

Erkennung von semantisch zusammenhängenden Quelltextabschnitten anhand von Komponententests

Bachelorarbeit
von

Martin Wittlinger

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

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf Reussner
Betreuender Mitarbeiter:	M.Sc. Tobias Hey

Bearbeitungszeit: 16.Juli 2019 – 15.November 2019

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

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 15.November 2019

Martin Wittlinger

(Martin Wittlinger)

Publikationsgenehmigung

Melder der Publikation

Hildegard Sauer

Institut für Programmstrukturen und Datenorganisation (IPD)

Lehrstuhl für Programmiersysteme

Leiter Prof. Dr. Walter F. Tichy

+49 721 608-43934

hildegard.sauer@kit.edu

Erklärung des Verfassers

Ich räume dem Karlsruher Institut für Technologie (KIT) dauerhaft ein einfaches Nutzungsrecht für die Bereitstellung einer elektronischen Fassung meiner Publikation auf dem zentralen Dokumentenserver des KIT ein.

Ich bin Inhaber aller Rechte an dem Werk; Ansprüche Dritter sind davon nicht berührt.

Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

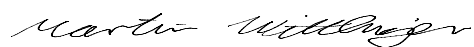
Art der Abschlussarbeit: Bachelorarbeit

Titel: Erkennung von semantisch zusammenhängenden Quelltextabschnitten anhand von Komponententests

Datum: 15.November 2019

Name: Martin Wittlinger

Karlsruhe, 15.November 2019



(Martin Wittlinger)

Kurzfassung

Die Rückverfolgbarkeit von Quelltext zu Anforderungen ist ein wichtiger werdendes Problem. Eine Garantie der Implementierung aller Anforderungen kann zur Steigerung von Softwarequalität führen. Für das Erstellen der Rückverfolgbarkeitsinformationen ist ein Verständnis des Quelltextes nötig. In dieser Arbeit wurden anhand von Komponententests semantisch zusammenhängende Methoden erkannt. Semantisch zusammenhängende Methoden erfüllen eine Funktionalität miteinander und verbessern das Verständnis von Quelltext. Für die Erkennung wurde ein heuristisches Verfahren entwickelt, welches aus mehreren Teilverfahren besteht, die sowohl auf den textuellen als auch den strukturellen Bestandteilen des Komponententest- und Quelltextes arbeiten. Für die Teilverfahren wurde eine Zerteilung und Transformation von Quelltext entwickelt. Es wurden verschiedene Textähnlichkeitsalgorithmen mit einem maschinellen Lernverfahren (fastText) verglichen. Zur Bewertung wurden drei Softwareprojekte verwendet, mit einer höchsten Präzision von 74%, bei einer Ausbeute von 19%. Mit einer anderen Parameterkonfiguration wurde ein F1-Wert von 46% erreicht.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Zielsetzung	1
1.2	Aufbau der Arbeit	1
2	Grundlagen	3
2.1	Komponententests	3
2.1.1	Testrahmenarchitektur JUnit	3
2.1.2	Regressionstests	4
2.2	Quelltextanalysemethoden	4
2.2.1	Statische und Dynamische Analyse	4
2.2.2	Aufrufgraph	5
2.2.3	Zyklomatische Komplexität	5
2.3	Sprachwerkzeuge	5
2.3.1	Stammformreduktion und Lemmatisierung	5
2.3.2	Zeichenkettenähnlichkeitsalgorithmen	6
2.3.3	Worteinbettung	8
2.4	Grundlegende Komponententest zu Quelltext Rückverfolgbarkeitsmethoden	9
2.4.1	Klassenobjektanalyse	9
2.4.2	Aufrufgraphanalyse	9
2.4.3	Letzter Aufruf vor Überprüfung	10
2.4.4	Datenflussanalyse	10
2.4.5	Benennungsregeln	10
2.4.6	Lexikalische Analyse	10
2.4.7	Co-Entwicklung	11
3	INDIRECT	13
3.1	Problemstellung und Lösungsansatz	13
3.2	Arbeiten in dem Projekt	13
4	Verwandte Arbeiten	15
4.1	Komponententest zu Quelltext Rückverfolgbarkeit	15
4.2	Andere Themengebiete	17
5	Analyse und Entwurf	21
5.1	Zielsetzung	22
5.2	Analyse der vorhandene Informationen	22
5.3	Analyse der Informationsquellen aus dem Quelltext	24
5.3.1	Paketbezeichner	24
5.3.2	Importanweisung	24
5.3.3	Klassenbezeichner	24
5.3.4	Klassenattribute	25
5.3.5	Dokumentation	25

5.3.6	Methode	27
5.3.7	Überprüfungen	28
5.3.8	Auf- und Abbaumethode	29
5.4	Zusammenfassung der gewonnen Informationen aus den einzelnen Informationsquellen	31
5.5	Auswertungsverfahren der einzelnen Informationsquellen	32
5.6	Entwurf des Gesamtverfahrens	36
6	Implementierung	37
6.1	Zerteilen und Transformieren des Quelltextes	37
6.2	Gewinnung der Informationen aus den Quelltextabschnitten	39
6.3	Auswertung der Informationen	41
7	Evaluation	43
7.1	Kriterien der Evaluation	43
7.2	Datenmenge	44
7.2.1	JPacman	44
7.2.2	Roller	45
7.2.3	Gitiles	45
7.3	Ergebnisse	45
7.3.1	JPacman	46
7.3.2	Roller	47
7.3.3	Gitiles	52
7.3.4	Gesamtergebnis	54
7.4	Fazit	55
8	Zusammenfassung und Ausblick	57
	Literaturverzeichnis	59
	Anhang	63
A	First Appendix Section	63
B	Literaturanalysen	63
B.1	Recovering traceability links between unit tests and classes under test: An improved method	64
B.2	Establishing Traceability Links between Unit Test Cases and Units under Test	66
B.3	Traceclipse: An eclipse plug-in for traceability link recovery and management	68
B.4	Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects	70
B.5	Automatically identifying focal methods under test in unit test cases	72
B.6	Towards Gamification in Software Traceability: Between Test and Code Artifacts	74
B.7	Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes	76
B.8	Analysis of Static and Dynamic Test-to-code Traceability Information	77
B.9	Evaluation of Textual Similarity Techniques in Code Level Traceability	79
B.10	Identification of High-Level Concept Clones in Source Code	81
B.11	Are my Unit Tests in the Right Package?	83
B.12	Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability	84

Abbildungsverzeichnis

3.1	Darstellung des prinzipiellen Vorgehens von <code>INDIRECT</code>	14
5.1	Verfahren zur Auswertung des Paketbezeichners einer Testklasse.	32
5.2	Verfahren zur Auswertung der Importanweisungen einer Testklasse.	33
5.3	Verfahren zur Auswertung der Klassennamen einer Testklasse.	33
5.4	Verfahren zur Auswertung der Methodennamen einer Testklasse.	34
5.5	Verfahren zur Auswertung der Methodenaufrufe einer Testmethode.	35
5.6	Verfahren zur Auswertung der Überprüfungen einer Testmethode.	35
5.7	Verfahren zur Auswertung der Zuweisungen einer Aufbaumethode.	35
6.1	Schematische Darstellung des Quelltextbaumes	38
6.2	Schematische Darstellung des Zerteilungsvorgangs	38
6.3	Schematische Darstellung des Datenhaltung	39
6.4	Schematische Darstellung der Informationsgewinnung	40
7.1	Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von <code>JPacman</code> in Abhängigkeit des Schwellwertes	46
7.2	Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von <code>Roller</code> in Abhängigkeit des Schwellwertes	48
7.3	Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von <code>Gitiles</code> in Abhängigkeit des Schwellwertes	53
7.4	Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von allen drei Projekten in Abhängigkeit des Schwellwertes	55

Tabellenverzeichnis

2.1	Verschiedene Test zu Quelltext Verbindungsmethoden und ihre Eigenschaften.	9
4.1	Verschiedene evaluierte Komponententest zu Quelltext Zuordnungsverfahren	19
5.1	Vorhandene Informationen im Quelltext, ihre Art und ihre Quelle	31
5.2	Rückschlüsse aus den Information nach der Auswertung	32
6.1	Rückschlüsse mit den Konfidenzwerte im Auswertungsverfahren	41
7.1	Programmgröße von JPacman aufgeteilt in Testteil und Hauptteil	44
7.2	Programmgröße von Roller aufgeteilt in Testteil und Hauptteil	45
7.3	Programmgröße von Gitiles aufgeteilt in Testteil und Hauptteil	45
7.4	Anzahl an Methodenpaaren der Ergebnismenge auf JPacman mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß	46
7.5	Anzahl an Methodenpaaren mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß	48
7.6	Punkte aus Abbildung 7.2 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß	49
7.7	Teilverfahren und verschiedene Ähnlichkeitsalgorithmen angewandt auf Roller mit ihrer Präzision, Ausbeute und F-Maß bei drei verschiedenen Schwellwerten.	50
7.8	Anzahl an Methodenpaaren mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß	53
7.9	Punkte aus Abbildung 7.3 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß	53
7.10	Punkte aus Abbildung 7.4 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß	55

Quelltextausschnitteverzeichnis

5.1	Beispiel eines Komponententests	21
5.2	Beispiel einer Testklasse mit farblich hervorgehobenen Informationsquellen .	23
6.1	Ergebnisse der Matcher	41
7.1	Testfall der zu Fehlern innerhalb von JPacman führt	47
7.2	Gekürzter Testfall aus Roller bei dem das Verfahren falsche Ergebnisse produziert.	52

1 Einleitung

Aus dem heutigen Leben ist Software nicht mehr wegzudenken und findet immer mehr Anwendung in kritischen Bereichen wie Infrastruktur oder Medizin. Deswegen wird die Qualität von Software immer wichtiger. Um dies zu gewährleisten gibt es verschiedene Verfahren wie Testen, formale Verifikation und Quelltextanalysen. Die Gewährleistung, dass alle gewünschten Anforderungen umgesetzt sind, ist schwer möglich. Einerseits wegen der natürlichen Sprache, in der die Anforderungen formuliert sind und andererseits weil Software sich wandelt und weiterentwickelt wird. Die Analyse von semantisch zusammenhängenden Quelltextabschnitten bietet eine Möglichkeit zur Verbesserung der automatisierten Zuordnung dieser. Durch eine automatisierte Zuordnung von Quelltextabschnitten zu Anforderungen wäre eine Verifikation der Implementierung der gewünschten Anforderungen möglich. Diese Zuordnung wird vom Projekt INDIRECT, worin sich diese Arbeit befindet, umgesetzt. INDIRECT gewinnt ein besseres Quelltextverständnis mit diesen zusammengehörigen Methoden. Aktuell gibt es aber noch kein Verfahren, das vollautomatisch eine zufriedenstellende Präzision und Ausbeute besitzt. Weil zusammenhängende Quelltextabschnitte zusammen getestet werden sollten, sollten Testfälle eine Informationsquelle für zusammengehörige Quelltextabschnitte sein. Komponententests [MLBK02] bieten sich deswegen an, weil sich Quelltextänderungen an ihnen widerspiegeln. Jede Quelltextänderung, die das Verhalten des Quelltexts ändert, bedingt eine Änderung der Komponententests. Somit können Komponententests eine aktuelle Dokumentation des Quelltexts sein.

1.1 Zielsetzung

Das Ziel der Arbeit, die sich in dem Project INDIRECT befindet, ist das Entwerfen und Evaluieren eines Verfahrens zum Finden von semantisch zusammenhängenden Methoden. Das Verfahren soll hierfür Informationen aus Testfällen gewinnen und auswerten. Eine adaptive Anpassung an die zur Verfügung stehenden oder fehlende Informationen soll sowohl manuell also auch automatisch möglich sein. Mit diesen Informationen soll das Quelltextverständnis von INDIRECT verbessert werden.

1.2 Aufbau der Arbeit

In der folgenden Arbeit wird die Erkennung von semantisch zusammengehörigen Methoden anhand von Komponententests behandelt. Hierfür werden in Kapitel 2 erst die notwendigen Grundlagen für dieses Thema erklärt. Dann wird das Projekt INDIRECT, in dessen

Rahmen diese Arbeit umgesetzt wird, in Kapitel 3 vorgestellt. Danach werden verwandte Arbeiten zu dem Thema der Test-zu-Quelltext Rückverfolgung in Kapitel 4 analysiert. Als nächstes werden in Kapitel 5 Komponententestklassen, ihre Informationen und die daraus (rück)gewinnbaren Informationen analysiert. Mittels diesen Informationen wird dann im Entwurf ein Verfahren erstellt, was daraus semantisch zusammengehörige Methoden erkennt. In der Implementierung in Kapitel 6 wird die Umsetzung des Verfahrens vorgestellt und in der Evaluation in Kapitel 7 wird das Verfahren bewertet und verglichen. Am Ende der Arbeit wird in Kapitel 8 eine Ausblick für mögliche zukünftige Arbeiten gegeben.

2 Grundlagen

In diesem Kapitel werden verschiedene Grundlagen erläutert, die für das spätere Verständnis wichtig sind. Erst wird das Hauptthema Komponententests und dann Quelltextanalysemethoden erläutert. Danach werden Sprachwerkzeuge wie Stammformreduktion und am Ende Grundlegende Test-zu-getestete Klasse Verfahren erläutert.

2.1 Komponententests

Komponententests dienen dazu die Funktionalität eines Programms zu überprüfen. Der Umfang eines Komponententests ist eine einzelne Funktionalität eines Programms. Durch Komponententests soll bewiesen werden, dass diese Funktionalität korrekt implementiert ist. Gute Komponententests erfüllen laut Literatur fünf Eigenschaften. Diese müssen automatisch ablaufen und testen alle Funktionalitäten eines Programms gründlich. Keine Komponententests soll abhängig von anderen Komponententests sein und muss einfach wiederholbar sein. Der Quelltext der Komponententests soll wie der Quelltext des Programms gepflegt und an verändertes Verhalten angepasst werden [HT03]. Um das Schreiben der Komponententests zu erleichtern sind in vielen Programmiersprachen Testhelfer bzw. Rahmenarchitekturen vorhanden. Diese erleichtern die Überprüfung des Zustands und ermöglichen es einheitliche Komponententests zu schreiben. In vielen Sprachen existiert eine sogenannte `Unit` Rahmenarchitektur, im Fall von Java heißt diese z.B. `JUnit`¹. `JUnit` ermöglicht die automatische Ausführung aller Komponententests und bietet Schnittstellen zu anderen APIs, die z.B. kontinuierliche Integration ermöglichen.

2.1.1 Testrahmenarchitektur `JUnit`

Eine Komponententest in `JUnit` besteht aus drei Phasen: Aufbau, Ausführung und Orakel [GGR15]. In der Aufbau-Phase werden Objekte erstellt und der Zustand vor der Ausführung vorbereitet. Dies könnte beispielsweise die Erstellung und Befüllung einer Liste sein. In der Ausführung wird meist die zu testende Methode aufgerufen. Die Ausführung simuliert eine Aktion des Systems. Dies könnte beispielsweise das Entfernen eines Elements aus einer Liste sein. In der Orakelphase findet die Überprüfung statt, ob der Zustand nach dem Zustandsübergang der korrekte ist. Dies könnte z.B. die Überprüfung sein, ob die Größe der Liste kleiner wurde.

¹<https://junit.org/junit5/>

Für diese Phasen stellt JUnit verschiedene Annotationen und Methoden bereit. Mittels der Annotation `@Test` wird dem Testläufer signalisiert, dass eine Methode einen Test darstellt. Für den Aufbau existieren die Annotationen `@Before` und `@BeforeClass`. Diese sorgen dafür, dass die Methode vor jeder Testmethode bzw. vor der gesamten Klassen ausgeführt wird. Für die Überprüfung gibt es verschiedene `assert`-Methoden, die z.B. überprüfen ob eine Zeichenkette auf ein Muster passt. Das Testen auf Ausnahmen ist in JUnit ein Sonderfall. Diese werden mit der `@Test(expected = Exception.class)` Notation über dem Methodenrumpf getestet. Ein Test gilt als fehlgeschlagen, wenn eine Überprüfung nicht erfolgreich war. Über die Reihenfolge der Testausführung trifft JUnit keine Annahmen, außer es wird explizit erwünscht.

Häufig verwendete Überprüfungen sind: `assertTrue` und `assertFalse` zur Überprüfung eines Wahrheitswertes auf wahr oder falsch. `assertNull` und `assertNotNull` zur Überprüfung eines Objektes auf null oder nicht null. `assertSame` und `assertNotSame` zur Überprüfung zweier Objekte auf (nicht) Gleichheit. Dies ist in Java der Vergleich mittels dem `==`-Operator. Im Gegensatz dazu gibt es `assertEquals` was zur Überprüfung zweier Objekte auf Gleichheit die `equals`-Methode verwendet. Für Arrays existiert der Vergleich auf Gleichheit durch die `assertArrayEquals`-Methode. Diese führt für jeden Index der Arrays ein Vergleich aus. Für komplexere Überprüfungen existiert die `assertThat` Methode, welche den Einsatz verschiedener Zuordner(engl. Matcher) erlaubt. Dadurch können z.B. Inhalte von Strings überprüft werden. Für eine vollständige Übersicht, sei auf die Dokumentation von JUnit² verwiesen.

2.1.2 Regressionstests

Regressionstests stellen eine besondere Form von Tests dar. Diese werden nach der Behebung eines Programmfehlers geschrieben, um diesen zukünftig zu verhindern. Deswegen kann ihr Umfang größer als ein Komponententest sein. Ein Regressionstest stellt dann den Zustand des Fehlers sowie den Fehler selber nach und überprüft die Nichtexistenz des Fehlers. Diese Tests sorgen dafür, dass ein Fehler nicht mehr wiederholt wird und werden meist gesondert gesammelt [HT03].

2.2 Quelltextanalysemethoden

In diesem Abschnitt werden Grundlagen der Analyse von Quelltext erklärt. Diese bilden die Grundlage für Verfahren zur Test-zu-Quelltextzuordnung und erleichtern das Verständnis dieser. Es wird erst der Unterschied zwischen dynamischer und statischer Analyse erläutert, dann Aufrufgraphen und am Ende eine Komplexitätsmetrik.

2.2.1 Statische und Dynamische Analyse

Die Analyse von Software ermöglicht es verschiedene Fehler zu finden. Es lassen sich zwei Arten von Analysen unterscheiden, Dynamische und Statische.

Bei der dynamische Quelltextanalyse wird das Programm ausgeführt und überprüft. Dies ermöglicht für bestimmte Nutzereingaben das tatsächliche Verhalten zu überprüfen. Dynamisches Analyseverfahren sind z.B. Debugger und Komponententests. Dafür muss der Quelltext nicht vorliegen, es reicht das Kompilat. Es ist aber erforderlich, dass der Quelltext ausführbar ist [Bal09].

Im Gegensatz dazu steht die statische Quelltextanalyse. Bei der statischen Analyse wird der Quelltext nicht ausgeführt. Es werden verschiedene formale Prüfungen auf dem Quelltext durchgeführt. Beispielsweise in Java die Überprüfung der Typsicherheit von Programmen. Statische Analyse kann sowohl vollautomatisch als auch manuell durchgeführt werden. Nachteil von statischen Analysen ist, dass der Quelltext vorliegen muss [Bal09].

²<https://github.com/junit-team/junit4/wiki/Assertions>, zuletzt besucht am 11.08.2019

2.2.2 Aufrufgraph

Ein Aufrufgraph ist eine Darstellungsform eines Programmflusses. Jeder Knoten stellt eine Methode dar und jede Kante ist der Aufruf einer Methode durch eine andere Methode. Aufrufgraphen können in statische und dynamische Aufrufgraphen unterteilt werden.

Dynamische Aufrufgraphen stellen einen Programmablauf dar. Also alle verwendeten Methoden nach dynamischer Bindung. Statische Aufrufgraphen stellen alle möglichen Abläufe eines Programms dar. Dies bedeutet z.B. alle vorhandenen Implementierungen eines Interfaces [RKG04].

2.2.3 Zyklomatische Komplexität

Die Zyklomatische Komplexität, auch als McCabe-Metrik bekannt, stellt ein Maß für die Komplexität von Quelltext dar. Der Gedanke dahinter ist, dass ab einer bestimmten Komplexität ein Quelltextabschnitt nicht mehr verständlich ist. Die Komplexität lässt sich über verschiedene Methoden berechnen. Die leichteste ist das Zählen aller Verzweigungen im Kontrollflussgraphen. Ab einer Komplexität von höher als zehn gilt ein Quelltextabschnitt als zu komplex und sollte überarbeitet werden. Die Komplexität steigt bei dieser Metrik nur bei Verzweigungen und nicht bei komplexen Methodenaufrufen [McC76].

2.3 Sprachwerkzeuge

In diesem Abschnitt werden verschiedene grundlegende Sprachwerkzeuge kurz erklärt. Sprachwerkzeuge verbessern die Informationsgewinnung aus textuellen Quellen. Es handelt sich dabei sowohl um Algorithmen zur Vorverarbeitung als auch Verfahren zur Auswertung. Diese Algorithmen erleichtern es ein tieferes Verständnis als die textuelle Darstellung zu erlangen.

2.3.1 Stammformreduktion und Lemmatisierung

Bei natürlicher Sprache gibt es mehrere Verfahren zur Vorverarbeitung eines Textes. Lemmatisierung und Stammformreduktion sind zwei davon. Sprache besitzt wegen Flexion bedeutungsäquivalente Wörter, die beispielsweise in verschiedenen Zeitformen sind. Bei einer Suche nach „Demokratie“ ist häufig auch ein Dokument mit „demokratisch“ ein sehr gutes Ergebnis. Beide Verfahren reduzieren diese bedeutungsgleichen Wörter auf eine Stammform. Dadurch wird die Anzahl der Wörter reduziert und Verfahren können besser funktionieren. Es handelt sich um eine Art Kompressionsverfahren. Die Stammform ist eine Grundform eines Wortes. Diese Grundform ist vergleichbar mit dem Wortstamm. Beide Verfahren sind Algorithmen für diese Reduktion, lösen dies aber mit verschiedene Ansätze.[MRS08].

Stammformreduktion ist ein Algorithmus zur Reduzierung einer Zeichenkette auf einen Kern. Dieser Kern muss kein gültiges Wort sein, weil nur die Äquivalenz der Wörter wichtig ist. Ein weit verbreiteter Algorithmus für Stammformreduktion ist der **Porter-Stemmer-Algorithmus**³. Dieser verkürzt ein Wort solange nach vorgegebenen Regeln bis es eine minimale Anzahl von Silben hat. Das Umwandeln des Suffixes „s“ zu „“ ist eine der Regeln. Stammformreduktionsalgorithmen arbeiten nicht mit 100% Genauigkeit und verkürzen in manchen Fällen zu wenig oder zu viel. Dies wird dann **Understemming** und **Overstemming** genannt. Die Folge davon ist, dass äquivalente Wörter als nicht äquivalent angenommen werden und umgekehrt. Dadurch entstehen z.B. bei Suchen falsche Ergebnisse, weil zwei nicht äquivalente Wörter als gleich gelten und somit die Suche zu wenig Ergebnisse liefert [Por80].

³<http://snowball.tartarus.org/algorithms/porter/stemmer.html>

Bei Lemmatisierung handelt es sich um ein fortgeschrittenes Verfahren zur Stammformreduktion. Eine einfache Art Lemmatisierung zu implementieren ist, eine Wörterbuchabfrage. Besser Implementierungen sind regelbasierte Systeme und tiefe Neuronale Netze ⁴. Im Vergleich zur vorherigen Stammformreduktion kann das Verfahren mit unregelmäßigen Wörtern umgehen und die Stammform stellt ein gültiges Wort dar. Ein großer Unterschied ist, dass Lemmatisierung je nach Implementierung den Kontext beachten kann und Stammformreduktion Wortweise arbeitet [MRS08].

2.3.2 Zeichenkettenähnlichkeitsalgorithmen

Für den Vergleich von Zeichenketten existieren verschiedene Algorithmen. Diese werden verwendet, weil ein Vergleich auf exakte Gleichheit nicht immer ein ausreichendes Ergebnis erzielt. Falls beispielsweise Tippfehler in dem Suchwort oder dem Korpus vorhanden sind.

Beispiel 2.1: Beispielsuche in einem Textkorpus mit exakter Gleichheit

Suchwort: Distanz
 Korpus: Metrik, Ähnlichkeit, Distance, Distanzen
 Suchergebnis: \emptyset

In Beispiel 2.1 ist die Suche auf exakte Gleichheit ohne Ergebnis, obwohl sowohl die englische Variante „Distance“ als auch der Plural „Distanzen“ vorhanden sind. Um diese Ähnlichkeit zu messen ist eine Metrik notwendig. Es existieren verschiedene Metriken hierfür, z.B. Distanz und Ähnlichkeit. Die Distanz beschreibt die notwendigen Schritte zur Umwandlung des einen Wortes in ein anderes. Die Ähnlichkeit beschreibt mit einer Zahl wie ähnlich die Zeichenketten sind. Im folgenden werden Levenshtein-Distanz, Fuzzystring, längste gleiche Teilsequenz, Kosinusähnlichkeit und JaroWinkler vorgestellt.

Levenshtein-Distanz

Die Levenshtein-Distanz misst die Distanz zweier Wörter anhand der notwendigen Operationen für die Umformungen des eines Wortes in das andere. Diese grundlegenden Operationen sind das Umwandeln, das Entfernen und das Hinzufügen eines Buchstabens. Jede Operation hat als Kosten eins und es werden die Gesamtkosten addiert. Die Gesamtkosten sind die Distanz [apo]. In Beispiel 2.2 ist diese für den Korpus aus dem einleitend Beispiel dargestellt. Der Plural und die englische Variante besitzen die geringste Distanz.

Beispiel 2.2: Beispielsuche in einem Textkorpus mit minimaler Levenshtein-Distanz

Suchwort: Distanz
 Korpus: Metrik, Ähnlichkeit, Distance, Distanzen
 Zwischenergebnis: Metrik(6), Ähnlichkeit(10), Distance(2), Distanzen(2)
 Suchergebnis: Distance, Distanzen

Längste gleiche Teilsequenz

Die längste gleiche Teilsequenz berechnet die Länge der längsten Teilsequenz zwischen mehreren Wörtern. Hierbei werden die Teilsequenz betrachtet und nicht das längste gleiche

⁴<https://stanfordnlp.github.io/stanfordnlp/>

Teilwort gesucht. Deswegen müssen die Buchstaben nicht im Wort direkt aufeinander folgen. Die Berechnung davon wird über den Vergleich aller Teilwörter erledigt, wobei für optimierte Algorithmen auf [BHR] verwiesen sei. In Beispiel 2.3 ist dies für den Korpus dargestellt.

Beispiel 2.3: Beispielsuche in einem Textkorpus mit maximaler Längster gleicher Teilsequenz

Suchwort: Distanz

Korpus: Metrik, Ähnlichkeit, Distance, Distanzen

Zwischenergebnis: Metrik(1,t), Ähnlichkeit(2,it), Distance(6,Distan),
Distanzen(7,Distanz)

Suchergebnis: Distanzen

Fuzzystring

FuzzyString berechnet einen Wert pro Zeichenkette. Je höher dieser Wert ist desto ähnlicher sind die Worte. Jedes Zeichen, was in beiden Eingaben an der gleichen Stelle gleich ist, gibt einen Punkt. Aufeinanderfolgende Gleichheit gibt mehr Punkte.^{5 6} Beispiel 2.4 stellt die Ergebnisse auf dem Beispielkorpus da.

Beispiel 2.4: Beispielsuche in einem Textkorpus mit FuzzyScore

Suchwort: Distanz

Korpus: Metrik, Ähnlichkeit, Distance, Distanzen

Zwischenergebnis: Metrik(0), Ähnlichkeit(0), Distance(22), Distanzen(16)

Suchergebnis: Distance

Kosinusähnlichkeit

Die Kosinusähnlichkeit zweier Wörter wird in zwei Schritten berechnet. Erst werden beide Wörter in Vektoren umgewandelt und dann mittels der Gleichung 2.1 berechnet. Ein Wert von Eins steht dabei für identisch und Null für verschieden [Kar18]. In Beispiel 2.5 ist die Berechnung für den Beispielkorpus dargestellt.

$$\text{Ähnlichkeit}(a, b) = \cos(\theta) = \frac{a \cdot b}{\|a\| \cdot \|b\|} \quad (2.1)$$

Beispiel 2.5: Beispielsuche in einem Textkorpus mit Cosinusähnlichkeit

Suchwort: Distanz

Korpus: Metrik, Ähnlichkeit, Distance, Distanzen

Zwischenergebnis: Metrik(0,43), Ähnlichkeit(0,54), Distance(0,99), Distanzen(0,85)

Suchergebnis: Distance

JaroWinkler

Die JaroWinkler Ähnlichkeit berechnet die Ähnlichkeit über gleiche Buchstaben an ungefähr der selben Stelle im Wort. Hierfür wird die Gleichung 2.2 verwendet um die Jaro

⁵<https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/FuzzyScore.html>

⁶<https://github.com/facebook-atom/nuclide-prebuilt-libs/tree/master/fuzzy-native>

Ähnlichkeit zu berechnen. A und B sind die Zeichenketten zum vergleichen, die Norm steht für die Länge der Zeichenkette. a' steht für die ähnlichen Zeichen von a in b an ungefähr der selben Stelle, und b' umgekehrt. Das Ungefähr wird mittels der halben Länge der längsten Zeichenkette -1 berechnet.

$$\text{JaroÄhnlichkeit } sim_j(a, b) = \frac{1}{3} \cdot \left(\frac{|a'|}{|a|} + \frac{|b'|}{|b|} + \frac{|a'| - T_{a'b'}}{2 \cdot |b'|} \right) \quad (2.2)$$

Die JaroWinkler Ähnlichkeit erweitert es, indem es noch Paare mit einer längeren gemeinsamen Teilsequenz bevorzugt. In Gleichung 2.3 ist die Berechnung dargestellt, wobei P für die längste gemeinsame Teilsequenz steht [BMC⁺].

$$\text{JaroWinklerÄhnlichkeit } sim_{jw}(a, b) = sim_j(a, b) + \frac{|P|}{10} \cdot (1 - sim_j(a, b)) \quad (2.3)$$

In Beispiel 2.6 ist die Berechnung für den Beispielkorpus dargestellt.

