architekturkomponenten und -schnittstellen

den Schnittstellen der anderen Architekturmodelle, keine definierten Methodensignaturen. Für die Schnittstellen der anderen Architekturmodelle existieren damit genug Informationen, um ihnen genau wenige Quelltext-Endpunkte zuzuordnen. Jeder Schnittstelle des groben *TEAMMATES*-Modells und des *BigBlueButton*-Modells hingegen werden die gleichen Quelltext-Endpunkte zugeordnet wie der gleichlautenden die Schnittstelle anbietenden Komponente. Schließlich können für eine Schnittstelle die Quelltext-Endpunkte weder durch den Namen der Schnittstelle noch durch ihre Methoden oder sonstige Informationen eingegrenzt werden. Bei der automatisierten Generierung der Nachverfolgbarkeitsverbindungen wird dies genauso gehandhabt und solchen Architekturschnittstellen werden die gleichen Quelltext-Endpunkte wie der entsprechenden Komponente zugeordnet.

7.3. Metriken

Zur Evaluation werden die in Abschnitt 2.3 eingeführten Metriken Präzision, Ausbeute und F_1-Maß verwendet. Jedes Endpunkttupel stellt dabei ein zu klassifizierendes Objekt dar. Dieselben Metriken werden zum Beispiel auch in der Evaluation von InMap verwendet [25]. Die Genauigkeit, die im Gegensatz zu den erwähnten Metriken auch von der Zahl der *True Negatives* abhängt, wird nicht verwendet. Der Grund dafür ist, dass die Klasse „Negativ“, also die Menge der Endpunkttupel ohne Nachverfolgbarkeitsverbindung, sehr viel größer ist als die Klasse „Positiv“. Der Datensatz ist also sehr unausgewogen. Eine sehr hohe Genauigkeit könnte allein dadurch erzielt werden, dass keine einzige Nachverfolgbarkeitsverbindung generiert würde.

In Abschnitt 7.4 und Abschnitt 7.5 werden die Metriken für jede der betrachteten Fallstudien einzeln berechnet. So kann überprüft werden, ob die Ergebnisse für manche Fallstudien besonders gut oder schlecht sind. Für *TeaStore* und *TEAMMATES* werden die

Metriken jeweils für beide Architekturmodellversionen berechnet. Die verschiedenen Quelltextversionen von *TEAMMATES* werden erst in Abschnitt 7.6 betrachtet.

Auch der Mikro- und der Makro-Durchschnitt über alle betrachteten Fallstudien und ihre Architekturmodellversionen wird berechnet. Beim Mikro-Durchschnitt fließt jede Klassifikationsentscheidung in gleichem Maße in das Endergebnis ein. Das heißt zum Beispiel, jedes Endpunkttupel, für das korrekterweise eine Nachverfolgbarkeitsverbindung generiert wurde, zählt bei der Berechnung des Endergebnisses gleich viel. Das bedeutet auch, größere Fallstudien beeinflussen das Ergebnis stärker als kleinere Fallstudien. Beim Makro-Durchschnitt beeinflusst jede Fallstudie bzw. Architekturmodellversion unabhängig von ihrer Größe das Endergebnis in gleichem Maße. Das heißt jedoch, dass den Klassifikationsentscheidungen kleinerer Fallstudien mehr Gewicht geschenkt wird als den Klassifikationsentscheidungen größerer Fallstudien.

Einzelne Komponenten und Architekturschnittstellen können im Goldstandard unterschiedlich vielen Quelltext-Endpunkten zugeordnet sein. Diese Unterschiede können sehr groß sein. Einer Komponente können zum Beispiel alle Inhalte eines Pakets mit 100 Kompilierungseinheiten zugeordnet sein. Einer zweiten Komponente kann hingegen nur einer einstelligen Zahl von Quelltext-Endpunkten zugeordnet sein. Bei den bisher beschriebenen Berechnungen beeinflusst die erste Komponente das Endergebnis in deutlich größerem Maße als die zweite Komponente. Es wird zusätzlich betrachtetet, was die Ergebnisse sind, wenn jede Komponente und jede Architekturschnittstelle das Endergebnis in gleichem Maße beeinflusst. Dafür wird das Problem als Multi-Label-Klassifikation aufgefasst. Jeder Quelltext-Endpunkt bekommt alle ihm zugeordneten Architektur-Endpunkte als Label. Somit werden Präzision, Ausbeute und F_1-Maß für jede Komponente und Architekturschnittstelle einzeln berechnet. Von diesen Ergebnissen wird dann der Makro-Durchschnitt gebildet. Ein solcher Durchschnitt über die Architekturelemente wird zum Beispiel auch von Florean et al. [12] in der Evaluation ihres Ansatzes verwendet. Architekturelemente, denen im Goldstandard kein Quelltext-Endpunkt zugeordnet wird, werden bei der Durchschnittsbildung ignoriert. Für solche Architekturelemente gibt es schließlich keine *True Positives* oder *False Negatives*. Wird für ein solches Architekturelement fälschlicherweise eine Nachverfolgbarkeitsverbindung generiert, fließt das hier somit nicht mit ein. Bei dem zuvor beschriebenen Mikro-Durchschnitt sowie dem Makro-Durchschnitt über die Fallstudien beeinflusst ein solcher *False Positive* jedoch die Werte von Präzision und F_1-Maß.

Die beschriebenen Metriken können auch nur für Architekturkomponenten bzw. nur für Architekturschnittstellen berechnet werden. Somit können Heuristiken, die zum Beispiel nur Architekturkomponenten betrachten, trotzdem Werte von 100 % erzielen. Zudem kann so verglichen werden, wie sich die Ergebnisse verschiedener Arten von Architekturelementen voneinander und vom Gesamtergebnis unterscheiden.

7.4. Evaluation einzelner Heuristiken und Aggregationen

Für die Namen-Heuristik wird der Einfluss von Lemmatisierung und Stammformreduktion auf die Ergebnisse untersucht. In Tabelle 7.3 werden dafür die Ergebnisse ohne Vorverarbeitung den Ergebnissen mit Lemmatisierung bzw. Stammformreduktion gegenübergestellt.

	einzelne Kandidaten			makro ü. Arch.-Endp.		
	Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
ohne Vorverarbeitung						
mikro/alle Arch.-Endp.	0,4890	0,1209	0,1939	0,4899	0,5083	0,3987
makro ü. Versionen	0,5483	0,3057	0,3023	0,4810	0,4620	0,3687
Lemmatisierung						
mikro/alle Arch.-Endp.	0,4844	0,1244	0,1980	0,5101	0,5319	0,4107
makro ü. Versionen	0,5370	0,3130	0,3062	0,4799	0,4790	0,3656
Stammformreduktion						
mikro/alle Arch.-Endp.	0,4817	0,1254	0,1990	0,5114	0,5409	0,4129
makro ü. Versionen	0,5326	0,3185	0,3076	0,4813	0,4884	0,3679

Tabelle 7.3.: Die Ergebnisse der Namen-Heuristik ohne Vorverarbeitung, mit Lemmatisierung und mit Stammformreduktion

Die in Abschnitt 7.3 beschriebenen Metriken werden über alle Fallstudien erhoben. Dabei werden zum einen alle Kandidaten für Nachverfolgbarkeitsverbindungen, also alle Endpunkttupel, einzeln betrachtet. Diese Ergebnisse sind in der linken Tabellenhälfte aufgeführt. Für jede Vorverarbeitungsmethode ist zuerst der Mikro-Durchschnitt aufgeführt. In der nächsten Zeile ist dann der Makro-Durchschnitt über alle betrachteten Fallstudien bzw. Fallstudien-Versionen aufgeführt. Zudem werden Präzision, Ausbeute und F_1-Maß für die einzelnen Architektur-Endpunkte berechnet und darüber der Makro-Durchschnitt gebildet. Diese Ergebnisse sind in der rechten Tabellenhälfte aufgeführt. Für jede Vorverarbeitungsmethode ist zuerst der Durchschnitt über alle Architektur-Endpunkte aufgeführt. Dabei wird also jeder Architektur-Endpunkt über alle Fallstudien hinweg gleich gewichtet. In der nächsten Zeile ist dann der Makro-Durchschnitt über alle betrachteten Versionen der Fallstudien aufgeführt. Dabei werden also die einzelnen Versionen der Fallstudien gleich gewichtet. Innerhalb dieser Fallstudien-Versionen werden wieder die einzelnen Architektur-Endpunkte gleich gewichtet. Da die Namen-Heuristik sowohl Architekturkomponenten als auch Architekturschnittstellen betrachtet, wird hier bei der Berechnung der Metriken beides miteinbezogen. Für die *TEAMMATES*-Variante mit dem detaillierten Architekturmodell wird die konsistentere Quelltextversion verwendet. Die inkonsistentere Quelltextversion wird erst in Abschnitt 7.6 untersucht.

