Karlsruhe Institute of Technology

Entwurf und Aufbau einer semantischen Repräsentation von Quelltext

Masterarbeit
von

Felix Eurich

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

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

Bearbeitungszeit: 01.08.2019 – 31.01.2020

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, 31.01.2020

Felix Eurich

(**Felix Eurich**)

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.

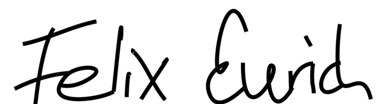
Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

Art der Abschlussarbeit:	Masterarbeit
Titel:	Entwurf und Aufbau einer semantischen Repräsentation von Quelltext
Datum:	31.01.2020
Name:	Felix Eurich

Karlsruhe, 31.01.2020



(**Felix Eurich**)

Kurzfassung

Eine Herausforderung bei der Rückverfolgung von Quelltext zu Anforderungen stellt die Analyse von Quelltext dar. Informationen über semantische Zusammenhänge zwischen Programmelementen sind darin nicht explizit dokumentiert, sondern müssen aus vorhandenen Informationen wie der natürlichen Sprache oder den strukturellen Abhängigkeiten abgeleitet werden. Im Rahmen des Forschungsprojekts INDIRECT wird eine semantische Repräsentation von Quelltext entworfen und umgesetzt, um die darin enthaltenen Informationen bei der Rückverfolgung von Anforderungen nutzen zu können. Die Repräsentation umfasst sowohl syntaktische Informationen als auch semantische Zusammenhänge im Quelltext. Für die Identifikation von semantischen Zusammenhängen wird eine Analyse der Syntax und eine Analyse der lexikalischen Bestandteile im Quelltext durchgeführt. Abschließend erfolgt eine Clusteranalyse auf Basis der gefundenen Zusammenhänge, um Gruppen aus semantisch zusammenhängenden Programmelementen zu identifizieren. Bei der Evaluation wurde eine Abdeckung der gefundenen Programmelementgruppen von bis zu 0,91 erzielt. Die Präzision der gefundenen Cluster betrug bis zu 0,9. Das harmonische Mittel aus der Cluster-Abdeckung und der Cluster-Präzision erreichte einen maximalen Wert von 0,73.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung 1
1.2 Aufbau der Arbeit 2

2 Grundlagen **3**
2.1 Syntax und Semantik 3
2.2 Informationen im Quelltext 3
2.2.1 Abstrakter Syntaxbaum 3
2.2.2 Kontrollflussgraph 4
2.2.3 Entwurfsmuster 5
2.2.4 Quelltextkommentare 6
2.2.5 Bezeichner 7
2.3 Verarbeitung von natürlicher Sprache 7
2.3.1 Normalisierung von Wortformen 8
2.3.2 Stoppwortentfernung 8
2.3.3 Informationsrückgewinnung 8
2.3.4 Worteinbettung 9
2.3.5 Zeichenkettenvergleich 10
2.4 Hierarchische Clusteranalyse 12

3 Rückverfolgung von Anforderungen mit INDIRECT **15**

4 Verwandte Arbeiten **17**
4.1 Repräsentation von Quelltext 17
4.2 Konzeptlokalisierung 19
4.2.1 Syntaxbasierte Quelltextanalyse 19
4.2.2 Lexikalische Quelltextanalyse 20
4.2.3 Kombinierte Ansätze 21
4.3 Automatisierte Softwarerefaktorisierung 23
4.4 Programmverständnis 25
4.4.1 Identifikation von Entwurfsmustern 26
4.4.2 Auflösung von Abkürzungen im Quelltext 26

5 Analyse und Entwurf **29**
5.1 Zieldefinition 30
5.2 Semantische Repräsentation von Quelltext 30
5.2.1 Darstellung semantischer Zusammenhänge 31
5.2.2 Entwurf der semantischen Repräsentation 32
5.3 Identifikation semantischer Zusammenhänge 34
5.3.1 Syntaxbasierte Quelltextanalyse 35
5.3.1.1 Abhängigkeiten im Quelltext 36
5.3.1.2 Diskussion verwandter Arbeiten 44

5.3.1.3 Entwurfsmuster . . . 51
5.3.1.4 Kontrollfluss . . . 52
5.3.2 Entwurf der syntaxbasierten Quelltextanalyse . . . 54
5.3.2.1 Abhängigkeitsmerkmale . . . 54
5.3.2.2 Semantische Zusammenhänge . . . 56
5.3.3 Natürliche Sprache im Quelltext . . . 58
5.3.3.1 Diskussion verwandter Arbeiten . . . 62
5.3.3.2 Worteinbettung . . . 66
5.3.3.3 Abkürzungen . . . 67
5.3.3.4 Vorverarbeitung . . . 69
5.3.3.5 Semantische Zusammenhänge . . . 72
5.3.4 Entwurf der lexikalischen Quelltextanalyse . . . 74
5.3.4.1 Vorverarbeitung . . . 74
5.3.4.2 Semantische Zusammenhänge . . . 75
5.4 Aufbau der semantischen Quelltextrepräsentation . . . 78
5.4.1 Semantische Zusammenhänge . . . 78
5.4.2 Absichtsknoten . . . 79
5.4.3 Entwurf zum Aufbau der semantischen Quelltextrepräsentation . . . 82
5.5 Systementwurf . . . 83

6 Implementierung **85**
6.1 Aufbau der initialen Graphrepräsentation . . . 85
6.2 Syntaxbasierte Quelltextanalyse . . . 86
6.3 Lexikalische Quelltextanalyse . . . 87
6.3.1 Dokumentabbildung . . . 88
6.3.2 Worteinbettung . . . 91
6.3.3 Vergleich von Programmelementen . . . 91
6.4 Semantische Clusteranalyse . . . 92
6.5 Parameteroptimierung . . . 94
6.5.1 Evaluationsmetriken . . . 94
6.5.2 Vorgehen . . . 95
6.5.3 Ergebnisse . . . 96

7 Evaluation **101**
7.1 Datengrundlage . . . 101
7.2 Ergebnisse . . . 103

8 Zusammenfassung und Ausblick **111**

Literaturverzeichnis **113**

Anhang **119**
A Parameterkonfigurationen . . . 120
B Elemente der Graphrepräsentation . . . 125

Abbildungsverzeichnis

2.1 Schematische Darstellung eines abstrakten Syntaxbaums. 4
2.2 Beispiel für einen Kontrollflussgraphen. 5
2.3 Klassendiagramm des Strategiemusters. 5
2.4 Schematische Darstellung der CSG-Architektur. 10
2.5 Beispiel für eine Clusterhierarchie. 12

3.1 Schematische Darstellung des Verarbeitungsprozesses von INDIRECT. . . . 15

5.1 Bestandteile der semantischen Repräsentation. 34
5.2 Abhängigkeitsmerkmale eines Ausdrucks. 37
5.3 Abhängigkeiten einer Deklaration, Initialisierung und Zuweisung. 38
5.4 Abhängigkeiten einer `for`- und `foreach`-Schleife. 39
5.5 Abhängigkeiten einer `try-catch`-Anweisung. 40
5.6 Abhängigkeiten einer Methodendeklaration. 41
5.7 Abhängigkeiten einer Klassensignatur. 42
5.8 Möglichkeiten der Dokumentabbildung. 72
5.9 Abstraktes Vorgehen der lexikalischen Analyse. 75
5.10 Abstrakter Systementwurf 84

6.1 Beispiel für die Dokumentabbildung einer Methode. 90

Tabellenverzeichnis

2.1 Mögliche Markierungen in einem Dokumentationskommentar. 6

5.1 Überblick der Abhängigkeitsmerkmale einer Anweisung. 40
5.2 Überblick der Abhängigkeitsmerkmale einer Methode. 42
5.3 Überblick der Abhängigkeitsmerkmale eines Typs. 43
5.4 Überblick der direkten Abhängigkeitsmerkmale. 56

6.1 Ergebnisse der initialen und optimierten Parameterbelegung für Dronology. 96
6.2 Optimierte Parameter der syntaxbasierten Quelltextanalyse. 97
6.3 Optimierte Parameter der lexikalischen Quelltextanalyse. 98
6.4 Optimierte Parameter der semantischen Clusteranalyse. 98

7.1 Kennzahlen der evaluierten Softwareprojekte. 102
7.2 Übersicht der Musterlösungen. 103
7.3 Ergebnisse für das Absichtsmodell aus Typen und Methoden. 104
7.4 Einfluss verschiedener Konfigurationen auf das Absichtsmodell. 105
7.5 Ergebnisse nach der Optimierung für verschiedene Konfigurationen. 106
7.6 Einfluss verschiedener Worteinbettungsmodelle. 107
7.7 Ergebnisse für das Absichtsmodell auf Typebene. 108
7.8 Übertragbarkeit von Parametern zwischen den Absichtsmodellen. 109

1 Einleitung

Bei der Entwicklung und Wartung von Softwaresystemen verbringen Programmierer einen Großteil ihrer Zeit mit dem Lesen und Nachvollziehen von Dokumentation und Quelltext [CZD+09]. Grund dafür ist, dass die Arbeit an einem Softwaresystem ein grundsätzliches Verständnis über die Implementierung und deren Zusammenhänge zu den Anforderungen erfordert [RW02]. Für die Erweiterung eines Buchungssystems um eine neue Bezahlart, muss ein Entwickler bspw. wissen, welche Stellen im Quelltext den Bezahlvorgang implementieren und wie die bisherigen Bezahlarten eingebunden sind. Insbesondere bei größeren Softwareprojekten ist der Aufbau von Programmverständnis mit einem hohen Zeitaufwand und folglich hohen Kosten verbunden. Diese Kosten können reduziert werden, wenn Zusammenhänge zwischen Anforderungen und Quelltext explizit dargestellt und für alle Entwickler zugänglich gemacht werden. Eine gut gepflegte Dokumentation kann dies leisten. Aufgrund des Kosten- und Zeitdrucks während der Entwicklung ist die Dokumentation jedoch häufig veraltet oder überhaupt nicht vorhanden. Folglich ist sie kaum nützlich, um die relevanten Stellen für eine Anforderung im Quelltext zu finden. Eine automatisierte Bereitstellung aktueller Informationen über die Zusammenhänge zwischen den Softwareanforderungen und dem Quelltext wäre daher wünschenswert. Eine Herausforderung bei der Suche nach Rückverfolgbarkeitsinformationen ist die Aufbereitung von Quelltext, um auf die enthaltenen semantischen Informationen zugreifen zu können. Im Gegensatz zur Syntax sind semantische Zusammenhänge im Quelltext nicht explizit gekennzeichnet. Welche Programmelemente eine gemeinsame Absicht verfolgen oder demselben Zweck dienen, muss bspw. aus Kommentaren abgeleitet werden.

1.1 Zielsetzung

In der vorliegenden Arbeit soll eine semantische Repräsentation von Quelltext entworfen und ein Verfahren für dessen Aufbau entwickelt werden. Die Repräsentation soll Quelltext und die darin enthaltenen semantischen Zusammenhänge beschreiben und einen einfachen Zugriff auf diese Informationen erlauben. Eine Erweiterung der Repräsentation um zusätzliches Wissen soll problemlos möglich sein. Die Informationen für den Aufbau der semantischen Quelltextrepräsentation sollen von verschiedenen Analysen bereitgestellt werden. Im Rahmen dieser Arbeit ist die Implementierung einer Syntaxanalyse und einer Analyse der lexikalischen Bestandteile vorgesehen. Das Verfahren zum Aufbau der semantischen Repräsentation soll die Integration weiterer Analysen für die Berücksichtigung zusätzlicher Informationen auf einfache Weise erlauben.

1.2 Aufbau der Arbeit

Der Inhalt dieser Arbeit ist wie folgt strukturiert. Kapitel 2 umfasst die theoretischen Grundlagen. Das Forschungsprojekt INDIRECT, zu dem diese Arbeit beiträgt, wird in Kapitel 3 vorgestellt. Anschließend liefert Kapitel 4 einen Überblick verwandter Arbeiten zur Repräsentation von Quelltext und der Suche nach semantischen Zusammenhängen. Kapitel 5 befasst sich mit der Problemstellung im Detail, analysiert mögliche Lösungen und stellt den Entwurf der semantischen Repräsentation sowie das Lösungskonzept für deren Aufbau vor. Die Umsetzung des Entwurfs wird in Kapitel 6 beschrieben. Die Bewertung der Ergebnisse dieser Arbeit erfolgt in Kapitel 7. Abschließend umfasst Kapitel 8 eine Zusammenfassung und einen Ausblick auf weiterführende Arbeiten.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen der vorliegenden Arbeit vermittelt. Neben verschiedenen Informationen im Quelltext wird auf die Verarbeitung von natürlicher Sprache und auf die hierarchische Clusteranalyse eingegangen.

2.1 Syntax und Semantik

Die Begriffe *Syntax* und *Semantik* werden im weiteren Verlauf dieser Arbeit häufig verwendet. Beide Begriffe besitzen sowohl im Kontext von Quelltext als auch bei der Verarbeitung von natürlicher Sprache eine Bedeutung. Um Unklarheiten zu vermeiden werden die Begriffe im Folgenden kurz beschrieben.
Die Syntax einer Programmiersprache ist eine formale Beschreibung des textuellen Aufbaus zulässiger Quelltexte und wird häufig in Form einer kontextfreien Grammatik angegeben [RPP06]. Sie besteht aus einer Reihe von Regeln, die definieren, welche Programmelemente im Quelltext vorkommen dürfen, wie diese aufgebaut bzw. kombiniert werden können und an welchen Stellen sie erlaubt sind. Im Gegensatz dazu beschreibt die Semantik eines Quelltextes dessen Bedeutung, also was bei der Ausführung passiert [Fer14].
Mit dem Begriff der Syntax werden im Kontext der natürlichen Sprache die Regeln zur Bildung von wohlgeformten Sätzen aus einzelnen Wörtern oder Konstituenten beschrieben [Buß02]. Folglich liefert eine Syntaxanalyse von natürlicher Sprache Informationen über die Struktur von Texten. Die Semantik der natürlichen Sprache widmet sich der Bedeutung von natürlichsprachlichen Ausdrücken und beschreibt bspw. die Aussage eines Satzes oder die Bedeutung einzelner Wörter [Buß02].

2.2 Informationen im Quelltext

Im Folgenden werden verschiedene Informationen im Quelltext betrachtet. Neben unterschiedlichen Darstellungsformen der Quelltextstruktur werden dessen natürlichsprachlichen Bestandteile betrachtet.

2.2.1 Abstrakter Syntaxbaum

Der abstrakte Syntaxbaum (AST) ist das Resultat einer statischen Syntaxanalyse und beschreibt die Quelltextstruktur durch einen Ableitungsbaum [LR03]. Programmelemente wie Typen, Methoden, Variablen oder Anweisungen bilden die inneren Knoten des AST. Die im Quelltext verwendeten Werte, Bezeichner, Modifikatoren und Typverweise werden

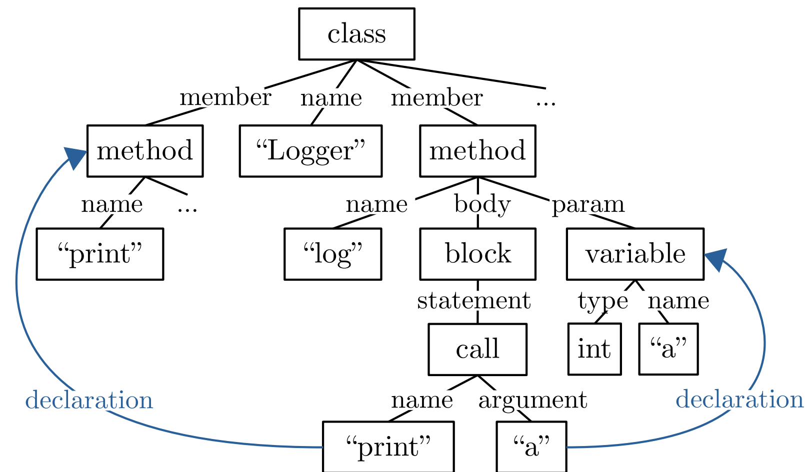


Abbildung 2.1: Schematische Darstellung eines abstrakten Syntaxbaums.

von den Blättern des Baums repräsentiert. Andere Symbole wie Klammern oder Semikola werden abstrahiert.
Zur Veranschaulichung zeigt Abbildung 2.1 eine schematische Darstellung des abstrakten Syntaxbaums einer Klasse. Der Wurzelknoten repräsentiert die Klassendefinition. Über *member*-Kanten sind die beiden enthaltenen Methodendefinitionen mit dem Klassenknoten verbunden. Darüber hinaus ist der strukturelle Aufbau der Methode `log` dargestellt. Sie besitzt einen Parameter `a` vom Typ `int` und enthält in ihrem Rumpf eine Anweisung, welche die Methode `print` aufruft. Der Bezeichner für die Klasse sowie die Methoden und Variablennamen finden sich in den Blättern des Baums. Der Verweis auf den Typ `int` der Variable a ist ebenfalls als Blatt dargestellt.

Mit Hilfe des AST kann auf den strukturellen Aufbau und die syntaktischen Abhängigkeiten eines Programmelements zugegriffen werden. Eine Abhängigkeit zwischen Programmelementen existiert bspw. aufgrund von Methodenaufrufen, Typ- oder Variablenreferenzen. In Abbildung 2.1 sind diese Abhängigkeiten durch die beiden *declaration*-Kanten angedeutet, die jeweils auf den Definitionsursprung der Methode `print` bzw. der Variable `a` verweisen. Aufgrund der umfassenden Beschreibung der Quelltextstruktur wird der AST auch häufig als Informationsquelle für den Aufbau individueller Quelltextrepräsentationen wie dem Aufrufgraphen verwendet. Der Aufrufgraph ist ein gerichteter Graph, der Methoden durch Knoten und Methodenaufrufe durch Kanten repräsentiert [ALSU07].

2.2.2 Kontrollflussgraph

Eine weitere Darstellungsmöglichkeit für Zusammenhänge im Quelltext bietet der Kontrollflussgraph [RPP06]. Er repräsentiert alle möglichen Ausführungspfade eines Programms in einem gerichteten Graphen und wird bspw. zur Programmoptimierung oder Datenflussanalyse eingesetzt.
Beim Aufbau des Kontrollflussgraphen wird Quelltext in Blöcke aus größtmöglichen Folgen von sequentiell ablaufenden Anweisungen ohne Verzweigungen oder Einsprünge unterteilt. Für alle diese Blöcke gilt, sobald die erste Anweisung des Blocks ausgeführt wird, werden auch alle anderen Anweisungen im Block ausgeführt. Die Blöcke bilden die Knoten des Kontrollflussgraphen. Eine Kante zwischen zwei Knoten v_1 und v_2 existiert genau dann, wenn die erste Anweisung im Block von v_2 direkt auf die letzte Anweisung des von v_1 repräsentierten Blocks folgen kann.
Abbildung 2.2 zeigt den Kontrollflussgraph für Quelltextausschnitt 2.1. Die Anweisungen innerhalb der Methode von Quelltextausschnitt 2.1 bilden vier Anweisungsblöcke. Der erste Block enthält als letzte Anweisung die Bedingung, weshalb der korrespondierende Knoten

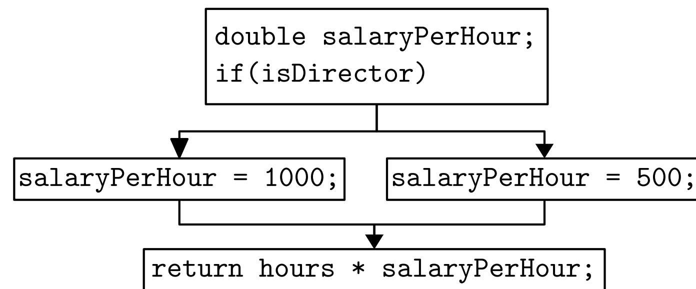


Abbildung 2.2: Beispiel für einen Kontrollflussgraphen.

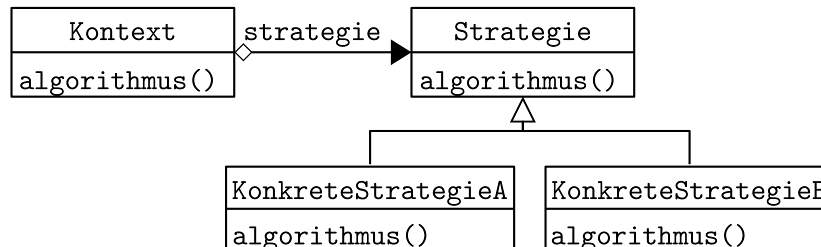


Abbildung 2.3: Klassendiagramm des Strategiemusters. Quelle: [GHJV04]

im Graphen zwei ausgehende Kanten besitzt. Unabhängig von der Bedingung wird in der letzten Zeile der Methode dieselbe Anweisung ausgeführt, weshalb der vierte Block zwei eingehende Kanten besitzt. Jeder Pfad im Graphen vom Start- zum Endknoten entspricht einem möglichen Programmablauf.

Quelltextausschnitt 2.1: Beispielmethode für den Kontrollflussgraphen aus Abbildung 2.2.

```
public double calcCompensation(double hours, boolean isDirector) {
    double salaryPerHour;
    if(isDirector) {
        salaryPerHour = 1000;
    } else {
        salaryPerHour = 500;
    }
    return hours * salaryPerHour;
}
```

2.2.3 Entwurfsmuster

Entwurfsmuster sind Schablonen für die Lösung wiederkehrender Probleme beim objektorientierten Entwurf von Softwaresystemen [GHJV04]. Sie umfassen abstrakte Beschreibungen von Typen, Beziehungen, Rollen und Interaktionen, die von den Entwicklern beim Softwareentwurf auf eine konkrete Problemstellung übertragen werden können.
Zur Visualisierung zeigt Abbildung 2.3 die Struktur des Entwurfsmusters *Strategie*. Es wird unter anderem genutzt, um die Verhaltensweise eines Programms zur Laufzeit ändern zu können. Die Klasse `Strategie` definiert hierzu eine gemeinsame Schnittstelle, die von den verschiedenen Algorithmen (Verhaltensweisen) implementiert wird. Die Implementierung der Algorithmen findet sich in den beiden Klassen `KonkreteStrategieA` und `KonkreteStrategieB`. Die Klasse `Kontext` besitzt eine Referenz zu einem Objekt vom Typ `Strategie`, welche mit dem Objekt einer konkreten Strategie besetzt wird. Zur Laufzeit kann durch das Setzen der entsprechenden Objektreferenz zwischen den implementierten Algorithmen gewechselt werden.

Tabelle 2.1: Ein Auszug der möglichen Markierungen in einem Dokumentationskommentar in Java. Quelle: `https://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html`, zuletzt Besucht am 01.08.2019

Markierung	Erklärung
@param	Beschreibung eines Methodenparameters.
@return	Beschreibung der Rückgabe einer Methode.
@see oder @link	Verweis auf eine Klasse, Methode, Attribut oder Paket.

2.2.4 Quelltextkommentare

Entwickler nutzen Kommentare, um den Zweck oder das Verhalten eines Quelltextausschnitts zu beschreiben. Obwohl Kommentare ein Bestandteil des Quelltextes sind, tragen sie nicht zu seiner Semantik bei, sondern dienen lediglich dem Zweck der Dokumentation. Grundsätzlich kann zwischen Block- und Zeilenkommentaren unterschieden werden [Pep07]. Wohingegen Zeilenkommentare genau einer Zeile im Quelltext entsprechen, kann sich ein Blockkommentar über mehrere Quelltextzeilen erstrecken. Der Text eines Kommentars unterliegt keinen Einschränkungen und wird keiner Syntaxüberprüfung unterzogen, weshalb Kommentare neben natürlichsprachlichen Texten bspw. Internetadressen, Quelltext und Sonderzeichen enthalten können. Auch Kommentare ohne jeglichen Inhalt sind möglich. Da es im Quelltext keine Vorschriften dazu gibt, wo ein Kommentar platziert werden muss, können Kommentare prinzipiell an jeder beliebigen Stelle platziert werden.

Werkzeuge wie JavaDoc[1] bieten mit Dokumentationskommentaren eine weitere Art von Kommentaren im Quelltext an. Dokumentationskommentare werden genutzt, um automatisiert eine Dokumentation des Quelltextes im HTML-Format[2] zu generieren. Sie erlauben den Entwicklern die Angabe zusätzlicher Informationen wie explizite Verweise auf Programmelemente anzugeben und können darüber hinaus mit Hilfe von HTML-Elementen formatiert werden. Einige mögliche Markierungen sind in Tabelle 2.1 beschrieben. Dokumentationskommentare finden sich vor allem an Typ- und Methodendefinitionen sowie an Attributen. Grundsätzlich können jedoch auch Dokumentationskommentare an beliebigen Stellen im Quelltext platziert werden.
Quelltextausschnitt 2.2 liefert ein Beispiel für einen Dokumentationskommentar. In der ersten Zeile ist eine allgemeine Beschreibung des Zwecks der Methode angegeben. Die letzten beiden Zeilen umfassen Beschreibungen des Eingabeparameters bzw. des Rückgabewerts der Methode. Durch die Markierungen am Anfang der letzten beiden Zeilen kann erkannt werden, welche Zeile sich worauf bezieht.

Quelltextausschnitt 2.2: Beispiel für einen Dokumentationskommentar an einer Methode.

```java
/**
 * Checks whether the given number is even or not.
 *
 * @param number the integer to check.
 * @return true if the number is even, otherwise false.
 */
public boolean isEven(int number) {
    return number % 2 == 0;
}
```

[1] `https://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html`, zuletzt besucht am 14.01.2020

[2] `https://html.spec.whatwg.org/multipage/`, zuletzt besucht am 13.01.2020

2.2.5 Bezeichner

Bezeichner sind Namen, die für Typen, Methoden und Variablen vergeben werden, um diese im Quelltext referenzieren zu können [Ull04]. Um das Programmverständnis zu erleichtern, sind Entwickler in der Regel bemüht, aussagekräftige Bezeichner zu wählen. Grundsätzlich kann ein Bezeichner aus einer beliebigen Folge von Zeichen bestehen. Allerdings sind Bezeichner, die mit einem Schlüsselwort der verwendeten Programmiersprache übereinstimmen, nicht erlaubt. Darüber hinaus darf ein Bezeichner keine Leerzeichen enthalten.
Häufig werden Bezeichner aus mehreren aneinandergereihten Begriffen zusammengesetzt, um deren Aussagekraft zu erhöhen. Damit sich die einzelnen Wörter dennoch leicht erkennen lassen, werden Namenskonventionen wie die Binnenmajuskel-Schreibweise verwendet. Hierbei wird ein neues Wort wie in `myIdentifier` durch einen Großbuchstaben gekennzeichnet. Weit verbreitet ist darüber hinaus auch die Trennung der einzelnen Begriffe durch einen Unterstrich wie in `my_identifier`. Grundsätzlich ist die Namenskonvention im Quelltext nicht vorgegeben und kann von den Entwicklern frei gewählt werden.
Damit Bezeichner die Lesbarkeit des Quelltextes nicht beeinträchtigen werden lange Wörter häufig durch Abkürzungen ersetzt. Ein Beispiel liefert der Bezeichner `myId`, welcher das Wort *identifier* durch *id* abkürzt.

Quelltextausschnitt 2.3 veranschaulicht die Relevanz von Bezeichnern für das Programmverständnis. Sowohl die obere als auch die untere Methode implementieren dieselbe Funktionalität. Durch die Verwendung von aussagekräftigen Bezeichnern wie `isMorning` oder `hours` in der oberen Methode ist ihr Zweck leicht erkennbar. Im Gegensatz dazu nutzt die untere Methode ausschließlich kryptische Bezeichner wie `type_1` oder `method_1`, wodurch die Nachvollziehbarkeit der implementierten Funktionalität erheblich erschwert wird.

Quelltextausschnitt 2.3: Beispiel aus [KDG07] für die Relevanz von Bezeichnern.

```
public boolean isMorning(int hours, int minutes, int seconds) {
    if(!isDate(hours, minutes, seconds)) {
        throw new Exception("Invalid input: not a time value");
    }
    return hours < 12 && minutes < 60 && seconds < 60;
}

public type_1 method_1(type_2 a, type_2 b, type_2 c) {
    if(!method_2(a, b, c)) {
        throw new type_3(literal_1);
    }
    return a < A && b < B && c < C;
}
```

2.3 Verarbeitung von natürlicher Sprache

Im Forschungsgebiet der Verarbeitung von natürlicher Sprache wird versucht, Computern ein Verständnis natürlichsprachlicher Ausdrücke zu ermöglichen. Populäre Anwendungsbeispiele für die rechnergestützte Sprachverarbeitung sind unter anderem Dialogsysteme wie der Google Assistent[3] oder automatische Übersetzungsdienste wie DeepL[4]. Darüber hinaus finden sich die Techniken und Methoden aus der Sprachverarbeitung in vielen weiteren Kontexten wie der automatisierten Generierung von Quelltext [WT15] oder der Analyse von Anforderungsdokumenten [KB09] wieder. Die folgenden Abschnitte führen in verschiedene Vorverarbeitungsschritte der Sprachverarbeitung ein und stellen darüber hinaus Möglichkeiten vor, Texte bzw. Wörter miteinander zu vergleichen.

[3] `https://assistant.google.com/`, zuletzt besucht am 14.01.2020
[4] `https://www.deepl.com/home`, zuletzt besucht a, 14.01.2020

2.3.1 Normalisierung von Wortformen

Wörter besitzen verschiedene Formen, deren Anwendung von der Grammatik der natürlichen Sprache bestimmt wird. In Abhängigkeit von der Formulierung eines Textes kann dasselbe Wort in unterschiedlichen Formen auftreten. Als Beispiel dienen die beiden Sätze „I'm writing a test“ und „I wrote a test“. In beiden Sätzen tritt das Wort *write* in unterschiedlicher Form auf. Um zu verhindern, dass Wörter wie *writing* und *wrote* aufgrund der unterschiedlichen Form als verschiedene Wörter aufgefasst werden, wird im Rahmen der Vorverarbeitung häufig eine Stammformreduktion oder Lemmatisierung durchgeführt, um die Wörter auf einheitliche Terme abzubilden [JM08]. Bei der Stammformreduktion handelt es sich um einen regelbasierten Ansatz, der Suffixe abschneidet, um Wörter auf ihren Wortstamm zurückzuführen. Hierbei kann es vorkommen, dass unterschiedliche Wörter wie *execute* und *executive* auf den gleichen Wortstamm, hier *execut*, abgebildet werden. Diesen Nachteil besitzt die Lemmatisierung nicht. Im Gegensatz zur Stammformreduktion wird bei der Lemmatisierung ein Wörterbuch genutzt, um ein Wort auf seine Grundform abzubilden. Da die Wörter *execute* und *executive* bereits in ihrer Grundform vorliegen, bleiben sie bei der Lemmatisierung unverändert. Allerdings stellen unbekannte Wörter bei der Lemmatisierung ein Problem dar, da ihre Grundform nicht im Wörterbuch gefunden werden kann.

2.3.2 Stoppwortentfernung

Stoppwörter sind Wörter, die häufig in der natürlichen Sprache auftreten und selbst kaum zur Bedeutung eines Textes beitragen [JM08]. Beispiele für Stoppwörter sind unter anderem *the*, *to* oder *a*. Sie erlauben isoliert betrachtet keine Rückschlüsse auf das Thema des Textes indem sie auftreten. Um den Fokus bei der Verarbeitung von natürlicher Sprache auf semantisch wertvollere Wörter zu lenken, ist es weit verbreitet, Stoppwörter im Rahmen der Vorverarbeitung zu entfernen. In der Regel werden die zu entfernenden Stoppwörter manuell als Liste vorgegeben.

2.3.3 Informationsrückgewinnung

Der Begriff Informationsrückgewinnung (engl. Information Retrieval) oder kurz IR ist ein Sammelbegriff für Verfahren, die nach Informationen in großen Dokumentensammlungen suchen [MRS08]. Als Dokument wird hierbei jegliche Art von Text wie einzelne Sätze oder ganze Zeitungsartikel bezeichnet. Die Menge aller Dokumente wird Dokumentensammlung genannt. Ein Informationsbedürfnis wird durch eine Anfrage in natürlicher Sprache ausgedrückt. Im Folgenden wird das Vorgehen des Vektorraummodells näher betrachtet. Sowohl die Anfrage als auch die Dokumente werden zunächst in eine Vektorrepräsentation überführt. Die Bestimmung der Relevanz einzelner Dokumente erfolgt anschließend durch den Vergleich ihrer Vektorabbildungen.

Die Grundlage des Vektorraummodells bildet das gemeinsame Vokabular aller Dokumente in der Dokumentensammlung. Die Anzahl der Wörter im Vokabular entspricht der Dimension des Vektorraums. Jedes Dokument d_j kann als Vektor $\vec{d_j} = (w_{1,j}, w_{2,j}, ..., w_{n,j})$ dargestellt werden, wobei jeder Eintrag $w_{x,j}$ die Relevanz des Terms t_x aus dem Vokabular für das Dokument d_j angibt. Als Indikator für die Relevanz eines Terms kann bspw. dessen Häufigkeit im Dokument herangezogen werden. Die Einträge der Dokumentvektoren können in einer Term-Dokument-Matrix abgelegt werden. Jedem Term des Vokabulars wird hierbei eine Spalte zugeordnet, jedem Dokument eine Zeile. Die Einträge der Matrix entsprechen den Gewichten der Terme für die einzelnen Dokumente.
Analog zur Dokumentabbildung kann eine Anfrage q als Vektor $\vec{q} = (w_{1,q}, w_{2,q}, ..., w_{n,q})$ dargestellt werden. Bei der Suche nach relevanten Dokumenten wird der Anfragevektor mit allen Dokumentvektoren verglichen. Zur Bewertung eines Dokuments d_j im Bezug auf

eine Anfrage q wird standardmäßig der Kosinus des Winkels zwischen den Vektorabbildungen bestimmt [JM08]. Gleichung 2.1 wurde aus [JM08] adaptiert und zeigt die Formel zur Berechnung der Kosinus-Ähnlichkeit.

$$cosineSimilarity(\vec{q}, \vec{d_j}) = \frac{\sum_{i=1}^{N} w_{i,q} * w_{i,j}}{\sqrt{\sum_{i=1}^{N} w_{i,q}^2} * \sqrt{\sum_{i=1}^{N} w_{i,j}^2}} \tag{2.1}$$

Das Ergebnis liegt zwischen 0 und 1. Eine vollständige Übereinstimmung des Anfragevektors mit einem Dokumentvektor führt zum Ergebnis 1. Ein Ergebnis von 0 liegt vor, wenn eine Anfrage und ein Dokument keine gemeinsamen Terme besitzen. Beispiel 2.1 veranschaulicht das beschriebene Vorgehen.

Beispiel 2.1: Vorgehen beim Vektorraummodell

Angenommen, das Vokabular der Dokumentensammlung besteht aus den Wörtern *Obst*, *Gemüse* und *Essen* und die Relevanz eines Wortes entspricht der Häufigkeit, mit der ein Wort in einem Dokument auftritt. Ein Dokument d_i, welches das Wort *Obst* dreimal, *Essen* zweimal und sonst keine Wörter enthält, kann folglich durch den Vektor $\vec{d_i} = (3, 0, 2)$ repräsentiert werden. Ein anderes Dokument d_k umfasst die Wörter *Gemüse* fünfmal und *Essen* dreimal. Es wird durch den Vektor $\vec{d_k} = (0, 5, 3)$ repräsentiert.

Eine exemplarische Anfrage mit dem Wort *Obst* kann durch den Vektor $\vec{q} = (1, 0, 0)$ dargestellt werden. Offensichtlich zielt die Anfrage auf d_i ab. Die berechneten Ähnlichkeitswerte $sim(\vec{q}, \vec{d_i}) \approx 0.83$ und $sim(\vec{q}, \vec{d_k}) = 0$ bestätigen diese Intuition. Im Beispiel würde daher d_k als relevantestes Dokument gekennzeichnet werden.

Gewichtung von Termen

Bei der Bewertung von Wörtern auf der Basis ihrer Vorkommenshäufigkeit (engl. term frequency), kurz TF, wird die Relevanz eines Wortes für ein Dokument höher gewichtet, je häufiger es im Dokument vorkommt. Ergänzend zu dieser Bewertungsstrategie kann die inverse Dokumenthäufigkeit (engl. inverse document frequency), kurz IDF, zur Bewertung der Relevanz von Wörtern genutzt werden. Hierbei wird ein Wort höher gewichtet, wenn es in wenigen Dokumenten der Dokumentensammlung auftritt. Gleichung 2.2 aus [MRS08] zeigt die Formel zur Berechnung des IDF-Wertes eines Terms t. Die Anzahl aller Dokumente in der Dokumentensammlung entspricht N und der Wert n_t beschreibt die Anzahl der Dokumente, die den Term t beinhalten. Die inverse Dokumenthäufigkeit erreicht ihren niedrigsten Wert, wenn ein Wort in allen Dokumenten vorkommt und ihren höchsten Wert, wenn lediglich ein Dokument den Term enthält.

$$idf_t = log \frac{N}{n_t} \tag{2.2}$$

Die Kombination beider Gewichtungen wird TF-IDF genannt und ist bei der Berechnung der Termgewichte weit verbreitet [JM08]. Das Gewicht eines Terms t für ein Dokument d errechnet sich durch Multiplikation des TF-Wertes mit dem IDF-Wert. Das TF-IDF-Maß bewertet folglich Terme, die in einem Dokument häufig auftreten und gleichzeitig nur selten in anderen Dokumenten vorkommen, am höchsten.

2.3.4 Worteinbettung

Unter dem Begriff der Worteinbettung sind Verfahren zusammengefasst, mit deren Hilfe Wörter auf reellwertige Vektoren abgebildet werden können. Die Vektorabbildung repräsentiert hierbei den Nutzungskontext eines Wortes. Worteinbettungen basieren auf der

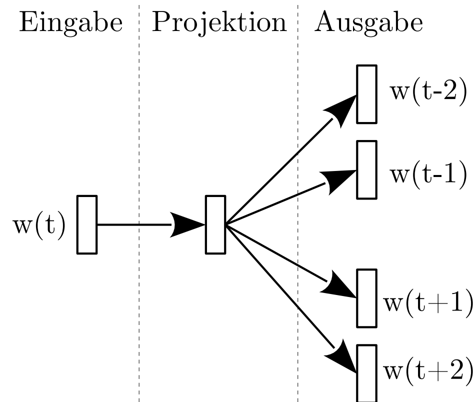


Abbildung 2.4: Schematische Darstellung der CSG-Architektur. Mit *w(t)* wird das Wort an der Stelle t im Eingabetext beschrieben. Quelle: [MCCD13]

Annahme, dass Wörter, die in ähnlichen Kontexten verwendet werden, eine ähnliche Bedeutung besitzen [Har54]. Der Kontext eines Wortes wird durch seine lokale Umgebung, also den Wörtern, die in der unmittelbaren Nähe eines Wortes im Text auftreten, beschrieben. Wörter, die in ähnlichen Kontexten verwendetet werden, liegen im Vektorraum nahe beieinander. Mit Hilfe der Wortvektoren lassen sich Zusammenhänge zwischen Wörtern, z.B. über die Kosinus-Ähnlichkeit, berechnen.
Weit verbreitet für das Lernen geeigneter Wortvektoren ist bspw. das *Continious Skip-Gram* (CSG) Modell aus [MCCD13]. Hierbei handelt es sich um ein neuronales Netz [Alp19], welches auf großen, nicht annotierten Textmengen trainiert wird. Das Modell besitzt eine Eingabe-, eine Projektions- und eine Ausgabeschicht. Beim Training wird der Trainingskorpus über ein gleitendes Fenster verarbeitet, welches die Größe des Kontextes festlegt, die als Wortumgebung betrachtet wird. Abbildung 2.4 zeigt die abstrakte Architektur des Modells. Das Ziel des CSG-Modells ist es, die Umgebung eines Wortes vorherzusagen. Bei einer Fensterlänge von fünf dient bspw. das mittlere Wort als Eingabe. Die Ausgabe des Modells liefert eine Vorhersage der beiden Wörter links und rechts neben dem Eingabewort. Beim Training des Worteinbettungsmodells wird eine Abbildungsfunktion gelernt, welche den Vorhersagefehler auf dem Trainingskorpus minimiert. Diese Abbildung kann anschließend genutzt werden, um Wörter auf Vektoren abzubilden.

Neuere Arbeiten wie [BGJM17] von Bojanowski et al. lernen N-Gramm Vektorrepräsentationen auf Basis der Zeichenketten im Trainingskorpus. Für die Berechnung eines Wortvektors wird die Summe der N-Gramm Vektoren gebildet. Um den Anfang und das Ende eines Wortes zu berücksichtigen werden Wörtern vorab die Sonderzeichen < bzw. > zur Markierung hinzugefügt. Das Wort *bestResult* wird bspw. durch die Zeichenkette <*bestResult*> repräsentiert. Für N=3 werden die 3-Gramme <*be*, *bes*, *est*, *stR*, *tRe*, *Res*, *esu*, *sul*, *ult* und *lt*> sowie die komplette Zeichenkette <*bestResult*> betrachtet, um den Wortvektor zu berechnen. Durch die Arbeit mit N-Grammen ist das Worteinbettungsmodell in der Lage, auch für unbekannte Wörter Vektoren zu erzeugen, solange die Vektoren der enthaltenen N-Gramme bekannt sind.

2.3.5 Zeichenkettenvergleich

Mit Hilfe von Zeichenkettenvergleichen können Ähnlichkeiten zwischen Zeichenketten berechnet werden. Ein bekanntes Distanzmaß ist die Levenshtein-Distanz [Lev66]. Sie berechnet die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um eine Zeichenkette in eine andere zu überführen. Für die beiden Zeichenketten *rate* und *tare* ist die Levenshtein-Distanz gleich zwei, da die Zeichen r und t in der Zeichenkette *rate* durch die Zeichen t und r ersetzt werden müssen, um die Zeichenkette *tare* zu erhalten.

Ein weiteres bekanntes Ähnlichkeitsmaß ist die Jaro-Winkler-Ähnlichkeit [Win90]. Sie basiert im Gegensatz zur Levenshtein-Distanz auf der Anzahl gemeinsamer Zeichen in zwei Zeichenketten. Ein Zeichen, dass in beiden Zeichenketten auftritt, wird bei der Berechnung berücksichtigt, wenn die Positionen der beiden Zeichen nicht weiter als $\lfloor\frac{max(|s1|,|s2|)}{2}\rfloor - 1$ voneinander entfernt liegt. Es wird zwischen gemeinsamen Zeichen, die an der gleichen Position stehen und zwischen gemeinsamen Zeichen, die in beiden Zeichenketten an verschiedenen Position auftreten differenziert. Gleichung 2.3 zeigt den ersten Berechnungsschritt der Jaro-Winkler-Ähnlichkeit aus [Win90]. Die beiden Eingabezeichenketten werden s_1 und s_2 genannt. Die Anzahl gemeinsamer Zeichen an der gleichen Position wird durch m beschrieben. Die Zahl t entspricht der Anzahl gemeinsamer Zeichen an unterschiedlichen Positionen geteilt durch zwei. Für $m = 0$ ist das Ergebnis gleich 0. Bei identischen Zeichenketten ist das Ergebnis gleich 1.

$$sim(s_1, s_2) = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) \tag{2.3}$$

Für die beiden Zeichenketten *rate* und *tare* ist $m = 2$, da die Zeichen a und e an den gleichen Positionen auftreten. Die Zeichen r und t kommen ebenfalls in beiden Zeichenketten vor, allerdings an verschiedenen Positionen. Da ihre Positionen weiter als $\frac{4}{2} - 1 = 1$ voneinander entfernt liegt, werden sie nicht berücksichtigt. Demnach ist t gleich 0. Das Ergebnis von Gleichung 2.4 für die beiden Zeichenketten ist $\frac{1}{3}(\frac{2}{4} + \frac{2}{4} + \frac{2}{2}) \approx 0,66$.
Das Endergebnis der Jaro-Winkler-Ähnlichkeit wird über Gleichung 2.4 aus [Win90] berechnet. Hierbei wird der Ähnlichkeitswert aus Gleichung 2.4 vergrößert, wenn beide Zeichenketten ein gemeinsames Präfix besitzen. Die Länge des gemeinsamen Präfix wird durch l angegeben.

$$jw(s_1, s_2) = sim(s_1, s_2) + l * 0.1 * (1 - sim(s_1, s_2)) \tag{2.4}$$

Für die Zeichenketten *rate* und *tare* ist das Ergebnis von Gleichung 2.4 mit dem Ergebnis von Gleichung 2.3 identisch, da kein gemeinsames Präfix vorliegt.

Der FuzzyScore[5] weist einige Analogien zur Jaro-Winkler-Ähnlichkeit auf, da er ebenfalls auf der Anzahl gemeinsamer Zeichen basiert. Allerdings spielt es beim FuzzyScore keine Rolle, ob ein gemeinsames Zeichen an der gleichen Position innerhalb der Zeichenketten auftritt oder nicht. Einzig die Reihenfolge der Zeichen wird berücksichtigt. Darüber hinaus bewertet der FuzzyScore die Ähnlichkeit zweier Zeichenketten höher, wenn die gemeinsamen Zeichen an aufeinanderfolgenden Stellen in den Zeichenketten vorkommen.
Bei der Berechnung des FuzzyScore für zwei Zeichenketten s_1 und s_2 wird der Ähnlichkeitswert zunächst mit 0 initialisiert. Für jedes Zeichen in s_1 wird eine Übereinstimmung in s_2 gesucht. Beide Zeichenketten werden genau einmal durchlaufen. Für ein Zeichen in s_1, welches auch in s_2 vorhanden ist, wird der Ähnlichkeitswert um eins erhöht. Bei aufeinanderfolgenden Übereinstimmungen von Zeichen in s_1 und s_2, wird das Ergebnis zusätzlich um zwei erhöht. Der maximale FuzzyScore wird bei einer vollständigen Übereinstimmung zweier Zeichenketten erreicht. Jedes Zeichen, mit Ausnahme des ersten Zeichens, führt hierbei zu einer Erhöhung des Ähnlichkeitswerts um drei. Das Gesamtergebnis liegt im Bereich zwischen 0 und $(3 * max(s_1, s_2)) - 2$, wobei max die Anzahl der Zeichen in der Längeren der beiden Zeichenketten zurückliefert.
Für die bereits zuvor betrachteten Zeichenketten $s_1 = rate$ und $s_2 = tare$ berechnet sich das Ergebnis wie folgt. Zu Beginn wird das erste Zeichen aus s_1 in s_2 gesucht. Aufgrund der Übereinstimmung an Position drei in s_2 wird das Ergebnis um 1 erhöht. Anschließend wird das zweite Zeichen aus s_1 in s_2 gesucht. Insgesamt wird s_1 nur einmal durchlaufen, deshalb startet die Suche an der Stelle vier in s_2. Da es sich hierbei um das letzte Zeichen in s_2 handelt und keine Übereinstimmung vorliegt, ist das Gesamtergebnis gleich 1.

[5] `https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/FuzzyScore.html`, zuletzt besucht am 16.12.2019

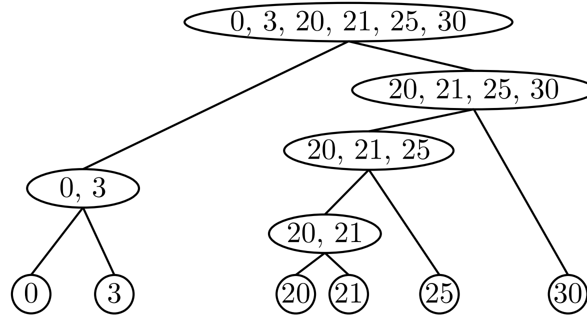


Abbildung 2.5: Beispiel für eine Clusterhierarchie.

2.4 Hierarchische Clusteranalyse

Bei einer Clusteranalyse, teilweise auch Ballungsanalyse genannt, wird nach Ähnlichkeitsstrukturen in Datenmengen gesucht. Datenobjekte, die im Bezug auf ein konkretes Merkmal eine hohe Ähnlichkeit zueinander aufweisen, werden hierbei in Cluster zusammengefasst. Die Berechnung der Ähnlichkeit zwischen den Datenobjekten ist frei wählbar. Handelt es sich bei den Datenobjekten um Punkte im Raum, wäre bspw. die euklidische Distanz für die Bewertung der Ähnlichkeit denkbar.

Unter dem Begriff der hierarchischen Clusteranalyse werden Clusteranalysen zusammengefasst, die eine Clusterhierarchie erzeugen. Im Allgemeinen wird zwischen der agglomerativen und der divisiven hierarchischen Clusteranalyse unterschieden [Alp19]. Die agglomerative Clusteranalyse arbeitet von unten nach oben (engl. bottom-up). Alle Datenobjekte werden zunächst als einzelne Cluster aufgefasst. Anschließend werden iterativ die beiden Cluster mit der größten Ähnlichkeit vereint. Dies wird solange wiederholt, bis ein Cluster entstanden ist, welches alle Datenobjekte enthält.
Die divisive Clusteranalyse arbeitet von oben nach unten (engl. top-down) und bildet somit das Gegenstück zum agglomerativen Vorgehen. Am Anfang der divisiven Clusteranalyse existiert lediglich ein Cluster, welches alle Datenobjekte umfasst. Anschließend wird iterativ eines der bestehenden Cluster in zwei kleinere Cluster aufgeteilt, bis am Ende alle Datenobjekte in einem eigenen Cluster liegen.
Abbildung 2.5 zeigt ein Beispiel für eine Clusterhierarchie. Insgesamt existieren sechs Datenobjekte, die jeweils eine Zahl repräsentieren. Die Ähnlichkeit der Datenobjekte entspricht der absoluten Differenz ihrer repräsentierten Zahlen. Eine hohe Ähnlichkeit ist gleichbedeutend mit einer geringen Differenz. Durch das iterative Vereinigen oder Auftrennen von Clustern entsteht die für die hierarchische Clusteranalyse typische Baumstruktur. Die Knoten des Baums repräsentieren die gefundenen Cluster. Je näher ein Cluster bei der Wurzel der Hierarchie liegt, desto geringer ist die durchschnittliche Ähnlichkeit der enthaltenen Elemente.

Bei der agglomerativen Clusteranalyse existieren verschiedene Möglichkeiten, um zu entscheiden, welche zwei Cluster in einem Iterationsschritt vereint werden. Einige grundlegende Methoden werden im Folgenden kurz vorgestellt. Das gemeinsame Fundament aller Vorgehensweisen bildet eine Funktion $sim(d_i, d_j)$, welche die Ähnlichkeit zweier Datenobjekte d_i und d_j berechnet.
Die Single-Linkage-Methode [Alp19] nutzt die maximale Ähnlichkeit zwischen den Datenobjekten zweier Cluster, um deren Zusammenhang zu bewerten. Für zwei Cluster C_i und C_j entspricht der Zusammenhang dem Ergebnis aus $max_{x \in C_i, y \in C_j}(sim(x, y))$. Die Complete-Linkage-Methode [Alp19] ist hierzu ähnlich, nutzt jedoch die minimale Ähnlichkeit zwischen den Datenobjekten zweier Cluster für die Bewertung des Zusammenhangs.
Eine weitere Vorgehensweise, welche alle Datenobjekte zweier Cluster in die Bewertung

des Zusammenhangs einbezieht, ist die Group-Average-Methode [MRS08]. Gleichung 2.5 aus [MRS08] zeigt die Formel für die Berechnung des Zusammenhangs zweier Cluster C_i und C_j.

$$linkage(C_i, C_j) = \frac{1}{(N_i + N_j) * (N_i + N_j - 1)} \sum_{d_1 \in C_i \cup C_j} \sum_{d_2 \in C_i \cup C_j, d_1 \neq d_2} sim(d_1, d_2) \quad (2.5)$$

Hierbei wird die durchschnittliche Ähnlichkeit zwischen allen Datenobjektpaaren aus der Vereinigung der Datenobjekte in C_i und C_j berechnet. Mit N_i bzw. N_j ist die Anzahl der Elemente im Cluster C_i bzw. C_j beschrieben.

3 Rückverfolgung von Anforderungen mit INDIRECT

Das Ziel des Forschungsprojekts INDIRECT (*Intent-driven Requirements-to-Code Traceability*) [Hey19] von Hey ist eine automatisierte Bereitstellung von Rückverfolgbarkeitsinformationen zwischen Anforderungen und deren Entsprechung im Quelltext. Der Ansatz von INDIRECT basiert auf dem Vergleich der Absichten hinter den Anforderungsdokumenten und den Quelltextartefakten. Abbildung 3.1 zeigt eine schematische Darstellung der Vorgehensweise. Sowohl für die Anforderungen als auch für die Quelltextartefakte wird eine semantische Zwischenrepräsentation, im folgenden Absichtsmodell genannt, aufgebaut. Ein Vergleich der beiden Absichtsmodelle soll daraufhin Rückschlüsse auf die Zusammenhänge zwischen den Anforderungen und dem Quelltext erlauben. Durch die Verlagerung der Suche nach Zusammenhängen von der Syntax- auf die Semantikebene, ist INDIRECT unabhängiger gegenüber dem verwendeten Vokabular in den Softwareartefakten.

Die Implementierung von INDIRECT erfolgt auf der Basis der Rahmenarchitektur *Programming ARchitecture for Spoken Explanations* (PARSE) aus [WT15]. Die Verarbeitung in PARSE basiert auf Agenten, die unabhängig voneinander auf einem zentralen Graphen arbeiten. Der Graph dient hierbei als gemeinsamer Datenspeicher und wird zu Beginn aus der Eingabe aufgebaut. Ein Agent implementiert einen Verarbeitungsschritt und hat die

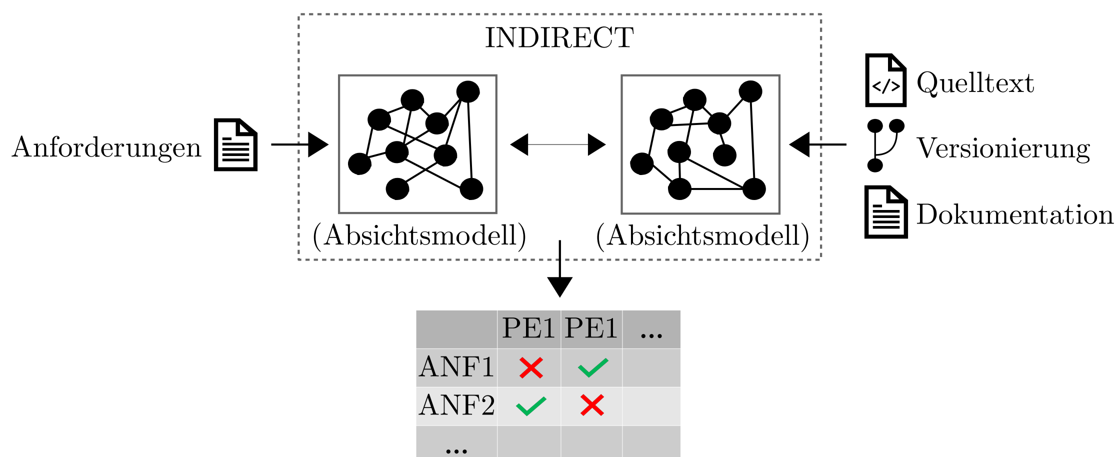


Abbildung 3.1: Schematische Darstellung des Verarbeitungsprozesses von INDIRECT. Die Abkürzung *ANF* steht für Anforderung und *PE* für Programmelement.