Beispiel 2.6: Beispielsuche in einem Textkorpus mit JaroWinkler

Suchwort: Distanz

Korpus: Metrik, Ähnlichkeit, Distance, Distanzen

Zwischenergebnis: Metrik(0,37), Ähnlichkeit(0,32), Distance(0,92), Distanzen(0,96)

Suchergebnis: Distanzen

2.3.3 Worteinbettung

Worteinbettung ist ein Überbegriff von verschiedenen maschinellen Lernverfahren, die Textfragmente in Vektoren umwandeln. Diese Verfahren lernen den Zusammenhang von Textfragmenten und wandeln diese in ähnliche Vektoren um. Mit diesen Vektoren wird ein mehrdimensionaler Vektorraum aufgespannt. Diese vereinfachte Darstellungsform ermöglicht es verschiedene Operationen auf dem aufgespannten Vektorraum auszuführen [Kar18]. Bekannte Verfahren sind `fastText`, `Word2Vec` und `GloVe`.

Für die Umwandlung von Textfragmenten zu Vektoren existieren verschiedene zugrundeliegende Netzarchitekturen und Verfahren. `fastText` nutzt `Skip-Gram` als Netzarchitektur, welches in „*Efficient Estimation of Word Representations in Vector Space*“ von Mikolov et al. [MCCD13] erstmals veröffentlicht wurde. `Skip-Gram` sagt die vorherigen und folgenden Textfragmente anhand eines einzelnen Textfragmentes vorher. Die Textfragmente werden bei `fastText` durch eine Zerlegung des Wortes in Teilworten mit Größe N erzeugt. Vor und hinter das Wort werden die Sonderzeichen „<“ und „>“ hinzugefügt. Zu den Fragmenten wird noch das gesamte Wort vor der Zerteilung hinzugefügt. In Beispiel 2.7 ist die Zerteilung des Wortes „Vektor“ in die einzelnen Fragmente durchgeführt.

Beispiel 2.7: Zerlegung eines Wortes in Fragmente

Eingabewort: Vektor

Nach Zerlegung zu Fragmenten: <Ve,Vek,ekt,kt,tor,or> und <Vektor>

Nach der Zerteilung wird von jedes Fragment die Vektordarstellung gelernt. Der Vektor eines Wortes ist die Summe der Vektordarstellungen seiner Fragmente. Das Ergebnis des Verfahrens bilden die Gewichtsvektoren und nicht die Ausgabe des Netzes. Durch die einzelnen Fragmente sollen selten Wörter besser gelernt werden [BGJM16].

Tabelle 2.1: Verschiedene Test zu Quelltext Verbindungsmethoden und ihre Eigenschaften.

Methode	Datengrundlage	Vorraussetzungen	Granularität
Klassenobjektanalyse	Klassenobjekte	Existenz von Klassenobjekten	Klassenlevel
Aufrufgraphanalyse	statischer Aufrufgraph	keine	Klassenlevel
Datenflussanalyse	Datenflussgraph	keine	Klassenlevel
LCBA	Aufrufgraph	keine	Methodenlevel
Benennungsregeln	textuelle Informationen	Benennungsschema	Methodenlevel
Lexikalische Analyse	textuelle Ähnlichkeit	keine	Methodenlevel
Co-Entwicklung	Versionskontrollsystem	VCS, zeitgleiche Entwicklung von Komponententests	Klassenlevel

Nach der Umwandlung bilden die enkodierten Vektoren einen Vektorraum. Der Vektorraum stellt erkannte Zusammenhänge dar. Bestimmte Richtungen stellen bestimmte Eigenschaften der Textfragmente dar. Vektoren die nahe beieinander sind besitzen ähnliche Eigenschaften. Diese Nähe lässt sich über die Kosinusähnlichkeit berechnen. Der Vorteil dessen als Ähnlichkeitsmaß ist, dass es auf das Intervall $[-1,1]$ normiert ist [Kar18].

2.4 Grundlegende Komponententest zu Quelltext Rückverfolgbarkeitsmethoden

Es sind sieben grundlegende Komponententests zu Quelltext Verbindungsmethoden in der Literatur vorhanden. In „*Test-to-code traceability: Why and how?*“ von Qusef [Qus13] und „*Establishing Traceability Links between Unit Test Cases and Units under Test*“ von Van Rompaey et al. [VRD09] werden diese sieben verschiedene Ansätze vorgestellt. Die Unterschiede sind die Art der Informationsgewinnung und den notwendigen Bedingungen für die Anwendbarkeit. In Tabelle 2.1 sind diese dargestellt. In der Spalte Granularität ist ersichtlich, dass viele Methoden auf Klassenlevel arbeiten. Jede der Methoden benötigt mindestens den Quelltext, wobei manche Methoden mehr Vorbedingungen haben. Die Co-Entwicklung hat mit der Existenz eines Versionsverwaltungssystem (VCS) die größte Voraussetzung.

2.4.1 Klassenobjektanalyse

Die Analyse von Klassenobjekten wird in „*Establishing Traceability Links between Unit Test Cases and Units under Test*“ von van Rompaey et al. [VRD09] vorgestellt. Die Annahme hierbei ist, dass alle Komponententests auf einer festen Menge an Klassenobjekten arbeiten. In der `SetUp` Methode werden diese erstellt und danach von jedem Testfall benutzt. Nachdem alle Klassenobjekte identifiziert wurden, findet eine Filterung statt. Übrig bleiben die getesteten Klassen. Problem hierbei ist, dass die Methode keine Ergebnisse liefert, falls die Komponententests keine Klassenobjekte verwenden. Dann gewinnt die Methode keine Informationen. Es liefert nur Ergebnisse auf Klassenlevel und nicht Methodenlevel.

2.4.2 Aufrufgraphanalyse

Anstatt sich auf die Klassenobjekte bei der Analyse zu verlassen, verfolgt die Aufrufgraphanalyse einen anderen Ansatz. Die Klasse, auf der am häufigsten Methoden aufgerufen werden, wird als getestete Klasse angenommen. Das Vorgehen nutzt für die Analyse den

Aufrufgraphen. Problem hierbei ist, dass eine zu große Menge an getesteten Klassen entstehen kann. Bei Komponententests, die selber ersten den Zustand erzeugen statt einer `SetUp` Methode, liefert die Methode schlechte Ergebnisse. Eine Verfeinerung auf Methodenlevel ist nicht möglich. Es wird nur erkannt welche Klasse von einem Komponententest getestet wird [VRD09].

2.4.3 Letzter Aufruf vor Überprüfung

Aufrufgraphanalyse liefert keine Ergebnisse, falls keine Klasse in der Aufrufanzahl hervorsteht. Dies probiert die Methode Letzter Aufruf bevor Überprüfung(LCBA) zu lösen. Es wird angenommen, dass eine Methode auf der getesteten Klasse in der Überprüfung aufgerufen wird. Dies verändert entweder den Zustand des getesteten Objektes oder fragt diesen veränderten Zustand ab. Dadurch wird die getestete Klasse erkannt. Bei mehreren Überprüfungen, ist das Ergebnis aber nicht eindeutig und/oder fehlerhaft. Die fehlerhaften Ergebnisse entstehen, wenn z.B. bei einem Komponententest Überprüfungen verwendet werden um den Auf- und Abbau zu überprüfen. Die Verbindungen sind sowohl auf Klassen als auch auf Methodenlevel möglich, wobei bei Methodenlevel die Zugriffsfunktionen ein Problem sind. Eine Zugriffsfunktion liefert nur einen Zustand zurück und verändert diesen nicht, somit sollte es nicht die getestete Methode sein [QOD10].

2.4.4 Datenflussanalyse

Datenflussanalyse ist ein neuerer Ansatz, der seine Ursprünge in der Informationsflusstheorie hat. Es wird analysiert welche Klassen die in der Überprüfung verwendeten Objekte verwenden. Die Analyse startet bei der letzten Überprüfung und verarbeitet den Datenflussgraph rückwärts. Alle Klassen, die die getesteten Objekte direkt oder transitiv nutzen, bilden die Menge an Klassen, die getestet sein können. Nach der Analyse wird das Analyseergebnis von allen Standardklassen bereinigt. Das Ergebnis ist auf Klassenlevel [QOD10].

2.4.5 Benennungsregeln

Benennungsregeln sind ein einfacher aber erfolgsversprechender Ansatz aus „*Establishing Traceability Links between Unit Test Cases and Units under Test*“ von van Rompaey et al. [VRD09]. Die Annahme hierbei ist, dass Testfälle so benannt werden wie die getestete Methode. Eine Testmethode hat den Namen der getesteten Methode im Methodennamen und die Klasse heißt wie die getestete Klasse nur mit `Test` davor. Falls es ein Benennungsschema gibt, wird eine sehr gute Präzision erzielt, fehlt dieses aber eine sehr schlechte. Entwicklungsumgebungen unterstützen diese Benennungsschema nicht immer. In größeren Projekten ist selten ein striktes Benennungsschema der Tests durchgesetzt. Bei mehreren Entwicklern verstärkt sich das Problem, weil verschiedene Schemen vorhanden sein können. Bei Benennungsregeln kann immer nur ein Komponententest einer Methode zugeordnet werden.

2.4.6 Lexikalische Analyse

Eine Erweiterung der Methode der Benennungsregeln ist die Nutzung von Lexikalischer Analyse. Die Annahme hier ist, dass das Vokabular in Testfällen und der getesteten Methode ähnlich sein sollte. Quellen für das Vokabular können z.B. Methodennamen, Variablennamen, Kommentare und Dokumentation sein. Die Ähnlichkeitsanalyse kann mittels Stammformreduktion und Lemmatisierung oder tiefen neuronalen Netzen unterstützt werden. Ähnlich zu Benennungsregeln erreicht die Methode eine hohe Präzision, auch bei nicht konsequenter Benennungen. Es braucht keine strikten Benennungsregeln und eignet sich deswegen sehr gut bei größeren Projekten mit mehreren Entwicklern. Problem ist, dass bei fehlenden Informationsquellen wie z.B. keiner sinnvollen Variablenbenennung, die Methode keine Ergebnisse produziert [KGDP11].

2.4.7 Co-Entwicklung

Co-Entwicklung hat im Vergleich zu den vorherigen Methoden eine andere Informationsquelle, das Versionsverwaltungssystem. Die Annahme hierbei ist, dass sich der Quelltext zeitgleich mit den Komponententest ändert. Die Verfolgung der Änderungen wird von einer Versionsverwaltung übernommen. Theoretisch sollte dies sehr gute Ergebnisse produzieren, aber es gibt viele Probleme. Komponententests werden nicht immer zusammen mit dem korrespondierenden Quelltext in die Versionsverwaltung eingefügt. Bei Änderungen am Quelltext und daraufhin fehlschlagenden Tests werden diese teilweise erst später verändert [Qus13].

3 INDIRECT

Diese Arbeit ist Teil des Projektes „*INDIRECT: INtent-DrIven Requirements-to-Code Tracability*“ von Hey [Hey19] am IPD Tichy. Das Projekt beschäftigt sich mit der Rückverfolgbarkeit von Quelltextabschnitten und Anforderungen in natürlicher Sprache. Bei der Entwicklung größerer Softwareprojekte ist meist eine Anforderungsbeschreibung vorhanden, welche die gewünschten Funktionalitäten beschreibt. Im Quelltext werden diese dann implementiert. Die Informationen welche Quelltextabschnitte eine Anforderung implementieren gehen aber häufig verloren. Falls diese Informationen notwendig sind, müssen diese im nachhinein gesucht werden. Das Ziel des Projekts ist die Entwicklung und Evaluation einer vollautomatischen Informationsgewinnung der Verbindungen zwischen Anforderung und Quelltext.

3.1 Problemstellung und Lösungsansatz

Quelltext und natürliche Sprache haben verschiedene Eigenschaften. Quelltext folgt einer vorgegebenen Struktur und dessen syntaktische Bedeutung ist eindeutig. Natürliche Sprache besitzt eine ungefähre Struktur wegen der zugrundeliegenden Grammatik, ist aber redundant und die Bedeutung ist uneindeutig. Zusätzlich existieren Sonderfälle wegen der Flexion der Wörter. Für eine erfolgreiche Zuordnung von Quelltextabschnitten zu Anforderungen ist ein Verständnis der Bedeutung dieser beiden hilfreich.

Das Vorgehen ist in der Abbildung 3.1 dargestellt. Statt Quelltextabschnitte und Anforderungen direkt einander zuzuordnen, werden erst zwei Absichtsmodelle erstellt. Diese sollen jeweils die Absicht der Anforderung bzw. des Quelltextes darstellen. Das Anforderungsabsichtsmodell wird mittels iterativer Verarbeitung von natürlicher Sprache aus den Anforderungen erzeugt. Das Quelltextabsichtsmodell soll aus den verschiedenen Quelltextartefakten wie z.B. dem Quelltext und den Komponententests erstellt werden. Für die verschiedenen Arten von Quelltextartefakten werden verschiedene Analysemethoden verwendet. Nach den Erstellungen findet eine Zuordnung zwischen den Absichtsmodellen statt. Das Ergebnis davon sind die wiedergewonnenen Informationen der Quelltextabschnitte, welche die Anforderungen implementieren.

3.2 Arbeiten in dem Projekt

Für die Erstellung der beiden Absichtsmodelle sind verschiedene Forschungsarbeiten am laufen. Für das Absichtsmodelle der natürlichen Sprache sind schon verschiedene Arbeiten

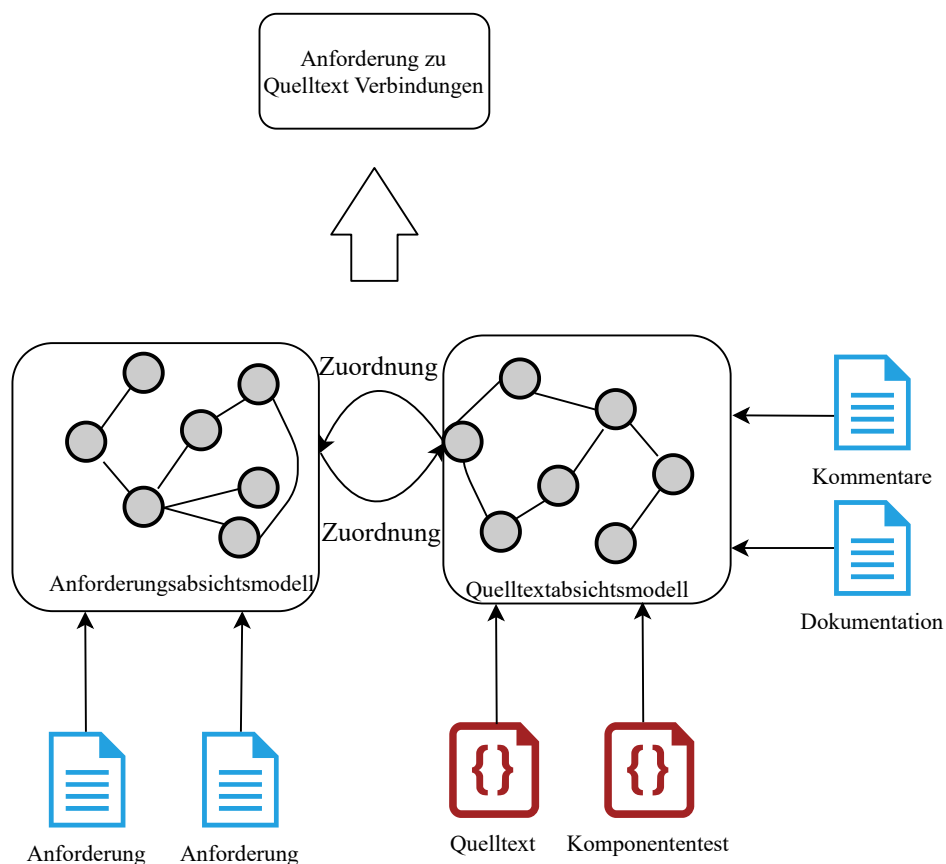


Abbildung 3.1: Darstellung des prinzipiellen Vorgehens von INDIRECT.

abgeschlossen und weitere befinden sich gerade in der Umsetzung, die ein Verständnis der Absicht der Anforderungen ermöglichen sollen. Auf der Quelltextseite sind noch keine Arbeiten innerhalb des Projektes abgeschlossen, die ein Verständnis ermöglichen. Die Analyse der Informationen aus Versionskontrollsystemen, die Analyse von Quelltextdokumentation und diese Arbeit sind erste angedachte Arbeiten auf der Quelltextseite.

4 Verwandte Arbeiten

Diese Arbeit behandelt die Rückverfolgbarkeit von Komponententests zu Quelltext. Zu diesem Thema existieren verwandte Arbeiten, sowohl aus dem Bereich als auch aus anderen Bereichen. Im folgenden werden erst die Arbeiten aus dem Bereich und dann aus anderen Bereichen vorgestellt.

4.1 Komponententest zu Quelltext Rückverfolgbarkeit

In Bereich der Komponententest zu Quelltext Rückverfolgung existieren bereits andere Arbeiten. Die Verfahren aus den Arbeiten basieren auf den in Kapitel 2 vorgestellten Methoden. In Tabelle 2.1 sind verschiedene grundlegende Methoden zusammengefasst. Aus Komponententests extrahieren diese Verfahren Informationen über die getestete Klasse bzw. Methode. Die Verfahren unterscheiden sich hinsichtlich der Präzision, Ausbeute und Anforderungen an die Komponententests.

In „*Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects*“ von Arshia et al. [AMH16] wird ein mehrstufiges Datengewinnungsverfahren vorgestellt. Es werden Benennungsregeln und statische Quelltextanalyse kombiniert. Erst werden über Benennungsregeln die Information der getesteten Klassen gewonnen und als Vergleichsdaten verwendet. Dann werden Quelltextmetriken aus den Vergleichsdaten extrahiert. Als mögliche Quelltextmetriken sind Komplexität oder Designmetriken vorgeschlagen. Diese extrahierten Vergleichsdaten werden verwendet, um mittels eines Datengewinnungsverfahrens Rückschlüsse für die Klassen, die kein Benennungsschema einhalten, zu gewinnen. Die Autoren nutzen dabei die Vermutung aus, dass diese Metriken Rückschlüsse auf die getestete Klasse zulassen, wegen einem Datenzusammenhang. Die Benennungsregeln produzieren gute Ergebnisse, wenn ein Benennungsschema vorhanden ist. Das Verfahren wurde auf drei quelloffenen Projekten evaluiert. Im Vergleich zu Scotch „*SCOTCH: Test-to-code traceability using slicing and conceptual coupling*“ von Qusef et al. [QBO⁺11] lieferte das Verfahren besser Ergebnisse. Es wird im Durchschnitt eine Ausbeute und Präzision von 90% erreicht, wobei dies nur die Zuordnung von Testmethode zu getesteter Klasse beinhaltet. Eine große Schwäche des Verfahrens ist, dass die Benennungsregeln ein Ergebnis liefern müssen. Liefern diese ein schlechtes oder gar falsches Ergebnis, ist das darauf aufbauende Verfahren nicht gut.

In „*Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes*“ von Badri und Toure [BT12] wird ein Vorgehen vorgeschlagen, welches die

Testbarkeit von Quelltext beurteilt. Die Motivation dafür ist, dass in „*A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability*“ von Gupta et al. [GAS05] festgestellt wurde, dass keine Quelltextmetriken alleine ausreichend ist für eine Beurteilung der Testbarkeit. Es wird anhand der Komplexität des Quelltextes festgestellt, wie schwer der Quelltext testbar ist. Je schwerer ein Quelltext testbar ist, desto komplexer werden die Komponententests sein und die Analyse von komplexem Quelltext ist schwieriger. Dies wird dann mit der Größe der Komponententests und der Anzahl der Überprüfungen in Korrelation gesetzt. Die besten Ergebnisse ergeben gewichtete Methoden pro Klasse und Anzahl Quelltextzeilen. Die Metrik gewichtete Methoden pro Klasse wird dadurch bestimmt, dass in jeder Methode einer Klasse die Komplexität bestimmt und summiert wird. Die Analyse findet auf ANT, JFC und POI statt.

In „*Analysis of Static and Dynamic Test-to-code Traceability Information*“ von Gergely et al. [GBH⁺18] wird das Problem des fehlenden vollautomatischen Verfahrens zur Erkennung von schlechter Testquelltext behandelt. Schlechter Testquelltext verletzt das Prinzip, dass ein Komponententest nur eine abgeschlossene Einheit testen soll und in dieser platziert sein soll. Um diese zu erkennen werden zwei Mengen an gefundenen Verbindungen von Komponententest zu Quelltext verglichen. Diese zwei Mengen werden mithilfe statischer und dynamischer Analyse erzeugt. Der Vergleich findet mittels Clusteranalyse statt. Die Verbindungen der statischen Analyse werden über Benennungsregeln erzeugt. Die Verbindungen der dynamischen Analyse über Quelltextüberdeckung. Die Quelltextüberdeckung wird mit einer binären Überdeckungsmatrix aus den Komponententests realisiert. Es wurden auf vier verschiedenen Softwaresystemen acht gefundene schlechte Komponententests analysiert. Dabei wurde genannt, warum es das Prinzip verletzt und wie es möglich wäre den Komponententest zu verbessern.

In „*Automatically identifying focal methods under test in unit test cases*“ von Ghafari et al. [GGR15] wird ein Ansatz vorgestellt, der mittels Datenflussanalyse die getestete Methode innerhalb eines Komponententests identifiziert. Es wird eine Kombination aus Datenflussanalyse und Klassenobjekten verwendet. Erst wird mittels des Aufrufgraphen jede Methode des Quelltextes in Zugriffs- und Änderungsfunktion unterteilt. Dann werden in den Komponententests, die überprüften Objekte analysiert. Als nächstes wird die letzte Methode gesucht, die dieses Objekt verändert. Dafür werden die aus dem ersten Schritt gewonnenen Daten verwendet und der Datenfluss betrachtet. Das Ergebnis davon gilt als die getestete Methode. Der Ansatz Methoden in Zugriffsfunktion und Änderungsfunktion einzuteilen wirkt sehr erfolgsversprechend in dem Gebiet, weil ein Komponententest niemals eine Zugriffsfunktion testet. Die Ergebnisse sind auf Methodenlevel. Es wird ein F-Maß von 66% bis 87% auf Commons Email, JGAP, PureMVC, XStream erreicht.

In „*SCOTCH: Test-to-code traceability using slicing and conceptual coupling*“ von Qusef et al. [QBO⁺11] werden über Datenflussanalyse Informationen über die Verbindung von Komponententest zur getesteten Klasse gewonnen. Der Vorteil davon ist, dass der Ansatz immer nutzbar ist im Vergleich zu Benennungsregeln oder letzter Aufruf vor Überprüfung. Das Verfahren hat als Annahme, dass wenn eine Methode aufgerufen wird, die den Zustand des in der Überprüfung verwendeten Objektes verändert, es zur getesteten Klassen gehört. Es wird von der letzten Überprüfung rückwärts eine Datenflussanalyse gemacht, welche alle Methoden findet, die den Zustand verändern. Dann werden diese gefunden Klassen gefiltert um Standardklassen zu entfernen. Als letztes wird dann zwischen den gefundenen Klassen und dem Komponententest die semantische Ähnlichkeit mittels Latenter Semantischer Analyse berechnet. Alle Klassen die über einen Grenzwert liegen gelten dann als mögliche getestete Klasse. Dieser Grenzwert ist der höchste Ähnlichkeitswert multipliziert mit einem Faktor. Dabei hat sich 0.95 als erfolgsversprechend rausgestellt. In der Evaluation erreichten die Autoren auf ArgoUML, AgilePlanner und eX Vantage ein durchschnittliches F-Maß von 77%, bei einem Faktor von 0.95.

In „*Traceclipse: An eclipse plug-in for traceability link recovery and management*“ von Klock et al. [KGDP11] werden Komponententest zu Quelltext Verbindungen über textuelle Ähnlichkeit hergestellt. Lucene¹ dient hierbei als Rahmenarchitektur zur Durchführung davon. Es wird angenommen, dass der Quelltext und die getestete Klasse ein ähnliches Vokabular verwenden. Diese textuelle Ähnlichkeit soll Rückschlüsse auf die getestete Klasse liefern. Der Fokus des Papiers ist eher das Erstellen einer Rahmenarchitektur, welche leicht erweiterbar ist und verschiedene Methoden durchführen kann, als das Erstellen eines verbesserten Verfahrens. Es werden 40% Präzision und 10% bis 40% Ausbeute erreicht. Die Granularität ist einstellbar auf Klassen oder Methodenlevel.

In „*Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability*“ von Kicsi et al. [Kic18] werden Komponententest zu Quelltext Verbindungen über textuelle Ähnlichkeit hergestellt. Es wird angenommen, dass der Quelltext und die getestete Klasse ein ähnliches Vokabular verwenden. Diese textuelle Ähnlichkeit soll Rückschlüsse auf die getestete Klasse liefern. Es werden zwei Datensätze erstellt, aus den Komponententests und dem Quelltext. Dann wird die Ähnlichkeit dazwischen berechnet und eine Liste mit N Klassen, die ähnlich sind, erstellt. Die textuelle Ähnlichkeit besitzt eine höhere Anwendbarkeit als Benennungsregeln, weil diese kein festes Schema benötigen. Anders als in [KGDP11] werden nicht eins zu eins sondern eins zu N Verbindungen evaluiert. Die höhere Anzahl an Vorschlägen verbessern die Ausbeute. Die Ausbeute für N = 5 erreicht auf Comm. Lang, Comm. Math, ArgoUML, Mondiran und JFreeChart im Durchschnitt 68,4%. Die Präzision wird nicht gemessen.

In „*Evaluation of Textual Similarity Techniques in Code Level Traceability*“ von Csuvi et al. [CKV19] wird das Verfahren von [Kic18] als Vergleichswert genommen und mit maschinellem Lernen verbessert. Benennungsregeln erzielen eine sehr hohe Präzision, aber besitzen das Problem der Anwendbarkeit. Textuelle Ähnlichkeit soll Rückschlüsse auf die getestete Klasse liefern. Es gilt die selbe Annahme wie bei [Kic18]. Bevor die Ähnlichkeit berechnet wird, wird der Quelltext transformiert. Es wurden drei verschiedene Quelltexttransformationen getestet, abstrakte Syntaxbäume, Quelltext mit Stammformreduktion und eine vereinfachte abstrakte Syntaxbaum Darstellung (IDENT). In der IDENT Darstellung werden alle terminalen Knoten genutzt, wobei der Wert durch seinen Typ ersetzt wird. Auf die drei verschiedenen Transformationen werden dann drei verschiedene Ähnlichkeitsalgorithmen angewendet. Die drei verschiedenen Analysen sind Latente Semantische Analyse, Doc2Vec² und Inverse Dokumenten Häufigkeit. In der Evaluation waren die Ergebnisse mit der IDENT-Transformation am besten. Von den Ähnlichkeitsalgorithmen alleine ausgeführt ist das Ergebnis von Doc2Vec am besten. Aber eine Kombination aller drei erzielt bessere Ergebnisse. Die Kombination wird über eine Schnittmenge der drei Ähnlichkeitsalgorithmen erzeugt. Als Evaluationssoftware wurden die selben wie in [Kic18] verwendet. Doc2Vec erzielt eine durchschnittliche Präzision von 56,75% und die Kombination aller drei eine von 59,75%. Die Ausbeute wurde nicht berechnet.

In Tabelle 4.1 sind die vorgestellten verwandten Arbeiten aus dem Bereich tabellarisch dargestellt. Alle Arbeiten nutzen als Eingabeprogrammiersprache Java. Die verwendeten Grundlagenmethoden ist in fast allen Fällen eine Kombination aus mehreren. Bis auf ein Verfahren sind alle Verfahren vollautomatisch. Bei der Evaluationssoftware wird häufig ArgoUML und Software von Apache Commons verwendet.

4.2 Andere Themengebiete

Im folgenden werden Arbeiten aus anderen Bereichen vorgestellt. Diese behandeln nicht die Rückverfolgbarkeit von Komponententests zu Quelltext, aber lösen ähnliche Probleme

¹<http://lucene.apache.org/>

²eine Umformung eines ganzen Dokumentes in eine numerische Darstellung

wie diese Arbeit. Dieses Wissen hilft bei der Lösung des eigenen Problems.