Die Ausbeute steigt bei der Verwendung von Lemmatisierung im Vergleich zu den Ergebnissen ohne Vorverarbeitung. Die Präzision bei der Betrachtung einzelner Kandidaten für Nachverfolgbarkeitsverbindungen sinkt hingegen leicht. Es werden also neue, korrekte Hinweise auf Nachverfolgbarkeitsverbindungen gefunden. Zum Beispiel wird in *TEAMMATES* eine Zuordnung zwischen der Komponente *Bundles* und der Kompilierungseinheit *SessionResultsBundle* gefunden. Es kommen aber auch ein paar inkorrekte Hinweise hinzu.

Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten steigt leicht um 0,41 Prozentpunkte. Der Makro-Durchschnitt des F_1-Maßes über alle Architektur-Endpunkte steigt um 1,2 Prozentpunkte.

Analog verhält sich der Vergleich von der Stammformreduktion mit der Lemmatisierung. Während die Ausbeute steigt, sinkt die Präzision bei der Betrachtung einzelner Kandidaten leicht. Es werden also wieder neue, korrekte Hinweise auf Nachverfolgbarkeitsverbindungen gefunden. Das kann vorkommen, wenn sich die Wortart bei verglichenen Wörtern unterscheidet. Bei der Lemmatisierung werden Wörter mit unterschiedlichen Wortarten im Allgemeinen nicht auf dasselbe Lemma zurückgeführt. Die Wörter können aber denselben Wortstamm haben. So wird zum Beispiel in *MediaStore* eine Zuordnung zwischen der Komponente *Reencoding* und der Kompilierungseinheit *ReEncoderImpl* gefunden. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten steigt leicht um 0,1 Prozentpunkte. Der Makro-Durchschnitt des F_1-Maßes über alle Architektur-Endpunkte steigt um 0,22 Prozentpunkte.

Die Stammformreduktion erzielt also das beste F_1-Maß und wird für die Namen-Heuristik im Berechnungsbaum verwendet. Es lässt sich jedoch insgesamt feststellen, dass die Werte sich immer nur leicht unterscheiden. In den Fallstudien wird bei den betrachteten Namen die Wortform also nur selten verändert. Generell hängt die geeignetste Vorverarbeitungsmethode auch davon ab, ob Präzision oder Ausbeute als wichtiger erachtet werden.

In Unterabschnitt 7.4.1 werden nun die Heuristiken evaluiert, die zur Generierung der Nachverfolgbarkeitsverbindungen für Architekturschnittstellen verwendet werden. Anschließend wird in Unterabschnitt 7.4.2 auf die Generierung der Nachverfolgbarkeitsverbindungen für Komponenten eingegangen.

7.4.1. Heuristiken für Architekturschnittstellen

Im Folgenden werden die Knoten im Berechnungsbaum, die die Nachverfolgbarkeitsverbindungen für Architekturschnittstellen generieren, betrachtet. Bei den Metriken in diesem Abschnitt werden daher nur Architekturschnittstellen und keine Komponenten betrachtet. Die Schnittstellen des groben Architekturmodells von *TEAMMATES* sowie des Architekturmodells von *BigBlueButton* werden ignoriert. Wie in Abschnitt 7.2 beschrieben erhalten diese Schnittstellen die gleichen Nachverfolgbarkeitsverbindungen wie die ihnen entsprechenden Komponenten. Für diese Schnittstellen gelten also die in Unterabschnitt 7.4.2 beschriebenen Evaluationsergebnisse der ihnen entsprechenden Komponenten.

In Tabelle 7.4 sind die Ergebnisse der Namen-Heuristik mit Stammformreduktion für die einzelnen Fallstudien aufgelistet. Dabei werden nur Architekturschnittstellen und keine Komponenten betrachtet. Das entspricht dem Knoten mit der Namen-Heuristik im Berechnungsbaum, der nur Architekturschnittstellen betrachtet. Die Ausbeute ist hier deutlich höher als die Präzision. Der Mikro-Durchschnitt der Ausbeute über einzelne Kandidaten beträgt 82,8 %. Die Präzision hingegen beträgt nur 18,6 %. Zudem unterscheiden sich die Werte zwischen den Fallstudien recht stark. Bei *TEAMMATES* beträgt das F_1-Maß über einzelne Kandidaten nur 13,84 %. Bei *MediaStore* hingegen beträgt das F_1-Maß 82,98 %. Die Namen von Architekturschnittstellen und ihren Quelltextentsprechungen können sich also je nach Fallstudie unterschiedlich stark ähneln. Genauso können sich die Namen

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,2182	0,6000	0,3200	0,5256	0,7143	0,5327
	V2	0,1539	0,6667	0,2500	0,3243	0,6875	0,3343
TEAMMATES	det.-kons.	0,0758	0,8000	0,1384	0,4599	0,9091	0,5523
MediaStore		0,7091	1,0000	0,8298	0,6906	1,0000	0,8027
mikro/alle Arch.-Endp.		0,1864	0,8280	0,3044	0,4922	0,8365	0,5520
makro ü. Versionen		0,2892	0,7667	0,3846	0,5001	0,8277	0,5555

Tabelle 7.4.: Die Ergebnisse der Namen-Heuristik mit Stammformreduktion für Architekturschnittstellen

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,6522	0,7500	0,6977	0,6381	0,7679	0,6548
	V2	1,0000	0,1111	0,2000	0,1250	0,1250	0,1250
TEAMMATES	det.-kons.	0,7097	0,8800	0,7857	0,7106	0,8640	0,7601
MediaStore		0,3846	0,1282	0,1923	0,3021	0,1875	0,2177
mikro/alle Arch.-Endp.		0,6324	0,4624	0,5342	0,5381	0,6202	0,5506
makro ü. Versionen		0,6866	0,4673	0,4689	0,4440	0,4860	0,4394

Tabelle 7.5.: Die Ergebnisse der Methoden-Heuristik für Architekturschnittstellen

von Architekturschnittstellen und Quelltextelementen, die nicht diesen Schnittstellen entsprechen, je nach Fallstudie unterschiedlich stark ähneln.

In Tabelle 7.5 sind die Ergebnisse der Methoden-Heuristik für die einzelnen Fallstudien aufgelistet. Der Mikro-Durchschnitt der Präzision über einzelne Kandidaten beträgt 63,24 %. Die Ausbeute hingegen beträgt 46,24 %. Im Gegensatz zur Namen-Heuristik ist die Präzision hier höher als die Ausbeute. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten beträgt 53,42 %. Das ist höher als der entsprechende Wert von 30,44 % bei der NamenHeuristik. Wie bei der Namen-Heuristik unterscheiden sich die Werte zwischen den Fallstudien recht stark. Bei *MediaStore* beträgt das F_1-Maß über einzelne Kandidaten nur 19,23 %. Bei *TEAMMATES* hingegen beträgt das F_1-Maß 78,57 %. Das ist genau andersherum wie bei der Namen-Heuristik. Dort war das F_1-Maß bei *TEAMMATES* deutlich schlechter als bei *MediaStore*. Das deutet darauf hin, dass die Heuristiken unterschiedliche Hinweise finden. Damit ist eine Kombination der beiden Heuristiken vielversprechend.