Möglichkeit, bestehende Informationen aus dem Graphen auszulesen bzw. neue Informationen einzufügen. Die Agenten werden iterativ ausgeführt, sodass sie auf neue Informationen im Graphen reagieren können.

Das Absichtsmodell der Anforderungsbeschreibungen beschreibt die semantischen Beziehungen zwischen den Anforderungen. Hierbei werden die Konzepte, Entitäten und Aussagen in den Beschreibungen untersucht. Die Analyse der natürlichen Sprache umfasst unter anderem das seichte Zerteilen, bei dem Wortarten markiert und anschließend grammatikalisch zusammenhängende Bedeutungseinheiten wie Nominal- oder Verbphrasen gebildet werden. Darüber hinaus werden semantische Analysen zur Auflösung von Mehrdeutigkeiten und dem Aufbau von Konzepthierarchien durchgeführt und die Anforderungsbeschreibungen mit Welt- und Domänenwissen angereichert. Die einzelnen Verarbeitungsschritte und Analysen sind als PARSE-Agenten umgesetzt. Das Resultat der sprachlichen Analyse ist ein Graph, der die Konzepte, Entitäten und Aussagen in den Anforderungsdokumenten, inklusive ihrer semantischen Beziehungen zueinander, repräsentiert.

Das Absichtsmodell des Quelltextes beschreibt die semantischen Zusammenhänge zwischen Programmstellen. Hierbei soll der Quelltext und weitere Artefakte wie Testfälle und die Dokumentation oder Versionsverwaltungsdaten analysiert werden. Das Ziel ist es, Programmelemente, die derselben Absicht dienen, zu identifizieren. Hierzu werden bspw. syntaktische Abhängigkeiten im Quelltext und die Semantik der natürlichen Sprache in den verschiedenen Artefakten betrachtet. Die Analysen werden, genau wie bei der Verarbeitung der Anforderungen, durch verschiedene Agenten implementiert.

Die beiden aufgebauten Absichtsmodelle werden in einem abschließenden Schritt miteinander verglichen. Hierbei werden semantische oder strukturelle Analogien in den Graphen gesucht. Gefundene Zusammenhänge zwischen den Absichtsmodellen werden daraufhin auf die ursprünglichen Artefakte zurückgeführt und bspw. in einer Korrelationsmatrix dargestellt.

4 Verwandte Arbeiten

Das folgende Kapitel beschreibt existierende Arbeiten aus verschiedenen Forschungsgebieten, deren Zielsetzung oder Herangehensweise relevant für die vorliegende Arbeit ist.

4.1 Repräsentation von Quelltext

Die effiziente Darstellung von Syntax und Semantik ist eine der zentralen Herausforderungen bei der Verarbeitung von Quelltext [ZWZ+19]. Häufig wird hierfür eine graphbasierte Repräsentation gewählt. Die Knoten repräsentieren Programmelemente wie Klassen [KGH+19, MAK+12], Methoden [SM11] oder eine Kombination unterschiedlicher Quelltextfragmente [RM02]. Kanten drücken Zusammenhänge aus, die bspw. durch eine Analyse der statischen Quelltextabhängigkeiten wie in [Rob08, GAL18] identifiziert werden.

Eine graphische Quelltextrepräsentation, die ausschließlich auf den textuellen Bestandteilen im Quelltext basiert, stellen Liu et al. in ihrer Arbeit **PFN: A novel program feature network for program comprehension** [LSLZ14] vor. Ihr Ziel ist es, die semantischen Merkmale einer Klasse und die Zusammenhänge zwischen Klassen effektiv in einem Graphen darzustellen, um Entwicklern das Programmverständnis zu erleichtern. Die Knoten des Graphen repräsentieren die Klassen im Quelltext. Aus jeder Klasse werden der Name, die Kommentare und die verwendeten Bezeichner extrahiert und als Attribute im Knoten gespeichert. Hierbei werden durch einen Unterstrich oder die Binnenmajuskel-Schreibweise zusammengesetzte Wörter getrennt und Stoppwörter wie *the*, *it* oder *on* entfernt. Darüber hinaus entfernen sie Wörter, die in mehr als 80% oder weniger als 2% der Klassen auftreten, da diese Wörter nur wenig Einfluss auf die Kookkurrenz-Statistik besitzen. Auf der Basis der extrahierten Wörter berechnen sie anschließend mit Hilfe des Relational Topic Model [CB09] die Zusammenhänge zwischen den Klassen. Klassen, die denselben Themen zugeordnet wurden, werden mit einer ungerichteten Kante verknüpft.
In ihrer abschließenden Diskussion vergleichen sie ihren Graphen, im Folgenden PFN genannt, qualitativ mit dem Klassen-Abhängigkeitsgraphen (engl. class dependency graph). Der Klassen-Abhängigkeitsgraph, im Folgenden mit CDG bezeichnet, besitzt ebenfalls die Klassen im Quelltext als Knoten und beschreibt mit seinen Kanten Abhängigkeiten wie Erweiterungen oder Assoziationen. Da die Repräsentation von Liu et al. lediglich einen Kantentyp besitzt, der zudem ungerichtet ist, sind sie der Meinung, dass die Komplexität im Vergleich zum Klassen-Abhängigkeitsgraphen geringer ist. Darüber hinaus beschreibt der Graph von Liu et al. semantische Zusammenhänge, weshalb es für das Verständnis einer Klasse genügt, seine benachbarten Knoten zu betrachten. Im Gegensatz dazu stellt

der Klassen-Abhängigkeitsgraph lediglich logische Abhängigkeiten dar, sodass Zusammenhänge manuell abgeleitet werden müssen.

Einen anderen Ansatz zur Repräsentation von Quelltext stellen Zhang et al. in ihrer Arbeit **A Novel Neural Source Code Representation Based on Abstract Syntax Tree** [ZWZ+19] vor, in der sie Quelltext in eine Vektorrepräsentation überführen. Im ersten Schritt wird der abstrakte Syntaxbaum eines Quelltextfragments herangezogen und entsprechend der darin enthaltenen Anweisungen in Teilbäume aufgeteilt, sodass jede Anweisung durch einen Teilbaum repräsentiert wird. Im nächsten Schritt werden die Teilbäume von einem neuronalen Netz auf Vektoren abgebildet. Die Sequenz der Vektoren wird im letzten Schritt von einem weiteren neuronalen Netz in einen finalen Vektor überführt.
In der Evaluation nutzen sie ihre Repräsentation zum einen, um Quelltext hinsichtlich seiner Funktionalität zu klassifizieren und zum anderen, um Quelltextduplikate zu identifizieren. Ihr Modell war in der Lage, Quelltextfragmente mit einer Genauigkeit von 98,2% zu einer von 104 Kategorien zuzuordnen. Bei der Identifikation von Quelltextduplikaten in zwei Datensätzen wurde ein F1-Maß [Tin16] von 95,5% bzw. 93,8% erzielt.

Tufano et al. differenzieren in ihrer Arbeit **Deep Learning Similarities from Different Representations of Source Code** [TWB+18] zwischen Repräsentationen, die auf dem AST, Kontrollfluss, Bytecode und den lexikalischen Bestandteilen basieren. Bei der Repräsentation auf der Basis des AST wird ein Programmelement durch die Menge an Knotentypen des korrespondierenden AST dargestellt. Knoten im AST, die Bezeichner oder Konstanten beschreiben, werden hierbei ignoriert, da sie für die lexikalische Repräsentation von Programmelementen verwendet werden. Die Grundlage für die lexikalische Repräsentation bilden die Bezeichner und Konstanten im Quelltext, wobei konstante Werte durch ihren Datentyp ersetzt werden. Die Bytecoderepräsentation entspricht dem Bytecode des jeweiligen Programmelements. Der Kontrollfluss wird durch den Kontrollflussgraphen beschrieben. Für jede Klasse und Methode im Quelltext werden diese Repräsentationen erzeugt. Anschließend wird jede Repräsentation in eine Vektordarstellung überführt. Hierfür kommen beim abstrakten Syntaxbaum, dem Bytecode und der lexikalischen Repräsentation neuronale Netze zum Einsatz. Der Kontrollflussgraph wird über eine Grapheinbettungstechnik [OCP+16] in einen Vektor überführt. Zur Bewertung der Ähnlichkeit zweier Programmelemente wird die durchschnittliche Distanz zwischen den Vektorrepräsentationen der Programmelemente herangezogen.
Für den Vergleich der verschiedenen Repräsentationen, werden in der Evaluation Quelltextduplikate auf Methoden- und Klassenebene in 10 Java Projekten gesucht. Die Repräsentation auf Grundlage der lexikalischen Bestandteile führt hierbei zur besten Präzision (100%) bei einer Ausbeute von 40% auf Klassenebene und 52% auf Methodenebene. Die beste Balance zwischen Präzision (91% auf Klassenebene und 96% auf Methodenebene) und Ausbeute (63% auf Klassenebene und 75% auf Methodenebene) kann mit einer Repräsentation, welche auf dem abstrakten Syntaxbaum basiert, erreicht werden. Ein kombinierter Ansatz, der alle Repräsentationen verwendet, erreichte ein F1-Maß von 85% auf Klassenebene und 96% auf Methodenebene. Insgesamt kommen Tufano et al. zu dem Schluss, dass die verschiedenen Repräsentationen komplementäre Informationen liefern und deren Kombination daher sinnvoll ist.

Efstathiou und Spinellis stellen in ihrer Arbeit **Semantic Source Code Models Using Identifier Embeddings** [ES19] einen Ansatz zur Nutzung von Worteinbettungen (Siehe Abschnitt 2.3.4) für die Repräsentation von Quelltext vor. Für sechs verschiedene Programmiersprachen trainieren sie jeweils ein Worteinbettungsmodell mit Hilfe von fastText aus [BGJM17]. Die Basis des Trainingskorpus sind 13.144 GitHub[1]-Softwaredepots. Für jede der betrachteten Programmiersprachen extrahieren sie den Text aus den, in der jeweili-

[1] `https://github.com/`, zuletzt besucht am 04.01.2020

gen Programmiersprache geschriebenen, Softwareprojekten. Vor dem Training der Worteinbettungsmodelle werden die extrahierten Textkorpora vorverarbeitet. Hierbei werden Schlüsselwörter der jeweiligen Programmiersprache, Satzzeichen und Wörter, bestehend aus lediglich einem Zeichen, sowie Zahlen entfernt.
Bei der Evaluation bewerten Efstathiou und Spinellis die Ähnlichkeit zwischen Softwaredepots. Hierfür bilden sie die vorverarbeitete Textmenge der Softwareprojekte Log4j[2], SLF4j[3] und Spatial4j[4], mit Hilfe ihres trainierten Java-Worteinbettungsmodells, auf Vektoren ab. Anschließend berechnen sie die Distanz zwischen den Vektoren über die Word-Movers-Distance [KSKW15]. Je niedriger die Distanz zweier Vektoren ist, desto ähnlicher ist ihre Semantik. Die Distanz der Softwareprojekte Log4j und SLF4j, die eine ähnliche Semantik besitzen, beträgt 0,59. Die Distanz der beiden Projekte zum Softwareprojekt Spatial4j, welches in einer anderen Domäne liegt, beträgt 2,39 bzw. 1,99.

4.2 Konzeptlokalisierung

Neben einer geeigneten Repräsentation von Quelltext ist das Ziel der vorliegenden Arbeit, die Extraktion der im Quelltext enthaltenen semantischen Zusammenhänge. Hierzu existiert eine Vielzahl von Arbeiten in der Literatur, die sich im Hinblick auf die verwendeten Merkmale in syntax-, semantikbasierte oder kombinierte Ansätze unterteilen lassen.

4.2.1 Syntaxbasierte Quelltextanalyse

In ihrer Arbeit **Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies** [RM02] beschäftigen sich Robillard und Murphy bereits 2002 mit einer Repräsentation von Quelltext, die den Zugriff auf zusammenhängende Programmelemente erlaubt. Hierzu wird ein Graphmodell aufgebaut, das den Quelltext kompakt und einfach beschreibt. Klassen, Attribute und Methoden bilden die Knoten, Kanten repräsentieren strukturelle Zusammenhänge wie Methodenaufrufe, Lese- und Schreibzugriffe, Definitionen, Instanziierungen und Überprüfungen. Eingebettet in ein Werkzeug unterstützt der Graph Anwender bei der Navigation durch den Quelltext. Ausgehend von einer Programmstelle, z.B. der Hauptmethode, zeigt er dem Anwender weitere relevante Quelltextstellen an. Um diese automatisiert priorisieren zu können, führt Robillard in einer späteren Arbeit **Topology analysis of software dependencies** [Rob08] das Konzept der Spezifität (engl. specificity) und Verstärkung (engl. reinforcement) ein. Ausgangspunkt für die Betrachtung bildet eine vorgegebene Menge an relevanten Quelltextstellen. Im Hinblick auf die Spezifität wird bspw. ein Programmelement, welches exklusiv auf eine relevante Stelle referenziert, höher priorisiert, als ein Programmelement, welches zusätzlich noch auf weitere Stellen verweist. Beim Konzept der Verstärkung wird ein Quelltextfragment mit vielen Abhängigkeiten zu relevanten Quelltextstellen gegenüber solchen bevorzugt, die im Verhältnis weniger Abhängigkeiten zu relevanten Programmstellen aufweisen.
Für die Evaluation ihres Ansatzes untersuchen sie die, von ihrem Werkzeug gefundenen, Programmelemente mit und ohne Priorisierung in zwei Fallstudien. Bei der ersten Fallstudie werden insgesamt 31 relevante Quelltextstellen von ihrem Werkzeug identifiziert, von denen 19% tatsächlich relevant sind. Durch die Priorisierung kann der Anteil relevanter Stellen unter den ersten neun Vorschlägen auf 56% angehoben werden. Der gleiche Effekt zeigte sich in der zweiten Fallstudie, bei der die insgesamt 54 vorgeschlagenen Quelltextstellen 50% relevante Stellen beinhalten. Der Anteil relevanter Stellen unter den ersten 13 Vorschlägen kann durch die Priorisierung auf 62% angehoben werden.

[2] `https://github.com/apache/log4j`, zuletzt besucht am 04.01.2020
[3] `https://github.com/qos-ch/slf4j`, zuletzt besucht am 04.01.2020
[4] `https://github.com/locationtech/spatial4j`, zuletzt besucht am 04.01.2020

Gharibi et al. verwenden in ihrer Arbeit **Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension** [GAL18] für die Unterteilung des Quelltextes ausschließlich syntaktische Merkmale. Hierfür wird ein Graph aufgebaut, dessen Knoten die Methoden und dessen Kanten die Methodenaufrufe im Quelltext widerspiegeln. Mit Hilfe dieses Aufrufgraphen analysieren sie alle möglichen Ausführungspfade und versuchen ähnliche Pfade über eine hierarchische Clusteranalyse (Siehe Abschnitt 2.4) zu gruppieren. Ein Ausführungspfad besteht aus einer Sequenz von Knoten im Graphen, dessen erster Knoten keine eingehenden und dessen letzter Knoten keine ausgehenden Kanten besitzt. Die Ähnlichkeit zwischen zwei Ausführungspfaden berechnen sie über das Verhältnis zwischen der Anzahl gemeinsamer und der Anzahl aller Knoten zweier Pfade. Am Ende weisen Gharibi et al. den gefundenen Clustern jeweils eine Menge von Wörter als Beschreibung zu. Hierzu wird jedes Cluster als Dokument, bestehend aus der Menge der Namen der darin platzierten Methoden, betrachtet. Mit Hilfe des TF-IDF-Maß (Siehe Abschnitt 2.3.3) werden die Methodennamen eines Clusters sortiert und die fünf am stärksten gewichteten Namen zur Beschreibung des Clusters verwendet. Bei einer qualitativen Evaluation testen sie ihren Ansatz in vier Softwareprojekten mit dem Ergebnis, dass die hierarchische Unterteilung des Aufrufgraphen, in Verbindung mit einer Beschreibung der gefundenen Cluster, hilfreich für den Aufbau eines Programmverständnisses ist. Allerdings wächst die Anzahl der Pfade mit der Größe des analysierten Softwareprojekts stark an, wodurch der Graph schnell unübersichtlich wird.

4.2.2 Lexikalische Quelltextanalyse

Alternativ zur Struktur lassen sich die lexikalischen Bestandteile im Quelltext analysieren. Ein Ansatz hierfür zeigen Kuhn et al. in ihrer Arbeit **Semantic clustering: Identifying topics in source code** [KDG07]. Sie verwenden Kommentare und Bezeichner, um die lexikalische Ähnlichkeit von Klassen zu bestimmen. Eine Klasse wird auf eine Wortmenge, auch Klassendokument genannt, abgebildet, die aus den Wörtern in den Kommentaren und Bezeichnern der Klasse besteht. Darüber hinaus berücksichtigen sie die Vererbungsstruktur, indem abgeleitete Klassen mit den Wörtern der Oberklasse angereichert werden. Bei der anschließenden Vorverarbeitung der Wortmengen werden Stoppwörter entfernt, eine Stammformreduktion (Siehe Abschnitt 2.3.1) durchgeführt und zusammengesetzte Bezeichner aufgetrennt. Auf der Basis der Klassendokumente werden anschließend die Ähnlichkeiten zwischen den Klassen über Latent Semantic Indexing [MRS08] (LSI) berechnet. LSI ist eine Abwandlung des Vektorraummodells aus Abschnitt 2.3.3. Die Term-Dokument-Matrix wird über das TF-IDF-Maß gewichtet. Als Ähnlichkeitsmaß dient die Kosinus-Ähnlichkeit der Vektorabbildungen der Klassendokumente. Mit Hilfe einer hierarchischen Clusteranalyse wird abschließend eine passende Unterteilung der Klassen in semantisch zusammenhängende Partitionen gesucht.
Die Nützlichkeit ihres Ansatzes analysieren Kuhn et al. qualitativ in zwei Fallstudien. Hierbei untersuchen sie manuell die Plausibilität der Clusteringergebnisse ihres Ansatzes. Sie kommen zu dem Schluss, dass die semantische Unterteilung der Klassen sinnvolle Ergebnisse liefert und Entwicklern dabei helfen kann, relevante Quelltextstellen im Bezug auf ein Anliegen zu finden. Darüber hinaus geben Kuhn et al. an, dass die Ergebnisse stark von der Qualität der Kommentare und Bezeichner abhängen. Werden bspw. viele generische Namen oder kryptische Abkürzungen verwendet, hat dies einen negativen Einfluss auf die Aussagekraft der Ergebnisse.

Einen anderen Ansatz zur Nutzung der lexikalischen Bestandteile zeigen Mahmoud und Bradshaw in ihrer Arbeit **Semantic topic models for source code analysis** [MB17]. Sie extrahieren Bezeichner, Kommentare und statische Zeichenketten aus dem Quelltext und bauen damit ein quelltextspezifisches Vokabular auf. Bei der Vorverarbeitung werden zusammengesetzte Bezeichner aufgetrennt, Schlüsselwörter und Stoppwörter entfernt und

eine Stammformreduktion durchgeführt. Zu Beginn berechnen Mahmoud und Bradshaw die semantische Ähnlichkeit zwischen einzelnen Termen im Vokabular über die Normalized Google Distance (NGD) [CV07]. Auf der Basis der Ähnlichkeiten führen sie anschließend eine hierarchische Clusteranalyse durch, um semantisch zusammenhängende Terme zu identifizieren. Innerhalb eines Clusters werden die Terme entsprechend ihrer durchschnittlichen Ähnlichkeit zu allen anderen Termen im Cluster gewichtet. Hierbei wird als Ähnlichkeitsmaß ebenfalls NGD herangezogen. Im letzten Schritt werden Ähnlichkeiten zwischen den Klassen aus dem Quelltext und den gefundenen Clustern berechnet. Die Ähnlichkeiten der Terme einer Klasse mit den Termen eines Clusters werden hierzu gewichtet aufsummiert. Ähnlichkeiten zwischen Termen, die vergleichsweise selten Teil einer Klasse oder eines Clusters sind, werden hierbei stärker gewichtet. Aufgrund der berechneten Ähnlichkeiten können die Klassen abschließend den verschiedenen Clustern zugeordnet werden, sodass eine semantische Unterteilung des Quelltextes vorliegt.
In der Evaluation vergleichen sie ihren Ansatz zum einen mit der Latent Dirichlet Allocation (LDA) [BNJ03] Methode und zum anderen mit dem zuvor beschriebenen Ansatz von Kuhn et al. [KDG07]. Experten beurteilten hierfür die Zuordnungen zwischen Klassen und Konzepten in drei Softwaresystemen auf ihre Sinnhaftigkeit. In allen drei Fällen schneidet der Ansatz von Mahmoud und Bradshaw am besten ab. Die Experten stimmen hierbei den gefundenen Zuordnungen zwischen 50% und 70% zu. Im Vergleich zu den Ergebnissen mit LDA führt der Ansatz von Mahmoud und Bradshaw zu signifikant besseren Ergebnissen. Beim Vergleich mit dem Ansatz von Kuhn et al. zeigt sich die gleiche Tendenz, allerdings ohne statistische Signifikanz.

4.2.3 Kombinierte Ansätze

Einige Ansätze zur Konzeptlokalisierung nutzen eine Kombination der lexikalischen und syntaktischen Merkmale im Quelltext, anstelle sich auf eine der beiden Aspekte zu beschränken. Ein Beispiel hierfür ist die Arbeit **Clustering Support for Static Concept Location in Source Code** [SM11] von Scanniello und Marcus, die sowohl lexikalische als auch strukturelle Informationen explizit nutzen, um semantische Zusammenhänge zwischen Methoden zu identifizieren. Ausgangspunkt bildet ein gerichteter Graph, dessen Konten die Methoden im Quelltext widerspiegeln. Als strukturelles Merkmal nutzen sie Methodenaufrufe. Wird im Rumpf einer Methode m_1 eine andere Methode m_2 aufgerufen, existiert eine gerichtete Kante vom Knoten m_1 zum Knoten m_2 im Graphen. Die lexikalische Ähnlichkeit zwischen Methoden wird durch Kantengewichte ausgedrückt. Hierzu wird jede Methode auf ein Dokument abgebildet, welches aus den Wörtern der vorhandenen Kommentare und den benutzten Bezeichnern besteht. Anschließend erfolgt eine Vorverarbeitung der Dokumente, bei der Zahlen, Symbole und Stoppwörter entfernt, zusammengesetzte Wörter aufgetrennt und eine Stammformreduktion durchgeführt wird. Für die Berechnung der Ähnlichkeiten werden die Methodendokumente daraufhin mit Hilfe des Vektorraummodells auf Vektoren abgebildet. Der Kosinus des Winkels zwischen zwei Vektoren entspricht der lexikalischen Ähnlichkeit zwischen Methoden und wird als Kantengewicht in den Graph eingefügt. Abschließend erfolgt eine Clusteranalyse auf dem Graphen über den Border-Flow [NS09] Algorithmus, um Gruppen semantisch zusammenhängender Methoden zu identifizieren.
In der Evaluation nutzen sie ihren Ansatz, um Quelltextstellen, die von einer Fehlerbeschreibung (engl. bug report) betroffen sind, zu finden. Hierbei werden Fehlerbeschreibungen aus fünf Softwareprojekten herangezogen, zu denen bekannt ist, welche Quelltextstellen für die Behebung geändert wurden. Eine Fehlerbeschreibung wird auf die gleiche Weise vorverarbeitet, wie die Methodendokumente. Für die Bewertung der Relevanz eines Clusters wird die maximale Kosinus-Ähnlichkeit zwischen der Vektorabbildung der Fehlerbeschreibung und den Vektorabbildungen der Methodendokumente im Cluster herangezogen. Anschließend werden die Cluster entsprechend ihrer Relevanz absteigend sortiert.

Die Rangfolge der Methodendokumente innerhalb eines Clusters wird ebenfalls über die Kosinus-Ähnlichkeit zur Vektorabbildung der Fehlerbeschreibung ermittelt. Der Rang der ersten Methode in der sortierten Liste aller Methoden, die tatsächlich von der betrachteten Fehlerbeschreibung betroffen ist, dient als Kennzahl für die Güte ihres Ansatzes. Die erste relevante Methode bei der Nutzung des Ansatzes von Scanniello und Marcus befindet sich durchschnittlich auf einem Rang im Bereich zwischen 524 und 639. Werden ausschließlich lexikalische Informationen berücksichtigt, liegt die erste relevante Methode durchschnittlich auf einem Rang im Bereich zwischen 629 und 746. Insgesamt kommen Scanniello und Marcus zu dem Ergebnis, dass die Berücksichtigung von Methodenaufrufen, zusätzlich zu der lexikalischen Ähnlichkeit, zu einer signifikanten Verbesserung der Ergebnisse führt.

Misra et al. nutzen in ihrer Arbeit **Software Clustering: Unifying Syntactic and Semantic Features** [MAK+12] ebenfalls eine Kombination aus lexikalischen und strukturellen Merkmalen bei der Suche nach semantisch zusammenhängenden Klassen. Als lexikalische Merkmale werden Kommentare, Bezeichner, Paketpfade, Methoden- und Klassennamen verwendet. Bei der Vorverarbeitung werden Stoppwörter entfernt, zusammengesetzte Wörter aufgetrennt und eine Stammformreduktion durchgeführt. Für die Bestimmung der lexikalischen Ähnlichkeit zwischen den Wörtern aus Kommentaren und Bezeichnern wird das Vektorraummodell, in Verbindung mit der Kosinus-Ähnlichkeit der Vektorabbildungen, herangezogen. Für den Vergleich von Paketpfaden, Methoden- und Klassennamen wird ein Zeichenkettenvergleich eingesetzt. Die Vererbungsstruktur, Methodenaufrufe, gemeinsame Parameter und Rückgabetypen dienen als syntaktische Merkmale. Zur Bewertung der Vererbungsstruktur werden Klassen auf Namensmengen abgebildet. Für eine Klasse c_1 besteht die Menge aus dem Namen von c_1, allen von c_1 implementierten oder erweiterten Klassen und den Klassen die c_1 als Oberklasse besitzen. Anschließend wird das Verhältnis zwischen der Anzahl gemeinsamer Namen und der Anzahl aller Namen für die Bewertung der Vererbungsstruktur zweier Klassen herangezogen. Methodenaufrufe werden gemeinsam mit Parameter- und Rückgabetypabhängigkeiten betrachtet. Parameter- und Rückgabetypabhängigkeiten existieren, wenn eine Klasse als Parameter- oder Rückgabetyp einer Methode in einer anderen Klasse verwendet wird. Für die Bewertung der Abhängigkeit zwischen zwei Klassen c_1 und c_2 wird die Anzahl der Aufruf- oder Typabhängigkeiten von c_1 zu c_2 durch die Gesamtzahl eingehender Abhängigkeiten in c_2 normiert. Den Zusammenhang zweier Klassen bewerten sie durch den Mittelwert der berechneten lexikalischen und strukturellen Ähnlichkeitskennzahlen. Am Ende wird ein Graph aufgebaut, dessen Knoten die Klassen im Quelltext repräsentieren. Die Kanten zwischen zwei Knoten entsprechen den berechneten Zusammenhängen. Ausgehend von den Knoten mit den am stärksten gewichteten Kanten, suchen Misra et al. nach einer Partitionierung mit maximalem inneren Zusammenhang und minimalen Abhängigkeiten zwischen den Partitionen.
In ihrer Evaluation vergleichen sie für sechs Softwareprojekte das Ergebnis ihres Verfahrens mit manuell erstellten Musterpartitionierungen. Zur Bewertung der Übereinstimmung untersuchen sie die Anzahl an Operationen die notwendig sind, um die gefundene Clusteraufteilung in die Musterlösung zu überführen. Eine Übereinstimmung von 1 bedeutet, dass die gefundenen Cluster mit der Musterlösung übereinstimmen. Die Ergebnisse ihres Ansatzes liegen im Bereich zwischen 0,43 und 0,81. Zum Vergleich führen Misra et al. ihren Ansatz mit unterschiedlichen Gewichten für die Ähnlichkeitskennzahlen aus, sodass bspw. nur die Klassennamen berücksichtigt werden, um den Einfluss der einzelnen Ähnlichkeitsmerkmale bewerten zu können. Hierbei zeigt sich, dass, abhängig vom analysierten Projekt, unterschiedliche Ähnlichkeitsmerkmale zum besten Ergebnis führen. Bei zwei Projekten konnte das beste Ergebnis von 0,55 bzw. 0,64 mit einer Beschränkung auf den Klassennamen als einziges Ähnlichkeitsmerkmal erzielt werden. Das Ergebnis bei einer gleichgewichteten Betrachtung aller Ähnlichkeitsmerkmale ist für fünf der sechs Softwareprojekte unter den besten zwei.

Einen Mehrwert in der Kombination von lexikalischen und strukturellen Merkmalen sehen auch Kuang et al. in ihrer Arbeit **Analyzing Closeness of Code Dependencies for Improving IR-Based Traceability Recovery** [KNH+17]. Bei der Suche nach Rückverfolgbarkeitsinformationen kombinieren sie lexikalische Ähnlichkeiten mit direkten und indirekten Quelltextabhängigkeiten um die, für eine Anforderung relevanten, Klassen im Quelltext zu finden. Im ersten Schritt suchen Kuang et al. relevante Klassen zu einer Anforderungsbeschreibung über eine lexikalische Analyse. Hierfür werden Klassen auf Dokumente abgebildet, die deren Kommentare und Bezeichner umfassen. Bei der Vorverarbeitung werden zusammengesetzte Wörter aufgetrennt, Schlüssel- und Stoppwörter entfernt und eine Stammformreduktion durchgeführt. Anschließend werden relevante Klassendokumente über Methoden der Informationsrückgewinnung (Siehe Abschnitt 2.3.3) identifiziert. In einem weiteren Schritt erweitern Kuang et al. die Menge der relevanten Klassen mit Klassen, die eine hohe direkte oder indirekte Abhängigkeit zu einer der bereits gefundenen Klassen aufweisen. Eine direkte Abhängigkeit besteht, sobald ein Methodenaufruf, eine Klassenbenutzung oder eine Vererbungsbeziehung zwischen Klassen existiert. Bei der Bewertung der direkten Abhängigkeiten zwischen zwei Klassen c_1 und c_2 wird die Anzahl direkter Abhängigkeiten von c_1 zu c_2 ins Verhältnis zur Gesamtzahl an ausgehenden Abhängigkeiten von c_1 und eingehenden Abhängigkeiten von c_2 gesetzt. Mit der indirekten Abhängigkeit quantifizieren die Autoren, in wie weit zwei Klassen dieselben Datentypen verwenden. Die Kardinalität der Menge an Datentypen, die in beiden Klassen referenziert werden, wird hierfür ins Verhältnis zur Kardinalität der Menge an Datentypen, die in mindestens eine der beiden Klassen referenziert werden, gesetzt.
Bei ihrer Evaluation suchen sie für Anforderungen aus drei verschiedenen Softwareprojekten die dazugehörenden Klassen im Quelltext. Als Vergleich dient die lexikalische Analyse ohne Hinzunahme der strukturellen Abhängigkeiten, ein auf PageRank basierender Ansatz [SMP15] und ein alternativer Ansatz zur Kombination von strukturellen und lexikalischen Informationen [MPR09]. Zur Bewertung der gefundenen Klassen dienen Musterlösungen, die für alle drei Projekte die Zuordnung der Klassen zu ihren Anforderungen beschreiben. In 81% der Fälle ist der F1-Wert des Ansatzes von Kuang et al. signifikant höher, als bei den Vergleichsansätzen.

4.3 Automatisierte Softwarerefaktorisierung

Die Bewertung von Zusammenhängen im Quelltext ist ein zentrales Thema im Forschungsgebiet der automatisierten Softwarerefaktorisierung, weshalb auch Ansätze aus diesem Gebiet für die vorliegende Arbeit relevant sind. Corazza et al. untersuchen in ihrer Arbeit **Weighing lexical information for software clustering in the context of architecture recovery** [CDMMS16] die Gewichtung der lexikalischen Bestandteile in Abhängigkeit von der Stelle, an der sie im Quelltext auftauchen. Die Verarbeitung von Quelltext erfolgt auf Klassenebene. Eine Klasse wird in sechs Zonen unterteilt:

1. Die *Klassenzone* enthält den Klassennamen und die Namen von erweiterten Klassen oder implementierten Schnittstellen.
2. Die *Attributzone* enthält die Attributnamen und deren Typbezeichnungen.
3. Die *Methodenzone* enthält Methodennamen und die Namen von Rückgabetypen.
4. Die *Parameterzone* enthält alle Parameterbezeichnungen von Methoden, inklusive ihrer Typbezeichnungen.
5. Die *Kommentarzone* enthält alle Kommentare innerhalb einer Klasse, wobei Lizenzbeschreibungen entfernt werden.
6. Die *Anweisungszone* enthält alle Terme der Anweisungen in den Methoden einer Klasse.

Primitive Datentypen, die in den verschiedenen Zonen einer Klasse auftreten, werden von Corazza et al. bei der weiteren Verarbeitung nicht berücksichtigt. Bei der Vorverarbeitung der Terme in den einzelnen Zonen werden HTML-Markierungen, Stoppwörter und Java-Schlüsselwörter entfernt, zusammengesetzte Bezeichner aufgetrennt und eine Stammformreduktion durchgeführt. Darüber hinaus werden alle Buchstaben auf Kleinbuchstaben abgebildet. Die Dokumentabbildung einer Klasse entspricht daraufhin den vorverarbeiteten Termmengen aus den sechs Zonen.
Für den Vergleich der Klassendokumente wird das Vektorraummodell herangezogen. Hierzu wird eine Term-Dokument-Matrix aufgebaut. Für jede Zone werden die Terme über das TF-IDF-Maß gewichtet. Darüber hinaus wird der Einfluss der verschiedenen Zonen durch einen zusätzlichen Gewichtsfaktor berücksichtigt. Hierzu wird für jede Zone ein probabilistisches Modell erzeugt, welches die Wahrscheinlichkeit, mit der ein Term in einer Zone im Quelltext auftaucht, beschreibt. Die sechs probabilistischen Modelle werden in einer Mischverteilung vereint, welche die Zonengewichte als Parameter besitzt und die Wahrscheinlichkeit beschreibt, mit der ein Term in einer Klasse auftritt. Die Parameter des Modells werden über den Expectation-Maximization Algorithmus [MK07] bestimmt. Anschließend werden die Einträge der Term-Dokument-Matrix mit dem Gewicht der Zone, aus dem der durch einen Eintrag repräsentierte Term stammt, multipliziert. Die resultierende Term-Dokument-Matrix nutzen Corazza et al. um die Klassendokumente auf Vektoren abzubilden. Der Vergleich zweier Vektorabbildungen erfolgt über die Kosinus-Ähnlichkeit. Eine abschließende Unterteilung der Klassen in zusammenhängende Cluster erfolgt durch eine Clusteranalyse auf der Grundlage des k-Medoid Algorithmus [KR09].
In der Evaluation vergleichen sie für 19 Softwareprojekte ihre Systemunterteilung mit einer Musterpartitionierung. Für die Bewertung der Übereinstimmung ihrer Ergebnisse wird das MojoFM-Maß [WT04] herangezogen. Der berechnete Wert liegt zwischen 0 und 1, wobei 1 eine perfekte Übereinstimmung mit der Musterlösung beschreibt. Ohne Berücksichtigung der Zonen im Quelltext liegt die durchschnittliche Übereinstimmung ihrer Ergebnisse bei 0,51. Durch die Unterscheidung der Zonen aus denen die Terme einer Klasse stammen, ohne dabei die Zonengewichte bei der Gewichtung der Term-Dokument-Matrix zu berücksichtigen, kann eine signifikante Verbesserung um durchschnittlich 44% erzielt werden. Die Erweiterung um die individuelle Gewichtung der Zonen führt darüber hinaus zu einer signifikanten Verbesserung von durchschnittlich 11% im Vergleich zu der Berücksichtigung der Zonen ohne Gewichtung.

Bei der Bewertung der Zusammenhänge zwischen Klassen beschäftigen sich Amarjeet und Chhabra in ihrer Arbeit **Improving modular structure of software system using structural and lexical dependency** [AC17] mit der Gewichtung von Abhängigkeitsmerkmalen. Ihr Ziel ist die automatisierte Restrukturierung von Softwaresystemen, um eine gute Modularisierung zu erreichen. Für die Gewichtung der strukturellen Abhängigkeiten untersuchen sie die Ausprägung der jeweiligen Merkmale in zwei existierenden Sofwareprojekten, die sie als gut strukturiert bezeichnen. Ihre Untersuchung zeigt, dass Vererbungsbeziehungen das stärkste Indiz für einen Zusammenhang zweier Klassen darstellen. Methodenaufrufe, Implementierungen von Schnittstellen, Benutzung anderer Klassen als Parameter- oder Attributtyp und die Erzeugung neuer Objekte besitzen eine mittlere Aussagekraft. Die geringste Relevanz für den Zusammenhang zwischen Klassen besitzt die Verwendung einer Klasse bei der Deklaration von Ausnahmen oder als Rückgabetyp. Für die Berechnung des strukturellen Zusammenhangs zwischen Klassen vergleichen sie verschiedene Kombinationsmöglichkeiten mit und ohne Gewichtung. Jede Klasse wird mit jeder anderen Klasse verglichen und eine Klassenmatrix aufgebaut, welche die strukturellen Zusammenhänge zwischen allen Klassen im Quelltext beschreibt. Für die Berechnung der Zusammenhänge bilden Amarjeet und Chhabra die Klassen auf Vektoren ab, deren Elemente die strukturellen Abhängigkeiten zu den anderen Klassen im System beschreiben. Die Bewertung des strukturellen Zusammenhangs zweier Klassen erfolgt durch die Berechnung der Kosinus-

Ähnlichkeit zwischen den Vektorrepräsentationen der Klassen.
Für die lexikalische Analyse extrahieren Amarjeet und Chhabra die Bezeichner von Klassen, Attributen, Methoden und Parametern, sowie Anweisungen und Kommentare im Quelltext. Die Bewertung der lexikalischen Ähnlichkeit erfolgt anschließend auf der Basis des Vektorraummodells über die Berechnung der Kosinus-Ähnlichkeiten zwischen den Vektorabbildungen der Klassen. In Anlehnung an die Vorgehensweise von Corazza et al. [CDMMS16] wird bei den Einträgen in der Term-Dokument-Matrix die Stelle innerhalb einer Klasse, aus der ein Wort stammt, berücksichtigt.
In der Evaluation untersuchen Amarjeet und Chhabra acht Softwareprojekte. Hierbei werden jeweils 20 Klassen hinsichtlich ihres Zusammenhangs bewertet und mit einer, von 19 externen Entwicklern bereitgestellten, Musterlösung verglichen. Die besten Ergebnisse besitzen eine Übereinstimmung mit den Expertenbewertungen im Bereich zwischen 0,75 und 0,97. Insgesamt kommen Amarjeet und Chhabra zu dem Schluss, dass sowohl die Kombination von strukturellen und lexikalischen Informationen als auch die individuelle Gewichtung von Abhängigkeitsmerkmalen sinnvoll ist.

Ebenfalls aus dem Bereich der automatisierten Softwarerefaktorisierung stammt die Arbeit **Using structural and semantic metrics to improve class cohesion** [LOV08] von De Lucia et al., in der ein Ansatz zur Auftrennung schwach zusammenhängender Klassen vorgestellt wird. Um den Zusammenhang zwischen den Methoden innerhalb einer Klasse zu beurteilen, werden strukturelle als auch lexikalische Merkmale in einer separaten Analyse betrachtet. Das Ergebnis der Analysen ist jeweils ein gewichteter Graph, dessen Knoten die Methoden der betrachteten Klasse repräsentieren und dessen Kanten den Zusammenhang zwischen diesen widerspiegeln. Für die semantische Ähnlichkeit zwischen Methoden wird die Kosinus-Ähnlichkeit der Vektorabbildungen der Kommentare und Bezeichner einer Methode als Kantengewicht verwendet. Bei der Beurteilung des syntaktischen Zusammenhangs wird die Anzahl von gemeinsam verwendeten Variablen, im Verhältnis zur Anzahl aller verwendeten Variablen, zweier Methoden als Kantengewichtung verwendet. Aufgrund der isolierten Betrachtung einer Klasse ist die Anzahl verfügbarer struktureller Merkmale stark eingeschränkt. Vor der Suche nach einer geeigneten Unterteilung durch den Maximum Flow [CLRS09] Algorithmus werden beide Graphen vereint. Die Knotenmenge des neuen Graphen ist identisch zur Knotenmenge der beiden Ausgangsgraphen. Das Gewicht der Kanten zwischen zwei Knoten im neuen Graphen entspricht dem gewichteten Mittelwert der bestehenden Kantengewichte aus den Ausgangsgraphen.
Ausgangspunkt der Evaluation stellen 20 vorab vereinigte Klassen dar, welche durch den Ansatz von De Lucia et al. aufgeteilt werden sollen. Beim Vergleich unterschiedlicher Gewichte bei der Zusammenführung der strukturellen und lexikalischen Kanten stellte sich heraus, dass es keine signifikanten Unterschiede zwischen einer Gewichtung von 50:50, 70:30 oder 30:70 gibt. Tendenziell wurden die Ergebnisse jedoch schlechter, wenn den strukturellen Kanten ein höheres Gewicht als 0,5 zugesprochen wurde. Das beste Ergebnis in Form eines durchschnittliches F1-Maßes von 0,71 konnte mit einer Gleichgewichtung der strukturellen und lexikalischen Kanten erzielt werden. Des Weiteren zeigten die Ergebnisse eines weiteren Experiments, bei dem 50 Klassen aufgeteilt wurden, dass die Kombination syntaktischer und semantischer Merkmale mit einem durchschnittlichen F1-Maß von 0,65, im Vergleich zur ausschließlich lexikalischen (0,54) oder strukturellen (0,46) Betrachtung, zu besseren Ergebnissen führt.

4.4 Programmverständnis

Neben Arbeiten zur Quelltextrepräsentation, Konzeptlokalisierung und Softwarerefaktorisierung gibt es weitere Ansätze aus dem Bereich des rechnergestützten Programmverständnisses, die für Teile der vorliegende Arbeit relevant sind.

4.4.1 Identifikation von Entwurfsmustern

Die Arbeit **Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach** [OAS16] von Oruc et al. befasst sich mit der automatisierten Identifikation von Entwurfsmustern in Softwaresystemen. Hierzu werden der Quelltext und die gesuchten Entwurfsmuster als Graphen repräsentiert, wobei die Klassen in Form von Knoten dargestellt werden. Wie auch schon in anderen Arbeiten repräsentieren die Kanten syntaktische Abhängigkeiten. Insgesamt werden 12 verschiedene Beziehungstypen definiert, um im Graphen, z.B. zwischen der Abhängigkeit aufgrund einer Methodenüberschreibung und der Abhängigkeit aufgrund einer Vererbungsbeziehung, unterscheiden zu können. Die Suche nach Übereinstimmungen erfolgt auf dem gesamten Quelltextgraphen über den Graphisomorphie-Algorithmus von Subdue[5].
Oruc et al. geben an, dass ihr Ansatz in der Lage ist, 23 Entwurfsmuster zu erkennen. In der Evaluation werden fünf Entwurfsmuster in drei Softwareprojekten mit einer Präzision im Bereich zwischen 22% und 100% und einer Ausbeute im Bereich zwischen 25% und 100% erkannt.

Auch Mayvan et al. befassen sich in ihrer Arbeit **Design pattern detection based on the graph theory** [MR17] mit der automatisierten Identifikation von Entwurfsmustern in Softwaresystemen. Hierbei verwenden sie ebenfalls Graphrepräsentationen des Quelltextes und der gesuchten Entwurfsmuster. Im Gegensatz zu Oruc et al. verwenden sie nur die Abhängigkeitsbeziehungen aus den Klassendiagrammen als Kanten ihrer Graphrepräsentation. Darüber hinaus unterteilen sie den Systemgraphen bevor sie nach Übereinstimmungen mit einem gesuchten Entwurfsmuster suchen. Hierfür prüfen sie für alle Knoten im Systemgraphen, ob es einen Knoten in einem Entwurfsmustergraphen gibt, der die gleichen ein- und ausgehenden Kanten besitzt. Alle Knoten im Systemgraphen, die in einem der Entwurfsmustergraphen vorkommen, werden hierbei markiert. Als nächstes wird für alle Kanten des Systemgraphen geprüft, ob sie in einem der Entwurfsmustergraphen auftreten. Hierbei werden die Markierungen der angrenzenden Knoten im Systemgraphen und dem Entwurfsmustergraphen verglichen. Kanten, die in keinem Entwurfsmustergraphen vorkommen, werden aus dem Systemgraphen entfernt. Die entstandenen Teilgraphen werden anschließend auf Übereinstimmungen mit den Entwurfsmustergraphen untersucht. Für jedes identifizierte Entwurfsmuster wird zusätzlich geprüft, ob es die syntaktischen Implikationen des Entwurfsmusters erfüllt. Bei einer Übereinstimmung mit dem Entwurfsmuster *Einzelstück* (engl. Singleton) wird bspw. geprüft, ob der Konstruktor privat ist und ob es eine öffentliche Methode gibt, die eine Instanz der Klasse zurückliefert. Hierdurch sollen falsch erkannte Übereinstimmungen herausgefiltert werden.
Bei der Evaluation untersuchen sie drei Softwareprojekte auf 10 Entwurfsmuster. Die gefundenen Entwurfsmuster werden von 10 Master-Studierenden überprüft. Für jede der 30 Entwurfsmuster-Projekt Kombinationen berechnen sie die Präzision und die Ausbeute. In 77% der Fälle erzielt ihr Ansatz eine Präzision von 100% und in 40% der Fälle wird eine Ausbeute von 100% erreicht.

4.4.2 Auflösung von Abkürzungen im Quelltext

Das Ergebnis einer lexikalischen Analyse der natürlichen Sprache im Quelltext ist abhängig von der Qualität der im Quelltext vorhandenen Kommentare und den verwendeten Bezeichnern [MAK+12]. Jiang et al. zeigen in ihrer Arbeit **Automatic and Accurate Expansion of Abbreviations in Parameters** [JLZZ18] einen Ansatz, um die Aussagekraft der textuellen Informationen, durch die automatische Auflösung von Abkürzungen in Parameternamen, zu erhöhen. Hierzu werden zusammengesetzte Parameter wie bei der

[5] `http://ailab.wsu.edu/subdue/`, zuletzt besucht am 20.01.2020

Binnenmajuskel-Schreibweise, durch Trennung an den Großbuchstaben, in ihre Teilwörter zerlegt. Teilwörter die nicht im Wörterbuch zu finden sind werden als Abkürzungen betrachtet. Anschließend wird geprüft, ob eine Abkürzung ein Akronym, Präfix oder das Resultat von weggelassenen Buchstaben eines Wortes ist. Hierzu wird eine Abkürzung mit den Bezeichnern der Variablen, die beim Aufruf der Methode als Parameter übergeben werden, bzw. die in der Methodensignatur auftreten, verglichen. Lässt sich die Abkürzung nicht auflösen, wird in einem weiteren Schritt der Parametertyp betrachtet. Hierbei wird geprüft, ob die Abkürzung ein Akronym oder Präfix der Typbezeichnung des Parameters ist. Kann die Abkürzung weiterhin nicht aufgelöst werden, kommt als letzte Instanz ein Abkürzungswörterbuch zum Einsatz.
Zur Evaluation wenden Jiang et al. ihren Ansatz in acht Softwareprojekte an. Insgesamt erreicht ihr Verfahren eine Präzision zwischen 94% und 97% bei einer Ausbeute im Bereich zwischen 59% und 74%. Im Vergleich dazu findet LINSEN [CDMM12], ein alternativer Ansatz zur Auflösung von Abkürzungen, in durchschnittlich 59,1% der Fälle die korrekte Auflösung einer Abkürzung.

5 Analyse und Entwurf

Wissen über die Zusammenhänge zwischen den Anforderungen und dem Quelltext eines Softwareprojekts sind beim Aufbau von Programmverständnis, sowie bei der Durchführung von Wartungsarbeiten, Auswirkungsanalysen und weiteren Tätigkeiten bei der Softwareentwicklung hilfreich [ACC+02]. Rückverfolgbarkeitsinformationen beschreiben diese Zusammenhänge explizit und erlauben eine Zuordnung von Anforderungen zu Quelltext und umgekehrt. Der hohe manuelle Aufwand für die Extraktion und Pflege solcher Informationen führt jedoch dazu, dass sie in den meisten heutigen Softwareprojekten nicht dokumentiert werden. Um die Vorteile einer Rückverfolgbarkeit trotzdem nutzen zu können, befasst sich das Forschungsprojekt INDIRECT aus Kapitel 3 mit der automatisierten Bereitstellung von Rückverfolgbarkeitsinformationen. Das Vorgehen basiert auf einem Vergleich von Absichtsmodellen. Hierbei handelt es sich um Graphen, die eine semantische Beschreibung des Quelltextes bzw. der Anforderungen liefern. Im Hinblick auf das Absichtsmodell von Quelltext, stehen semantische Zusammenhänge zwischen Programmstellen im Mittelpunkt, da sie Rückschlüsse auf die Anzahl und Ausbreitung der implementierten Funktionalitäten erlauben. Im Rahmen dieser Arbeit ist ein semantischer Zusammenhang, wie in Definition 5.1 angegeben, definiert.

Definition 5.1: Semantischer Zusammenhang

Zwei Programmstellen besitzen einen semantischen Zusammenhang, wenn sie einen gemeinsamen Zweck verfolgen, oder im Hinblick auf ein bestimmtes Ziel zusammenarbeiten.

Das Absichtsmodell des Quelltextes soll die semantischen Zusammenhänge im Quelltext explizit darstellen, sodass sie bei der Rückverfolgbarkeitsanalyse in INDIRECT genutzt werden können. Die Suche nach semantischen Zusammenhängen stellt im Allgemeinen eine Herausforderung dar, da semantische Zusammenhänge nicht explizit dokumentiert sind, sondern aus Informationen wie der Syntax oder der natürlichen Sprache im Quelltext abgeleitet werden müssen. Erschwerend kommt hinzu, dass Programmstellen prinzipiell für mehrere Funktionalitäten von Bedeutung sein können und infolgedessen mehrere Zusammenhänge besitzen können. Beispiel 5.1 veranschaulicht dies anhand einer Methode, die in zwei unterschiedlichen Kontexten verwendet wird.

Beispiel 5.1: Semantische Zusammenhänge im Quelltext

Gegeben sei eine Methode `getContractByEmployeeId(Long emplId)`, die den Vertrag eines Mitarbeiters zurückliefert. Im Quelltext wird sie an zwei Stellen aufgerufen. Einmal innerhalb der Methode `calculateSalary(Long emplId)` für die Gehaltsberechnung und einmal innerhalb der Methode `showContract(Long emplId)` für die Anzeige eines Vertrags. Aufgrund von Definition 5.1 besitzt die Methode `getContractByEmployeeId(Long emplId)` daher sowohl einen semantischen Zusammenhang mit der Methode `calculateSalary(Long emplId)`, als auch mit der Methode `showContract(Long emplId)`.

Insgesamt erfordert die Identifikation semantischer Zusammenhänge daher eine umfassende Analyse und Bewertung verschiedener Informationen im Quelltext. Mit der vorliegenden Arbeit soll das Fundament eines Absichtsmodells des Quelltextes für INDIRECT erstellt werden. Hierzu wird eine semantische Repräsentation von Quelltext entworfen und umgesetzt, deren Fokus auf der Beschreibung semantischer Zusammenhänge liegt. Für die Suche nach semantischen Zusammenhängen werden zwei statische Quelltextanalysen implementiert, welche die syntaktischen und lexikalischen Informationen in Java-Quelltext[1] untersuchen. Eine zukünftige Erweiterung um weitere Analysen, die bei der Identifikation semantischer Zusammenhänge helfen, wird explizit berücksichtigt.

5.1 Zieldefinition

Das allgemeine Ziel dieser Arbeit besteht im Entwurf und Aufbau einer semantischen Repräsentation von Quelltext. Bei näherer Betrachtung können hieraus mehrere Zwischenziele (Z1) bis (Z3) abgeleitet werden. Jedes Zwischenziel besitzt eine eigene Fragestellung, die es im Hinblick auf das Erreichen des allgemeinen Ziels zu lösen gilt. Das restliche Kapitel befasst sich nacheinander im Detail mit den verschiedenen Zwischenzielen und stellt jeweils einen Lösungsentwurf vor.

(Z1) Entwurf der semantischen Repräsentation von Quelltext: Welche Informationen im Quelltext sind relevant und wie können sie in einem Graph passend dargestellt werden?

(Z2) Semantische Zusammenhänge im Quelltext erkennen: Welche syntaktischen und lexikalischen Informationen liefern Hinweise auf semantische Zusammenhänge und wie können sie im Rahmen einer statischen Quelltextanalyse ausgenutzt werden?

(Z3) Aufbau der semantischen Quelltextrepräsentation: Wie können die Ergebnisse einer flexiblen Anzahl von Analysen kombiniert werden, um eine gemeinsame semantische Quelltextrepräsentation aufzubauen?