In „*Identification of high-level concept clones in source code*“ von Marcus und Maletic [MM01] beschäftigen sich die Autoren mit der Erkennung von Quelltextkopien. Laut den Autoren sucht diese Arbeit, anstatt anderer Arbeiten in dem Bereich, nach ähnlicher Funktionalität statt nur ähnlichem Quelltext. Es wird mittels eines Vektorraumverfahrenes (Latent Sematic Indexing) nach ähnlichen Quelltextdokumenten gesucht. Dann wählt der Nutzer auf den gefundenen ähnlichen Dokumenten eine Gruppe aus, die eine bestimmte Funktionalität implementieren. Anhand dieser Information wird dann nach ähnlichem Quelltextdokumentengruppen zu dieser Gruppe gesucht. Für die Evaluation wird `Mosaic`, geschrieben in C, verwendet. Die Evaluation ist das Erkennen von Listenklonen innerhalb der Software. Als Ergebnis wird genannt, dass elf verschiedene Listenklone gefunden wurden, wobei 19 Dokumenten ähnlich zu `list.c` sind. Leider benötigt das Verfahren Nutzerinteraktion für die endgültige Identifikation von Quelltextkopien. Dieses ließe sich mittels einer Kombination von mehreren Ähnlichkeitsidentifikationsmethoden automatisieren.

In „*DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones*“ von Jiang et al. [JMSG07] beschäftigen sich die Autoren mit der Erkennung von Quelltextduplikaten. Anders als andere Verfahren in dem Bereich werden Abstrakte Syntaxbäume genutzt. Die Duplikatserkennung soll robust gegenüber minimalen Quelltextänderungen sein und eine gute Skalierbarkeit besitzen. Mittels YACC³ wird der Quelltext zerteilt und der Abstrakte Syntaxbaum erstellt. Dann wird der Syntaxbaum erst in Teilbäume, dann in Charakteristik-Vektoren umgewandelt. Die Charakteristik-Vektoren zählen die Vorkommen von relevanten Zeichen im Syntaxbaum und besitzen dafür jeweils eine Dimension. Die Anzahl der Vektoren wird dann verringert indem bestimmte Vektoren, wie z.B. am Ende einer Zuweisung, zusammengefasst werden. Ähnliche Vektoren lassen nun auf ähnlichen Quelltext schließen. Diese Erkennung findet auf den Vektoren statt. Die Laufzeit wird mittels einem speziellen Hashverfahren⁴ verbessert. Vorteil des Verfahrens ist, dass es wegen dem Hashing schneller Laufzeiten hat. Die Skalierbarkeit wird über eine Laufzeitanalyse im O-Kalkül nachgewiesen. Nachteil ist, dass das Verfahren auf eine YACC-Grammatik angewiesen ist. Die Autoren vergleichen es mit einer bestehenden Arbeit und zeigen, dass sie bei jeder Konfiguration besser sind.

In „*Are My Unit Tests in the Right Package?*“ von Balogh et al. [BGBG16] beschäftigen sich die Autoren mit der Frage, ob die Komponententests in der Paketstruktur richtig eingeordnet sind. Dies ist der Fall, falls die zwei primären Eigenschaften von Komponententests erfüllt sind. Die Atomarität und die sinnvolle Organisation. Es wird ein Algorithmus zur Überprüfung dieser Eigenschaften vorgestellt. Der Algorithmus besteht aus drei Schritten. Erst wird mittels statischer Analyse für jeden Komponententest die Information über die Klasse und das Paket gewonnen. Dann wird eine binäre Überdeckungsmatrix mittels dynamischer Analyse erstellt. Mittels der Überdeckungsmatrix für jeden Komponententest werden dann zusammengehörige Komponententests erkannt. Im letzten Schritt wird eine Clusteranalyse durchgeführt. Das Ergebnis sind Komponententests, die die Bedingungen nicht erfüllen und eine Überarbeitung in der Struktur benötigen.

³<https://de.wikipedia.org/wiki/Yacc>

⁴https://en.wikipedia.org/wiki/Locality-sensitive_hashing

Tabelle 4.1: Verschiedene evaluierte Komponententest zu Quelltext Zuordnungsverfahren. Falls in der Spalte Programmiersprache keine Sprache erwähnt ist, gibt es keine Einschränkung. Die Verfahren sind eine Erweiterung von den grundlegenden Verfahren. Vollautomatisch gibt an, ob ein Verfahren ohne Einwirkung/Mithilfe die Verbindungen erkennt. Die Evaluationssoftware sind die in der Evaluation genutzten Datensätze. Diese wurden nicht immer vollständig verwendet, sondern teilweise nur Teilmengen davon.

Autor	Programmiersprache	Verwendete Methoden	vollautomatisch?	Evaluationssoftware
Arshia et al.	Java	Benennungsregeln/ statische Analyse	X	Ant, ArgoUML, Dependency Finder
Gergely et al.	Java	Benennungsregeln		CheckStyle, Netty, Ori- entDB, Oryx
Ghafari et al.	Java	Datenflussanalyse/ Klassenob- jekte	X	Commons Email, JGAP, Pu- reMVC, XStream
Qusef et al.	Java	Datenflussanalyse/ LCBA/ Le- xikalische Analyse	X	ArgoUML AgilePlan- ner, eX Vantage
Klock et al.		Lexikalische Analyse	X	iTrust, Rea- lEstate, EasyClinic, CM1
Kicsi		Lexikalische Analyse		Comm. Lang, Comm. Math, Ar- goUML, Mondiran, JFreeChart
Csuvik et al.		Lexikalische Analyse	X	Comm. Lang, Comm. Math, Ar- goUML, Mondiran, JFreeChart

5 Analyse und Entwurf

Diese Arbeit entwirft eine Analyseverfahren zur Erkennung von semantisch zusammengehörige Quelltextabschnitte aus Komponententests. Semantisch zusammengehörige Quelltextabschnitte erfüllen gemeinsam eine Funktionalität des Programms. In Quelltextausschnitt 5.1 ist ein Komponententest dargestellt. In diesem wird getestet, ob die Summenoperation die gewünschte Funktionalität besitzt. Dafür werden erst zwei Objekte vom Typ `Number` erstellt und eine Operation vom Typ `Sum`. Dann werden der Operation die beiden Nummern übergeben und das Ergebnis überprüft. Die Methode `Sum::process` und die Klasse `Number` implementieren zusammen die Summenoperation von zwei Zahlen. Semantisch zusammengehörig ist `Sum::process` und `Number::of`. Das Analyseverfahren soll anhand von Komponententests solche zusammengehörigen Methodenpaare erkennen.

Quelltextausschnitt 5.1: Beispiel eines Komponententests

```
/**
 * tests if the sumOperation works as intended ,
 * with two real Numbers.
 */
@Test
public void sumTest () {
    Number i = Number.of(42);
    Number j = Number.of(13);
    Operation add = new Sum();
    assertThat(add.process(i, j), is(55));
}
```

Das Projekt `INDIRECT` behandelt die automatische Wiederherstellung von Rückverfolgbarkeitsinformationen zwischen Quelltext und Anforderungen. Anstatt zu versuchen Quelltext und Anforderungen direkt zuzuordnen, findet die Zuordnung über Absichtsmodelle statt. Eines für den Quelltext und eines für die Anforderungen. Diese sollen jeweils die Absicht der Anforderung bzw. des Quelltextes darstellen. Für die Erstellung der Absichtsmodelle ist ein Verständnis von Anforderungen bzw. Quelltext notwendig. Ein Verständnis der Absichten im Quelltext wird über verschiedene Analysen der Quelltextteile erlangt. Diese Arbeit liefert dem Quelltextabsichtsmodell die Information über semantisch zusammengehörige Methoden und soll beim Erlangen des Verständnisses mitwirken.

Die Funktionalität eines Programms wird mit dem Zusammenspiel mehrerer Komponenten umgesetzt. Jede Komponente erfüllt dabei eine oder mehrere Aufgaben. Diese Aufgabe sollte durch das Zusammenspiel von mehreren Methoden innerhalb einer Komponente imple-

mentiert sein. Deswegen ist die Betrachtung der einzelnen Komponenten erfolgsversprechend. Für die einzelnen Komponenten können Komponententests existieren. Diese Tests überprüfen die einzelne Komponente bzw. die Funktionalität der Komponente. Daher sollten in Komponententests einer Komponente semantisch zusammengehörige Methoden miteinander genutzt werden beim Testen der Funktionalität. Deswegen könnten die Tests der einzelnen Komponenten eine Informationsquelle über zusammengehörige Quelltextabschnitte sein.

5.1 Zielsetzung

Das Hauptziel dieser Arbeit ist die Entwicklung eines Analyseverfahrens zur Erkennung von semantisch zusammengehörigen Methoden anhand von Komponententests. Zur Entwicklung des Verfahrens werden erst alle Informationen innerhalb eines Komponententests analysiert. Dann die Auswertungen bisheriger Arbeiten auf diesen Informationen bewerten. Aufgrund dieser Auswertung wird dann entschieden, welche Informationen als zielführend betrachtet werden. Für diese Information werden dann einzelnen Verfahren zur Informationsgewinnung entworfen. Aus den einzelnen Verfahren wird im letzten Schritt dann das Gesamtverfahren entworfen.

Die konkreten Ziele dieser Arbeit sind:

- Z1 Analyse der vorhanden Informationen innerhalb von Komponententests.
- Z2 Entwicklung von Informationsgewinnungsverfahren für die einzelnen Informationen.
- Z3 Entwicklung eines Verfahrens zur Erkennung von semantischen zusammenhängenden Quelltextabschnitten anhand von Komponententests.
- Z4 Evaluation und Beurteilung des Verfahrens.

5.2 Analyse der vorhandene Informationen

Aus dem Quelltext der Komponententests und des Programms lassen sich verschiedene, für die Erkennung von semantisch zusammengehörigen Methoden relevante, Informationen gewinnen. In Quelltextausschnitt 5.2 ist der Ausschnitt einer Testklasse aus `JFreeChart`¹ dargestellt. Der Aufbau ist zu einer normalen Javaklasse ähnlich. Der Unterschied sind die Verwendung der JUnit-Annotationen und die Überprüfungen mittels `assert`. Eine Testklasse lässt sich in verschiedene größere Informationsquellen aufteilen. Diese sind innerhalb einer Testklasse:

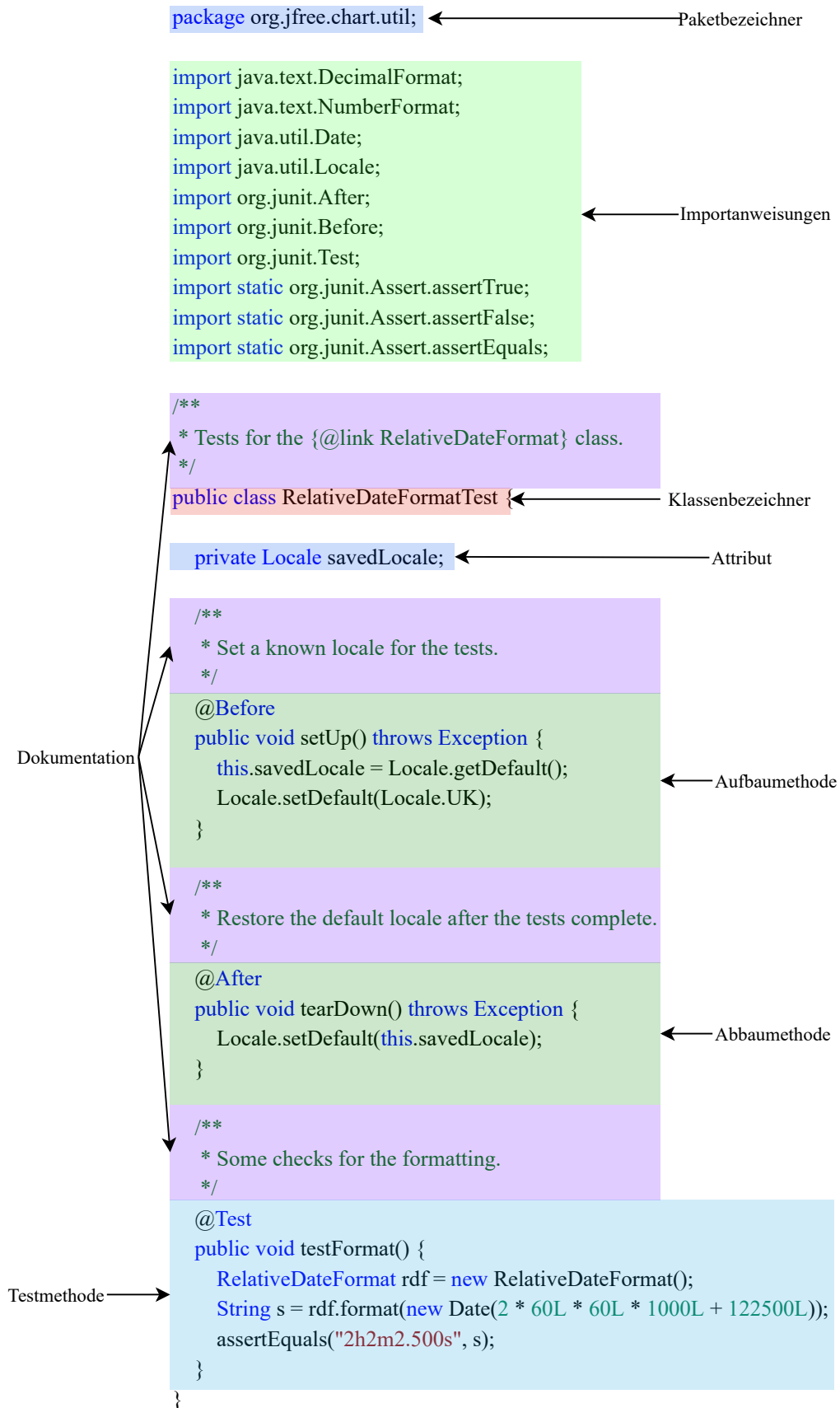
- Informationsquelle 1 Der Paketbezeichner.
- Informationsquelle 2 Die Importanweisungen.
- Informationsquelle 3 Der Klassenbezeichner.
- Informationsquelle 4 Die Dokumentation.
- Informationsquelle 5 Die Attribute.
- Informationsquelle 6 Die Testmethoden.
- Informationsquelle 7 Die Aufbau- und Abbaumethoden.

Diese Informationsquellen sind farblich hervorgehoben. Der Quelltext besitzt noch weitere Informationsquellen in den Hauptklassen. Diese Informationsquellen sind zusätzlich vorhanden und sind innerhalb einer Hauptklasse:

- Informationsquelle 1 Der Paketbezeichner.
- Informationsquelle 2 Die Importanweisungen.
- Informationsquelle 3 Der Klassenbezeichner.
- Informationsquelle 4 Die Dokumentation.
- Informationsquelle 5 Die Attribute.
- Informationsquelle 6 Die Methoden.

¹<http://www.jfree.org/jfreechart/>

Quelltextausschnitt 5.2: Beispiel einer Testklasse mit farblich hervorgehobenen Informationsquellen



5.3 Analyse der Informationsquellen aus dem Quelltext

Im folgenden werden die einzelnen Informationsquellen analysiert. Hierbei wird erst die Struktur der Informationsquelle erläutert. Dann die Art der Information, also semantisch oder textuell. Darauf folgend wird analysiert welche Rückschlüsse sich aus den einzelnen Teilen der Informationsquelle gewinnen lassen. Im folgenden ist mit Hauptklassen alle Klassen gemeint die keine Testklasse sind. Aus den Hauptklassen werden keine Rückschlüsse gewonnen sondern nur Methodenname, Klassenname und Paketbezeichner extrahiert für eine Verwendung. Textuelle Information sind Zeichenketten wie ein Bezeichner. Semantische Informationen sind programmtechnische Inhalte wie z.B. Übergabeparameter. Diese besitzen mehr Informationen als nur die Zeichenkette, z.B. der Typ eines Übergabeparameters.

5.3.1 Paketbezeichner

Der Paketbezeichner definiert das Paket einer Klasse. Die Struktur ist in Java `Oberpaket.Paket`, also z.B. `org.jfree.chart.util`. Das Oberpaket enthält beliebig viele Oberpakete. Es handelt sich um eine textuelle Information.

Jeder Komponententest besitzt entweder einen Paketbezeichner oder den leeren Paketbezeichner. In der Literatur wird vorgeschlagen, dass eine Komponententestklasse in dem selben Paket wie die getestete Klasse sein sollte [BGBG16]. Deswegen kann der Paketbezeichner Rückschlüsse auf die getestete Klasse ermöglichen. Jede Quelltextklasse im selben Paket wie die Testklasse ist ein potenzieller Kandidat für die getestete Hauptklasse.

5.3.2 Importanweisung

Die Importanweisungen einer Klasse definieren welche anderen Klassen bzw. Pakete geladen werden. Diese sind eine textuelle Information und deren Struktur ist das Schlüsselwort `import` gefolgt von einem Paketbezeichner, z.B. `import org.jfree.chart.util;`. Der unterste Paketbezeichner kann durch ein `*` ersetzt werden um alle Klassen eines Oberpaketes zu laden, also z.B. `import org.jfree.chart.*;`. Mittels dem Schlüsselwort `static` ist es möglich statische Felder einer Klasse zu laden. Dies ermöglicht den direkt Aufruf der Konstante ohne die Ursprungsklasse anzugeben. In Beispiel 5.1 sind die möglichen Varianten dargestellt.

Beispiel 5.1: Beispiel für mögliche Importanweisungen

```
import org.jfree.chart.util;  
import org.jfree.chart.*;  
import static java.lang.Math.*;  
import static java.lang.System.out;
```

Nur geladene Klassen und deren Methoden können im Quelltext verwendet werden. Deswegen lässt sich daraus eine Liste der potenziellen Kandidaten für die getestete Methoden und Klassen gewinnen.

5.3.3 Klassenbezeichner

Ein Klassenbezeichner besteht aus dem Schlüsselworten `class` gefolgt von dem Klassennamen. Vor dem Klassenbezeichner sind Sichtbarkeitsmodifizierer wie `public` oder `private` möglich. Der Klassenname ist eine textuelle Information und die Schlüsselworte davor eine semantische Information.

Für Klassenname wird in der Literatur vorgeschlagen, dass dieser den Namen der getesteten Klasse beinhaltet [KGDP11, CKV19]. Das häufig verwendete Erstellungsprogramm Maven² hat z.B. als voreingestellte Konvention, dass eine Testklasse entweder mit dem Wort „Test“ beginnen oder mit „Test“, „Tests“ oder „Testcase“ enden muss³. Der Klassenname sollte nach einer Bereinigung davon zu einer Quelltextklasse zurordenbar sein. Problem ist, dass diese Benennungsregeln projektabhängig sind und in größeren Projekten nicht immer vollständig eingehalten sind. Es werden manchmal die Tests einer Klasse in mehrere Testklassen aufgeteilt und die Klassen verschieden benannt, beispielsweise mit dem Präfix `Basic`. Falls die Benennungsregeln nicht eingehalten sind, wird aus dem Klassenname keine Information extrahiert. Der Klassenname ermöglicht trotzdem Rückschlüsse auf die getestete Klasse.

5.3.4 Klassenattribute

Ein Attribut einer Javaklasse besteht aus einem Typ gefolgt von einem Attributnamen und einer optionalen Initialisierung. Es sind verschiedene Schlüsselwörter wie z.B. Sichtbarkeitsmodifizierer vor dem Typ möglich um einem Attribut gewissen Eigenschaften wie die öffentliche Sichtbarkeit zu geben. Die Attribute einer Klasse werden dazu verwendet einen Zustand zwischenspeichern. Attributen können Werte zugewiesen werden. Dafür gibt es zwei verschiedene Arten der Zuweisung, statisch im Quelltext bei der Initialisierung oder innerhalb einer Methode. Der Attributname ist eine textuelle Information und die Schlüsselwörter davor, der Typ und der zugewiesene Wert eine semantische Information. Die Auswertung der Klassenattribute ist bei der Aufbaumethode lesbar, weil die Aufbaumethode bei Komponententests diese verwendet.

5.3.5 Dokumentation

Die Dokumentation einer Javaklasse ist in drei verschiedene Arten einteilbar. Die Methodendokumentation, Klassendokumentation und Kommentare. Methodendokumentation befindet sich vor der Methodensignatur und beschreibt die Übergabeparameter, den Rückgabeparameter und Ausnahmen. Zudem ist eine textuelle Beschreibung der Methode und einer Erklärung was bei der Benutzung zu beachten ist darin inkludiert. Die Klassendokumentation ist häufig nach den Importanweisungen und vor der Klassendefinition. Diese kann die Erklärung der Funktionalität der Klasse, den Autor und eine Versionsnummer beinhalten. Kommentare sind überall im Java Quelltext möglich und werden mit `//` am Anfang gekennzeichnet oder als Blockkommentare gestartet mit `/*` und beendet mit `*/`. Diese Kommentare werden für verschiedene Zwecke wie z.B. TODO-Erinnerungen, Weblinks oder Quelltexterklärungen verwendet. Als Standard für Kommentare wird bei Java JavaDoc⁴ verwendet. Dieser Standard definiert Annotationen zur Strukturierung der Dokumentation, wie z.B. `@param` für ein Übergabeparameter. Aufgrund des Standards ist die Dokumentation eine textuelle Information, die sowohl strukturiert wegen den Annotationen als auch unstrukturiert wegen des Freitext ist.

Klassendokumentation

Innerhalb des Freitextes sind HTML-Anweisungen erlaubt.⁵ Diese HTML-Anweisungen können für Hervorhebungen genutzt werden und erhöhen nur die Lesbarkeit durch hervorheben von wichtigen Elementen. Problem an Formatierung und Hervorhebungen ist, dass diese keinen festgelegten Regeln folgen müssen und somit die Analyse davon komplex ist.

²<http://maven.apache.org/>

³<http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>

⁴<https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

⁵<https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#format>, zuletzt besucht am 13.08.2019

HTML-Anweisungen bieten mehr Möglichkeiten als nur Hervorhebungen, deswegen ist eine Informationsgewinnung aus diesen HTML-Anweisungen nicht in dieser Arbeit. Der Freitext einer Testklasse kann beschreiben welche Klasse getestet wird. In dieser Beschreibung sollte ein ähnliches Vokabular, wie in der getesteten Klasse verwendet werden. Durch die Nutzung dieser Ähnlichkeit sollten Rückschlüsse auf die getestete Klasse bzw. Methode möglich sein. Für das Verarbeiten der Dokumentation ist ein Verständnis des Freitextes notwendig. Dies ist aufgrund der Komplexität nicht Teil dieser Arbeit. Eine Sonderform dieses Freitextes ist ein Lizenztext. Dieser beschreibt nur die rechtlichen Grundlagen des Quelltextes, z.B. die MIT-Lizenz. Dieser Text ist unabhängig von dem Programm und nur von der gewählten Lizenz abhängig. Deswegen enthält dieser Text keine verwendbaren Informationen für diese Arbeit. Die Beschreibungsanweisungen folgen strukturell auf dem Freitext. Vorhandene Beschreibungsanweisungen sind: `@author`, `@version`, `@since`, `@see`, `@deprecated` und `@link`⁶. `@deprecated` markiert eine Klasse als veraltet. Diese Annotation liefert keine Information für das Verfahren, weil „veraltete“ Methoden immer noch semantisch zusammengehörig sein können. `@version` und `@since` treffen Aussagen über die Version und ab wann die Klasse vorhanden war. Beides enthält keine für dieses Projekt nutzbare Informationen, weil die Komponententestversion unabhängig von der Version der getesteten Klasse ist. `@see` und `@link` erlaubt es auf andere Beschreibungsanweisungen zu verweisen oder Verweise auf andere Seiten einzubinden. Ein Verweise auf eine andere Klasse oder Methode kann Rückschlüsse auf die getestete Klasse oder Methode liefern. Ein Link zu einer anderen Seite im Internet würde das Verarbeiten der Internetseite erfordern. Dies ist aufgrund der Komplexität der Informationsgewinnung aus einer Internetseite nicht Teil dieser Arbeit. `@author` dient dazu den Autor einer Testklasse zu nennen. Die Annahme, dass in einem Projekt die Testklasse und die getestete Klasse den selben Autor besitzen erscheint nicht richtig. In kleinen Projekten mit wenigen Autoren, ist die Anzahl an Testklassen, die ein Autor erstellt, eine anteilmäßig große Menge und deswegen die Anzahl an Kandidaten zu groß. In größeren Projekten wird ist eine Klasse nicht mehr an eine Person zuordnenbar, weil mehrere Autoren an einer Klasse arbeiten. Dies sorgt dafür, dass die Menge der Autoren zu groß wird und somit die Menge der Kandidaten an getesteten Klassen. Die Dokumentationsrichtlinien von Google verweisen darauf diese Annotation explizit nicht zu verwenden, weil die Versionsverwaltung dies besser erfüllt⁷. Wegen diesen Gründen wird die `Author` Annotation in dieser Arbeit nicht verwendet. Zusammengefasst sind in Dokumentation viele Information vorhanden, die aber zu komplex und zu wenig erfolgsversprechend sind für diese Arbeit. Deswegen wird kein Teil der Klassendokumentation verwendet werden.

Testmethodendokumentation

Die Testmethodendokumentation besitzt die ähnliche Struktur wie die Klassendokumentation. Es kann ein Beschreibungstext gefolgt von Beschreibungsanweisungen existieren. Der Beschreibungstext kann erklären welche Methode bzw. Klasse getestet wird oder welche Funktionalität. In der Beschreibung sollte aber ein ähnliches Vokabular wie in der getesteten Klasse sein. Deswegen sollte dieser Rückschlüsse auf die getestete Methode bzw. Klasse liefern können. Ähnlich zu dem Freitext der Klassendokumentation wird dieser auch nicht verwendet in dieser Arbeit wegen der Komplexität. Bei Methoden gibt es drei wesentlichen Beschreibungsanweisung: `@param`, `@return` und `@throws`. `@param` dient zur Beschreibung der Parameter einer Methode. `@return` dient zur Beschreibung des Rückgabetypen falls vorhanden. `@throws` dient zur Beschreibung von Ausnahmen. In der Erklärung davon kann erklärt werden, warum die Ausnahme auftritt und wie diese vermeidbar wäre. `@return` und `@throws` besitzt die Dokumentation eines Komponententest nicht, weil die Me-

⁶https://en.wikipedia.org/wiki/Javadoc#Table_of_Javadoc_tags, zuletzt besucht am 13.08.2019

⁷<https://opensource.google/docs/copyright/#individual-author-information>, zuletzt besucht am 18.10.2019

Methodensignatur keine Übergabeparameter und Rückgabetypen erlaubt. `@throws` kann ein Komponententest besitzen, aber Ausnahmen sind keine Funktionalität sondern Grenzen von Funktionalität. Eine Ausnahme wenn z.B. eine Datei nicht gefunden wird, liefert keine Informationen über die Methode, die verwertbar für diese Arbeit sind. Deswegen folgen aus diesen drei Beschreibungsanweisungen keine Informationen. Die von Klassendokumentation bekannten Beschreibungsanweisungen wie z.B. `@Author` wären syntaktisch erlaubt, liefern aber keine Informationen wie bereits bei der Klassendokumentation erklärt wurde. Deswegen werden die Beschreibungsanweisung bei Testmethoden keine Rückschlüsse für diese Arbeit liefern.

Kommentare

Kommentare in Testklassen können sowohl innerhalb Methoden als auch außerhalb davon stehen. Diese können verschiedenen Bedeutungen wie z.B. Entwicklungsentscheidungen haben. Durch ein Verständnis der Bedeutung wäre die Rückführung auf die getestete Klasse oder Methode möglich. Der in anderen Arbeiten verwendete Ansatz über ein ähnliches Vokabular wäre dafür verwendbar [KGDP11]. Die Verarbeitung und Informationsgewinnung aus diesen Kommentaren ist wegen der Komplexität nicht teil dieser Arbeit. Deswegen liefern Kommentare für diese Arbeit keine Rückschlüsse.

5.3.6 Methode

Eine Methode lässt sich unterteilen in Rumpf und die Methodensignatur. Die Methodensignatur besteht aus einem Sichtbarkeitsmodifizierer, gefolgt von dem Rückgabotyp und dann dem Methodennamen. Der Methodename ist eine textuelle Information, z.B. `randomMove`. Der Rumpf besteht aus aufgerufenen Methoden z.B. `isEmpty` und Zuweisungen z.B. `Square square = getSquare()`. Aufgerufene Methoden beinhalten als Information das Aufrufobjekt, die Methode und die Übergabeparameter. Alle diese Information sind semantische Informationen. Zuweisungen bestehen in Java aus zwei Teilen. Einem Objekt mit einem textuellen Bezeichner und einer zugewiesenen Referenz, z.B. `Number i = Number.of(42)`; Dies kann eine direkte Zuweisung oder der Rückgabeparameter einer Methode sein. Deswegen beinhaltet eine Zuweisung textuelle Informationen mit dem Bezeichner. Der Typ des Objektes und die zugewiesene Referenz sind semantische Informationen.

Methodenname

Der Methodenname eines Komponententest kann die getestete Methode oder das getestete Verhalten beschreiben. Diese kann über Benennungsregeln innerhalb eines Projektes festgelegt werden. Falls diese Testmethode namentlich ähnlich wie die getestete Methode benannt wird, lässt sich daraus ein Rückschluss erzeugen. Testmethoden können auch nach Fehlermeldungen und Problemen (engl. Issue) benannt werden, z.B. `Github155`. In dem Fall wäre es möglich mittels der Fehlermeldungen Rückschlüsse zu gewinnen. Da diese Fehlermeldungen verarbeitet werden müssten, nicht teil der Eingabedaten sind und die Zuordnung komplex ist, wird dieser Fall nicht beachtet. Deswegen sollte dieser Rückschlüsse auf die getestete Methode oder Klasse liefern, falls diese gleich oder ähnlich benannt sind.