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,8824	0,7500	0,8108	0,9048	0,8214	0,8503
	V2	0,5556	0,5556	0,5556	0,6250	0,5625	0,5833
TEAMMATES	det.-kons.	0,9565	0,8800	0,9167	0,9773	0,9659	0,9576
MediaStore		0,9750	1,0000	0,9873	0,9583	1,0000	0,9750
mikro/alle Arch.-Endp.		0,9101	0,8710	0,8901	0,9006	0,8702	0,8738
makro ü. Versionen		0,8424	0,7964	0,8176	0,8663	0,8375	0,8416

Tabelle 7.6.: Die Endergebnisse für Architekturschnittstellen durch Kombination von Namen- und Methoden-Heuristik

In Tabelle 7.6 sind die finalen Ergebnisse für Architekturschnittstellen für die einzelnen Fallstudien dargestellt. Dafür werden die Namen- und Methoden-Heuristik mit *Match-Best*-Aggregationen kombiniert. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten beträgt 89,01 %. Damit ist der Wert deutlich höher als die Werte der Namen- und Methoden-Heuristiken. Dasselbe gilt auch, wenn der Durchschnitt über die Architektur-Endpunkte betrachtet wird. Durch die Aggregation konnten Präzision und Ausbeute gesteigert werden.

7.4.2. Heuristiken für Architekturkomponenten

Im Folgenden werden die Knoten im Berechnungsbaum, die die Nachverfolgbarkeitsverbindungen für Architekturkomponenten generieren, untersucht. Bei den Metriken in diesem Abschnitt werden daher nur Komponenten und keine Architekturschnittstellen betrachtet.

Für die Paket-Heuristik wird wie für die Namen-Heuristik der Einfluss von Lemmtatisierung und Stammformreduktion auf die Ergebnisse untersucht. In Tabelle 7.7 werden dafür die Ergebnisse ohne Vorverarbeitung den Ergebnissen mit Lemmatisierung bzw. Stammformreduktion gegenübergestellt.

Die Ausbeute steigt bei der Verwendung von Lemmatisierung im Vergleich zu den Ergebnissen ohne Vorverarbeitung. Es werden also neue, korrekte Hinweise auf Nachverfolgbarkeitsverbindungen gefunden. Die Präzision bei der Betrachtung einzelner Kandidaten für Nachverfolgbarkeitsverbindungen sinkt leicht. Beim Durchschnitt über die Komponenten steigt die Präzision hingegen leicht.

Ähnlich verhält sich der Vergleich von der Stammformreduktion mit der Lemmatisierung. Auch hier steigt die Ausbeute leicht. Die Präzision bei der Betrachtung einzelner Kandidaten für Nachverfolgbarkeitsverbindungen ändert sich bei diesem Vergleich nicht.

Bei der Betrachtung einzelner Kandidaten beeinflusst die Paketgröße die Ergebnisse. Je größer die Anzahl der im Paket enthaltenen Quelltext-Endpunkte, desto stärker beeinflusst eine richtige oder falsche Zuordnung dieses Pakets zu einer Komponente die Ergebnisse

	einzelne Kandidaten			makro ü. Arch.-Endp.		
	Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
ohne Vorverarbeitung						
mikro/alle Arch.-Endp.	0,8313	0,8896	0,8594	0,5941	0,8006	0,6210
makro ü. Versionen	0,7378	0,8093	0,7523	0,6291	0,8141	0,6532
Lemmatisierung						
mikro/alle Arch.-Endp.	0,8262	0,8924	0,8580	0,6050	0,8143	0,6334
makro ü. Versionen	0,7309	0,8133	0,7520	0,6367	0,8243	0,6622
Stammformreduktion						
mikro/alle Arch.-Endp.	0,8262	0,8927	0,8581	0,6187	0,8280	0,6471
makro ü. Versionen	0,7317	0,8201	0,7562	0,6497	0,8373	0,6752

Tabelle 7.7.: Die Ergebnisse der Paket-Heuristik ohne Vorverarbeitung, mit Lemmatisierung und mit Stammformreduktion für Architekturkomponenten

positiv oder negativ. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten ist ohne Vorverarbeitung am größten. Er beträgt 85,94 %.

Beim Durchschnitt über die Komponenten spielt die Paketgröße keine Rolle. Der Makro-Durchschnitt des F_1-Maßes über alle Komponenten ist mit Stammformreduktion am größten. Er beträgt 64,71 %. Für folgende Berechnungen wird die Paket-Heuristik somit mit Stammformreduktion verwendet. Das entspricht auch der Verwendung von der Stammformreduktion bei der Namen-Heuristik.

Es lässt sich wie bei der Namen-Heuristik feststellen, dass die Wahl der Vorverarbeitungsmethode die Ergebnisse nur leicht beeinflusst. Ein klarer genereller Vorteil einer bestimmten Vorverarbeitungsmethode lässt sich aus den Ergebnissen nicht ableiten. Das hängt auch davon ab, ob Präzision oder Ausbeute als wichtiger erachtet werden.

In Tabelle 7.10 sind die Ergebnisse der Paket-Heuristik mit Stammformreduktion nach den MatchBest-Aggregationen dargestellt. Es wurde also jede Komponente nur auf die Pakete mit der höchsten Konfidenz abgebildet. Zudem wurde jedes Paket nur auf die Komponenten mit der höchsten Konfidenz abgebildet.

Die Ergebnisse werden mit den in Tabelle 7.7 dargestellten Ergebnissen der Paket-Heuristik mit Stammformreduktion vor den MatchBest-Aggregationen verglichen. Durch die MatchBest-Aggregationen konnte die Präzision gesteigert werden. Der Mikro-Durchschnitt der Präzision über einzelne Kandidaten steigt von 82,62 % auf 91,11 %. Der Makro-Durchschnitt der Präzision über alle Komponenten steigt von 61,87 % auf 65,57 %. Die Ausbeute hingegen sinkt durch die Aggregationen. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten steigt von 85,81 % auf 89,91 %. Der Makro-Durchschnitt des F_1-Maßes über alle Komponenten steigt von 64,71 % auf 67,43 %. Trotz der sinkenden Ausbeute kann das F_1-Maß somit aufgrund der steigenden Präzision verbessert werden.

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,9036	0,9418	0,9223	0,6070	0,6883	0,6293
	V2	0,9351	0,9290	0,9320	0,5076	0,5407	0,5191
TEAMMATES	det.-kons.	0,4578	0,4464	0,4520	0,6264	0,7143	0,6375
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		0,8947	0,8095	0,8500	0,8485	0,9091	0,8727
JabRef		0,9650	0,9990	0,9817	0,7008	0,8331	0,7232
BigBlueButton		0,8927	0,4973	0,6387	0,4621	0,4134	0,4290
mikro/alle Arch.-Endp.		0,9111	0,8875	0,8991	0,6557	0,7047	0,6640
makro ü. Versionen		0,8641	0,8033	0,8253	0,6789	0,7284	0,6873
mit Unterpaket-Entfernung							
mikro/alle Arch.-Endp.		0,9326	0,8875	0,9095	0,6694	0,7047	0,6743
makro ü. Versionen		0,8807	0,8033	0,8306	0,7005	0,7284	0,7042

Tabelle 7.8.: Die Ergebnisse für Architekturkomponenten nach den MatchBest-Aggregationen der Paket-Heuristik mit Stammformreduktion sowie nach der zusätzlichen Unterpaket-Entfernung

Bei der Betrachtung der einzelnen Fallstudien fällt auf, dass *BigBlueButton* eine vergleichsweise niedrige Ausbeute hat. Das kann dadurch erklärt werden, dass in dieser Fallstudie der Anteil der Shell-Dateien besonders hoch ist. Für Shell-Dateien liegen keine Paketinformationen vor, die von der Heuristik genutzt werden könnten. Zudem gibt es bei *TEAMMATES* hohe Unterschiede in den Werten für die grobe und die detaillierte Architekturmodellversion. Die grobe Version erreicht ein F_1-Maß von 100 %. Bei der detaillierten Version hingegen beträgt das F_1-Maß über einzelne Kandidaten nur 45,20 %. Das kann als Hinweis betrachtet werden, dass die Paket-Heuristik für grobe Architekturmodelle besser funktioniert. Der Grund dafür kann zum Beispiel sein, dass die Paketstruktur besonders gut der Aufteilung in Komponenten entspricht. Zudem gibt es weniger Komponenten, denen ein Paket überhaupt zugeordnet werden kann. Das erleichtert die Entscheidung zwischen den Komponenten.