5.2 Semantische Repräsentation von Quelltext

Die semantische Repräsentation von Quelltext soll in INDIRECT bei der Rückverfolgbarkeitsanalyse eingesetzt werden. Da INDIRECT eine graphbasierte Beschreibung von Quelltext erwartet, wird die semantische Quelltextrepräsentation als Graph umgesetzt. Um hierbei keine Einschränkung gegenüber einer direkten Analyse von Quelltext darzustellen, ist es sinnvoll, eine allgemeine Beschreibung von Quelltext als Ausgangspunkt der Repräsentation zu wählen und diese mit der Darstellung semantischer Zusammenhänge zu ergänzen. Eine umfassende Beschreibung von Quelltext liefert bspw. der abstrakte

[1] `https://www.java.com/de/`, zuletzt besucht am 27.01.2020

Syntaxbaum (Siehe Abschnitt 2.2.1). Ein Graph, der aus den Informationen des abstrakten Syntaxbaums besteht, besitzt alle, im Quelltext vorkommenden, Programmelemente als Knoten und beschreibt die syntaktischen Abhängigkeiten zwischen ihnen mit Hilfe von verschiedenen Kanten. Infolgedessen erlaubt er eine Darstellung von Zusammenhängen auf verschiedenen Granularitätsebenen. Dies steht im Gegensatz zu vielen verwandten Arbeiten wie [SM11, KGH+19, GAL18], die lediglich Programmelemente eines bestimmten Typs, z.B. Klassen oder Methoden, als Knoten ihres Graphen verwenden und sich dadurch bei der Beschreibung semantischer Zusammenhänge einschränken. Eine solche Einschränkung ist ein Nachteil, da semantische Zusammenhänge zwischen verschiedenen Programmelementtypen bestehen können. Als Beispiel für einen semantischen Zusammenhang einzelner Anweisungen dient das Protokollieren (engl. logging). Offensichtlich besitzen alle Anweisungen, die einen Protokolleintrag erzeugen, einen semantischen Zusammenhang, welcher jedoch nicht zwangsläufig für die Methoden, in denen die Anweisungen platziert sind, gilt. Eine Repräsentation des Quelltextes durch Klassen oder Methoden wäre daher nicht in der Lage, diese Art von Querschnittsfunktionalitäten (engl. cross-cutting concerns) darzustellen. Umgekehrt ist eine Repräsentation von Quelltext auf der Anweisungsebene nicht in der Lage, Zusammenhänge zwischen Funktionalitäten, z.B. Methoden, die sich erst aus der gemeinsamen Betrachtung von mehreren Anweisungen ergeben, darzustellen. Um eine hohe Ausdrucksmächtigkeit der semantischen Quelltextrepräsentation zu gewährleisten ist es demzufolge notwendig, alle Programmelemente im Quelltext explizit darzustellen und die Darstellung semantischer Zusammenhänge zwischen beliebigen Programmstellen zu erlauben. Auch im Hinblick auf spätere Erweiterungen der Quelltextrepräsentation ist es sinnvoll, eine möglichst uneingeschränkte und allgemeine Beschreibung anzustreben. Insgesamt stellt der abstrakte Syntaxbaum daher eine geeignete Grundlage für die semantische Quelltextrepräsentation in dieser Arbeit dar.

5.2.1 Darstellung semantischer Zusammenhänge

Neben der grundsätzlichen Struktur des Graphen ist vor allem die Frage interessant, wie ein semantischer Zusammenhang dargestellt und in die Graphrepräsentation integriert werden kann. Semantische Zusammenhänge können als Beziehungen zwischen Programmelementen aufgefasst werden. Im Kontext der Graphrepräsentation lassen sie sich als Kanten zwischen Programmelementknoten interpretieren. Grundsätzlich kann eine Programmstelle mehrere semantische Zusammenhänge besitzen. Dementsprechend kann ein Programmelementknoten Kanten zu mehreren anderen Programmelementknoten besitzen. Darüber hinaus ist die Existenz mehrerer semantischer Zusammenhänge zwischen zwei Programmelementen möglich, wenn sie bspw. hinsichtlich verschiedener Absichten gemeinsam verwendet werden.
Eine Möglichkeit bei der Modellierung von semantischen Zusammenhängen als Kanten, besteht in der Wahl zwischen gerichteten und ungerichteten Kanten. Sind zwei Knoten v_1 und v_2 über eine ungerichtete Kante verbunden, so besitzt sowohl v_1 einen Bezug zu v_2 als auch umgekehrt. Bei gerichteten Kanten ist es hingegen möglich, dass v_1 einen Bezug zu v_2, v_2 aber nicht zu v_1 hat. Für die Darstellung der Syntax ist der Einsatz gerichteter Kanten sinnvoll, da es unidirektionale Abhängigkeiten wie Methodenaufrufe gibt. Im Gegensatz dazu besitzen semantische Zusammenhänge keine Richtung, da sie auf den Gemeinsamkeiten zwischen Programmstellen basieren. Infolgedessen ist es sinnvoll, ungerichtete Kanten zur Beschreibung semantischer Zusammenhänge einzusetzen.
Semantische Zusammenhänge müssen aus vorhandenen Informationen über den Quelltext abgeleitet werden. Abhängig von der Art und der Anzahl an Hinweisen die auf einen semantischen Zusammenhang hindeuten, kann die Sicherheit eines gefundenen semantischen Zusammenhangs bewertet werden. Einem semantischen Zusammenhang mit einer Sicherheit von 0,9 kann bspw. eine höhere Aussagekraft zugesprochen werden, als einem

semantischen Zusammenhang mit einer Sicherheit von 0,3. Informationen über die Sicherheit der gefundenen semantischen Zusammenhänge vergrößern den Informationsgehalt der Repräsentation und erlauben es Analysen, die auf der semantischen Quelltextrepräsentation aufsetztten, das Maß an Unsicherheit, mit dem gearbeitet werden soll, selbst festzulegen. Insgesamt ist eine explizite Darstellung der Sicherheiten von semantischen Zusammenhängen sinnvoll. Im Hinblick auf eine Beschreibung durch Kanten können Unsicherheiten als Kantengewichte angegeben werden. Die Hinzunahme von Kantengewichten stellt darüber hinaus keinen Nachteil gegenüber ungewichteten Kanten dar, da sich ein gewichteter Graph einfach in einen ungewichteten Graphen überführen lässt.

Arbeiten wie [KDG07] nutzen eine alternative Darstellung semantischer Zusammenhänge mit Hilfe von Konzeptclustern. Hierbei handelt es sich um Ansammlungen von Programmstellen mit einem gemeinsamen Ziel oder Zweck. Anstelle den Zusammenhang zwischen Programmstellen direkt zu beschreiben, geht ein semantischer Zusammenhang zwischen Programmstellen aus der Zugehörigkeit zum selben Konzeptcluster hervor. Eine ähnliche Darstellung wäre auch in dieser Arbeit möglich. Hierzu kann ein zusätzlicher Knotentyp definiert werden, der ein konkretes Konzept bzw. eine konkrete Absicht repräsentiert. Alle Programmstellen, die im Kontext dieses Konzepts relevant sind, können über eine Kante mit diesem Knoten verbunden werden. Da Programmstellen prinzipiell für verschiedene Absichten genutzt werden können, sollte es möglich sein, ein Programmelementknoten mit mehreren solcher Absichtsknoten zu verbinden. Ein Vorteil der expliziten Darstellung von Konzepten wie in [KDG07] ist, dass schnell und einfach alle Programmstellen identifiziert werden können, die einer gesuchten Absicht zugeordnet wurden. Demgegenüber steht der Nachteil, dass nicht mehr nachvollziehbar ist, welche Zusammenhänge zwischen Programmstellen innerhalb und zwischen Konzeptclustern bestehen, da Programmstellen lediglich Kanten zu Absichtsknoten besitzen und nicht zu anderen Programmelementknoten. Für das Ziel dieser Arbeit ist die Darstellung semantischer Zusammenhänge allein über Zuordnungen zu Absichtsknoten daher nicht ausreichend. Als Ergänzung zu einer feingranularen Beschreibung semantischer Zusammenhänge zwischen einzelnen Programmstellen ist die Einbeziehung von Absichtsknoten jedoch sinnvoll, um einen Überblick, der im Quelltext implementierten Konzepte und der dazugehörenden Programmstellen, anzubieten. Der resultierende Graph wäre damit in der Lage, sowohl Zusammenhänge zwischen Programmstellen als auch eine semantische Unterteilung des Quelltextes darzustellen. Darüber hinaus können Absichtsknoten mit weiteren Informationen über das repräsentierte Konzept versehen werden, um die Informationen in den verknüpften Quelltextstellen zusammenzufassen und somit weitere Ansatzpunkte für eine Rückverfolgbarkeitsanalyse zu liefern.

5.2.2 Entwurf der semantischen Repräsentation

Die Grundlage der semantischen Repräsentation bildet eine allgemeine Beschreibung von Quelltext. Angelehnt an den abstrakten Syntaxbaum (Siehe Abschnitt 2.2.1) umfasst der Graph Knoten zur Beschreibung von Paketen, Typen, Methoden, Anweisungen und Variablen. Bei Anweisungen wird zwischen Schleifen, Verzweigungen, Switch-Case, Try-Catch, Anweisungsblöcken und einzelnen Anweisungen unterschieden. Jeder Knoten besitzt Attribute, welche spezifische Informationen über das repräsentierte Programmelement im Quelltext beinhalten. Hierzu zählen vorhandene Kommentare, verwendete Modifikatoren, die Position, der Quelltext und der Name eines Programmelements.
Die strukturellen Zusammenhänge zwischen Programmelementknoten werden über gerichtete, ungewichtete Kanten ausgedrückt. Jeder Programmelementknoten v_i besitzt eine Kante vom Typ *parent* zu dem Knoten des Programmelements, in welchem das von v_i repräsentierte Programmelement definiert ist. Die Wurzel des Graphen bildet ein Knoten, der das Hauptverzeichnis, in dem der Quelltext liegt, repräsentiert. Neben dieser grund-

sätzlichen Baumstruktur umfasst die Repräsentation weitere Kanten, um die syntaktischen Beziehungen eines Programmelements zu beschreiben. Hierbei werden die Informationen über die Vererbungsstruktur, Aufrufabhängigkeiten, Typ- und Variablenreferenzen aus dem AST des Quelltextes extrahiert und als gerichtete, ungewichtete Kanten zwischen den entsprechenden Programmelementknoten dargestellt.

Mit den bisher vorgestellten Knoten und Kanten ist der Graph in der Lage, Quelltext auf ähnliche Weise wie der abstrakte Syntaxbaum zu repräsentieren. Diese allgemeine Darstellung ist wichtig, um eine möglichst uneingeschränkte Beschreibung semantischer Zusammenhänge zu erlauben. Darüber hinaus bleiben die Informationen, auf deren Grundlage die semantischen Zusammenhänge gefunden wurden, in der Quelltextrepräsentation erhalten, um die Nachvollziehbarkeit der semantischen Zusammenhänge sicherzustellen. Semantische Zusammenhänge werden schließlich durch ungerichtete, gewichtete Kanten zwischen Programmelementknoten gekennzeichnet. Das Gewicht quantifiziert die Gewissheit mit der ein semantischer Zusammenhang angenommen wird und liegt im Wertebereich $(0, 1]$. Ein semantischer Zusammenhang mit einem Gewicht von 0 bedeutet, dass kein semantischer Zusammenhang vorliegt. In diesen Fällen ist keine Kante zwischen den Programmelementen notwendig, weshalb das Gewicht einer semantischen Kante stets größer als 0 ist. Je größer das Kantengewicht, desto sicherer ist der semantische Zusammenhang zweier Programmstellen. Ein Programmelementknoten kann sowohl mehrere semantische Zusammenhänge zum gleichen, als auch zu verschiedenen anderen Programmelementknoten besitzen. Hierbei spielt der Typ des Knotens keine Rolle, sodass semantische Zusammenhänge zwischen Programmelementen beliebigen Typs zulässig sind.
Für die Beschreibung der implementierten Funktionalitäten im Quelltext existiert ein weiterer Knotentyp, der, in Anlehnung an die Beschreibungen aus Abschnitt 5.2.1, als Repräsentant einer Menge semantisch zusammenhängender Programmstellen fungiert. Knoten dieses Typs werden Absichtsknoten genannt und besitzen keine direkte Entsprechung im Quelltext. Sie beschreiben eine Unterteilung der Programmelemente im Quelltext in Gruppen mit gemeinsamem Zweck. Somit bieten sie eine alternative Sichtweise auf den Quelltext, bei dem die Absichten der Programmelemente im Mittelpunkt stehen. Darüber hinaus erlauben sie einen einfachen Zugriff sowohl auf die Absichten hinter einem Programmelement als auch auf die Programmelemente, die im Hinblick auf eine bestimmte Absicht relevant sind.
Absichtsknoten werden über gewichtete, ungerichtete Kanten mit allen Programmelementknoten, die im Hinblick auf die repräsentierte Absicht zusammenarbeiten oder gebraucht werden, verbunden. Alternativ zu dieser Darstellungsweise ist es möglich, die Zugehörigkeit zu einer Absicht als Kette von Programmelementknoten zu repräsentieren. Diese sind über Kanten verknüpft, wobei lediglich der erste Knoten eine Kante zum Absichtsknoten aufweist. Auf den Informationsgehalt der semantischen Repräsentation hat die Art der Darstellung keine Auswirkungen. Explizite Kanten zwischen einer Absicht und den dazugehörenden Programmelementknoten sind in ihrer Aussage jedoch einfacher zu verstehen, weshalb diese Darstellungsform für die Beschreibung von Absichten gewählt wird. Ein Programmelement kann mehrere Kanten zu unterschiedlichen Absichtsknoten besitzen, da es prinzipiell in verschiedenen Kontexten genutzt werden kann.

Abbildung 5.1 zeigt auf der linken Seite eine Übersicht der zuvor beschriebenen Elemente der semantischen Repräsentation. Die weißen Rechtecke repräsentieren Knoten und beschreiben die Programmelemente im Quelltext. Absichtsknoten werden durch grüne Rechtecke dargestellt. Die *parent*-Kante kennzeichnet den Definitionsursprung eines Programmelements. Hellgraue, gerichtete Kanten beschreiben die syntaktischen Beziehungen eines Programmelements. Semantische Zusammenhänge werden durch blaue Kanten zwischen Programmelementknoten dargestellt und besitzen ein Gewicht zwischen 0 und 1. Analog dazu werden Zugehörigkeiten zu Absichtsknoten durch grüne Kanten dargestellt, die eben-

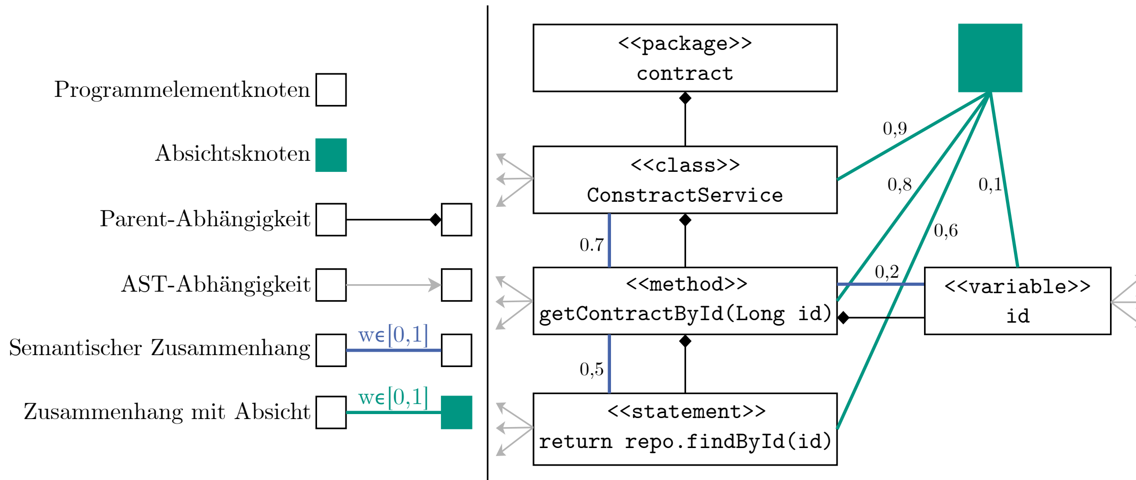


Abbildung 5.1: Auf der linken Seite sind die Bestandteile der semantischen Repräsentation schematisch dargestellt. Auf der rechten Seite ist ein beispielhafter Ausschnitt der semantischen Repräsentation zu sehen.

falls ein Gewicht zwischen 0 und 1 besitzen.
Auf der rechten Seite in Abbildung 5.1 ist ein beispielhafter Ausschnitt der semantischen Repräsentation zu sehen. Er umfasst insgesamt sechs Knoten. Fünf davon repräsentieren Programmelemente und ein weiterer Knoten beschreibt eine Absicht. Die *parent*-Abhängigkeiten zeigen an, dass die Klasse `ContractService` im Paket `contract` liegt und eine Methode `getContractById(Long id)` besitzt. Die Methode besitzt einen Parameter, der im Graph als Variablenknoten dargestellt ist und weist eine Kante zum Anweisungsknoten, der ihren Rumpf repräsentiert, auf. Variablenknoten können neben Methodenparametern auch lokale Variablen oder Typattribute repräsentieren und dementsprechend Typen- oder Anweisungsknoten als *parent*-Element besitzen.
Die AST-Abhängigkeiten beschreiben syntaktische Zusammenhänge zwischen Programmelementen. In der semantischen Repräsentation verweisen sie auf die Variablen, Typen oder Methoden, die von einem Programmelement verwendet werden. Einzige Ausnahme bilden Paketknoten. Sie besitzen keine AST-Abhängigkeiten, da sie lediglich die Ordnerstruktur eines Softwareprojekts beschreiben und keinen Quelltext darstellen. Aus Gründen der Übersichtlichkeit deutet der gezeigte Ausschnitt die AST-Abhängigkeiten lediglich an und verzichtet auf eine explizite Darstellung.
Insgesamt existieren drei semantische Zusammenhänge im rechten Teil von Abbildung 5.1. Der semantische Zusammenhang zwischen dem Typ `ContractService` und der Methode `getContractById` ist mit $0,7$ am sichersten. Der dargestellte Absichtsknoten besitzt Kanten zu vier Programmelementen. Auch hier geben die Kantengewichte die Sicherheit an, mit der ein Programmelement für eine konkrete Absicht wichtig ist.

5.3 Identifikation semantischer Zusammenhänge

Im Mittelpunkt der zuvor vorgestellten Quelltextrepräsentation steht die Beschreibung von semantischen Zusammenhängen (Siehe Definition 5.1). Um herauszufinden, welche Programmelemente semantisch zusammenhängen, können verschiedene Informationen im und über den Quelltext analysiert werden. Sowohl die syntaktische Struktur, als auch die textuellen Bestandteile von Quelltext können Hinweise auf Zusammenhänge zwischen Programmstellen liefern. Auf der Syntaxebene ist bspw. eine Vererbungsbeziehung zwischen zwei Klassen ein starkes Indiz für eine gemeinsame Absicht [AC17]. Im Bezug auf die textuellen Bestandteile liefert z.B. eine Überschneidung von Methodennamen, wie bei

`getEmployeeById` und `changeEmployeeId`, einen Hinweis auf einen Zusammenhang. Weitere Informationsquellen stellen Testfälle und Anwendungsszenarien dar. Mit ihrer Hilfe können Ausführungspfade und Laufzeitinformationen eines Programms untersucht werden, um Zusammenhänge zwischen Programmstellen zu erkennen [KQ05]. Allerdings ist eine dynamische Quelltextanalyse auf ausführbaren Quelltext eingeschränkt, da die Analyse von der Menge der vorhandenen Ausführungsszenarien oder Testfälle abhängig ist. Auch Versionsverwaltungsdaten können bei der Suche nach semantischen Zusammenhängen genutzt werden. So besitzen bspw. Programmstellen, die gemeinsam in einer Version geändert wurden, mit hoher Wahrscheinlichkeit einen Zusammenhang [RCS10]. Im Kontext von Versionsverwaltungsdaten können darüber hinaus die Versionsbeschreibungen (engl. commit messages) Informationen über den Zweck und die Zusammenhänge einer Änderung geben. Für den Aufbau der semantischen Quelltextrepräsentation sollen im Rahmen dieser Arbeit zwei statische Quelltextanalysen umgesetzt werden, um semantische Zusammenhänge zu identifizieren. Die erste Analyse soll die syntaktische Struktur von Quelltext und die zweite Analyse die natürlichsprachlichen Bestandteile im Quelltext untersuchen. Eine zukünftige Erweiterung um weitere Analysen für die Nutzung von Informationen aus z.B. Testfällen und Versionsverwaltungsdaten ist jedoch sinnvoll und wird in Abschnitt 5.4 explizit berücksichtigt.

5.3.1 Syntaxbasierte Quelltextanalyse

Eine Funktionalität, die es einer Klasse `ContractManager` erlaubt, Objekte vom Typ `Contract` zu manipulieren, führt zwangsläufig dazu, dass Abhängigkeiten zwischen beiden Klassen bzw. zwischen den Methoden der beiden Klassen entstehen. Folglich wirken sich semantische Zusammenhänge auf die Struktur von Quelltext aus. Die Analyse von Abhängigkeiten wie Methodenaufrufe oder Typreferenzen kann daher bei der Suche nach semantischen Zusammenhängen helfen. Im Allgemeinen existieren jedoch sehr viele Abhängigkeiten im Quelltext, von denen nicht alle gleichermaßen auf einen semantischen Zusammenhang hindeuten. Der Aufruf einer Methode die zusätzlich von diversen anderen Stellen aus aufgerufen wird, trägt vermutlich weniger zu einem semantischen Zusammenhang bei, als ein exklusiver Methodenaufruf. Die Herausforderung der syntaxbasierten Quelltextanalyse ist daher das Finden einer geeigneten Bewertungsstrategie der Abhängigkeiten im Quelltext, die Rückschlüsse auf die Existenz semantischer Zusammenhänge erlaubt.

Eine Möglichkeit semantische Zusammenhänge im Quelltext zu finden, bieten maschinelle Lernverfahren. Überwachte Lernverfahren können bspw. genutzt werden, um ein Modell zu trainieren, welches vorhersagt, ob zwei Programmelemente semantisch zusammenhängen oder nicht. Ein Vorteil dieser Verfahren ist, dass anhand von markierten Trainingsdaten automatisiert gelernt wird, welche Merkmale in den Eingabedaten relevant sind. In [ZWZ+19] werden bspw. neuronale Netze eingesetzt, um Vektorrepräsentationen von Quelltextfragmenten anhand ihrer AST Struktur zu lernen. Diese Vektoren werden anschließend genutzt, um die Ähnlichkeit zwischen Quelltextfragmenten zu berechnen und Klone zu identifizieren. Das Erlernen geeigneter Vektoren, die im Bezug auf die Aufgabenstellung dieser Arbeit eine Aussagekraft besitzen, erfordert eine repräsentative Menge an Trainingsdaten. Es gibt jedoch kaum Datensätze, die semantische Zusammenhänge im Quelltext explizit beschreiben. Die meisten verfügbaren Datensätze beschränken sich auf die Zusammenhänge zwischen Anforderungen und Klassen und sind für die Ziele dieser Arbeit folglich zu allgemein. Aufgrund der Vielzahl an Abhängigkeiten, die Quelltext naturgemäß aufweist, ist eine manuelle Extraktion von semantischen Zusammenhängen mit hohem Aufwand verbunden. Das Zusammenstellen eines hinreichend großen annotierten Datensatzes für das Training eines Lernverfahrens gestaltet sich daher schwierig. Allerdings können maschinelle Lernverfahren für die Bewertung von Teilaspekten, die im Bezug

auf semantische Zusammenhänge eine Rolle spielen, herangezogen werden. Ein neuronales Netz zur Quelltext-Einbettung (engl. code embedding), wie in [ZWZ+19] beschrieben, kann bspw. genutzt werden, um die funktionale Ähnlichkeit von Quelltextausschnitten bei der Suche nach semantischen Zusammenhängen mit einfließen zu lassen.
Unüberwachte Lernverfahren kommen ohne markierte Trainingsdaten aus und versuchen selbstständig Strukturen in den Eingabedaten zu erkennen, um eine Segmentierung großer unstrukturierter Datenmengen zu liefern. Inwieweit ein gelerntes Modell für den gewünschten Einsatzzweck geeignet ist, muss im Nachhinein manuell evaluiert werden. Bei Clusteranalysen die zu den unüberwachten Lernverfahren zählen, kann bspw. durch die Repräsentation der Eingabedaten und die Definition von Kriterien für die Clusterbildung das Ergebnis beeinflusst werden. Die Auswahl geeigneter Kriterien ähnelt dabei stark der Definition von Regeln bei regelbasierten Ansätzen. Unüberwachte Lernverfahren bringen bei der Suche nach semantischen Zusammenhängen in dieser Arbeit daher keinen erkennbaren Vorteil gegenüber regelbasierten Verfahren mit sich.

Alternativ zu Ansätzen aus dem Bereich des maschinellen Lernens können regelbasierte Ansätze bei der Syntaxanalyse von Quelltext eingesetzt werden. Ein Vorteil dieser Ansätze liegt darin, dass keine Trainingsdaten benötigt werden. Allerdings erfordern sie die manuelle Definition von Merkmalen und Heuristiken, aufgrund derer entschieden wird, ob ein semantischer Zusammenhang vorliegt. Naheliegende Merkmale die auf semantische Zusammenhänge hindeuten können, sind syntaktische Abhängigkeiten und Beziehungen zwischen Programmelementen. Des Weiteren kann ein regelbasierter Ansatz mit maschinellen Lernverfahren kombiniert werden. So können neben manuell festgelegten Regeln bspw. neuronale Netze eingesetzt werden, um die funktionale Ähnlichkeit von Quelltextausschnitten zu bewerten. Durch die manuelle Definition des Einflusses der berücksichtigten Merkmale können regelbasierte Ansätze einfach auf projektspezifische Charakteristiken angepasst werden. Im Vergleich zu maschinellen Lernverfahren stellt dies ein Vorteil dar, da hier ein gelerntes Modell nach dem Training in der Regel nicht einfach auf die Besonderheiten eines Softwareprojekts angepasst werden kann. Insbesondere bei der Suche nach semantischen Zusammenhängen kann die Anpassbarkeit wichtig sein, da die Ausprägung semantischer Zusammenhänge vermutlich projektspezifisch ist und von den Entwicklern und der Anwendungsdomäne abhängt. Darüber hinaus sind die Ergebnisse regelbasierter Ansätze transparent, sodass nachvollzogen werden kann, welche Merkmale zu einem semantischen Zusammenhang geführt haben. Insgesamt ist ein regelbasierter Ansatz für die Analyse der Syntax im Rahmen dieser Arbeit daher eine geeignete Wahl.

5.3.1.1 Abhängigkeiten im Quelltext

Abhängigkeitsmerkmale bilden die Grundlage der syntaxbasierten Quelltextanalyse. Sie beschreiben fundamentale Beziehungen zwischen Programmstellen und liefern hilfreiche Informationen für die Bewertung semantischer Zusammenhänge. Im Rahmen einer statischen Quelltextanalyse sind vor allem Typ- und Methodenreferenzen relevant, da Objektinformationen erst zur Laufzeit vorliegen. Eine Schwierigkeit hierbei ist, dass der tatsächliche Typ einer Variable oder die tatsächlich aufgerufene Methode in einigen Fällen, aufgrund von Polymorphie, erst zur Laufzeit ermittelt werden kann. Bei der statischen Quelltextanalyse liegt lediglich der Deklarationstyp einer Variable vor. Allerdings können gemeinsame Referenzen auf Basistypen oder Schnittstellen auch ohne Wissen über die konkreten Typen zur Identifikation semantischer Zusammenhänge beitragen.
Eine Besonderheit im Hinblick auf Typreferenzen sind parametrisierte Typen. Ein parametrisierter Typ besitzt Argumente, die generische Typen repräsentieren und erst bei der Instanziierung des Typs angegeben werden. Ein Beispiel hierfür ist die Klasse `ArrayList<E>`, wobei `E` einen generischen Typ repräsentiert, der z.B. bei der Objekterzeugung durch Angabe des Typs `Employee` wie in `new ArrayList<Employee>()` spezifiziert wird.

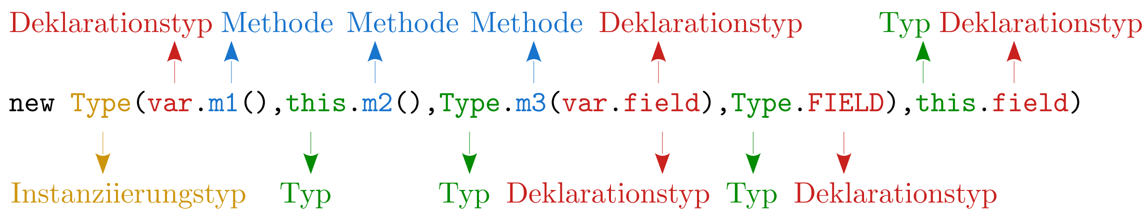


Abbildung 5.2: Abhängigkeitsmerkmale eines Ausdrucks.

Bei der Abhängigkeit zu einem Typ sollten Typargumente berücksichtigt werden, sodass eine Variable vom Typ `List<Employee>` zu einer Abhängigkeit zum Typ `Employee` führt. Sowohl Typreferenzen als auch Methodenreferenzen können auf Programmelemente verweisen, die nicht im Quelltext definiert sind, sondern aus externen Quellen importiert werden. Extern definierte Programmelemente wie die Klasse `String` besitzen keine Entsprechung in den Anforderungen und sind für die Suche nach semantischen Zusammenhänge in dieser Arbeit nicht relevant.
Auf dieser allgemeinen Betrachtung von Typ- und Methodenreferenzen aufsetzend, werden im Folgenden die Abhängigkeitsmerkmale von Anweisungen, Methoden, Typen und Paketen im Detail betrachtet.

Anweisungsebene

Anweisungen sind elementare Bausteine im Quelltext. Fundamentaler Bestandteil vieler Anweisungen sind auswertbare Konstrukte, die einen Wert oder eine Referenz repräsentieren und Ausdruck genannt werden. Im Rahmen dieser Arbeit sind die Elemente eines Ausdrucks interessant, die auf eine Abhängigkeit zu einem Typ oder einer Methode hindeuten. Abbildung 5.2 zeigt einen exemplarischen Ausdruck und kennzeichnet die Abhängigkeiten, die aus den verschiedenen Bestandteilen abgeleitet werden können. Grundsätzlich setzen sich die Abhängigkeiten eines Ausdrucks aus dessen Instanziierungen, Methodenaufrufen, den verwendeten Variablen und den explizit referenzierten Typen zusammen. Eine Instanziierung kann als Abhängigkeit zum instanziierten Typ betrachtet werden. Methodenaufrufe können als Abhängigkeit zur aufgerufenen Methode interpretiert werden. Die Parameter eines Aufrufs oder einer Instanziierung repräsentieren Werte oder Referenzen und können selbst als Ausdrücke aufgefasst werden.
Zugriffe auf lokale Variablen wie `var` in Abbildung 5.2 können als Typabhängigkeiten zum Deklarationstyp der Variable interpretiert werden. Grundsätzlich kann auf Variablen lesend oder schreibend zugegriffen werden. Ein schreibender Zugriff verändert den Wert einer Variable, ein lesender Zugriff nicht. Ob das dahinterliegende Objekt verändert wird oder nicht, kann aus der Art des Variablenzugriffs, im Rahmen einer statischen Quelltextanalyse, nicht abgeleitet werden. Eine Unterscheidung zwischen lesendem und schreibendem Variablenzugriff bringt daher keine erkennbaren Vorteile mit sich, weshalb es sinnvoll erscheint, die Verwendung einer Variable lediglich als Typabhängigkeit zu ihrem Deklarationstyp aufzufassen.
Aus Methodenaufrufen oder Attributzugriffen kann, neben der Abhängigkeit zur Methode bzw. dem Attributtyp, eine weitere Abhängigkeit zum Typ, in dem die Methode oder das Attribut definiert ist, abgeleitet werden. Diese Abhängigkeit wird im Folgenden als Typzugriffsabhängigkeit bezeichnet. Aus dem Methodenaufruf `var.m1()` oder dem Attributzugriff `this.field` aus Abbildung 5.2 können folglich jeweils zwei Abhängigkeiten extrahiert werden. Eine Abhängigkeit zur Methode `m1` bzw. zum Attribut `field` und eine Abhängigkeit zum Typ, den `var` bzw. `this` repräsentieren. Die Verwendung des Schlüsselwortes `this` hat hierbei keine Auswirkungen auf die Abhängigkeiten. Interne Attributzugriffe oder Methodenaufrufe können auch ohne das Schlüsselwort `this` als Abhängigkeit

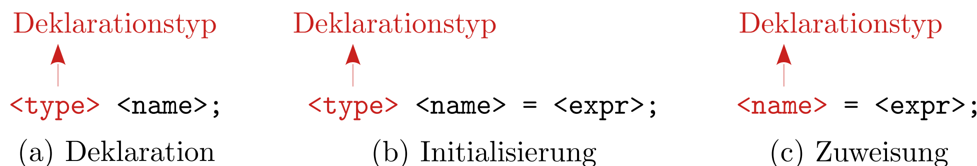
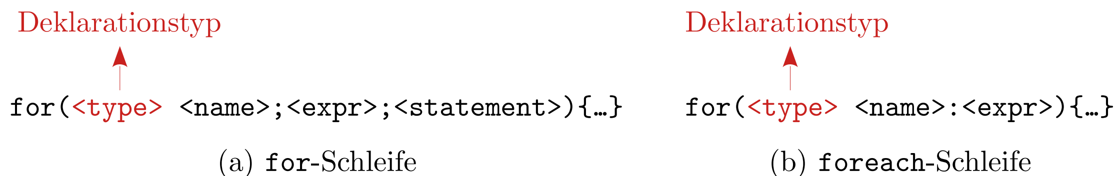


Abbildung 5.3: Abhängigkeiten einer Deklaration, Initialisierung und Zuweisung.

zum Typ, in dem eine Anweisung liegt, erkannt werden. Die Typzugriffsabhängigkeit bei Attributzugriffen ist wichtig, da ein Attribut potentiell einen primitiven Datentyp besitzen kann, wodurch die Abhängigkeit sonst nicht berücksichtigt werden würde. Bei Methodenaufrufen spielt der Typzugriff ebenfalls eine wichtige Rolle, da hierdurch Zusammenhänge zwischen Ausdrücken erkannt werden können, die unterschiedliche Methoden vom selben Typen aufrufen. Ohne die Hinzunahme der Typzugriffsabhängigkeit bei Methodenaufrufen gäbe es diese Abstufung nicht und ein Zusammenhang wäre lediglich aufgrund von Aufrufen derselben Methode möglich. Da im Rahmen einer statischen Quelltextanalyse keine Objektinformationen vorliegen, können Variablen- und Typzugriffe als ein gemeinsames Abhängigkeitsmerkmal betrachtet werden. Infolgedessen kann ein statischer Attributzugriff wie `Type.FIELD`, ebenso wie der Attributzugriff `this.field` durch zwei Typabhängigkeiten beschrieben werden.
Bei Typumwandlungen (engl. type casting) oder Überprüfungen mit dem `instanceof`-Operator werden, genau wie beim statischen Zugriff, Typreferenzen explizit angegeben. Allerdings wird hierbei im Gegensatz zu statischen Zugriffen kein Attribut ausgelesen bzw. keine Methode aufgerufen. Aufgrund dessen ist es sinnvoll Typreferenzen in Typumwandlungen und Typprüfungen als eigenständiges Abhängigkeitsmerkmal zu betrachten. Im Folgenden werden solche Abhängigkeiten als explizite Typreferenzen bezeichnet.
Statische Werte wie Zahlen oder Zeichenketten können ebenfalls Teil eines Ausdrucks sein. Sie beschreiben Abhängigkeiten zu Standard-Datentypen und werden daher bei der syntaxbasierten Quelltextanalyse in dieser Arbeit nicht berücksichtigt.
Insgesamt verdeutlicht Abbildung 5.2, dass ein Ausdruck viele Abhängigkeiten zu unterschiedlichen Programmelementen besitzen kann. Hierbei sind auch mehrere Abhängigkeiten zum gleichen Programmelement möglich, wenn z.B. eine Variable mehrmals im Ausdruck referenziert wird.

Auf der Grundlage der Abhängigkeitsmerkmale von Ausdrücken werden nachfolgend die Abhängigkeitsmerkmale der verschiedenen Anweisungsarten untersucht. Eine Deklaration wie (a) in Abbildung 5.3 dient der Definition von Variablen. Der Deklarationstyp einer Variable kann analog zum Variablen- oder Typzugriff als Typabhängigkeit aufgefasst werden. Bei einer Initialisierung wie es (b) in Abbildung 5.3 zeigt, wird der deklarierten Variable direkt ein Wert zugewiesen. Dieser Wert kann als Ausdruck aufgefasst werden, wodurch eine Initialisierung, neben der Abhängigkeit zum Typ der deklarierten Variable, weitere Abhängigkeiten aufgrund des angegebenen Ausdrucks besitzt. Eine Zuweisung wie es (c) in der Abbildung charakteristisch darstellt, ist einer Initialisierung sehr ähnlich. Der einzige Unterschied besteht darin, dass sie einer bereits deklarierten Variable einen Wert zuweist. Im Hinblick auf die Abhängigkeiten unterscheiden sich die beiden Anweisungen nicht. Der Variablentyp wird bei einer Zuweisung implizit über den Deklarationstyp der zugewiesenen Variable bestimmt.

Methodenaufrufe können sowohl Teil eines Ausdrucks sein als auch eigenständige Anweisungen im Quelltext darstellen. Für die Suche nach semantischen Zusammenhängen sind sie von besonderer Relevanz, da sie den Ausführungspfad beschreiben und die Ausführung verschiedener Programmstellen miteinander verknüpfen. Aus einem Methodenaufruf



(a) `for`-Schleife (b) `foreach`-Schleife

Abbildung 5.4: Abhängigkeiten einer `for`- und `foreach`-Schleife.

können, wie bereits zuvor beschrieben, zwei Abhängigkeiten abgeleitet werden. Eine zur aufgerufenen Methode und die zweite zum Typ, in dem die aufgerufene Methode definiert ist. Aufrufe wie `super()` können als Aufruf der überschriebenen Methode interpretiert werden und unterscheiden sich daher nicht von einem normalen Methodenaufruf. Weitere Abhängigkeiten können aufgrund von Parametern existieren, die jeweils einen eigenständiger Ausdruck darstellen.
Eine spezielle Rolle kommt den Abfragemethoden (engl. getter) und Änderungsmethoden (engl. setter) zu. Sie umfassen im Normalfall keine Logik und dienen lediglich zum Auslesen oder Setzen von Attributwerten eines Typs. Aufrufe solcher Methoden können daher mehr auf einen Zusammenhang zum Typ hindeuten, in dem die Abfrage- oder Änderungsmethode platziert ist und weniger auf einen Zusammenhang zu der aufgerufenen Methode. Allerdings können Abfrage- und Änderungsmethoden zusätzliche Funktionalitäten enthalten, wodurch eine Unterscheidung zu Methoden mit funktionaler Logik nicht immer möglich ist. Hinzu kommt, dass im Rahmen einer statischen Analyse nicht eindeutig bestimmt werden kann was beim Aufruf einer Methode ausgeführt wird, z.B. wenn die Methode eine Verzweigung enthält. Insgesamt erscheint es daher sinnvoll, auf eine gesonderte Behandlung von Abfrage- und Änderungsmethoden zu verzichten, um aufgrund der fehlenden Möglichkeiten zur Bestimmung der Semantik eines Methodenaufrufs eine Verfälschung der Analyseergebnisse zu vermeiden.

Eine spezielle Art von Anweisungen sind Kontrollstrukturen. Sie erlauben eine bedingte oder wiederholte Ausführung von Anweisungsblöcken. Im Hinblick auf die Abhängigkeitsmerkmale kann eine Kontrollstruktur getrennt von den Anweisungen in ihrem Rumpf betrachtet werden. Bei Verzweigungen entscheidet ein boolescher Ausdruck, ob ein Anweisungsblock ausgeführt wird oder nicht. Infolgedessen kann eine Verzweigung für die Analyse ihre Abhängigkeitsmerkmale durch ihren booleschen Ausdruck repräsentiert werden. Eine Switch-Case Anweisung dient ebenfalls der bedingten Ausführung von Anweisungen. Für die Entscheidung, welcher Anweisungsblock ausgeführt wird, wird das Ergebnis eines Switch-Ausdrucks mit einer Reihe von Case-Ausdrücken verglichen. Der Anweisungsblock, dessen Ausdruck mit dem Ergebnis des Switch-Ausdrucks übereinstimmt, wird ausgeführt. Somit können Switch-Case Anweisungen durch die Menge ihrer Ausdrücke beschrieben werden.
Schleifen werden eingesetzt, um Anweisungsblöcke wiederholt auszuführen. Eine `while`- und `do-while`-Schleife führt die Anweisungen in ihrem Rumpf solange aus, bis ein boolescher Ausdruck *falsch* zurück liefert. Demnach können `while`- und `do-while`-Schleifen durch die Abhängigkeiten ihrer Ausdrücke beschrieben werden. Eine `for`-Schleife umfasst neben einem booleschen Ausdruck noch weitere Informationen. Abbildung 5.4 zeigt den charakteristischen Aufbau einer `for`-Schleife auf der linken Seite. Der Schleifenkopf besteht aus drei Teilen. Im ersten Teil findet sich eine Deklaration, aus der eine Abhängigkeit zum Deklarationstyp abgeleitet werden kann. Im zweiten Teil befindet sich ein Ausdruck und der dritte Teil umfasst eine Anweisung, die nach jeder Iteration aufgerufen wird. Insgesamt kann eine `for`-Schleife also durch eine Abhängigkeit zum Typ der Variable, in Verbindung mit den Abhängigkeiten aus dem Ausdruck und der Anweisung beschrieben werden.
Die `foreach`-Schleife verlangt ebenfalls nach einer besonderen Behandlung. Im Gegen-

Deklarationstyp Deklarationstyp

```
try {…} catch(<type>|…|<type> <name>) {…}
```

Abbildung 5.5: Abhängigkeiten einer `try-catch`-Anweisung.

Tabelle 5.1: Überblick der Abhängigkeitsmerkmale einer Anweisung.

Abhängigkeit zu	Merkmal
Methode	Methodenaufruf
	Deklaration, Variablen- oder Typzugriff
Typ	Instanziierung
	Explizite Typreferenz

satz zu den vorherigen Schleifen besitzt sie keinen booleschen Ausdruck, sondern iteriert über eine Menge von Elementen. In Abbildung 5.4 ist die schematische Darstellung einer `foreach`-Schleife aufgeführt. Der Schleifenkopf umfasst die Deklaration einer lokalen Variable und einen Ausdruck, der eine Menge von Werten vom Typ der deklarierten Variable zurückliefert. Im Bezug auf die Abhängigkeiten kann eine `foreach`-Schleife durch die Abhängigkeit zum Deklarationstyp der Variable und den Abhängigkeiten des angegebenen Ausdrucks beschrieben werden.

Eine weitere Besonderheit bei der Betrachtung der Anweisungsebene stellen `try-catch`-Anweisungen dar. Sie erlauben es, auf Ausnahmen, die bei der Ausführung des `try`-Blocks ausgelöst wurden, individuell zu reagieren. In Abbildung 5.5 ist eine abstrakte `try-catch`-Anweisung dargestellt. Welche Ausnahmen zur Ausführung eines `catch`-Blocks führen, wird über den Variablentyp festgelegt. Hierbei kann es vorkommen, dass ein `catch`-Block mehrere Ausnahmen behandelt. Alle behandelten Ausnahmetypen werden, bei der Definition der Variable eines Catch-Blocks, durch einen vertikalen Strich getrennt, angegeben. Darüber hinaus kann eine `try-catch`-Anweisung mehrere `catch`-Blöcke umfassen, die jeweils unterschiedliche Ausnahmen behandeln. Für die Beschreibung der Abhängigkeiten einer `try-catch`-Anweisung können die behandelten Ausnahmetypen herangezogen werden.

Tabelle 5.1 fasst die möglichen Abhängigkeitsmerkmale einer Anweisung zusammen. Die bisherigen Erläuterungen zeigen, dass bereits einfache Anweisungen mehrere Abhängigkeitsbeziehungen zu unterschiedlichen Typen und Methoden aufweisen können. Allerdings ist die isolierte Betrachtung von Anweisungen und deren Abhängigkeiten nur bedingt hilfreich für die Suche nach semantischen Zusammenhängen, da sich der Zweck einer Anweisung erst durch die Einbeziehung ihres Kontextes ergibt. Entwickler fassen zusammengehörende Anweisungen in der Regel in Methoden zusammen. Daher erscheint es sinnvoll, die Abhängigkeiten einer Anweisung bei der Suche nach den semantischen Zusammenhängen der Methode oder des Typs, in der die Anweisung platziert ist, zu berücksichtigten. Hinzu kommt, dass Anforderungen eines Softwaresystems sich meist auf ganze Typen oder Methoden auswirken. Auch im Hinblick auf eine spätere Nutzung bei der Rückverfolgbarkeitsanalyse sind die Abhängigkeiten von Anweisungen daher vor allem für die Identifikation semantischer Zusammenhänge auf Klassen- und Methodenebene relevant.

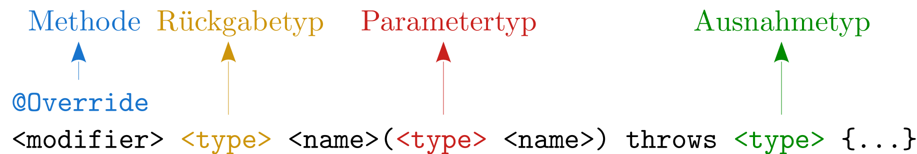


Abbildung 5.6: Abhängigkeiten einer Methodendeklaration.

Methodenebene

Methoden werden dafür genutzt Algorithmen, Funktionalitäten oder Verhaltensweisen zu beschreiben, die sich aus der Ausführung der darin platzierten Anweisungen ergeben. Bei der Beschreibung der Abhängigkeiten einer Methode ist es daher sinnvoll, die Abhängigkeiten der enthaltenen Anweisungen miteinzubeziehen. Darüber hinaus besitzt eine Methode weitere Abhängigkeiten, die aus ihrer Deklaration abgeleitet werden können. Grundsätzlich kann hierbei, wie auch schon bei den Anweisung, zwischen Abhängigkeiten zu Typen und Methoden unterschieden werden. Abbildung 5.6 zeigt den schematischen Aufbau einer Methodendeklaration und kennzeichnet, welche Abhängigkeiten aus ihr folgen können. Eine Abhängigkeit zu einem Typ kann aus einem Parameter-, Ausnahme- oder Rückgabetyp abgeleitet werden. Aufgrund der Übersichtlichkeit beinhaltet die dargestellte Methodendeklaration lediglich einen Parameter und einen Ausnahmetyp. Im Allgemeinen kann eine Methode jedoch mehr als einen Parameter erwarten oder mehr als eine Ausnahme auslösen. Infolgedessen sind mehrere Typabhängigkeiten zu Parameter- und Ausnahmetypen möglich. Methoden die keine Parameter erwarten oder keine Ausnahmen auslösen, weisen dementsprechend weniger Abhängigkeiten auf. Methoden die mit dem Schlüsselwort `void` definiert wurden, besitzen keine Abhängigkeit bezüglich ihres Rückgabetyps. Aufgrund der Annotation `@Override` kann eine Methode darüber hinaus eine Abhängigkeit zur überschriebenen Methode besitzen.
Schlüsselwörter einer Methodensignatur, wie `static` oder `abstract` beschreiben keine Abhängigkeiten. Auch die Sichtbarkeit einer Methode hat keinen Einfluss auf ihre ausgehenden Abhängigkeiten, da sie lediglich den Nutzungskontext einschränkt.
Eine Abhängigkeit die nicht auf der Signatur oder dem Rumpf einer Methode basiert, kann aus der Platzierung der Methode gefolgert werden. Hierbei kann eine Methode sowohl mit dem Typ, in dem sie definiert ist als auch mit anderen Methoden des gleichen Typs in Beziehung gesetzt werden. Ein Zusammenhang zu anderen Methoden aufgrund einer gemeinsamen Platzierung innerhalb eines Typs ist jedoch wenig sinnvoll, da die Methoden eines Typs grundsätzlich völlig unterschiedliche Zwecke besitzen können. Im Gegensatz dazu beschreibt ein Typ in der Regel die allgemeine Absicht zu der eine Methode mit ihrem Zweck beiträgt. Die Einbeziehung von Abhängigkeiten zwischen einem Typ und seinen Methoden ist daher sinnvoll.
In Tabelle 5.2 sind die Abhängigkeitsmerkmale einer Methode zusammengefasst. Die letzte Zeile bezieht hierbei die Abhängigkeiten der, im Rumpf platzierten, Anweisungen mit ein.

Typebene

Eine Granularitätsebene über den Methoden im Quelltext befinden sich die Typen. Ein Typ kann eine Klasse, eine Schnittstelle oder ein Aufzählungstyp repräsentieren. Die verschiedenen Typen unterscheiden sich im Bezug auf ihre Abhängigkeitsmerkmale, weshalb sie im Folgenden separat betrachtet werden.
Abbildung 5.7 zeigt eine schematische Klassendeklaration und markiert die Abhängigkeiten, die daraus abgeleitet werden können. Die verschiedenen Sichtbarkeiten sowie die

Tabelle 5.2: Überblick der Abhängigkeitsmerkmale einer Methode.

Abhängigkeit zu	Merkmal
Methode	Überschreiben
Typ	Ausnahme
	Parameter
	Rückgabe
	Umschließender Typ
Methode oder Typ	Anweisung im Rumpf

Oberklasse Schnittstelle

```
<modifier> class <type_name> extends <type> implements <type> {…}
```

Abbildung 5.7: Abhängigkeiten einer Klassensignatur.

Schlüsselwörter `static`, `final` und `abstract` haben keine Auswirkungen auf die ausgehenden Abhängigkeiten und besitzen für die Beschreibung der Abhängigkeiten eines Typs in dieser Arbeit keine Relevanz. Vorhandene Vererbungs- und Implementierungsbeziehungen können als Typabhängigkeiten aufgefasst werden. Eine Klasse kann maximal eine Oberklasse besitzen, jedoch beliebig viele Schnittstellen implementieren. Weitere Abhängigkeitsmerkmale können aus dem Rumpf einer Klasse abgeleitet werden. Attribute können bspw. als Typabhängigkeit zum Deklarationstyp interpretiert werden. Im Falle von initialisierten Attributen können die Abhängigkeiten des angegebenen Ausdrucks ebenfalls der Klasse zugeschrieben werden.
Da Konstruktoren ausschließlich bei der Objekterzeugung aufgerufen werden, erscheint es sinnvoll, ihre Abhängigkeiten der Klasse zuzuordnen. Im Hinblick auf seine Abhängigkeitsmerkmale kann ein Konstruktor ähnlich wie eine Methode behandelt werden. Aus den geforderten Parametern und den möglichen Ausnahmen werden Typabhängigkeiten abgeleitet. Durch die Anweisungen im Rumpf eines Konstruktors, existieren darüber hinaus weitere Abhängigkeiten. Neben den Konstruktoren können statische Quelltextblöcke Anweisungen enthalten. Sie dienen bspw. der statischen Initialisierung einer Klasse, weshalb es sinnvoll ist, die Abhängigkeiten der darin platzierten Anweisungen ebenfalls als Abhängigkeiten der Klasse zu interpretieren.
Ein weiterer Bestandteil von Klassen sind Methoden. Sie beschreiben die Funktionalitäten einer Klasse, weshalb es sinnvoll einen Zusammenhang zwischen einer Klasse und den darin definierten Methoden anzunehmen. Des Weiteren können Klassen Typdefinitionen in ihrem Rumpf enthalten. Die Definition innerer Typen ist nicht auf eine bestimmte Art von Typ beschränkt, sodass eine Klasse die Definition einer oder mehrerer Klassen, Schnittstellen oder Aufzählungstypen beinhalten kann. Häufig sind innere Typen vor allem im Kontext des umschließenden Typen relevant, weshalb ihnen eine Abhängigkeit zu diesem zugesprochen werden kann. Umgekehrt kann eine Abhängigkeit zwischen einem Typ und seinen inneren Typen angenommen werden.

Aufzählungstypen sind im Hinblick auf ihre Abhängigkeitsmerkmale den Klassen sehr ähnlich. Implementierte Schnittstellen führen, genau wie bei Klassen, zu Typabhängigkeiten. Da ein Aufzählungstyp keinen anderen Typ erweitern kann, existiert dieses Abhängigkeitsmerkmal bei Aufzählungstypen nicht. Die Werte eines Aufzählungstyps sind entweder konstante Zeichenketten oder repräsentieren Konstruktoraufrufe des Aufzählungstyps. Konstante Zeichenketten spielen bei der Beschreibung der ausgehenden Abhängigkeiten

Tabelle 5.3: Überblick der Abhängigkeitsmerkmale eines Typs.

Abhängigkeit zu	Merkmal
Methode	Methodendefinition
Typ	Attributtyp Erweiterung oder Implementierung Innerer Typ Umschließender Typ
Methode oder Typ	Attributinitialisierung Statischer Quelltextblock Konstruktor

keine Rolle. Da der Aufruf des eigenen Konstruktors eine Abhängigkeit mit sich selbst beschreibt, liefert diese Information ebenfalls keinen Mehrwert bei der Suche nach semantischen Zusammenhängen. Allerdings können bei Konstruktoraufrufen Parameter angegeben sein, die Abhängigkeiten zu anderen Programmelementen besitzen. Infolgedessen werden die Parameter bei der Initialisierung der Werte eines Aufzählungstyps als Ausdrücke aufgefasst und ihre Abhängigkeiten dem Aufzählungstyp zugeordnet. Darüber hinaus kann ein Aufzählungstyp Attribute, Konstruktoren, statische Quelltextblöcke, Methoden und innere Typdefinitionen besitzen. Diese können auf die gleiche Weise wie zuvor bei Klassen interpretiert werden.
Eine Schnittstelle ist von allen Typen am stärksten in der Vielfalt ihrer Abhängigkeitsmerkmale eingeschränkt. Schnittstellen können keine Konstruktoren oder statische Quelltextblöcke besitzen. Im Gegensatz zu Klassen und Aufzählungstypen können sie jedoch andere Schnittstellen erweitern. Demzufolge kann eine Schnittstelle Typabhängigkeiten zu mehreren anderen Schnittstellen besitzen. Eine Implementierungsbeziehung zu anderen Schnittstellen ist hingegen nicht möglich. Im Hinblick auf die Abhängigkeitsmerkmale von Attributen, Methoden und inneren Typdefinitionen gibt es keine Unterschiede zu Klassen und Aufzählungstypen.