Kontrollfluss

Anhand der Reihenfolge der Zuweisungen und Methodenaufrufen mit Parametern ist es möglich den Kontrollflussgraph zu extrahieren. Dieser Graph ist gerichtet und kann Zyklen enthalten. Nur Methoden die innerhalb eines Komponententests gemeinsam vorhanden sind können semantisch zusammengehörig erkannt werden. Deswegen lassen sich aus dem Kontrollfluss die verwendeten Methoden extrahieren und aus denen eine Menge an möglichen semantisch zusammengehörigen Methoden. Die Verkettung von Methoden ist vom

Programmierer und den Stilrichtlinien abhängig und nicht semantisch relevant. Deswegen wird aus einer Verkettung keine Information gewonnen, weil der Methodenkontext diese Information auch beinhaltet.

Beispiel 5.2: Beispiel für Verkettung von Methodenaufrufen

```
ProcessBuilder builder = new ProcessBuilder ( arg ).inheritIO ( ) ;  
ProcessBuilder builder = new ProcessBuilder ( arg ) ;  
builder.inheritIO ( ) ;
```

In Beispiel 5.2 ist der Unterschied gezeigt. Beide Aufrufe von `ProcessBuilder` erledigen semantisch das gleiche, haben dies aber mit und ohne Verkettung umgesetzt. In beiden Fällen ist der Methodenkontext `ProcessBuilder` und ermöglicht Aussagen über aufgerufene Methoden einer Klasse. Daraus entsteht die genannte Grundmenge. Die Reihenfolge von Methoden ist nicht relevant in Komponententests, weil sich Teilszenarien davon in der Reihenfolge austauschen lassen und es einen Abhängigkeitsgraph gibt. Der Abhängigkeitsgraph liefert die Information, welche Quelltextbefehle erst nach andere Quelltextbefehlen durchgeführt werden können. Der Abhängigkeitsgraph könnte Informationen über Teilszenarien liefern. Diese Teilszenarien könnten dann semantisch zusammengehörige Methoden sein. Dieser Graph ist für diese Arbeit zu komplex zu erstellen, wegen der Verarbeitung von Sprüngen und Lambdaausdrücken im Quelltext. Deswegen werden aus dem Kontrollfluss nur die verwendeten Methoden für Rückschlüsse verwendet.

Variablenbezeichner

Die Variablenbezeichner innerhalb eines Komponententests können ähnlich zu den Variablenbezeichnern oder Methodennamen in der getesteten Klasse sein. Diese Ähnlichkeit entsteht, wenn bei der Erstellung der Komponententests die Anforderungen oder der Hauptquelltext einfließen. Wenn beim Erstellen der Komponententests der Quelltext nicht hinzugezogen wird, sollte diese nicht vorhanden sein. Die Ähnlichkeit sollte durch ein ähnliches Vokabular rückverfolgbar sein [KGDP11]. Klock et al. erreichte mittels `Traceclipse` im Durchschnitt eine Präzision von 40% und Ausbeute von 30%. Dies zeigt, dass diese Ähnlichkeit nur bedingt verwertbare Ergebnisse produziert. Diese Ergebnisse sind damit begründbar, dass die Benennung den Typ der Variable nicht beinhalten soll ⁸ Dies erlaubt zwei Rückschlüsse: Programmierer erwähnen den Typ der Variable in dem Bezeichner so häufig, dass ein Richtlinie darauf hinweisen muss es nicht zu tun und Variablen sollten nach ihrer Funktionalität benannt werden. Im Fall einer Benennung nachdem Typ, ist der statische Typ der Variable ausreichend und beinhaltet diese Information vollständig. Im Fall einer Benennung nach Funktionalität wäre ein Verständnis der Intention der Benennung notwendig. Dies ist wie bei der Dokumentation und Kommentaren zu komplex für diese Arbeit. Deswegen werden in dieser Arbeit aus Variablenbezeichnern keine Informationen gewonnen.

5.3.7 Überprüfungen

Komponententestmethoden können Überprüfungen besitzen um einen Zustand zu überprüfen. `JUnit` bietet dafür statischen Methoden an, wie z.B. `assertTrue`. Diese besitzen bis zu drei Parameter: das Testobjekt, das Vergleichsobjekt und einer Zeichenkette für den Fehlerfall. Das Vergleichsobjekt und die Zeichenkette sind optional, z.B. `assertTrue` benötigt nur das Testobjekt. Als Test- und Vergleichsobjekt kann ein Methodenaufruf mit

⁸<https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md#variable-naming>, zuletzt besucht am 19.10.2019

Rückgabetyt verwendet werden. Das Test- und Vergleichsobjekt sind semantische Information, die Zeichenkette eine textuelle Information. **Hamcrest**⁹ erlaubt die Verwendung von Hilfsmethoden mit der Methode `assertThat`. Diese Hilfsmethoden sind Zuordner, wie z.B. `lt` für weniger als. `assertThat` hat als Signatur eine optionale Zeichenkette, ein Testobjekt und einen Zuordner.

Die Überprüfungen innerhalb eines Komponententests dienen dazu einen Zustand oder das Ergebnis der aufgerufen Methoden zu überprüfen. Durchgeführt wird diese Überprüfung mittels der `assert` Methoden. Diese besitzen als Parameter zumeist ein getestetes und ein Vergleichsobjekt. Das Testobjekt kann entweder vor der Überprüfung oder in der Überprüfung verändert werden. Ein Sonderfall sind die beiden Methoden `assertTrue` und `assertFalse`. Diese sind Bequemlichkeitsmethoden für `assertEquals(true, Methode())` bzw. `assertEquals(false, Methode())`. Es existieren zwei Fälle von Aufbauarten einer Überprüfung, welche in Beispiel 5.3 dargestellt sind.

Beispiel 5.3: Die zwei Aufbauarten von Überprüfungen

```
boolean value = true;
assertEquals(true, Boolean.xor(value, value));
value = Boolean.xor(value, value);
assertTrue(value);
```

Der erste Fall ist, dass das Objekt vorher erstellt wird und innerhalb der Überprüfung einer Methode aufgerufen wird darauf und der Rückgabewert überprüft wird. Aus diesem Fall folgt die Information über ein Methodenaufruf, der ein Hinweis auf die getestete Methode sein kann. Hierbei muss beachtet werden, dass dies eine Zugriffsfunktion sein kann. Im zweiten Fall wird das Objekt vor der Überprüfung verändert und in der Überprüfung nur noch verglichen ohne verändernden Methodenaufruf. Hieraus folgt die Information des getesteten Objektes, worauf über Rückverfolgung eine getestete Methode gewinnbar sein kann. Das Objekt in der Überprüfung kann ein Objekt einer Standardklasse wie z.B. `String` sein. In diesem Fall ist der Inhalt des Objektes und der Ursprung relevant. Bei einer Zeichenkette wären dies der textuelle Inhalt und die Methode, die diesen zurückgab. Bei einer Zahl kann die Bedeutung z.B. ein HTTP-Fehlernummer sein, was nur mit einem Verständnis der Intention erkennbar und auswertbar wäre. Der Inhalt von diesen Standardobjekt wird deswegen aufgrund der Komplexität nicht in dieser Arbeit betrachtet. Zusammengefasst erlaubt die Überprüfung Rückschlüsse auf die getestete Klasse und auf die getestete Methode.

5.3.8 Auf- und Abbaumethode

JUnit erlaubt die Definition von Auf- und Abbaumethoden. Aufbau- und Abbaumethode sind Methoden mit der Besonderheiten einer Annotation, welche den Ausführungszeitpunkt definiert. Der Ausführungszeitpunkt ist vor jedem Test `@Before`, nach jedem Test `@After`, vor der Klasse `@BeforeClass` oder nach der Klasse `@AfterClass`. Innerhalb des Aufbaus sollen Objekte für die Komponententests erzeugt oder ein Zustand hergestellt werden. Innerhalb des Abbaus kann der Zwischenzustand gespeichert oder aufgeräumt werden, z.B. durch Schließen einer Datenbankverbindung. Wie Komponententestmethoden lassen sich die Methoden in zwei Teile unterteilen: die Methodensignatur und den Rumpf. Die Methodensignatur ist gleich zu der eines Komponententest. Der Rumpf ist gleich zu der einer Methode.

⁹<https://github.com/hamcrest/JavaHamcrest>

Aufbaumethode

Der Methodenname der Aufbaumethode kann beschreiben, welche Objekte erstellt werden. Dann würde dieser Rückschlüsse auf die erstellten Objekte und daraus auf die getesteten Objekte geben könne. Aber in der `JUnit` Dokumentation wird `setUp` oder `setUpClass` verwendet¹⁰. Diese Benennung hat seinen Ursprung in `JUnit3`. In dieser Version war die Aufbaumethode eine Methode mit dem Name `setUp` und wurde überschrieben mit der `@Override` Annotation [HT03]. Dieser Methodenname hat sich deswegen durchgesetzt. Da die erstellten Objekte anhand der Zuweisungen und aufgerufenen Methoden eindeutig ist, liefert der Methodenname keine neuen Informationen. Deswegen können aus dem Methodennamen der Aufbaumethode keine für diese Arbeit verwendbaren Informationen gewonnen werden.

Zuweisungen von Klassenvariablen in der Aufbaumethode einer Komponententestklasse liefern die Information über Kandidaten für die ein Objekt, welches in mehreren Tests verwendet wird. Ein Objekt, welches vor jedem Test gesetzt wird kann eine Voraussetzung für eine Zustandsänderung in einem Tests sein. Auf diesem Objekt kann eine Methode aufgerufen werden zum Verändern des Zustandes oder als Parameter übergeben werden. Im Falle eines Methodenaufrufs auf diesem Objekt liefert dieses Objekt die Methode als Kandidat für die getestete Methode. Falls das Objekt als Parameter übergeben wird, so ist die Methode, die das Objekt als Parameter bekommt, ein Kandidat für die getestete Methode. Diese Methode kann auch nicht die getestete sein, falls z.B. eine Datenbankverbindung in der Aufbaumethode erstellt wird, die Anfrage ansich und nicht der Transport der Anfrage über die Verbindung getestete wird. In beiden Fällen lassen sich Rückschlüsse auf die getestete Klasse und Methode gewinnen. Die getestete Klasse lässt sich anhand des Typs der Variable und die Definitionsklasse der Methode folgern.

Die aufgerufen Methoden in der Aufbaumethode können zur Erstellung der Klassenobjekte dienen. Dieses Methoden können auch teil von Zuweisungen sein. Eine Zuweisung eines Objektes kann in Java über ein Konstruktoraufruf oder eine Methode durchgeführt werden. Deswegen sollten mit den aufgerufenen Methoden mögliche Kandidaten für Klassenobjekte und damit getestete Klasse und Methoden möglich sein. Ähnlich wie bei den Zuweisungen kann eine aufgerufene Methode auch zur Erstellung des Zustandes vor der Veränderung gedient haben und nicht die getestete Methode sein. Insgesamt sollten trotzdem Rückschlüsse auf die getestete Methode und Klasse möglich sein.

Für die Dokumentation der Aufbaumethode gelten die selben Folgerungen und Einschränkungen wie bei der Methodendokumentation.

Abbaumethode

Der Methodenname der Abbaumethode ist ähnlich zu dem der Aufbaumethode. Der Name sollte beschreiben, was innerhalb der Abbaumethode durchgeführt wird. Dies kann z.B. das Speichern des Ergebnisses oder Schließen von Verbindungen sein. Anhand dessen wäre mit der selben Begründung wie bei der Aufbaumethode ein Rückschluss möglich. Der Methodenname der Abbaumethode besitzt aber wie der Name der Aufbaumethode Hindernisse in der Informationsgewinnung. Der Unterschied ist, dass der Name `tearDown` statt `setUp` ist. Deswegen können aus dem Methodennamen der Abbaumethode keine nützlichen Informationen gewonnen werden.

In der Abbaumethode können Klassenvariablen mittels Zuweisungen zurückgesetzt werden. Dieses Zurücksetzen von Klassenvariablen ist ein Hinweis für ein Objekt, was innerhalb des Tests verwendet wurde und damit Rückschlüsse auf die getestete Methode und Klasse

¹⁰<https://github.com/junit-team/junit4/wiki/test-fixtures>, angesehen am 19.10.2019

Tabelle 5.1: Vorhandene Informationen im Quelltext, ihre Art und ihre Quelle

Quelle	Information	Art	Testklassen	Hauptklassen
Paketbezeichner	Paketbezeichner	textuell	X	X
Importanweisungen	Importanweisungen	textuell	X	X
Klassenbezeichner	Klassenname	textuell	X	X
Dokumentation	Klassendokumentation	textuell	X	X
	Methodendokumentation	textuell	X	X
	Kommentare	textuell	X	X
Methoden	Methodenname	textuell	X	X
	Zuweisungen	semantisch	X	X
	Bezeichner	textuell	X	X
	aufgerufene Methoden	semantisch	X	X
	Überprüfungen	semantisch	X	
Aufbaumethode	Methodenname	textuell	X	
	Zuweisungen	semantisch	X	
	Methodenaufrufe	semantisch	X	
Abbaumethode	Methodenname	textuell	X	
	Zuweisungen	semantisch	X	
	Methodenaufrufe	semantisch	X	

erlauben kann. Es gelten die selben Folgerungen wie bei der Analyse der Zuweisungen innerhalb einer Aufbaumethode.

Die aufgerufene Methoden innerhalb der Abbaumethode werden meist dazu verwendet entweder den Zustand zu speichern oder Ressourcen freizugeben. Dies kann z.B. das Speichern eines Bildes nach einem Test sein. Für die aufgerufenen Methoden in der Abbaumethode gelten die selben Folgerungen wie in für die aufgerufene Methoden in der Aufbaumethode.

Für die Dokumentation der Abbaumethode gelten die selben Folgerungen und Einschränkungen wie bei der Methodendokumentation.

5.4 Zusammenfassung der gewonnenen Informationen aus den einzelnen Informationsquellen

Wie in Tabelle 5.1 erkenntlich ist, existieren verschiedene Informationsquellen mit Teilinformationen. Diese wurden im vorherigen Abschnitt erklärt und analysiert. Aus diesen Teilinformationen folgen verschiedene Rückschlüsse, welche in Tabelle 5.2 zusammengefasst ist. Es ist erkenntlich, dass keine Informationsquelle direkt semantisch zusammengehörige Methoden als Information liefert, sondern nur Hinweise darauf. Diese Hinweise werden als Rückschlüsse bezeichnet. Es werden vier verschiedene Arten von Rückschlüssen gewonnen:

- Rückschlussart 1 verwendete Methoden
- Rückschlussart 2 getestete Klasse
- Rückschlussart 3 getestetes Objekt
- Rückschlussart 4 getestete Methode

Es ist also erst notwendig einzelne Verfahren zu entwickeln, die diese Rückschlüsse extrahieren und dann eine Zusammenführung der Rückschlüsse. Es werden erst die einzelnen Verfahren entworfen und dann ein Gesamtverfahren erstellt.

Tabelle 5.2: Rückschlüsse aus den Information nach der Auswertung

Quelle	Information	Kandidaten für		
		getestete Klasse	getestete Methode	getestetes Objekt
Paketbezeichner	Paketbezeichner	x		
Importanweisung	Importanweisung	x		
Klassenname	Klassenname	x		
Testmethode	Methodenname		x	
Testmethode	Kontrollfluss		x	
Testmethode	aufgerufene Methoden	x	x	
Testmethode	Überprüfung	x	x	x
Aufbaumethode	aufgerufene Methoden		x	
Aufbaumethode	Zuweisung		x	x
Abbaumethode	aufgerufene Methoden		x	
Abbaumethode	Zuweisung		x	x

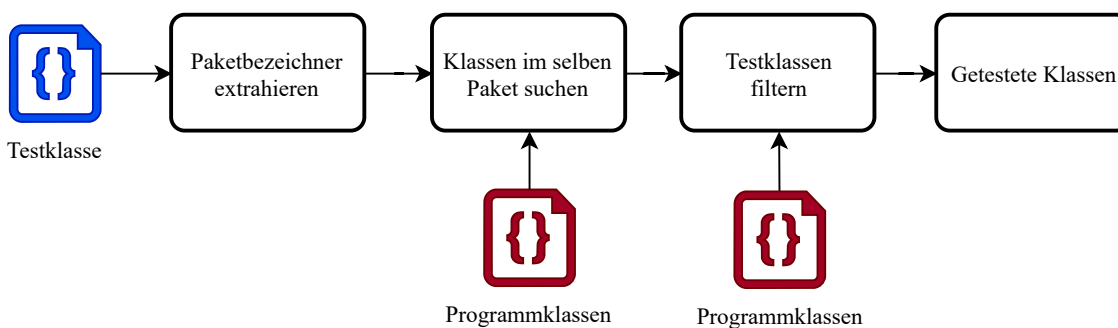


Abbildung 5.1: Verfahren zur Auswertung des Paketbezeichners einer Testklasse.

5.5 Auswertungsverfahren der einzelnen Informationsquellen

Nach der Analyse der einzelnen Informationsquellen und ihrer Rückschlüsse werden nur Verfahren dafür entworfen. Um aus den Information die analysierten Rückschlüsse zu gewinnen sind verschiedene Verfahren notwendig. Die einzelnen Verfahren müssen die Art der Information, also textuell und semantisch, berücksichtigen. Diese beiden Arten werden verschieden verarbeitet. Zeitgleich können manche Verfahren die Ausgangsmenge einschränken, wie z.B. über die verwendeten Klassen.

Paketbezeichner

Wie im vorherigen Kapitel bereits analysiert kann aus den Paketbezeichner die Kandidatenmenge der getesteten Klasse gewonnen werden. Für die Suche wird ein dreistufiges Verfahren verwendet, welches in Abbildung 5.1 dargestellt ist. Als erstes wird der Paketbezeichner der Testklasse extrahiert. Dann werden alle Klassen innerhalb des Pakets gesucht. Als letztes werden dann die Testklassen aus der Menge entfernt, um nur noch die Programmklassen zu haben. Das Ergebnis ist eine Kandidatenmenge an getesteten Klassen.

Importanweisungen

Wie bereits vorgestellt, kann aus den Importanweisungen die Information der verwendeten Klassen gewonnen werden. Die Analyse dieser wird durch ein dreistufiges Verfahren durchgeführt, welches in Abbildung 5.2 dargestellt ist. Als erstes werden alle Importanweisungen

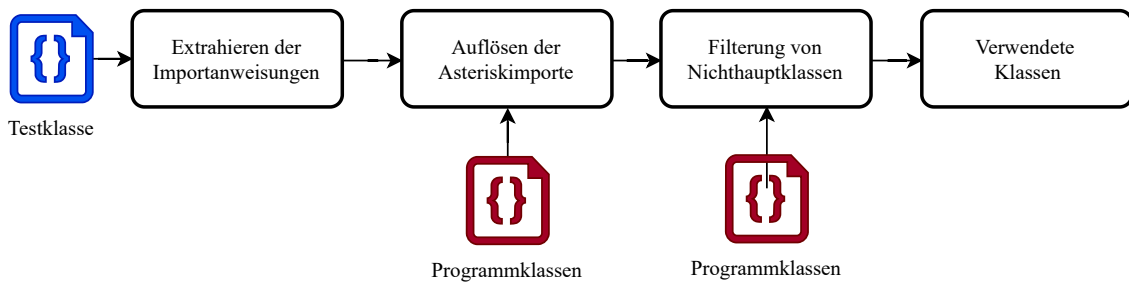


Abbildung 5.2: Verfahren zur Auswertung der Importanweisungen einer Testklasse.

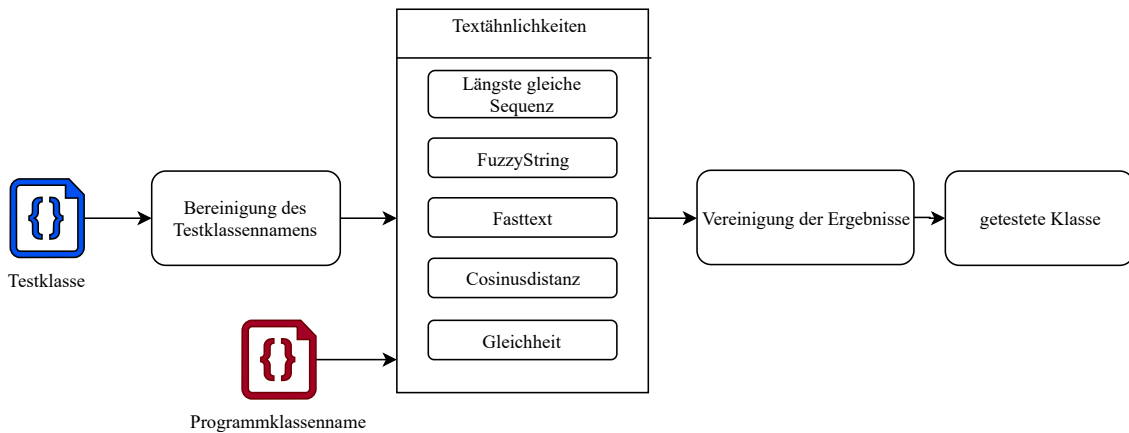


Abbildung 5.3: Verfahren zur Auswertung der Klassennamen einer Testklasse.

ausgelesen. Dann werden die Asteriskimportanweisungen aufgelöst, indem alle Klassen aus dem Paket betrachtet werden. Als letztes werden alle Klassen die nicht teil des Programms sind entfernt. Das Ergebnis ist eine vollständige Liste der verwendeten Klassen.

Klassennamen

Wie im vorherigen Kapitel vorgestellt wurde, kann aus dem Klassennamen einer Testklasse die Information der getesteten Klassen gewonnen werden. Diese Auswertung wird mittels eines mehrstufigen Verfahrens, welches in Abbildung 5.3 dargestellt ist, durchgeführt. Da der Klassenname textueller Natur ist findet erst eine Vorverarbeitung statt. Hierbei werden die genannten Benennungsregeln für Testfälle rückgängig gemacht, sodass nur noch der Klassenname ohne Zusatz wie z.B. „test“ vorhanden ist. Die Bereinigung entfernt mittels dem regulären Ausdruck „(?!i)test“ alle Fragmente die darauf passen (engl. matchen) ohne Berücksichtigung der Groß- und Kleinschreibung. Im nächsten Schritt wird der Klassenname der Testklasse mit allen anderen Namen der Programmklassen verglichen. Für den Vergleich werden verschiedene Zeichenkettenalgorithmen verwendet, welche in Kapitel 2 erläutert sind. Diese sind ein einfacher textueller Vergleich, fastText, Kosinusdistanz, JaroWinkler, längstes gleiches Teilwort und Fuzzystring. Der einfache textuelle Vergleich hat den Vorteil, dass das Ergebnis exakt ist, falls die Klasse exakt wie die getestete Klasse heißt. fastText wird verwendet, weil die Ähnlichkeit von Teilfragmenten hierbei bevorzugt wird. Wörter die aus ähnlichen Teilfragmenten bestehen könnten immer noch auf eine Ähnlichkeit hindeuten. Die Kosinusdistanz und JaroWinkler haben den Vorteil, dass diese mit Tippfehlern und leicht verschiedenen Worten noch gute Ergebnisse produzieren. Dies liegt an der Unschärfe für die Position der gleichen Buchstaben. Fuzzystring und längstes gleiches Teilwort ist dann erfolgsversprechend wenn Klassenname und Testklassenname an gleichen Stellen im Wort die selben Buchstaben haben. Dies funktioniert sehr gut wenn das eine Wort ungefähr Teilwort des anderen ist und Worte gleiche Textfragmente be-

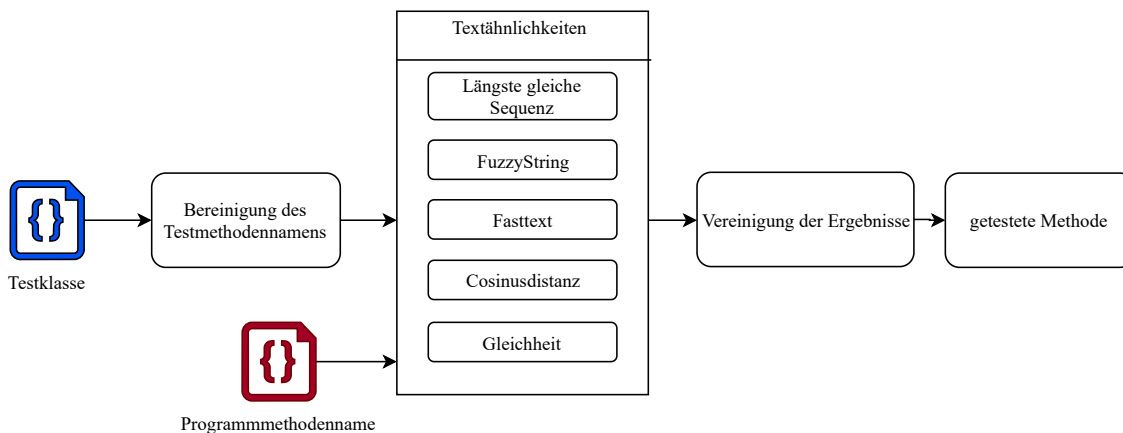


Abbildung 5.4: Verfahren zur Auswertung der Methodenname einer Testklasse.

sitzen. Für die Vergleiche haben die Autor in [CKV19] eine Algorithmus vorgeschlagen, der eine Schnittmenge von verschiedene Zeichenkettenalgorithmen ist. Diese Schnittmenge ist, aber schlechter als eine Zusammenführung der verschiedenen Algorithmen. Dies hat zwei Gründe: Erstens haben die verschieden Algorithmen in dieser Arbeit eine hohe Ähnlichkeit als Minimum festgelegt. Somit sind die Ergebnismengen der Algorithmen kleiner als in dem Papier und eine Schnittmenge würde die leere Menge produzieren. Zweitens werden für die Zuordnung nur die Klassennamen der im Quelltext verwendeten Typen genutzt, die Teil der Hauptklassen sind und nicht von einer Programmabhängigkeit oder der Standardbibliothek stammen. Somit ist die mögliche Ergebnismenge nochmal kleiner als in dem Papier. Im letzten Schritt werden die Ergebnismengen deswegen vereinigt und bilden eine Liste von Klassennamen der Kandidaten für die getestete Klasse.

Methodenname

Aus dem Methodenname einer Testmethode kann die Information der getesteten Methode gewonnen werden, wie im vorherigen Kapitel vorgestellt. Die Analyse wird wie bei Klassennamen durchgeführt mit der Änderung, dass statt Klassennamen nur die Namen der aufgerufenen Methoden für die Zuordnung übergeben werden. In Abbildung 5.4 ist dies dargestellt. Das Ergebnis ist eine Liste an Kandidaten für die getestete Methode und ihre Definitionsklassen.

Aufgerufene Methoden

Aus den aufgerufenen Methoden in einem Komponententest wird die Information der verwendeten Methoden gewonnen. Diese bilden eine Grundmenge der möglichen zusammengehörigen Methoden. Die Analyse dieser wird in einem dreistufigen Verfahren durchgeführt, welches in Abbildung 5.5 dargestellt ist. Als erstes werden alle aufgerufenen Methoden innerhalb der Testmethode extrahiert. Dann werden mithilfe der Methoden, die teil der Hauptklassen sind, alle aufgerufenen Methoden, die nicht teil der Hauptklassen sondern einer Abhängigkeit sind, gefiltert. Zusätzlich werden alle Methoden, die mit `get` oder `is` beginnen entfernt. Diese Methodenname sind ein Zeichen für eine Zugriffsfunktion. Das Ergebnis sind die verwendeten Methoden innerhalb des Komponententests. Dieses Methoden sind dann Kandidaten für die semantisch zusammengehörigen Methoden.

Überprüfung

Aus den Überprüfungen in einem Komponententest lassen sich Rückschlüsse auf die getestete Methode, die getestete Klasse und das getestete Objekt ziehen. Dies ist abhängig

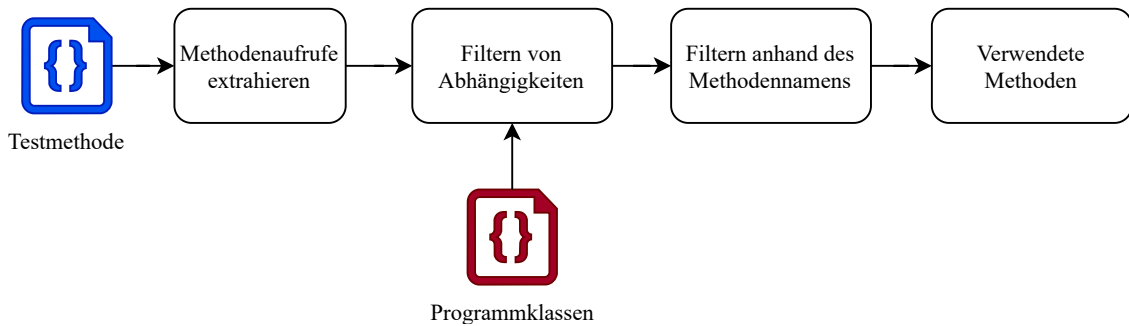


Abbildung 5.5: Verfahren zur Auswertung der Methodenaufrufe einer Testmethode.