Im unteren Teil der Tabelle 7.10 sind die Ergebnisse dargestellt, wenn zusätzlich noch die Unterpaket-Entfernung mit der gleichlautenden Heuristik durchgeführt wird. Die Präzision kann durch die Unterpaket-Entfernung gesteigert werden. Der Mikro-Durchschnitt der Präzision über einzelne Kandidaten steigt von 91,11 % auf 93,26 %. Der Makro-Durchschnitt der Präzision über alle Komponenten steigt von 65,57 % auf 66,94 %. Die Ausbeute sinkt dabei nicht. Das heißt, für die betrachteten Fallstudien wird kein Unterpaket inkorrekterweise entfernt. Die Unterpaket-Entfernung-Heuristik selbst erzielt also mit ihren Hinweisen, welche Unterpakete entfernt werden sollen, eine Präzision von 100 %.

In Tabelle 7.9 sind die Ergebnisse der Namen-Heuristik mit Stammformreduktion nach der *MatchBest*-Aggregation aufgelistet. Es werden wie bei der Paket-Heuristik nur Architekturkomponenten betrachtet. Die Betrachtung der Architekturschnittstellen findet sich in Abschnitt 7.4. Jeder Quelltext-Endpunkt wurde durch die *MatchBest*-Aggregation den Architekturkomponenten mit der höchsten Konfidenz zugewiesen. Die Präzision ist hier insgesamt deutlich höher als die Ausbeute. Der Mikro-Durchschnitt der Präzision über einzelne Kandidaten beträgt 60,2 %. Die Ausbeute hingegen beträgt nur 12,21 %. Diese Werte liegen unter den Werten der Paket-Heuristik. Insbesondere die Ausbeute ist deutlich niedriger.

Bei Betrachtung der einzelnen Fallstudien fällt auf, dass sich die Werte recht stark unterscheiden. Für *JabRef* beträgt das F_1-Maß über einzelne Kandidaten nur 2,3 %. Für *TEAMMATES* hingegen beträgt das F_1-Maß für die detaillierte Architekturmodellversion 72,81 %. Das ist das beste Ergebnis unter allen Fallstudien und Versionen. Die grobe Architekturmodellversion erreicht nur ein F_1-Maß von 17,85 %. Auch die unterschiedlichen Architekturmodellversionen von *TEAMMATES* liefern also klar unterschiedliche Ergebnisse. Das kann als Hinweis betrachtet werden, dass die Namen-Heuristik für detaillierte Architekturmodelle besser funktioniert. Bei einer feingranulareren Aufteilung in Komponenten werden einer Komponente im Goldstandard weniger Kompilierungseinheiten zugeordnet. Damit kommt der Name der Komponente in den Namen der zugeordneten Kompilierungseinheiten und ihren Datentypen offensichtlich häufiger vor.

Im Gegensatz zur Namen-Heuristik liefert die Paket-Heuristik eher für die grobgranulareren Architekturmodelle bessere Ergebnisse. Bei der Namen-Heuristik liefert die detaillierte *TEAMMATES*-Version wie beschrieben bei Betrachtung einzelner Kandidaten das höchste F_1-Maß unter allen Fallstudien und Versionen. Bei der Paket-Heuristik liefert diese Metrik für die gleiche Version hingegen den niedrigsten Wert unter allen Fallstudien

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,9091	0,2646	0,4098	0,8255	0,5044	0,5662
	V2	0,9231	0,2323	0,3711	0,9535	0,6155	0,6647
TEAMMATES	det.-kons.	0,8605	0,6309	0,7281	0,6923	0,4856	0,4766
	grob	0,9405	0,0986	0,1785	0,4286	0,1379	0,1930
MediaStore		0,2642	0,6667	0,3784	0,3046	0,7500	0,3945
JabRef		0,1042	0,0130	0,0230	0,3260	0,3590	0,1822
BigBlueButton		0,7895	0,0408	0,0775	0,3523	0,0630	0,0952
mikro/alle Arch.-Endp.		0,6020	0,1221	0,2030	0,5903	0,4409	0,3997
makro ü. Versionen		0,6844	0,2781	0,3095	0,5547	0,4165	0,3675
mit Hinweis-Vererbung							
mikro/alle Arch.-Endp.		0,6214	0,1337	0,2201	0,5972	0,4662	0,4222
makro ü. Versionen		0,7014	0,2998	0,3296	0,5608	0,4370	0,3859

Tabelle 7.9.: Die Ergebnisse für Architekturkomponenten nach der MatchBest-Aggregation der Namen-Heuristik mit Stammformreduktion sowie nach der zusätzlichen Hinweis-Vererbung

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,9087	1,0000	0,9521	0,9147	1,0000	0,9390
	V2	0,9394	1,0000	0,9688	0,9621	1,0000	0,9761
TEAMMATES	det.-kons.	0,6376	0,9651	0,7679	0,8761	0,9620	0,8737
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		0,9130	1,0000	0,9546	0,9394	1,0000	0,9636
JabRef		0,9979	0,9995	0,9987	0,8667	0,9998	0,8888
BigBlueButton		0,9586	0,5027	0,6595	0,5704	0,4274	0,4601
mikro/alle Arch.-Endp.		0,9331	0,9488	0,9409	0,8705	0,9064	0,8653
makro ü. Versionen		0,9079	0,9239	0,9002	0,8756	0,9127	0,8716

Tabelle 7.10.: Die Ergebnisse der Aggregation von Paket- und Namen-Hinweisen für Architekturkomponenten

und Versionen. Das deutet darauf hin, dass sich Paket- und Namen-Hinweise gut zur Kombination eignen.

Im unteren Teil der Tabelle 7.9 sind die Ergebnisse dargestellt, wenn zusätzlich noch die Hinweis-Vererbung mit der gleichlautenden Heuristik durchgeführt wird. Sowohl Präzision als auch Ausbeute können dadurch gesteigert werden. Der Makro-Durchschnitt des F_1-Maßes über alle Komponenten steigt von 39,97 % auf 42,22 %. Auch alle anderen Mikro- und Makro-Durchschnitte der Metriken steigen.

In Tabelle 7.10 sind die Ergebnisse der Aggregation von Paket- und Namen-Hinweisen dargestellt. Für die Kombination wird wie in Unterabschnitt 6.2.3 beschrieben die *Match-Sequentially*-Aggregation eingesetzt. Die Ergebnisse sind deutlich besser als die zuvor beschriebenen Ergebnisse der Paket- bzw. Namen-Heuristik allein. Sowohl Präzision als auch Ausbeute steigen durch die Kombination. Der Makro-Durchschnitt des F_1-Maßes über alle Komponenten beträgt 86,53 %. Die aggregierten Paket- und Namen-Hinweise erreichen wie beschrieben nur 67,43 % bzw. 42,22 %. Neben der groben *TEAMMATES*-Version können nun auch *MediaStore* und beide *TeaStore*-Versionen eine Ausbeute von 100 % erreichen.

In Tabelle 7.11 sind die Ergebnisse verschiedener Heuristiken dargestellt, die die zuvor beschriebene Aggregation von Paket- und Namen-Hinweisen weiter verbessern. Zuerst werden die von der Allgemeine-Wörter-Heuristik gefundenen Hinweise den aggregierten Paket- und Namen-Hinweisen hinzugefügt. Dadurch können sowohl Präzision als auch Ausbeute leicht gesteigert werden. Es werden also für die betrachteten Fallstudien neue, korrekte Hinweise auf Nachverfolgbarkeitsverbindungen gefundenen. Zudem haben die gefundenen Hinweise eine höhere Präzision als die aggregierten Paket- und Namen-

	einzelne Kandidaten			makro ü. Arch.-Endp.		
	Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
Hinzufügen von Allgemeine-Wörter-Hinweisen						
mikro/alle Arch.-Endp.	0,9333	0,9519	0,9425	0,8717	0,9129	0,8704
makro ü. Versionen	0,9088	0,9279	0,9023	0,8768	0,9178	0,8757
Entfernen von Komponenten-Verbindungen-Hinweisen						
mikro/alle Arch.-Endp.	0,9890	0,9519	0,9701	0,9106	0,9129	0,9004
makro ü. Versionen	0,9841	0,9279	0,9459	0,9096	0,9178	0,9005
Hinzufügen von Pfad-Hinweisen						
mikro/alle Arch.-Endp.	0,9887	0,9951	0,9919	0,9673	0,9902	0,9725
makro ü. Versionen	0,9857	0,9928	0,9891	0,9633	0,9910	0,9688

Tabelle 7.11.: Die Ergebnisse verschiedener auf der Aggregation von Paket- und Namen-Hinweisen aufbauenden Heuristiken für Architekturkomponenten

Hinweise. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten steigt von 94,09 % auf 94,25 %.