Eine Zusammenfassung der möglichen Abhängigkeitsmerkmale eines Typs ist in Tabelle 5.3 aufgeführt. Aufgrund der enthaltenen Anweisungen bzw. Ausdrücke können aus Konstruktoren, Attributinitialisierungen und statischen Quelltextblöcken sowohl Abhängigkeiten zu Typen, als auch zu Methoden abgeleitet werden. Diese Merkmale sind im dritten Block der Tabelle aufgeführt.

Paketebene

Ergänzend zu den bisher betrachteten Abhängigkeitsmerkmalen kann die Paketstruktur Hinweise auf semantische Zusammenhänge liefern. In gut strukturierten Softwareprojekten beschreiben Pakete logisch oder funktional zusammenhängende Mengen von Typen. Infolgedessen ist es sinnvoll, die Paketstruktur als weiteres Abhängigkeitsmerkmal zwischen Typen zu berücksichtigen. Typen im selben Paket besitzen hinsichtlich der Paketstruktur vermutlich den stärksten Zusammenhang. Je stärker sich die Paketpfade zweier Typen unterscheiden, desto schwächer kann ihre Paketabhängigkeit bewertet werden. Im Gegensatz zu den zuvor betrachteten Abhängigkeitsmerkmalen kann die Paketabhängigkeit berechnet werden. Misra et al. nutzen in ihrer Arbeit [MAK$^+$12] die Gleichung 5.1 zur Berechnung der Paketabhängigkeit zweier Typen t_1 und t_2. Mit P_t wird die Menge der Pakete auf dem Pfad zum Typ t bezeichnet. Bei Typen desselben Pakets ist das Ergebnis von Gleichung 5.1

gleich 1. Typen, deren Paketpfade sich nicht überschneiden, werden mit 0 bewertet.

$$dep_{package}(t_1, t_2) = \frac{|P_{t_1} \cap P_{t_2}|}{|P_{t_1} \cup P_{t_2}|} \tag{5.1}$$

Die hierarchische Struktur der Pakete in einem Softwareprojekt kann dazu führen, dass Pakete in allen Paketpfaden vorkommen. Ein Beispiel hierfür ist das Paket, welches die Wurzel der Paketstruktur darstellt. Bei der Existenz solcher Pakete kann die Paketabhängigkeit aus Gleichung 5.1 den Wert 0 nicht erreichen, da die Schnittmenge der Paketpfade niemals leer ist. Für die Suche nach semantischen Zusammenhängen besitzen Pakete, welche in allen Paketpfaden enthalten sind, kaum eine Aussagekraft, da sie nicht zur Unterscheidung von Paketpfaden beitragen. Somit bietet es sich an, diesen Paketen bei einer Einbeziehung der Paketpfade als Abhängigkeitsmerkmal zwischen Typen keine Beachtung zu schenken.

Darüber hinaus wäre es denkbar, Abhängigkeiten zwischen Paketen und Typen oder Methoden zu berücksichtigen. Allerdings gibt es bis auf den Definitionsort eines Programmelements keine direkten Hinweise auf Abhängigkeiten zwischen einem Paket und einem Programmelement. Aus diesem Grund müssten Abhängigkeiten zu einem Paket aus den Abhängigkeiten eines Programmelements zu anderen Programmelementen gefolgert werden. Aus der Abhängigkeit einer Methode m_1 zu einer Methode m_2 könnte bspw. eine Abhängigkeit von m_1 zum Paket, in dem die Methode m_2 liegt, abgeleitet werden. Dies ist jedoch mit einer großen Unsicherheit verbunden, da ein Paket viel abstrakter und allgemeiner als ein konkretes Programmelement ist und weitere Funktionalitäten umfassen kann, die nicht direkt etwas mit der spezifischen Absicht von m_1 gemeinsam haben. Infolgedessen ist die Einbeziehung von Abhängigkeiten zwischen Paketen und Typen oder Methoden nicht sinnvoll.
Des Weiteren können Abhängigkeiten zwischen Paketen untersucht werden. Enthält ein Paket `p1` bspw. einen Typ, der eine Abhängigkeit zu einem Typ in Paket `p2` aufweist, kann dies als Abhängigkeit von `p1` zu `p2` interpretiert werden. Solche Schlussfolgerungen sind jedoch ebenfalls mit großer Unsicherheit verbunden. Hinzu kommt, dass Anforderungen sich in der Regel auf Methoden oder Typen und nicht auf ganze Pakete beziehen.

5.3.1.2 Diskussion verwandter Arbeiten

Insgesamt zeigen die vorangestellten Beschreibungen, dass es eine große Menge an Abhängigkeitsmerkmalen auf verschiedenen Granularitätsebenen und zwischen verschiedenen Programmelementen gibt. Allesamt besitzen das Potential, wertvolle Hinweise bei der Suche nach semantischen Zusammenhängen zu liefern. Im Folgenden sollen verschiedene Ansätze zur Bewertung der Abhängigkeiten im Quelltext diskutiert und Entwurfsanforderungen für die syntaxbasierte Quelltextanalyse in dieser Arbeit abgeleitet werden.

Einige Ansätze, wie die Arbeit [SM11] von Scanniello und Marcus, beschränken sich bei der Suche nach Zusammenhängen auf Methodenaufrufe als einziges Abhängigkeitsmerkmal. Ein semantischer Zusammenhang zwischen zwei Methoden wird aufgrund der Existenz eines Aufrufs zwischen ihnen abgeleitet. Scanniello und Marcus berücksichtigen dabei nicht, wie viele Aufrufe von einer Methode insgesamt aus- oder eingehen. Dies kann als Nachteil aufgefasst werden, da der Aufruf einer Methode, die zusätzlich von vielen anderen Methoden aufgerufen wird, vermutlich eine geringere Aussagekraft bei der Suche nach semantischen Zusammenhängen besitzt, als ein exklusiver Methodenaufruf. Ein weiterer Nachteil ihres Ansatzes ist, dass semantische Zusammenhänge zwischen Methoden, die keine Aufrufabhängigkeit zueinander aufweisen, nicht erkannt werden können.
Auch Gharibi et al. nutzen in ihrem Ansatz [GAL18] ausschließlich Aufrufabhängigkeiten. Im Gegensatz zu Scanniello und Marcus, analysieren sie Ausführungspfade. Ihre Arbeit

beruht auf der Annahme, dass alle Methoden entlang eines Ausführungspfades eine gemeinsame Absicht verfolgen. Ausführungspfade, die eine große Überschneidung im Bezug auf die enthaltenen Methoden besitzen, ordnen sie der gleichen Absicht zu und erhalten somit eine grobe Unterteilung des Quelltextes. Im Vergleich zum Ansatz von Scanniello und Marcus besitzt der Ansatz von Gharibi et al. den Vorteil, dass semantische Zusammenhänge zwischen Methoden vorliegen können, ohne dass direkte Aufrufabhängigkeiten zwischen ihnen existieren. Durch die Abstraktion auf Ausführungspfade sind die gefundenen Zusammenhänge jedoch sehr allgemein und spezifische Zusammenhänge zwischen einzelnen Methoden können nicht erkannt werden.
Durch die Beschränkung auf Methodenaufrufe als einziges Abhängigkeitsmerkmal, werden viele Informationen im Quelltext ignoriert. Parameter-, Rückgabe- und Ausnahmetypen können bspw. ebenfalls Hinweise auf semantische Zusammenhänge liefern. Dass die Berücksichtigung von lediglich einem Abhängigkeitsmerkmal nicht optimal ist, zeigen die Evaluationsergebnisse von Scanniello und Marcus. Ihr Ansatz wurde in fünf Softwareprojekten herangezogen, um die enthaltenen Methoden hinsichtlich ihrer Relevanz für gegebene Fehlerbeschreibungen zu sortieren. Die erste tatsächlich relevante Methode befand sich durchschnittlich auf einem Rang im Bereich zwischen 524 und 639. Um ein breiteres Spektrum der verfügbaren Informationen im Quelltext ausnutzen und Zusammenhänge auf unterschiedlichen Merkmalsebenen erkennen zu können, ist es für die vorliegende Arbeit sinnvoll, eine Kombination verschiedener Abhängigkeitsmerkmale bei der Quelltextanalyse zu berücksichtigen.

Entwurfsanforderung 5.1: Kombination von Abhängigkeitsmerkmalen

Bei der syntaxbasierten Quelltextanalyse soll ein breites Spektrum von Abhängigkeitsmerkmalen im Quelltext berücksichtigt werden, um unterschiedliche Ausprägungsformen semantischer Zusammenhänge erkennen zu können.

In der Arbeit [MAK$^+$12] von Misra et al. werden zusätzlich zu den Methodenaufrufen die Vererbungsstruktur und die Typabhängigkeiten berücksichtigt, um Zusammenhänge zwischen Klassen zu finden. Bei der Bewertung von Abhängigkeiten zwischen Klassen beziehen Misra et al. die generelle Abhängigkeitsstruktur mit ein. Je weniger Abhängigkeiten eine Klasse besitzt, desto stärker wird die Aussagekraft einzelner Abhängigkeiten eingestuft. Dies steht im Einklang mit dem Konzept der Spezifität aus [Rob08], bei dem eine exklusive Referenz auf ein Programmelement als stärkeres Indiz für einen Zusammenhang aufgefasst wird, als eine Referenz auf ein Programmelement, auf das noch viele weitere Programmelemente referenzieren. Allerdings kann eine Normierung durch bspw. die Gesamtzahl an eingehenden Abhängigkeiten auch Nachteile mit sich bringen. Bei Klassen die das Datenmodell beschreiben und naturgemäß häufig im Quelltext referenziert werden, könnte es hierdurch zu einer Unterbewertung der Abhängigkeiten kommen.
Für die Bewertung des strukturellen Zusammenhangs zweier Klassen, wird der Vererbungszusammenhang mit der Typ- und Aufrufabhängigkeit verrechnet. Misra et al. nutzen hierbei eine Gleichgewichtung und geben an, dass dies für unbekannte Softwareprojekte eine gute Wahl ist, da sich Zusammenhänge prinzipiell auf verschiedene Merkmalsebenen auswirken. Allerdings bleibt offen, welche positiven Auswirkungen eine individuelle Gewichtung von Abhängigkeitsmerkmalen auf die Güte der Ergebnisse haben kann. Da ein Methodenaufruf verglichen mit einer Typreferenz vermutlich eine höhere Aussagekraft besitzt, erscheint die Möglichkeit einer individuellen Gewichtung von Abhängigkeitsmerkmalen in dieser Arbeit sinnvoll.

Entwurfsanforderung 5.2: Gewichtung von Abhängigkeitsmerkmalen

Bei der Suche nach semantischen Zusammenhängen soll es möglich sein, die berücksichtigten Abhängigkeitsmerkmale unterschiedlich zu gewichten.

Die Gewichtung von Abhängigkeitsmerkmalen spielt in der Arbeit [AC17] von Amarjeet und Chhabra eine zentrale Rolle. Im Vergleich zu [MAK+12], differenzieren sie zwischen verschiedenen Arten von Typreferenzen wie Parameter, Rückgaben, Ausnahmen, Attributen und Instanziierungen. Um geeignete Gewichte für die Abhängigkeitsmerkmale zu bestimmen, analysieren Amarjeet und Chhabra deren Ausprägung in bestehenden Softwareprojekten. Die Übertragbarkeit ihrer Gewichte auf die vorliegende Arbeit ist jedoch fragwürdig, da sich ihre Arbeit der Remodularisierung von Softwareprojekten widmet. Dennoch wäre es denkbar, ihr Vorgehen für eine Suche nach geeigneten Gewichten zu adaptieren. Mit Hilfe einer manuell erstellten Musterlösung der bestehenden semantischen Zusammenhänge in einem Softwareprojekt könnte eine optimierte Gewichtskonfiguration für die betrachteten Abhängigkeitsmerkmale iterativ durch einen Vergleich der gefundenen semantischen Zusammenhänge mit der Musterlösung angenähert werden. Für den Entwurf der syntaxbasierten Quelltextanalyse folgt daher Entwurfsanforderung 5.3. Daran anknüpfend könnten ideale Gewichtskonfigurationen für verschiedene Stereotypen von Softwareprojekten angenähert werden. Bei der Analyse unbekannter Softwareprojekte kann daraufhin, durch die Zuordnung zu einem Stereotyp, eine passende Gewichtsverteilung abgeleitet werden. Die Zuordnung eines Softwareprojekts zu einem Stereotyp könnte bspw. durch einen Vergleich verschiedener Softwaremetriken erfolgen.

Entwurfsanforderung 5.3: Finden einer passenden Gewichtskonfiguration

Die Gewichte der eingesetzten Abhängigkeitsmerkmale sollen durch einen Vergleich der gefundenen semantischen Zusammenhänge mit einer Musterlösung optimiert werden.

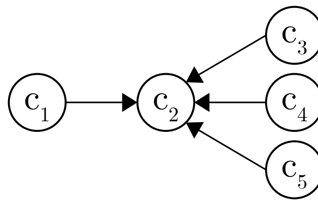
Bei einer gegebenen Gewichtsverteilung berechnen Amarjeet und Chhabra den Zusammenhang zwischen Klassen in zwei Schritten. Im ersten Schritt werden die direkten Abhängigkeiten zwischen allen Klassen im Quelltext bewertet, sodass eine Klasse auf einen Vektor abgebildet werden kann, dessen Elemente die Abhängigkeit zu jeder anderen Klasse beschreiben. Im zweiten Schritt wird der Zusammenhang zweier Klassen durch Berechnung des Kosinus des Winkels zwischen ihren Vektorabbildungen bestimmt. Im Vergleich zu den bisher betrachteten Ansätzen können somit Zusammenhänge zwischen Klassen erkannt werden, die nicht zwangsläufig direkte Abhängigkeiten zueinander aufweisen, dafür jedoch im gleichen Kontext verwendet werden.
Gleichung 5.2 zeigt die Formel aus [AC17], welche im ersten Schritt eingesetzt wird, um die direkten Abhängigkeiten zwischen zwei Klassen c_1 und c_2 zu bewerten. Die Menge der betrachteten Abhängigkeitsmerkmale wird durch R festgelegt. Die Anzahl der Abhängigkeiten einer Klasse c_1 zu einer Klasse c_2, im Bezug auf ein konkretes Abhängigkeitsmerkmal r, wird mit $n_r(c_1, c_2)$ beschrieben. Existieren keine direkten Abhängigkeiten einer Klasse c_1 zu einer Klasse c_2 bzw. von c_2 zu c_1, wird der entsprechende Summand auf 0 gesetzt. Die Gesamtzahl an Klassen eines Softwaresystems entspricht $|C|$ und die Anzahl an Klassen, die mindestens eine strukturelle Abhängigkeit zu einer Klasse c_1 aufweisen, wird mit n_{c_1} gekennzeichnet. Des Weiteren repräsentiert der Faktor w_r das Gewicht eines Abhängigkeitsmerkmals r.

$$dep(c_1, c_2) = \frac{\sum_{r \in R} w_r * n_r(c_1, c_2)}{\sum_{c \in C} \sum_{r \in R} w_r * n_r(c, c_2)} * log \frac{|C|}{n_{c_2}} + \frac{\sum_{r \in R} w_r * n_r(c_2, c_1)}{\sum_{c \in C} \sum_{r \in R} w_r * n_r(c, c_1)} * log \frac{|C|}{n_{c_1}} \quad (5.2)$$

Ein Vorteil der Formel ist die Flexibilität im Bezug auf die Anzahl der berücksichtigten Abhängigkeitsmerkmale im Quelltext. Darüber hinaus bezieht die Formel die Gesamtzahl an eingehenden Abhängigkeiten in c_2 bzw. c_1 bei der Berechnung mit ein. Abhängigkeiten zu Klassen, die wenig eingehende Abhängigkeiten von anderen Klassen besitzen, erhalten demnach eine größere Aussagekraft. Allerdings wird bei der Berechnung nicht berücksichtigt, wie viele ausgehende Abhängigkeiten eine Klasse besitzt. Dies kann zu einer Unterbewertung von Abhängigkeiten, wie in Beispiel 5.2 beschrieben, führen.

Beispiel 5.2: Unterbewertung von Abhängigkeiten

Gegeben sei der folgende Graph, dessen Knoten die Programmstellen eines Softwaresystems repräsentieren und dessen Kanten Aufrufabhängigkeiten darstellen. Darüber hinaus existieren keine weiteren Programmstellen oder Abhängigkeiten.



Die Programmstelle c_2 wird sowohl von c_1 als auch von c_3, c_4 und c_5 aufgerufen und besitzt daher vier eingehende Kanten. Welches Programmelement den stärksten Bezug zu c_2 hat, kann aufgrund der Darstellung nicht ausgesagt werden.
Gleichung 5.2 bewertet alle direkten Zusammenhänge zu c_2 mit 0,08. Da c_2 mehrere eingehende Abhängigkeiten besitzt, werden die einzelnen Abhängigkeiten schwächer bewertet. Hierbei bleibt die Tatsache unberücksichtigt, dass $c_{1,3,4,5}$ lediglich eine ausgehende Abhängigkeit besitzen und somit ausschließlich zu c_2 abhängig sind. Insgesamt erscheint die Bewertung von 0,08 daher zu niedrig.

Im zweiten Schritt des Ansatzes von Amarjeet und Chhabra, wird eine Klasse c_1 auf einen Vektor $\vec{v} = (v_1, ..., v_n)$ abgebildet, dessen Elemente die, durch Gleichung 5.2 berechnete, Abhängigkeit von c_1 zu allen anderen Klassen $c_1, ..., c_n$ im Quelltext beschreiben. Der finale Zusammenhang zweier Klassen entspricht dem Kosinus des Winkels zwischen den Vektorabbildungen zweier Klassen. Gleichung 5.3 aus [AC17] zeigt die Formel hierfür.

$$sim(\vec{v}, \vec{w}) = \frac{\sum_{k=1}^{n}(v_k * w_k)}{\sqrt{\sum_{k=1}^{n}(v_k)^2} * \sqrt{\sum_{k=1}^{n}(w_k)^2}} \tag{5.3}$$

Im Kontext dieser Arbeit kann die Ähnlichkeit der Abhängigkeiten zweier Programmelemente ebenfalls als Indiz für einen semantischen Zusammenhang aufgefasst werden. Ein Beispiel liefern die beiden Klassen `Kreditkarte` und `Lastschrift`. Sie implementieren Zahlungsmittel und werden im Quelltext vermutlich in den gleichen Kontexten verwendet. Offensichtlich besitzen sie einen semantischen Zusammenhang, aber nicht zwangsläufig eine direkte Abhängigkeit zueinander. Aufgrund der ähnlichen Verwendung kann ein Vergleich wie in Gleichung 5.3 eine hohe Ähnlichkeit zwischen ihnen feststellen. Somit können Zusammenhänge auch ohne eine direkte Abhängigkeit zwischen Quelltextstellen erkannt werden.
Bei der Bewertung der Abhängigkeiten zweier Klassen über Gleichung 5.3 verliert die direkte Abhängigkeit zwischen den betrachteten Klassen an Bedeutung. Zwei Klassen c_1 und c_2, die eine starke direkte Abhängigkeit zueinander aufweisen, sich jedoch im Bezug auf ihre direkten Abhängigkeiten zu den restlichen Klassen im Quelltext drastisch unterscheiden,

wird ein niedrigerer Zusammenhang zugesprochen, als wenn beide Klassen eine schwächere direkte Abhängigkeit besitzen, dafür jedoch die gleichen Abhängigkeiten zu anderen Klassen aufweisen. Dies kann der Tatsache geschuldet sein, dass die Arbeit von Amarjeet und Chhabra aus dem Bereich der Remodularisierung von Softwaresystemen stammt. Bei der Suche nach semantischen Zusammenhängen sind sowohl direkte Abhängigkeiten, als auch Ähnlichkeiten der Abhängigkeitsstruktur gleichermaßen relevant.

Entwurfsanforderung 5.4: Direkte und indirekte Zusammenhänge

Bei der Suche nach semantischen Zusammenhängen sollen sowohl direkten als auch indirekte Abhängigkeiten zwischen Programmelementen berücksichtigt werden.

Kuang et al. bewerten Zusammenhänge zwischen Klassen in [KNH+17] ebenfalls anhand von direkten und indirekten Abhängigkeiten. Im Gegensatz zum Ansatz von Amarjeet und Chhabra nutzen sie lediglich Vererbungsbeziehungen, Attributtypen und Methodenaufrufe als direkte Abhängigkeitsmerkmale. Dies kann als Nachteil aufgefasst werden, da weitere Typreferenzen wie Instanziierungen und Variablentypen auf Anweisungsebene unberücksichtigt bleiben und somit potentielle Hinweise auf semantische Zusammenhänge außen vor gelassen werden.
Gleichung 5.4 zeigt die Formel zur Berechnung des direkten Zusammenhangs zweier Klassen c_1 und c_2. Sie stammt aus [KNH+17] und wurde entsprechend der hier verwendeten Notation angepasst. Die Anzahl an verschiedenen Abhängigkeiten zwischen c_1 und c_2 beschreibt der Ausdruck $n(c_1, c_2)$. Die Gesamtzahl an Klassen im Quelltext entspricht $|C|$. Qualitativ formuliert werden alle direkten Abhängigkeiten zwischen c_1 und c_2 gezählt, verdoppelt und anschließend durch die Summe aller ausgehenden Abhängigkeiten von c_1 und aller eingehenden Abhängigkeiten von c_2 geteilt.

$$direct(c_1, c_2) = \frac{2 * n(c_1, c_2)}{\sum_{c \in C} n(c, c_2) + \sum_{c \in C} n(c_1, c)} \tag{5.4}$$

Im Gegensatz zu den bisher vorgestellten Ansätzen, besitzt der Zusammenhang zweier Klassen bei Kuang et al. eine Richtung. Dennoch wäre eine Adaption ihres Vorgehens in dieser Arbeit denkbar. Voraussetzung hierfür ist eine Erweiterung von Gleichung 5.4, sodass Abhängigkeiten in beide Richtungen bei der Berechnung berücksichtigt werden. Hierzu kann bspw. der Mittelwert der berechneten Abhängigkeit beider Richtungen verwendet werden.
Ein Vorteil von Gleichung 5.4 ist, dass das Ergebnis auf den Bereich zwischen 0 und 1 normiert ist, wodurch eine Vergleichbarkeit der berechneten Zusammenhänge gewährleistet ist. Im Gegensatz zum Ansatz in [AC17] berücksichtigen Kuang et al. neben der Anzahl an eingehenden Abhängigkeiten auch die Anzahl ausgehender Abhängigkeiten bei der Normierung. Eine Abhängigkeit von c_1 zu c_2 wird folglich niedriger bewertet, je mehr ausgehende Abhängigkeiten c_1 bzw. je mehr eingehende Abhängigkeiten c_2 besitzen. Dies erscheint grundsätzlich sinnvoll, da eine Abhängigkeit eines Programmelements, welches lediglich eine ausgehende Abhängigkeit besitzt, vermutlich ein stärkeres Indiz für einen semantischen Zusammenhang ist, als eine Abhängigkeit eines Programmelements, welches zusätzlich noch viele weitere ausgehende Abhängigkeiten besitzt.
Ein Nachteil der Vorgehensweise von Kuang et al. zeigt sich bei der Betrachtung von Programmelementen wie den Klassen des Datenmodells, die in der Regel viele Abhängigkeiten zu unterschiedlichen Programmelementen aufweisen. Durch die Normierung über die Summe aus allen ein- und ausgehenden Abhängigkeiten, erhalten diese Programmelemente über Gleichung 5.4 in der Regel kleine Werte (Siehe Beispiel 5.3).

Beispiel 5.3: Benachteiligung von vielen Abhängigkeiten

Die Klasse c_1 besitzt insgesamt 20 eingehende Abhängigkeiten. Die Klasse c_2 hat drei ausgehende Abhängigkeiten zu c_1 und keine weiteren Abhängigkeiten. Des Weiteren existieren 17 Klassen, die jeweils eine ausgehende Abhängigkeit zu c_1 besitzen.
Über Gleichung 5.4 würde die Abhängigkeit von c_2 zu c_1 mit $\frac{2*(3)}{3+20} \approx 0,26$ bewertet werden. Die vielen eingehenden Abhängigkeiten von c_1 haben zur Folge, dass der Wert, obwohl c_2 ausschließlich Abhängigkeiten zu c_1 besitzt, relativ niedrig ist.

Alternativ kann für die Normierung der Abhängigkeit zweier Programmelemente die Summe der maximalen Anzahl an eingehenden und ausgehenden Abhängigkeiten zu einem beliebigen anderen Programmelement herangezogen werden. Hierdurch würde eine Abhängigkeit mit dem Wert 1 bewertet werden, wenn es kein anderes Programmelement gibt, zu dem c_1 mehr ausgehende Kanten und c_2 mehr eingehende Abhängigkeiten besitzt. Ein Vorteil dieser Normierung ist, dass die Bewertung relativ zur stärksten Abhängigkeit eines Programmelements erfolgt und somit auch Programmelemente mit vielen Abhängigkeiten adäquat behandelt werden können. Im Hinblick auf den Entwurf der syntaxbasierten Quelltextanalyse sollte daher darauf geachtet werden, dass Programmelemente, die naturgemäß viele Abhängigkeiten aufweisen, bei der Berechnung nicht benachteiligt werden.

Entwurfsanforderung 5.5: Programmelemente mit vielen Abhängigkeiten

Programmelemente mit vielen Abhängigkeiten sollen bei der Berechnung des direkten Zusammenhangs nicht unterbewertet werden.

Eine weitere Schwäche von Gleichung 5.4 zeigt sich bei einer unausgewogenen Ausprägung der betrachteten Abhängigkeitsmerkmale im Quelltext. Weniger häufig vorkommende Abhängigkeiten werden hierbei von den stark vertretenen Abhängigkeitsmerkmalen dominiert. Der Grund dafür liegt in der gemeinsamen Betrachtung der Abhängigkeitsmerkmale über $n(c_i, c_j)$. In Beispiel 5.4 wird dies veranschaulicht.

Beispiel 5.4: Ungleiche Behandlung von Abhängigkeitsmerkmalen

Gegeben seien zwei Klassen c_1 und c_2. Die Klasse c_1 besitzt eine Typreferenz zu c_2 und darüber hinaus einen Methodenaufruf zu einer Methode in c_2. Insgesamt besitzt c_1 fünf ausgehende Typreferenzen, jedoch nur eine ausgehende Aufrufabhängigkeit. Die Klasse c_2 besitzt insgesamt fünf eingehende Typreferenzen und lediglich eine eingehende Aufrufabhängigkeiten.
Die Berechnung der direkten Abhängigkeit von c_1 zu c_2 mit Gleichung 5.4 liefert ein Ergebnis von $\frac{2*(1+1)}{6+6} \approx 0,33$. Wird der Methodenaufruf bei der Berechnung außen vor gelassen, liegt die direkte Abhängigkeit bei $\frac{2*1}{5+5} \approx 0,2$. Bleiben hingegen die Typreferenzen bei der Berechnung unbeachtet, liegt die direkte Abhängigkeit bei $\frac{2*1}{1+1} \approx 1$. Bei einer gemeinsamen Betrachtung der beiden Abhängigkeitsmerkmale wird das Ergebnis demzufolge maßgeblich von den Typreferenzen beeinflusst, wohingegen der Methodenaufruf kaum einen Einfluss besitzt.

Um zu verhindern, dass die Ausprägung einzelner Abhängigkeitsmerkmale zur Unterbewertung von Abhängigkeiten führt, bietet es sich an, den direkten Zusammenhang für jedes Abhängigkeitsmerkmal individuell zu berechnen und anschließend die Ergebnisse durch bspw. einen gewichteten Mittelwert zusammenzufassen.

Entwurfsanforderung 5.6: Berechnung direkter Abhängigkeiten

Die direkte Abhängigkeit soll für jedes Abhängigkeitsmerkmal separat berechnet und anschließend zusammengefasst werden, um eine Unterdrückung einzelner Abhängigkeitsmerkmale zu vermeiden.

Neben den direkten Abhängigkeiten untersuchen Kuang et al. indirekte Abhängigkeiten. Eine indirekte Abhängigkeit zwischen zwei Klassen c_1 und c_2 besteht in ihrer Arbeit, wenn sowohl c_1 als auch c_2 auf dieselben Typen referenzieren. Repräsentiert wird eine indirekte Abhängigkeit durch die Menge gemeinsam referenzierter Typen. Hierbei wird die Seltenheit mit der ein Typ im Allgemeinen referenziert wird berücksichtigt. Ein Typ, der nur selten Teil einer indirekten Abhängigkeit ist, erhöht die indirekte Abhängigkeit stärker, als ein Typ der häufig referenziert wird. Gleichung 5.5 zeigt die in [KNH+17] verwendete Formel zur Berechnung der Relevanz eines Typs t in Abhängigkeit von N, der Gesamtzahl indirekter Abhängigkeiten im Quelltext und n_t, der Anzahl indirekter Abhängigkeiten, die den Typ t beinhalten.

$$idtf(t) = log\frac{N}{n_t} \tag{5.5}$$

Die Relevanz eines Typs wird in [KNH+17] anschließend genutzt, um mit Gleichung 5.6 die indirekte Abhängigkeit zwischen zwei Klassen c_1 und c_2 zu berechnen. Mit DT_{c_1} bzw. DT_{c_2} wird hierbei die Menge der referenzierten Typen in der Klasse c_1 bzw. c_2 bezeichnet.

$$indirect(c_1, c_2) = \frac{\sum_{x \in \{DT_{c_1} \cap DT_{c_2}\}} idtf(x)}{\sum_{y \in \{DT_{c_1} \cup DT_{c_2}\}} idtf(y)} \tag{5.6}$$

Ein Nachteil von Gleichung 5.6 ist, dass sie auf Typabhängigkeiten beschränkt ist und somit bspw. nicht in der Lage ist, einen indirekten Zusammenhang zwischen Programmelementen mit ähnlichen Aufrufabhängigkeiten zu erkennen. Hinzu kommt, dass Gleichung 5.6 lediglich gemeinsame ausgehende Typabhängigkeiten betrachtet. Potentiell können jedoch auch gemeinsame eingehende Typabhängigkeiten ein Indiz für einen semantischen Zusammenhang sein.
Im Vergleich zu Gleichung 5.6 verwendet Gleichung 5.3 dieselben Abhängigkeitsmerkmale bei der Berechnung des direkten und indirekten Zusammenhangs. Da die, über Gleichung 5.2 berechneten, direkten Zusammenhänge ungerichtet sind, berücksichtigt Gleichung 5.3 sowohl gemeinsame ausgehende als auch eingehende Abhängigkeiten bei der Bewertung des indirekten Zusammenhangs. Allerdings bezieht Gleichung 5.3 beim Vergleich der Vektorabbildungen zweier Programmelemente c_1 und c_2 auch die direkte Abhängigkeit zwischen ihnen mit ein. Um bei einer separaten Beurteilung des direkten und indirekten Zusammenhangs Redundanzen zu vermeiden, sollte sichergestellt werden, dass der indirekte Zusammenhang zwischen c_1 und c_2 unabhängig von ihrem direkten Zusammenhang ist.

Entwurfsanforderung 5.7: Berechnung indirekter Zusammenhänge

Alle betrachteten Abhängigkeitsmerkmale sollen sowohl zu direkten als auch zu indirekten Zusammenhängen führen können. Für eine redundanzfreie Betrachtung sollte der direkte Zusammenhang zweier Programmelemente keinen Einfluss auf ihren indirekten Zusammenhang besitzt.

Aufgrund der Vielzahl von Abhängigkeiten die eine Programmstelle naturgemäß aufweist, werden ihr voraussichtlich viele Zusammenhänge zugeschrieben. Eine Möglichkeit, um

schwache bzw. unsichere Zusammenhänge herauszufiltern, stellen Schwellwerte dar. In [KNH+17] liegt der Schwellwert für die Berücksichtigung eines direkten Zusammenhangs bei 0,7 und für einen indirekten Zusammenhang bei 0,8. Bei Zusammenhängen, die über diesen Schwellwerten liegen, ist die Wahrscheinlichkeit hoch, dass tatsächlich ein semantischer Zusammenhang vorliegt. Bei der Verwendung von Schwellwerten kann das Maß an Unsicherheit, mit dem gearbeitet wird, kontrolliert werden. Die Definition eines Schwellwerts für die syntaxbasierte Quelltextanalyse in dieser Arbeit erscheint daher sinnvoll. Allerdings schränken zu hohe Schwellwerte die Quelltextanalyse ein und können im Zweifelsfall dazu führen, dass vorhandene semantische Zusammenhänge nicht erkannt werden. Es muss daher gewährleistet werden, dass Schwellwerte einfach verändert werden können.

Entwurfsanforderung 5.8: Berücksichtigung von Schwellwerten

Bei der syntaxbasierten Quelltextanalyse soll es durch die Definition von Schwellwerten möglich sein, ein Mindestmaß an Sicherheit, mit dem ein semantischer Zusammenhang vorliegen muss, festzulegen.

Insgesamt fällt bei der Diskussion verwandter Arbeiten auf, dass sich bestehende Arbeiten bei ihrer Suche nach Zusammenhängen vorrangig auf eine Granularitätsebene beschränken. Dies unterscheidet sie fundamental vom Ziel dieser Arbeit, Zusammenhänge auf und zwischen unterschiedlichen Granularitätsebenen darzustellen. Darüber hinaus nutzen die meisten verwandten Arbeiten aus dem Bereich der Konzeptlokalisierung bei ihrer Suche nach semantischen Zusammenhängen lediglich einige wenige Abhängigkeitsmerkmale und ignorieren zum Teil Schnittstellen und Aufzählungstypen komplett. Die Arbeit [AC17] von Amarjeet und Chhabra sticht, aufgrund der Einbeziehung vergleichsweise vieler Abhängigkeitsmerkmale, bei ihrer Analyse zwar hervor, sie evaluieren ihr Vorgehen allerdings lediglich im Bereich der Remodularisierung von Softwaresystemen. Der Einfluss einer umfangreichen Menge an Abhängigkeitsmerkmalen auf die Suche nach semantischen Zusammenhängen bleibt unbeantwortet. Des Weiteren gibt es einige Abhängigkeitsmerkmale, die bei der Beschreibung von Programmelementen bislang unbeachtet blieben. Beispielsweise können die Abhängigkeiten einer Methode durch Hinzunahme der Abhängigkeiten ihrer Anweisungen weiter spezifiziert werden und Klassenbeschreibungen durch die Abhängigkeiten ihrer Konstruktoren erweitert werden.

5.3.1.3 Entwurfsmuster

Eine weitere Sichtweise auf die Abhängigkeiten im Quelltext bieten Entwurfsmuster (Siehe Abschnitt 2.2.3). Sie beschreiben erprobte Lösungen für verschiedene Herausforderung beim Softwareentwurf und lassen sich bspw. als Klassendiagramme darstellen. In [GHJV04] werden drei Arten von Entwurfsmustern unterschieden.
Die erste Art wird Erzeugungsmuster genannt und umfasst Klassenverbände zur Objektgenerierung. Folglich können sie Hinweise darüber liefern, welche Klassen im Bezug auf die Instanziierung bestimmter Typen einen Zusammenhang aufweisen.
Strukturmuster bilden die zweite Art und dienen als Vorlage für die Umsetzung größerer Klassenstrukturen, die bestimmte Aspekte wie Abstraktion, Flexibilität oder Wartbarkeit adressieren. Die Gefahr von Strukturmustern, bezogen auf die vorliegende Arbeit, ist, dass der Zusammenhang zwischen den teilhabenden Klassen zu allgemein ist und sich nicht auf eine Anforderung zurückführen lässt. Eine Fassade [GHJV04] kann bspw. viele verschiedene Funktionalitäten bündeln, die nicht zwangsläufig eine gemeinsame Absicht besitzen. Die Abhängigkeiten in Verbindung mit dem Entwurfsmuster Fassade, können daher für die Identifikation von Zusammenhängen in dieser Arbeit zu abstrakt sein.

Die dritte Art wird Verhaltensmuster genannt und hilft bei der Modellierung und Umsetzung von bspw. Algorithmen, um die Flexibilität einer Software zu unterstützen. Sie beschreiben Klassenstrukturen, die im Hinblick auf ein bestimmtes Verhalten zusammenarbeiten. Allerdings besteht auch hier die Gefahr, dass die Absicht hinter der Umsetzung eines Verhaltensmusters zu allgemein ist. Die konkreten Strategien eines Strategiemusters (Siehe Abschnitt 2.2.3) können unter Umständen sehr unterschiedliche Anforderungen implementieren, die nicht notwendigerweise der Implementierung einer gemeinsamen Anforderung dienen.

Ansätze zur Identifikation von Entwurfsmustern wie in [OAS16, MR17] basieren im Kern auf einem Vergleich der Abhängigkeitsstrukturen im Quelltext mit den Abhängigkeitsstrukturen der gesuchten Entwurfsmuster. Infolgedessen überschneiden sich die Informationen von Entwurfsmustern zum Teil mit den Abhängigkeitsmerkmalen aus Abschnitt 5.3.1.1. Ein Zusammenhang zwischen verschiedenen abgeleiteten Klassen, welche dieselbe abstrakte Methode überschreiben, kann bspw. auch ohne zu Wissen, dass die Klassen das Strategiemuster implementieren, erkannt werden. Dennoch können Entwurfsmuster dazu beitragen, die Vielzahl an Abhängigkeiten im Quelltext zu ordnen. Hierbei wäre es denkbar, Abhängigkeiten, die aus der Implementierung eines Entwurfsmusters folgen, eine größere Aussagekraft bei der Suche nach semantischen Zusammenhängen zuzusprechen. Somit können sie dazu beitragen, die Sicherheit gefundener semantischer Zusammenhänge zu erhöhen. Allerdings besteht die Gefahr, dass sie, im Hinblick auf den späteren Einsatz bei der Rückverfolgbarkeit von Anforderungen, zu allgemeine oder abstrakte Beziehungen beschreiben. Darüber hinaus stellt die Einbeziehung von Entwurfsmustern eine zusätzliche Fehlerquelle dar, wenn diese nicht zuverlässig erkannt werden. Bei der Evaluation von Oruc et al. in [OAS16] wurden fünf Entwurfsmuster in drei Softwareprojekten mit einer Präzision im Bereich zwischen 22% und 100% und einer Ausbeute im Bereich zwischen 25% und 100% erkannt. Insofern unterliegt die Erkennung von Entwurfsmustern starken Schwankungen. Hinzu kommt, dass aufgrund der Einschränkung der Evaluation auf fünf Entwurfsmuster die Ergebnisse in ihrer Aussagekraft eingeschränkt sind. Bei einer Suche nach mehr als fünf Entwurfsmustern wird die durchschnittliche Präzision und Ausbeute vermutlich sinken. Darüber hinaus geben Oruc et al. in ihrer Evaluation an, dass die Präzision und Ausbeute mit zunehmender Projektgröße aller Wahrscheinlichkeit nach weiter sinkt. Für die syntaxbasierte Quelltextanalyse in dieser Arbeit erscheint die Hinzunahme von Entwurfsmustern aufgrund der hohen Unsicherheit nicht sinnvoll.

5.3.1.4 Kontrollfluss

Eine weitere Informationsquelle bei der Suche nach semantischen Zusammenhängen, der bislang in der Literatur kaum Beachtung geschenkt wurde, ist der Kontrollfluss. Der Kontrollflussgraph (Siehe Abschnitt 2.2.2) beschreibt die zeitliche Abfolge der Anweisungen eines Programms und stellt alle möglichen Ausführungspfade auf Anweisungsebene dar. Er liefert eine andere Sichtweise auf die Zusammenhänge im Quelltext als der abstrakte Syntaxbaum und kann eine sinnvolle Ergänzung bei der Analyse von Quelltext sein [TWB+18]. Insbesondere bei einer dynamischen Programmanalyse kann der Kontrollfluss hilfreich sein, um bspw. die Ausführungspfade verschiedener Anwendungsszenarien zu untersuchen. Bei einer statischen Quelltextanalyse liegen hingegen keine Informationen über konkrete Ausführungspfade vor. Allerdings kann die allgemeine Struktur des Kontrollflussgraphen untersucht werden. Eine zentrale Rolle nehmen hierbei Schleifen und Verzweigungen ein. Solche Kontrollstrukturen steuern den Programmablauf indem sie eine Abweichung von der sequentiellen Ausführung erlauben. Verzweigungen führen dazu, dass Anweisungen nur in bestimmten Fällen ausgeführt werden und Schleifen bieten die Möglichkeit, Anweisungen wiederholt auszuführen. Häufig besitzen die Anweisungen im Rumpf einer Kontrollstruktur eine gemeinsame Absicht. Quelltextausschnitt 5.1 zeigt zur Veranschaulichung

eine Verzweigung, die in ihrem Rumpf Anweisungen enthält, welche im Hinblick auf die Durchführung einer Gehaltserhöhung zusammenhängen.

Quelltextausschnitt 5.1: Zusammenhängende Anweisungen innerhalb einer Verzweigung.

```
if(increaseSalary) {
    Salary salary = getIncreasedSalary(currentContract);
    Contract newContract = generateContract(currentContract, salary);
    informEmployee(newContract);
}
```

Eine gemeinsame Platzierung von Anweisungen innerhalb von Kontrollstrukturen kann demzufolge als Hinweis auf einen semantischen Zusammenhang zwischen ihnen aufgefasst werden. Im Hinblick auf das Ziel dieser Arbeit könnte daher ein Zusammenhang zwischen den, in den Anweisungen referenzierten, Typen und Methoden angenommen werden. Für das obige Beispiel läge demnach ein semantischer Zusammenhang zwischen allen, im Rumpf referenzierten, Typen und Methoden vor. Im Allgemeinen kann jedoch nicht aufgrund einer gemeinsamen Referenz innerhalb einer Kontrollstruktur davon ausgegangen werden, dass ein semantischer Zusammenhang vorliegt. Es kann bspw. vorkommen, dass mehrere Funktionalitäten im Rumpf einer Kontrollstruktur implementiert sind, wodurch die Bewertung semantischer Zusammenhänge innerhalb eines Blocks erschwert wird. Verzweigungen besitzen darüber hinaus häufig den Zweck, Fehler wie den Zugriff auf `null`-Referenzen zu vermeiden. Semantische Zusammenhänge durch eine isolierte Betrachtung von Kontrollstrukturen abzuleiten ist daher mit großer Unsicherheit verbunden. Werden Typen oder Methoden hingegen vergleichsweise häufig, im Hinblick auf die Menge aller Verzweigungen und Schleifen im Quelltext, gemeinsam referenziert, liegt mit hoher Wahrscheinlichkeit ein semantischer Zusammenhang zwischen ihnen vor. Eine Möglichkeit zur Bewertung der Häufigkeit gemeinsamer Referenzen zweier Programmelemente e_1 und e_2 bietet Gleichung 5.7. Sie setzt die Anzahl an Kontrollstrukturrümpfen, in denen beide Programmelemente referenziert werden, ins Verhältnis zur Gesamtzahl an Kontrollstrukturrümpfen, die auf e_1 referenzieren. Infolgedessen kann mit Gleichung 5.7 die gerichtete Abhängigkeit von e_1 zu e_2 auf Basis deren gemeinsamer Referenzen innerhalb von Kontrollstrukturrümpfen berechnet werden. Mit B_{e_1} wird die Menge an Kontrollstrukturrümpfen, die eine Referenz zum Programmelement e_1 beinhalten, bezeichnet. Für ein Programmelement e_2, welches ausschließlich gemeinsam mit e_1 referenziert wird, ist das Ergebnis gleich 1. Programmelemente die nie gemeinsam referenziert werden, bekommen den Wert 0 zugeordnet. Gleichung 5.7 berücksichtigt sowohl gemeinsame Referenzen in einem Schleifenblock als auch in einem Verzweigungsblock.

$$cfg_{block}(e_1, e_2) = \frac{|B_{e_1} \cap B_{e_2}|}{|B_{e_1}|} \tag{5.7}$$

Einer Verzweigung können im Gegensatz zu einer Schleife mehrere Quelltextblöcke, bspw. einen für den positiven und einen für den negativen Ausgang der Bedingung, zugeordnet werden. Eine mögliche Interpretation von Verzweigungen ist die Beschreibung alternativer Implementierungen mit der gleichen Absicht. Da `switch`-Anweisungen ohne Beschränkung der Allgemeinheit in bedingte Anweisungen überführt werden können, liegt der Fokus im Folgenden auf der Analyse von `if`-Anweisungen. Quelltextausschnitt 5.2 liefert ein Beispiel für die Angabe alternativer Aktionen. Im `if`- und `else`-Block finden sich unterschiedliche Methodenaufrufe, die das gleiche konzeptionelle Ziel besitzen. Infolgedessen kann `sendEmail` und `sendLetter` in diesem Fall ein semantischer Zusammenhang zugesprochen werden.

Quelltextausschnitt 5.2: Alternative Quelltextblöcke mit gleicher Absicht.

```
if(isEmailPreferred) {
    sendEmail();
} else {
    sendLetter();
}
```

Allerdings gilt auch hier, dass im Allgemeinen nicht davon ausgegangen werden kann, dass zwischen den Anweisungen in den `if`- und `else`-Blöcken von Verzweigung ein semantischer Zusammenhang besteht. Dennoch kann, wie bereits zuvor, eine vergleichsweise häufige Referenzierung von Programmelementen in alternativen Quelltextblöcken einer Verzweigung als Indiz für einen semantischen Zusammenhang aufgefasst werden. Als alternative Quelltextblöcke werden hierbei die `if`- und `else`-Blöcke einer Verzweigung bezeichnet. Eine Möglichkeit zur Bewertung bietet Gleichung 5.8. Mit $n_{alt}(e_1, e_2)$ wird die Zahl der Verzweigungen angegeben, welche die Programmelemente e_1 und e_2 in alternativen Blöcken referenzieren. Werden zwei Programmelemente ausschließlich in alternativen Quelltextblöcken aufgerufen, ist das Ergebnis gleich 1. Treten sie nie in alternativen Quelltextblöcken auf, ist das Ergebnis gleich 0.

$$cfg_{alt}(e_1, e_2) = \frac{n_{alt}(e_1, e_2)}{|B_{e_1}|} \tag{5.8}$$

Zusammenfassend können Kontrollstrukturen wichtige Hinweise über semantische Zusammenhänge von Typen und Methoden liefern. Es ist daher sinnvoll, Kontrollflussabhängigkeiten als zusätzliches Abhängigkeitsmerkmal bei der Bewertung von direkten Zusammenhängen zwischen Programmelementen zu berücksichtigen. Ein starker Hinweis auf einen semantischen Zusammenhang stellt sowohl eine hohe Bewertung von Gleichung 5.7 als auch eine hohe Bewertung von Gleichung 5.8 dar. Darüber hinaus ist es für die Existenz eines semantischen Zusammenhangs nicht erforderlich, dass beide Kennzahlen hoch ausfallen. Um eine Gesamtbewertung der Kontrollflussabhängigkeit zweier Programmelemente e_1 und e_2 zu erhalten, erscheint daher eine Kombination der beiden Gleichungen über das Maximum wie in Gleichung 5.9 sinnvoll.

$$dep_{cfg}(e_1, e_2) = max(cfg_{block}(e_1, e_2), cfg_{alt}(e_1, e_2)) \tag{5.9}$$

5.3.2 Entwurf der syntaxbasierten Quelltextanalyse

Bei der syntaxbasierten Quelltextanalyse sollen Zusammenhänge zwischen den Programmelementen im Quelltext auf der Basis der statischen Quelltextstruktur gesucht werden. Ausgangspunkt hierfür stellen verschiedene syntaxbasierte Abhängigkeitsmerkmale dar.

5.3.2.1 Abhängigkeitsmerkmale

Im Rahmen dieser Arbeit werden semantische Zusammenhänge zwischen Typen, zwischen Methoden und zwischen Typen und Methoden gesucht. Zusammenhänge auf Anweisungsebene bzw. zwischen Anweisungen und Typen oder Methoden werden nicht berücksichtigt, da sich der Zweck einer Anweisung erst aus dessen Verwendungskontext ergibt und die Informationen aus isoliert betrachteten Anweisungen daher wenig hilfreich für das Ziel dieser Arbeit sind. Dennoch sind die Abhängigkeiten auf Anweisungsebene bei der Beschreibung von Typen oder Methoden, wie in Abschnitt 5.3.1.1 beschrieben, relevant, um bspw. den Rumpf einer Methode oder statische Quelltexte eines Typs miteinzubeziehen. Die Abhängigkeiten von Anweisungen werden hierzu auf den umschließenden Typ oder die umschließende Methode projiziert. Semantische Zusammenhänge zwischen Paketen bzw.

zwischen Paketen und Typen werden nicht berücksichtigt, da der Zweck eines Pakets in der Regel sehr allgemein ist und solche Zusammenhänge für eine Rückverfolgbarkeitsanalyse keinen erkennbaren Vorteil mit sich bringen. Bei der Suche nach Zusammenhängen zwischen Typen ist die Hinzunahme der Paketstruktur als Abhängigkeitsmerkmal jedoch sinnvoll (Siehe Abschnitt 5.3.1.1). Da Gleichung 5.1 sowohl Klassen im selben Paket als auch Klassen, die sich in ihrem Paketpfad überschneiden, berücksichtigt und somit eine adäquate Bewertung liefert, wird sie für die Berechnung der Paketabhängigkeit in dieser Arbeit verwendet.

Aus Entwurfsanforderung 5.1 geht hervor, dass es sinnvoll ist, bei der Suche nach semantischen Zusammenhängen ein möglichst breites Spektrum an Abhängigkeitsmerkmalen zu berücksichtigen. In Abschnitt 5.3.1.1 wurden verschiedene Abhängigkeitsmerkmale auf unterschiedlichen Granularitätsebenen im Quelltext analysiert. Hierbei zeigte sich, dass potentiell alle Informationen bei der Suche nach semantischen Zusammenhängen hilfreich sein können. Infolgedessen werden in dieser Arbeit alle Abhängigkeitsmerkmale aus Abschnitt 5.3.1.1 berücksichtigt. Aus Abschnitt 5.3.1.4 geht darüber hinaus hervor, dass die Hinzunahme des Kontrollflusses als zusätzliches Abhängigkeitsmerkmal sinnvoll ist. Eine geeignete Bewertungsformel die sowohl auf Typen als auch auf Methoden angewendet werden kann liefert Gleichung 5.9. Sie untersucht, wie häufig zwei Programmelemente gemeinsam in einem Quelltextblock einer Verzweigung oder einer Schleife bzw. in alternativen Quelltextblöcken von Verzweigungen referenziert werden.

In Tabelle 5.4 sind die Abhängigkeitsmerkmale für die verschiedenen Programmelementpaare aufgelistet. Einige Abhängigkeitsmerkmale können nur zwischen bestimmten Programmelementen vorliegen, weshalb für alle betrachteten Porgrammelementkombinationen eine eigene Merkmalsmenge definiert wird.
Der erste Block in Tabelle 5.4 umfasst die Abhängigkeitsmerkmale für die Bewertung des semantischen Zusammenhangs zwischen Methoden. Methodenaufrufe werden aus den, im Rumpf enthaltenen, Anweisungen abgeleitet. Die Kontrollflussabhängigkeit bewertet die Häufigkeit, mit der zwei Methoden gemeinsam in einem Quelltextblock einer Kontrollstruktur bzw. in alternativen Quelltextblöcken einer Verzweigung aufgerufen werden. Als letztes Merkmal zwischen Methoden dient die Überschreibungsbeziehung. Sie wird aus der `Override`-Annotation einer Methodendeklaration abgeleitet.
Die Abhängigkeitsmerkmale zwischen Methoden und Typen sind im zweiten Block von Tabelle 5.4 aufgeführt. Variablen- und Typzugriffe, explizite Typreferenzen in Typüberprüfungen oder Typumwandlungen und Instanziierungen werden hierbei aus den, im Rumpf platzierten, Anweisungen abgeleitet (Siehe Abschnitt 5.3.1.1). Abhängigkeiten aufgrund von Ausnahme-, Parameter- und Rückgabetypen werden aus der Methodendeklaration extrahiert. Die Abhängigkeit zum umschließenden Typ verweist auf den Typ, in dem eine Methode definiert ist. Da die Kontrollflussabhängigkeit für Typen und Methoden gleichermaßen herangezogen werden kann, wird sie als weiteres Abhängigkeitsmerkmal von Methoden zu Typen berücksichtigt.
Für die Beschreibung der Abhängigkeiten zwischen Typen werden die Merkmale im dritten Block von Tabelle 5.4 verwendet. Die Typhierachie beschreibt hierbei Implementierungs- und Erweiterungsabhängigkeiten zu anderen Typen. Innere Typdefinitionen und Attributtypen führen zu weiteren Typabhängigkeiten. Das Gegenstück zur Abhängigkeiten zu inneren Typen ist die Abhängigkeit zum umschließenden Typ. Sie zeigt auf den Typ, innerhalb dessen der betrachtete Typ definiert ist. Auch die Paketstruktur wird als Abhängigkeitsmerkmal zwischen Typen berücksichtigt. Die Paketabhängigkeit wird hierbei mit Gleichung 5.1 bewertet. Aufgrund von Anweisungen und Ausdrücken innerhalb von Konstruktoren, statischen Quelltextblöcken oder Attributinitialisierungen, können Typen darüber hinaus Abhängigkeiten aus der Anweisungsebene besitzen. Hierzu zählen Variablen oder Typzugriffe, explizite Typreferenzen und Instanziierungen. Parameter- und Ausnah-

Tabelle 5.4: Überblick der direkten Abhängigkeitsmerkmale.

Abhängigkeit	Merkmal
Methode → Methode	Methodenaufruf Kontrollfluss Überschreiben
Methode → Typ	Variablen- oder Typzugriff Explizite Typreferenz Instanziierung Ausnahme Parameter Umschließender Typ Rückgabe Kontrollfluss
Typ → Typ	Typhierarchie Innerer Typ Attributtyp Umschließender Typ Paketstruktur Variablen- oder Typzugriff Explizite Typreferenz Instanziierung Ausnahme Parameter Kontrollfluss
Typ → Methode	Methodendefinition Methodenaufruf Kontrollfluss

medefinitionen von Konstruktoren werden ebenfalls als Typabhängigkeiten interpretiert. Wie bereits zuvor wird die Kontrollflussabhängigkeit auch zur Bewertung der Abhängigkeit zwischen Typen herangezogen.
Im letzten Block von Tabelle 5.4 sind die Abhängigkeitsmerkmale von Typen zu Methoden zusammengefasst. Die Methodendefinition beschreibt die Abhängigkeit eines Typs zu allen Methoden, die in seinem Rumpf definiert sind. Aus den Anweisungen oder Ausdrücken innerhalb von Konstruktoren, statischen Quelltextblöcken oder Attributinitialisierungen werden Methodenaufrufe als Abhängigkeiten zu den aufgerufenen Methoden betrachtet. Als letztes Merkmal dient auch hier der Kontrollfluss.