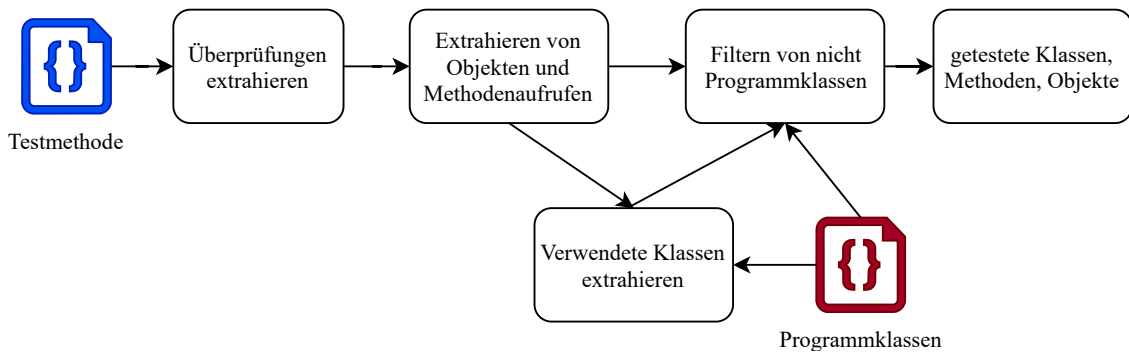


Abbildung 5.6: Verfahren zur Auswertung der Überprüfungen einer Testmethode.

von dem Aufbau der Überprüfung. Die Analyse wird durch mehrstufiges Verfahren umgesetzt, welches in Abbildung 5.6 dargestellt ist. Als erstes werden die Überprüfungen der Testmethode extrahiert. Als zweites werden aus den Parametern der Überprüfung alle Methodenaufrufe und verwendeten Objekte extrahiert. Von den Objekten wird dann der Typ extrahiert. Aus den Methodenaufrufen wird die Definitionsklasse und der Aufrufkontext extrahiert. Mittels dem Aufrufkontext und den Objekten werden dann Methoden gesucht, die diese innerhalb des Tests als Aufrufkontext oder Parameter nutzen. Aus den Objekten und Methodenaufrufen werden dann die Klassen, wo diese definiert sind, extrahiert. Als letztes werden dann alle Objekte, Klassen und Methoden, die nicht teil des Hauptprogramms sind, entfernt. Übrig bleibt eine Menge an Kandidaten für getestete Methoden, Klassen und Objekten.

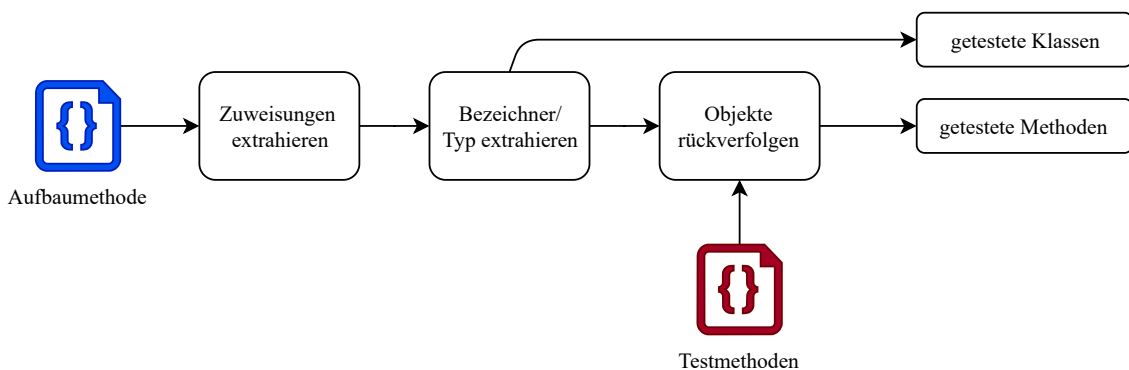


Abbildung 5.7: Verfahren zur Auswertung der Zuweisungen einer Aufbaumethode.

Aufbaumethode

Die Zuweisungen der Aufbaumethoden werden mit einem dreistufigen Verfahren ausgewertet, welches in Abbildung 5.7 dargestellt ist. Im ersten Schritt werden alle Zuweisungen extrahiert. Dann werden die Bezeichner und Typen der zugewiesenen Objekte extrahiert. Mittels dem Bezeichner und den Typen werden dann alle Methoden aus allen Testmethoden extrahiert, die auf diesem Objekt eine Methode aufrufen oder es als Parameter übergeben. Das Ergebnis sind die Kandidaten für die getestete Klasse und Methoden.

Die aufgerufene Methoden in der Aufbaumethode werden auf die gleiche Weise wie die Methoden einer Testmethode ausgelesen.

Abbaumethode

Die Zuweisungen und aufgerufene Methoden der Abbaumethoden werden auf die gleiche Weise wie bei der Aufbaumethoden ausgewertet.

5.6 Entwurf des Gesamtverfahrens

Nachdem die einzelnen Verfahren für die Auswertung der Informationsquellen vorgestellt wurden, wird aus diesen Information das endgültige Ergebnis erstellt. In Tabelle 5.2 sind die Informationen nochmal zusammengefasst. Die Quelle ist die Informationsquelle innerhalb der Testklasse, die Information ist ein Teil der Informationsquelle und der Rückschluss ist das Ergebnis daraus. Nun sind verschiedenen Rückschlüsse vorhanden, deren Gewinnung aber keine direkten Informationen über semantisch zusammenhängende Methoden. Deswegen muss aus den vorhanden Informationen ein Verfahren entworfen werden. Da die einzelnen Rückschlüsse, aber ein Hinweis auf die Zusammengehörigkeit sein können, ist eine Heuristik als Verfahren denkbar. Deswegen werden den einzelnen Informationen verschiedene Konfidenzwerte gegeben. Diese sind höher, je genauer die Information ist. Eine Methode die in der Überprüfung aufgerufen wird ist z.B. ein deutlich feingranularer Information als Klassen die im selben Paket sind. Um diese Konfidenzwerte zu verteilen auf die Methoden wird erst eine Grundmenge erstellt.

Die Grundmenge aller möglichen zusammengehörigen Methoden bilden dabei die aufgerufenen Methoden. Aus diesen werden alle möglichen Kombination von Paaren erstellt. Zwei Methoden die in keinem Komponententest miteinander vorhanden sind, sind für dieses Verfahren nicht als zusammengehörig erkennbar. Dann werden aus dieser Grundmenge alle Paare entfernt, welche beide in der selben Klassen definiert sind. Klassen sollten in sich zusammengehörig sein und deswegen sind diese Paare nicht relevant für das Verfahren. Für jeden Rückschluss wird nun der Konfidenzwerte erhöht, falls ein Paar die selbe Methode oder Klasse hat. Nun besitzt jedes Methodenpaar ein Konfidenzwert. Dieses gibt an, für wie wahrscheinlich das Verfahren die zusammengehörige hält. Der Konfidenzwert nimmt Werte aus dem Intervall $[0, 1]$ an. Eins steht hier für sehr wahrscheinlich, null für keine Zusammengehörigkeit.

6 Implementierung

Im folgenden Kapitel wird die Implementierung des im Entwurf vorgestellten Verfahrens vorgestellt. Das Programm zur Lösung des Problems besteht aus 3 Phasen:

Phase 1 Zerteilen und Transformieren des Quelltextes

Phase 2 Gewinnung der Informationen aus einzelnen Quelltextteilen

Phase 3 Auswertung der Informationen

Für diesen Aufbau wurde sich entschieden, weil das Zerteilen des Quelltextes, später in dem Projekt, durch eine andere Arbeit ersetzt wird und dies nur vorläufig ist. Die Trennung der Auswertung und der einzelnen Verfahren erlaubt das Hinzufügen von weiteren Verfahren mit geringem Aufwand. Die einzelnen Phasen bauen aufeinander auf. In der Phase des Zerteilens wird der Quelltext transformiert und währenddessen für das Verfahren wichtige Informationen extrahiert. Die Darstellung dessen und das Vorgehen wird erklärt im Verlauf dieses Kapitels erklärt. In der Phase der Informationsgewinnung werden mittels verschiedener `Matcher` Rückschlüsse aus dem transformierten Quelltext gewonnen. Zum Schluss werden die verschiedenen Rückschlüsse zusammengeführt und die im Entwurf vorgestellt Aufteilung der Konfidenzen für die einzelnen Rückschlüsse verwendet.

6.1 Zerteilen und Transformieren des Quelltextes

Mittels `Spoon` [PMP⁺15] wird der Java Quelltext aus beliebig vielen Pfaden eingelesen und der `Spoon`-Syntaxbaum des Programms erstellt. Dieser ist ein Baum, der aus allen abstrakten Syntaxbäumen des Programms besteht und Typinformationen enthält. Dieser Baum wird dann mittels einer Tiefensuche traversiert und dabei die `Spoon`klassen in eine eigene Darstellung transformiert. Diese Transformation wird mittels einem Besucher durchgeführt, welcher später vorgestellt wird. Aus dieser Transformation entsteht dann ein eigener Baum, der nur noch die für diese Arbeit relevanten Informationen enthält. Der Baum wird durch zwei Klassen und eine Schnittstelle repräsentiert, welche in Abbildung 6.1 dargestellt sind. Bei der Tiefensuche werden zwei verschiedene Knotenarten erstellt, `SourceClassNode` und `PackageNode`. Die Knoten besitzen die für die Konstruktion eines Baumes notwendigen Methoden wie `einfügen` und `verbinden`. Durch diese Methoden wird der Baum rekursiv, von der Wurzel startend, aufgebaut während dem Zerteilen. Die beiden Knotenarten stellen die Unterscheidung zwischen einem Paket, also einem Ordner und einer Quelltextklasse also einer Java-Datei dar. Beim Zerteilen wird beim Erreichen eines Objektes zwischen einer Datei oder eines Ordners unterschieden. Je nach Typ

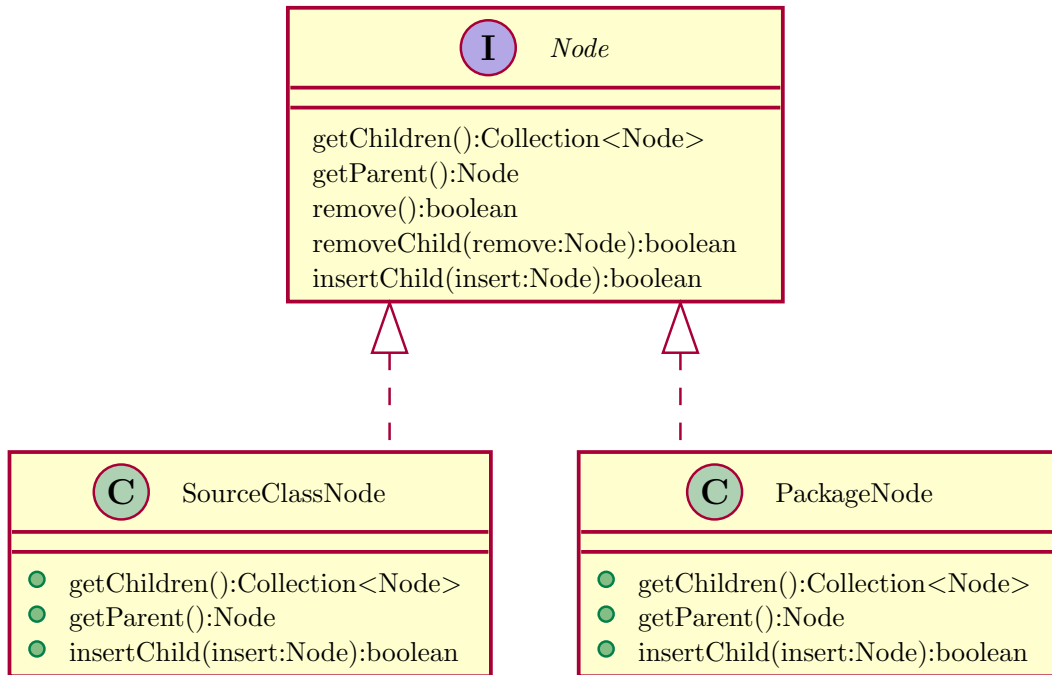


Abbildung 6.1: Schematische Darstellung des Quelltextbaumes

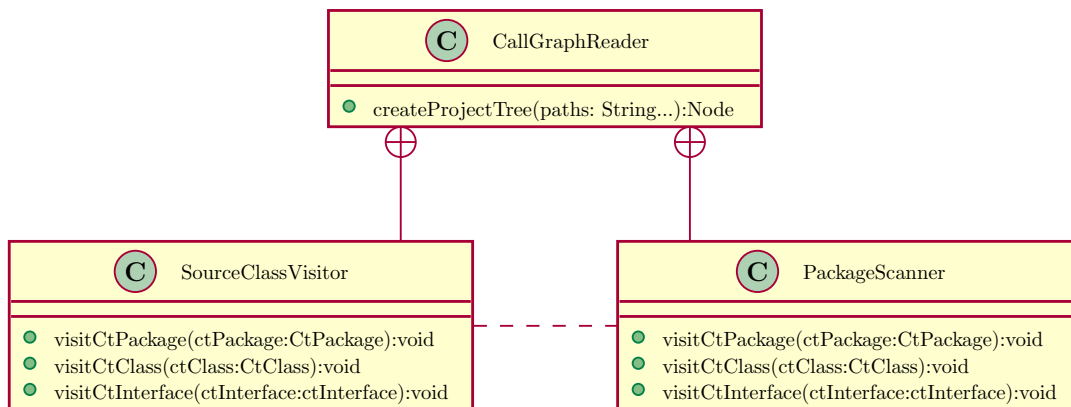


Abbildung 6.2: Schematische Darstellung des Zerteilungsvorgangs

des Objektes wird dann die passende Knotenart erstellt. Die Tiefensuche wird von dem `PackageScanner` durchgeführt, welcher in Abbildung 6.2 dargestellt ist. Dieser besitzt zwei innere Klassen, den `PackageScanner` für die Erstellung der `PackageNode` und den `SourceClassVisitor` für die Erstellung der `SourceClassNode`. Die beiden Besucher besitzen Methoden für Klassen, Schnittstellen und Paketdefinitionen. Der `PackageScanner` nutzt den `SourceClassVisitor` bei dem Besuchen einer Klassendefinition. Der `SourceClassVisitor` extrahiert alle Informationen über den Klassennamen, die Dokumentation, die Paketdefinition, die Importe, die Klassenfelder und die Methoden. Aus diesen Methoden werden dann Informationen, wie der Methodenname, die Parameter und die Annotationen extrahiert. Die Testmethoden werden dann mittels eines `StatementVisitor` und einem `ExpressionVisitor` zerteilt. Die Ergebnisklassen der Besucher sind in Abbildung 6.3 dargestellt. Die einzelnen Klassen besitzen eine gemeinsame Schnittstelle, um eine Liste aller Operationen innerhalb des Methodenrumpfs zu erstellen. Diese besitzen die notwendigen Informationen, um mit diesen weiterzuarbeiten. Ein Methodenaufruf besitzt z.B. den Aufruf, die Klasse der Definition, seine Parameter, den Aufrufkontext und den Name des Aufrufkontextes.

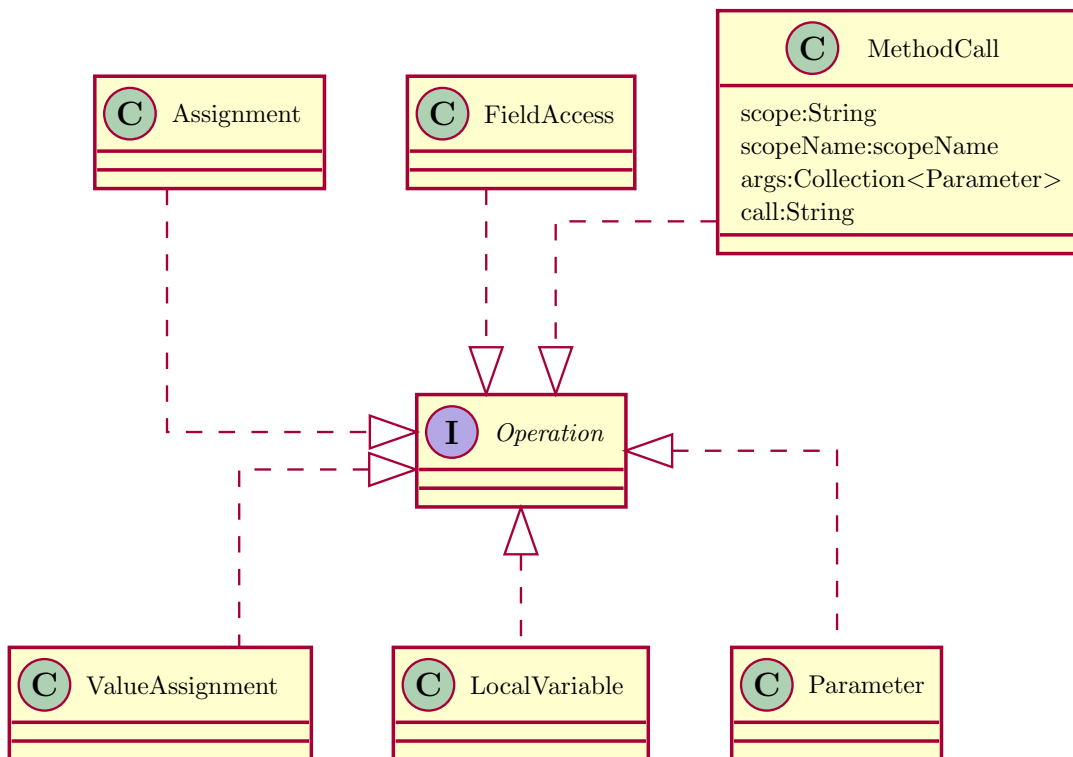


Abbildung 6.3: Schematische Darstellung des Datenhaltung

6.2 Gewinnung der Informationen aus den Quelltextabschnitten

Nachdem der Quelltext zerteilt und transformiert wurde, werden die Klassen einzeln weiterverarbeitet mittels verschiedener *Matcher*. Diese *Matcher* sind die im Entwurf vorgestellten Teilverfahren zur Auswertung einzelner Informationen wie z.B. dem Klassennamen. Die Verwaltung dieser übernimmt eine Steuerungseinheit, welcher die verschiedenen *Matcher* auf jeden Testmethode anwendet. Diese ist in Abbildung 6.4 dargestellt. Als erstes wird von der Steuerungseinheit mittels des *NodeSplitters* in Hauptklassen und Testklassen aufgeteilt. Die *Matcher* lassen sich in vier verschiedene Arten einteilen: *Ast-Matching*, *ClassNameMatching*, *MethodNameMatching* und *PackageMatching*. Die *Ast-Matcher* analysieren den Methodenrumpf. Die werten die Überprüfungen, die Aufbau- und Abbaumethoden, wie in Kapitel 5 vorgestellt, aus und erstellen Rückschlüsse daraus. Hierbei werden sowohl die Überprüfungen als auch die Aufbau- und Abbaumethoden ausgewertet. Der *ClassNameMatcher* und *MethodNameMatcher* arbeiten auf Zeichenketten und berechnen textuelle Ähnlichkeit. Die textuelle Ähnlichkeit wird mittels Zeichenkettenähnlichkeitsalgorithmen aus *Apache Commons*¹ und *fastText*² berechnet. *FuzzyScore*, *Längste Gleiche Teilsequenz* und *JaroWinkler* werden aus *Apache Commons* geladen und *fastText* aus *fastText*.

fastText

Als Trainingskorpus für *fastText* wurden alle Methoden bzw. Klassennamen verwendet. Es wurden zwei Modelle trainiert, eines für Methoden und eines für Klassennamen. Die Klassennamen wurden für das Training nicht qualifiziert verwendet, d.h. ohne den vollständigen Paketbezeichner. Dies liegt daran, dass sonst ein Großteil des qualifizierten Klas-

¹<https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/SimilarityScore.html>

²<https://fasttext.cc/>

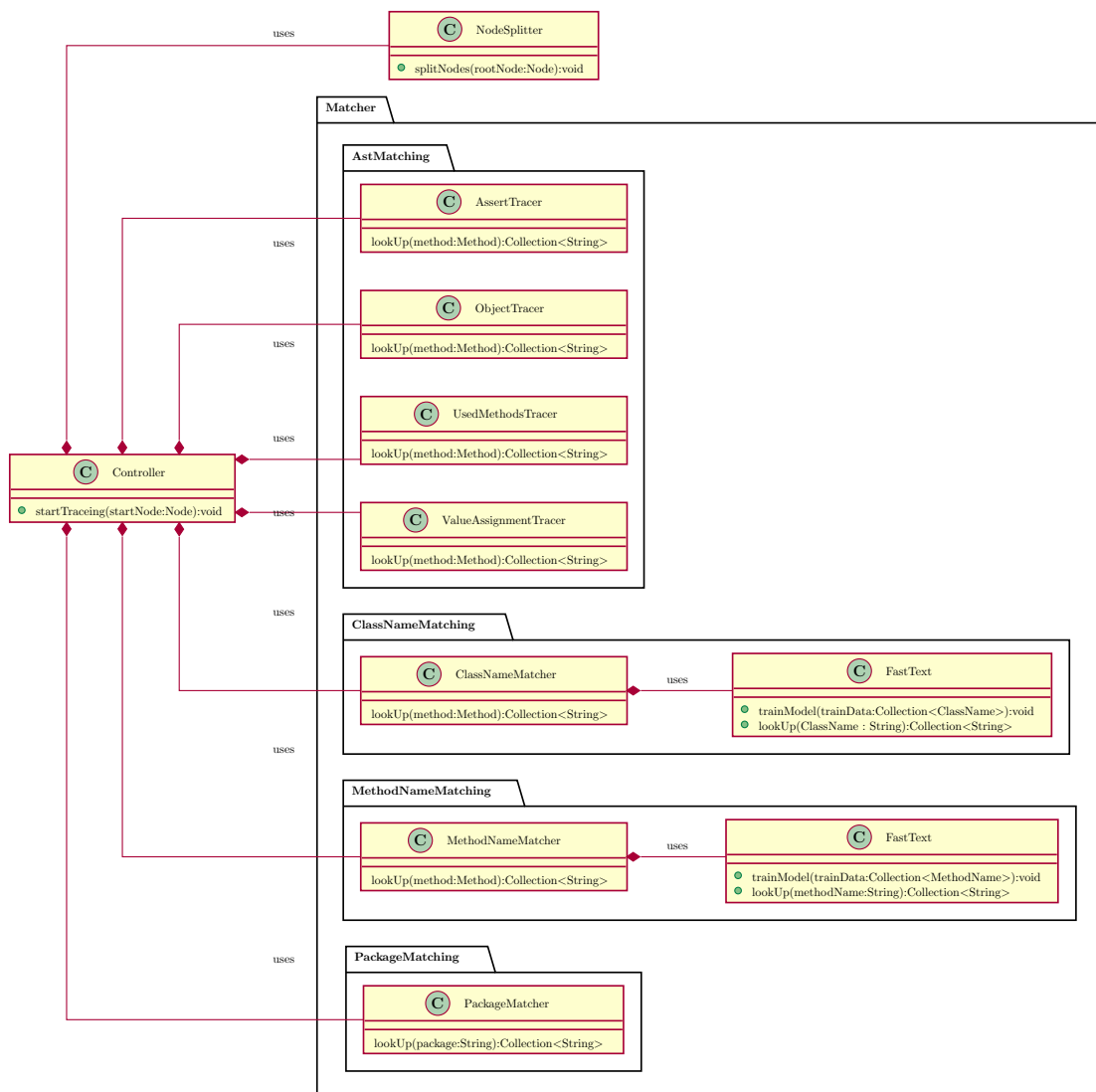


Abbildung 6.4: Schematische Darstellung der Informationsgewinnung

sennamen ähnlich wäre und das Ergebnis schlechter. Es wurde sich gegen die bereits trainierten Vektoren von Facebook entschieden. Diese würden Wörter vorhersagen, die nicht teil des Korpus wären. Zusätzlich würde sie länger trainiert werden müssen, weil die vortrainierten Vektoren eine deutlich größere Größe besitzen. Die Vorhersagen von Wörtern außerhalb des Korpus war für dieses Verfahren nicht notwendig und hätte das Ergebnis verschlechtert. Eine Rückführung von einem Wort außerhalb des Korpus zu einem Wort des Korpus wäre möglich über eine Ähnlichkeitsabfrage von dem Testklassennamen gegen alle Klassennamen. Wenn also der Testklassenname zu dem selben Wort ähnlich wäre wie ein Hauptklassenname, so wäre dieser zuordenbar. Dieser Aufwand wurde aus genannten Gründen vermieden und gibt den zusätzlichen Vorteil, dass alle Ähnlichkeitsalgorithmen unabhängig von der Textsprache sind.

Nachdem alle *Matcher* ausgeführt wurden, werden die Ergebnisse zu den Methoden zurückgeschrieben. In Quelltextausschnitt 6.1 sind die Ergebnisse der verschiedenen *Matcher* dargestellt. Die Klassen im selben Paket werden z.B. von dem *PackageMatcher* erzeugt. Das Zurückschreiben wird von der Steuerungseinheit ausgeführt, die *Matcher* werden nur angewandt.

Quelltextausschnitt 6.1: Ergebnisse der Matcher

```

private Collection<Pair<ClassName, String>> matchingMethodNames;
private Collection<Pair<String, String>> testedObjectsWithIdentifier;
private Collection<MethodCall> methodsUsedInBefore;
private Collection<MethodCall> methodsUsedInAfter;
private Collection<Pair<String, String>> fieldsInBefore;
private Collection<Pair<String, String>> fieldsInAfter;
private Collection<MethodCall> methodsUsingFieldsInBefore;
private Collection<MethodCall> methodsUsingFieldsInAfter;
private Collection<MethodCall> methodsUsingObjectOfAssert;
private Collection<String> testedClassInAssert;
private Collection<MethodCall> testedMethodInAssert;
private Collection<ClassName> matchingClassNames;
private Collection<MethodCall> usedMethodsInTestCase;
private Collection<ClassName> classesInSamePackage;

```

Tabelle 6.1: Rückschlüsse mit den Konfidenzwerte im Auswertungsverfahren

Rückschluss	Konfidenzwert
Gleiches Paket	0.05
Ähnlicher Klassenname	0.2
Ähnlicher Methodenname	0.3
Methoden in @Before	0.1
Methoden in @After	0.1
Felder in @Before	0.1
Felder in @After	0.1
Methoden die Felder nutzen in @Before	0.1
Methoden die Felder nutzen in @After	0.1
Methoden die das Objekt in der Überprüfung nutzen	0.2
Getestete Klasse in der Überprüfung	0.1
Getestete Methode in der Überprüfung	0.1

6.3 Auswertung der Informationen

In der letzten Phase werden die Informationen, die extrahiert wurden, ausgewertet und zusammengeführt. Dafür werden alle verwendeten Methoden innerhalb einer Testmethode ausgelesen und das kartesische Produkt der Methoden gebildet.

Beispiel 6.1: Methodenpaar nach Ausführung des kartesisches Produktes

```

[call=setDescription, confidenceScore=0,
definitionClass=org.apache.commons.mail.EmailAttachment]
[call=getToAddresses, confidenceScore=0,
definitionClass=org.apache.commons.mail.mocks.MockHtmlEmailConcrete]

```

In Beispiel 6.1 ist ein Methodenpaar dargestellt. Der `call` ist die aufgerufene Methode, die `definitionClass` die Klasse in der die Methode implementiert ist. Der `confidenceScore` gibt an wie wahrscheinlich die semantische Zusammengehörigkeit des Methodenpaares ist. Dann werden mit den vorher gewonnenen Rückschlüssen die Konfidenzwerte der Methodenpaare erhöht. Diese Rückschlüsse mit ihrem Konfidenzwert sind in Tabelle 6.1 dargestellt.

Für diese Werte wurde eine Teilmenge aus der Evaluationssoftware entfernt und die Parameter damit trainiert. Deswegen sind diese Werte so gewählt. Diese verschiedenen Rückschlüsse werden nun auf die Methodenpaare zugeordnet und im Falle einer Zuordnung die Konfidenz mit dem jeweiligen Wert für den Rückschluss erhöht. Diese Zuordnung wird mittels der `ResultCalculation` durchgeführt. Diese wird von der Steuerungseinheit aufgerufen. Das Ergebnis sind die semantisch zusammengehörigen Methodenpaare mit einem Konfidenzwert.

7 Evaluation

Im folgendem Abschnitt wird das Verfahren auf Testdaten evaluiert und verschiedene Metriken anhand der Ergebnisse berechnet. Dafür werden erst die Kriterien und Metriken der Evaluation erklärt. Dann wird die Datengrundlage und die Erstellung dieser erläutert. Danach werden die Softwareprojekte vorgestellt und einzeln die Ergebnisse besprochen. Als Abschluss wird ein Fazit gezogen und die Frage „Ist die Erkennung von semantisch zusammengehörigen Methodenpaaren anhand von Komponententests möglich?“ beantwortet.

7.1 Kriterien der Evaluation

Um den Erfolg dieses Verfahrens zu evaluieren werden drei Messgrößen verwendet. Diese Messgrößen werden anhand der Methodenpaare in der Ergebnismenge berechnet. Die gefundenen Methodenpaare werden in drei Arten kategorisiert:

- Richtige Methodenpaare(rt) sind sowohl in der Musterlösung also auch in der Ergebnismenge des Verfahrens.
- Falsche Methodenpaare(wt) sind nicht in der Musterlösung aber in der Ergebnismenge des Verfahrens.
- Fehlende Methodenpaare(mt) sind in der Musterlösung aber nicht in der Ergebnismenge des Verfahrens.