Nach der Allgemeine-Wörter-Heuristik wird die Komponenten-Verbindungen-Heuristik angewandt. Diese Heuristik findet fehlerhafte Hinweise auf Nachverfolgbarkeitsverbindungen. Durch das Entfernen der von der Heuristik gefundenen Hinweisen kann die Präzision erhöht werden. Der Mikro-Durchschnitt der Präzision über einzelne Kandidaten steigt von 93,33 % auf 98,90 %. Die Ausbeute sinkt dabei nicht. Das heißt, es wird kein Hinweis inkorrekterweise entfernt. Die Komponenten-Verbindungen-Heuristik erkennt also für die Fallstudien fehlerhafte Hinweise mit einer Präzision von 100 %.

Anschließend werden noch die Pfad-Hinweise hinzugefügt. Dadurch können die Ergebnisse nochmal verbessert werden. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten steigt von 97,01 % auf 99,19 %. Der Makro-Durchschnitt des F_1-Maßes über alle Komponenten steigt von 90,04 % auf 97,25 %. Durch die Pfad-Heuristik kann insbesondere für die *BigBlueButton*-Fallstudie mit ihrer großen Anzahl von Shell-Dateien die Ausbeute erhöht werden.

Zum Schluss der Berechnung werden die Hinweise für die Architekturschnittstellen genutzt, um die Hinweise für die Komponenten zu verbessern. Die zuvor beschriebenen Ergebnisse nach dem Hinzufügen der Pfad-Hinweise werden also mithilfe der in Unterabschnitt 7.4.1 gefundenen Hinweise verbessert. Dafür wird die Schnittstellen-Bereitstellung-Heuristik angewandt. Die Ergebnisse sind in Tabelle 7.12 dargestellt. Die Heuristik findet fehlerhafte Hinweise auf Nachverfolgbarkeitsverbindungen. Werden diese Hinweise entfernt, steigt der Mikro-Durchschnitt der Präzision über einzelne Kandidaten von 98,87 % auf 99,33 %. Die Ausbeute sinkt dabei nicht. Das heißt, kein Hinweis wird fälschlicherweise

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
	V2	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
TEAMMATES	det.-kons.	0,9730	0,9900	0,9815	0,9423	0,9903	0,9580
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
JabRef		0,9979	1,0000	0,9990	0,8667	1,0000	0,8889
BigBlueButton		0,9698	0,9592	0,9645	0,9543	0,9469	0,9465
mikro/alle Arch.-Endp.		0,9933	0,9951	0,9942	0,9711	0,9902	0,9748
makro ü. Versionen		0,9915	0,9928	0,9921	0,9662	0,9910	0,9705

Tabelle 7.12.: Die Endergebnisse für Architekturkomponenten nach der Schnittstellen-Bereitstellung-Heuristik

entfernt. Die Heuristik erkennt also für die Fallstudien fehlerhafte Hinweise mit einer Präzision von 100 %.

Die somit erzielten Endergebnisse für die Architekturkomponenten sind sehr hoch. Insgesamt beträgt der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten 99,42 %. Für beide *TeaStore*-Versionen, die grobe *TEAMMATES*-Version und *MediaStore* kann sogar ein F_1-Maß von 100 % erzielt werden.

7.5. Evaluation der generierten Nachverfolgbarkeitsverbindungen

In Tabelle 7.13 sind die Evaluationsergebnisse für die finalen generierten Nachverfolgbarkeitsverbindungen dargestellt. Hier sind also alle Architekturkomponenten und alle Architekturschnittstellen berücksichtigt. Die Ergebnisse nur für die Komponenten sind in Tabelle 7.12 dargestellt. Ergebnisse spezifisch für Architekturschnittstellen sind in Tabelle 7.6 dargestellt. Es wird insgesamt ein Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten von 99,11 % erreicht. Präzision und Ausbeute sind dabei fast gleich groß. Für alle Fallstudien und Versionen werden hohe F_1-Maße erzielt. Bei der Betrachtung einzelner Kandidaten unterscheiden sich die Werte für die einzelnen Fallstudien und Versionen nur leicht. Sie reichen von 96,45 % für *BigBlueButton* zu 100 % für die grobe *TEAMMATES*-Version. Beim Makro-Durchschnitt über die Architektur-Endpunkte unterscheiden sich die Werte etwas stärker. Der Makro-Durchschnitt über alle Architektur-Endpunkte liegt mit 93,71 % unter dem Mikro-Durchschnitt. Das heißt, Architekturelemente, denen im Goldstandard viele Kompilierungseinheiten zugeordnet sind, werden von den Heuristiken

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,9903	0,9761	0,9831	0,9506	0,9074	0,9224
	V2	0,9756	0,9756	0,9756	0,8421	0,8158	0,8246
TEAMMATES	det.-kons.	0,9722	0,9836	0,9778	0,9637	0,9754	0,9578
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		0,9836	1,0000	0,9917	0,9825	1,0000	0,9895
JabRef		0,9979	1,0000	0,9990	0,8667	1,0000	0,8889
BigBlueButton		0,9698	0,9592	0,9645	0,9543	0,9469	0,9465
mikro/alle Arch.-Endp.		0,9912	0,9910	0,9911	0,9456	0,9437	0,9371
makro ü. Versionen		0,9842	0,9849	0,9845	0,9371	0,9494	0,9328

Tabelle 7.13.: Die Evaluationsergebnisse für die generierten Nachverfolgbarkeitsverbindungen

	Präz.	Ausb.	F_1
Java	0,9926	0,9944	0,9935
Shell	0,9474	0,8889	0,9172

Tabelle 7.14.: Vergleich der Mikro-Durchschnitte von Präzision, Ausbeute und F_1-Maß für Java und Shell

besonders gut erkannt. Insbesondere werden Komponenten besser erkannt als Architekturschnittstellen. Komponenten erzielen einen Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten von 99,42 %. Für Architekturschnittstellen liegt dieser Wert nur bei 89,01 %.

Die Metriken können auch nur für Java- bzw. nur für Shell-Dateien berechnet werden. Für Java ergibt sich ein Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten von 99,35 %. Für Shell liegt diese Metrik bei 91,72 %. Der Grund für die besseren Ergebnisse für Java kann sein, dass für Shell keine Paketinformationen vorliegen. Auch *extends*- und *implements*-Beziehungen sind für Shell nicht vorhanden. Allerdings ist die Aussagekraft der Evaluationsergebnisse für Shell begrenzt, da bis auf *BigBlueButton* in den betrachteten Fallstudien nur wenige Shell-Dateien enthalten sind.