5.3.2.2 Semantische Zusammenhänge

Aus Entwurfsanforderung 5.4 geht hervor, dass sowohl direkte als auch indirekte Abhängigkeiten zwischen Programmelementen bei der Suche nach semantischen Zusammenhängen berücksichtigt werden sollen. Für die Berechnung der direkten Abhängigkeit wird eine Modifikation der Gleichung 5.4 aus [KNH+17] verwendet. Gleichung 5.4 bietet sich an, da die Ergebnisse auf den Bereich zwischen $[0, 1]$ normiert sind und sowohl die ein- als auch die ausgehenden Abhängigkeiten der Programmelemente betrachtet werden. Im Einklang mit Entwurfsanforderung 5.5 wird die maximale aus- bzw. eingehende Abhängigkeit der betrachteten Programmelemente für die Normierung verwendet, um zu verhindern, dass Programmelemente mit vielen Abhängigkeiten benachteiligt werden. Darüber hinaus wird

die Abhängigkeit für jedes Abhängigkeitsmerkmal separat berechnet, um entsprechend Entwurfsanforderung 5.6 die Unterdrückung einzelner Abhängigkeitsmerkmale zu vermeiden. Die, den Anforderungen aus Abschnitt 5.3.3.1 angepasste, Formel zu Berechnung der direkten Abhängigkeit zwischen zwei Programmelementen e_1 und e_2 zeigt Gleichung 5.10.

$$dep(e_1, e_2) = \sum_{r \in R(e_1,e_2)} w_r * \frac{2 * n_r(e_1, e_2)}{max_{k \in E(e_2)}(n_r(e_1, k)) + max_{k \in E(e_1)}(n_r(k, e_2))} \tag{5.10}$$

Die Menge der möglichen Abhängigkeitsmerkmale zwischen zwei Programmelementen e_1 und e_2 wird durch $R(e_1, e_2)$ beschrieben. Handelt es sich bei e_1 bspw. um eine Methode und bei e_2 um einen Typ, beschreibt $R(e_1, e_2)$ die Merkmale im zweiten Block von Tabelle 5.4. Die Menge $E(e)$ umfasst alle Programmelemente derselben Art, wie das Programmelement e im Quelltext. Mit $n_r(e_1, e_2)$ wird die Anzahl bestehender Abhängigkeiten eines Typs r von e_1 zu e_2 beschrieben. Um Abhängigkeitsmerkmale, wie in Entwurfsanforderung 5.2 gefordert, individuell gewichten zu können, werden Gewichte w_r berücksichtigt, die die Relevanz eines Abhängigkeitsmerkmals r bestimmen. Für alle Abhängigkeitsmerkmale $r \in R(e_1, e_2)$ gilt, dass ihr Gewicht w_r zwischen 0 und 1 liegt und die Summe der Gewichte gleich 1 ist. Initial wird eine Gleichgewichtung aller Abhängigkeitsmerkmale gewählt, da aufgrund der projektspezifischen Ausprägung der Abhängigkeitsmerkmale eine individuelle Bewertung von deren Aussagekraft bei der Suche nach semantischen Zusammenhängen im Quelltext schwierig ist. Eine Optimierung der Gewichte auf der Basis einer Musterlösung, wie in Entwurfsanforderung 5.3 beschrieben, erfolgt in Abschnitt 6.5. Falls es keine direkte Abhängigkeit von e_1 zu e_2 gibt, ist das Ergebnis gleich 0. Eine maximale Bewertung von 1 erhalten Programmelemente, die voneinander abhängen und zu keinem anderen Programmelement mehr eingehende oder ausgehende Abhängigkeiten besitzen.
Aus Abschnitt 5.2.2 geht hervor, dass semantische Zusammenhänge ungerichtet sind. Folglich wird die direkte Abhängigkeit über den Mittelwert der direkten Abhängigkeit von e_1 zu e_2 und von e_2 zu e_1 berechnet. Gleichung 5.11 zeigt die finale Formel zur Bewertung des direkten syntaxbasierten Zusammenhangs zweier Programmelemente.

$$direct(e_1, e_2) = \frac{1}{2}(dep(e_1, e_2) + dep(e_2, e_1)) \tag{5.11}$$

Bei der Bewertung von indirekten Zusammenhängen werden die berechneten direkten Zusammenhänge genutzt, um Programmelemente mit ähnlichen Abhängigkeiten zu identifizieren (Siehe Entwurfsanforderung 5.7). Da Gleichung 5.3 aus [AC17] für die Bewertung von indirekten Zusammenhängen alle direkten Abhängigkeitsmerkmale berücksichtigt, liefert sie eine geeignete Grundlage für die Berechnung in dieser Arbeit. Voraussetzung für die Verwendung Gleichung 5.3 sind Vektorabbildungen der Programmelemente, deren Elemente die direkten Zusammenhänge mit allen anderen Programmelementen im Quelltext beschreiben. Der Kosinus des Winkels zwischen zwei Vektorabbildungen entspricht daraufhin dem indirekten Zusammenhang. Um Entwurfsanforderung 5.7 zu erfüllen, werden die Vektorabbildungen $\vec{u}$ sowie $\vec{v}$ von e_1 und e_2 vorverarbeitet. Hierbei werden die Elemente v_1 und u_1 für den direkten Zusammenhang mit e_1 bzw. v_2 und u_2 für den direkten Zusammenhang mit e_2 aus den Vektoren entfernt. Folglich fließen die direkten Zusammenhänge zu e_1 und e_2 nicht in die Berechnung mit ein. Gleichung 5.12 aus [AC17] zeigt die Gleichung zur Bewertung des indirekten Zusammenhangs zweier Programmelemente in dieser Arbeit.

$$indirect(\vec{u}, \vec{v}) = \frac{\sum_{k=1}^{n}(u_k * v_k)}{\sqrt{\sum_{k=1}^{n}(u_k)^2} * \sqrt{\sum_{k=1}^{n}(v_k)^2}} \tag{5.12}$$

Bei Programmelementen, die exakt die gleichen direkten Abhängigkeitsmerkmale ausweisen, ist das Ergebnis von Gleichung 5.12 gleich 1. Da direkte Zusammenhänge nicht kleiner

als 0 sein können, werden Programmelemente mit maximal unterschiedlichen direkten Zusammenhängen mit 0 bewertet.

Insgesamt erlauben Gleichung 5.11 und Gleichung 5.12 eine Berechnung von indirekten und direkten Zusammenhängen unabhängig von der Art der betrachteten Programmelemente. Darüber hinaus bieten sie eine hohe Flexibilität im Hinblick auf die Menge der berücksichtigten Abhängigkeitsmerkmale durch die Definition von R. Um eine finale Bewertung des Zusammenhangs zweier Programmelemente zu erhalten ist es notwendig, die direkte Abhängigkeit mit der indirekten Abhängigkeit zu kombinieren. Eine Möglichkeit hierfür wäre die Berechnung eines gewichteten Mittelwerts der beiden Kennzahlen. Allerdings besteht hierbei die Gefahr, dass der Zusammenhang zwischen Programmelementen, die lediglich von einer der beiden Gleichungen eine hohe Bewertung erhalten, unterbewertet wird. Da ein hoher direkter oder indirekter Zusammenhang als Indiz für die Existenz eines semantischen Zusammenhangs vermutlich ausreicht, ist es sinnvoll, das Maximum der beiden berechneten Kennzahlen zur Bewertung des syntaxbasierten Zusammenhangs zwischen zwei Programmelementen e_1 und e_2 heranzuziehen. Gleichung 5.13 zeigt die Formel hierzu. Mit $e2v$ ist hierbei die Abbildungsfunktion beschrieben, die ein Programmelement, wie zuvor dargelegt, auf einen reellwertigen Vektor abbildet.

$$sda(e_1, e_2) = max(direct(e_1, e_2), indirect(e2v(e_1), e2v(e_2))) \tag{5.13}$$

Um das Maß an Unsicherheit, mit dem die Suche nach semantischen Zusammenhängen arbeitet, beeinflussen zu können, wird in Folge von Entwurfsanforderung 5.8 ein Schwellwert $\alpha_{sda} \geq 0$ definiert. Die mit Gleichung 5.13 berechneten semantischen Zusammenhänge, die kleiner als α_{sda} sind, werden verworfen. Vorerst wird $\alpha_{sda} = 0$ gesetzt, sodass alle berechneten Zusammenhänge berücksichtigt werden. Die Optimierung des Schwellwerts ist Bestandteil von Abschnitt 6.5.

5.3.3 Natürliche Sprache im Quelltext

Ergänzend zur Analyse der Syntax, wird im Rahmen dieser Arbeit die natürliche Sprache im Quelltext untersucht. Sie bietet eine alternative Sichtweise auf den Quelltext und erlaubt die Einbeziehung der sprachlichen Semantik bei der Suche nach zusammenhängenden Quelltextstellen. Grundsätzlich können Paketnamen, Kommentare, Bezeichner und statische Werte im Quelltext, Informationen in natürlicher Sprache beinhalten. Im Vergleich zur natürlichsprachlichen Dokumente fällt auf, dass Quelltext wenig Fließtext enthält. Der Grund dafür ist, dass Fließtext lediglich in Kommentaren oder statischen Zeichenketten vorkommen kann. Ein Großteil des Quelltextes besteht jedoch aus Anweisungen, Klassen- und Methodendefinitionen. Infolgedessen besteht die natürliche Sprache im Quelltext hauptsächlich aus einzelnen Wörtern oder Phrasen. Da Entwickler bemüht sind Begriffe möglichst konsistent zu verwenden, um Unklarheiten zu vermeiden, gibt es im Quelltext außerdem viele Wortwiederholungen. Insgesamt besitzt Quelltext im Vergleich zu herkömmlichen Texten ein kleineres Vokabular und fordert aufgrund seiner Charakteristik ein differenzierteres Vorgehen bei seiner Analyse. Bevor auf die Möglichkeiten der Analyse im Detail eingegangen wird, werden im Folgenden die verschiedenen natürlichsprachlichen Bestandteile im Quelltext näher betrachtet.

Paketpfade

Mit Hilfe des Paketpfades kann der Definitionsort eines Typs eindeutig bestimmt werden. Der Paketpfad setzt sich aus einer Reihe von Verzeichnisnamen wie `com.company.service` zusammen, die durch einen Punkt getrennt aufgelistet werden. Folglich enthält der Paketname vor allem strukturelle Informationen. Bei der syntaxbasierten Quelltextanalyse in Abschnitt 5.3.2 wurde die Paketstruktur als Indiz für einen semantischen Zusammenhang

zwischen Typen bereits berücksichtigt.
Im Hinblick auf die lexikalische Analyse wäre es denkbar, die im Paketpfad enthaltenen Wörter zur Beschreibung eines Typs heranzuziehen. Allerdings enthalten Paketnamen häufig sehr allgemeine Begriffe wie `service`, wodurch ihre Aussagekraft bei der Suche nach semantischen Zusammenhängen in dieser Arbeit fragwürdig ist. Darüber hinaus besteht bei einer lexikalischen Analyse der Paketnamen die Gefahr, implizit die Paketstruktur mit zu berücksichtigen, wodurch eine Redundanz zur syntaxbasierten Quelltextanalyse entsteht.

Bezeichner

Bezeichner sind Namen für Typen, Methoden oder Variablen. In der Regel versuchen Entwickler möglichst aussagekräftige Bezeichner zu wählen, welche die Rolle oder Funktion eines Programmelements beschreiben, um die Verständlichkeit von Quelltext zu fördern. Folglich kann ein Vergleich der verwendeten Bezeichner in Quelltextausschnitten Hinweise auf semantische Zusammenhänge liefern. Im Vergleich zur natürlichen Sprache besitzen Bezeichner eine besondere Charakteristik. Auf der einen Seite bestehen sie häufig aus der Konkatenation verschiedener Wörter. Weit verbreitet ist hierbei die Binnenmajuskel-Schreibweise, bei der durch eine Trennung an den Großbuchstaben die einzelnen Worte aus der Bezeichnung extrahiert werden können. Auf der anderen Seite enthalten Bezeichner oftmals Abkürzungen wie der Variablenname `empl`, um die Länge einer Bezeichnung zu reduzieren. Welche Wörter auf welche Weise abgekürzt werden obliegt dem Entwickler, weshalb die Auflösung von Abkürzungen im Allgemeinen eine Herausforderung darstellt.

Ein Typ, der eine bestimmte Entität, wie einen Mitarbeiter repräsentiert, besitzt häufig dessen Namen, z.B. `Employee`, als Bezeichnung. Die Bezeichnungen von Typen, welche Funktionalitäten wie Berechnungen oder Validierungen implementieren, unterscheiden sich hiervon. Ihre Bezeichnung setzt sich in der Regel aus der Bezeichnung des Typs, mit dem gearbeitet wird und einer stereotypischen Rollenbezeichnung, die den Aufgabenbereich beschreibt, zusammen. Ein Beispiel hierfür ist die Bezeichnung `ContractValidator`, aus der abgeleitet werden kann, dass der Typ die Überprüfung von Verträgen realisiert. Die Rollenbezeichnungen sind nicht standardisiert. Welche Typbezeichnungen es gibt und welche Bedeutung diese besitzen, ist daher projektspezifisch. Eine Herausforderung im Bezug auf die Rollenbezeichnungen ist, dass es grundsätzlich viele verschiedene Typen gibt, die alle die gleiche Rolle implementieren. Es kann bspw. vorkommen, dass mehrere Typen den Begriff *Repository* in ihrer Typbezeichnung beinhalten. Obwohl z.B. `EmployeeRepository` und `ContractRepository` die gleiche Rollenbezeichnung besitzen, ist ihr semantischer Zusammenhang, aufgrund einer gemeinsamen Anforderung fragwürdig, da sie unterschiedliche Entitäten verarbeiten. Bei der lexikalischen Analyse ist es daher sinnvoll, den Kontext bzw. die Umgebung eines Begriffs zu berücksichtigen und häufig vorkommenden Begriffen weniger Aussagekraft zuzusprechen.
Darüber hinaus können Typbezeichner Begriffe umfassen, welche auf die Art oder Eigenschaften des Typs schließen lassen. Das Präfix *I*, wie in `IEmployeeService`, deutet bspw. darauf hin, dass es sich bei diesem Typ um eine Schnittstelle handelt. Analog dazu sind die Terme *Abstract* oder *Base* Hinweise darauf, dass der beschriebene Typ eine abstrakte Klasse ist. Solche allgemeinen Begriffe und Präfixe sind bei der Suche nach semantischen Zusammenhängen wenig hilfreich, da sie für sich betrachtet keine Aussage über den Zweck eines Typs zulassen.

Methodenbezeichner enthalten in der Regel Verben wie `calculate` oder `find` welche die Aktion einer Methode beschreiben. Häufig ist das Verb wie in `updateEmployee` mit der Bezeichnung des Typs verknüpft, auf welchen sich die Aktion bezieht. Des Weiteren lassen sich aus Methodenbezeichnungen oftmals Informationen über die Parameter und die Rückgabe einer Methode ableiten. Die Methodenbezeichnung `getContractWithLatestStartDate-ByEmployee` sagt bspw. aus, dass ein Objekt vom Typ `Contract` zurückgegeben wird. Der

hintere Teil der Bezeichnung deutet darauf hin, dass als Parameter ein Mitarbeiter-Objekt erwartet wird. Im mittleren Teil der Bezeichnung wird die Voraussetzung beschrieben, welche die Rückgabe erfüllen muss. Das Beispiel zeigt, dass Methodenbezeichnungen, in Abhängigkeit von ihrem Detaillierungsgrad, viele wichtige Informationen über die Funktionsweise einer Methode beinhalten können.
Ähnlich zu den Typbezeichnern gibt es auch bei der Betrachtung von Methodenbezeichnern viele Begriffe die häufig vorkommen. Ein Beispiel hierfür sind die Verben *get* und *set*, die in den Methoden für den Attributzugriff standardmäßig vorkommen. Bei der Suche nach semantischen Zusammenhängen besitzen solche allgemein verwendeten Terme wenig Aussagekraft. Infolgedessen kann es sinnvoll sein, den Einfluss inflationär verwendeter Begriffe bei der lexikalischen Analyse zu verringern.

Variablennamen beinhalten häufig den Bezeichner des Variablentyps wie `contractValidator`, um zu kennzeichnen, dass sie auf ein Objekt dieses Typs, hier `ContractValidator`, referenzieren. Für die Suche nach semantischen Zusammenhängen sind solche Bezeichnungen wenig hilfreich, da sie keine zusätzlichen Informationen über den Zweck der Variable beinhalten. Anders verhält es sich mit Variablennamen, die den Zweck einer Variable beschreiben. Ein Beispiel hierfür ist der Bezeichner `newEmployee`, welcher darauf hindeutet, dass es sich beim referenzierten Objekt um einen neu angelegten Mitarbeiter handelt. Darüber hinaus können Variablennamen die semantische Rolle eines Werts beschreiben, ohne Bezug auf den Variablentyp zu nehmen. Eine Variable vom Typ `Employee`, dessen Bezeichner `ceo` ist, deutet bspw. darauf hin, dass es sich beim referenzierten Objekt um den Geschäftsführer handelt.
Bezeichner von statischen Variablen unterscheiden sich häufig von normalen Variablenbezeichnungen. Weit verbreitet ist die ausschließliche Verwendung von Großbuchstaben, wobei mehrere Wörter durch einen tief liegenden waagerechten Strich, wie in `DEFAULT_SALARY`, verbunden sind. Abgesehen von der Schreibweise, können Bezeichner für statische Variablen auf die gleiche Weise wie herkömmliche Variablennamen interpretiert werden.

Neben den Begriffen eines Bezeichners kann der Kontext, indem ein Bezeichner verwendet wird, Hinweise auf semantische Zusammenhänge liefern. Dies ist vor allem im Hinblick auf Variablenbezeichner sinnvoll, da es viele Bezeichnungen wie `var` oder `result` gibt, die häufig im Quelltext auftreten. Ohne Einbeziehung der Umgebung von Bezeichnern besteht die Gefahr Zusammenhänge aufgrund von einzelnen Bezeichnungen zu erkennen, obwohl sich der Großteil der Bezeichner zweier Programmelemente stark unterscheidet. Es erscheint daher sinnvoll, den Kontext von Bezeichnungen einzubeziehen und z.B. die durchschnittliche Ähnlichkeit zwischen den Bezeichnern zweier Programmelemente bei der lexikalischen Analyse heranzuziehen, anstelle die Bezeichner einzeln zu vergleichen.

Statische Werte

Statische Werte sind Zeichen, Zeichenketten und Zahlen, die im Quelltext explizit angegeben werden. Häufig werden sie zur Initialisierung von Variablen genutzt oder beschreiben Standardwerte, Fehler- und Protokollmeldungen. Statische Zahlenwerte und einzelne Zeichen sind im Rahmen der lexikalischen Analyse grundsätzlich irrelevant, da sie keine Aussagekraft, im Hinblick auf das Ziel dieser Arbeit, besitzen. Es ist bspw. nicht möglich, aufgrund zweier Zahlen wie 100,0 und 12,5 eine Aussage über den semantischen Zusammenhang der jeweiligen Quelltextstellen abzuleiten.
Statische Zeichenketten besitzen im Vergleich dazu häufig einen höheren Informationsgehalt. Die verwendeten Zeichenketten in der Zuweisung `String[] var = {"Monday", ..., "Sunday"};` geben bspw. einen Hinweis darauf, dass die Quelltextstelle mit Wochentagen arbeitet. Infolgedessen kann die Berücksichtigung von statischen Zeichenketten zur Identifikation weiterer semantischer Zusammenhänge führen. Bezogen auf die obige Initialisierung wäre eine Analyse potentiell in der Lage, einen semantischen Zusammenhang

zu einer Quelltextstelle, die den Bezeichner `workingDays` beinhaltet, zu erkennen.
Eine besondere Aussagekraft besitzen statische Zeichenketten, die für Protokoll- oder Fehlermeldungen genutzt werden. Im Gegensatz zu den zuvor beschriebenen Zeichenketten beinhalten sie Informationen über den aktuellen Programmablauf und beschreiben bspw. den Grund für aufgetretene Fehler. In der Regel werden hierfür vollständige Sätze verwendet, wodurch diese Zeichenketten mehr Analysepotential besitzen. Aus der Fehlermeldung „the contract already exists and could not be stored" lässt sich bspw. ableiten, dass versucht wurde, ein Vertragsobjekt zu speichern. Informationen über die verarbeiteten Objekte oder die ausgeführte Aktion einer Quelltextstelle, sind für die Suche nach semantischen Zusammenhängen von erheblichem Wert. Mit Hilfe der Informationen aus der Fehlermeldung, kann bspw. ein semantischer Zusammenhang zur Methode `saveContract` gefolgert werden. Allerdings können Zeichenketten in Fehler- und Protokollmeldungen auch weniger hilfreiche Informationen enthalten wie die Meldung „require non null" aus der nicht ersichtlich ist, um welche Aktion und welche Objekte es sich handelt.
Insgesamt können statische Zeichenketten sowohl in Meldungen als auch in Standardwerten dazu beitragen, den Kontext oder die Semantik einer Programmstelle genauer zu beschreiben und die Suche nach semantischen Zusammenhängen zu verbessern.

Kommentare

Bei der Suche nach semantischen Zusammenhängen spielen Kommentare eine zentrale Rolle, da sie, nach dem Quelltext, die am häufigsten genutzte Informationsquelle für das Nachvollziehen der Funktionsweise eines Programms sind [SAO05]. Demzufolge finden sich in Kommentaren Hinweise auf die Absichten und Zusammenhänge von Programmelementen. Grundsätzlich besteht ein Kommentar aus einer Beschreibung in natürlicher Sprache und wird oberhalb des beschriebenen Quelltextelements platziert. Bei Kommentaren oberhalb von Typen, Attributen oder Methoden ist erkennbar, auf welches Programmelement sie sich beziehen. Bei Kommentaren an anderen Stellen im Quelltext, wie z.B. oberhalb einer Quelltextzeile, ist nicht eindeutig ersichtlich, ob sich der Kommentar lediglich auf eine Quelltextzeile oder auf einen Anweisungsblock bezieht. Für das Ziel dieser Arbeit, semantische Zusammenhänge zwischen Typen und Methoden zu finden, ist dies jedoch zu vernachlässigen. Da die Anweisungsebene nicht gesondert betrachtet wird, können alle Kommentare innerhalb eines Methodenrumpfs zur Beschreibung der jeweiligen Methode genutzt werden. Analog dazu können alle Kommentare innerhalb eines Typs, die keiner Methode zugeordnet sind, zur Beschreibung des Typs herangezogen werden.
Ein fundamentaler Bestandteil des verwendeten Vokabulars in Kommentaren sind die, im Softwareprojekt genutzten, Bezeichner. Infolgedessen sind Abkürzungen und zusammengesetzte Wörter auch in Kommentaren zu finden.

Grundsätzlich kann zwischen Zeilen- bzw. Blockkommentaren und Dokumentationskommentaren unterschieden werden. Dokumentationskommentare sind im Vergleich zu Zeilen- und Blockkommentaren semi-strukturiert, da sie Strukturinformationen enthalten können. Sie erlauben den Entwicklern durch verschiedene Markierungen, bspw. explizit auf Programmelemente zu verweisen. Neben den Markierungen können Dokumentationskommentare Formatierungselemente wie HTML enthalten. Abschnitt 2.2.4 geht ausführlich auf die Charakteristik von Dokumentationskommentaren ein. Ein Beispiel für einen Dokumentationskommentar zu einer Methode zeigt Quelltextausschnitt 5.3. Informationen über den Parameter und die potentiell geworfene Ausnahme sind über die Markierungen `@param` und `@throws` explizit gekennzeichnet. Die Markierung `@link` verweist auf den Typ `Employee`. Durch die Markierungen in Dokumentationskommentaren wird das Verständnis, insbesondere wenn bspw. der Methodenname und die Parameterbezeichnungen kryptisch gewählt sind, erleichtert.

Quelltextausschnitt 5.3: Dokumentationskommentar einer Methode.

```
/**
 * Stores a new employee in the database.
 *
 * @param employee the new {@link Employee}.
 * @throws EntityAlreadyExistsException if the given {@link Employee}
 *    already exists.
 */
public void saveEmployee(Employee employee) throws
   EntityAlreadyExistsException { ... }
```

Eine Herausforderung bei der Verarbeitung von Dokumentationskommentaren ist es, die Markierungen und Verweise korrekt einzuordnen. Aus expliziten Verweisen folgen bspw. nicht zwangsläufig semantische Zusammenhänge. Der Kommentar „The functionality of this method must not be confused with the functionality of {@link saveEmployee}." nutzt zwar eine explizite Referenz auf die Methode `saveEmployee`, allerdings nur, um sich von dieser abzugrenzen und nicht, um eine Gemeinsamkeit zu kennzeichnen. Für eine adäquate Interpretation expliziter Verweise auf Programmelemente ist es daher notwendig, die Semantik des jeweiligen Satzes zu analysieren.
Weder Dokumentationskommentare noch Zeilen- und Blockkommentare werden einer Syntaxüberprüfung unterzogen. Folglich ist es möglich, dass bspw. Teile einer Methodensignatur im dazugehörenden Dokumentationskommentare nicht beschrieben sind. Auf der anderen Seite können Entwickler in Zeilen- oder Blockkommentaren prinzipiell Markierungen oder HTML-Formatierungen verwenden. Der Unterschied der beiden Kommentararten liegt darin, dass Dokumentationskommentare bei der automatisierten Generierung der Dokumentation eines Softwareprojekts berücksichtigt und weiteren Verarbeitungsschritten unterzogen werden. Da es keine Restriktionen im Bezug auf den Inhalt eines Kommentars gibt, ist es für die Zwecke dieser Arbeit sinnvoll, alle Kommentare auf die gleiche Weise zu behandeln.

Zusammenfassend kann zwischen zwei Arten von natürlicher Sprache im Quelltext unterschieden werden. Die erste Art umfasst Bezeichner und Paketnamen. Typischerweise sind sie aus mehreren, oft domänenspezifischen, Begriffen zusammengesetzt und enthalten viele Abkürzungen. Die zweite Art umfasst Kommentare und statische Zeichenketten im Quelltext. Sie beschreiben den Zweck und die Hintergründe von Programmstellen und enthalten häufig Fließtext. Folglich besitzen sie ein vielfältigeres und größeres Vokabular als Bezeichner.

5.3.3.1 Diskussion verwandter Arbeiten

Nachdem im vorangegangenen Abschnitt ein Überblick über die verschiedenen Informationsquellen für natürliche Sprache im Quelltext gegeben wurde, steht die Frage im Raum, wie diese Informationen bei der Suche nach semantischen Zusammenhängen genutzt werden können. Aus Definition 5.1 ist bekannt, dass zwei Programmstellen einen semantischen Zusammenhang besitzen, wenn sie einen gemeinsamen Zweck verfolgen. Werden die natürlichsprachlichen Bestandteile einer Programmstelle als Beschreibung ihres Zwecks aufgefasst, kann die Anzahl gemeinsam verwendeter oder bedeutungsgleicher Wörter zweier Programmelemente als Indiz für ihren semantischen Zusammenhang aufgefasst werden.
Bei der Suche nach semantischen Zusammenhängen in den verwandten Arbeiten werden zu Beginn die untersuchten Programmstellen aus dem Quelltext extrahiert und anschließend auf Textdokumente abgebildet. Die Textdokumente enthalten bspw. Kommentare, Bezeichner und statische Werte der jeweiligen Programmstellen. Im Folgenden werden häufig eingesetzte Vorverarbeitungsschritte sowie verschiedene Vorgehensweisen bei der Suche nach semantischen Zusammenhängen zwischen den entstandenen Textdokumenten diskutiert.

Vorverarbeitung

Die Vorverarbeitung von natürlicher Sprache hat zum Ziel, die vorhandenen Informationen vor der Analyse in eine einheitliche Form zu bringen, um bspw. von syntaktischen Feinheiten in der Sprache zu abstrahieren. In der Literatur sind die nachfolgenden Verarbeitungsschritte bei der Vorverarbeitung der natürlichsprachlichen Bestandteile im Quelltext weit verbreitet.

- **Zerteilen:** Bei diesem Schritt wird ein Textdokument auf eine Menge von zusammenhängenden Zeichenketten, z.B. Wörter, abgebildet. Hierzu kann ein Text an seinen Leer- oder Satzzeichen unterteilt werden. Nachfolgende Verarbeitungsschritte können somit einfach auf die einzelnen Terme eines Textes zuzugreifen.
- **Auftrennen zusammengesetzter Wörter:** Um Übereinstimmungen von Teilwörtern, z.B. bei den Methodenbezeichnern `getNewEmployee` und `saveEmployee`, erkennen zu können, kann es sinnvoll sein, zusammengesetzte Ausdrücke aufzutrennen. Die Auftrennung erfolgt in Anlehnung an existierende Namenskonventionen wie der Binnenmajuskel-Schreibweise. Durch Trennung an den Großbuchstaben kann bspw. die Methodenbezeichnung `saveEmployee` auf die Wörter *save* und *Employee* abgebildet werden. Da die Namenskonvention in einem Softwareprojekt frei wählbar ist, sollte ein solcher Vorverarbeitungsschritt in der Lage sein, unterschiedliche Namenskonventionen zu erkennen und entsprechend zu reagieren.
- **Normalisieren von Wortformen:** Die Wörter im Quelltext können in verschiedenen Varianten auftreten. Der Bezeichner für eine Variable, welche eine Liste repräsentiert, ist bspw. häufig im Plural geschrieben. Arbeiten wie [MB17, KNH$^+$17] und viele andere nutzen die Stammformreduktion (Siehe Abschnitt 2.3.1), um Wörter zu vereinheitlichen. Wörter wie *employees* oder *employee* werden hierbei bspw. auf den Wortstamm *employe* abgebildet. Ein Nachteil der Stammformreduktion ist, dass teilweise Wörter mit einer unterschiedlichen Bedeutung wie *executive* und *execution* auf den selben Wortstamm abgebildet werden. Das Lemmatisieren (Siehe Abschnitt 2.3.1) ist eine Alternative, welche diesen Nachteil nicht besitzt. Hierbei wird die Grundform eines Wortes in einem Wörterbuch nachgeschlagen. Folglich stellen jedoch unbekannte Wörter ein Problem bei der Lemmatisierung dar.
- **Entfernen von Stoppwörtern:** Einige Wörter im Sprachgebrauch wie *the*, *a* oder *to* tragen kaum zur Beschreibung des Zwecks einer Programmstelle bei, treten jedoch, aufgrund ihrer syntaktischen Rolle, vergleichsweise häufig in der natürlichen Sprache auf. Um sich bei der lexikalischen Analyse auf die semantisch wertvollen Wörter zu fokussieren werden Stoppwörter häufig entfernt. Die Menge der Stoppwörter wird in der Regel als Liste vorgegeben. Insbesondere bei der Analyse von Kommentaren kann dieser Vorverarbeitungsschritt sinnvoll sein, da die Beschreibung als Fließtext erfolgt und daher Stoppwörter enthält.

Neben diesen grundlegenden Vorverarbeitungsschritten führen einige verwandte Arbeiten zusätzliche Vorverarbeitungen durch. Corazza et al. bilden in ihrer Arbeit [CDMMS16] alle Buchstaben auf Kleinbuchstaben ab. Da nach dem Auftrennen zusammengesetzter Wörter viele Wörter mit einem Großbuchstaben beginnen, oder bspw. Bezeichner statischer Variablen häufig ausschließlich in Großbuchstaben geschrieben sind, ist dieses Vorgehen sinnvoll. Darüber hinaus entfernen Corazza et al. HTML-Elemente und andere Markierungen aus den Kommentaren im Quelltext.
In [LSLZ14] werden neben Stoppwörtern auch Wörter, die in mehr als 80% oder weniger als 2% der betrachteten Programmstellen auftreten, entfernt. Hierdurch sollen Begriffe, die sehr häufig genutzt werden oder sehr speziell sind, von der lexikalischen Analyse ausgeschlossen werden. Die Autoren sind der Meinung, dass sie aufgrund ihrer Vorkommensstruktur wenig Aussagekraft für den Zusammenhang zwischen Programmelementen

besitzen. Auch in der vorliegenden Arbeit sind Begriffe, die in nahezu allen Programmelementen auftreten, vermutlich wenig hilfreich, da sie zu allgemein sind, um einen Bezug zu einer konkreten Anforderung zu beschreiben. Seltene Begriffe können hingegen starke Hinweise auf semantische Zusammenhänge liefern und sollten bei der lexikalischen Analyse berücksichtigt werden. Ein Begriff, der bspw. nur zweimal im Quelltext auftaucht, kann ein starker Indikator für einen semantischen Zusammenhang sein.

Der Auflösung von Abkürzungen wurde bisher bei der Suche nach semantischen Zusammenhängen in der Literatur kaum Beachtung geschenkt. Dabei können Abkürzungen durchaus einen negativen Einfluss auf die Analyse der natürlichen Sprache haben, da sie unter Umständen dazu führen, dass das ausgeschriebene Wort und dessen Abkürzung als unterschiedliche Begriffe aufgefasst werden. Wünschenswert wäre hingegen, dass bspw. der Bezeichner `employee` und der Bezeichner `empl` als identische Begriffe aufgefasst werden. Jiang et al. zeigen in ihrer Arbeit [JLZZ18], dass es möglich ist, Abkürzungen in Parameterbezeichnungen von Methoden, mit einer Präzision zwischen 94% und 97% bei einer Ausbeute im Bereich zwischen 59% und 74%, aufzulösen. LINSEN [CDMM12], ein alternativer Ansatz zur Auflösung von Abkürzungen an beliebigen Stellen im Quelltext, löst durchschnittlich 59,1% der Abkürzungen korrekt auf. Im Hinblick auf eine generelle Auflösung von Abkürzungen besteht aufgrund der niedrigen Präzision die Gefahr, Abkürzungen falsch aufzulösen und damit den Fehler der lexikalischen Analyse zu erhöhen.

Semantische Zusammenhänge

Viele verwandte Arbeiten nutzen für den Vergleich der Beschreibungen von Programmelementen Verfahren aus der Informationsrückgewinnung. Hierbei werden die Programmelementbeschreibungen auf Vektoren abgebildet und anschließend der Kosinus des Winkels zwischen zwei Vektoren als Ähnlichkeitswert der Programmelemente herangezogen. Grundlage für die Abbildung einer Beschreibung auf einen Vektor ist die Term-Dokument-Matrix (Siehe Abschnitt 2.3.3). Sie umfasst Informationen darüber, wie häufig ein Wort in einem Dokument auftritt. Beim Vektorraummodell, wie unter anderem in [KNH+17] angewendet, werden die Einträge der Matrix über das TF-IDF-Maß (Siehe Abschnitt 2.3.3) gewichtet, um seltene und häufig vorkommende Wörter in der Menge der Dokumente entsprechend zu behandeln. Anschließend werden die Einträge der Matrix herangezogen, um die Beschreibungen der Programmelemente auf Vektoren abzubilden.
In der Arbeit [CDMMS16] von Corazza et al. kommt ebenfalls das Vektorraummodell zum Einsatz. Allerdings umfasst ihre Term-Dokument-Matrix zusätzliche Informationen darüber, an welchen Stellen im Quelltext die Wörter aufgetreten sind. Daher wird beim Vergleich von Programmelementbeschreibungen die Position der Begriffe berücksichtigt. Folglich spielt es eine Rolle, ob ein Begriff bspw. in einem Kommentar oder als Teil eines Bezeichners verwendet wurde. Kommt ein Begriff bspw. in der Signatur zweier Methoden vor, kann das ein stärkeres Indiz für einen semantischen Zusammenhang sein, als wenn der Begriff an unterschiedlichen Stellen in beiden Methoden auftritt. Die Einbeziehung der Position kann daher Vorteile mit sich bringen. Gleichzeitig können jedoch Nachteile entstehen, wenn Gemeinsamkeiten zwischen Wörtern an verschiedenen Stellen nicht berücksichtigt werden.
Ein grundsätzlicher Schwachpunkt des Vektorraummodells ist, dass die Bewertung der Ähnlichkeit auf der Verwendung von gleichen Ausdrücken basiert. Zwei Texte, die dasselbe Konzept beschreiben, jedoch unterschiedliche Wörter nutzen, können folglich nicht als ähnlich erkannt werden [CEBE09]. Kuhn et al. verwenden in ihrer Arbeit [KDG07] Latent Semantic Indexing [MRS08], kurz LSI, um mit Synonymen und Polysemen umgehen zu können. LSI setzt auf dem Vektorraummodell auf und versucht durch die Dimensionsreduktion der Term-Dokument-Matrix robuster gegenüber den gewählten Formulierungen zu werden. Kuhn et al. geben an, dass die Ergebnisse von LSI besser werden, je größer

das Vokabular und die Textgröße sind. Im Hinblick auf die charakteristischen Eigenschaften der natürlichen Sprache im Quelltext und der damit verbundenen geringen Größe des Vokabulars ist es fragwürdig, ob der Einsatz von LSI sinnvoll ist.

Einige Arbeiten aus dem Bereich der Konzeptlokalisierung nutzen Themenmodellierungen, um Zusammenhänge zwischen den natürlichsprachlichen Beschreibungen von Programmelementen zu erkennen. Der Ansatz [LSLZ14] von von Liu et al. basiert bspw. auf der Latent Dirichlet Allocation [BNJ03]. Liu et al. ordnen Programmelementbeschreibungen Verteilungen zu, die eine Aussage darüber treffen, aus welchen Themen sie sich zusammensetzen. Zwei Programmelemente werden anschließend als zusammenhängend betrachtet, wenn sie den gleichen Themen zugeordnet wurden. Ein zentraler Nachteil bei diesem Vorgehen ist, dass die Anzahl der Themen manuell vorgegeben werden muss. Bei der Analyse von Softwaresystemen liegt diese Information im Allgemeinen jedoch nicht vor.
Einen alternativen Ansatz zur Themenmodellierung zeigen Mahmoud und Bradshaw in ihrer Arbeit [MB17]. Sie berechnen zu Beginn die paarweise Ähnlichkeit aller Terme im Vokabular des Quelltextes. Die Berechnung basiert auf der Anzahl der Dokumente, in denen zwei Terme gemeinsam vorkommen. Auf der Basis der Termähnlichkeiten wird anschließend eine hierarchische Clusteranalyse durchgeführt. Die hierdurch entstehenden Termcluster repräsentieren die Themen im Quelltext. Der Zusammenhang eines Programmelements mit einem Thema wird durch den Vergleich seiner natürlichsprachlichen Beschreibung mit den Termen im Themencluster bewertet. Die Zugehörigkeit eines Programmelements zu einem Themencluster wird mit einem Wert zwischen 0 und 1 ausgedrückt. Ein Vorteil des Ansatzes von Mahmoud und Bradshaw ist, dass die Anzahl der Themen nicht vorgegeben werden muss. Ein zentraler Nachteil hingegen ist, dass die Berechnung der Ähnlichkeiten keine Abkürzungen und Synonyme berücksichtigt, wodurch vermutlich viele zusammenhängende Begriffe nicht erkannt werden können.

Neben regelbasierten Ansätzen können Verfahren aus dem Bereich des maschinellen Lernens genutzt werden, um Programmelemente zu vergleichen. Eine Möglichkeit hierfür bieten Methoden zur Worteinbettung (Siehe Abschnitt 2.3.4). Hierbei lernt ein neuronales Netz unüberwacht Wortrepräsentationen auf der Basis eines Textkorpus. Das Ziel solcher Verfahren ist es, Wörtern, die eine ähnliche Bedeutung haben bzw. in ähnlichen Kontexten verwendet werden, auf ähnliche Vektoren abzubilden. Ein Vorteil von Worteinbettungsmethoden ist, dass bei einem ausreichend großen Trainingskorpus Polyseme und Synonyme automatisch mit berücksichtigt werden.
Tufano et al. nutzen dieses Vorgehen in ihrer Arbeit [TWB+18] zur Identifikation von Quelltextduplikaten. Für die Bewertung des semantischen Zusammenhangs werden Programmelemente auf Vektoren abgebildet. Im ersten Schritt werden die Bezeichner eines Programmelements über die Worteinbettung auf eine Menge von Vektoren abgebildet. Im zweiten Schritt wird diese Vektormenge von einem weiteren neuronalen Netz auf einen einzigen Vektor abgebildet. Zur Bestimmung der Ähnlichkeit wird der Kosinus des Winkels zwischen Programmelementvektoren berechnet. Tufano et al. berücksichtigen lediglich die Bezeichner bei ihrer lexikalischen Analyse. Diese sind häufig aus mehreren Wörtern zusammengesetzt oder enthalten Abkürzungen. Für das Training ihres Worteinbettungsmodells nutzen sie den Quelltext des Softwareprojekts und verzichten auf die Einbeziehung eines externer Textkorpus. Hierdurch geht der Vorteil der Worteinbettung verloren, externes Wissen über die Wortzusammenhänge einzubeziehen. Im Hinblick auf die vorliegende Arbeit ist die Einbeziehung von externen Wissensquellen hingegen sinnvoll, da bspw. in Kommentaren allgemein gebräuchliche Wörter zu finden sind.
Efstathiou und Spinellis verwenden in ihrer Arbeit [ES19] Worteinbettungen, um Softwareprojekte miteinander zu vergleichen. Ihr Worteinbettungsmodell wurde auf Texten aus verschiedenen Softwareprojekten trainiert. Dies kann ein Vorteil darstellen, da das Modell Wortbedeutungen aus dem Kontext der Softwareentwicklung lernt. Allerdings um-

fasst ihr Modell vermutlich kaum Informationen über Zusammenhänge zwischen Wörter aus dem allgemeinen Sprachgebrauch, da ausschließlich Quelltext für die Erzeugung des Trainingskorpus genutzt wird. Dies kann ein Nachteil sein, wenn sich die gelernten Bedeutungen von Wörter aus den Softwareprojekten von der Bedeutung im herkömmlichen Sprachgebrauch unterscheidet. Ein Vorteil ihres Vorgehens hingegen ist, dass die Wahrscheinlichkeit bei der Analyse von Quelltext einen zusammengesetzten Bezeichner oder eine Abkürzungen bereits aus den Trainingsdaten zu kennen, erhöht ist. Es ist jedoch fragwürdig, ob die Bedeutung von Bezeichnern zwischen Softwareprojekten übertragbar ist. Des Weiteren kann nicht davon ausgegangen werden, dass alle Bezeichner im Trainingskorpus des Worteinbettungsmodells vorkommen, da diese in der Regel projektspezifisch sind.

Insgesamt ist die Vorgehensweise vieler Ansätze zur lexikalischen Analyse von Quelltext ähnlich. Auffällig ist, dass obwohl sich die Charakteristik von natürlicher Sprache in Anweisungen und Kommentaren mitunter sehr stark unterscheidet, viele bestehende Ansätze bei ihrer Analyse nicht zwischen ihnen differenzieren. Darüber hinaus besitzen die bestehenden Ansätze für die lexikalische Analyse Optimierungspotential im Hinblick auf die Verarbeitung von Abkürzungen. Die folgenden Abschnitte analysieren verschiedene Möglichkeiten zur Optimierung der lexikalischen Analyse. Der Fokus liegt hierbei auf dem Einsatz von Worteinbettungstechniken. Sie besitzen den Vorteil, dass sie den Verwendungskontext eines Wortes mit berücksichtigen. Im Vergleich zum Vektorraummodell ist eine lexikalische Analyse auf Basis von Worteinbettungen folglich robuster gegenüber variierenden Formulierungen. Darüber hinaus erlauben Worteinbettungstechniken die Einbeziehung von externem Wissen über die Zusammenhänge von Wörtern durch die Wahl passender Trainingsdaten.

5.3.3.2 Worteinbettung

Die grundsätzliche Funktionsweise von Worteinbettungsverfahren ist in Abschnitt 2.3.4 beschrieben. Im Folgenden sollen verschiedene Möglichkeiten beim Einsatz von Worteinbettungstechniken analysiert werden.
Aktuelle Verfahren arbeiten auf Wort- oder Zeichenebene. Die Arbeit [MSC$^+$13] von Mikolov et al. betrachtet Wörter als kleinste Texteinheit. Ihr Modell lernt Vektorrepräsentationen von Wörtern auf Basis ihrer Umgebungen. Wörter, die häufig von den gleichen Wörtern umgeben sind, werden auf ähnliche Vektoren abgebildet. Ein Schwachpunkt dieses Vorgehens besteht im Umgang mit unbekannten Wörtern. Für Wörter, die nicht in den Trainingsdaten vorkommen, können in der Regel keine adäquaten Vektoren erzeugt werden. Eine alternative Herangehensweise zur Worteinbettung beschreiben Bojanowski et al. in ihrer Arbeit [BGJM17]. Ihr Ansatz beruht auf N-Grammen als kleinste Texteinheit und ihr Modell lernt Vektorrepräsentationen für N-Gramme anstelle von Wörtern. Bei der Abbildung von Wörtern auf Vektoren, wird die Summe der N-Gramm-Vektoren eines Wortes gebildet. Ein Vorteil des Verfahrens ist, dass Vektoren für unbekannte Wörter, auf der Basis der enthaltenen N-Gramme, gebildet werden können. Auch die Verarbeitung von seltenen oder falsch geschriebenen Wörtern profitiert von der Einbeziehung der N-Gramme. Im Hinblick auf das mit unter projektspezifische Vokabular in Softwareprojekten und der damit verbundenen erhöhten Wahrscheinlichkeit unbekannte Wörter zu verarbeiten, bietet sich der Einsatz von Worteinbettungsmodellen auf der Basis von N-Grammen an.

Ein weiterer Einflussfaktor auf die Güte der Worteinbettungstechnik sind die verwendeten Trainingsdaten. Im Hinblick auf den Einsatz einer Worteinbettungstechnik in dieser Arbeit, existieren verschiedene Möglichkeiten. Die erste Möglichkeit besteht darin, das Modell auf der, im Quelltext enthaltenen, Textmenge zu trainieren. Ein Vorteil bei diesem Vorgehen ist, dass projektspezifische Zusammenhänge zwischen Wörtern gelernt werden können. Aufgrund der geringen Trainingsdatengröße und der Einschränkung auf die Informationen im Quelltext kann es vorkommen, dass Wörter, die im allgemeinen Sprachgebrauch als

zusammenhängend betrachtet werden, nicht adäquat vom Modell verarbeitet werden können.
Eine potentielle Lösung hierfür stellt die Einbeziehung von externem Wissen dar. Da Kommentare und Zeichenketten häufig Beschreibungen in Form von Fließtext enthalten, beinhalten sie viele Wörter, die sich auch in anderen Texten, z.B. auf Wikipedia[2], finden lassen. Eine Möglichkeit allgemeine Zusammenhänge zwischen Wörtern zu berücksichtigen besteht darin, das Modell auf einem solchen externen Textkorpus zu trainieren. Problematisch ist dieses Vorgehen, wenn sich die Bedeutung der verwendeten Wörter im Softwareprojekt von der Bedeutung im Trainingskorpus unterscheidet oder viele der Wörter aus dem Softwareprojekt nicht im Trainingskorpus vorkommen. Daher sollte ein Trainingskorpus Texte aus vielen unterschiedlichen Themengebieten beinhalten, um die Verwendung von Wörter in unterschiedlichen Kontexten zu berücksichtigen. Darüber hinaus wäre es denkbar, Texte aus dem Bereich der Softwareentwicklung für das Training des Worteinbettungsmodells heranzuziehen. Eine Möglichkeit hierfür bieten bspw. Texte aus GitHub oder StackOverflow[3]-Beträgen. Sie haben den Vorteil, dass sie die Verwendung verschiedener Wörter im Kontext der Softwareentwicklung darstellen. Des Weiteren umfasst das Vokabular solcher Texte vermutlich viele Abkürzungen, was im Bezug auf die Verarbeitung von natürlicher Sprache im Quelltext ein Vorteil sein kann. Allerdings kann die Verwendung von Texten aus GitHub oder StackOverflow auch Nachteile mit sich bringen. Da Softwareprojekte in der Regel domänenspezifische Zusammenhänge abbilden, ist die Übertragbarkeit von Bedeutungszusammenhängen zwischen verschiedenen Softwareprojekten nicht notwendigerweise gegeben. Es besteht die Gefahr, dass Wortbedeutungen gelernt werden, die dem allgemeinen Verständnis eines Wortes widersprechen. Hinzu kommt, dass das Vokabular von Softwarebeschreibungen in der Regel klein ist, da die enthaltene natürliche Sprache sich aus vereinzelten Worten, kurzen Fließtexten und vielen Wortwiederholungen zusammensetzt. Infolgedessen ist das Vokabular eines auf Software-Texten trainierten Modells vermutlich kleiner und einseitiger als bei einem auf Wikipedia-Texten trainierten Modell, wodurch die Wahrscheinlichkeit erhöht ist, bei der lexikalischen Analyse unbekannte Wörter zu verarbeiten.
Eine weitere Möglichkeit zur Zusammenstellung eines Trainingskorpus wäre eine Kombination aus allgemeinen natürlichsprachlichen Texten, wie aus Wikipedia und Texten aus dem Bereich der Softwareentwicklung, z.B. der StackOverflow-Beiträge. Im Idealfall können somit sowohl allgemeine Bedeutungen als auch spezifische Charakteristiken von Softwarebeschreibungen gelernt werden.
Alternativ zum Training auf einem externen Korpus oder dem Training auf der Textmenge des analysierten Softwareprojekts besteht die Möglichkeit, beide Herangehensweisen zu kombinieren. Ein Modell kann hierzu auf einem externen Textkorpus vortrainiert und anschließend auf dem Textkorpus des Softwareprojekts nachtrainiert werden. Ein Vorteil dieser Kombination wäre, dass sowohl allgemeine Zusammenhänge als auch projektspezifische Besonderheiten berücksichtigt werden könnten. Eine Schwierigkeit stellt jedoch die vergleichsweise geringe Größe der Textmenge eines Softwareprojekts dar. Die Herausforderung läge daher in einer adäquaten Gewichtung der beiden Trainingsdatensätze, sodass die Textmenge aus einem Softwareprojekt einen Einfluss auf die Vektorrepräsentationen hat. Eine weitere Problematik liegt in der Tatsache, dass bei diesem Vorgehen vor jeder lexikalischen Analyse das Worteinbettungsmodell nachtrainiert und die Trainingsparameter auf die Textmenge des zu analysierenden Softwareprojekts angepasst werden müssen.

5.3.3.3 Abkürzungen

Bei der Analyse der im Quelltext vorhandenen Sprache können Abkürzungen ein Problem darstellen. Wenn Abkürzungen nicht oder nur selten im Trainingskorpus eines Worteinbet-

[2] `https://www.wikipedia.de/`, zuletzt besucht am 15.12.2019
[3] `https://stackoverflow.com/`, zuletzt besucht am 15.12.2019

tungsmodells vorkommen, kann nicht davon ausgegangen werden, dass die Vektorabbildung einer Abkürzung nahe bei der Vektorabbildung des ausgeschriebenen Wortes liegt. Um zu gewährleisten, dass Wörter wie *empl* und *employee* dennoch als identisch angesehen werden, wäre eine Identifikation und Auflösung von Abkürzungen im Rahmen der Vorverarbeitung notwendig. Für Standardabkürzungen wie *tmp* oder *db*, deren Verwendung weit verbreitet ist, können Abkürzungswörterbücher wie der SEThesaurus aus [CCZX19] herangezogen werden. Es basiert auf den Texten in StackOverflow und umfasst ca. 19.000 Abkürzungen. Ein Vorteil beim Einsatz eines Wörterbuchs ist die zu erwartende hohe Präzision mit der Abkürzungen aufgelöst werden, da vor allem weit verbreitete oder häufig genutzte Abkürzungen im Wörterbuch enthalten sind. Darüber hinaus ist jedoch eine vergleichsweise geringe Ausbeute zu erwarten, da projektspezifische Abkürzungen vermutlich nicht enthalten sind und somit nicht aufgelöst werden können. Beim Einsatz bestehender Ansätze zur Auflösung von Abkürzungen, wie die Arbeit [CDMM12] von Corazza et al., die durchschnittlich 59,1% der Abkürzungen korrekt auflösen, ist die Gefahr groß, viele falsche Informationen bei der Analyse einzubeziehen. Dies ist ein großer Nachteil, da Abkürzungen, die falsch aufgelöst werden, das Potential besitzen, die Semantik eines Textes zu verfälschen.