Anhand dieser drei Arten von Methodenpaaren lassen sich die drei Messgrößen berechnen, Präzision, Ausbeute und F-Maß. Die Präzision ist ein Maß für wie korrekt die Ergebnismenge ist. Diese wird berechnet anhand der Gleichung 7.1.

$$\text{Präzision} = \frac{|rt|}{|rt + wt|} \quad (7.1)$$

Die Ausbeute als Messgröße wie vollständig das Ergebnis ist. Diese wird berechnet anhand der Gleichung 7.2.

$$\text{Ausbeute} = \frac{|rt|}{|rt + mt|} \quad (7.2)$$

Das F-Maß zur Bestimmung einer globalen Messzahl, die aus Präzision und Ausbeute berechnet wird. Dieses Maß erlaubt die Vergleichbarkeit zu anderen Informationsgewinnungsverfahren. Diese wird berechnet anhand der Gleichung 7.3.

$$\text{F-Maß} = 2 * \frac{\text{Präzision} * \text{Ausbeute}}{\text{Präzision} + \text{Ausbeute}} \% \quad (7.3)$$

Tabelle 7.1: Programmgröße von JPacman aufgeteilt in Testteil und Hauptteil

	Klassen	Methoden	Quelltextzeilen
Haupt	46	201	2013
Test	14	54	613
Gesamt	60	255	2626

7.2 Datenmenge

Zur Evaluation des entwickelten Verfahrens ist eine Datenmenge notwendig. In diesem Forschungsbereich sind dem Autor zum Zeitpunkt der Arbeit keine öffentlichen Datenmengen von Komponententests zu semantisch zusammengehörigen Methoden bekannt. Deswegen wurde eine eigene Datenmenge erstellt. Die ausgewählte Software soll ein möglichst breites Spektrum an Software abdecken, hinsichtlich Größe, Qualität und Ursprung. Es wurde Software ausgewählt, die sowohl semantisch zusammengehörige Methoden beinhaltet als auch diese an Komponententests erkennbar sind. Für diese Einschränkungen sind Komponententests notwendig, die das Zusammenspiel zweier Klassen testen und nicht nur Veränderungs- und Zugriffsfunktionen einer Klasse testen. Wegen der zeitlichen Einschränkungen wurden nur drei verschiedene Projekte ausgewählt, JPacman, Roller und Gitiles. Diese drei Projekte wurden gewählt um verschiedene Größen von Softwareprojekten, Entwicklungshintergründe und Softwareaufbau evaluiert zu haben.

Nach der Auswahl der Software wurde die Grundwahrheit(Gold Standard/engl. Ground Truth) erstellt. Ein Methodenpaar gilt als semantisch zusammengehörig, wenn eine umgesetzte Funktionalität benennbar ist und es nicht zwei Zugriffsfunktionen sind. Für die Markierung wurden nur die Komponententests betrachtet und nicht der Hauptquelltext. Dies wurde gemacht, weil das Verfahren nur auf den Komponententests arbeitet. Aus dem Hauptquelltext werden nur Klassen und Methodennamen extrahiert. Semantisch zusammengehörige Methoden, die nicht an Komponententests erkennbar sind, sind von diesem Verfahren nicht erkennbar. Deswegen wurden nur die Testklassen betrachtet. Obwohl nach diesem Kriterium gearbeitet wurde, ist ein Fehler in der Datenmenge nicht auszuschließen, da die semantische Zusammengehörigkeit zweier Methoden eine subjektive Entscheidung ist. Die Daten wurden von dem Autoren selber annotiert. Dies war Aufgrund der zeitlichen Anforderungen und Art dieser Arbeit nicht anders durchführbar. Für eine Markierung mit mehreren Personen wäre ein deutlich größerer, nicht vertretbarer Aufwand notwendig gewesen.

Die Projektzahlen wurden im Fall der Methoden mittels Spoon ermittelt und im Fall der Klassen und Quelltextzeilen mit VscCounter¹. Im folgenden werden die einzelnen Softwareprojekte kurz vorgestellt.

7.2.1 JPacman

Bei JPacman handelt es sich um eine Java Implementierung des Spiels Pacman. JPacman wird in der Lehre eingesetzt, um Studenten Git, Junit und Mockito beizubringen. Die Software wird von Doktoranden der Uni Delft entwickelt. 22 Autoren besitzt das Projekt, wobei drei Hauptautoren erkennbar sind. In Tabelle 7.1 ist die Größe des Projektes dargestellt. Mit 46 Klassen und 201 Methoden im Hauptquelltext ist es das kleinste Projekt der Evaluation. Das Projekt besitzt 14 Testklassen, wovon drei noch Vorlagen sind, mit insgesamt 54 Testmethoden. In dem Projekt wurden sieben Methodenpaare als semantisch zusammengehörig markiert. Die Musterlösung des Projektes wurde auf Anfrage des Autors von M.Finavaro Aniche und C. Boone zur Verfügung gestellt. Diese Software wurde

¹<https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter>

Tabelle 7.2: Programmgröße von `Roller` aufgeteilt in Testteil und Hauptteil

	Klassen	Methoden	Quelltextzeilen
Haupt	537	4616	47124
Test	50	222	5326
Gesamt	587	4838	52450

Tabelle 7.3: Programmgröße von `Gitiles` aufgeteilt in Testteil und Hauptteil

	Klassen	Methoden	Quelltextzeilen
Haupt	91	666	9812
Test	34	389	4835
Gesamt	125	1055	14647

ausgewählt, um akademische Softwareprojekte in der Evaluationsmenge zu besitzen und wegen der kleinen Größe. Akademische Softwareprojekte unterscheiden sich deutlich in der Qualität und Zielsetzung von industriellen Softwareprojekten.

7.2.2 Roller

Bei `Roller` handelt es sich um ein Programm zur Verwaltung und Bereitstellung von Weblogs. `Roller`² wird von der `Apache Software Foundation` als öffentliche Software entwickelt. Das Projekt wird von 27 Autoren entwickelt, mit 2 Hauptautoren. In Tabelle 7.2 ist die Projektgröße anhand des Quelltexts dargestellt. Es existieren 537 Hauptklassen und 50 Testklassen. Mit 52450 gesamten Quelltextzeilen ist dies das größte Projekt der Evaluation. Insgesamt besitzt `Roller` 92 semantisch zusammengehörige Methodenpaare. `Roller` wurde ausgewählt um das Verfahren auf einem großen öffentlichem Projekt, welches von freiwilligen Autoren entwickelt wird, zu evaluieren.

7.2.3 Gitiles

`Gitiles` ist ein Gitbrowser, der auf `JGit`³ aufbaut, zur vereinfachten Ansicht von Gitverzeichnissen (engl. Git repository). Das Projekt wird von Google als öffentliche Software entwickelt und ist auf GitHub verfügbar⁴. 59 Autoren besitzt das Projekt, wobei drei Autoren als Hauptautoren erkennbar sind. In Tabelle 7.3 sind die Programmgrößen dargestellt. Mit 91 Hauptklassen und 666 Methoden darin ist dies das mittlere Projekt dieser Evaluation. Es sind 34 Testklassen und 389 Testmethoden vorhanden. Es sind an den Komponententests nur sechs zusammengehörige Methodenpaare erkennbar. Dies liegt an der Verwendung von privaten Methoden und dem Einsatz von Testattrappen (engl. mocks). Die Software wurde ausgewählt um industriell verwendete Software mit hoher Qualität zu evaluieren. `Gitiles` verwendet andere Projekte als Abhängigkeit. Dies ist weder in `Roller` noch `JPacman` gegeben. Deswegen wird mit dieser Software evaluiert, um das Verhalten des Verfahrens bei Abhängigkeiten zu testen.

7.3 Ergebnisse

Im folgendem Abschnitt werden die Ergebnisse auf den einzelnen Projekten vorgestellt und analysiert. Die Ergebnisse wurden anhand der in den Kriterien vorgestellten Metriken und anhand der in Abschnitt 7.2 vorgestellten Markierungsmethode erstellt. Wegen der verschiedenen Größen der Projekte werden bei `JPacman` mehr auf einzelnen Beispiele

²<https://github.com/apache/roller>

³<https://www.eclipse.org/jgit/>

⁴<https://github.com/google/gitiles>

Abbildung 7.1: Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von JPacman in Abhängigkeit des Schwellwertes

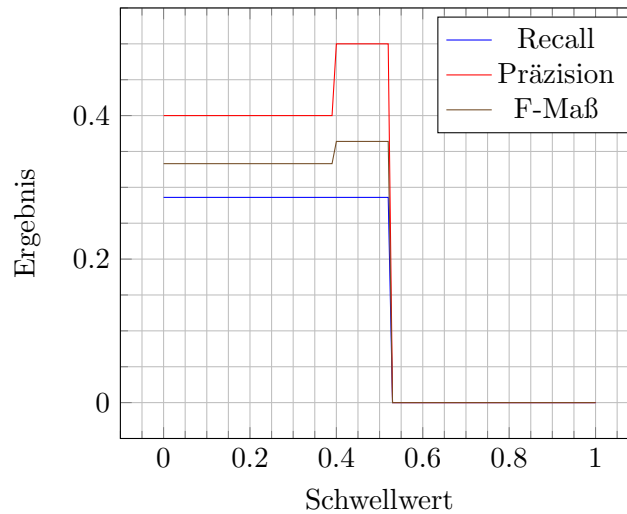


Tabelle 7.4: Anzahl an Methodenpaaren der Ergebnismenge auf JPacman mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß

Metrik	Schwellwert	Anzahl Methodenpaare		
		richtige	falsche	fehlende
Präzision	0.40	2	2	5
Ausbeute	0.39	2	3	5
F-Maß	0.40	2	2	5

eingegangen, während bei Roller das Gesamtergebnis und die Gründe dafür mehr erläutert werden. Im folgenden ist mit F-Maß immer das F_1 -Maß gemeint.

7.3.1 JPacman

Im folgenden werden die Ergebnisse des Verfahrens auf JPacman analysiert. Sowohl die Höchstwerte als auch der Verlauf werden erläutert.

In Abbildung 7.1 sind die Präzision, Ausbeute und F-Maß dargestellt in Relation zum Schwellwert. Der Schwellwert ist hierbei der minimale Konfidenzwert, welcher in Kapitel 5 erläutert wurde, den ein Methodenpaar erreichen muss.

Die Ausbeute ist zwischen den Schwellwerten 0 bis 0.52 bei durchgehend 0.29. Dies heißt es existieren viele Methodenpaare die nicht erkannt. Die Erkennung von Zugriffsfunktionen, welche in Abschnitt 5.5 erklärt ist, verursacht diesen Effekt. Ab einem Schwellwert von 0.53 wird kein Methodenpaar als zusammengehörig erkannt. Das passiert, weil kein Methodenpaar über diesen Wert kommt. Die geringe Eingabemenge scheint dies zu verursachen, was bei der spätere Betrachtung von **Roller** ersichtlich ist.

Die Präzision ist zwischen den Schwellwerten 0 und 0.39 bei 0.40. Danach steigt die Präzision auf 0.50 bei einem Schwellwert von 0.40. Ab 0.53 sinkt die Präzision auf 0. Dies liegt daran, dass kein Methodenpaar mehr über dem Schwellwert liegt.

Ab einem Schwellwert von 0.40 wird das höchste F-Maß von 0.36 erreicht. Bei diesem Wert ist die Präzision bei 0.50 und die Ausbeute bei 0.29. Ab 0.53 ist das F-Maß 0.

Nun werden die konkreten Zahl betrachtet. In Tabelle 7.4 sind die richtigen, falschen und fehlenden Verbindungen dargestellt. Es sind insgesamt sieben semantisch zusammengehörige Methodenpaare vorhanden. Bei höchster Präzision werden nur zwei von sieben markiert.

Dabei existieren auch zwei falsche Paare. In Quelltextausschnitt 7.1 ist eine der Fehlerquellen dargestellt. Die Methode `Navigation::findNearest` wird von dem Verfahren mit `parser::parseMap` als zusammengehörig erkannt. Das Zerteilen der Eingabe, die das Spielfeld erzeugt, ist aber nicht zusammengehörig zu einer Positionsabfrage mit `findNearest`. Das Verfahren erkennt dieses Paar fälschlicherweise als zusammengehörig und erreicht den höchsten Konfidenzwert von allen Paaren. Eine Lösung dafür kann das Hinzufügen von weiteren Merkmalen bei der Analyse wie z.B. die Dokumentation oder Bezeichner. Negative Konfidenzwertänderungen könnten auch verwendet werden. Ein Teilverfahren könnte erkennen, dass `parseMap` nur eine Zwischenfunktion ist um ein Spielfeld zu erstellen und die Konfidenzwerte aller Paare mit der Methode verringern.

Quelltextausschnitt 7.1: Testfall der zu Fehlern innerhalb von JPacman führt

```
/**
 * Verifies that there is no such location if there is no nearest object
 */
public void testNoNearestUnit() {
    Board b = parser.parseMap(Lists.newArrayList("□")).getBoard();
    Square s1 = b.squareAt(0, 0);
    Unit unit = Navigation.findNearest(Pellet.class, s1);
    assertThat(unit).isNull();
}
```

Als Fazit für JPacman lassen sich zwei Schlüsse ziehen: Es gibt semantisch zusammengehörige Methoden und diese sind erkennbar, es ist aber mit 0.50 eine relativ niedrige Präzision vorhanden. Vergleicht man die Ergebnisse mit anderen Rückverfolgungsverfahren in dem Bereich, die z.B. nur die getestete Methode erkennen, relativiert dies die niedrige Präzision, weil diese Verfahren auch in der Nähe von 0.60 sind [CKV19]. Die Erkennung einer getesteten Methode ist ein deutlich leichteres Problem, als die Erkennung von semantisch zusammengehörigen Methoden. Diese Arbeit besitzt auch kein Vorwissen von anderen Arbeiten, die sich mit dem selben Problem befassen haben. Insgesamt hat das Verfahren auf JPacman keinen ausreichenden Wert für die Präzision erreicht. Sowohl eine deutliche Erhöhung der Ausbeute als auch eine Erhöhung der Präzision ist wünschenswert. Die Ausbeute kann mit einer besseren Implementierung der Zerteilung des Java-Quelltextes gesteigert werden, die Präzision über weitere Teilverfahren. Weitere Teilverfahren sind wegen dem drei Phasen Aufbau, relativ einfach hinzuzufügen. Mit mehr Informationen kann das Verfahren bessere Ergebnisse, besonders bei der Präzision, erzielen.

7.3.2 Roller

Im folgenden werden die Ergebnisse des Verfahrens auf `Roller` vorgestellt, analysiert und bewertet. Hierzu werden erst die Werte für Präzision und Ausbeute betrachtet, dann die quantitativen Werte und zum Schluss die einzelnen Teilverfahren.

In Abbildung 7.2 ist Präzision, Ausbeute und F-Maß des Verfahrens in Abhängigkeit des Schwellwertes dargestellt. Auf der X-Achse ist der Schwellwert dargestellt.

Die Präzision des Verfahrens ist zwischen den Schwellwerten 0 und 0.35 konstant bei 0.19. Dies bedeutet, dass kein erkanntes Methodenpaar unter 0.35 als Konfidenzwert hat. Dann steigt die Präzision treppenartig bis zu dem Schwellwert 0.75. Dort erreicht die Präzision mit 0.74 und 17 richtigen zu sechs falschen Methodenpaaren den Höchstwert. Danach sinkt diese auf 0.58 bei einem Konfidenzwert von 0.80 und steigt dann wieder leicht auf eine Präzision von 0.58. Über einen Konfidenzwert von 0.87 kommt kein erkanntes Methodenpaar.

Abbildung 7.2: Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von `Roller` in Abhängigkeit des Schwellwertes

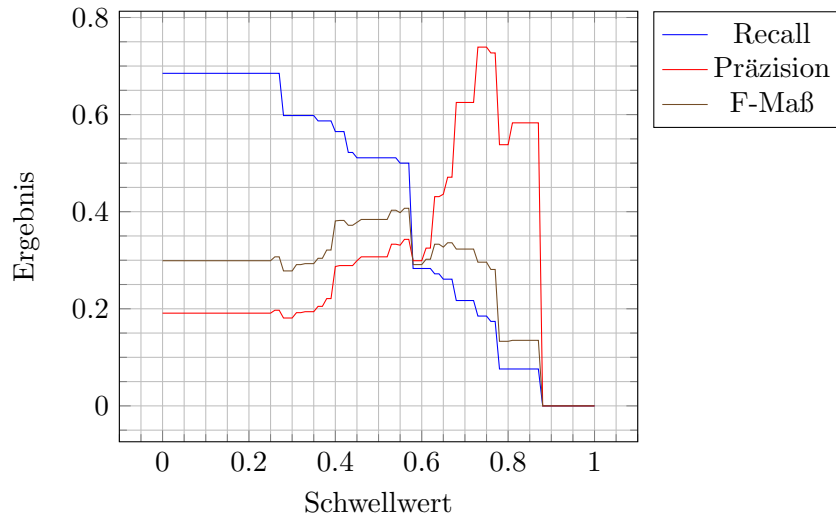


Tabelle 7.5: Anzahl an Methodenpaaren mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß

Metrik	Schwellwert	Anzahl Methodenpaare		
		richtige	falsche	fehlende
Präzision	0.74	17	6	75
Ausbeute	0.27	63	256	29
F-Maß	0.57	46	88	46

Die Ausbeute des Verfahrens ist zwischen den Schwellwerten 0 bis 0.27 konstant bei 0.685. Eine Ausbeute von 1.00 wird nicht erreicht, weil Zugriffsfunktionen gefiltert werden, wie in Abschnitt 5.5 erklärt. Dies filtert auch Funktionen, die keine Zugriffsfunktionen sind. Von 92 semantisch zusammengehörigen Methodenpaaren werden 63 in diesem Bereich erkannt. Ab einem Schwellwert von 0.27 sinkt die Ausbeute treppenabartig bis zu einem Minimum von 0 bei einem Konfidenzwert von 0.88. Bei dem Konfidenzwert 0.58 sinkt die Ausbeute von 0.50 zu 0.34. Mit steigendem Schwellwert sinkt die Ausbeute. Bei der höchsten Präzision ist eine Ausbeute von ungefähr 0.20 noch vorhanden.

Der Verlauf des F-Maß ist bis zum Schwellwert 0.57 ähnlichen zum Verlauf der Präzision. Im Bereich 0 bis 0.27 liegt es bei 0.31. Der Höchstwert des F-Maß liegt bei 0.31 bei einem Schwellwert von 0.56. Bei dem Schwellwert 0.58 scheidet sich die Kurven der Präzision, Ausbeute und F-Maß.

In Tabelle 7.5 sind die quantitativen Werte bei höchster Präzision, Ausbeute und F-Maß dargestellt. An den Werten ist ersichtlich, dass bei einer hohen Präzision nur noch wenige Methodenpaare markiert werden und bei einer hohen Ausbeute umgekehrt.

In Tabelle 7.6 sind Punkte des in Abbildung 7.2 dargestellten Verlaufs mit den jeweiligen Werte für Präzision, Ausbeute und F-Maß dargestellt. Bei steigendem Schwellwert steigt die Präzision, außer bei den Schwellwerten 0.75 und 0.85. Dies zeigt, dass ein höherer Schwellwert die Präzision nicht durchgehend verbessert. Bei 0.75 wird die maximale Präzision von 0.74 erreicht,

Nachdem die Ergebnisse für das Gesamtverfahren mit allen Teilverfahren aktiv vorgestellt wurde, werden nun die einzelnen Teilverfahren betrachtet. Wie in Abschnitt 5.2 erläutert besitzt eine Komponententestklasse zwei Unterschiede zu einer Hauptklasse. Die Über-

Tabelle 7.6: Punkte aus Abbildung 7.2 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß

Schwellwert	Präzision	Ausbeute	F-Maß
0.25	0.19	0.69	0.30
0.40	0.29	0.57	0.38
0.58	0.30	0.28	0.29
0.65	0.44	0.26	0.33
0.75	0.74	0.19	0.30
0.85	0.58	0.08	0.14

prüfungen in den Testmethoden und die Aufbau/Abbaumethoden. Für die Auswertung existieren mehrere Teilverfahren, in dem in Kapitel 6 vorgestellten Verfahren. Um die Auswirkungen der einzelnen Teilverfahren zu vergleichen, werden nun aufeinander aufbauend verschiedene Kombinationen erstellt. Diese Kombinationen von Teilverfahren werden bei drei konstanten Konfidenzwerten verglichen. Diese Werte sind 0.25, 0.40 und 0.60. Mit 0.25 wird ein niedriger Wert verwendet um die Veränderungen in der Ausbeute zu analysieren. In Abbildung 7.2 ist ersichtlich, dass bis 0.28 wenig Veränderungen im Gesamtverfahren sind. Dies könnte mit verschiedenen Kombinationen anders sein. Der Wert von 0.40 wird verwendet, weil bei diesem Wert das F-Maß einen der höchsten Werte hat. Die Präzision ist bei dem Schwellwert 0.75 am höchsten. Dieser Wert wird aber nicht verwendet sondern ein leicht geringerer, weil durch fehlende Teilverfahren die Konfidenzwerte geringer werden. An dem Schwellwert 0.60 wird deswegen verglichen.

In Tabelle 7.7 sind Teilverfahren und ihre Ergebnisse dargestellt. Die Komponenten der Teilverfahren sind in Kapitel 5 vorgestellt worden. Die Tabelle besteht aus zwei Teilen, der obere für die Teilverfahren und der untere für die verschiedenen Ähnlichkeitsalgorithmen. In der ersten Zeile sind Werte der drei verschiedenen Schwellwert 0.25, 0.40 und 0.60 abgebildet. Für jeden der Schwellwert sind in drei Spalten die Werte für Präzision, Ausbeute und F-Maß dargestellt. In der Spalte Teilverfahren ist genannt welche Teilverfahren für die jeweiligen Werte genutzt werden. Als erstes wird der obere Teil der Tabelle ausgewertet und analysiert, dann der untere Teil. In der Beschreibung wird spaltenweise vorgegangen. In der ersten Spalten sind die Werte für Präzision, Ausbeute und F-Maß für den Schwellwert 0.25 dargestellt. Das Teilverfahren bestehend aus überprüften Methode, Klasse und Objekt erreicht eine Präzision von 0.37 bei einer Ausbeute von 0.21. Durch das Hinzufügen der Aufbau/Abbaumethode verringert sich die Präzision auf 0.20 aber die Ausbeute steigt auf 0.53. Dies liegt an der größeren Menge an verteilten Konfidenzwerten und den damit steigenden Werten die jedes Methodenpaar erreichen kann. Es findet eine Verschiebung mit dem Schwellwert statt. Das Hinzufügen des ähnlichen Klassennamens verringert die Präzision leicht, erhöht aber die Ausbeute. Die Analyse des Methodenname und Paketbezeichners erhöhen die Präzision und Ausbeute. Dies zeigt, dass beide Teilverfahren für das Problem geeignet sind. Dieser Schwellwert wurde ausgewählt um die Auswirkung auf die Ausbeute der Teilverfahren zu erkennen. Es ist ersichtlich, dass jedes Teilverfahren für eine Steigerung der Ausbeute sorgt. Dies zeigt, dass für eine maximale Ausbeute die vollständig Kombinationen geeignet ist.

Die zweite Spalte stellt die Werte am Schwellwert 0.40 dar. Dieser Messpunkt wurde ausgewählt, um die Teilverfahren hinsichtlich des F-Maß zu vergleichen. Das Teilverfahren bestehend aus überprüften Methode, Klasse und Objekt erreicht ein F-Maß von 0, weil kein Methodenpaar den Schwellwert erreicht hat. Dies liegt an der zu geringen vergebenen Konfidenzwerte. Jedes Teilverfahren bis auf die Analyse der Klassennamen erhöht das F-Maß. Der höchste Wert ist hierbei 0.38 bei dem vollständigen Verfahren.

Tabelle 7.7: Teilverfahren und verschiedene Ähnlichkeitsalgorithmen angewandt auf Roller mit ihrer Präzision, Ausbeute und F-Maß bei drei verschiedenen Schwellwerten.

Teilverfahren	Schwellwert 0.25			Schwellwert 0.40			Schwellwert 0.60		
	Präzision	Ausbeute	F-Maß	Präzision	Ausbeute	F-Maß	Präzision	Ausbeute	F-Maß
überprüfte Methode/Klasse/Objekt	0.37	0.21	0.27	0.00	0.00	0.00	0.00	0.00	0.00
überprüfte Methode/Klasse/Objekt + Auf/Abbau	0.20	0.53	0.29	0.37	0.21	0.27	0.00	0.00	0.00
überprüfte Methode/Klasse/Objekt + Auf/Abbau + Klassenname	0.17	0.55	0.26	0.25	0.29	0.27	0.45	0.10	0.16
überprüfte Methode/Klasse/Objekt + Auf/Abbau + Klassen/Methodennamen	0.18	0.60	0.28	0.29	0.52	0.37	0.43	0.27	0.33
überprüfte Methode/Klasse/Objekt + Auf/Abbau + Klassen/Paket/Methodennamen	0.19	0.69	0.30	0.29	0.57	0.38	0.30	0.28	0.29
Ähnlichkeitsalgorithmus									
fastText 1 Epoche	0.19	0.69	0.30	0.33	0.52	0.40	0.34	0.17	0.23
fastText 5 Epochen	0.19	0.69	0.30	0.33	0.52	0.40	0.34	0.17	0.23
fastText 25 Epochen	0.19	0.69	0.30	0.32	0.51	0.40	0.34	0.19	0.24
Kosinus	0.19	0.69	0.30	0.21	0.49	0.29	0.28	0.15	0.20
FuzzyScore	0.19	0.69	0.30	0.21	0.55	0.30	0.33	0.36	0.34
JaroWinkler	0.19	0.69	0.30	0.20	0.67	0.31	0.41	0.50	0.45
Längste Gleiche Sequenz(LGS)	0.19	0.69	0.30	0.20	0.53	0.29	0.52	0.37	0.43
Gleichheit	0.19	0.69	0.30	0.37	0.42	0.40	0.35	0.10	0.15
Jaro+LGS+fastText	0.21	0.69	0.33	0.22	0.67	0.33	0.41	0.53	0.46

Die Analyse der Klassennamen verändert das F-Maß zwar nicht, dies liegt an der höheren Ausbeute und niedrigeren Präzision. Diese Veränderung entsteht wegen der Verschiebung durch höhere mögliche Konfidenzwerte.

Die dritte Spalte stellt die Werte am Schwellwert 0.60 dar. An diesem Messpunkt wird die Präzision der Teilverfahren verglichen. Die Verwendung der überprüften Methode, Klasse und Objekte mit und ohne der Aufbau/Abbaumethode erreichen diesen Schwellwert nicht. Dies liegt an dem zu geringen maximalen erreichbaren Konfidenzwert. Durch Hinzufügen der Analyse des Klassennamens wird eine Präzision von 0.45 erreicht bei einer Ausbeute von 0.10. Die ist der höchste Wert bei dem Schwellwert 0.60. Durch die zusätzliche Analyse des Methodennamens sinkt die Präzision leicht auf 0.43 aber die Präzision steigt auf 0.27. Die leichte Verringerung der Präzision entsteht durch mehr Teilverfahren die Konfidenzwerte erhöhen. Die Ausbeute steigt signifikant. Dies zeigt, dass die Information des Methodennamens bei Komponententests wichtig ist. Das Gesamtverfahren besitzt eine deutlich geringe Präzision von 0.30 bei einer Ausbeute von 0.27. Dies liegt daran, dass die Analyse des Paketbezeichners ungenauer ist. Wie in Tabelle 7.6 ersichtlich, kann das Gesamtverfahren deutlich höhere Präzisionswerte erreichen.

Nun wird der untere Teil der Tabelle 7.7 erläutert. Das Verfahren verwendet für die Ähnlichkeit zweier Zeichenketten verschiedene Algorithmen. Diese sind in Abschnitt 5.5 und Abschnitt 5.5 genannt und erläutert. Nun sollen diese Algorithmen verglichen werden hinsichtlich Präzision, Ausbeute und F1-Wert an drei Schwellwerten. Bei dem Schwellwert 0.25 erreichen alle Verfahren die selben Werte für Präzision, Ausbeute und F-Maß. Diese sind 0.19 für Präzision, 0.69 für Ausbeute und 0.30 für F-Maß. Diese liegt daran, dass niedrige Schwellwert ohne Benennungsregeln bereits erreicht werden. Deswegen haben die verschiedenen Algorithmen keine Auswirkung auf das Ergebnis bei diesem Schwellwert. Bei dem Schwellwert 0.40 ändert sich dieses Verhalten. `fastText` erreicht für jede Epochenanzahlen bei diesem Schwellwert fast die selben Werte für Präzision, Ausbeute und F-Maß. Die einfache Gleichheit erreicht mit 0.37 die höchste Präzision bei einer Ausbeute von 0.40. `FuzzyScore` erreicht mit 0.67 eine Ausbeute von 0.67. Dies ist bei eine rErhöhung des Schwellwertes um 0.15 nur eine Verringerung der Ausbeute um 0.02. Das höchste F-Maß erreichen alle `fastText` Einstellungen und die einfache Gleichheit mit 0.40. Nun wird die Spalte am Schwellwert 0.60 betrachtet. An diesem Schwellwert sind 25 `fastText` Epochen den anderen Einstellungen um 0.02 Ausbeute überlegen. Die höchsten Präzisionswerte erreichen längsten gleiche Sequenz mit 0.52 und `JaroWinkler` mit 0.41. Mit `JaroWinkler` wird zusätzlich der höchste Wert für die Ausbeute und F-Maß mit 0.50 und 0.45 erreicht.