7.6. Einfluss der Konsistenz auf die Ergebnisse

In Tabelle 7.15 sind für das detaillierte *TEAMMATES*-Architekturmodell die Evaluationsergebnisse zweier verschiedener Quelltextversionen dargestellt. Die ältere und konsistentere

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TEAMMATES	det.-kons.	0,9722	0,9836	0,9778	0,9637	0,9754	0,9578
	det.-inkons.	0,8933	0,9427	0,9173	0,9353	0,9191	0,9037

Tabelle 7.15.: Vergleich der Evaluationsergebnisse für die konsistentere und die inkonsistentere Quelltextversion von TEAMMATES

Version erzielt im Vergleich bessere Ergebnisse als die neuere und inkonsistentere Version. Der Mikro-Durchschnitt des F_1-Maßes über einzelne Kandidaten ist um 6,05 Prozentpunkte höher. Der Makro-Durchschnitt des F_1-Maßes über alle Architektur-Endpunkte ist um 5,41 Prozentpunkte höher. Daraus lässt sich schließen, dass die Qualität der generierten Nachverfolgbarkeitsverbindungen abnehmen kann, wenn die Implementierung weiterentwickelt wird, das Architekturmodell aber nicht aktualisiert wird. Der Grund für schlechtere Werte einer neueren Quelltextversion kann zum Beispiel sein, dass manche Komponenten oder Architekturschnittstellen gar nicht mehr im Quelltext vorhanden sind. Diesen Architekturelementen werden im TLG-Schritt aber möglicherweise trotzdem Quelltext-Endpunkte zugeordnet. Dadurch sinkt die Präzision. Zudem können die Namen der Komponenten, Schnittstellen und Methoden im Architekturmodell weniger mit den Namen in der neueren Quelltextversion übereinstimmen. Zum Beispiel gibt es im detaillierten *TEAMMATES*-Architekturmodell die Komponente *RemoteApi*. Im Goldstandard ist in der alten konsistenteren Quelltextversion dieser Komponente die Klasse *RemoteApi* zugeordnet. In der neueren inkonsistenteren Quelltextversion wurde diese Klasse zu *Datastore* umbenannt. Damit stimmt der Name nicht mehr mit dem Komponentennamen überein. Die entsprechende Nachverfolgbarkeitsverbindung kann von den Heuristiken nicht mehr gefunden werden. Eine weitere die Ergebnisse verschlechternde Quelltextänderung kann sein, dass die Paketstruktur weniger stark mit der Komponentenstruktur übereinstimmt. Zum Beispiel gibt es im detaillierten Architekturmodell von *TEAMMATES* die Komponente *LogicApi*. Im Goldstandard ist dieser Komponente die Klasse *GateKeeper* zugeordnet. In der alten konsistenteren Quelltextversion befindet sich die Klasse im Paket *teammates.logic.api*. In der neueren, inkonsistenteren Quelltextversion wurde die Klasse ins Paket *teammates.ui.webapi* verschoben.

7.7. Gefährdung der Validität

Es lassen sich mehrere Aspekte identifizieren, die die Validität gefährden. Der Goldstandard wurde eigens für diese Arbeit angefertigt. Eine Benutzerstudie wurde dabei nicht durchgeführt. Verschiedene Personen können jedoch unterschiedliche Ansichten darüber haben, welche Nachverfolgbarkeitsverbindungen korrekt sind. Das gilt besonders für bestimmte, nicht komplett eindeutige Einzelfälle. Sowohl der Goldstandard als auch die Heuristiken wurden nur vom Ersteller dieser Arbeit definiert. Damit liegt dem Goldstandard und den

Heuristiken dieselbe Ansicht über korrekte Nachverfolgbarkeitsverbindungen zugrunde. Das kann die Evaluationsergebnisse verbessern.

Es wurden mit Java und Shell nur zwei Programmiersprachen untersucht. Das kann die Aussagekraft der Evaluationsergebnisse für andere Programmiersprachen einschränken. Allerdings sind Java und Shell zumindest recht verschiedene Programmiersprachen. Java ist eine objektorientierte Sprache, während Shell eine Skriptsprache ist. Zudem bietet nur Java die Aufteilung von Kompilierungseinheiten in Pakete.

Es wurden in dieser Arbeit nur fünf verschiedene Fallstudien für die Evaluation verwendet. Das kann die Aussagekraft der Evaluationsergebnisse für andere Fallstudien beeinträchtigen. Insbesondere ähneln sich Paketstruktur und Komponentenaufteilung oft sehr. Auch im Quelltext und im Architekturmodell verwendete Namen ähneln sich oft sehr. Für andere Fallstudien ist das nicht unbedingt der Fall. Dass eine geringere Übereinstimmung von Struktur und Namensgebung die Ergebnisse negativ beeinflussen kann, wurde in Abschnitt 7.6 gezeigt. Für andere Fallstudien mit geringerer Namensähnlichkeit könnten zum Beispiel weniger strenge Namensvergleiche bessere Ergebnisse erzielen.

8. Zusammenfassung und Ausblick

Bei der Entwicklung von Softwaresystemen fallen viele Artefakte wie Architekturmodelle und Quelltext an. Nachverfolgbarkeitsverbindungen zwischen den Artefakten können das Wissen über ein System erweitern und in der Softwareentwicklung helfen [16]. Die Erstellung von Nachverfolgbarkeitsverbindungen kann manuell erfolgen, was jedoch aufwendig, fehlerbehaftet und teuer ist. Aufgrund des Erstellungsaufwands existieren in Softwareprojekten oft keine oder nur unvollständige Nachverfolgbarkeitsinformationen [11].

Diese Masterarbeit hatte daher das Ziel, automatisiert Nachverfolgbarkeitsverbindungen zwischen Architekturmodellen und Quelltext zu generieren. Modelle und Quelltext müssen dabei keine besonderen Konsistenzeigenschaften erfüllen. Im Gegensatz zu anderen Ansätzen [12, 9] benötigt der Ansatz dieser Arbeit auch keine manuell erstellten initialen Nachverfolgbarkeitsverbindungen als Grundlage. Zudem werden nicht nur für Komponenten, sondern auch für Architekturschnittstellen Nachverfolgbarkeitsverbindungen generiert. Der für dieser Arbeit gewählte Ansatz besteht aus zwei aufeinanderfolgenden Schritten.

Im ersten Schritt, dem Modellgenerierungsschritt, werden aus den vorliegenden Artefakten Modelle erstellt. Diese Modelle sind unabhängig von einer konkreten Programmiersprache oder einem konkreten Architektur-Metamodell. Damit soll die Erstellung von Nachverfolgbarkeitsverbindungen für verschiedene Programmiersprachen und Architektur-Metamodelle vereinheitlicht werden. Zur Verwirklichung dieses Schrittes wurden Metamodelle erstellt, die auf dem von der OMG spezifizierten KDM [1] sowie PCM basieren. Für die Modellerstellung werden entsprechende Extrahierer für die Architekturmetamodelle PCM und UML sowie die Programmiersprachen Java und Shell genutzt.

Die somit erhaltenen Modelle dienen als Grundlage für den zweiten Schritt, den TraceLink-Generierungsschritt. Für diesen Schritt wurden verschiedene Heuristiken und Aggregationen definiert, die die Nachverfolgbarkeitsverbindungen zwischen den Elementen der Modelle generieren. Bei der Berechnung ordnen Heuristiken und Aggregationen Kandidaten für Nachverfolgbarkeitsverbindungen Konfidenzen zu. Mit einer Konfidenz kann die Sicherheit bewertet werden, dass es sich bei einem Kandidaten tatsächlich um eine Nachverfolgbarkeitsverbindung handelt. Die Heuristiken nutzen zum Beispiel Paket-, Pfad-, Namen- und Methoden-Informationen. Andere Heuristiken nutzen Beziehungen zwischen Modellelementen wie die *extends*-Beziehungen im Quelltextmodell oder die Beziehungen zwischen Komponenten und Schnittstellen im Architekturmodell. Aggregiert wird beispielsweise, indem jeder Architektur-Endpunkt den Quelltext-Endpunkten mit der höchsten Konfidenz zugeordnet wird.

In der Evaluation wurden die einzelnen Heuristiken und Aggregationen evaluiert. Dafür wurde ein Goldstandard mit fünf Fallstudien erstellt. Für manche der Fallstudien wur-

den auch mehrere Architekturmodellversionen sowie Quelltextversionen betrachtet. Im Goldstandard treten sowohl Java- als auch Shell-Dateien auf.

Es wurden verschiedene Vorverarbeitungsmethoden untersucht. Die Ergebnisse bei der Verwendung von Lemmatisierung sind dabei leicht besser als die Ergebnisse ohne Vorverarbeitung. Dasselbe gilt für die Stammformreduktion im Vergleich zur Lemmatisierung. Insgesamt sind die Unterschiede aber sehr gering.