Eine alternative Möglichkeit mit Abkürzungen umzugehen wäre eine Zeichenkettenanalyse. Hierbei können Abkürzungen mit anderen Wörtern verglichen werden, ohne das eine Auflösung erforderlich ist. Ein derartiger Vergleich wäre darüber hinaus nicht nur für Abkürzungen, sondern für alle, dem Worteinbettungsmodell unbekannten, Wörter denkbar. Da es vorkommen kann, dass Wörter mit unterschiedlichen Bedeutungen eine ähnliche Zeichenkette besitzen, besteht beim Einsatz von Zeichenkettenvergleichen die Gefahr, mehr Zusammenhänge zu erkennen, als tatsächlich vorhanden sind. Folglich ist es sinnvoll, einem Zeichenkettenvergleich lediglich eine geringe Aussagekraft zuzusprechen, um die Unsicherheit zu berücksichtigen.
Ein einfacher Ansatz zur Bewertung der Ähnlichkeit zweier Zeichenketten wäre bspw. das Verhältnis aus der Schnittmenge der, in beiden Zeichenketten, enthaltenen Zeichen und der Gesamtzahl an unterschiedlichen Zeichen. Offensichtlich bringt dieses Vorgehen jedoch einige Nachteile mit sich, da die Position der Zeichen nicht berücksichtigt wird. Auch Ansätze, welche die längste gemeinsame Sequenz (engl. longest common subsequence) zweier Wörter betrachteten, eignen sich für die Behandlung von Abkürzungen nicht, da Abkürzungen häufig aus wenigen Buchstaben bestehen und somit in der Regel keine langen gemeinsamen Sequenzen existieren.
Ein populäres Distanzmaß für den Vergleich von Zeichenketten ist die Levenshtein-Distanz (Siehe Abschnitt 2.3.5). Sie berechnet die minimale Anzahl an Operationen die benötigt werden, um ein Wort in ein anderes Wort zu überführen. Eine Operation kann entweder ein Zeichen einfügen, löschen oder ersetzen. Ein Problem der Levenshtein-Distanz, im Hinblick auf die Behandlung von Abkürzungen ist, dass sie sich sehr stark auf die Unterschiede zweier Zeichenketten fokussiert. Für die beiden Wörter *employee* und *empl* ist die Levenshtein-Distanz bspw. gleich vier, da die letzten vier Zeichen des Wortes *employee* gelöscht werden müssen, um daraus das Wort *empl* zu machen. Die gleiche Distanz besitzen auch die Wörter *aaaa* und *bbbb*. Offensichtlich sollte den Wörtern *empl* und *employee* jedoch eine höhere Ähnlichkeit zugeschrieben werden, da sie sich ein gemeinsames Präfix teilen.
Eine Metrik, die sich auf die Gemeinsamkeiten zweier Zeichenketten fokussiert, ist die Jaro-Winkler-Ähnlichkeit (Siehe Abschnitt 2.3.5). Sie basiert auf der Anzahl gemeinsamer Zeichen, wobei auch die Position der Zeichen berücksichtigt wird. Setzt sich ein Wort aus Zeichen zusammen, die an der gleichen Position auch in einem anderen Wort auftreten, ist die Ähnlichkeit maximal. Zusätzlich werden die Präfixe der betrachteten Wörter untersucht. Ein gemeinsames Präfix trägt dazu bei, die Ähnlichkeit weiter zu erhöhen. Ein Nachteil der Jaro-Winkler-Ähnlichkeit zeigt sich bei der Bewertung von Teilwörtern und

Suffixen. Als Beispiel dient das Wort *workshop*. Wohingegen das Teilwort *shop* mit einer Ähnlichkeit von 0.46 bewertet wird, wird dem fiktiven Begriff *wx* eine Ähnlichkeit von 0.54 zugesprochen. Der Grund für die unterschiedliche Bewertung ist, dass das Wort *wx* ein gemeinsames Präfix mit dem Wort *workshop* besitzt. Intuitiv besitzt das Wort *shop* jedoch eine größere Ähnlichkeit zum Wort *workshop*, als der Begriff *wx*.
Ein Ähnlichkeitsmaß, welches für die Bewertung von Suffixen und Teilwörtern besser geeignet ist, ist der FuzzyScore (Siehe Abschnitt 2.3.5). Das Maß basiert auf der Anzahl gemeinsamer Zeichen, wobei eine Übereinstimmung aufeinander folgender Zeichen stärker gewichtet wird. Bezogen auf das obige Beispiel wird die Ähnlichkeit des Wortes *shop* vom FuzzyScore mit 0.45 höher eingestuft, als die Ähnlichkeit von 0.25 des Wortes *wx*. Allerdings besitzt auch der FuzzyScore Schwachstellen. Beispielsweise wird dem Akronym *db* lediglich eine Ähnlichkeit von 0.09 zu dem Wort *database* zugesprochen. Im Vergleich dazu liegt die Ähnlichkeit bei einer Berechnung mit der Jaro-Winkler-Distanz bei 0.78.
Insgesamt bieten Zeichenkettenvergleiche eine gute Möglichkeit unbekannte Wörter zu behandeln. Aufgrund der unterschiedlichen Eigenschaften verschiedener Distanzmaße erscheint es sinnvoll, mehrere Maße bei der Bewertung der Ähnlichkeit zweier Zeichenketten zu kombinieren. So können die Stärken der einzelnen Verfahren ausgenutzt werden. Insbesondere die Jaro-Winkler-Ähnlichkeit und der FuzzyScore sind hierbei vielversprechend. Da es bei der Analyse von Abkürzungen und unbekannten Wörtern nicht darum geht, grundsätzliche Ähnlichkeiten zwischen Zeichenketten festzustellen, sondern zu beurteilen, ob zwei Zeichenketten dasselbe Wort beschreiben, ist es sinnvoll, eine Schranke festzulegen, die den Wert vorgibt, ab dem davon ausgegangen wird, dass zwei Zeichenketten das selbe Wort beschreiben. Diese Schranke sollte individuell für jedes genutzte Distanzmaß definiert werden, um auf dessen individuelle Besonderheiten eingehen zu können. Je höher der Schwellwert ist, desto sicherer ist es, dass es sich bei zwei Zeichenketten um dasselbe Wort handelt. Allerdings kann ein zu hoher Schwellwert dazu führen, dass Abkürzungen unter Umständen nicht erkannt werden können.

5.3.3.4 Vorverarbeitung

Bei der Verarbeitung von natürlicher Sprache existieren eine Reihe von Verarbeitungsschritten, die genutzt werden können, um die extrahierten Texte aus dem Quelltext zu bereinigen und für die spätere Verarbeitung aufzubereiten.
Ein möglicher Vorverarbeitungsschritt betrifft zusammengesetzte Wörter. Durch das Auftrennen zusammengesetzter Wörter wird der Fokus der späteren Analyse auf die Bedeutung der Teilwörter gesetzt. Im Hinblick auf den Einsatz von Worteinbettungen ist dies ein Vorteil, da die Teilwörter in der Regel nicht projektspezifisch sind und daher von z.B. externem Wissen über Zusammenhänge in der natürlichen Sprache profitiert werden kann. Ein Beispiel liefern die Wörter *goodEmployee* und *bestEmployee*. Wohingegen bei einer Auftrennung der zusammengesetzten Begriffe sowohl von der Ähnlichkeit zwischen den Wörtern *good* und *best* als auch von der gemeinsamen Verwendung des Wortes *Employee* profitiert werden kann, wird die Bedeutung der einzelnen Teilwörter bei einem Verzicht auf die Auftrennung nicht explizit berücksichtigt. Eine Verarbeitung von zusammengesetzten Begriffen als Ganzes ist hingegen abhängig von deren Vorkommen im Trainingskorpus des Worteinbettungsmodells, was aufgrund der, in der Regel, projektspezifischen Nutzung von zusammengesetzten Wörtern ein Nachteil ist. Allerdings kann auch eine Auftrennung Nachteile mit sich bringen, wenn sich z.B. die Bedeutung der Teilwörter von der Bedeutung des zusammengesetzten Begriffs unterscheidet. Um die Information, dass es sich um ein zusammengesetztes Wort handelt, beizubehalten, wäre es möglich, die Auftrennung nach der Verarbeitung mit dem Worteinbettungsmodell wieder rückgängig zu machen, indem bspw. die Vektorabbildungen der einzelnen Teilwörter wieder zusammengesetzt werden. So kann gewährleistet werden, dass ein zusammengesetztes Wort, genau wie ein einzelnes Wort, behandelt und auf einen Vektor abgebildet wird.

Im Gegensatz zum allgemeinen englischen Sprachgebrauch unterliegt die Groß- und Kleinschreibung im Quelltext weiteren Regeln. Variablen- und Methodenbezeichner beginnen bspw. immer mit einem Kleinbuchstaben, unabhängig davon, um welches Wort es sich handelt. Bei zusammengesetzten Wörtern wird häufig die Binnenmajuskel-Schreibweise eingesetzt, bei der die einzelnen Wörter durch einen Großbuchstaben getrennt werden. Bezeichner von statischen Variablen werden darüber hinaus häufig ausschließlich mit Großbuchstaben geschrieben. Typnamen beginnen grundsätzlich mit einem Großbuchstaben. Insgesamt ist die Schreibweise im Quelltext daher nicht an die Wortart oder die Regeln der natürlichen Sprache geknüpft, sondern im Bezug auf Bezeichner z.B. abhängig von der Art des Programmelements, welches beschrieben wird oder der verwendeten Namenskonvention. Infolgedessen ist es bei der Analyse der natürlichen Sprache im Quelltext sinnvoll sicherzustellen, dass die Groß- und Kleinschreibung keinen Einfluss auf die Bedeutung eines Wortes besitzt. Es empfiehlt sich daher, alle Großbuchstaben in einem Vorverarbeitungsschritt auf Kleinbuchstaben abzubilden.

Normalisierungen, wie die Stammformreduktion oder das Lemmatisieren, werden häufig als Vorverarbeitungsschritt eingesetzt, um Wörter zu vereinheitlichen und die Suche nach identischen Wörtern in Texten zu vereinfachen. Grundsätzlich gehen hierbei jedoch Informationen verloren. Der Name `employee` eines Objekts der Klasse `Employee`, hat bspw. eine andere Bedeutung als die Bezeichnung `employees`, die auf eine Liste von Objekten des Typs `Employee` verweist. Durch eine Normalisierung der Wortformen wäre keine Unterscheidung der beiden Bezeichner möglich. Ein Worteinbettungsmodell ist in der Lage, die in der Wortform enthaltenen Information zu berücksichtigen. Folglich empfiehlt es sich, auf einen derartigen Vorverarbeitungsschritt zu verzichten.

Wörter, die im natürlichen Sprachgebrauch häufig vorkommen und selbst kaum eine semantische Aussagekraft besitzen, können zu Fehlinterpretationen führen. Das Stoppwort *to* ist ein Beispiel hierfür. Es kommt in der natürlichen Sprache sehr häufig vor und findet sich dadurch bspw. in vielen Kommentaren wieder. Dadurch können unter Umständen Zusammenhänge erkannt werden, die eigentlich nicht existent sind. Analog verhält es sich mit Zahlen, die ebenfalls kontextunabhängig in der natürlichen Sprache genutzt werden. Um Fehlinterpretationen bei der lexikalischen Analyse zu vermeiden und den Fokus der Analyse auf semantisch wertvolle Begriffe zu lenken, ist es sinnvoll, Zahlen sowie allgemein genutzte Wörter, die isoliert betrachtet kaum einen Beitrag zur Semantik eines Textes leisten, vorab zu entfernen. Bestehende Ansätze nutzen hierfür vordefinierte Listen von Stoppwörtern. Da diese Listen in der Regel für die Verarbeitung von allgemeinen natürlichsprachlichen Texten erstellt wurden, werden die Besonderheiten von Quelltext nicht berücksichtigt. Eine mögliche Ergänzung bestehender Stoppwortlisten stellen die Schlüsselwörter der im Softwareprojekt genutzten Programmiersprache dar. Da Schlüsselwörter, ähnlich wie Stoppwörter, isoliert betrachtet keine Rückschlüsse auf die spezifische Semantik eines Textes zulassen, wirkt sich deren Entfernung vermutlich positiv auf die Ergebnisse der lexikalischen Analyse aus.
Im Hinblick auf die Bezeichner im Quelltext ergeben sich weitere Wörter, die häufig auftreten und kaum einen semantischen Mehrwert liefern. Hierzu zählen allgemeine Begriffe wie *var* oder sehr generische Variablenbezeichnungen wie *v*, die lediglich aus einem Buchstaben bestehen. Auch Präfixe wie der Buchstabe *I* zur Kennzeichnung von Schnittstellenbezeichnungen sind, aufgrund ihrer Allgemeinheit, bei der Suche nach semantischen Zusammenhängen nicht relevant.

Aus Abschnitt 5.3.3 geht hervor, dass Kommentare neben natürlicher Sprache auch strukturelle Informationen wie Markierungen oder Formatierungselemente enthalten können. Formatierungselemente verbessern die Lesbarkeit der Kommentare für den Menschen. Sie sind für die Bedeutung der Beschreibung jedoch irrelevant, weshalb alle Formatierungselemente aus den Kommentaren entfernt und nur der Text beibehalten werden kann.

Markierungen besitzen im Gegensatz zu Formatierungselementen eine semantische Bedeutung. Explizite Verweise auf Programmelemente könnten bspw. genutzt werden, um Kommentare zu identifizieren, die auf die gleichen Programmelemente referenzieren. Allerdings ist die Schlussfolgerung eines semantischen Zusammenhangs aufgrund solcher Verweise mit einer hohen Unsicherheit verbunden, da ein Verweis auch zur Abgrenzung oder zur Erklärung genutzt werden kann, ohne dass ein semantischer Zusammenhang zum verwiesenen Programmelement existiert. Die Verwendung von Programmelementverweisen in Kommentaren, als Indiz für semantische Zusammenhänge, ist folglich mit der Gefahr verbunden, den Fehler der lexikalischen Analyse zu erhöhen. Es bietet sich daher an, lediglich den Text innerhalb der Markierungen beizubehalten, um die sprachliche Bedeutung dieser zu berücksichtigen und die Strukturinformation zu entfernen. Weitere Markierungen wie `@author`, `@version`, `@since` und `@deprecated` beschreiben Metainformationen und haben keine Relevanz für die Beschreibung der Absicht eines Programmelements. Sie können komplett aus den Kommentaren entfernt werden.
Kommentare die mit den Wörtern *todo* oder *fixme* markiert sind stellen eine Besonderheit dar. Entwickler nutzen diese Begriffe in der Regel, um Kommentare zu markieren, die offene Aufgaben im Quelltext beschreiben. Um Zusammenhänge zwischen Texten aufgrund dieser Wörter zu vermeiden, ist es sinnvoll, sie vorab aus den Kommentaren zu entfernen. Darüber hinaus wäre es denkbar, ganze Kommentare, die mit *todo* oder *fixme* markiert sind, zu entfernen, da sie unter Umständen sehr allgemeine Informationen wie in *todo: null check* enthalten können, die für die individuelle Beschreibung eines Programmelements wenig hilfreich sind. Allerdings können *todo*- oder *fixme*-Kommentare auch ausführliche Beschreibungen neuer oder anzupassender Funktionalitäten umfassen. Solche Informationen beziehen sich speziell auf ein Programmelement und besitzen das Potential die lexikalische Analyse zu bereichern. Folglich empfiehlt es sich für *todo*- oder *fixme*-Kommentare zu prüfen, ob sie sich auf die Funktionalität einer Programmstelle beziehen oder eine allgemeine Aktion beschreiben. Eine Möglichkeit hierfür wäre, die Wörter im Kommentar mit den Bezeichnern im Quelltext zu vergleichen. Verweist ein *todo*- oder *fixme*-Kommentar auf einen Bezeichner, ist die Wahrscheinlichkeit höher, dass es sich um eine spezifische Beschreibung handelt. Allerdings ist dieses Vorgehen mit einer hohen Unsicherheit verbunden, da Kommentare, die wie *todo: rename employee* einen Bezeichner enthalten, nicht notwendigerweise zur Beschreibung der Funktionalität einer Programmstelle beitragen. Dennoch kann diese Heuristik einen Teil der im Quelltext vorhandenen irrelevanten Informationen herausfiltern.
Eine besondere Art von Kommentar, die häufig im Quelltext auftritt, ist die Lizenzbeschreibung. Sie wird in der Regel zu Beginn einer Datei angegeben und beschreibt die Lizenz unter der der Quelltext genutzt werden kann. Für die Suche nach semantischen Zusammenhängen ist diese Art von Kommentar nicht von Bedeutung, weshalb es sinnvoll ist, solche Kommentare in einem Vorverarbeitungsschritt herauszufiltern. Für die Identifikation solcher Kommentare wäre es denkbar, nach Wörtern wie *copyright* oder *license* zu suchen. Diese Wörter sind vergleichsweise häufig in Lizenzbeschreibungen enthalten und bieten sich daher als Indikator an.
Des Weiteren können Kommentare Quelltext enthalten. Da im Rahmen der lexikalischen Analyse nicht bestimmt werden kann, welchen Zweck auskommentierter Quelltext besitzt, ist dessen Einbeziehung mit dem Risiko verbunden, veraltete oder irrelevante Informationen miteinzubeziehen. Folglich erscheint es sinnvoll, Kommentare, die aus Quelltext bestehen, zu entfernen. Um auskommentierten Quelltext zu erkennen, kann bspw. die Dichte der, im Hinblick auf eine Programmiersprache spezifischen, Schlüsselwörter berechnet werden. Bei Kommentaren, die zu einem hohen Anteil aus Schlüsselwörtern bestehen, kann daraufhin angenommen werden, dass es sich um auskommentierten Quelltext handelt.
Teilweise geben Entwickler in Kommentaren Internetadressen an, um auf die Quelle eines Quelltextausschnittes oder weiterführende Beschreibungen zu referenzieren. Internetadres-

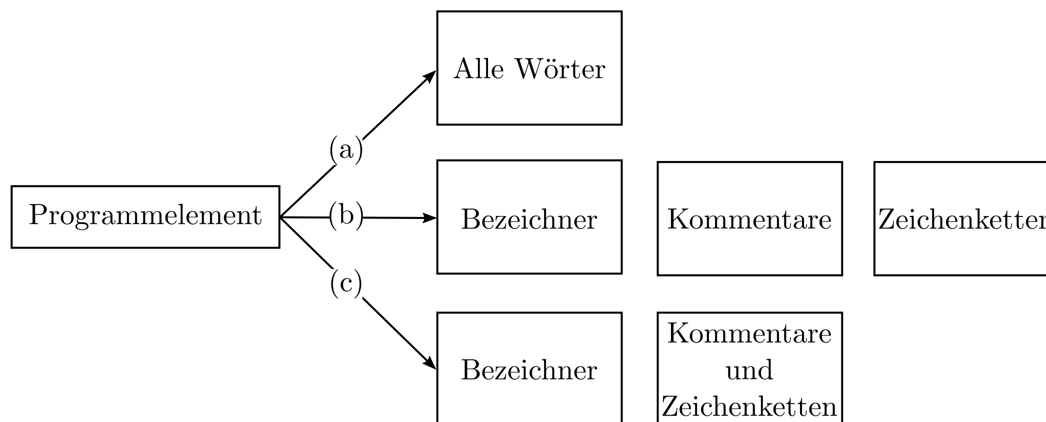


Abbildung 5.8: Möglichkeiten der Dokumentabbildung.

sen sind häufig aus kryptischen Zeichenketten zusammengesetzt und liefern keine relevanten Informationen für die lexikalische Analyse. Folglich erscheint es sinnvoll, Verweise auf Internetseiten aus den Kommentaren zu entfernen.

5.3.3.5 Semantische Zusammenhänge

Das Fundament der lexikalischen Quelltextanalyse sind Dokumentabbildungen der Programmelemente im Quelltext. Ein Programmelementdokument kann die Wörter aus den Kommentaren, Zeichenketten und Bezeichnern beinhalten. Abbildung 5.8 zeigt drei grundsätzliche Möglichkeiten bei der Dokumentabbildung. Die erste Möglichkeit, in der Abbildung mit (a) gekennzeichnet, vereint alle natürlichsprachlichen Bestandteile eines Programmelements in einem Dokument. Eine Herausforderung bei diesem Vorgehen ist der Umgang mit der unterschiedlichen Charakteristik von Bezeichnern und Kommentaren oder Zeichenketten. Wohingegen sich in Bezeichnern einzelne Wörter finden, umfassen Kommentare und Zeichenketten in der Regel Fließtext. Folglich ist der Textumfang aus Kommentaren und Zeichenketten erheblich größer als die Textmenge aus den Bezeichnern. Eine gemeinsame Betrachtung bürgt daher die Gefahr den Einfluss von Bezeichnern zu vernachlässigen. Diesbezüglich bringt die zweite Möglichkeit aus Abbildung 5.8, gekennzeichnet mit (b), einige Vorteile mit sich, da sie erlaubt, die Zusammenhänge zweier Programmelemente im Hinblick auf die Bezeichner, Kommentare und Zeichenketten getrennt voneinander zu analysieren und hierbei auf die jeweiligen Besonderheiten einzugehen. Ein weiterer Vorteil der Abbildung auf verschiedene Programmelementdokumente ist, dass der Einfluss der verschiedenen lexikalischen Bestandteile im Quelltext bei der Analyse flexibel konfiguriert werden kann. Alternativ zur Dokumentabbildung (b) aus Abbildung 5.8 wäre eine gemeinsame Betrachtung von Kommentaren und Zeichenketten wie bei (c) denkbar, da sich die natürlichsprachlichen Bestandteile in Kommentaren und Zeichenketten sehr ähnlich sind.

Unabhängig davon wie sich die Textabbildung eines Programmelements zusammensetzt, ist es für das Auffinden semantischer Zusammenhänge notwendig, diese zu vergleichen. Durch die Worteinbettung können die vorverarbeiteten Textabbildungen als Vektormengen repräsentiert werden, indem jedes Wort auf einen Vektor abgebildet wird. Um den Zusammenhang zwischen zwei Texten bewerten zu können, ist es anschließend notwendig, die Vektormengen zu vergleichen. Hierfür können bspw. die geometrischen Schwerpunkte der Vektormengen berechnet werden. Der geometrische Schwerpunkt fasst die Wortbedeutungen eines Programmelementdokuments zusammen und ermöglicht es, die Bedeutung eines Programmelements durch einen einzelnen Vektor zu beschreiben. Je ähnlicher sich die geometrischen Schwerpunkte zweier Beschreibungen sind, desto wahrscheinlicher ist

es, dass ein semantischer Zusammenhang zwischen ihnen vorliegt. Infolgedessen eignet sich die Kosinus-Ähnlichkeit zwischen Schwerpunktvektoren zur Bewertung des semantischen Zusammenhangs. Ein Nachteil bei der Berechnung von Schwerpunktvektoren ist, dass vereinzelte Wörter, deren Vektorabbildungen sich stark von den restlichen Wortvektoren eines Textes unterscheiden, unter Umständen von diesen unterdrückt werden können und somit kaum einen Einfluss auf die Berechnung des Zusammenhangs haben. Um dies zu verhindern kann der Einfluss der Wörter eines Programmelementdokuments auf den geometrischen Schwerpunkt, durch entsprechende Gewichtsverteilungen, kontrolliert werden. Wörter, die eine höhere Aussagekraft im Bezug auf den Zweck einer Programmstelle besitzen, können hierdurch einen stärkeren Einfluss auf den geometrischen Schwerpunktvektor erhalten. Eine Möglichkeit, die Relevanz von Wörtern zu bewerten, bietet die Stelle, an der ein Wort im Quelltext zu finden ist. Wörter, die zur Signatur einer Methode oder Klasse gehören wie Signaturkommentare oder Methodenbezeichner beschreiben in der Regel den Zweck eines Programmelements als Ganzes und liefern eine zusammenfassende Beschreibung, wohingegen bspw. lokale Variablennamen lediglich im Kontext der jeweiligen Methode oder Klasse relevant sein können. Folglich kann es sinnvoll sein, Wörter aus den Signaturen höher zu gewichten. Allerdings sollte darauf geachtet werden, den Einfluss der Anweisungsebene nicht zu vernachlässigen. Eine mögliche initiale Gewichtsverteilung könnte den Wörtern aus den Signaturen ein Gewicht von 0,6 und den Wörtern aus der Anweisungsebene ein Gewicht von 0,4 zusprechen.
Neben der Stelle im Quelltext, kann die Häufigkeit eines Wortes ein Indiz für dessen Aussagekraft sein. Wie Beispiel 5.5 zeigt, besitzen Wörter, die häufig in Programmelementdokumenten vorkommen, vermutlich eine geringere Aussagekraft als seltener vorkommende Wörter, da sie kaum etwas zur Differenzierung von Beschreibungen beitragen. Demzufolge kann es hilfreich sein, die Anzahl der Programmelementdokumente, in denen ein Wort vorkommt, zur Bewertung dessen Relevanz heranzuziehen. Die IDF-Formel (Siehe Abschnitt 2.3.3) aus dem Bereich der Informationsrückgewinnung kann genutzt werden, um Wörter, die bezogen auf die Menge aller Programmelementdokumente selten auftreten, höher zu gewichten. Darüber hinaus kann die Häufigkeit eines Wortes innerhalb eines Programmelements Hinweise auf dessen Relevanz liefern. Bei der TF-Formel aus Abschnitt 2.3.3 gilt bspw., je häufiger ein Wort innerhalb eines Dokuments auftritt, desto wichtiger ist es für die Beschreibung des Dokuments. Übertragen auf die Analyse von natürlicher Sprache im Quelltext würde dies bedeuten, dass Wörter, die in der Textabbildung eines Programmelements häufig vorkommen, stärker gewichtet werden. Allerdings ist es fragwürdig, ob die Häufigkeit, mit der ein Wort innerhalb eines Programmelementdokuments vorkommt, bei der Bewertung der Relevanz in dieser Arbeit hilft. In der Dokumentabbildung einer Methode kommen bspw. Wörter aus den lokalen Variablenbezeichnern deutlich häufiger vor, als die Wörter aus dem Methodennamen, da Variablen potentiell mehrmals im Rumpf referenziert werden, wohingegen der Methodenname in der Regel nur einmal in der Signatur auftaucht. Dennoch sind die Wörter aus dem Methodennamen vermutlich wertvoller für die Beschreibung des Programmelements. Infolgedessen erscheint die Häufigkeit eines Wortes in einem Programmelementdokument als wenig hilfreich für die Gewichtung von Wörtern.

Beispiel 5.5: Relevanz häufiger Wörter

Gegeben seien zwei Programmelementdokumente $doc_1 = \{get, employee, provision\}$ und $doc_2 = \{get, employee, name\}$, die jeweils die natürlichsprachlichen Bestandteile einer Methode im Quelltext enthalten. Die Wörter *get* und *employee* kommen häufig im Quelltext vor, wohingegen die Wörter *provision* und *name* selten auftreten. Die von doc_1 beschriebene Methode berechnet die Provision eines Mitarbeiter und die von doc_2 beschriebene Methode liefert den Namen eines Mitarbeiters zurück. Obwohl in beiden Dokumenten die Wörter *get* und *employee* vorkommen, besitzen die von ihnen repräsentierten Programmelemente kaum einen semantischen Zusammenhang. Folglich sind die Wörter *provision* und *name* entscheidend bei der Bewertung des semantischen Zusammenhangs. Ein Zusammenhang aufgrund von häufig vorkommenden Wörtern ist, bezogen auf dieses Beispiel, demnach niedriger zu gewichten als ein Zusammenhang aufgrund von selteneren Wörtern.

5.3.4 Entwurf der lexikalischen Quelltextanalyse

Das Ziel der lexikalischen Quelltextanalyse ist das Finden semantischer Zusammenhänge zwischen Typen, Methoden, sowie zwischen Typen und Methoden auf Basis der natürlichen Sprache im Quelltext. Wie bereits in Abschnitt 5.3.3.1 dargestellt, bilden Worteinbettungen das Fundament der lexikalischen Analyse in dieser Arbeit. Aufgrund der in Abschnitt 5.3.3.2 aufgeführten Vorteile eines N-Gramm-basierten Modells, wird das in [BGJM17] vorgestellte Worteinbettungsmodell von Bojanowski et al. eingesetzt. Um externes Wissen über die Zusammenhänge und Kontexte von Wörtern einfließen zu lassen, wird das Modell auf einem externen Korpus vortrainiert. In Abschnitt 5.3.3.2 wurden verschiedene Möglichkeiten zur Bildung eines Trainingskorpus diskutiert. Da zum Zeitpunkt des Entwurfs nicht festgestellt werden kann, ob ein auf allgemeinen Texten oder ein auf Texten aus der Softwareentwicklungsdomäne trainiertes Modell zu besseren Ergebnissen führt, werden verschiedene Modelle bereitgestellt. Ein Modell wird auf allgemeinen Texten aus Wikipedia trainiert, das zweite auf Texten aus GitHub-Softwareprojekten und das dritte auf einer Kombination aus Texten von StackOverflow. Standardmäßig wird das auf Wikipedia trainierte Worteinbettungsmodell genutzt, da dieses die Verwendung der Wörter im allgemeinen Sprachgebrauch gelernt und vermutlich kontextunabhängig am besten funktioniert. Die Evaluation der verschiedenen Worteinbettungsmodelle erfolgt in Kapitel 7. Aufgrund des geringen Umfangs der natürlichen Sprache im Quelltext, im Vergleich zu der Größe von externen Trainingskorpora, hat ein Nachtrainieren des Worteinbettungsmodells kaum Auswirkungen auf die gelernten Wortvektoren, weshalb auf ein erweitertes Training verzichtet wird.

5.3.4.1 Vorverarbeitung

Aus den in Abschnitt 5.3.3.4 diskutierten Möglichkeiten ergeben sich die unten aufgeführten Vorverarbeitungsschritte für die lexikalischen Bestandteile im Quelltext. Auf eine Normalisierung der Wortformen wird hierbei verzichtet, da durch ein Training der Worteinbettungsmodelle auf nicht normalisierten Korpora auch die Form eines Wortes zur Beschreibung ihrer Bedeutung beiträgt.

- **Auftrennen zusammengesetzter Terme:** Zusammengesetzte Wörter werden entsprechend gängiger Namenskonventionen an den Großbuchstaben oder Unterstrichen aufgetrennt. Darüber hinaus werden die entstandenen Wörter markiert, sodass die Informationen, welche Wörter als zusammengesetzte Begriffe im Quelltext aufgetreten sind, beibehalten werden.

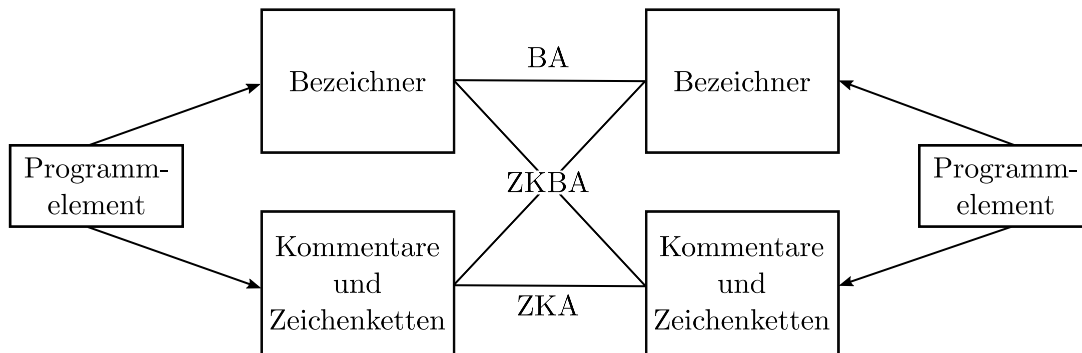


Abbildung 5.9: Abstraktes Vorgehen der lexikalischen Analyse.

- **Abbildung auf Kleinbuchstaben:** Um sicherzustellen, dass die Groß- und Kleinschreibung der Begriffe im Quelltext keine Auswirkungen auf die Bedeutung hat, werden alle Buchstaben auf Kleinbuchstaben abgebildet.
- **Stoppwortentfernung:** In diesem Schritt werden Stoppwörter in der natürlichen Sprache und Schlüsselwörter der genutzten Programmiersprache entfernt. Darüber hinaus wird das Wort *var* sowie Zahlen und Wörter, die lediglich aus einem Buchstaben bestehen, entfernt. Die Menge der zu entfernenden Stoppwörter wird durch eine Liste definiert und ist leicht anpassbar.
- **Abkürzungsauflösung:** Für die Auflösung von Abkürzungen wird ein Abkürzungswörterbuch verwendet, um Verfälschungen durch falsch aufgelöste Abkürzungen zu vermeiden. Alle Terme, die nicht im Vokabular des Worteinbettungsmodells enthalten sind, werden als potentielle Abkürzungen aufgefasst. Für jede Abkürzung wird geprüft, ob das Wörterbuch dessen ausgeschriebene Form enthält. Ist das der Fall, wird geprüft, ob die ausgeschriebene Form der Abkürzung im Vokabular des Worteinbettungsmodells enthalten ist und falls ja, wird die Abkürzung durch ihre ausgeschriebene Form ersetzt. Die letzte Überprüfung wird vollzogen, um sicherzustellen, dass die Abkürzungsauflösung zu einer Verringerung der unbekannten Wörter führt.

Für die Vorverarbeitung von Kommentaren erfolgen weitere Vorverarbeitungsschritte. Kommentare, die Lizenzierungsbeschreibungen oder Quelltext darstellen, werden, in Anlehnung an die Ausführungen aus Abschnitt 5.3.4.1, entfernt. Kommentare, die mit *todo* oder *fixme* markiert sind und keinen Bezeichner in ihrer Textmenge enthalten, werden ebenfalls entfernt, da angenommen werden kann, dass es sich um allgemeine Kommentare ohne spezifischen Bezug zu einem Programmelement handelt. Darüber hinaus werden die Wörter *todo* und *fixme* aus den Kommentaren entfernt. Auch Markierungen wie `@author`, `@version`, `@since` und `@deprecated`, die keine relevanten Informationen für die Suche nach semantischen Zusammenhängen enthalten, werden entfernt. Bei allen weiteren Markierungen wird der Text innerhalb der Markierung beibehalten. In einem weiteren Schritt werden Internetadressen entfernt und Formatierungselemente verarbeitet, sodass lediglich der Text innerhalb dieser erhalten bleibt. Um die nachfolgenden Verarbeitungsschritte zu vereinfachen, werden die verbleibenden Texte durch eine Trennung an den Leerstellen und Satzzeichen auf Wortmengen abgebildet. Anschließend durchlaufen die Wortmengen die zuvor aufgezählten Vorverarbeitungsschritte.

5.3.4.2 Semantische Zusammenhänge

Abbildung 5.9 zeigt den Ablauf der lexikalischen Analyse exemplarisch für zwei Programmelemente. Im ersten Schritt wird ein Programmelement auf zwei Dokumente abgebildet. Eines umfasst die Bezeichner und eines die Kommentare und Zeichenketten. Die Dokumente durchlaufen anschließend die zuvor beschriebenen Vorverarbeitungsschritte. Bei der

Erzeugung der Dokumentabbildungen kann es vorkommen, dass ein Dokument keine Wörter enthält, wenn z.B. das Programmelement keine Kommentare und Zeichenketten besitzt. Beim Vergleich eines leeren Dokuments ist das Ergebnis stets 0. Die Bewertung des semantischen Zusammenhangs zweier Programmelemente erfolgt in Folge der Ausführungen in Abschnitt 5.3.3.5 durch einen Vergleich ihrer Dokumentabbildungen. Diese Vorgehensweise bietet die größtmögliche Flexibilität und erlaubt eine individuelle Gewichtung der lexikalischen Informationen im Quelltext. Die erste Analyse, in der Abbildung mit *BA* gekennzeichnet, vergleicht die Bezeichner zweier Programmelemente. Die zweite Analyse betrachtet die Zeichenketten und Kommentare der Programmelemente und wird im Folgenden *ZKA* genannt. In der dritten Analyse werden Zusammenhänge zwischen Kommentaren bzw. Zeichenketten und Bezeichnern gesucht. In Abbildung 5.9 ist diese Analyse mit *ZKBA* gekennzeichnet. Alle drei Analyse basieren in Anlehnung an Abschnitt 5.3.3.3 und Abschnitt 5.3.3.5 auf der Berechnung von Schwerpunktvektoren in Kombination mit einer gesonderten Behandlung von unbekannten Wörtern durch den Einsatz einer Zeichenkettenanalyse.

Schwerpunktanalyse

Bei der Berechnung des Schwerpunktvektors eines Programmelementdokuments e wird in einem ersten Schritt zwischen Wörtern aus der Signatur W_{sig} und Wörtern aus den Anweisungen W_{st} unterschieden. Für beide Wortmengen wird ein Schwerpunktvektor $\vec{c}_{e,sig}$ und $\vec{c}_{e,st}$ berechnet. Hierfür werden die Wortmengen W_{sig} bzw. W_{st} mit Hilfe des Worteinbettungsmodells auf Vektormengen V_{sig} bzw. V_{st} abgebildet. Aus Abschnitt 5.3.3.4 geht hervor, dass es sinnvoll ist, die Vektorabbildungen der Teilwörter eines zusammengesetzten Terms vor der Weiterverarbeitung zu einem Vektor zusammenzufassen. Folglich werden alle Wortgruppen, die einen zusammengesetzten Begriff im Quelltext repräsentieren, durch einen über Gleichung 5.14 berechnet Schwerpunktvektor repräsentiert. Dieser Schwerpunktvektor ersetzt die Vektoren der Teilwörter in V_{sig} bzw. V_{st}.

$$centroid(V) = \frac{1}{|V|} \sum_{\vec{v} \in V} \vec{v} \tag{5.14}$$

Bei der Berechnung der Schwerpunktvektoren von V_{sig} bzw. V_{st} fließt die Relevanz eines Begriff als Gewicht des Vektors mit ein. Hierzu wird entsprechend der Erläuterungen in Abschnitt 5.3.3.5 die Anzahl der Programmelementdokumente, in denen ein Begriff vorkommt, über Gleichung 5.15 bewertet. Die Formel ist eine Adaption des IDF-Maß aus Abschnitt 2.3.3. Die Menge D umfasst alle Programmelementdokumente im Quelltext.

$$weight(t) = log \frac{|D|}{|\{d : t \in d \land d \in D\}|} \tag{5.15}$$

Je seltener ein Term Teil eines Programmelementdokuments ist, desto höher wird er gewichtet. Ein Begriff, der in allen Programmelementen vorhanden ist, wird mit einem Gewicht von 0 bewertet.
Mit Hilfe von Gleichung 5.15 wird für V_{sig} bzw. V_{st} jeweils ein gewichteter Schwerpunktvektor über Gleichung 5.16 berechnet. Die Abbildung $term(vec)$ liefert die natürlichsprachliche Repräsentation eines Vektors.

$$centroid(V) = \sum_{\vec{v} \in V} weight(term(\vec{v})) * \vec{v} \tag{5.16}$$

Um den zusammengesetzten Schwerpunktvektor $\vec{c}_e$ eines Programmelementdokuments e zu erhalten werden beide Schwerpunktvektoren $\vec{c}_{e,sig}$ und $\vec{c}_{e,st}$ gewichtet addiert. Als initiale Gewichtung dienen die Werte aus Abschnitt 5.3.3.5. Der Schwerpunktvektor der

Signatur wird mit einem Gewicht von 0,6 und der Schwerpunktvektor der Anweisungen mit einem Gewicht von 0,4 multipliziert. Beide Gewichte sind flexibel anpassbar, sodass zukünftige Veränderungen kein Problem darstellen. Die Optimierung der Gewichte wird in Abschnitt 6.5 gesondert betrachtet.
Nachdem jedem Programmelement ein Schwerpunktvektor zugeordnet wurde, kann der Zusammenhang zweier Programmelemente e_1 und e_2 durch die Berechnung des Kosinus des Winkels zwischen den Schwerpunktvektoren $\vec{c}_{e_1}$ und $\vec{c}_{e_2}$ bewertet werden. Der Wertebereich der Kosinus-Ähnlichkeit liegt zwischen -1 und 1, wobei 1 eine perfekte Übereinstimmung bedeutet. Um die Vergleichbarkeit mit den Ergebnissen der Zeichenkettenanalyse sicherzustellen wird das Ergebnis der Kosinus-Ähnlichkeit auf den Bereich $[0, 1]$ normiert.
Im Gegensatz zur *BA*- und *ZKA*-Analyse liegen bei der *ZKBA*-Analyse zwei Kosinus-Ähnlichkeiten vor, da sowohl die Ähnlichkeit zwischen dem Schwerpunktvektor der Bezeichner von e_1 und dem Schwerpunktvektor der Kommentare und Zeichenketten von e_2 als auch umgekehrt berechnet wird. Für die Berechnung des Ergebnisses der *ZKBA*-Analyse ist daher ein weiterer Berechnungsschritt notwendig, um die beiden Ähnlichkeiten zu einem Gesamtergebnis zusammenzufassen. Hierbei wird der Mittelwert aus den beiden Ergebnissen berechnet.

Zeichenkettenanalyse

Aufgrund der Verwendung eines statischen Wörterbuchs, können bspw. projektspezifische Abkürzungen nicht aufgelöst werden. Aus Abschnitt 5.3.3.3 geht hervor, dass es sinnvoll ist, eine zusätzliche Analyse auf Basis von Zeichenkettenvergleichen für die Berücksichtigung von unbekannten Abkürzungen bzw. unbekannten Wörtern in die lexikalische Analyse zu integrieren. Hierbei werden Übereinstimmungen zwischen den unbekannten Wörtern eines Programmelementdokuments mit den Wörtern eines anderen Programmelementdokuments gesucht. Als unbekannte Abkürzung gelten alle Wörter, die im vorverarbeiteten Programmelementdokument existieren und nicht im Vokabular des Worteinbettungsmodells liegen. Als Basis der Analyse dient eine Kombination aus der Jaro-Winkler-Ähnlichkeit und dem FuzzyScore. Im Gegensatz zur Jaro-Winkler-Ähnlichkeit ist das Ergebnis des FuzzyScore nicht normiert. Um die Vergleichbarkeit sicherzustellen, wird der FuzzyScore auf einen Bereich zwischen 0 und 1 normiert. Hierzu werden die Ergebnisse des FuzzyScore durch die maximal erreichbare Ähnlichkeit geteilt. Der maximale FuzzyScore für zwei Zeichenketten s_1 und s_2 entspricht $(3 * max(|s_1|, |s_2|)) - 2$.
Für beide Kennzahlen wird ein Schwellwert definiert, der festlegt, ab welcher Übereinstimmung zwei Zeichenketten als dasselbe Wort betrachtet werden. Da die Jaro-Winkler-Ähnlichkeit dazu neigt, die Ähnlichkeit von Zeichenketten überzubewerten, wird ein hoher Schwellwert von $\alpha_{jw} = 0,75$ gewählt. So ist sichergestellt, dass Zeichenketten wie *employee* und *edv*, die eine Jaro-Winkler-Ähnlichkeit von 0.49 besitzen, nicht als identisch betrachtet werden. Im Gegensatz dazu bewertet der FuzzyScore konservativer, weshalb der Schwellwert mit $\alpha_{fs} = 0,4$ niedriger angesetzt ist. Beide Schwellwerte sind einfach konfigurierbar, um spätere Anpassungen problemlos durchführen zu können. Gleichung 5.17 zeigt die Formel, die entscheidet, ob zwei Zeichenketten s_1 und s_2 übereinstimmen oder nicht. Mit jw ist die Jaro-Winkler-Ähnlichkeit und mit fs der FuzzyScore gekennzeichnet.

$$compare(s_1, s_2) = \begin{cases} 1, & \text{if jw}(s_1, s_2) > \alpha_{jw} \vee \text{ fs}(s_1, s_2) > \alpha_{fs}. \\ 0, & \text{otherwise.} \end{cases} \tag{5.17}$$

Beim Vergleich der Dokumentabbildungen d_1 und d_2 zweier Programmelemente wird für jedes unbekannte Wort in d_1 geprüft, ob es eine Übereinstimmung mit einem Wort aus d_2 gibt. Die Summe der Übereinstimmungen wird anschließend durch die Anzahl der Abkürzungen geteilt, sodass der Zusammenhang mit einer Zahl zwischen 0 und 1 beschrieben werden kann. Gleichung 5.18 zeigt die Formel hierfür. Mit *unknown* wird die Abbildung

beschrieben, welche die Menge der unbekannten Wörter aus einem Programmelementdokument zurückliefert.

$$match(d_1, d_2) = \frac{1}{|unknown(d_1)|} \sum_{u \in unknown(d_1)} max_{w \in d_2}(compare(u, w)) \tag{5.18}$$

Im Allgemeinen ist $match(d_1, d_2) \neq match(d_2, d_1)$. Für eine abschließende Bewertung ist es daher notwendig, beide berechneten Zusammenhänge zu kombinieren. Hierfür eignet sich der Mittelwert der berechneten Übereinstimmungen. Die finale Formel zur Bewertung der Zeichenkettenähnlichkeit zeigt Gleichung 5.19.

$$sim_{string}(d_1, d_2) = \frac{1}{2} * (match(d_1, d_2) + match(d_2, d_1)) \tag{5.19}$$

Jedes Programmelementpaar durchläuft die in Abbildung 5.9 dargestellten Analysen (*BA*, *ZKA* und *ZKBA*), wobei jede Analyse jeweils einen Schwerpunktvergleich und einen Zeichenkettenvergleich durchführt. Aus Abschnitt 5.3.3.3 geht hervor, dass das Ergebnis des Zeichenkettenanalyse eine hohe Unsicherheit besitzt. Folglich wird das Ergebnis der Schwerpunktanalyse mit $w_{cent} = 0,9$ höher gewichtet. Das Ergebnis des Zeichenkettenvergleichs fließt lediglich mit einem Gewicht von $w_{string} = 0,1$ in das Ergebnis mit ein. Eine Optimierung der initialen Gewichtung ist Bestandteil von Abschnitt 6.5. Ein Sonderfall der lexikalischen Analyse tritt ein, wenn Programmelementdokumente keine unbekannten Wörter oder Abkürzungen enthält. In solchen Fällen wird kein Zeichenkettenvergleich durchgeführt und das Ergebnis der Schwerpunktanalyse wird mit einem Gewicht von $w_{cent} = 1,0$ berücksichtigt.

Das Endergebnis der lexikalischen Analyse setzt sich aus den Ergebnissen der drei Vergleiche zwischen den Dokumentabbildungen zweier Programmelemente zusammen. Die berechneten lexikalischen Zusammenhänge werden mit individuellen Gewichten multipliziert und anschließend aufsummiert. Die Zusammenhänge der Kommentare bzw. Zeichenketten w_{ZKA}, der Bezeichner w_{BA} und der Kommentare bzw. Zeichenketten und Bezeichner w_{ZKBA} werden initial mit jeweils 0,333 gewichtet. Eine Anpassung der Gewichtsbelegung, um die Ergebnisse der lexikalischen Analyse zu optimieren, ist Gegenstand von Abschnitt 6.5.

Analog zum Vorgehen bei der syntaxbasierten Quelltextanalyse ist es sinnvoll, einen Schwellwert α_{la} bei der lexikalischen Quelltextanalyse zu definieren, der das Mindestmaß an Sicherheit festlegt, den ein gefundener semantischer Zusammenhang besitzen muss, um für den Aufbau der semantischen Quelltextrepräsentation relevant zu sein. Initial wird $\alpha_{la} = 0$ gesetzt, um alle gefundenen Zusammenhänge zu berücksichtigen. Die Optimierung des Schwellwerts ist ebenfalls Gegenstand von Abschnitt 6.5.

5.4 Aufbau der semantischen Quelltextrepräsentation

Durch die Analyse der strukturellen Abhängigkeiten und der lexikalischen Bestandteile im Quelltext liegen Informationen über Zusammenhänge zwischen Programmelementen vor. Im Folgenden soll untersucht werden, wie diese Informationen ausgenutzt werden können, um die semantische Quelltextrepräsentation aus Abschnitt 5.2 aufzubauen.

5.4.1 Semantische Zusammenhänge

Aus dem Entwurf der semantischen Repräsentation in Abschnitt 5.2.2 geht hervor, dass semantische Zusammenhänge durch ungerichtete Kanten mit einem Gewicht zwischen 0 und 1 beschrieben werden. Als Ausgangspunkt für die Erzeugung solcher Kanten können die gefundenen Zusammenhänge der Quelltextanalysen aus Abschnitt 5.3 genutzt werden.

Wenn keine der Quelltextanalysen einen Zusammenhang zwischen zwei Programmelementen erkennen konnte, kann angenommen werden, dass sie keinen semantischen Zusammenhang besitzen. Umgekehrt kann es vorkommen, dass mehrere Quelltextanalysen einen Zusammenhang zwischen zwei Programmelementen erkennen. Grundsätzlich sind hierbei alle Ergebnisse der Quelltextanalysen wichtig, da sich ein existierender semantischer Zusammenhang auf verschiedenen Ebenen im Quelltext zeigen kann. Bewerten die verschiedenen Quelltextanalysen den Zusammenhang zwischen zwei Programmstellen stark unterschiedlich, kann das als Hinweis auf die Unsicherheit des semantischen Zusammenhangs aufgefasst werden. Folglich sollten sowohl hohe als auch niedrige Zusammenhangswerte bei der Berechnung der semantischen Zusammenhänge berücksichtigt werden. Eine Bewertungsstrategie, die alle gefundenen Zusammenhänge berücksichtigt, ist die Berechnung des arithmetischen Mittelwerts. Hierbei können die erkannten Zusammenhänge zwischen zwei Programmelementen aufsummiert und anschließend durch die Anzahl der Quelltextanalysen geteilt werden. Ein Vorteil dieses Vorgehens ist, dass die Berechnung leicht um weitere Quelltextanalysen ergänzt werden kann. In Anlehnung an Z3 aus Abschnitt 5.1 kann folglich eine flexible Anzahl von Quelltextanalysen beim Aufbau der semantischen Quelltextrepräsentation berücksichtigt werden. Ein potentieller Nachteil bei der Berechnung des arithmetischen Mittelwerts ist, dass die Ergebnisse aller Quelltextanalysen gleich stark gewichtet werden. Es kann jedoch sinnvoll sein, den Quelltextanalysen eine unterschiedlich starke Aussagekraft zuzusprechen. In [AC17] führte im Kontext der Remodularisierung bspw. ein Gewicht von $0,8$ für die lexikalische Analyse und ein Gewicht von $0,2$ für die strukturelle Analyse zu den besten Ergebnissen. Auch De Lucia et al. kombinieren in ihrer Arbeit [LOV08] die Ergebnisse einer strukturellen und lexikalischen Analyse, um die Zusammenhänge innerhalb von Klassen zu beschreiben und deren Kohäsion zu verbessern. Sie erzielten die besten Ergebnisse mit einer Gleichgewichtung beider Analysen. Ob eine der Gewichtsverteilungen auf die Problemstellung dieser Arbeit übertragbar ist, bleibt offen. Insgesamt erscheint es jedoch sinnvoll, die semantischen Zusammenhänge über ein gewichtetes Mittel der gefundenen Zusammenhänge aus den einzelnen Analysen zu berechnen und dabei eine individuelle Gewichtung zu erlauben.

Durch die Kombination von den Ergebnissen der verschiedenen Quelltextanalysen können sehr kleine semantische Zusammenhänge entstehen, wenn bspw. lediglich eine der Quelltextanalysen einen Zusammenhang zwischen zwei Programmelementen erkennt. Analog zur syntaxbasierten und lexikalischen Quelltextanalyse erscheint es daher sinnvoll, einen Schwellwert für die Berücksichtigung der berechneten semantischen Zusammenhänge beim Aufbau der semantischen Quelltextrepräsentation zu definieren. Dieser lässt das Mindestmaß an Sicherheit, mit der die semantische Repräsentation arbeitet, einfach steuern.

5.4.2 Absichtsknoten

Die Quelltextrepräsentation aus Abschnitt 5.2.2 umfasst neben semantischen Zusammenhängen sogenannte Absichtsknoten. Diese repräsentieren Gruppen von Programmelementen, die im Hinblick auf eine bestimmte Absicht zusammenhängen. Ob Programmelemente eine gemeinsame Absicht besitzen, kann über ihren semantischen Zusammenhang bewertet werden. Ein hoher semantischer Zusammenhang lässt sich bspw. als Hinweis auf eine gemeinsame Absicht auffassen. Um die Programmelemente im Quelltext hinsichtlich ihrer Absichten zu unterteilen, kann eine Clusteranalyse auf Basis der semantischen Zusammenhänge durchgeführt werden. Hierbei gilt es auf die Besonderheiten von Quelltext zu achten. Zum einen kann vorab nicht bestimmt werden, wie viele verschiedene Absichten und damit Cluster im Quelltext vorhanden sind. Zum anderen sollte die Clusteranalyse in der Lage sein, hierarchische Strukturen zu beschreiben und die Zuordnung eines Programmelements zu mehreren Clustern zu erlauben. Angenommen, zwei Methoden implementieren die Anforderung *Vertrag anzeigen*. Für die Clusteranalyse der Programmelemente bedeutet dies,

dass die beiden Methoden ein Cluster bilden. Gleichzeitig werden beide Methoden bei der Implementierung der Funktionalität *Vertrag bearbeiten* genutzt. Folglich sollte ein zweites Cluster für die Funktionalität *Vertrag bearbeiten* existieren, welches das vorherige Cluster und gegebenenfalls weitere zusätzlichen Programmelemente enthält. Eine hierarchische Clusteranalyse (Siehe Abschnitt 2.4) wäre in der Lage, die zuvor beschriebene Situation adäquat darzustellen. Durch die hierarchische Struktur der gefundenen Cluster, ist eine Betrachtung der Zusammenhänge im Softwaresystem auf verschiedenen Granularitätsebenen möglich. Größere Cluster bauen auf kleineren auf genau wie sich die Funktionalitäten eines Softwareprojekts aus spezifischen Quelltextstellen zusammensetzen. Grundsätzlich kann zwischen der agglomerativen und der divisiven hierarchischen Clusteranalyse unterschieden werden. Bei der agglomerativen Clusteranalyse wird zunächst jedes Programmelement als eigenständiges Cluster aufgefasst werden. Bei der divisiven Clusteranalyse läge zu Beginn ein Cluster, welches alle Programmelemente beinhaltet, vor. Da die Programmelemente eines Softwareprojekts in der Regel nicht mit allen anderen Programmelementen zusammenhängen, erscheint eine agglomerative Clusteranalyse für die Zwecke der vorliegenden Arbeit sinnvoll.

Beim Einsatz einer agglomerativen hierarchischen Clusteranalyse stellt sich die Frage nach der Kennzahl, die entscheidet, welche zwei Cluster in einer Iteration vereint werden. Eine Möglichkeit den Zusammenhang zwischen Clustern zu bewerten, ist der maximale semantische Zusammenhang zwischen ihren enthaltenen Programmelementen. Allerdings ist dieses Vorgehen anfällig für Rauschen. Es kann bspw. vorkommen, dass zwei Cluster vereint werden, obwohl die Programmelemente der Cluster generell einen niedrigen semantischen Zusammenhang zueinander aufweisen und lediglich zwei Programmelemente einen starken semantischen Zusammenhang besitzen. Alternativ kann der minimale semantische Zusammenhang zwischen zwei Programmelementen in verschiedenen Clustern zur Bewertung herangezogen werden. In jeder Iteration würden daraufhin die beiden Cluster vereint werden, deren minimaler semantischer Zusammenhang am größten ist. Gegenüber der vorangegangenen Methode ist dieses Vorgehen weniger anfällig für Rauschen. Da größere Cluster jedoch mit höherer Wahrscheinlichkeit ein Programmelement mit einem niedrigen semantischen Zusammenhang zu einem Programmelement eines anderen Clusters besitzen, können diese benachteiligt werden. Insgesamt ist eine Bewertung der Ähnlichkeiten zwischen Clustern, auf der Basis einzelner semantischer Zusammenhänge, daher eher ungeeignet.
Eine Möglichkeit mehrere semantische Zusammenhänge in die Ähnlichkeitsbewertung von Clustern zu integrieren, bietet die Berechnung des Mittelwerts der semantischen Zusammenhänge zwischen den Programmelementen zweier Cluster. Ein Vorteil hiervon ist, dass alle bestehenden semantischen Zusammenhänge zwischen den Programmelementen in den beiden Clustern berücksichtigt werden und die Bewertung folglich alle verfügbaren Informationen berücksichtigt.
Eine Schwierigkeit bei der agglomerativen Clusteranalyse ergibt sich, wenn mehrere Cluster einen gleich starken Zusammenhang aufweisen. Da keine Informationen über das korrekte Ergebnis der Clusteranalyse vorliegen, können lediglich Heuristiken für die Entscheidungsfindung eingesetzt werden. Der durchschnittliche Zusammenhang zu Programmelementen außerhalb des eigenen Clusters kann bspw. als Indiz für die Güte eines Clusters aufgefasst werden. Das Cluster mit dem niedrigsten durchschnittlichen Zusammenhang zu Programmelementen anderer Cluster könnte daraufhin bevorzugt werden. Allerdings ist nicht gewährleistet, dass die Einbeziehung des durchschnittlichen äußeren Zusammenhangs zu einem eindeutigen Ergebnis führt. Folglich kann es vorkommen, dass weiterhin mehrere Clusterkandidaten für eine Vereinigung vorliegen. Als weitere Möglichkeit kann eine randomisierte Wahl des zu vereinenden Clusterpaars erfolgen. Durch die Hinzunahme von Randomisierung wäre die Clusteranalyse jedoch nicht mehr deterministisch, wodurch die Reproduzierbarkeit der Ergebnisse gefährdet wird. Um den Determinismus der Cluster-

analyse zu bewahren ist es daher sinnvoll, auf Randomisierung zu verzichten und in diesen Fällen bspw. immer das erste Clusterpaar, aus einer Liste an gleich bewerteten Clusterpaaren, zu wählen.