Zusammenfassend ist ersichtlich, dass Kosinus und `FuzzyScore` in keinem Teilbereich einen Höchstwert erreichen. Dies zeigt, dass diese Algorithmen auf der Datenmenge nicht geeignet sind. `fastText` profitiert nur sehr gering von höheren Epochenanzahlen und ist hinsichtlich maximaler Präzision nicht gut. Die geringen Änderungen in der steigenden Epochenanzahlen zeigt, dass `fastText` nach einer Epoche schon die Worteinbettung gelernt hat. `JaroWinkler` und `Längste Gleiche Sequenz` sind bei dem Schwellwert 0.40 und 0.60 hervorstechend. Für ein Verfahren mit maximaler Präzision sind diese also sehr geeignet. Für ein möglichst allgemeines Verfahren ist die Kombinationen von allen Algorithmen aber besser als die einzelnen. Kein einzelner Algorithmus ist in allen drei Schwellwerten besser als die Vereinigung von allen Algorithmen.

Nachdem die einzelnen Teilverfahren und verschiedene Ähnlichkeitsteilverfahren evaluiert wurden, wird nun ein konkretes Beispiel für eine fehlerhafte Erkennung gezeigt. In Quelltextausschnitt 7.2 ist der Test dargestellt⁵. Der Test überprüft die Erstellung, Veränderung und Löschung eines Kommentares. Dies wird im Englischen mit `CRUD` abgekürzt.

⁵Für die Lesbarkeit wurde der Updateteil des Testfalles gekürzt

Quelltextausschnitt 7.2: Gekürzter Testfall aus Roller bei dem das Verfahren falsche Ergebnisse produziert.

```
/**
 * Test basic persistence operations ... Create, Update, Delete
 */
public void testCommentCRUD() {
    WeblogEntryManager mgr = WebloggerFactory.getWeblogger().
        getWeblogEntryManager();
    WeblogEntryComment comment = new WeblogEntryComment();
    comment.setName("test");
    comment.setEmail("test");
    comment.setUrl("test");
    comment.setRemoteHost("foofoo");
    comment.setContent("this_is_a_test_comment");
    // create a comment
    mgr.saveComment(comment);
    TestUtils.endSession(true);
    // make sure comment was created
    comment = mgr.getComment(id);
    assertNotNull(comment);
    assertEquals("this_is_a_test_comment", comment.getContent());
    // delete a comment
    mgr.removeComment(comment);
    TestUtils.endSession(true);
}
```

Der Test erstellt erst einen Kommentar und setzt Eigenschaften dessen, wie z.B. der Name oder den Inhalt. Dann wird der Kommentar mittels der Methode `WeblogEntryManager::saveComment` gespeichert. Das Setzen der Eigenschaften bildet zusammen mit dem Speichern jeweils ein semantisch zusammengehöriges Methodenpaar. Sowohl das Speichern als auch das Setzen von Eigenschaften ist in diesem Fall nur zusammengehörig sinnvoll. Danach wird überprüft, ob der Kommentar erstellt wurde. Dies wird mittels einer Zugriffsfunktion durchgeführt und ist nicht zusammenhängend mit anderen Methoden. Bis zu diesem Punkt erkennt das Verfahren die Methodenpaare. Am Ende des Tests wird der Kommentar gelöscht. Für das Verfahren sind die Lösch- und Speicherfunktion nicht unterscheidbar hinsichtlich ihre umgesetzten Funktionalität. Es wird also das Setzen des Inhaltes mit dem Löschen des Kommentares als zusammengehörig erkannt. Dies ist aber offensichtlich falsch und sorgt für falsche Methodenpaare. Die Analyse der Dokumentation würde in diesem Fall keine Möglichkeit der Lösung bieten. Eine Analyse von Clustern könnte diesen Fall lösen. Es wird der Komponententest zu einem vollverbundenen Graph transformiert. Die verwendeten Methode werden Knoten. Die Kanten besitzen ein Gewicht, was dem Konfidenzwert des Methodenpaares, durch die beiden Knoten beschrieben, entspricht. Die Änderungsfunktionen wie z.B. `setName` besitzen nun Kanten zu der Löschen und Erstellen Methoden, mit genug Gewicht für die Erkennung. Nun werden beide Verbindungen nur akzeptiert wenn die Löschen- und Erstellenfunktionen auch miteinander verbunden sind und als semantisch zusammengehörig erachtet werden. Dieses Verfahren könnte nun dieses Problem lösen, indem z.B. die frühere Methode als wahrscheinlicher gilt. Eine anderes Entscheidungsverfahren als die frühere Methode ist natürlich auch möglich.

7.3.3 Gitiles

Nun werden die Ergebnisse des Verfahrens auf `Gitiles` vorgestellt. In Abbildung 7.3 sind Präzision, Ausbeute und F-Maß dargestellt. Die Abbildung besitzt die selben Konventionen

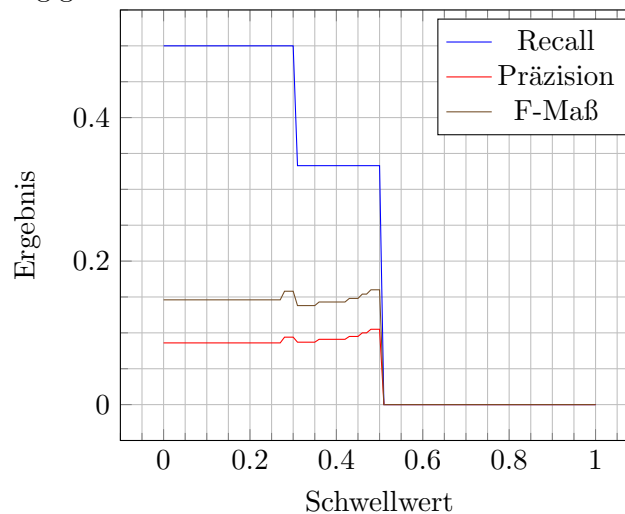
Abbildung 7.3: Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von `Gitiles` in Abhängigkeit des Schwellwertes

Tabelle 7.8: Anzahl an Methodenpaaren mit ihrer Klassifikation bei höchster Präzision, Ausbeute und F-Maß

Metrik	Schwellwert	Anzahl Methodenpaare		
		richtige	falsche	fehlende
Präzision	0.48	2	17	4
Ausbeute	0.00	3	32	3
F-Maß	0.30	3	29	3

wie die vorherigen. Die Ausbeute startet bei 0.50 und sinkt dann ab einem Schwellwert von 0.31 stark ab auf erst 0.33 und dann auf 0 bei dem Schwellwert 0.51. Die Präzision ist zwischen dem Schwellwert 0 und 0.42 bei 0.09 und steigt dann auf einen Maximalwert 0.11. In Tabelle 7.9 sind die Werte für Präzision, Ausbeute und F-Maß für verschiedene Werte nochmal dargestellt. Die Werte für Präzision und Ausbeute sind für den Bereich der Rückverfolgung sehr niedrig. Im Vergleich zu den Ergebnis von `Roller` und `JPacman` ist ersichtlich, dass das Verfahren auf `Gitiles` nicht anwendbar ist. Bei der Betrachtung der absoluten Werten in Tabelle 7.8 ist erkennbar, dass sehr viele falsche Methodenpaare markiert werden. Dies sind bei der höchsten Präzision noch 17 Stück.

Nun ist die Ursache, warum das Verfahren unanwendbar auf `Gitiles` ist, wichtig. Das erste Problem ist der starke Einsatz des Erbauer-musters (engl. Builder-Pattern). Dieses Muster wird häufig verwendet, weil zur Erstellung dafür eine Annotation⁶ entwickelt wur-

Tabelle 7.9: Punkte aus Abbildung 7.3 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß

Schwellwert	Präzision	Ausbeute	F-Maß
0.0	0.08	0.50	0.15
0.20	0.09	0.50	0.16
0.38	0.09	0.33	0.14
0.45	0.10	0.33	0.15
0.50	0.11	0.33	0.16

⁶<https://github.com/google/CallBuilder>

de und es in Quelltexttrichtlinien [Blo17] empfohlen wird. Das Problem daran ist, dass eine innere Klasse Werte setzt. Das Verfahren erkennt also die Funktionen des Erstellers mit den anderen Funktionen. Diese erkannten Methodenpaare sind aber auf Gitiles häufig eine Zugriffsfunktion mit einer Änderungsfunktion. Das Verfahren erkennt die Zusammengehörigkeit der äußeren Klasse mit anderen Klassen schlechter bzw. gar nicht mehr. Werden diese Methodenpaare entfernt, sind von den 20 falschen Methodenpaaren bei der höchsten Ausbeute noch sechs übrig. Das Verfahren kann diese Ersteller nicht richtig verwenden bei der Informationsgewinnung. Es müsste also aus diesen Erstellern die Information gewinnen, welche Methoden darauf arbeiten und könnte damit möglicherweise mehr Informationen sammeln. Diese Erbauer wären erkennbar und mit einer gesonderten Bearbeitung lösbar. Problem hierbei wird sein, dass die Erstellung der Erzeuger über Annotation oder Präprozessor aufwendig in der Erkennung ist.

Ein weiteres Problem ist die starker Nutzung von Testhelfern. `Gitiles` nutzt ein Gitverzeichnis von Junit statt dem eigenen. Somit sind alle Methoden, die mit diesem Testhelfer arbeiten, nicht wertvoll für die Erkennung von semantischer Zusammengehörigkeit. In Abschnitt 7.2.3 ist erklärt, warum dieses Softwareprojekt trotzdem zur Evaluation verwendet wurde. Die Testhelfer sind Klassen, die ähnlich zur Hauptklasse implementiert wurden, aber leichter testbar sind. Aus diesen Testhelfern wäre es möglich die ursprüngliche Methoden als Information zu gewinnen, z.B. über Methodenname und Signatur. Dadurch könnten aber neue Fehler entstehen durch die falsche Rückführung. Eine `commit` Methode auf dem Testhelfer, wäre z.B. in den Hauptklassen such und nachverfolgbar. Die leichteste Variante wäre dies über Signatur und Methodennamen zu lösen. Diese Rückverfolgung kann das Verfahren aufgrund der Komplexität nicht.

Als letztes würde bei der Betrachtung des Quelltextes auffallen, dass dieser moderner ist als `JPacman` oder `Roller`. So fehlen bei `Roller` z.B. teilweise generische Typen. Dies ist ein Zeichen für Quelltext der vor der Java Version fünf geschrieben wurde. Die Java Version zum Zeitpunkt der Veröffentlichung dieser Arbeit ist 13. `Gitiles` hat sein ersten Beitrag(engl. `commit`) 2012. Zu diesem Zeitpunkt war die Java Version zwar sieben, aber der Quelltext wurde im Lauf der Zeit verbessert. Modernerer Quelltext ist komplexer zu Zerteilen. Dies ist eine der Ursache, warum die Ergebnisse so niedrig sind. Der Einsatz von inneren Klassen und Lambda Ausdrücken war in den beiden anderen Projekten nicht vertreten. Mit einer besseren Implementierung der Zerteilung des Java-Quelltextes könnte das Ergebnis besser werden.

Wegen dem Testhelfer, den Erstellern und eine starken Kohäsion zu `JGit` wurden auch nur verhältnismäßig wenige semantisch zusammengehörige Methoden erkannt. Aufgrund der genannten Probleme sind die Ergebnisse zu erklären. Diese Ergebnisse zeigen, dass auf verhältnismäßig komplexer Software dieses Verfahren momentan nicht einsetzbar ist. Mit den genannten Lösungen könnte dies besser werden, wobei dies nur Vermutungen des Autors sind.

7.3.4 Gesamtergebnis

Im folgenden Abschnitt wird ein Gesamtergebnis gebildet. Hierfür wurden die Ergebnisse der drei Verfahren addiert. Anhand dessen wird nun beurteilt wie gut das Gesamtverfahren funktioniert hat.

In Abbildung 7.4 sind die addierten Ergebnisse des Verfahrens dargestellt. Die Präzision und Ausbeute wurden auf den addierten Markierungen und nicht mittels dem Durchschnitt berechnet. Die Präzision ist zwischen den Schwellwerten 0 bis 0.27 bei ungefähr 0.20. Dieses Verhalten war bei allen Projekten zu beobachten. Die Ausbeute liegt zwischen 0 und 0.27 bei ungefähr 0.65. Diese hohe Ausbeute war auch bei allen Projekten am Anfangen vorhanden. Bei niedrigen Schwellwerten sind also hohe Ausbeuten möglich. Bei

Abbildung 7.4: Präzision, Ausbeute und F-Maß des Verfahrens innerhalb von allen drei Projekten in Abhängigkeit des Schwellwertes

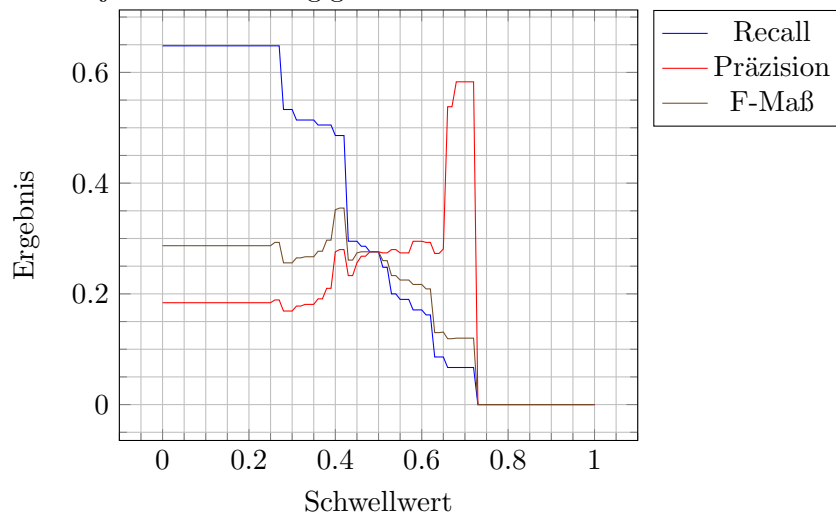


Tabelle 7.10: Punkte aus Abbildung 7.4 mit ihren exakten Werten für Präzision, Ausbeute und F-Maß

Schwellwert	Präzision	Ausbeute	F-Maß
0.00	0.18	0.65	0.29
0.10	0.18	0.65	0.29
0.20	0.18	0.65	0.29
0.30	0.17	0.53	0.26
0.40	0.28	0.49	0.35
0.50	0.28	0.28	0.28
0.60	0.30	0.17	0.22
0.70	0.58	0.07	0.12

dem Schwellwert 0.50 schneiden sich die Graphen von Präzision, Ausbeute und F-Maß. Dieser Schnittpunkt war bei *Roller* höher und bei *Gitiles* und *JPacman* vom Verhalten anders. Bei *JPacman* und *Gitiles* war der Schnittpunkt auch der Punkt, an dem die Werte stark abfielen. Die hohe Präzision bei dem Schwellwert von 0.70 von ungefähr 0.65 wird durch *Roller* entstanden sein. Auf diesem Projekt hatte das Verfahren sehr gute Ergebnisse.

In Tabelle 7.10 sind Werte für Präzision, Ausbeute und F-Maß für einzelne Werte angegeben. Es ist ersichtlich das ein Schwellwert in der Nähe von 0.40 das höchste F-Maß erzeugt. Eine Schwellwert zwischen 0.60 und 0.70 erzielt eine hohe Präzision.

7.4 Fazit

Nachdem die drei Softwareprojekte einzeln evaluiert wurden wird nun das Gesamtergebnis beurteilt, hinsichtlich der Frage ob dieses Verfahren semantisch zusammengehörige Methoden ableiten kann. Hierbei muss das Ergebnis in Relation zu dem Projekt *INDIRECT* betrachtet werden. Das Projekt benutzt verschiedene Verfahren, welche zusammen ein Gesamtergebnis erzeugen. Deswegen ist eine höhere Präzision wichtiger zu bewerten als eine hohe Ausbeute, weil falsche Ergebnisse eines Verfahrens die anderen Verfahren schlechter machen könnten. In Abschnitt 5.1 sind die vier Ziele für diese Arbeit genannt worden. Die Lösungen der Ziele, Analyse der vorhandenen Informationen und die Entwicklung von Teil-

verfahren zur Informationsgewinnung, wurde in Kapitel 5 und Kapitel 6 bereits vorgestellt. Deswegen wird das dritte Ziel als erstes diskutiert.

Die Erkennung von semantisch zusammengehörigen Methoden ist mit diesem Verfahren möglich. Es wurden auf **JPacman** und **Roller** semantisch zusammengehörige Methoden erkannt. Also ist die Grundfrage: „Ist die Erkennung von semantisch zusammengehörigen Methoden anhand von Komponententests möglich?“ aus Sicht des Autors mit Ja zu beantworten. Auf **Roller** wurden Höchstwerte von 0.74 Präzision mit 0.19 Ausbeute erreicht. Auf **Gitiles** ist aber deutlich erkennbar, dass das Verfahren Probleme auf komplexer Software besitzt. Es werden zwar Paare erkannt, aber die Präzision und Ausbeute sind sehr gering. Zusätzlich muss die geringe Datenmenge und das Markieren der Datenmenge durch den Autor beachtet werden. In Abschnitt 7.2 ist dies erklärt. Es wurde also ein Verfahren entwickelt, was die Zielsetzung erfüllt

Als nächstes wird diskutiert, wie gut das Verfahren ist. Auf **Roller** wurden für den Bereich vergleichbare Ergebnisse erzielt. Csuvik et al. [CKV19] erreichte eine Präzision von 0.60 in der Rückverfolgung von Test zu getesteter Klasse. In dieser Arbeit wurden auch nur Klassen berücksichtigt die Benennungsregeln eingehalten haben, was die Rückverfolgung vereinfacht. Es wurden also auf einer guten Datenmenge 0.60 erzielt. Csuvik et al. betrachtete die Rückverfolgung auf Klassenlevel. Arshia et al. [AMH16] erzielte bei der Erkennung der getesteten Klasse eine Ausbeute und Präzision von über 0.90. Beide Arbeiten lösen ein leichteres Problem. Deswegen ist im Vergleich dazu, das Ergebnis auf **Roller** annehmbar. Bei der Betrachtung der Ergebnisse von **Gitiles** ist erkenntlich, dass dieses Verfahren nicht allgemein tragfähig ist. Mit den genannten Verbesserungen für das Verfahren und der Nutzung von mehr Informationen könnte das Verfahren auf allen Projekten Ergebnisse wie auf **Roller** erzielen.

8 Zusammenfassung und Ausblick

Semantisch zusammengehörige Methoden helfen beim Erlangen des Verständnisses von Quelltext. Dieses Verständnis hilft bei der Rückverfolgung von Methoden zu Anforderungen. Durch diese Rückverfolgung ist es möglich, die Implementierung aller Anforderungen zu garantieren. Zur Erkennung von semantisch zusammengehörigen Methodenpaaren wurden Komponententests verwendet. Daraufhin wurde Java-Quelltext analysiert hinsichtlich der Informationsquellen und ihrer Art. Die einzelnen Informationsquellen wurden dann detailliert hinsichtlich der Struktur und möglicher Informationen analysiert. Es wurde festgestellt, dass keine einzelnen Informationen direkt semantisch zusammengehörige Methodenpaare als Rückschluss zulässt. Daher wurde aus den Informationen ein heuristisches Verfahren entworfen. Die einzelnen Informationen, wie z.B. die getestete Methode, werden zusammengeführt und erstellen gemeinsam einen Konfidenzwert für jedes Methodenpaar. In der Implementierung wurden die Umsetzung des Entwurfs vorgestellt und dargestellt. Hierbei wurde die Aufteilung in drei aufeinander aufbauende Phasen vorgestellt, dem Zerteilen und Transformieren, der Informationsgewinnung mittels Teilverfahren und der Zusammenführung der Teilverfahren. Die Zerteilung wurde mit einer Tiefensuche implementiert und ein Besucher transformiert den Quelltext in eine interne Darstellung. Danach wird mit einer Steuerungseinheit die Anwendung der verschiedenen Teilverfahren vorgestellt. Nach den Teilverfahren findet eine Ergebniszusammenführung statt. In der Evaluation wurde auf `JPacman`, `Roller` und `Gitiles` das Verfahren ausgewertet. Hierfür wurden Präzision, Ausbeute und F-Maß der Ergebnisse berechnet. Es wurden die verschiedenen Ähnlichkeitsalgorithmen verglichen und gezeigt, dass eine Kombination besser ist als ein einzelner Algorithmus. Bei `Roller` und `JPacman` wurden Präzisionswerte von 0.74 und 0.50 erreicht, bei Ausbeuten von 0.19 und 0.29. Auf `Gitiles` wurde gezeigt, dass bei komplexer Software das Verfahren noch besser werden muss. Es wurden nur Präzisionswerte von 0.11 erreicht bei einer Ausbeute von 0.33 erreicht. Am Ende wurde das Verfahren mit ähnlichen Verfahren aus anderen Arbeiten verglichen. Hierbei wurden vergleichbare Werte erreicht, im Hinblick dass die Arbeiten leichtere Probleme lösten.

Es wäre es möglichen den Ansatz dieser Arbeit zu verbessern. Die gesamte Dokumentation ist eine Informationsquelle die nicht verwendet wurde. Dieser Bereich wäre mit der Verarbeitung von natürlicher Sprache lösbar. Die Dokumentation kann Hinweise auf die getestete Methode oder Klasse liefern. Dies kann die Erkennung von semantisch zusammengehörigen Methoden verbessern.

Als anderer Ansatz wäre es möglich dieses Problem mittels maschinellen Lernen zu lösen. Mittels einer binären Klassifikation über z.B. Entscheidungswälder(engl. random forest).

Hierbei ist es einfacher weitere Merkmale hinzuzufügen wie z.B. die Dokumentation oder Bezeichnen. Über die Verwendung von mehr Merkmale als in diesem Verfahren könnte es bessere Ergebnisse erreichen. Problem kann die notwendige Datengrundlage sein, wobei diese bei tiefen Neuronalen Netzen ein größeres Problem wäre.

Literaturverzeichnis

- [AMH16] ARSHIA, Amir H. ; MOSTAFA, Seyed ; HADI, Mohammad: Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects. 7 (2016). <http://dx.doi.org/10.14569/IJACSA.2016.071043>. – DOI 10.14569/IJACSA.2016.071043 (zitiert auf den Seiten 15, 56 und 70).
- [apo] APOSTOLICO, Alberto (Hrsg.) ; GALIL, Zvi (Hrsg.): *Pattern Matching Algorithms* (zitiert auf Seite 6).
- [Bal09] BALZERT, Helmut: Werkzeuge. In: BALZERT, Helmut (Hrsg.): *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Heidelberg : Spektrum Akademischer Verlag, 2009. – ISBN 978-3-8274-2247-7, 59–97 (zitiert auf Seite 4).
- [BGBG16] BALOGH, G. ; GERGELY, T. ; BESZÉDES, Á ; GYIMÓTHY, T.: Are My Unit Tests in the Right Package? In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, S. 137–146 (zitiert auf den Seiten 18, 24 und 83).
- [BGJM16] BOJANOWSKI, Piotr ; GRAVE, Edouard ; JOULIN, Armand ; MIKOLOV, Tomas: Enriching Word Vectors with Subword Information. In: *arXiv:1607.04606 [cs]* (2016), Juli. <http://arxiv.org/abs/1607.04606>. – arXiv: 1607.04606 (zitiert auf Seite 8).
- [BHR] BERGROTH, L. ; HAKONEN, H. ; RAITA, T.: A survey of longest common subsequence algorithms. In: *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, S. 39–48 (zitiert auf Seite 7).
- [Blo17] BLOCH, J.: *Effective java*. Pearson Education, 2017 <https://books.google.de/books?id=BIpDDwAAQBAJ>. – ISBN 978-0-13-468604-2 (zitiert auf Seite 54).
- [BMC⁺] BILENKO, M. ; MOONEY, R. ; COHEN, W. ; RAVIKUMAR, P. ; FIENBERG, S.: Adaptive name matching in information integration. 18, Nr. 5, S. 16–23. <http://dx.doi.org/10.1109/MIS.2003.1234765>. – DOI 10.1109/MIS.2003.1234765 (zitiert auf Seite 8).
- [BT12] BADRI, Mourad ; TOURE, Fadel: Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes. In: *Journal of Software Engineering and Applications* 5 (2012), Nr. 7, S. 513 (zitiert auf den Seiten 15 und 76).
- [CKV19] CSUVIK, Viktor ; KICSI, András ; VIDÁCS, László: Evaluation of Textual Similarity Techniques in Code Level Traceability. In: MISRA, Sanjay (Hrsg.) ; GERVASI, Osvaldo (Hrsg.) ; MURGANTE, Beniamino (Hrsg.) ; STANKOVA, Elena (Hrsg.) ; KORKHOV, Vladimir (Hrsg.) ; TORRE, Carmelo (Hrsg.) ; ROCHA,

- Ana Maria A. (Hrsg.) ; TANIAR, David (Hrsg.) ; APDUHAN, Bernady O. (Hrsg.) ; TARANTINO, Eufemia (Hrsg.): *Computational Science and Its Applications – ICCSA 2019*, Springer International Publishing, 2019 (Lecture Notes in Computer Science). – ISBN 978-3-030-24305-0, S. 529–543 (zitiert auf den Seiten 17, 25, 34, 47, 56 und 79).
- [GAS05] GUPTA, Vandana ; AGGARWAL, K ; SINGH, Yogesh: A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability. In: *Journal of Computer Science* 1 (2005), 02. <http://dx.doi.org/10.3844/jcssp.2005.276.282>. – DOI 10.3844/jcssp.2005.276.282 (zitiert auf Seite 16).
- [GBH⁺18] GERGELY, Tamás ; BALOGH, Gergo ; HORVATH, Ferenc ; VANCICS, Béla ; BESZÉDES, Árpád ; GYIMÓTHY, Tibor: Analysis of Static and Dynamic Test-to-code Traceability Information. In: *Acta Cybernetica* 23 (2018), 01, S. 903–919. <http://dx.doi.org/10.14232/actacyb.23.3.2018.11>. – DOI 10.14232/actacyb.23.3.2018.11 (zitiert auf den Seiten 16 und 77).
- [GGR15] GHAFARI, M. ; GHEZZI, C. ; RUBINOV, K.: Automatically identifying focal methods under test in unit test cases. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, S. 61–70 (zitiert auf den Seiten 3, 16 und 72).
- [Hey19] HEY, Tobias: INDIRECT: INTent-DrIven Requirements-to-Code Traceability. In: MUSSBACHER, Gunter (Hrsg.) ; ATLEE, Joanne M. (Hrsg.) ; BULTAN, Tevfik (Hrsg.): *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.*, IEEE / ACM, 2019, 190–191 (zitiert auf Seite 13).
- [HT03] HUNT, Andy ; THOMAS, Dave: *Pragmatic unit testing in Java with JUnit*. The Pragmatic Bookshelf, 2003 (zitiert auf den Seiten 3, 4 und 30).
- [JMSG07] JIANG, L. ; MISHERRHI, G. ; SU, Z. ; GLONDU, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: *29th International Conference on Software Engineering (ICSE'07)*, 2007. – ISSN 0270-5257, S. 96–105 (zitiert auf Seite 18).
- [Kar18] KARANI, Dhruvil: *Introduction to Word Embedding and Word2Vec*. <https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>. Version: September 2018 (zitiert auf den Seiten 7, 8 und 9).
- [KGDP11] KLOCK, Samuel ; GETHERS, Malcom ; DIT, Bogdan ; POSHYVANYK, Denys: Traceclipse: An eclipse plug-in for traceability link recovery and management. In: *Proceedings - International Conference on Software Engineering* (2011), 01. <http://dx.doi.org/10.1145/1987856.1987862>. – DOI 10.1145/1987856.1987862 (zitiert auf den Seiten 10, 17, 25, 27, 28 und 68).
- [Kic18] KICSI, András: Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability. (2018), S. 7 (zitiert auf den Seiten 17 und 84).
- [McC76] MCCABE, T. J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dezember, Nr. 4, S. 308–320. <http://dx.doi.org/10.1109/TSE.1976.233837>. – DOI 10.1109/TSE.1976.233837. – ISSN 0098-5589 (zitiert auf Seite 5).
- [MCCD13] MIKOLOV, Tomas ; CHEN, Kai ; CORRADO, Greg S. ; DEAN, Jeffrey: *Efficient Estimation of Word Representations in Vector Space*. <http://arxiv.org/abs/1301.3781>. Version: 2013 (zitiert auf Seite 8).