Für Architekturschnittstellen wurde festgestellt, dass eine Aggregation von Namen- und Methoden-Hinweisen gute Ergebnisse liefert. Die Namen- und Methoden-Heuristiken liefern für unterschiedliche Fallstudien bessere Ergebnisse. Die Aggregation verbessert die Einzelergebnisse der Namen- und Methoden-Heuristiken. So wird ein Mikro-Durchschnitt des F_1-Maßes von 89,01 % erreicht. In diesen Wert fließt jeder Kandidat für eine Nachverfolgbarkeitsverbindung in gleichem Maße ein.

Für Komponenten liefert die Paket-Heuristik allein schon sehr gute Ergebnisse. Das gilt besonders für grobgranulare Architekturmodelle. Durch Kombination mit weiteren Heuristiken können die Ergebnisse verbessert werden. Dabei werden neben Namen- und Pfad-Hinweisen zum Beispiel die *extends*-Beziehungen genutzt. Auch die gefundenen Hinweise für die Architekturschnittstellen werden in Verbindung mit Architekturmodell-Beziehungen zwischen Komponenten und Schnittstellen verwendet, um die Ergebnisse zu verbessern. Insgesamt beträgt der Mikro-Durchschnitt des F_1-Maßes für die Komponenten 99,42 %. Mit den gewählten Heuristiken und Aggregationen kann also ein sehr gutes Ergebnis erzielt werden.

Werden Komponenten und Schnittstellen gemeinsam betrachtet, wird ein Mikro-Durchschnitt des F_1-Maßes von 99,11 % erreicht. Der Makro-Durchschnitt des F_1-Maßes über alle Architektur-Endpunkte beträgt 93,71 %. In diesen Wert fließt nicht jeder Kandidat für eine Nachverfolgbarkeitsverbindung, sondern jeder Architektur-Endpunkt in gleichem Maße ein. Für alle Fallstudien werden sehr gute Ergebnisse erzielt.

Für die *TEAMMATES*-Fallstudie wurde mithilfe mehrerer Quelltextversionen der Einfluss der Konsistenz auf die Evaluationsergebnisse untersucht. Die ältere und konsistentere Version erzielt im Vergleich bessere Ergebnisse als die neuere und inkonsistentere Version. Der Mikro-Durchschnitt des F_1-Maßes ist um 6,05 Prozentpunkte höher. Der Grund für schlechtere Werte einer neueren Quelltextversion kann zum Beispiel sein, dass manche Architekturelemente gar nicht mehr im Quelltext vorhanden sind. Zudem können Namen oder Paket- und Komponentenstruktur weniger übereinstimmen.

Insgesamt kann also festgestellt werden, dass mit dem Ansatz dieser Arbeit sehr gute Ergebnisse erzielt werden können. Es wurde aber auch gezeigt, dass die Ergebnisse schlechter werden, wenn Architekturmodell und Quelltext weniger konsistent sind. Die Qualität der Ergebnisse hängt somit auch davon ab, wie sehr Architekturmodell und Quelltext sich in Eigenschaften wie Namen oder Struktur ähneln.

Der Ansatz dieser Arbeit kann in folgenden Arbeiten noch erweitert werden. Zum Beispiel können neue Heuristiken definiert werden, die neue Informationsquellen nutzen. Solche weiteren Informationsquellen können zum Beispiel Quelltextkommentare oder Architekturdokumentationen sein. Dabei können dann Methoden zur Verarbeitung natürlicher Sprache verwendet werden. Auch die Metamodelle können in Zukunft erweitert werden, um neue Aspekte des Quelltextes oder der Architektur zu erfassen. Dafür können zum Beispiel weitere Elemente aus dem KDM übernommen werden. Eine Möglichkeit ist

das Hinzufügen von Methodenparametern und Rückgabetypen. Dafür können neue Arten von Datentypen wie primitive Typen als Datentyp-Unterklassen hinzugefügt werden. Es können also neue Unterklassen oder Beziehungen in die Metamodelle aufgenommen werden. Die existierenden Strukturen müssen somit nicht grundlegend verändert werden. Die Heuristiken oder Aggregationsmethoden müssen dann auch nicht an die Erweiterungen angepasst werden. Es müssen aber neue Heuristiken erstellt werden, um die durch die Modellerweiterungen neu hinzugefügten Informationen bei der Generierung der Nachverfolgbarkeitsverbindungen zu nutzen. Die Ergebnisse für die in dieser Masterarbeit untersuchten Fallstudien sind schon hervorragend. Durch die beschriebenen Erweiterungen könnten aber zum Beispiel auch die Ergebnisse für inkonsistentere Fallstudien verbessert werden.

Zudem kann die Quelltextmodellerstellung in Zukunft noch um neue Programmiersprachen erweitert werden. Die Generierung der Nachverfolgbarkeitsverbindungen arbeitet auf den im Modellgenerierungsschritt erstellten, von konkreten Programmiersprachen unabhängigen Modellen. Somit müssen die Heuristiken oder Aggregationsmethoden nicht an die neuen Programmiersprachen angepasst werden. Die den Heuristiken und Aggregationsmethoden zugrunde liegenden Datenstrukturen bleiben schließlich weiter bestehen. Da die Metamodelle in dieser Arbeit auf KDM basieren, können für die Unterstützung neuer Programmiersprachen Ansätze, die KDM-Modelle erzeugen, verwendet werden. Solche Ansätze gibt es zum Beispiel für Cobol [3], PHP [28] und C# [30]. Mit einer entsprechenden Erweiterung des Goldstandards kann der Ansatz dieser Arbeit dann auch für die neuen Programmiersprachen evaluiert werden.

Allgemein kann der Goldstandard in Zukunft noch um weitere Fallstudien ergänzt werden. Das kann die Aussagekraft der Evaluationsergebnisse verbessern. Insbesondere inkonsistentere Fallstudien können für zukünftige Evaluation hinzugefügt werden. Dass eine geringere Übereinstimmung von Struktur und Namensgebung die Ergebnisse negativ beeinflussen kann, wurde für *TEAMMATES* gezeigt. Für Fallstudien mit geringerer Namensähnlichkeit könnten zum Beispiel andere weniger strenge Namensvergleiche untersucht werden.

Verschiedene Personen können unterschiedliche Ansichten darüber haben, welche Nachverfolgbarkeitsverbindungen korrekt sind. Das gilt besonders für bestimmte, nicht komplett eindeutige Einzelfälle. Der Goldstandard kann daher in Zukunft überprüft und verbessert werden, indem zum Beispiel eine Benutzerstudie durchgeführt wird. Dabei kann untersucht werden, wie sehr die von den Studienteilnehmern erstellten Nachverfolgbarkeitsverbindungen variieren.

Andere Arbeiten befassen sich mit der Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Architekturdokumentationen und Architekturmodellen [19]. Solche Arbeiten können mit dieser Masterarbeit kombiniert werden. So können die erzeugten Nachverfolgbarkeitsverbindungen zwischen Dokumentation und Modell und die erzeugten Verbindungen zwischen Modell und Quelltext aneinandergereiht werden, um Verbindungen zwischen Dokumentation und Quelltext zu erhalten. Dieser Ansatz kann dann mit Ansätzen verglichen werden, die direkt Nachverfolgbarkeitsverbindungen zwischen Architekturdokumentation und Quelltext erstellen.

Literatur

[1] Object Management Group (OMG). *Knowledge Discovery Metamodel (KDM) Specification, Version 1.4*. https://www.omg.org/spec/KDM/1.4/PDF. (letzter Zugriff am 10.03.2023). 2006.

[2] Craig Anslow u. a. „Visualizing the Word Structure of Java Class Names“. In: *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. Nashville TN USA: ACM, Okt. 2008, S. 777–778. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449857.

[3] F. Barbier u. a. „Model Driven Reverse Engineering: Increasing Legacy Technology Independence“. In: *International Symposium on Electronic Commerce*. Feb. 2011. (Besucht am 16. 03. 2023).

[4] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. Third edition. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 978-0-13-294277-5.

[5] Steffen Becker u. a. „Reverse Engineering Component Models for Quality Predictions“. In: *2010 14th European Conference on Software Maintenance and Reengineering*. Madrid: IEEE, März 2010, S. 194–197. ISBN: 978-1-61284-369-8. DOI: 10.1109/CSMR.2010.34.