Im Hinblick auf die Konstruktion von Absichtsknoten, sind nicht alle Cluster der Clusterhierarchie hilfreich. Da ein Absichtsknoten eine Menge aus zusammenhängenden Programmelementen beschreiben soll, sind Cluster, welche lediglich ein Programmelement umfassen, uninteressant.
Mit Blick auf den späteren Einsatz der Absichtsknoten bei der Suche nach Rückverfolgbarkeitsinformationen, erscheint es darüber hinaus sinnvoll, ein Abbruchkriterium für die Bildung neuer Cluster zu definieren. Hierdurch kann vermieden werden, dass Cluster entstehen, deren Programmelemente nur schwach zusammenhängen und vermutlich keiner Anforderung zugeordnet werden können. Als mögliches Abbruchkriterium kann der durchschnittliche semantische Zusammenhang der Programmelemente eines Clusters herangezogen werden. Liegt der durchschnittliche Zusammenhang eines neu erzeugten Clusters unter einem bestimmten Schwellwert, kann die Clusteranalyse abgebrochen werden. Allerdings stellt die Wahl eines adäquaten Schwellwerts eine Herausforderungen dar, da sowohl ein zu hoher als auch ein zu niedriger Wert die Clusteranalyse negativ beeinflussen könnte. Initial erscheint ein mittlerer Wert wie 0,6 sinnvoll. Er ist vermutlich klein genug, um die Bildung von Clustern mit mehr als zwei oder drei Programmelementen zu erlauben. Gleichzeitig verhindert er die Bildung von Clustern, die nur einen sehr schwachen Zusammenhang besitzen.
Cluster, die aufgrund des Abbruchkriteriums mit keinem anderen Cluster vereint werden konnten, beschreiben eine Unterteilung der Programmelemente in disjunkte Cluster. Bei einer adäquaten Wahl des Abbruchkriteriums kann davon ausgegangen werden, dass diese Cluster unterschiedliche Absichten repräsentieren. Folglich erscheint es sinnvoll, alle diese Cluster als Absichtsknoten in die semantische Repräsentation zu importieren. Ein Nachteil der Beschränkung auf die Cluster der höchsten Hierarchieebene ist, dass ein Programmelement maximal einem Cluster zugeordnet wird. Eine Möglichkeit die Zuordnung von Programmelementen zu verschiedenen Clustern zu erlauben bietet die Untersuchung der strukturellen Zusammensetzung der Cluster. Hierbei kann nach Clustern entlang der Hierarchie gesucht werden, die zwar in einem Cluster auf einer höheren Hierarchieebene enthalten sind, allerdings auch eine eigene differenzierte Absicht repräsentieren. Ein möglicher Hinweis auf eine unterschiedliche Absicht könnte eine starke Veränderungen des durchschnittlichen inneren Zusammenhangs liefern. Ein Cluster, dessen innerer durchschnittlicher Zusammenhang im Vergleich zu den durchschnittlichen inneren Zusammenhängen der Kindcluster deutlich geringer ist, kann ein Indiz dafür sein, dass sich der Grund für die Platzierung der Programmelemente im Cluster geändert hat. Das Beispiel 5.6 veranschaulicht diese Überlegung.

Beispiel 5.6: Cluster mit unterschiedlichen Absichten

Gegeben seien drei Cluster A, B und C. Cluster A enthält Programmelemente, die der Funktionalität *Vertrag anlegen* dienen. Cluster B enthält Programmelemente für die Funktionalität *Vertrag löschen*. Beide Cluster besitzen einen hohen durchschnittlichen inneren Zusammenhang von 0,8.
Cluster C stellt die Vereinigung der Cluster A und B dar und besitzt einen durchschnittlichen inneren Zusammenhang von 0,6. Cluster C kann der Funktionalität *Vertrag verwalten* zugeordnet werden. Der niedrigere innere Zusammenhang von Cluster C im Vergleich zu Cluster A und B ist ein Hinweis darauf, dass sich die Absicht, welche den Programmelementen im Cluster zugrunde liegt, geändert hat und nun allgemeiner ist.

Die zuvor beschriebene Vorgehensweise ist jedoch mit einer hohen Unsicherheit verbunden, da eine Verringerung des inneren Zusammenhangs eines Clusters viele Gründe besitzen kann und nicht notwendigerweise auf eine Änderung der Absicht hindeutet. Beispielsweise kann eine starke Verringerung des inneren Zusammenhangs nach der Vereinigung zweier Cluster auch dafür sprechen, dass die Cluster fälschlicherweise vereint wurden. Eine feingranulare Differenzierung der Cluster, hinsichtlich ihrer repräsentierten Absicht, erfordert daher weitere Analysen, die über den Rahmen dieser Arbeit hinausgehen.

Aus Abschnitt 5.2.2 geht hervor, dass die Absichtsknoten im Graph Kanten zu allen Programmelementen des entsprechenden Clusters besitzen. Diese Kanten besitzen ein Gewicht zwischen 0 und 1, welches die Stärke des Zusammenhangs zwischen dem Programmelement und dem Absichtsknoten quantifiziert. Um die Zugehörigkeit eines Programmelements zu einem Cluster zu bewerten, kann bspw. der Mittelwert der semantischen Zusammenhänge zu allen anderen Programmelementen im Cluster herangezogen werden. Je stärker ein Programmelement mit den anderen Programmelementen verbunden ist, desto stärker ist die Zugehörigkeit zum Cluster. Programmelemente, die weniger stark mit den übrigen Programmelementen eines Clusters zusammenhängen, besitzen eine schwächere Clusterzugehörigkeit. Infolgedessen eignet sich die Bewertung der Zugehörigkeit über den Mittelwert der semantischen Zusammenhänge als Kantengewicht zwischen Programmelementen und Absichtsknoten.

5.4.3 Entwurf zum Aufbau der semantischen Quelltextrepräsentation

Der Aufbau der semantischen Quelltextrepräsentation gliedert sich in zwei Schritte. Im ersten Schritt werden die Ergebnisse der Quelltextanalysen kombiniert, um semantische Zusammenhänge zu erzeugen. Für jedes Programmelementpaar werden, in Übereinstimmung mit den Ausführungen aus Abschnitt 5.4.1, die gefundenen Zusammenhänge der verschiedenen Analysen genutzt, um einen gewichteten Mittelwert zu berechnen. Gleichung 5.20 zeigt die Formel zur Berechnung des semantischen Zusammenhangs zweier Programmelemente e_1 und e_2. Mit A ist die Menge der Quelltextanalysen beschrieben und w_a ist das Gewicht der jeweiligen Analyse. Initial wird die lexikalische Analyse in Anlehnung an die Gewichtung in [AC17, LOV08] mit $0,6$ etwas stärker als die strukturelle Quelltextanalyse gewichtet. Die Optimierung der Gewichte wird in Abschnitt 6.5 explizit betrachtet.

$$semRel(e_1, e_2) = \sum_{a \in A} w_a * a(e_1, e_2) \tag{5.20}$$

Eine spätere Erweiterung um weitere Quelltextanalysen wird durch eine einfache Konfigurierbarkeit der Gewichte gewährleistet. So ist es für die Hinzunahme einer zusätzlichen Quelltextanalyse lediglich notwendig, die Menge A zu erweitern und die Gewichte anzupassen.
Semantische Zusammenhänge kleiner als der Schwellwert $\alpha_{semantic}$ werden, wie in Abschnitt 5.4.1 beschrieben, nicht weiter berücksichtigt. Zu Beginn wird $\alpha_{semantic} = 0$ gesetzt, sodass alle semantischen Zusammenhänge beachtet werden. Der Schwellwert ist Teil der Parameteroptimierung in Abschnitt 6.5.

Nachdem die semantischen Zusammenhänge berechnet wurden, wird eine Clusteranalyse auf den Programmelementen durchgeführt. Aus Abschnitt 5.4.2 geht hervor, dass sich eine agglomerative hierarchische Clusteranalyse hierfür eignet. Für den Vergleich zwischen Programmelementen dienen die zuvor berechneten semantischen Zusammenhänge. Die Bewertung des Zusammenhangs zweier Cluster c_1 und c_2 erfolgt durch die Berechnung des durchschnittlichen semantischen Zusammenhangs zwischen den Programmelementen der beiden Cluster. Gleichung 5.21 zeigt die Formel hierzu. Sie entspricht der Group-Average-Formel aus [MRS08]. In jeder Iteration werden die beiden Cluster miteinander vereint,

welche den größten, über Gleichung 5.21 ermittelten, Wert besitzen.

$$sim(c_1, c_2) = \frac{1}{(|c_1| + |c_2|) * (|c_1| + |c_2| - 1)} \sum_{e_1 \in c_1 \cup c_2} \sum_{e_2 \in c_1 \cup c_2, e_1 \neq e_2} semRel(e_1, e_2) \quad (5.21)$$

Bei Mehrdeutigkeiten wird für jedes potentielle neue Cluster der durchschnittliche Zusammenhang der Programmelemente im Cluster zu den Programmelementen außerhalb des Clusters berechnet. Das Cluster mit dem geringsten durchschnittlichen Zusammenhang zu Programmelementen außerhalb des Clusters wird als neues Cluster ausgewählt. Sollte bei dieser Berechnung ebenfalls eine Mehrdeutigkeit vorliegen, wird zur Sicherstellung des Determinismus der Clusteranalyse das erste Cluster aus der Kandidatenliste als neues Cluster des Iterationsschritts ausgewählt.

Als Folge der Beschreibungen in Abschnitt 5.4.2 wird ein Abbruchkriterium der Clusteranalyse definiert. Dieses soll verhindern, dass zu große bzw. allgemeine Cluster entstehen, die keine semantische Aussagekraft besitzen. Als Kriterium dient der über Gleichung 5.21 berechnete durchschnittliche semantische Zusammenhang der Programmelemente zweier Cluster. Als initiales Abbruchkriterium wird $\alpha_{cluster} = 0,6$ gewählt. Liegt der maximale Zusammenhang zweier Cluster in einem Iterationsschritt der Clusteranalyse unterhalb dieses Schwellwerts, wird die Clusteranalyse abgebrochen. In Abschnitt 6.5 wird die Optimierung des Abbruchkriteriums betrachtet.
Aus der resultierenden Clusterhierarchie werden anschließend die Absichtsknoten der semantischen Quelltextrepräsentation erzeugt. Für die Wahl der Absichtsknoten in dieser Arbeit werden die Cluster der höchsten Hierarchieebene herangezogen. Cluster, die lediglich ein Programmelement umfassen, werden nicht weiter berücksichtigt. Um Programmelemente mehreren Clustern zuordnen zu können, ist eine feingranulare Differenzierung zwischen den Clustern in der Clusterhierarchie denkbar. Die, in Abschnitt 5.4.2 beschriebene, Möglichkeit hierfür ist jedoch mit einer hohen Unsicherheit verbunden, weshalb für die Hinzunahme weiterer Cluster aus der Clusterhierarchie zusätzliche Analysen erforderlich sind, die über den Rahmen dieser Arbeit hinausgehen.
Für jedes Cluster der höchsten Hierarchieebene, welches mehr als 1 Programmelement enthält, wird ein Absichtsknoten im Graphen erzeugt, der Kanten zu allen enthaltenen Programmelementen besitzt. Darüber hinaus wird den Kanten ein Gewicht zugeordnet, welches die Sicherheit, mit der ein Programmelement zum Cluster gehört, beschreibt. Das Gewicht wird, in Anlehnung an Abschnitt 5.4.2, mit Gleichung 5.22 berechnet und entspricht dem durchschnittlichen semantischen Zusammenhang eines Programmelements zu den anderen Programmelementen im selben Cluster.

$$rel(e, c) = \frac{1}{|c| - 1} \sum_{k \in c \setminus e} semRel(e, k) \quad (5.22)$$

5.5 Systementwurf

Der nachfolgende Abschnitt fasst die zuvor beschriebenen Lösungsentwürfe der einzelnen Teilziele zusammen. Abbildung 5.10 zeigt den abstrakten Systementwurf für den Aufbau der semantischen Quelltextrepräsentation. Als Eingabe wird Quelltext erwartet. Dieser wird im ersten Schritt in die Graphstruktur der semantischen Repräsentation überführt. Im Einklang mit dem Entwurf der semantischen Repräsentation aus Abschnitt 5.2.2 werden Programmelemente als Knoten und syntaktische Abhängigkeiten als Kanten dargestellt. Auf der Graphrepräsentation werden anschließend eine lexikalische und eine syntaxbasierte Quelltextanalyse durchgeführt, um semantische Zusammenhänge zwischen Methoden, Typen und zwischen Methoden und Typen zu identifizieren. Die Quelltextanalysen arbeiten unabhängig voneinander und fügen ihre gefundenen Zusammenhänge als gewichtete Kanten der Graphrepräsentation hinzu. Die **syntaxbasierte Quelltextanalyse** aus Abschnitt 5.3.2 untersucht die strukturellen Abhängigkeiten im Quelltext und berücksichtigt

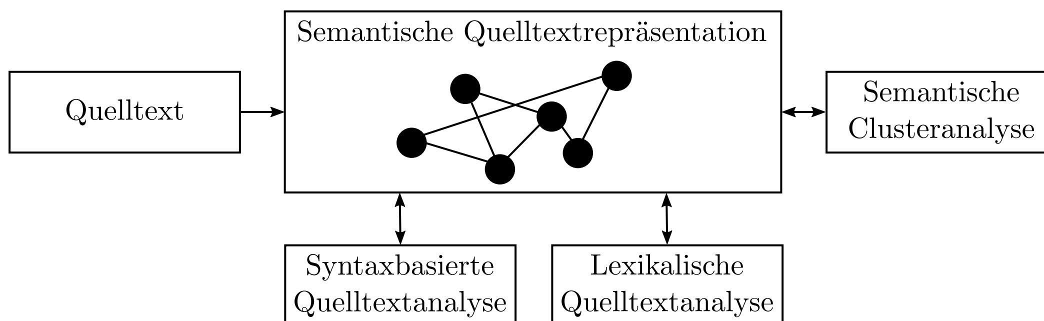


Abbildung 5.10: Abstrakter Systementwurf

sowohl direkte als auch indirekte Abhängigkeiten zwischen Programmelementen. Bei der **lexikalischen Quelltextanalyse** aus Abschnitt 5.3.4 werden die Kommentare, Bezeichner und Zeichenketten im Quelltext auf Zusammenhänge untersucht. Nach Ablauf der beiden Analysen erfolgt die **semantische Clusteranalyse** aus Abschnitt 5.4.3. Sie liest die gefundenen Zusammenhänge der beiden Analysen aus dem Graphen aus, fasst sie zu semantischen Zusammenhängen zwischen Programmelementen zusammen und fügt diese als neue Kanten in den Graphen ein. Auf der Basis dieser semantischen Zusammenhänge erfolgt anschließend eine hierarchische Clusteranalyse, um Gruppen von Programmelementen mit einer gemeinsamen Absicht zu identifizieren. Für jedes Cluster wird ein Absichtsknoten in die Graphrepräsentation eingefügt und mit den jeweiligen Programmelementknoten verbunden. Insgesamt entsteht somit die, mit dieser Arbeit entworfene und umgesetzte, semantische Quelltextrepräsentation.

6 Implementierung

In diesem Kapitel wird die Implementierung der, in Kapitel 5 entworfenen, Schritte für den Aufbau der semantischen Quelltextrepräsentation vorgestellt. Ausgangspunkt der Entwicklung ist die PARSE-Rahmenarchitektur [WT15]. Als Eingabe wird Quelltext erwartet. In einem Vorverarbeitungsschritt wird dieser in eine initiale Graphrepräsentation überführt. Die Graphrepräsentation der Eingabe ist das zentrale Element der PARSE-Architektur und bildet die Grundlage für alle ablaufenden Analysen. Sowohl die beiden Quelltextanalysen als auch die Clusteranalyse werden als PARSE-Agenten (Siehe Kapitel 3) umgesetzt. Sie arbeiten unabhängig voneinander und beziehen alle notwendigen Informationen für die Ausführung aus dem Quelltextgraph. Die Ergebnisse der Analysen werden von den einzelnen Agenten in den Quelltextgraph eingefügt, sodass die Graphrepräsentation am Ende der semantischen Quelltextrepräsentation entspricht. Die folgenden Abschnitte gehen auf die Implementierung der Vorverarbeitung sowie auf die Implementierung der einzelnen Agenten im Detail ein. Abschließend werden in Abschnitt 6.5 die Parameter des implementierten Systems betrachtet und eine optimierte Konfiguration vorgestellt.

6.1 Aufbau der initialen Graphrepräsentation

Der Aufbau der initialen Graphrepräsentation erfolgt auf Basis der Informationen des AST. Entsprechend dem Entwurf aus Abschnitt 5.2.2 werden Programmelemente als Knoten dargestellt und Abhängigkeiten durch gerichtete Kanten repräsentiert. Eine Auflistung der Knoten und Kanten des Graphen findet sich in Abschnitt B des Anhangs.
Der Aufbau des Graphen erfolgt in zwei Schritten. Im ersten Schritt werden die benötigten Informationen aus dem Quelltext extrahiert und in einer Zwischenrepräsentation gespeichert. In dieser Arbeit wird die Informationsextraktion für Java-Quelltext umgesetzt. Als Werkzeug für den Zugriff auf die Informationen im Quelltext wird Spoon[1] aus [PMP+16] genutzt. Im zweiten Schritt werden die Informationen aus der Zwischenrepräsentation genutzt, um die Knoten und Kanten des Graphen zu erzeugen. Durch die Entkopplung der Informationsextraktion vom Aufbau der Graphrepräsentation, kann der Aufbau des initialen Quelltextgraphen einfach für andere objektorientierte Programmiersprachen implementiert werden.

Mit Hilfe von Spoon werden im ersten Schritt Pakete, Klassen, Aufzählungstypen, Schnittstellen, Methoden und Anweisungen aus dem Quelltext extrahiert. Für jedes dieser Pro-

[1] `http://spoon.gforge.inria.fr/`, zuletzt besucht am 11.12.2019

grammelemente wird ein eigenes Verarbeitungsfließband (engl. processing pipeline) definiert, welches sich aus einer Reihe von Extraktionsschritten zusammensetzt. Jeder Extraktionsschritt implementiert ein konkretes Informationsbedürfnis wie die Extraktion des Namen eines Typs oder der Aufrufabhängigkeiten einer Anweisung. Hierbei liest ein Extraktionsschritt alle notwendigen Informationen aus dem Spoon-Objekt aus und speichert sie in der Zwischenrepräsentation. Durch die Umsetzung als Verarbeitungsfließband ist die Verarbeitung der Programmelemente einfach konfigurier- und erweiterbar. Am Ende der Informationsextraktion liegt für jedes Programmelement im Quelltext eine Zwischenrepräsentation vor, welche die benötigten Informationen für den Aufbau der initialen Graphrepräsentation aus Abschnitt 5.2.2 enthält. Programmelemente wie Typen oder Methoden, die nicht im Quelltext definiert, sondern über externe Quellen eingebunden wurden, bleiben hierbei unberücksichtigt.
Auf der Basis der Zwischenrepräsentation wird im zweiten Schritt der Quelltextgraph aufgebaut. Für jedes Programmelement der Zwischenrepräsentation wird ein neuer Knoten im Graph angelegt. Die Art eines Programmelements wird hierbei als Knotentyp genutzt. Vorhandene Kommentare, die Position im Quelltext und weitere Merkmale wie der Name eines Programmelements werden als Attribute den Knoten hinzugefügt. Nachdem alle Programmelemente als Knoten im Graph vorliegen, werden die Abhängigkeiten zwischen den Programmelementen als gerichtete Kanten hinzugefügt. Um zwischen den Kanten differenzieren zu können, wird ihnen ein Typ zugeordnet, der die Art der Abhängigkeit beschreibt. Ein Methodenaufruf führt bspw. zu einer Kante mit dem Typ *methodCall* vom Anweisungsknoten zum Knoten der aufgerufenen Methode. Insgesamt entsteht auf diese Weise die, in Abschnitt 5.2.2 beschriebene, initiale Graphrepräsentation. Eine Zusammenfassung der Knoten- und Kantentypen des Graphen ist in Abschnitt B des Anhangs beigefügt.

6.2 Syntaxbasierte Quelltextanalyse

Die syntaxbasierte Quelltextanalyse aus Abschnitt 5.3.2 wird als PARSE-Agent umgesetzt. Die strukturellen Abhängigkeiten im Quelltext werden aus der Graphrepräsentation des Quelltextes ausgelesen. Für die Bewertung der Zusammenhänge im Quelltext werden für alle Knotenpaare aus der Menge der Typ- und Methodenknoten jeweils die direkte und indirekte Abhängigkeit berechnet. Die Präzision der ablaufenden Berechnungen beträgt drei Dezimalstellen. Algorithmus 1 zeigt das abstrakte Vorgehen.
Für die Berechnung der direkten Abhängigkeit zweier Programmelemente werden die in Tabelle 5.4 aufgeführten Abhängigkeitsmerkmale genutzt. Für die Implementierung der Abhängigkeitsmerkmale existiert eine Schnittstelle `IDependency`, welche eine Methode `getDependency` besitzt, die als Parameter zwei Programmelementknoten erwartet und den Abhängigkeitswert, bezogen auf ein konkretes Abhängigkeitsmerkmal, berechnet. Beim Aufruf von `getDependency` in Zeile fünf von Algorithmus 1 werden alle Abhängigkeitsmerkmale durchlaufen und der gewichtete Mittelwert über die berechneten Werte gebildet. Die Gewichte der einzelnen Abhängigkeitsmerkmale werden von $\text{weights}_{\text{sda}}$ definiert und über eine Konfigurationsdatei eingelesen.
Das Fundament für die Bewertung der indirekten Abhängigkeiten bilden die zuvor berechneten direkten Abhängigkeiten. Beim Aufruf von `getDependency` in Zeile 10 von Algorithmus 1 wird ein Vergleich zwischen den Vektorabbildungen der beiden übergebenen Programmelemente durchgeführt (Siehe Abschnitt 5.3.2). Die Einträge der Vektorabbildung eines Programmelements beschreiben die direkten Abhängigkeiten zu allen anderen Programmelementen im Quelltext. Um Redundanzen mit der direkten Abhängigkeit zwischen e_1 und e_2 zu vermeiden, werden die Dimensionen der beiden zu vergleichenden Programmelemente aus den Vektorabbildungen entfernt. Folglich entspricht die Dimension der Vektoren der Anzahl der Programmelemente im Quelltext minus zwei. Das Ergebnis der indirekten Abhängigkeit wird über den Kosinus des Winkels zwischen den Vektorab-

Algorithmus 1 Syntaxbasierte Quelltextanalyse

Require: graph, $\text{weights}_{\text{dependencies}}, \alpha_{\text{sda}}$
1: elements $\leftarrow$ graph.getTypeNodes() $\cup$ graph.getMethodNodes()
2: $\text{analyzer}_{\text{direct}} \leftarrow$ DirectDepAnalyzer($\text{weights}_{\text{dependencies}}$)
3: $\text{dependencies}_{\text{direct}} \leftarrow \{\}$
4: **for** $(e_1, e_2) \in$ elements **do**
5: $\quad \text{dependencies}_{\text{direct}}[(e_1, e_2)] \leftarrow \text{analyzer}_{\text{direct}}$.getDependency($e_1, e_2$)
6: **end for**
7: $\text{analyzer}_{\text{indirect}} \leftarrow$ IndirectDepAnalyzer($\text{dependencies}_{direct}$)
8: $\text{relations}_{\text{sda}} \leftarrow \{\}$
9: **for** $(e_1, e_2) \in$ elements **do**
10: $\quad \text{dependency}_{\text{indirect}} \leftarrow \text{analyzer}_{\text{indirect}}$.getDependency($e_1, e_2$)
11: $\quad$ relation $\leftarrow$ max($\text{dependency}_{\text{indirect}}, \text{dependencies}_{\text{direct}}[(e_1, e_2)]$)
12: $\quad$ **if** relation $> \alpha_{\text{sda}}$ **then**
13: $\quad\quad \text{relations}_{\text{sda}}[(e_1, e_2)] \leftarrow$ relation
14: $\quad$ **end if**
15: **end for**
16: insertRelations(graph, $\text{relations}_{\text{sda}}$)

bildungen berechnet.
Der syntaxbasierte Zusammenhang zweier Programmelemente wird abschließend durch das Maximum aus der direkten und indirekten Abhängigkeit beschrieben. Liegt dieser Wert über dem Schwellwert α_{sda}, wird eine neue ungerichtete Kante vom Typ *sdaRelation* zwischen den beiden Programmelementknoten im Graphen angelegt. Als Gewicht erhält die Kante den berechneten Zusammenhang. Analog zu den Gewichten der Abhängigkeitsmerkmale wird der Schwellwert α_{sda} über eine Konfigurationsdatei eingelesen.

6.3 Lexikalische Quelltextanalyse

Die lexikalische Quelltextanalyse implementiert den Entwurf aus Abschnitt 5.3.4 und wird ebenfalls als PARSE-Agent umgesetzt. Analog zur syntaxbasierten Quelltextanalyse werden alle notwendigen Informationen für die Analyse aus der Graphrepräsentation des Quelltextes bezogen. Zur Bewertung der Zusammenhänge werden für alle Knotenpaare aus der Menge der Typ- und Methodenknoten eine Bezeichneranalyse (*BA*), eine Zeichenketten- und Kommentaranalyse (*ZKA*) und eine Analyse für den Vergleich zwischen Zeichenketten bzw. Kommentaren und Bezeichnern (*ZKBA*) durchgeführt. Die Präzision der durchgeführten Berechnungen beträgt drei Dezimalstellen. Algorithmus 2 zeigt den abstrakten Ablauf der lexikalischen Analyse. Alle drei durchgeführten Analysen nutzen eine Kombination aus der, in Abschnitt 5.3.4 beschriebenen, Schwerpunkt- und Zeichenkettenanalyse. Abschnitt 6.3.3 beschreibt die Implementierung der beiden Analysen. Der Unterschied zwischen *BA*, *ZKA* und *ZKBA* liegt in der genutzten Dokumentabbildung. Die *BA*-Analyse nutzt die Abbildung `ident2doc`, welche ein Programmelement auf seine Bezeichner abbildet. Die *ZKA*-Analyse hingegen nutzt die Abbildung `commentsAndStrings2doc`, welche ein Programmelement auf die Menge seiner Kommentare und Zeichenketten abbildet. Die dritte Analyse (*ZKBA*) vergleicht die verschiedenen Dokumentabbildungen untereinander und nutzt daher beide Abbildungsfunktionen. In Abschnitt 6.3.1 werden die Dokumentabbildungen im Detail vorgestellt.
Neben der Abbildungsfunktion wird bei der Instanziierung einer Analyse ein Gewichtsvektor übergeben. Dieser enthält die, bei der Schwerpunktberechnung, genutzten Gewichte zur Bestimmung der Relevanz der Signatur- und Anweisungsebene. Darüber hinaus enthält er zwei weitere Gewichte für die Definition des Einflusses des Schwerpunktvergleichs

Algorithmus 2 Lexikalische Quelltextanalyse

Require: graph, $\text{weights}_{\text{la}}, \text{weights}_{\text{zka}}, \text{weights}_{\text{zkba}}, \text{weights}_{\text{la}}, \alpha_{\text{la}}$
1: ba ← Analyzer(ident2doc, $\text{weights}_{\text{ba}}$)
2: zka ← Analyzer(commentsAndStrings2doc, $\text{weights}_{\text{zka}}$)
3: zkba ← Analyzer(commentsAndStrings2doc, ident2doc, $\text{weights}_{\text{zkba}}$)
4: analyzers ← [ba, zka, zkba]
5: elements ← graph.getTypeNodes() ∪ graph.getMethodNodes()
6: relations ← {}
7: **for** $(e_1, e_2) \in$ elements **do**
8: relation ← 0
9: **for** analyzer ∈ analyzers **do**
10: relation ← relation + $\text{weights}_{\text{la}}$[analyzer] ∗ analyzer.rate(e_1, e_2)
11: **end for**
12: **if** relation > α_{la} **then**
13: relations[(e_1, e_2)] ← relation
14: **end if**
15: **end for**
16: insertRelations(graph, relations)

und des Zeichenkettenvergleichs.
Der lexikalische Zusammenhang zweier Programmelemente setzt sich aus der gewichteten Summe der Ergebnisse der Teilanalysen (*BA*, *ZKA* und *ZKBA*) zusammen. Die Gewichte der Teilanalysen werden durch $\text{weights}_{\text{la}}$ bestimmt. Analog zur syntaxbasierten Quelltextanalyse, legt ein Schwellwert α_{la} die Mindestgröße fest, die ein berechneter Zusammenhang besitzen muss, um in den Graphen eingefügt zu werden. Alle in der lexikalischen Quelltextanalyse genutzten Gewichte werden über eine Konfigurationsdatei eingelesen und sind auf einfache Weise anpassbar. Die Funktion `insertRelations` fügt die gefundenen Zusammenhänge in den Graphen ein. Hierbei werden ungerichtete Kanten vom Typ *laRelation* zwischen den, als zusammenhängend erkannten, Programmelementen im Graph eingefügt. Das Gewicht der Kanten entspricht dabei dem berechneten lexikalischen Zusammenhang.

6.3.1 Dokumentabbildung

Voraussetzung für die lexikalische Quelltextanalyse sind die zwei Abbildungsvorschriften `ident2doc` und `commentsAndStrings2doc`. Sie bilden ein Programmelement auf eine Menge von Wörtern ab. Hierbei werden die lexikalischen Bestandteile eines Programmelements extrahiert und anschließend vorverarbeitet.

Textextraktion

Für die Extraktion der natürlichen Sprache aus den Programmelementknoten werden verschiedene Extraktionsschritte implementiert. Jeder Extraktionsschritt widmet sich entweder den Bezeichnern, den Kommentaren oder den Zeichenketten eines Programmelements. Die Auftrennung der Textextraktion in einzelne Schritte ermöglicht eine flexible Konfiguration der Dokumentabbildung. Die lexikalischen Informationen eines Programmelements liegen als Attribute im Programmelementknoten vor. Bei der Extraktion der Textbestandteile wird die Information beibehalten, ob ein Begriff aus der Signatur- oder der Anweisungsebene stammt. Die Textextraktion in `ident2dec` besteht aus vier Extraktionsschritten, um für Methoden und Typen jeweils die Bezeichner aus der Signatur- und der Anweisungsebene zu extrahieren. Bei der Textextraktion in `commentsAndStrings2doc` werden sechs Extraktionsschritte verwendet. Vier, um für Methoden und Typen jeweils die Kommentare aus der Signatur- und der Anweisungsebene zu extrahieren und zwei weitere, um die Zeichenketten

aus Methoden und Typen zu extrahieren. Bei der Extraktion von Zeichenketten wird nicht zwischen Signatur- und Anweisungsebene unterschieden, da Zeichenketten in Java nicht in der Signatur auftreten können. Alle extrahierten Informationen werden entsprechend ihrer Position im Quelltext sortiert, sodass ihre Originalreihenfolge erhalten bleibt. Das Ergebnis der Textextraktion ist eine sortierte Liste von Zeichenketten-Objekten, jeweils ein Objekt pro Bezeichner, Kommentar oder Zeichenkette eines Programmelements.

Vorverarbeitung

In Übereinstimmung mit dem Entwurf der Vorverarbeitung aus Abschnitt 5.3.4 wird für die extrahierten Bezeichner und Kommentare bzw. Zeichenketten ein Vorverarbeitungsfließband (engl. preprocessing pipeline) implementiert. Es setzt sich aus den folgenden Verarbeitungsschritten zusammen, die nacheinander auf der extrahierten Textmenge eines Programmelements ausgeführt werden. Die ersten drei Verarbeitungsschritte sind hierbei den Kommentaren bzw. Zeichenketten vorbehalten.

- **Kommentare filtern:** In diesem Schritt werden irrelevante Kommentare herausgefiltert. Hierzu zählen Lizenzbeschreibungen, allgemeine *todo*- oder *fixme*-Kommentare und Kommentare, die aus auskommentiertem Quelltext bestehen. Für die Identifikation von Lizenzbeschreibungen wird das Vorkommen der Schlüsselwörter *license* und *copyright* im Kommentar gezählt. Ein Untersuchung verschiedener Lizenzbeschreibungen ergab, dass bei drei oder mehr Vorkommen der Schlüsselwörter angenommen werden kann, dass ein Kommentar eine Lizenzbeschreibung darstellt. Folglich werden alle Kommentare entfernt, die insgesamt mindestens drei solcher Schlüsselwörter enthalten. Auskommentierter Quelltext wird über die Dichte der Java-Schlüsselwörter im Kommentar erkannt. Kommentare, die zu mehr als 40% aus Java-Schlüsselwörtern bestehen, werden entfernt. Dieser Wert hat sich ebenfalls in mehreren Tests bewährt. Kommentare, die mit *todo* oder *fixme* beginnen und keinen der Bezeichner im Quelltext als Wort enthalten, werden ebenfalls entfernt, um zu vermeiden, dass zu allgemeine Kommentare in die Analyse einfließen. Falls ein Bezeichner im *todo*- oder *fixme*-Kommentar enthalten ist, wird dieser beibehalten. Allerdings wird das Wort *todo* bzw. *fixme* aus dem Kommentar entfernt.

- **Kommentare bereinigen:** Mit Hilfe von regulären Ausdrücken werden JavaDoc-Markierungen in den Kommentaren erkannt. Markierungen wie `@author`, `@version`, `@since`, `@code` und `@deprecated`, die keine relevanten Informationen tragen, werden vollständig entfernt. Bei allen anderen Markierungen wie `@link` wird der Text innerhalb der Markierung beibehalten und lediglich die JavaDoc-Syntax entfernt. HTML-Formatierungselemente werden über das Werkzeug jsoup[2] entfernt, wobei der Text innerhalb der HTML-Elemente beibehalten wird. Des Weiteren werden Internetadressen über reguläre Ausdrücke erkannt und entfernt.

- **Kommentare und Zeichenketten zerteilen:** Der Eingabetext wird an den Leer- und Satzzeichen aufgetrennt, um eine Menge von Termen zu erhalten. Hierbei werden Zeilenumbrüche und Satzzeichen entfernt.

- **Auflösen zusammengesetzter Terme:** Alle Wörter, die in der Binnenmajuskel-Schreibweise geschrieben sind, Unterstriche oder Zahlen enthalten, werden als zusammengesetzte Terme aufgefasst. Jeder zusammengesetzte Term wird an den Großbuchstaben, Zahlen oder Unterstrichen aufgetrennt. Gleichzeitig werden die aufgetrennten Wörter markiert, sodass im späteren Verlauf der Analyse bekannt ist, welche Wörter einen zusammengesetzten Term im Quelltext darstellen.

[2] `https://jsoup.org/`, zuletzt besucht am 22.12.2019

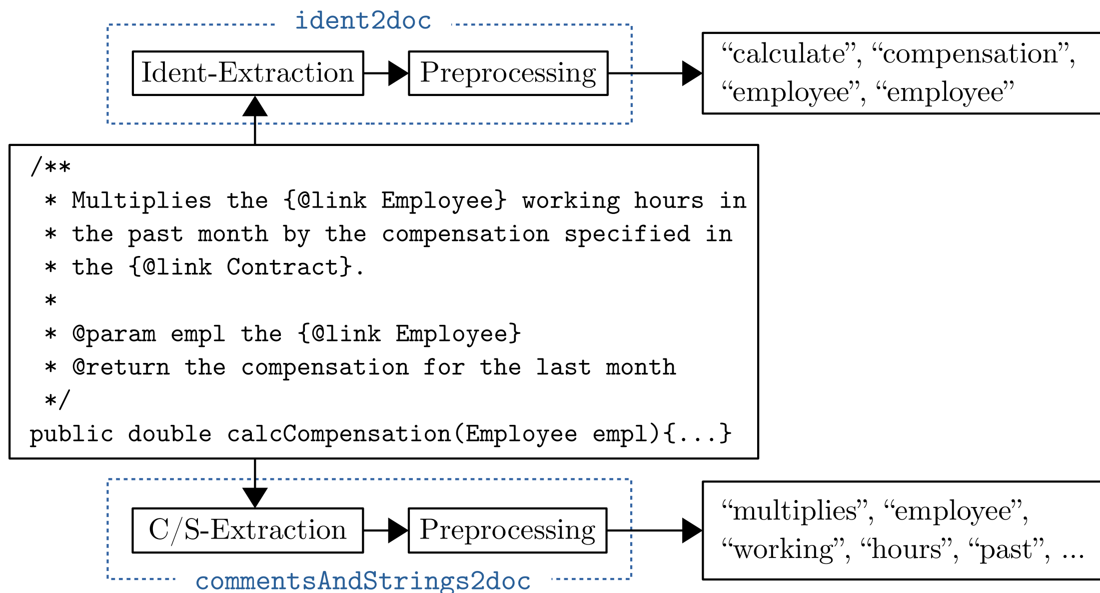


Abbildung 6.1: Beispiel für die Dokumentabbildung einer Methode.

- **Abbildung auf Kleinbuchstaben:** Alle Buchstaben werden auf Kleinbuchstaben abgebildet.
- **Stoppwortentfernung:** Für die Stoppwortentfernung wird eine Liste von Wörtern definiert, die aus der Eingabe entfernt werden. Die Liste setzt sich aus den Stoppwörtern der natürlichen englischen Sprache[3], den Schlüsselwörtern der Programmiersprache Java[4] und dem Wort *var* zusammen. Des Weiteren werden alle Zahlen und Wörter, die lediglich aus einem Buchstaben bestehen, entfernt.
- **Abkürzungsauflösung:** Bei der Abkürzungsauflösung wird das Vokabular des Worteinbettungsmodells genutzt, um unbekannte Wörter zu identifizieren. Jedes unbekannte Wort wird als potentielle Abkürzung aufgefasst. Anschließend wird der SE-Thesaurus[5] aus [CCZX19] als Wörterbuch verwendet, um Abkürzungen durch ihre ausgeschriebene Bezeichnung zu ersetzen. Hierbei wird geprüft, ob das Wörterbuch ein unbekanntes Wort als Abkürzung enthält. Ist das der Fall, wird geprüft, ob die ausgeschriebene Form der Abkürzung im Vokabular des Worteinbettungsmodells enthalten ist und falls ja, wird die Abkürzung durch ihre ausgeschriebene Form ersetzt.

Zusammengefasst besteht eine Dokumentabbildung aus einer individuellen Textextraktion und dem Vorverarbeitungsfließband. Abbildung 6.1 veranschaulicht die Dokumentabbildung für eine Methode. Bei der Abbildung `ident2doc` werden die Bezeichner extrahiert und anschließend vorverarbeitet. Im Rahmen der Vorverarbeitung wird der Bezeichner `calcCompensation` aufgetrennt und die Abkürzung „calc“ durch ihre ausgeschriebene Form „calculate“ ersetzt. Darüber hinaus wird die Parameterbezeichnung als Abkürzung erkannt und durch „employee“ ersetzt.
Die Abbildung `commentsAndStrings2doc` extrahiert den Text im Kommentar der Methode. Bei der Vorverarbeitung wird die JavaDoc-Markierungen `@link` entfernt, wobei die darin enthaltenen Wörter beibehalten werden. Auch die Markierungen für die Beschreibung des Parameters (`@param`) und der Rückgabe (`@return`), sowie die im Text enthaltenen Stoppwörter, werden entfernt. Die Ausgabe der Abbildungen entspricht der Dokumentabbildung des Programmelements hinsichtlich der betrachteten textuellen Bestandteile.

[3] `https://www.lextek.com/manuals/onix/stopwords2.html`, zuletzt besucht am 22.12.2019
[4] `https://en.wikipedia.org/wiki/List_of_Java_keywords`, zuletzt besucht am 17.12.2019
[5] `https://sedict.github.io/`, zuletzt besucht am 22.12.2019

Jedes Wort in der Dokumentabbildung wird zusätzlich mit Informationen angereichert, sodass später nachvollziehbar ist, welche Wörter in der Signatur- bzw. Anweisungsebene aufgetreten sind. Darüber hinaus wird die Information beibehalten, welche Wörter als zusammenhängender Term im Quelltext vorkamen.

6.3.2 Worteinbettung

Die Grundlage für die Schwerpunktanalyse aus Abschnitt 5.3.4 sind Wortvektoren. Für die Abbildung von Wörtern auf Vektoren wird fastText[6] aus [BGJM17] verwendet. Da fastText lediglich als Python[7]-Modul verfügbar ist, kann es nicht direkt in die Java-Implementierung der lexikalischen Quelltextanalyse eingebunden werden. Eine Importierung des Worteinbettungsmodells über die Java-Bibliothek Deeplearning4j[8] ist aufgrund des hohen Hauptspeicherbedarfs von mehr als 16 Gigabyte ebenfalls nicht möglich. Für die Nutzung von fastText in dieser Arbeit wird eine minimalistische Webanwendung in Python implementiert. Hierfür wird das Programmiergerüst (engl. framework) flask[9] genutzt. Die implementierte Webanwendung bietet drei Schnittstellen an, um fastText aus der Java-Implementierung heraus aufrufen zu können. Über die erste Schnittstelle kann ein bestehendes Worteinbettungsmodell in fastText geladen werden. Hierbei wird der Dateipfad zum Modell als Parameter mitgegeben. Die zweite Schnittstelle erwartet eine Liste von Wörtern als Parameter und liefert die Informationen zurück, welche der Wörter im Vokabular des Worteinbettungsmodells vorhanden sind. Die letzte Schnittstelle erwartet ebenfalls eine Liste von Wörtern als Eingabe und liefert eine Abbildung der Wörter auf reellwertige Vektoren zurück. Die Webanwendung wird zu Beginn der lexikalischen Analyse automatisiert als separater Prozess gestartet und am Ende der Analyse wieder beendet.

Aus dem Entwurf der lexikalischen Analyse in Abschnitt 5.3.4 geht hervor, dass der Einsatz verschiedener Worteinbettungsmodelle auf der Grundlage unterschiedlicher Trainingskorpora evaluiert werden soll. Das erste und standardmäßig genutzte Worteinbettungsmodell stammt aus der Arbeit [BGJM17] von Bojanowski et al. und wurde auf Texten aus Wikipedia trainiert. Es kann auf der Website von fastText heruntergeladen werden.
Das zweite Worteinbettungsmodell wird auf Beiträgen und Kommentaren von StackOverflow[10] trainiert. Bei der Vorverarbeitung werden alle Quelltextausschnitte, Satzzeichen, Formatierungs- und Strukturelemente entfernt sowie alle Buchstaben auf Kleinbuchstaben abgebildet. Die vorverarbeitete Textmenge ist ca. 60 Gigabyte groß. Beim Training des Worteinbettungsmodells werden die Standardparameter von fastText verwendet.
Das dritte Worteinbettungsmodell stammt aus der Arbeit [ES19] von Efstathiou und Spinellis und wurde auf den Texten von ca. 3000 Java-Softwareprojekten aus GitHub trainiert.

6.3.3 Vergleich von Programmelementen

Der Vergleich von Programmelementen basiert auf einer Kombination aus einer Schwerpunktanalyse und einer Zeichenkettenanalyse. Als Eingabe werden die Dokumentabbildungen zweier Programmelemente erwartet.

Schwerpunktanalyse

Bei der Schwerpunktanalyse werden die Schwerpunktvektoren der Dokumentabbildungen über die Kosinus-Ähnlichkeit verglichen. Wie in Abschnitt 5.3.4 beschrieben, setzt sich der Schwerpunktvektor eines Programmelements aus den Schwerpunktvektoren der Wörter aus

[6] `https://fasttext.cc/`, zuletzt besucht am 23.12.2019
[7] `https://www.python.org/`, zuletzt besucht am 23.12.2019
[8] `https://deeplearning4j.org/`, zuletzt besucht am 23.12.2019
[9] `https://palletsprojects.com/p/flask/`, zuletzt besucht am 27.12.2019
[10] `https://archive.org/download/stackexchange`, zuletzt besucht am 27.12.2019

der Signatur- und Anweisungsebene zusammen. Bei der Berechnung werden zunächst alle Wörter in den Dokumentabbildungen mit Hilfe des Worteinbettungsmodells auf Vektoren abgebildet. Für zusammengesetzter Terme besteht ein zusätzlicher Verarbeitungsschritt. Die Vektoren der Wörter, die im Quelltext einen zusammengesetzten Term gebildet haben, werden über Gleichung 5.14 zu einem Vektor zusammengefasst. Dieser Vektor ersetzt daraufhin die Wortvektoren der Teilwörter des zusammengesetzten Terms. Auf der Grundlage der entstandenen Vektormenge werden über Gleichung 5.16 anschließend Schwerpunktvektoren für die Signatur- und Anweisungsebene eines Programmelementdokuments berechnet. Hierbei werden die Wortvektoren entsprechend des, über Gleichung 5.15 berechneten, Gewichts des repräsentierten Begriffs gewichtet. Der finale Schwerpunktvektor eines Programmelementdokuments entspricht der gewichteten Summe der Schwerpunktvektoren der Signatur- und Anweisungsebene. Abschließend werden die zusammengesetzten Schwerpunktvektoren zweier Programmelemente über die Kosinus-Ähnlichkeit bewertet.

Zeichenkettenanalyse

Der Zeichenkettenvergleich dient, wie in Abschnitt 5.3.4 beschrieben, der Behandlung von unbekannten Wörtern. Die Identifikation unbekannter Wörter erfolgt durch einen Abgleich mit dem Vokabular des verwendeten Worteinbettungsmodells. Für zwei Programmelementdokumente d_1 und d_2 wird die Menge der unbekannten Wörter aus d_1 mit der Menge an Wörtern in d_2 über Gleichung 5.18 verglichen und umgekehrt. Als Ähnlichkeitsmaße werden die Jaro-Winkler-Ähnlichkeit und der FuzzyScore genutzt. Beide Metriken werden über Apache Commons Text[11] eingebunden. Im Gegensatz zur Jaro-Winkler-Ähnlichkeit, ist das Ergebnis des FuzzyScore nicht normiert. Die Schwellwerte aus Gleichung 5.17 erfordern jedoch für beide Ähnlichkeitsmaße ein Ergebnis zwischen 0 und 1. Infolgedessen wird die Implementierung des FuzzyScore erweitert, sodass beim Vergleich zweier Zeichenketten s_1 und s_2 das Ergebnis mit der maximal erreichbaren Ähnlichkeit $(3 * max(|s_1|, |s_2|)) - 2$ normiert wird.
Das Ergebnis des Zeichenkettenvergleichs ist der, über Gleichung 5.19 berechnete, Mittelwert aus den Übereinstimmungen der unbekannten Wörter aus d_1 mit den Wörtern aus d_2 und umgekehrt.

Im Einklang mit dem Entwurf aus Abschnitt 5.3.4 wird der Zeichenkettenvergleich nur dann ausgeführt, wenn unbekannte Wörter in einem Programmelementdokument vorliegen. Existieren unbekannte Wörter in einem der Eingabedokumente, wird das Ergebnis der Schwerpunktanalyse mit $\text{weight}_{\text{centroid}}$ und das Ergebnis der Zeichenkettenanalyse mit $\text{weight}_{\text{string}}$ gewichtet und anschließend addiert. Beide Gewichte sind Teil der, bei der Initialisierung der Analysen in Algorithmus 2 übergebenen, Gewichtsvektoren. Enthalten die Eingabedokumente keine unbekannten Wörter, wird das Ergebnis der Schwerpunktanalyse mit 1 gewichtet. Die berechneten lexikalischen Zusammenhänge zwischen den Programmelementen werden abschließend in die Graphrepräsentation des Quelltextes eingefügt. Hierbei wird für jeden Zusammenhang größer als α_{la} eine neue ungerichtete Kante vom Typ *la-Relation* zwischen den jeweiligen Programmelementknoten erzeugt. Das Gewicht der Kante entspricht dem berechneten lexikalischen Zusammenhang.

6.4 Semantische Clusteranalyse

Die semantische Clusteranalyse aus Abschnitt 5.4.3 wird ebenfalls als Agent implementiert. Alle benötigten Informationen für die Analyse werden aus dem Quelltextgraphen ausgelesen. Da die semantische Clusteranalyse auf den Ergebnissen der zuvor beschriebenen Quelltextanalysen aufsetzt, erfolgt die Ausführung der Clusteranalyse stets nachdem die

[11] `https://commons.apache.org/proper/commons-text/javadocs/api-release/index.html`, zuletzt besucht am 23.12.2019

verschiedenen Quelltextanalysen abgeschlossen wurden. Algorithmus 3 zeigt den abstrakten Ablauf der Clusteranalyse. Alle Parameter und Gewichte der semantischen Clusteranalyse werden über eine Konfigurationsdatei eingelesen und sind folglich einfach anpassbar. Zu Beginn werden die gefundenen Zusammenhänge der lexikalischen und syntaxbasier-

Algorithmus 3 Semantische Clusteranalyse

Require: graph, $\text{weight}_{\text{la}}, \text{weight}_{\text{sda}}, \alpha_{\text{cluster}}, \alpha_{\text{semantic}}$
1: elements ← graph.getTypeNodes() ∪ graph.getMethodNodes()
2: semanticRelations ← {}
3: **for** $(e_1, e_2) \in$ elements **do**
4: $\text{relation}_{\text{la}}$ ← graph.getEdge(e_1, e_2, "laRelation").weight
5: $\text{relation}_{\text{sda}}$ ← graph.getEdge(e_1, e_2, "sdaRelation").weight
6: semanticRelation ← $\text{relation}_{\text{la}} * \text{weight}_{\text{la}} + \text{relation}_{\text{sda}} * \text{weight}_{\text{sda}}$
7: **if** semanticRelation $> \alpha_{\text{semantic}}$ **then**
8: semanticRelations[(e_1, e_2)] ← semanticRelation
9: **end if**
10: **end for**
11: insertRelations(graph, semanticRelations)
12: clusterHierarchy ← hierarchicalClustering(elements, semanticRelations, α_{cluster})
13: clusters ← getIntentClusters(clusterHierarchy)
14: insertClusters(graph, clusters)

ten Quelltextanalyse aus dem Graph ausgelesen und, wie in Abschnitt 5.4.3 beschrieben, zu semantischen Zusammenhängen kombiniert. Falls zwei Programmelemente keine Kante vom Typ *sdaRelation* oder *laRelation* besitzen, wird ein Zusammenhang von 0 angenommen. Die Gewichte der einzelnen Quelltextanalysen werden über die Parameter α_{la} bzw. α_{sda} bestimmt. Alle berechneten semantischen Zusammenhänge größer als der Schwellwert α_{semantic} werden als Kanten vom Typ *semanticRelation* in den Graph eingefügt, wobei die berechneten semantischen Zusammenhänge als Kantengewichte genutzt werden.

Um eine Unterteilung der Methoden und Typen in semantisch zusammenhängenden Gruppen zu finden, wird die, in Abschnitt 5.4.3 beschriebene, agglomerative hierarchische Clusteranalyse umgesetzt. Zu Beginn wird jedes Programmelement einem eigenen Cluster zugeordnet. Anschließend werden iterativ die beiden Cluster vereint deren, über Gleichung 5.21 berechneter, Zusammenhang maximal ist. Bei Mehrdeutigkeiten wird der Clusterkandidat mit dem niedrigsten durchschnittlichen Zusammenhang zu Programmelementen außerhalb des Clusters gewählt. Kann die Mehrdeutigkeit hierdurch nicht aufgelöst werden, wird der erste Clusterkandidat als neues Cluster gewählt. Die Clusteranalyse wird abgebrochen, sobald der durchschnittliche semantische Zusammenhang der Elemente eines neuen Clusters kleiner als der Schwellwert α_{cluster} ist.

Aus der aufgebauten Clusterhierarchie werden in Zeile 13 des Algorithmus 3 alle Cluster extrahiert, die in Anlehnung an die Beschreibung aus Abschnitt 5.4.3 eine zusammenhängende Menge von Programmelementen verkörpern. Hierzu zählen im Rahmen dieser Arbeit alle Cluster der obersten Hierarchieebene, die aus mehr als einem Programmelement bestehen.

Für jedes der gefundenen Cluster aus Zeile 13 wird anschließend ein neuer Knoten vom Typ *intent* im Graph angelegt. Die Knoten repräsentieren die Absichtsknoten aus Abschnitt 5.2.2 und besitzen als Attribut den durchschnittlichen Zusammenhang aller Programmelemente im Cluster. Darüber hinaus werden neue Kanten vom Typ *member* zwischen den Absichtsknoten und den, im korrespondierenden Cluster enthaltenen, Programmelementen in den Graph eingefügt. Als Kantengewicht wird der durchschnittliche Zusammenhang des Programmelements mit allen anderen Programmelementen im Cluster genutzt.