- [MLBK02] MALAIYA, Y. K. ; LI, M. N. ; BIEMAN, J. M. ; KARCICH, R.: Software reliability growth with test coverage. In: *IEEE Transactions on Reliability* 51 (2002), Dec, Nr. 4, S. 420–426. <http://dx.doi.org/10.1109/TR.2002.804489>. – DOI 10.1109/TR.2002.804489. – ISSN 0018–9529 (zitiert auf Seite 1).
- [MM01] MARCUS, A. ; MALETIC, J. I.: Identification of high-level concept clones in source code. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001. – ISSN 1938–4300, S. 107–114 (zitiert auf den Seiten 18 und 81).
- [MRS08] MANNING, Christopher D. ; RAGHAVAN, Prabhakar ; SCHÜTZE, Hinrich: *The term vocabulary and postings lists*. <http://dx.doi.org/10.1017/CB09780511809071.003>. Version: Juli 2008 (zitiert auf den Seiten 5 und 6).
- [PKA15] PARIZI, R. M. ; KASEM, A. ; ABDULLAH, A.: Towards gamification in software traceability: Between test and code artifacts. In: *2015 10th International Joint Conference on Software Technologies (ICSOFT)* Bd. 1, 2015, S. 1–8 (zitiert auf Seite 74).
- [PMP⁺15] PAWLAK, Renaud ; MONPERRUS, Martin ; PETITPREZ, Nicolas ; NOGUERA, Carlos ; SEINTURIER, Lionel: Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. In: *Software: Practice and Experience* 46 (2015), 1155–1179. <http://dx.doi.org/10.1002/spe.2346>. – DOI 10.1002/spe.2346 (zitiert auf Seite 37).
- [Por80] PORTER, Martin F.: An algorithm for suffix stripping. In: *Program* 40 (1980), S. 211–218 (zitiert auf Seite 5).
- [QBO⁺11] QUSEF, A. ; BAVOTA, G. ; OLIVETO, R. ; LUCIA, A. D. ; BINKLEY, D.: SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, S. 63–72 (zitiert auf den Seiten 15 und 16).
- [QOD10] QUSEF, A. ; OLIVETO, R. ; DE LUCIA, A.: Recovering traceability links between unit tests and classes under test: An improved method. In: *2010 IEEE International Conference on Software Maintenance*, 2010. – ISSN 1063–6773, S. 1–10 (zitiert auf den Seiten 10, 64 und 74).
- [Qus13] QUSEF, A.: Test-to-code traceability: Why and how? In: *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, 2013, S. 1–8 (zitiert auf den Seiten 9 und 11).
- [RKG04] ROUNTEV, Atanas ; KAGAN, Scott ; GIBAS, Michael: Static and Dynamic Analysis of Call Chains in Java. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA : ACM, 2004 (ISSTA '04). – ISBN 978–1–58113–820–7, 1–11. – event-place: Boston, Massachusetts, USA (zitiert auf Seite 5).
- [VRD09] VAN ROMPAEY, Bart ; DEMEYER, Serge: Establishing Traceability Links between Unit Test Cases and Units under Test, 2009, S. 209–218 (zitiert auf den Seiten 9, 10 und 66).

Anhang

A First Appendix Section

B Literaturanalsen

B.1 Recovering traceability links between unit tests and classes under test: An improved method

[QOD10] QUSEF, A. ; OLIVETO, R. ; DE LUCIA, A.: Recovering traceability links between unit tests and classes under test: An improved method. In: *2010 IEEE International Conference on Software Maintenance*, 2010. – ISSN 1063–6773, S. 1–10

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Informationen über Zusammenhang zwischen Komponententest und getestete Klasse finden.
- I2 Entwickeln einer neuen Methode dazu, die Datenflussanalyse verwendet.
- I3 Extrahieren der Informationsgewinnung automatisieren.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Benennungsregeln liefern nur Ergebnisse wenn ein striktes Schema vorhanden ist.
- D2 Letzter Aufruf vor Bestätigung (Last Call Before Assert (LCBA)) macht die Annahme, dass in der Bestätigung die zu testende Klasse genutzt wird.
- D3 Beide Ansätze erzeugen nur eins zu eins Verbindungen zwischen Komponententest und getestete Klasse.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Jede Methode in dem Komponententest gehört zu einer getesteten Klasse
- P2 Es wird nur die letzte Bestätigung betrachtet.
- P3 Die Datenflussanalyse kann nur lokale Deklarationen beachten.
- P4 Es wird nur Zusammengehörigkeit auf Klassenlevel betrachtet.
- P5 Jeder Komponententest muss eine Überprüfung besitzen.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Datenflussanalyse von letzter Überprüfung rückwärts, die alle Parameter darin betrachtet.
- L2 Standardklassen werden gefiltert und der Rest sind nun zusammenhängende Klassen.
- L3 Wegen der Datenflussanalyse ist das Verfahren unabhängig von Benennungsschema.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Analyse auf zwei Softwaresystemen, AgilePlanner und Mondiran.
- N2 Datengrundlage wurde von Doktoranden erstellt, die bei der Erstellung dies nicht wussten.
- N3 Vergleich mit LCBA und Benennungsregeln. Als Metrik wurde Vollständigkeit, Präzision und F-Maß hinsichtlich des erstellten Datensatzes verwendet.
- N4 Vergleich von verschiedenen Kombinationen der Verfahren.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F 1 Übertragbarkeit auf mehrere Überprüfungen, bzw. gibt es eine Transformation für jeden Komponententest in Komponententests die nur eine Überprüfung besitzen.
- F 2 Vorgehen falls in der Überprüfung nicht die getestete Klasse aufgerufen wird sondern z.B. eine Datenbankabfrage gemacht wird.
- F 3 Erkennung wenn Benennungsschema vorhanden sind und diese dann adaptiv nutzen, da diese eine sehr hohe Genauigkeit haben.
- F 4 Wie ist die Skalierbarkeit auf große Projekte ? Hier wurden nur 27 bzw. 41 Komponententest pro Projekt betrachtet.
- F 5 Nicht nutzbare Test wie NetzwerkSockets wurden per Hand gefiltert. Geht dies auch automatisch?

Notizen

Die Studie mit nur 2 Projekten wirkt sehr klein. Das Problem der fehlenden Datensätze scheint häufig ein Hindernis zu werden.

B.2 Establishing Traceability Links between Unit Test Cases and Units under Test

[VRD09] VAN ROMPAEY, Bart ; DEMEYER, Serge: Establishing Traceability Links between Unit Test Cases and Units under Test, 2009, S. 209–218

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I 1 Erläuterung und Vergleich der bekannten Methoden mit ihren Problemen.
- I 2 Entwicklung einer neuen Methode die auf statischen Aufrufgraphen basiert.
- I 3 Evaluation der neuen Methode im Vergleich zu bereits bekannten Methoden.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D 1 Benennungsregeln liefern nur Ergebnisse wenn ein striktes Schema vorhanden ist.
- D 2 Analyse der Instanzvariablen liefert nur Ergebnisse, falls dies strikt verwendet wurde. (Fixture Element Types)
- D 3 Analyse des statischen Aufrufgraphen liefert potentiell alle Helferklassen und Objekte. Dadurch ist die Menge zu groß.
- D 4 Die Testklasse hat bei lexikalische Analyse ein größeres Vokabular, wegen Wörtern wie `test.setUp`.
- D 5 Bei einheitlicher Entwicklung (Co-Evolution) müssen Getestete und Testklasse zeitgleich geändert werden.
- D 6 Die Tests müssen während der Entwicklung entstehen

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P 1 Die Überprüfung muss die getestete Methode aufrufen.
- P 2 Die Vergleichsdaten sind von den Entwicklern selber geliefert worden.
- P 3 Es wurden maximal 30 Komponententests verwendet.
- P 4 Es wird nur eine Verbindung auf Klassenlevel erzeugt.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L 1 Letzter Aufruf vor Bestätigung(Last Call Before Assert(LCBA)) wird nur überprüft und betrachtet.
- L 2 Nutzung eines statischen Aufrufgraphen, der aber früher reduziert wird.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N 1 Es wird die Anwendbarkeit, Präzision und Vollständigkeit von LCBA mit den anderen Ansätzen verglichen.
- N 2 Testsysteme waren JPacman, ArgoUML und Mondrian.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F 1 Ist die automatische Erkennung von der Anwendbarkeit der Algorithmen möglich ?
- F 2 Ist die Entwicklung einer automatischen Toolbox möglich, die eine Menge an Algorithmen enthält ?
- F 3 Wie wird das Problem mit mehreren Überprüfungen in einem Testfall gelöst ?

Notizen

B.3 Traceclipse: An eclipse plug-in for traceability link recovery and management

[KGD11] KLOCK, Samuel ; GETHERS, Malcom ; DIT, Bogdan ; POSHYVANYK, Denys: Traceclipse: An eclipse plug-in for traceability link recovery and management. In: *Proceedings - International Conference on Software Engineering* (2011), 01. <http://dx.doi.org/10.1145/1987856.1987862>. – DOI 10.1145/1987856.1987862

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Vorstellung einer Eclipseerweiterung, welches die Verbindungen zwischen Komponententests zu Quellcode analysiert und extrahiert.
- I2 Implementierung eines Präprozessors für Java, als Vorbereitung für die Verbindungen.
- I3 Herstellung der Verbindungen über ein auf Eclipse Lucene¹ basierendes Verfahren.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Keine allgemeine Plattform für die Forschung verfügbar.
- D2 Hinzufügen von Verbindungen nicht in jeder Tool möglich.
- D3 Granularität meist nur auf Klassenlevel.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Textuelle Ähnlichkeit muss gegeben sein, sonst sind keine oder nur sehr schlechte Ergebnisse vorhanden.
- P2 Das erstellen einer allgemeinen Plattform ist im Vordergrund.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Navigation und Steuerung über grafische Oberfläche.
- L2 Auswählbarkeit der Granularität und der zu verwendenden Quellen möglich.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Auf vier Open-Source Projekten fand die Evaluation statt, *iTrust*, *RealEstate*, *CM1* und *EasyClinic*.
- N2 Es wurden als Vergleichsmetriken Vollständigkeit und Präzision hinsichtlich der korrekten Lösung genutzt.

¹<http://lucene.apache.org/core/>

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

F 1 Durchsetzbarkeit der Interface nicht klar.

F 2 Hinzufügen von weiteren Tools und Kombination davon möglich, aber wie schwer ist dies?

F 3 Implementierung ist im Vergleich zu ähnlichen sehr schlecht vom Ergebnis her. Liegt dies an dem Datensatz?

Notizen

Es handelt sich eher um eine Plattform auf die aufgebaut werden sollte, statt einer vollkommen neuen Technik. Der Datensatz wurde von den Autoren nicht erstellt, sondern es wurden nur die gefunden überprüft ob diese stimmen. Wie dabei die Metrik Vollständigkeit berechnet werden konnte ist nicht ganz offensichtlich.

B.4 Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects

[AMH16] ARSHIA, Amir H. ; MOSTAFA, Seyed ; HADI, Mohammad: Recovering and Tracing Links between Software Codes and Test Codes of the Open Source Projects. 7 (2016). <http://dx.doi.org/10.14569/IJACSA.2016.071043>. – DOI 10.14569/IJACSA.2016.071043

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Entwicklung eines Algorithmus zur Informationsgewinnung von zusammenhängenden Klassen.
- I2 Darstellung der Entwicklungsentscheidungen und Auswertung des Verfahrens auf Software.
- I3 Die Implementierung des Datagewinnungsverfahrens und die Vorteile davon.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 *Es werden die Methoden genannt, aber nicht warum diese nicht gut genug sind.*

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Der erste Schritt muss Ergebnisse produzieren und ist maßgeblich verantwortlich für den Rest.
- P2 Ohne textuelle Ähnlichkeit ist keine Analyse möglich.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Ein vierstufiger Algorithmus auf Datensuche basierend.
- L2 Im ersten Schritt werden alle Namen von Testklassen und Klassen extrahiert, die gleich benannt sind.
- L3 Im zweiten Schritt werden Besonderheiten wie die Komplexität und Design Metriken extrahiert.
- L4 Die Verbindung von Schritt eins werden als Vergleichsdaten genommen um aus den Daten von Schritt zwei Informationen zu gewinnen.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 `ArgoUML`, `Apache Ant` und `Dependency Finder` wurden zur Evaluation verwendet.
- N2 Es wurden als Vergleichsmetriken Vollständigkeit und Präzision hinsichtlich der korrekten Lösung genutzt.
- N3 Als Vergleichsmethode wurde `Scotch+` verwendet.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F 1 Wie verhält sich das Verfahren wenn es in Phase eins keine Informationen gewinnen kann?
- F 2 Wie ist die Skalierbarkeit des Verfahrens hinsichtlich größerer Software?
- F 3 Woher kommt die Lösung der Verbindungen in den Testprojekten?

Notizen

B.5 Automatically identifying focal methods under test in unit test cases

[GGR15] GHAFARI, M. ; GHEZZI, C. ; RUBINOV, K.: Automatically identifying focal methods under test in unit test cases. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, S. 61–70

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Informationen über Zusammenhang zwischen Komponententest und getestete Klasse finden.
- I2 Entwickeln einer neuen Methode dazu, die Datenflussanalyse verwendet.
- I3 Extrahieren der Informationsgewinnung automatisieren.
- I4 Wechsel der Granularität der Verbindungen zu Methodenlevel vom Klassenlevel aus anderen Arbeiten.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Bestehende Arbeiten gewinn nur die Information welche Klasse getestet wird, aber nicht welche Methoden.
- D2 Letzter Aufruf vor Überprüfung (LCBA) betrachtet nicht immer die getestete Methode sondern manchmal ein Getter.
- D3 Dynamische Aufrufgraphen liefern eine zu große Methodenmenge zurück.
- D4 Benennungsregeln benötigen ein feste Schema.
- D5 Textuelle Ähnlichkeit funktioniert nur bei konsistenter Benennung.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Es muss Zustandsobjekte geben für die Analyse.
- P2 Zielsprache ist Java, mit Komponententests die das JUnit Rahmenwerk nutzen.
- P3 Dynamisches Nachladen von Implementierungen von Interfaces nicht verfolgbar.
- P4 Hilfsmethoden und vererbte Methoden sind für die Analyse nicht nutzbar.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Gewinnung der getesteten Methode über Dataflussanalyse und Abhängigkeiten.
- L2 Analysen finden auf dem Quellcode der Tests und dem Produktivcode statt.
- L3 Betrachten von jedem Testfall als abgeschlossene Einheit.
- L4 Einteilung der Methoden in Abfragemethoden(Inspector) ohne Seiteneffekt und Änderungsmethoden(mutators).
- L5 Letzte Änderungsmethode im Unterszenario wird als getestete Methode betrachtet.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 300 Komponententests aus vier Open-Source Projekten (Commons Email, JGAP, PureMVC und XStream).
- N2 Als Metrik wurde Vollständigkeit, Präzision und F-Maß hinsichtlich des erstellten Datensatzes verwendet.
- N3 Analyse der Ergebnisse und Begründung der unterschiedlichen Ergebnisse.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F1 Vergleich der Methode mit anderen Lösungen wie z.B. Scotch wäre spannend.
- F2 Lassen sich die Prämissen verringern, also könnte man die Hilfsmethoden trotzdem analysieren ?
- F3 Wäre ein manueller Input des Programmierers möglich und würde dies dem System zu besseren Ergebnissen helfen ?
- F4 Ist die Auswahl der Komponententests, die analysiert werden, automatisch möglich. ?

Notizen

Im Paper selbst wird geschrieben, dass das Ergebnis mit einer sauberen Implementierung besser wäre. Dies lässt Zweifel bei dem Datensatz aufkommen. Starker Abhängigkeiten von dem Eclipse Rahmenwerk.

B.6 Towards Gamification in Software Traceability: Between Test and Code Artifacts

[PKA15] PARIZI, R. M. ; KASEM, A. ; ABDULLAH, A.: Towards gamification in software traceability: Between test and code artifacts. In: *2015 10th International Joint Conference on Software Technologies (ICSOFT)* Bd. 1, 2015, S. 1–8

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Begründung einer neuen Forschungsrichtung in der Test zu Quelltext Forschung.
- I2 Transfer von Ansätzen aus Spieltheorie zur Informationsgewinnung.
- I3 Erläuterung der Probleme der aktuellen Arbeit in dem Bereich.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Bisher ist die Granularität nur auf Klassenlevel.
- D2 Es wird in keiner Methode eine korrekte Lösung erzeugt. Es gibt sowohl falsche als auch redundante Ergebnisse.
- D3 Alle gefunden Verbindungen müssen manuell verifiziert werden.
- D4 Fehlende Visualisierung der Verbindungen.
- D5 Keines der Verfahren ist interaktiv und selbstlernend beim Fortschreitend des Projektes.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Es wird nur der gedachte Ansatz vorgestellt.
- P2 Der Quelltext muss in Java geschrieben sein.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Zielplattform des Spiels sollte auch Android sein.
- L2 Mittelhilfe genetischer Algorithmen sollt sich das System anpassen können.
- L3 Nutzer besitzt die Möglichkeit dem System Informationen zu geben.
- L4 Es sind verschiedene grafische Darstellung wie z.B. Baumdiagramm vorgesehen.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 20 Leute und vier Softwareprojekte wurden verwendet.
- N2 Vergleich mit `Scotch` [QOD10].

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F 1 Wie realistisch ist es, dass Programmierer gute Ergebnisse produzieren, wenn sie nebenbei die Verbindungen anklicken ?
- F 2 Wie hoch wird die Akzeptanz dieser Anwendung sein ?
- F 3 Ist es überhaupt möglich ein vollständiges fehlerfreies Ergebnis zu produzieren ?
- F 4 Ist der Ansatz nicht nur das manuelle labeln von Daten ?

Notizen

Hier kommen Notizen hin, die in keine der obigen Kategorien passen.

Das Paper setzt sich viele Ziele, aber liefert keine Nachweise warum es diese erreichen sollte. Kein Folgepaper mit Ergebnissen auffindbar. Die 90% Erfolg wirken sehr unglaubwürdig

B.7 Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes

[BT12] BADRI, Mourad ; TOURE, Fadel: Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes. In: *Journal of Software Engineering and Applications* 5 (2012), Nr. 7, S. 513

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Vergleich von verschiedenen Metriken, um den Aufwand von testen des Quelltextes zu quantifizieren.
- I2 Transfer von bekannten Metriken aus der Objektorientierung in ein neues Feld.
- I3 Analyse der Aussagen verschiedener Metriken auf die Testbarkeit.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Vorherige Arbeiten haben nicht die verschiedenen Methoden verglichen und kombiniert.
- D2 Die Definition von Softwaretestbarkeit ist nicht präzise genug.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Es wurden nur Tests betrachtet die ein striktes Benennungsschema einhalten.
- P2 Die Testfälle müssen mithilfe des JUnit Rahmenwerks geschrieben sein.
- P3 Es wird nur in zwei Kategorien, hoch und niedrig, eingeteilt.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Es wird die Korrelation der Methoden berechnet anhand einer Studie.
- L2 Es wird getestet ob die verschiedenen Methoden kombiniert sinnvoll sind.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Evaluation auf drei Javaprojekten(ANT, JFC und POI).
- N2 Auswertung und Interpretation der Daten.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F1 Sind diese Einschränkungen notwendig ?
- F2 Wäre es möglich das ganze in mehrere Kategorien einzuordnen, oder einer Skala ?

Notizen

Hier kommen Notizen hin, die in keine der obigen Kategorien passen.

B.8 Analysis of Static and Dynamic Test-to-code Traceability Information

[GBH⁺18] GERGELY, Tamás ; BALOGH, Gergo ; HORVATH, Ferenc ; VANCSICS, Béla ; BESZÉDES, Árpád ; GYIMÓTHY, Tibor: Analysis of Static and Dynamic Test-to-code Traceability Information. In: *Acta Cybernetica* 23 (2018), 01, S. 903–919. <http://dx.doi.org/10.14232/actacyb.23.3.2018.11>. – DOI 10.14232/actacyb.23.3.2018.11

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Erkennung von schlechtem Testquelltext (Codesmell) hinsichtlich bekannter Kriterien.
- I2 Fokus auf die Erkennung von indirekter Nutzung von Quelltext innerhalb eines Tests. (Tests macht mehr als er sollte)
- I3 Kategorisierung der gefunden Fehler und Zuordnung dieser zu festgelegten Testdesignfehlern.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Es gibt noch kein vollautomatisches Verfahren zur Erkennung von indirekten Tests.
- D2 Verbindungen sind häufig auf Klassenlevel.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Klare Benennungsregeln müssen vorhanden sein.
- P2 Jeder Test muss eigenständig sein und nicht abhängig von anderen Tests sein.
- P3 Quelltext muss in Java geschrieben sein und die Komponententests müssen das JUnit Rahmenwerk nutzen.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Benutzung von statischer und dynamischer Quelltextanalyse.
- L2 Nutzung von vorhanden Methoden aus anderen Arbeiten.
- L3 Abgleich der statischen und dynamischen Analyseergebnisse und Zusammenführung dieser.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Vier Softwaresysteme, `checkstyle`, `netty`, `orientdb` und `oryx`.
- N2 Manuelle Überprüfung von Acht gefunden Testdesignfehlern.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

F 1 Kann die letzte Analyse nicht automatisiert werden?

F 2 Ist eine halbautomatische Analyse überhaupt sinnvoll, weil die Analyse deutlich mehr Informationen braucht als der Algorithmus verwendet?

Notizen

B.9 Evaluation of Textual Similarity Techniques in Code Level Traceability

[CKV19] CSUVIK, Viktor ; KICSI, András ; VIDÁCS, László: Evaluation of Textual Similarity Techniques in Code Level Traceability. In: MISRA, Sanjay (Hrsg.) ; GERVASI, Osvaldo (Hrsg.) ; MURGANTE, Beniamino (Hrsg.) ; STANKOVA, Elena (Hrsg.) ; KORKHOV, Vladimir (Hrsg.) ; TORRE, Carmelo (Hrsg.) ; ROCHA, Ana Maria A. (Hrsg.) ; TANIAR, David (Hrsg.) ; APDUHAN, Bernady O. (Hrsg.) ; TARANTINO, Eufemia (Hrsg.): *Computational Science and Its Applications – ICCSA 2019*, Springer International Publishing, 2019 (Lecture Notes in Computer Science). – ISBN 978-3-030-24305-0, S. 529–543

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Gewinnung von Test zu Quelltext Verbindungen über textuelle Ähnlichkeit.
- I2 Evaluation eines Test zu Quelltext Verfahren, welches textuelle Ähnlichkeit mit drei Maschinellen Lernverfahren berechnet.
- I3 Vergleich von den drei Techniken und einer Kombination davon.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Benennungsregeln besitzen keine ausreichende Anwendbarkeit, aber hohe Präzision.
- D2 Statische Quelltextanalyse haben eine höhere Anwendbarkeit, aber eine niedrige Präzision.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Der Quelltext muss in Java mit der JUnit Rahmenarchitektur geschrieben schreiben.
- P2 Ähnlichkeitsalgorithmen liefern meist nur ein Vorschlag.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Erstellung von drei verschiedenen Quelltextrepräsentationen.
- L2 Anwendung von drei Ähnlichkeitsverfahren(LSI, Doc2Vec, TF-IDF) auf den Quelltext.
- L3 Vergleich der Ergebnisse und Auswertung auf Basis davon.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Evaluation auf Commons Math, Commons Lang, JFreeChart und Mondrian.
- N2 Vergleich mit Benennungsregeln.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

F 1 Wie wurden die Algorithmen trainiert?

F 2 Wie gute ist der Vergleich mit der Musterlösung?

Notizen

Es wird nur eine Vergleich mit Benennungsregeln erstellt und nicht mit dem gesamten. Also wird keine Musterlösung genutzt. Die echten Ergebnisse wären spannend.

B.10 Identification of High-Level Concept Clones in Source Code

[MM01] MARCUS, A. ; MALETIC, J. I.: Identification of high-level concept clones in source code. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001. – ISSN 1938–4300, S. 107–114

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Erkennung von Quelltextkopien innerhalb eines Programms.
- I2 Quelltextkopien werden als Implementierung einer ganzen Funktionalität betrachtet und nicht auf einzelnen Quelltextzeilen.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Verschiedene NLP Methoden benötigen ein Grunddatensatz, dessen Erstellung teuer ist.
- D2 Bisherige Informationsgewinnungsmethoden, wie statisch oder Heuristik basierte, sind zwar leichter in der Erstellung aber produzieren schlechtere Ergebnisse.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Klassen die Funktionalität zusammen implementieren sind im zweiten Schritt verwendet und müssen mit anderen Methoden gefunden werden.
- P2 Die Methode alleine ist nicht vollautomatisch.
- P3 Die Methode basiert auf Benennung und ist dann nicht nutzbar, falls der Entwickler Kopien umbenennt.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Mittels Latent Semantic Indexing wird die Ähnlichkeit von semantische Quelltextartefakten erzeugt.
- L2 Die Ähnlichkeit wird über strukturelle Informationen gewonnen.
- L3 Dann gibt der Nutzer Quelltextartefakte an, welche Funktionalität zusammen implementieren.
- L4 Im nächsten Schritt werden ähnliche Cluster gesucht, wie im vorherigen Schritt angegeben wurden.
- L5 Dann wird endgültig mittels Dataflussanalyse entschieden, ob es eine Kopie ist.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Es wurde die Methode auf Mosaic, geschrieben in C, getestet.
- N2 Mosaic hat eine niedrige Quelltextqualität laut den Autoren.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F 1 Das Paper schlägt vor verschiedene Methoden zu kombinieren. Es wäre gut die Ergebnisse zu sehen.

Notizen

Die Erkennung von ähnlichen ASTs könnte Aufschluss geben. Bei gleichen ASTs sollte gleiche Verbindungen entstehen, falls nicht muss man von einem Fehler im Verfahren ausgehen.

B.11 Are my Unit Tests in the Right Package?

[BGBG16] BALOGH, G. ; GERGELY, T. ; BESZÉDES, Á ; GYIMÓTHY, T.: Are My Unit Tests in the Right Package? In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, S. 137–146

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Überprüfung der Atomarität und Organisation von Komponententest mittels automatischer Überprüfung.
- I2 Entwicklung eines Algorithmus für die Erkennung.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Es wird auf ähnliche Forschung in dem Bereich nicht eingegangen.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Wegen der dynamischen Analyse muss der Quelltext ausführbar sein.
- P2 Wegen der statischen Analyse muss der Quelltext vorliegen.
- P3 Die Zielsprache ist Java mit der JUnit Rahmenarchitektur.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 Vergleich von dynamischer und statischer Analyse des Quelltexts.
- L2 Erstellung einer binären Testüberdeckungsmatrix für jeden Komponententest und Erkennung von zusammengehörigen Gruppen.
- L3 Statische Analyse der Einteilung in Klassen und Pakete von dem Quelltext.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Evaluation auf acht Quelltextoffenen Projekten wie z.B. `netty`
- N2 Beispielanalyse von gefundenen Fehlern.

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

- F1 Lassen sich die Verbesserungsvorschläge automatisch erzeugen?

Notizen

B.12 Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability

[Kic18] KICSÍ, András: Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability. (2018), S. 7

Inhalte

Was sind die zentralen Inhalte (Themen, interessante Aussagen, Botschaften, Fragestellungen), die in der Arbeit (d.h., in der analysierten Literatur) behandelt werden?

- I1 Gewinnung von Test zu Quelltextverbindungen, wovon die Analyse halbautomatisch ist.
- I2 Erweiterung von LSI von eins zu eins Verbindungen zu eins zu N.
- I3 Vergleich mit LSI ohne mehrerer Vorschläge.

Defizite

Welche Defizite bestehender Arbeiten und Lösungen werden als Motivation der eigenen Lösungen genannt?

- D1 Benennungsregeln sind nur anwendbar wenn ein festes Benennungsschema vorliegt. Wenn diese anwendbar sind wird aber eine sehr hohe Präzision erreicht.

Prämissen

Welche Einschränkungen und Vorgaben werden hinsichtlich der eigenen Lösungen gemacht?

- P1 Es muss eine textuelle Ähnlichkeit zwischen Komponententest und Quelltext vorliegen.
- P2 Die Zielsprache ist Java mit der JUnit Rahmenarchitektur.
- P3 Das System liefert nur bis zu zehn Vorschläge, diese müssen dann manuell bewertet werden.
- P4 Die Verbindungen werden auf Klassenlevel gesucht.

Lösungen

Was sind die eigenen Lösungen der Autoren?

- L1 In einer ersten Evaluation wurde festgestellt, dass Benennungsregeln in manchen Projekten nur zu 20% eingehalten wurden.
- L2 Erstellung von zwei Datensätzen mittels statischer Analyse und Textverarbeitung von Quelltext und Komponententests.
- L3 Vergleich von diesen beiden mittels Latent Semantic Indexing.
- L4 Erstellung einer Liste mit N ähnlichen Klassen pro Testfall.

Nachweise

Welche Nachweise (Evidence) werden hinsichtlich der Tragfähigkeit der eigenen Lösungen geliefert?

- N1 Evaluation auf fünf Quelltextoffenen Projekten `Comm. Lang`, `Comm. Math`, `ArgoUML`, `Mondirán` und `JFreeChart`
- N2 Vergleich mit LSI mit nur eins zu eins Verbindung

Offene Fragen

Welche Fragen sind noch ungelöst geblieben beziehungsweise stellen sich dem Leser?

F1 Lässt sich das Verfahren auf Methodenlevel verbessern?

Notizen

Der Fakt dass LSI unabhängig von Benennungsregeln ist, ist ein wichtiges Ergebnis was das Paper hervorheben sollte Bei Top10 produziert das verfahren wirklich extrem gute Dinge. Wenn man diese Top10 Filtert ist man am Ziel.