[6] *BigBlueButton*. https://github.com/bigbluebutton/bigbluebutton. (letzter Zugriff am 05.03.2023).

[7] Hugo Bruneliere u. a. „MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering“. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering - ASE '10*. Antwerp, Belgium: ACM Press, 2010, S. 173. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859032.

[8] Sofia Charalampidou u. a. „Empirical Studies on Software Traceability: A Mapping Study“. In: *Journal of Software: Evolution and Process* 33.2 (Feb. 2021). ISSN: 2047-7473. DOI: 10.1002/smr.2294.

[9] Andreas Christl, Rainer Koschke und Margaret-Anne Storey. „Automated Clustering to Support the Reflexion Method“. In: *Information and Software Technology*. 12th Working Conference on Reverse Engineering 49.3 (März 2007), S. 255–274. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2006.10.015.

[10] Jane Cleland-Huang, Orlena Gotel und Andrea Zisman, Hrsg. *Software and Systems Traceability*. London: Springer, 2012. ISBN: 978-1-4471-2238-8 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5.

[11] Jane Cleland-Huang u. a. „Software Traceability: Trends and Future Directions“. In: *Future of Software Engineering Proceedings*. FOSE 2014. New York, NY, USA: Association for Computing Machinery, Mai 2014, S. 55–69. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593891.

[12] Alexander Florean u. a. „A Comparison of Machine Learning-Based Text Classifiers for Mapping Source Code to Architectural Modules“. In: *15th European Conference on Software Architecture*. Bd. 2978. CEUR Workshop Proceedings. CEUR-WS, 2021.

[13] Birgit Grammel, Stefan Kastenholz und Konrad Voigt. „Model Matching for Trace Link Generation in Model-Driven Software Development“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von David Hutchison u. a. Bd. 7590. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 609–625. ISBN: 978-3-642-33665-2 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_39.

[14] J.H. Hayes, A. Dekhtyar und S.K. Sundaram. „Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods“. In: *IEEE Transactions on Software Engineering* 32.1 (Jan. 2006), S. 4–19. ISSN: 1939-3520. DOI: 10.1109/TSE.2006.3.

[15] Florian Heidenreich u. a. „Closing the Gap between Modelling and Java“. In: *Proceedings of the Second International Conference on Software Language Engineering*. SLE'09. Berlin, Heidelberg: Springer-Verlag, Okt. 2009, S. 374–383. ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4_25.

[16] Suhaimi Ibrahim, Malcolm Munro und Aziz Deraman. „A Requirements Traceability to Support Change Impact Analysis“. In: *Asian Journal of Information Technology* 4 (Jan. 2005), S. 335–344.

[17] „ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary“. In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), S. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[18] *JabRef*. https://github.com/JabRef/jabref. (letzter Zugriff am 05.03.2023).

[19] Jan Keim u. a. „Trace Link Recovery for Software Architecture Documentation“. In: *Software Architecture*. Hrsg. von Stefan Biffl u. a. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, S. 101–116. ISBN: 978-3-030-86044-8. DOI: 10.1007/978-3-030-86044-8_7.

[20] Heiko Klare und Joshua Gleitze. „Commonalities for Preserving Consistency of Multiple Models“. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany: IEEE, Sep. 2019, S. 371–378. ISBN: 978-1-72815-125-0. DOI: 10.1109/MODELS-C.2019.00058.

[21] Heiko Klare u. a. „Enabling Consistency in View-Based System Development — The Vitruvius Approach“. In: *Journal of Systems and Software* 171 (Jan. 2021). ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110815.

[22] Tobias Olsson, Morgan Ericsson und Anna Wingkvist. „Semi-Automatic Mapping of Source Code Using Naive Bayes“. In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. Paris France: ACM, Sep. 2019, S. 209–216. ISBN: 978-1-4503-7142-1. DOI: 10.1145/3344948.3344984.

[23] Ralf H. Reussner u. a., Hrsg. *Modeling and Simulating Software Architectures – the Palladio Approach*. MIT Press, 2016. ISBN: 978-0-262-03476-0.

[24] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language Reference Manual*. 2nd ed. The Addison-Wesley Object Technology Series. Boston: Addison-Wesley, 2005. ISBN: 978-0-321-24562-5.

[25] Zipani Tom Sinkala und Sebastian Herold. „InMap: Automated Interactive Code-to-Architecture Mapping“. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. New York, NY, USA: Association for Computing Machinery, Apr. 2021, S. 1439–1442. ISBN: 978-1-4503-8104-8. DOI: 10.1145/3412841.3442124.

[26] Thomas Stahl und Markus Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt-Verlag, 2005. ISBN: 978-3-89864-310-8.

[27] *TEAMMATES*. https://github.com/TEAMMATES/teammates. (letzter Zugriff am 05.03.2023).

[28] F. Trias u. a. „A Toolkit for ADM-based Migration: Moving from PHP Code to KDM Model in the Context of CMS-based Web Applications“. In: *Integrated Spatial Databases*. 2014. (Besucht am 16. 03. 2023).

[29] Jóakim von Kistowski u. a. „TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research“. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sep. 2018, S. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[30] Christian Wulf, Sören Frey und Wilhelm Hasselbring. *A Three-Phase Approach to Efficiently Transform C# into KDM*. Techn. Ber. TR-1211. Department of Computer Science, Kiel University, Germany, Aug. 2012.

A. Anhang

A.1. Weitere Evaluationsergebnisse

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,6225	0,9947	0,7658	0,6026	0,9960	0,6658
	V2	0,5620	0,9936	0,7180	0,5455	0,9952	0,5890
TEAMMATES	det.-kons.	0,3269	0,4190	0,3672	0,4080	0,6429	0,4400
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		0,8889	0,7619	0,8205	0,7576	0,8182	0,7818
JabRef		0,9650	0,9990	0,9817	0,7008	0,8331	0,7232
BigBlueButton		0,7991	0,4973	0,6131	0,3896	0,4134	0,3723
mikro/alle Arch.-Endp.		0,8313	0,8896	0,8594	0,5941	0,8006	0,6210
makro ü. Versionen		0,7378	0,8093	0,7523	0,6291	0,8141	0,6532

Tabelle A.1.: Die Ergebnisse der Paket-Heuristik ohne Vorverarbeitung für Architekturkomponenten

		einzelne Kandidaten			makro ü. Arch.-Endp.		
		Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
TeaStore	V1	0,6225	0,9947	0,7658	0,6026	0,9960	0,6658
	V2	0,5620	0,9936	0,7180	0,5455	0,9952	0,5890
TEAMMATES	det.-kons.	0,3340	0,4464	0,3821	0,4795	0,7143	0,5115
	grob	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
MediaStore		0,8947	0,8095	0,8500	0,8485	0,9091	0,8727
JabRef		0,9650	0,9990	0,9817	0,7008	0,8331	0,7232
BigBlueButton		0,7439	0,4973	0,5961	0,3712	0,4134	0,3640
mikro/alle Arch.-Endp.		0,8262	0,8927	0,8581	0,6187	0,8280	0,6471
makro ü. Versionen		0,7317	0,8201	0,7562	0,6497	0,8373	0,6752

Tabelle A.2.: Die Ergebnisse der Paket-Heuristik mit Stammformreduktion für Architekturkomponenten

	einzelne Kandidaten			makro ü. Arch.-Endp.		
	Präz.	Ausb.	F_1	Präz.	Ausb.	F_1
vor den MatchBest-Aggregationen						
mikro/alle Arch.-Endp.	0,2902	0,0499	0,0852	0,0960	0,0877	0,0806
makro ü. Versionen	0,0415	0,0749	0,0534	0,0910	0,0832	0,0764
nach den MatchBest-Aggregationen						
mikro/alle Arch.-Endp.	0,9824	0,0432	0,0828	0,1096	0,0772	0,0829
makro ü. Versionen	0,1403	0,0648	0,0887	0,1039	0,0732	0,0786

Tabelle A.3.: Die Ergebnisse der Pfad-Heuristik für Architekturkomponenten vor und nach den MatchBest-Aggregationen