6.5 Parameteroptimierung

In der Implementierung werden an verschiedenen Stellen Schwellwerte und Gewichte definiert, welche eine flexible Konfiguration des Systems erlauben. Das Ziel im Folgenden ist die Optimierung der initialen Parameterbelegungen. Um zu bewerten wie gut eine Parameterbelegung abschneidet, wird die Güte des aufgebauten Absichtsmodells bewertet. Das Absichtsmodell ist das Ergebnis der semantischen Clusteranalyse und liefert eine Unterteilung der Methoden und Typen in semantisch zusammenhängende Gruppen von Programmelementen. Es baut auf den Ergebnissen der durchgeführten Quelltextanalysen auf und eignet sich daher für die Bewertung der Güte des Gesamtsystems.
Für die Optimierung der Parameter wird das Softwareprojekt Dronology[12] herangezogen. Dronology ist eine, in Java geschriebene, Anwendung, welche 59 Typen und 343 Methoden umfasst. Das Projekt bietet sich aus zwei Gründen für die Parameteroptimierung an. Einerseits kann aufgrund der geringen Projektgröße die semantische Quelltextrepräsentation schnell aufgebaut werden, wodurch eine Evaluation vieler unterschiedlicher Parameterbelegungen in kurzer Zeit durchgeführt werden kann. Andererseits beinhaltet das Projekt Informationen über die Zuordnung zwischen den Klassen im Quelltext und den Anforderungen. Auf der Grundlage dieser Zuordnungen wird eine Musterlösung erstellt, die 28 Typen und 61 Methoden auf insgesamt 15 Cluster verteilt.

6.5.1 Evaluationsmetriken

Aufgrund der Vielzahl an Programmelementen, die ein Softwareprojekt typischerweise umfasst, ist der manuelle Aufwand einer Beschreibung aller semantischer Zusammenhänge im Quelltext nicht praktikabel. Infolgedessen decken Musterlösungen in der Regel nur einen Teil der existierenden semantischen Zusammenhänge im Quelltext ab. Um zu bewerten, wie hoch die Übereinstimmung des, vom System aufgebauten, Absichtsmodells und einer Musterlösung ist, wird jedem Cluster der Musterlösung das Cluster mit der größten Übereinstimmung und der minimalen Größe aus dem Absichtsmodell zugeordnet. Jedes Cluster aus dem Absichtsmodell wird hierbei maximal einem Cluster aus der Musterlösung zugeordnet. Cluster aus der Musterlösung, die keinem Cluster aus dem Absichtsmodell zugeordnet werden können, werden einem leeren Cluster zugeordnet. Die Erstellung dieser Paarung verhindert, dass Cluster in der Ergebnismenge, die nicht Teil der Musterlösung sind, die Ergebnisse beeinflussen. Für die Bewertung aller Cluster im Absichtsmodell wäre eine Musterlösung notwendig, die alle semantischen Zusammenhänge im Quelltext enthält. Die Menge der Cluster-Paare, im Folgenden $pairs$ genannt, umfasst Tupel, die aus einem Cluster der Musterlösung ($gold$) und einem Cluster des Absichtsmodells (sys) bestehen.

Auf der Grundlage der Cluster-Paare wird die Cluster-Abdeckung ($CCov$), die Cluster-Präzision ($CPrec$) sowie deren harmonisches Mittel (CHm) berechnet. Gleichung 6.1 zeigt die Formel zur Berechnung der Cluster-Abdeckung. Hierbei wird die Übereinstimmung der Cluster im Absichtsmodell mit den Clustern aus der Musterlösung berechnet. Das Ergebnis liegt zwischen 0 und 1, wobei 1 bedeutet, dass alle Cluster der Musterlösung einem Cluster aus dem Absichtsmodell zugeordnet wurden, welches alle Elemente des Clusters aus der Musterlösung enthält.

$$CCov(pairs) = \frac{\sum_{(gold,sys)\in pairs} |gold \cap sys|}{\sum_{(gold,sys)\in pairs} |gold|} \tag{6.1}$$

Gleichung 6.2 zeigt die Formel zur Berechnung der Cluster-Präzision. Sie betrachtet das Verhältnis aus der Anzahl gemeinsamer Elemente in den Clustern der Cluster-Paare und der Gesamtanzahl der Elemente in den Clustern des Absichtsmodells. Folglich liefert sie

[12] `https://github.com/JaneClelandHuang/Dronology`, zuletzt besucht am 28.12.2019

eine Aussage darüber wie viele zusätzliche Elemente in einem Cluster des Absichtsmodells enthalten sind. Das Ergebnis liegt ebenfalls zwischen 0 und 1, wobei 1 bedeutet, dass die betrachteten Cluster im Absichtsmodell ausschließlich die Elemente der Cluster aus der Musterlösung enthalten.

$$CPrec(pairs) = \frac{\sum_{(gold,sys)\in pairs} |gold \cap sys|}{\sum_{(gold,sys)\in pairs} |sys|} \tag{6.2}$$

Bei der Berechnung der Cluster-Präzision werden alle Elemente im Cluster des Absichtsmodells, welche nicht im korrespondierenden Cluster der Musterlösung enthalten sind, als *falsch* erachtet. Dies ist ein notwendiger Kompromiss der eingegangen wird, um eine automatisierte Bewertung zu erlauben. Im Allgemeinen kann ohne eine manuelle Verifikation der zusätzlichen Elemente nicht entschieden werden, ob diese korrekt oder falsch zugeordnet wurden.

Sowohl die Cluster-Abdeckung als auch die Cluster-Präzision sind gleichermaßen wichtig für Bewertung des Absichtsmodells. Besitzt das Absichtsmodell große Cluster, ist die Cluster-Abdeckung tendenziell höher. Gleichzeitig ist die Cluster-Präzision vermutlich gering, da die Cluster im Absichtsmodell viele zusätzliche Elemente enthalten, die in der Musterlösung nicht Teil der Cluster sind. Umgekehrt führen kleine Cluster im Absichtsmodell mit hoher Wahrscheinlichkeit zu höheren Werten der Cluster-Präzision und niedrigeren Werten bei der Cluster-Abdeckung. Eine Bewertung, die beide zuvor vorgestellten Metriken einschließt, wird in Anlehnung an den F-Metrik [MRS08] durch das harmonische Mittel berechnet. Gleichung 6.3 zeigt die Formel hierzu. Ein Ergebnis von 1 bedeutet, dass sowohl die Cluster-Abdeckung als auch die Cluster-Präzision einen Wert von 1 erzielten.

$$CHm(pairs) = 2 * \frac{CCov(pairs) * CPrec(pairs)}{CCov(pairs) + CPrec(pairs)} \tag{6.3}$$

6.5.2 Vorgehen

Bei der Optimierung werden alle Parameter gemeinsam optimiert. Eine getrennte Optimierung der Parameter der verschiedenen Quelltextanalysen ist aufgrund der gegenseitigen Abhängigkeiten wenig sinnvoll. Als Kennzahl für die Bewertung einer Parameterkonfiguration wird das harmonische Mittel (*CHm*) aus der Cluster-Abdeckung und der Cluster-Präzision über Gleichung 6.3 ermittelt. Als Werkzeug für die Optimierung der Parameterbelegung wird Jenetics[13] eingesetzt. Jenetics evaluiert iterativ verschiedene Parameterkonfigurationen. Eine Parameterbelegung wird hierbei als Vektor dargestellt und im folgenden Individuum genannt. Pro Iteration werden 35 Individuen evaluiert. Neben dem Individuum, welches die initiale Parameterbelegung aus Kapitel 5 repräsentiert, werden alle weiteren Individuen mit zufällig gewählten Parameterwerten von Jenetics erzeugt. In jeder Iteration werden die fünf besten Individuen beibehalten. Zusätzlich werden 20 neue Individuen durch Mutation dieser fünf Individuen erzeugt und 10 weitere Individuen randomisiert generiert. Bei der Mutation kommen zwei Strategien zum Einsatz, welche beide als Eingabe zwei Individuen $\vec{p}_1$ und $\vec{p}_2$ fordern. Bei der ersten Strategie wird ein neues Individuum erzeugt, indem eine Kopie von $\vec{p}_1$ angelegt wird und einer der Parameter durch den Mittelwert aus den entsprechenden Werten in $\vec{p}_1$ und $\vec{p}_2$ ersetzt wird. Die Wahl des ersetzten Parameters erfolgt zufällig. Die zweite Strategie erzeugt ein neues Individuum aus einem Teil der Parameter von $\vec{p}_1$ und einem Teil der Parameter von $\vec{p}_2$. Die Wahl der übernommenen Parameter erfolgt ebenfalls randomisiert. Um sicherzustellen, dass die Parameter gültig sind, werden alle Individuen zu Beginn einer Iteration einem Normierungsschritt unterzogen.

[13] `http://jenetics.io/`, zuletzt besucht am 28.12.2019

Tabelle 6.1: Ergebnisse der initialen und optimierten Parameterbelegung für Dronology.

Parameter	CPrec	CCov	CHm
init	0,38	0,48	0,42
opt_{dron}	0,51	0,57	0,54

Maximal werden 30 Iterationen durchgeführt. Bei drei aufeinander folgenden Iterationen ohne Anstieg des *CHm*-Wertes, wird die Optimierung abgebrochen. Die Beschränkung der Individuen und der Anzahl an Iterationen ist eine Maßnahme, um die Ausführungszeit der Optimierung zu limitieren und kann einfach angepasst werden.
Um eine möglichst gute Parameterbelegung zu finden wird das zuvor beschriebene Vorgehen mehrfach wiederholt, wobei die beste Parameterbelegung jeweils als initiales Individuum der nächsten Optimierung genutzt wird. Die gesamte Optimierung wird beendet, sobald zwei aufeinanderfolgende Optimierungen dieselbe Parameterbelegung als bestes Individuum ausgeben.

6.5.3 Ergebnisse

Tabelle 6.1 fasst die Ergebnisse vor und nach der Optimierung zusammen. Die Werte in der Tabelle wurden durch den Vergleich des aufgebauten Absichtsmodells mit der, zu Beginn von Abschnitt 6.5 beschriebenen, Musterlösung berechnet. Mit opt_{dron} ist die, auf Dronology optimierte, Parameterbelegung gekennzeichnet. Unter Verwendung der initialen Parameterbelegung (*init*) aus Kapitel 5 wurde ein CHm-Wert von 0,42 erzielt. Durch die Parameteroptimierung konnte dieser Wert um 0,12 auf 0,54 angehoben werden. Hierbei verbesserte sich sowohl die Cluster-Abdeckung als auch die Cluster-Präzision.
Die Parameter der einzelnen Agenten vor und nach der Optimierung werden nachfolgend im Detail betrachtet. Der Aufbau der Tabellen folgt hierbei stets demselben Schema. Die erste Spalte enthält den Parameternamen, die zweite Spalte das initiale Gewicht aus Kapitel 5, die dritte Spalte das optimierte Gewicht und die vierte Spalte die Veränderung des initialen Gewichts. Die horizontalen Linien unterteilen die Parameter in Gruppen aus zusammengehörenden Gewichten, die in Summe die Zahl 1 ergeben müssen.

Tabelle 6.2 zeigt die Parameter der syntaxbasierten Quelltextanalyse. Die Abhängigkeitsmerkmale sind in vier Gruppen aufgeteilt. Mit *MM* sind die Gewichte der Abhängigkeitsmerkmale zwischen Methoden gekennzeichnet. Hier verliert vor allem das Merkmal *Überschreiben* an Einfluss. Vermutlich liegt dies daran, dass der Quelltext von Dronology nur wenige Vererbungsbeziehungen enthält. Der *Kontrollfluss* als Merkmal ist nach der Optimierung das wichtigste Merkmal für einen Zusammenhang zwischen Methoden.
Die Abhängigkeitsmerkmale zwischen Methoden und Typen sind mit dem Präfix *MT* gekennzeichnet. In dieser Gruppe verlieren alle Merkmale, mit Ausnahme von *Instanziierung*, *Parameter* und *Kontrollfluss*, an Gewicht. Interessant ist, dass das Merkmal *Umschließender Typ*, welches die Beziehung zwischen einer Methode und dem Typ, der sie definiert, beschreibt, kaum einen Einfluss bei der syntaxbasierten Quelltextanalyse besitzt. Am meisten Gewicht verliert das Merkmal *Rückgabe*. Folglich ist der Rückgabetyp einer Methode kaum relevant für die Suche nach semantischen Zusammenhängen in Dronology.
Mit dem Präfix *TT* sind die Merkmale zwischen Typen markiert. Der größte Einfluss nach der Optimierung wird den Merkmalen *Paketstruktur* und *Explizite Typreferenz* zugesprochen. Auch *Attributtyp*, *Innerer Typ* und *Ausnahme* gewinnen an Bedeutung. Erstaunlich ist, dass die *Typhierarchie* nach der Optimierung nahezu keinen Einfluss besitzt. Allerdings kann dies, genau wie der niedrige Einfluss des Merkmals *Überschreiben* zwischen Methoden, auf die wenigen Vererbungsbeziehungen in Dronology zurückgeführt werden.

Tabelle 6.2: Optimierte Parameter der syntaxbasierten Quelltextanalyse.

Parameter	Initial	Optimiert	Veränderung
MM-Methodenaufruf	0,333	0,242	$-0,091$
MM-Kontrollfluss	0,333	0,675	$+0,342$
MM-Überschreiben	0,333	0,083	$-0,250$
MT-Variablen- oder Typzugriff	0,125	0,078	$-0,047$
MT-Explizite Typreferenz	0,125	0,072	$-0,053$
MT-Instanziierung	0,125	0,235	$+0,110$
MT-Ausnahme	0,125	0,039	$-0,086$
MT-Parameter	0,125	0,244	$+0,119$
MT-Umschließender Typ	0,125	0,042	$-0,083$
MT-Rückgabe	0,125	0,006	$-0,119$
MT-Kontrollfluss	0,125	0,284	$+0,159$
TT-Typhierarchie	0,091	0,029	$-0,062$
TT-Innerer Typ	0,091	0,180	$+0,089$
TT-Attributtyp	0,091	0,130	$+0,039$
TT-Umschließender Typ	0,091	0,005	$-0,086$
TT-Paketstruktur	0,091	0,225	$+0,134$
TT-Variablen- oder Typzugriff	0,091	0,026	$-0,065$
TT-Explizite Typreferenz	0,091	0,201	$+0,110$
TT-Instanziierung	0,091	0,004	$-0,087$
TT-Ausnahme	0,091	0,148	$+0,057$
TT-Parameter	0,091	0,041	$-0,050$
TT-Kontrollfluss	0,091	0,011	$-0,081$
TM-Methodendefinition	0,333	0,164	$-0,169$
TM-Methodenaufruf	0,333	0,622	$+0,289$
TM-Kontrollfluss	0,333	0,214	$-0,119$
Schwellwert α_{sda}	0,0	0,066	$+0,066$

Im Vergleich zum Einfluss des Merkmals *Kontrollfluss* zwischen Methoden bzw. zwischen Methoden und Typen, ist dieses Merkmal zwischen Typen kaum relevant.
Der vorletzte Block in Tabelle 6.2 enthält die Abhängigkeitsmerkmale zwischen Typen und Methoden. Das Merkmal *Methodenaufruf* besitzt nach der Optimierung die größte Bedeutung bei der Suche nach Zusammenhängen zwischen Typen und Methoden.
In der letzten Zeile der Tabelle findet sich der Schwellwert α_{sda}, der festlegt, ab welchem Wert ein Zusammenhang als Kante in die Quelltextrepräsentation eingefügt wird. Der Schwellwert verzeichnet nur einen geringen Anstieg durch die Optimierung. Folglich sind nahezu alle syntaktischen Zusammenhänge bei der Suche nach semantischen Zusammenhängen in Dronology hilfreich.

In Tabelle 6.3 sind die Parameter der lexikalischen Quelltextanalyse dargestellt. Jede der drei Teilanalysen *BA*, *ZKA* und *ZKBA* besitzt vier Parameter. Zwei für die Bestimmung des Einflusses des Zeichenketten- bzw. Schwerpunktvergleichs und zwei weitere, die bei der Berechnung der Schwerpunktvektoren festlegen, welchen Einfluss die Signatur- und Anweisungsebene haben.
Bei den Analysen *ZKA* und *ZKBA* steigt der Einfluss der Zeichenkettenanalyse deutlich an und wird nach der Optimierung stärker gewichtet als die Schwerpunktanalyse. Ein möglicher Grund hierfür ist, dass der Quelltext von Dronology vergleichsweise wenig und kurze Kommentare enthält und somit der Schwerpunktvergleich vermutlich kaum zur Differen-

Tabelle 6.3: Optimierte Parameter der lexikalischen Quelltextanalyse.

Parameter	Initial	Optimiert	Veränderung
BA-Schwerpunkt	0,9	0,704	$-0,196$
BA-Zeichenketten	0,1	0,296	$+0,196$
BA-Signaturebene	0,6	0,594	$-0,006$
BA-Anweisungsebene	0,4	0,406	$+0,006$
ZKA-Schwerpunkt	0,9	0,081	$-0,819$
ZKA-Zeichenketten	0,1	0,919	$+0,819$
ZKA-Signaturebene	0,6	0,770	$+0,170$
ZKA-Anweisungsebene	0,4	0,230	$-0,170$
ZKBA-Schwerpunkt	0,9	0,211	$-0,689$
ZKBA-Zeichenketten	0,1	0,789	$+0,689$
ZKBA-Signaturebene	0,6	0,446	$-0,154$
ZKBA-Anweisungsebene	0,4	0,554	$+0,154$
BA-Gewicht	0,333	0,294	$-0,039$
ZKA-Gewicht	0,333	0,260	$-0,073$
ZKBA-Gewicht	0,333	0,446	$+0,113$
Schwellwert α_{la}	0,0	0,333	$+0,333$

Tabelle 6.4: Optimierte Parameter der semantischen Clusteranalyse.

Parameter	Initial	Optimiert	
LA-Gewicht	0,6	0,389	$-0,211$
SDA-Gewicht	0,4	0,611	$+0,211$
Schwellwert α_{semantic}	0.0	0.384	$+0,384$
Schwellwert α_{cluster}	0.6	0.604	$+0,004$

zierung von Programmelementen beiträgt.
Der Einfluss der Signatur- bzw. Anweisungsebene wird bei der *BA*- und *ZKA*-Analyse kaum verändert. Bei der *ZKBA*-Analyse hingegen wird der Einfluss umgekehrt und der Anweisungsebene mit 0,554 eine etwas größere Bedeutung als der Signaturebene zugesprochen.
Neben den internen Gewichten der Teilanalysen existieren drei Gewichte, welche den Einfluss der einzelnen Analysen bestimmen. Sie sind im vorletzten Block von Tabelle 6.3 angegeben. Die *BA* und *ZKA* Analysen verlieren leicht an Gewicht, wohingegen die *ZKBA*-Analyse etwas an Bedeutung gewinnt.
Im Gegensatz zum Schwellwert der syntaxbasierten Analyse steigt der Schwellwert α_{la} für die Ergebnisse der lexikalischen Quelltextanalyse durch die Optimierung deutlich auf 0,333. Infolgedessen besitzen kleine lexikalische Zusammenhänge keine Relevanz bei der Suche nach semantischen Zusammenhängen in Dronology.

In Tabelle 6.4 sind die Parameter der semantischen Clusteranalyse aufgelistet. Der Einfluss der lexikalischen und strukturellen Zusammenhänge wird von den beiden Parametern *LA-Gewicht* und *SDA-Gewicht* bestimmt. Entgegen der Erwartung steigt der Einfluss der syntaxbasierten Quelltextanalyse auf 0,611 an, wodurch ihr mehr Bedeutung bei der Suche nach semantischen Zusammenhängen zugesprochen wird als der lexikalischen Quelltextanalyse.
Durch den Anstieg von α_{semantic}, welcher den Schwellwert für die Berücksichtigung von

semantischen Zusammenhängen festlegt, wird der Fokus im Vergleich zur initialen Parameterbelegung auf stärkere Zusammenhänge gelegt. Der Schwellwert für den Abbruch der semantischen Clusteranalyse bleibt nahezu unverändert.

Insgesamt führt die Parameteroptimierung zu einem Anstieg des *CHm*-Wertes um 0,12. Sowohl die Cluster-Abdeckung als auch die Cluster-Präzision konnten durch die Optimierung auf einen Wert größer als 0,5 angehoben werden. Viele der zuvor beschriebenen Parameteränderungen erscheinen sinnvoll, einige sind allerdings auf die spezielle Charakteristik von Dronology zurückzuführen, wodurch ihre Übertragbarkeit auf andere Softwareprojekte offen bleibt. Eine detaillierte Analyse der Übertragbarkeit der optimierten Parameterbelegung ist Teil der Evaluation in Kapitel 7.

7 Evaluation

Bei der Evaluation der semantischen Quelltextrepräsentation werden die vom System gefundenen semantischen Zusammenhänge bewertet. Darüber hinaus wird der Einfluss verschiedener Systemparameter auf die semantische Repräsentation und die Übertragbarkeit von Parameterbelegungen zwischen verschiedenen Softwareprojekten untersucht. Die Grundlage der Evaluation bildet das von der semantischen Clusteranalyse aufgebaute Absichtsmodell (Siehe Abschnitt 5.4.3). Es vereint die Ergebnisse der syntaxbasierten und lexikalischen Quelltextanalyse und eignet sich daher für die Bewertung des Gesamtsystems.
Im Gegensatz zur Mehrheit der bestehenden Arbeiten aus dem Gebiet der Anforderungsrückverfolgung, beschränkt sich die vorliegende Arbeit nicht auf die Typen oder Methoden im Quelltext, sondern sucht nach Zusammenhängen auf verschiedenen Granularitätsebenen. Hierdurch können Zusammenhänge feingranularer dargestellt werden. Im Hinblick auf die Evaluation stellt dies allerdings eine Herausforderung dar, da Evaluationsdatensätze, wie durch das *Center of Excellence for Software & System Traceability*[1] (CoEST) angeboten, auf die Typebene eines Softwareprojekts beschränkt sind. Um dennoch eine Bewertung der semantischen Quelltextrepräsentation durch einen offiziellen Datensatz zur Rückverfolgbarkeit von Typen zu Anforderungen zu erlauben, umfasst die Evaluation neben der Bewertung des Absichtsmodells für Typen und Methoden eine Bewertung des Absichtsmodells nach dessen Projektion auf die Typebene. In Abschnitt 7.1 werden die evaluierten Softwareprojekte beschrieben und die jeweiligen Musterlösungen charakterisiert. Im Anschluss werden die Ergebnisse in Abschnitt 7.2 vorgestellt und analysiert.

7.1 Datengrundlage

Für die Evaluation des Absichtsmodells werden Musterlösungen benötigt, die eine Unterteilung der Programmelemente im Quelltext in Gruppen aus semantisch zusammenhängenden Programmelementen anbieten. Aufgrund der hohen Anzahl an Typen und Methoden ist eine manuelle Erstellung von Musterlösungen, welche alle Programmelemente im Quelltext abdecken, nicht praktikabel. Die in der Evaluation verwendeten Musterlösungen liefern daher keine vollständigen Beschreibungen der semantischen Zusammenhänge im Quelltext.

Insgesamt werden zwei Arten von Musterlösungen genutzt. Die erste umfasst ausschließlich Typen und die zweite umfasst Typen und Methoden. Als Grundlage dienen die beiden Soft-

[1] `http://coest.org/`, zuletzt besucht am 12.01.2020

Tabelle 7.1: Kennzahlen der evaluierten Softwareprojekte.

Projekt	Typen	Methoden	Quelltextzeilen	Kommentarzeilen
iTrust	226	1579	14.573	6.338
Easymock	87	753	4.952	4.998

wareprojekte iTrust[2] und Easymock[3]. Bei iTrust handelt es sich um eine Webanwendung zur Verwaltung von medizinischen Gesundheitsdaten. Easymock ist eine Java-Bibliothek, die das Erstellen von Attrappen (engl. mock) in Java-Testfällen ermöglicht. Das Softwareprojekt iTrust wurde ausgewählt, da CoEST Rückverfolgbarkeitsinformationen für dieses Projekt auf der Typebene anbietet, welche bei der Erstellung der Musterlösung genutzt werden. Easymock wurde ausgewählt, da es sich im Gegensatz zu iTrust um ein Softwareprojekt handelt, an dem weiterhin aktiv entwickelt wird. Darüber hinaus unterscheidet sich die Charakteristik des Quelltextes von Easymock stark von iTrust, was durch die Kennzahlen in Tabelle 7.1 deutlich wird. Der Quelltext von iTrust ist umfangreicher und umfasst wesentlich mehr Typen und Methoden als der Quelltext von Easymock. Diese Unterschiede können darauf zurückgeführt werden, dass es sich bei iTrust im Gegensatz zu Easymock um eine Webanwendung mit vielseitigem Funktionsumfang handelt. Bei der Betrachtung der Kommentarzeilen zeigt sich, dass Easymock fast doppelt so viele Kommentarzeilen pro Quelltextzeile enthält wie iTrust. Der hohe Kommentaranteil im Quelltext folgt vermutlich aus der Tatsache, dass es sich bei Easymock um eine Programmbibliothek handelt, welche von anderen Entwicklern verstanden werden muss, um benutzt zu werden.

Wie bereits zu Beginn dieses Kapitels erwähnt, existieren keine offiziellen Musterlösungen für die semantischen Zusammenhänge zwischen den Typen und Methoden bestehender Softwareprojekte. Für die Evaluation des Absichtsmodells aus Typen und Methoden wurden daher eigene Musterlösungen erstellt. Hierzu wurde die semantische Quelltextrepräsentation einmal für iTrust bzw. Easymock aufgebaut und die Ergebnisse der semantischen Clusteranalyse manuell vom Autor dieser Arbeit gemeinsam mit einem Doktoranden des KIT in eine Musterlösung überführt. Dieses Vorgehen erlaubt die Erstellung von Musterlösungen mit akzeptablem Umfang und vertretbarem Zeitaufwand, da das Löschen, Vereinen und Manipulieren von bestehenden Clustern, im Vergleich zur Suche nach neuen Clustern in der Menge von z.B. 1805 Programmelementen in iTrust, den manuellen Aufwand erheblich verringert. Die Qualität der Musterlösung wird hiervon nicht beeinträchtigt, da alle Cluster in der Musterlösung manuell verifiziert wurden. Bei der Erstellung der Musterlösungen wurden folgende Operationen durchgeführt:

- Entfernen von fehlerhaften Clustern
- Entfernen von fehlerhaften Elementen aus einem Cluster
- Hinzufügen fehlender Elemente zu einem Cluster
- Vereinigen zusammengehörender Cluster
- Auftrennen von nicht zusammenhängenden Clustern

Beim Aufbau der Musterlösung wurde zudem der Quelltext und die zur Verfügung stehende Quelltextdokumentation von iTrust bzw. Easymock herangezogen. Der erste Block in Tabelle 7.2 zeigt die Kennzahlen der entstandenen Musterlösungen. In der Musterlösung von iTrust umfassen die Cluster durchschnittlich sieben Programmelemente. Die Cluster in der Musterlösung von Easymock enthalten durchschnittlich sechs Programmelemente.

[2] `https://sourceforge.net/projects/itrust/`, zuletzt besucht am 12.01.2020
[3] `https://github.com/easymock/easymock`, zuletzt besucht am 12.01.2020

Tabelle 7.2: Übersicht der Musterlösungen.

Musterlösung	Projekt	Typen	Methoden	Cluster
Typen und Methoden	iTrust	172	104	40
	Easymock	46	211	43
Typen	iTrust	201	-	59
	Easymock	58	-	10

Darüber hinaus werden zwei weitere Musterlösungen bereitgestellt, die eine Evaluation des Absichtsmodells auf Typebene erlauben. Für iTrust wird der, durch CoEST angebotene, Datensatz zur Rückverfolgbarkeit von Quelltext zu Anforderungen als Musterlösung verwendet. Er umfasst Typzuordnungen zu insgesamt 103 Anforderungen. Einige der Anforderungen sind hierbei derselben Menge an Typen zugeordnet. Um zu verhindern, dass sich diese Duplikate auf die Ergebnisse der Evaluation auswirken, werden sie vorab entfernt. Darüber hinaus werden Zuordnungen, die lediglich aus einem Typ bestehen, entfernt. Abgesehen von diesen beiden Vorverarbeitungen bleibt der Datensatz unverändert. Die Kennzahlen der Musterlösung für die Typebene auf iTrust sind im zweiten Block von Tabelle 7.2 zusammengefasst. Insgesamt umfasst die Musterlösung 201 Typen, die auf 59 Cluster aufgeteilt werden. Im Vergleich zu der durchschnittlichen Clustergröße von sieben in der Musterlösung der Typen und Methoden für iTrust, umfasst ein Cluster in der Musterlösung der Typen durchschnittlich zwischen drei und vier Programmelemente.
Für Easymock existiert keine offizielle Musterlösung der semantischen Zusammenhänge zwischen den Typen im Quelltext. Sie wird manuell vom Autor dieser Arbeit, unter Zuhilfenahme des Quelltextes und der Dokumentation, erstellt. Die Musterlösung umfasst insgesamt 58 Typen, die auf 10 Cluster aufgeteilt werden.

7.2 Ergebnisse

Für die Bewertung des, vom System aufgebauten, Absichtsmodells werden die Metriken aus Abschnitt 6.5.1 verwendet. Mit *CPrec* wird die Cluster-Präzision, mit *CCov* die Cluster-Abdeckung und mit *CHm* das harmonische Mittel der beiden Kennzahlen beschrieben.

Zunächst wird das Absichtsmodell auf der Typ- und Methodenebene evaluiert. Hierzu wird das Ergebnis der semantischen Clusteranalyse mit den, in Abschnitt 7.1 beschriebenen, Musterlösungen für Typen und Methoden verglichen. Tabelle 7.3 fasst die Ergebnisse zusammen. Die erste Zeile enthält die Bewertung des Absichtsmodells unter Anwendung der initialen Parameterbelegung (*init*) aus Kapitel 5. Alle weiteren Zeilen zeigen die Ergebnisse für verschiedene Parameterbelegungen, wobei jeweils die Veränderung bezogen auf den Wert in der ersten Zeile angegeben ist. In jeder Spalte sind die besten Werte hervorgehoben.
In iTrust kann mit *init* bereits ein akzeptabler *CHm*-Wert von 0,52 erzielt werden. Im Gegensatz dazu ist der *CHm*-Wert für Easymock vergleichsweise niedrig, was eine Folge der geringen Cluster-Präzision von 0,33 ist. In beiden Projekten kann mit der Parameterbelegung *init* eine Cluster-Abdeckung von mind. 0,6 erreicht werden. Bei Easymock führt überdies keine andere Parameterbelegung zu einer höheren Cluster-Abdeckung.
In der zweiten Zeile wurde die optimierte Parameterbelegung (opt_{dron}) aus Abschnitt 6.5.3 genutzt. Im Vergleich zu *init* zeigt sich für iTrust ein deutlicher Anstieg in allen Werten. Vor allem die Cluster-Abdeckung erreicht mit 0,91 einen sehr guten Wert. Bei Easymock führt die Anwendung von opt_{dron} hingegen zu kaum einer Veränderung. Da es sich bei Dronology ähnlich wie bei iTrust um eine Anwendung und nicht wie bei Easymock um

Tabelle 7.3: Ergebnisse für das Absichtsmodell aus Typen und Methoden.

Parameter	iTrust			Easymock		
	CPrec	CCov	CHm	CPrec	CCov	CHm
init	0,44	0,66	0,53	0,33	**0,60**	0,43
opt_{dron}	$+0,11$	**+0,25**	$+0,16$	$+0,01$	$-0,06$	$-0,01$
opt_{iTrust}	**+0,17**	**+0,25**	**+0,20**	$+0,02$	$-0,14$	$-0,03$
opt_{easy}	$+0,16$	$+0,07$	$+0,13$	**+0,18**	$-0,09$	**+0,08**

eine Bibliothek handelt, können die Ergebnisse von opt_{dron} auf eine Übertragbarkeit der Parameterbelegung zwischen Softwareprojekten mit ähnlicher Charakteristik hindeuten.
Die letzten beiden Zeilen in Tabelle 7.3 zeigen die Ergebnisse, nachdem die Parameter, wie in Abschnitt 6.5.2 beschrieben, optimiert wurden. Die Parameterbelegung opt_{iTrust} wurde auf der Musterlösung von iTrust und die Parameterbelegung opt_{easy} auf der Musterlösung von Easymock optimiert. Beide Parameterbelegungen sind in Abschnitt A des Anhangs beschrieben. Mit opt_{iTrust} wird auf iTrust ein *CPrec*-Wert von 0,61 und ein *CHm*-Wert von 0,71 erzielt. Im Vergleich zu opt_{dron} können die Ergebnisse mit opt_{iTrust} folglich weiter verbessert werden. Eine weniger starke Verbesserung der Ergebnisse zeigt sich bei der Anwendung von opt_{easy} auf Easymock. Dennoch liefert sie den höchsten *CPrec*- und *CHm*-Wert für Easymock.
Bei näherer Betrachtung der verschiedenen optimierten Parameterbelegungen (Siehe Abschnitt A) fällt auf, dass die Parameter der semantischen Clusteranalyse in opt_{dron}, opt_{iTrust} und opt_{easy} nahezu identisch sind. Alle drei gewichten die Ergebnisse der lexikalischen Analyse mit 0,331 bis 0,389 schwächer als die Ergebnisse der syntaxbasierten Quelltextanalyse. Darüber hinaus besitzen opt_{dron} und opt_{iTrust} dieselbe Parameterbelegung für die lexikalische Analyse. In opt_{easy} werden die Parameter der lexikalischen Analyse hingegen mit anderen Werten belegt. Vor allem die Analyse der Bezeichner im Quelltext nimmt hierbei an Gewicht zu und wird mit 0,602 in opt_{easy} deutlich höher bewertet als ihr Wert von 0,294 in opt_{dron} und opt_{iTrust}. Die *ZKBA*-Analyse hat in opt_{easy} mit einem Gewicht von 0,01 kaum mehr einen Einfluss auf das Ergebnis der lexikalischen Quelltextanalyse.
Im Hinblick auf die Parameter der syntaxbasierten Quelltextanalyse unterscheiden sich alle drei Parameterbelegungen voneinander. Die individuelle Gewichtung der verschiedenen Abhängigkeitsmerkmale im Quelltext ist vermutlich auf die individuelle Quelltextstruktur, die jedes Softwareprojekt mit sich bringt, zurückzuführen. Darüber hinaus sticht der hohe Schwellwert von 0,528 für die Berücksichtigung eines syntaktischen Zusammenhangs in opt_{easy} hervor. In opt_{dron} bzw. opt_{iTrust} ist der Schwellwert mit 0,066 erheblich niedriger. Infolgedessen scheint es in Easymock viele syntaktische Zusammenhänge zu geben, die keine Aussagekraft bei der Suche nach semantischen Zusammenhängen besitzen. Möglicherweise liegt dies daran, dass Easymock eine Bibliothek mit sehr spezifischen Funktionalitäten und vielen Abhängigkeiten zwischen den einzelnen Programmelementen ist.
Für iTrust liegt *CPrec* durchschnittlich bei 0,55 und *CCov* durchschnittlich bei 0,8. Auch für Easymock zeigt sich ein deutlicher Unterschied zwischen dem durchschnittlichen *CPrec*-Wert von 0,38 und dem durchschnittlichen *CCov*-Wert von 0,53. In beiden Projekten ist die durchschnittliche Cluster-Präzision niedriger als die durchschnittliche Cluster-Abdeckung. Ein Grund für die vergleichsweise niedrigen *CPrec*-Werte ist, dass in den, vom System gefundenen Clustern, häufig mehr Elemente enthalten sind, als in dem korrespondierenden Cluster der Musterlösung. Bei der Berechnung von *CPrec* (Siehe Gleichung 6.2) werden alle Elemente eines Cluster, die nicht im korrespondierenden Cluster in der Musterlösung enthalten sind, als Fehler bewertet. In einigen Fällen sind die vom System platzierten zusätzlichen Elemente in einem Cluster jedoch korrekt, weshalb die niedrigen *CPrec*-Werte

Tabelle 7.4: Einfluss verschiedener Konfigurationen auf das Absichtsmodell von iTrust.

Konfiguration	CPrec	CCov	CHm
opt_{iTrust}	0,61	0,91	0,73
Ohne Syntaxinformationen	$-0,50$	$-0,46$	$-0,55$
Ohne lexikalische Informationen	$-0,08$	$-0,10$	$-0,09$
Ohne indirekte Abhängigkeiten	$-0,53$	$-0,91$	$-0,73$
Ohne direkte Abhängigkeiten	$\pm 0,00$	$\pm 0,00$	$\pm 0,00$
Ohne Zeichenkettenanalyse	$-0,03$	$\pm 0,00$	$-0,02$

zum Teil auf den beschränkten Umfang der Musterlösungen zurückzuführen sind.
Grundsätzlich zeigen die Ergebnisse in Tabelle 7.3 bessere Werte für iTrust als für Easymock. Der durchschnittliche *CHm*-Wert für iTrust liegt bei 0,65. Für Easymock beträgt er hingegen lediglich 0,44. Die vergleichsweise schwächeren Ergebnisse von Easymock können eine Folge dessen spezifischer Quelltextcharakteristik sein. Easymock umfasst im Gegensatz zu iTrust erheblich weniger Programmelemente, die jedoch viele syntaktische Abhängigkeiten zueinander aufweisen. Darüber hinaus weist die verwendete natürliche Sprache in den Bezeichnern und Kommentaren der verschiedenen Typen und Methoden in Easymock zum Teil hohe Überschneidungen und Ähnlichkeiten auf. Beide Aspekte führen dazu, dass die Differenzierung zwischen semantisch zusammenhängenden und nicht zusammenhängenden Programmelementen in den Quelltextanalysen erschwert wird.
Zusammenfassend lässt sich feststellen, dass durch die Optimierung der Parameter auf der eigenen Musterlösung die besten *CPrec*- und *CHm*-Werte erzielt werden. Gleichzeitig deuten die Ergebnisse in Tabelle 7.3 an, dass die positiven Effekte einer optimierten Parameterbelegung wie opt_{dron} auf andere Projekte wie iTrust übertragen werden können.

Um den Einfluss der verschiedenen Quelltextanalysen bzw. deren Konfiguration auf die Güte des Absichtsmodells zu untersuchen, werden in Tabelle 7.4 die Ergebnisse verschiedener Systemkonfigurationen mit dem besten Ergebnis aus Tabelle 7.3 verglichen. Die erste Zeile zeigt die Ergebnisse von opt_{iTrust} für das Absichtsmodell aus Typen und Methoden in iTrust aus Tabelle 7.3. Alle weiteren Zeilen zeigen die Veränderungen der Werte nach verschiedenen Änderungen an der Systemkonfiguration.
Für die Ergebnisse der zweiten Zeile wurde das Gewicht der lexikalischen Zusammenhänge bei der Berechnung der semantischen Zusammenhänge in Gleichung 5.20 auf 1 und das Gewicht der syntaktischen Zusammenhänge auf 0 gesetzt. Analog dazu wurden für die Ergebnisse der dritten Zeile das Gewicht der syntaktischen Zusammenhänge auf 1 gesetzt und die lexikalischen Zusammenhänge mit 0 bewertet. Aus den Ergebnissen der zweiten und dritten Zeile geht hervor, dass die syntaxbasierte Quelltextanalyse für das Absichtsmodell von iTrust wichtiger ist, als die lexikalische Analyse. Allerdings verringern sich alle Werte, wenn eine der beiden Quelltextanalysen ausgeschlossen wird. Infolgedessen ist der Ansatz zur Kombination verschiedener Quelltextanalysen sinnvoll.
Den stärksten Einfluss auf das Ergebnis von opt_{iTrust} besitzen die indirekten Abhängigkeiten der syntaxbasierten Quelltextanalyse. Die vierte Zeile zeigt die Ergebnisse, wenn bei der Berechnung der syntaxbasierten Zusammenhänge in Gleichung 5.13 ausschließlich die direkten Abhängigkeiten berücksichtigt werden. Auffällig hierbei ist, dass *CCov* und *CHm* in dieser Konfiguration auf 0 abfallen. Der geringe Einfluss der direkten Abhängigkeiten wird von den Werten in der fünften Zeile bestätigt. Ein Ausschluss der direkten Abhängigkeiten hat keine Auswirkungen auf das Ergebnis.
Die letzte Zeile in Tabelle 7.4 zeigt die Ergebnisse, wenn bei der lexikalischen Analyse ausschließlich Schwerpunktvergleiche genutzt werden und auf die Zeichenkettenanalyse

Tabelle 7.5: Ergebnisse des Absichtsmodells von iTrust nach der Optimierung für verschiedene Konfigurationen.

Konfiguration	CPrec	CCov	CHm
opt_{iTrust}	0,61	0,91	0,73
Opt. ohne Syntaxinformationen	$-0,05$	$-0,45$	$-0,22$
Opt. ohne lexikalische Informationen	$\pm 0,00$	$-0,12$	$-0,04$
Opt. ohne indirekte Abhängigkeiten	$-0,02$	$-0,45$	$-0,21$
Opt. ohne direkte Abhängigkeiten	$\pm 0,00$	$\pm 0,00$	$\pm 0,00$
Opt. ohne Zeichenkettenanalyse	$-0,03$	$\pm 0,00$	$-0,02$

verzichtet wird. Hierzu werden die Parameter für die Gewichtung der Schwerpunktanalyse in den Teilanalysen *BA*, *ZKA* und *ZKBA* der lexikalischen Analyse auf 1 und die Gewichte der Zeichenkettenanalyse auf 0 gesetzt. Auf das Ergebnis des Absichtsmodells hat der Ausschluss der Zeichenkettenanalysen kaum Auswirkungen. Lediglich die Cluster-Präzision und der *CHm*-Wert verringern sich geringfügig. Da die Zeichenkettenanalyse ausschließlich unbekannte Wörter behandelt, kann der schwache Einfluss auf eine geringe Anzahl von unbekannten Wörtern im Quelltext von iTrust hindeuten.

In der vorangegangenen Beschreibung wurden die Auswirkungen unterschiedlicher Systemkonfigurationen auf die Ergebnisse der Parameterbelegung opt_{iTrust} untersucht. Hierbei wurden nur die für die jeweilige Betrachtung notwendigen Änderungen an der Parameterbelegung vorgenommen und die restlichen Parameter unverändert gelassen. Welche Ergebnisse sich mit den verschiedenen Konfigurationen erzielen lassen, wenn die restlichen Parameter der Parameterbelegung nach der jeweiligen manuellen Konfiguration optimiert werden, zeigt Tabelle 7.5. Als Vergleichswert dient wie bereits zuvor das Ergebnis von opt_{iTrust} für das Absichtsmodell von iTrust aus Tabelle 7.3. Alle weiteren Zeilen zeigen die maximal erreichten Werte für verschiedene Systemkonfigurationen.
Der Rückgang der *CHm*-Werte bei Ausschluss der syntaxbasierten oder lexikalischen Quelltextanalyse konnte durch die Optimierung halbiert werden. Auffällig ist, dass vor allem die Cluster-Präzision von der Optimierung profitiert und trotz der Einschränkung auf eine der beiden implementierten Quelltextanalysen nach der Optimierung eine ähnlich gute Cluster-Präzision wie mit opt_{iTrust} erreicht werden kann. Auf die Cluster-Abdeckung hat die Parameteroptimierung bei der Beschränkung auf eine Quelltextanalyse kaum einen Einfluss. Ohne die Berücksichtigung von syntaktischen Zusammenhängen konnte ein maximaler *CCov*-Wert von 0,46 erreicht werden. Beim Ausschluss von lexikalischen Zusammenhängen konnte *CCov* maximal 0,79 erreichen. Beide Werte liegen deutlich unterhalb des *CCov*-Wert von 0,91, der mit opt_{iTrust} erreicht wird. Ein Grund hierfür ist, dass bei der Kombination der Quelltextanalysen semantische Zusammenhänge, die lediglich von einer Quelltextanalyse entdeckt wurden, schwächer eingestuft werden, als Zusammenhänge, die von beiden Quelltextanalysen erkannt wurden. Die berechneten semantischen Zusammenhänge haben vermutlich eine höhere Aussagekraft, da deren Bewertung verschiedene Informationsquellen berücksichtigt. Die Ergebnisse in Tabelle 7.5 sprechen folglich dafür, dass die Quelltextanalysen komplementäre Informationen über die semantischen Zusammenhänge im Quelltext liefern. Hierdurch kann eine Cluster-Abdeckung erreicht werden, welche bei einer Beschränkung auf eine der Quelltextanalysen nicht möglich ist.
Wohingegen der Ausschluss von indirekten Abhängigkeiten in Tabelle 7.4 zu einer Verringerung des *CHm*-Wertes von 0,73 geführt hat, sinkt der *CHm*-Wert nach der Parameteroptimierung lediglich um 0,21. Trotz des Ausschlusses der indirekten Abhängigkeiten bei der syntaxbasierten Quelltextanalyse kann ein akzeptabler *CHm*-Wert von 0,52 erzielt

Tabelle 7.6: Einfluss verschiedener Worteinbettungsmodelle auf das Absichtsmodell von iTrust.

Modell	CPrec	CCov	CHm
Wikipedia	0,61	0,91	0,73
StackOverflow	$-0,02$	$+0,01$	$-0,01$
GitHub	$-0,03$	$-0,07$	$-0,04$

werden. Folglich besitzt die direkte Abhängigkeit zweier Programmelemente durchaus eine Aussagekraft bei der Bewertung ihres semantischen Zusammenhangs. Die Tatsache, dass der Ausschluss der direkten Abhängigkeiten in der fünften Zeile keinen Einfluss auf das Ergebnis hat, kann darauf hindeuten, dass bei der Kombination der beiden Merkmale in Gleichung 5.13 die direkten Abhängigkeiten vernachlässigt werden. Eine Erklärung hierfür ist, dass die indirekten Abhängigkeiten im Durchschnitt höher ausfallen als die direkten Abhängigkeiten und bei deren Kombination über das Maximum die direkten Abhängigkeiten daher kaum berücksichtigt werden. Möglicherweise können die Ergebnisse weiter verbessert werden, wenn die direkte Abhängigkeit zwischen zwei Programmelementen ebenfalls in den Vektorvergleich der syntaxbasierten Quelltextanalyse einfließt oder anstelle des Maximums ein gewichteter Mittelwert aus der direkten und indirekten Abhängigkeit berechnet wird.
Für die Konfigurationen in den letzten beiden Zeilen führte die Parameteroptimierung zu keiner Verbesserung. Insgesamt zeigt Tabelle 7.5, dass die Optimierung der Parameter die negativen Auswirkungen der verschiedenen Konfigurationen eingrenzen kann. Mit Ausnahme der Konfiguration aus Zeile fünf wirken sich alle Änderungen negativ auf die Ergebnisse aus. Dies spricht dafür, dass die verschiedenen Systembestandteile sich gegenseitig ergänzen und deren Kombination daher sinnvoll ist.

Eine zentrale Rolle bei der lexikalischen Quelltextanalyse spielen die Schwerpunktvergleiche. In Abschnitt 5.3.3.2 wurden die Auswirkungen verschiedener Trainingskorpora auf das Worteinbettungsmodell analysiert. Das, in dieser Arbeit standardmäßig verwendete, Worteinbettungsmodell wurde auf den Texten aus Wikipedia trainiert. Welche Auswirkungen ein Worteinbettungsmodell, welches auf Texten aus StackOverflow oder GitHub trainiert wurde, auf das Absichtsmodell hat, ist in Tabelle 7.6 zusammengefasst.
Die erste Zeile zeigt die Ergebnisse der Parameterbelegung opt_{iTrust} auf iTrust aus Tabelle 7.3. Alle weiteren Ergebnisse wurden ebenfalls mit dieser Parameterbelegung erzeugt. Alle Worteinbettungsmodelle liefern akzeptable Ergebnisse. Die Ergebnisse bei einem Training auf StackOverflow sind nahezu identisch mit den Ergebnissen aus der ersten Zeile. Beim Training auf GitHub zeigt sich hingegen ein leichter Rückgang aller berechneten Kennzahlen. Insgesamt zeigt Tabelle 7.6, dass ein Training auf allgemeinen Texten, wie in Wikipedia, tendenziell am besten funktioniert. Mit der Spezifität des Trainingskorpus sinkt die Güte des Absichtsmodells.

Ergänzend zu der Evaluation des Absichtsmodells, bestehend aus Typen und Methoden, wird eine Bewertung der Ergebnisse durchgeführt, bei der lediglich eine Unterteilung der Programmelemente auf Typebene betrachtet wird. Hierzu wird das, vom System generierte, Absichtsmodell auf die Typebene projiziert. Alle Methoden werden durch den Typ, der sie definiert, ersetzt. Cluster, die nach der Projektion nur ein Element enthalten, werden entfernt. Darüber hinaus bleiben die Cluster unverändert. Als Musterlösung dienen die, in Abschnitt 7.1 beschriebenen, Unterteilungen auf Typebene. Die Ergebnisse sind in Tabelle 7.7 zusammengefasst. Die erste Zeile zeigt die Ergebnisse unter Anwendung der initialen Parameterbelegung aus Kapitel 5. Alle weiteren Zeilen zeigen die Veränderung der Werte

Tabelle 7.7: Ergebnisse für das Absichtsmodell auf Typebene.

Parameter	iTrust			Easymock		
	CPrec	CCov	CHm	CPrec	CCov	CHm
init	0,23	0,27	0,25	0,42	0,53	0,47
opt_{dron}	$\pm0,00$	$+0,09$	$+0,03$	$+0,10$	**+0,26**	$+0,16$
$type_opt_{iTrust}$	**+0,22**	**+0,20**	**+0,21**	$+0,35$	$-0,13$	$+0,06$
$type_opt_{easy}$	$+0,09$	$+0,17$	$+0,12$	**+0,48**	$+0,07$	**+0,25**

bezogen auf die Ergebnisse in der ersten Zeile. Die jeweils besten Werte pro Spalte sind hervorgehoben.

Im Vergleich zu den Ergebnissen aus Tabelle 7.3 fallen die Ergebnisse der initialen Parameterbelegung für iTrust niedriger und für Easymock höher aus. Durch opt_{dron} zeigt sich darüber hinaus eine deutliche Verbesserung der Ergebnisse für Easymock. Die Cluster-Abdeckung erreicht unter dieser Parameterbelegung bereits einen sehr guten Wert von 0,79. Für iTrust liefert opt_{dron} kaum eine Verbesserung der Ergebnisse.

Durch eine individuelle Optimierung der Parameter auf der jeweiligen Musterlösung wird ein Anstieg des *CHm*-Wertes auf bis zu 0,46 bei iTrust bzw. bis zu 0,72 bei Easymock erzielt. Die Parameterbelegung $type_opt_{iTrust}$ wurde auf der Musterlösung für iTrust und die Parameterbelegung $type_opt_{easy}$ auf der Musterlösung für Easymock optimiert. Die konkreten Parameterwerte beider Belegungen finden sich in Abschnitt A des Anhangs. Bei iTrust führen die optimierten Parameterbelegungen vor allem zu einem Anstieg der Cluster-Abdeckung. Bei Easymock hingegen bewirken sie in erster Linie ein Anstieg der Cluster-Präzision. Darüber hinaus fällt auf, dass $type_opt_{easy}$ auch auf iTrust zu akzeptablen Ergebnissen führt. Umgekehrt kann mit $type_opt_{iTrust}$ nur ein vergleichsweise mittlerer *CCov* und *CHm* Wert erzielt werden..

Die optimierten Parameterbelegungen (Siehe Abschnitt A) zeigen einige Unterschiede. Bezogen auf die Parameter der semantischen Clusteranalyse besitzen opt_{dron} und $type_opt_{easy}$ ähnliche Parameterbelegungen. Die Ergebnisse der lexikalischen Quelltextanalyse werden mit 0,384 bzw. 0,331 ähnlich stark gewichtet. In $type_opt_{iTrust}$ erhalten die Ergebnisse der lexikalischen Quelltextanalyse mit 0,768 ein deutlich höheres Gewicht und besitzen damit sogar mehr Einfluss auf die gefundenen semantischen Zusammenhänge als die Ergebnisse der syntaxbasierten Quelltextanalyse. Auch der Schwellwert für die Berücksichtigung eines semantischen Zusammenhangs ist in $type_opt_{iTrust}$ mit 0,635 erheblich höher als der Schwellwert von 0,38 bei opt_{dron} und $type_opt_{easy}$. Im Hinblick auf die Parameter der lexikalischen Quelltextanalyse gewichtet $type_opt_{iTrust}$ die Analysen *ZKA* und *ZKBA* deutlich stärker als opt_{dron} und $type_opt_{easy}$. Folglich besitzen Kommentare bei der Suche nach semantischen Zusammenhängen in iTrust eine große Relevanz. Bei den Parametern der syntaxbasierten Quelltextanalyse zeigt sich wie bereits bei opt_{iTrust} und opt_{easy}, dass die Gewichtung der Abhängigkeitsmerkmale projektspezifisch ist. Allerdings gibt es eine Gemeinsamkeit von $type_opt_{iTrust}$ und $type_opt_{easy}$. Im Vergleich zu 0,066 bei opt_{dron} steigt der Schwellwert für die Berücksichtigung syntaktischer Zusammenhänge auf 0,503 bzw. 0,528 an.

Insgesamt zeigt Tabelle 7.7, dass auch für das Absichtsmodell auf Typebene gute Ergebnisse erzielt werden können. Im Vergleich zu den Ergebnissen aus Tabelle 7.3 sind die Ergebnisse für Easymock auf der Typebene besser als für iTrust. Ein möglicher Grund hierfür liegt in der Musterlösung von iTrust. Sie stammt, wie in Abschnitt 7.1 beschrieben, von CoEST. Die Cluster der Musterlösung beinhalten durchschnittlich 3,4 Typen. Bei einem manuellen Vergleich der Musterlösung von iTrust mit der Systemausgabe zeigte sich, dass viele der vom System gefundenen Elemente eines Clusters, die nicht in dem

Tabelle 7.8: Übertragbarkeit von Parametern zwischen den Absichtsmodellen.

Parameter	Absichtsmodell	CPrec	CCov	CHm
opt_{iTrust}	Typen und Methoden	0,61	0,91	0,73
	Typen	0,19	0,29	0,23
$type_opt_{iTrust}$	Typen und Methoden	0,45	0,41	0,43
	Typen	0,45	0,47	0,46

korrespondierenden Cluster der Musterlösung enthalten waren, sinnvoll sind. Ein Cluster der Musterlösung umfasst die beiden Typen `DrugInteractionAction` und `DrugInteractionDAO`. Das vom System aufgebaute Absichtsmodell umfasst ebenfalls ein Cluster, welches diese beiden Typen enthält. Zusätzlich umfasst das Cluster den Typ `DrugInteractionValidator`. In der Musterlösung fehlt dieser Typ gänzlich. Im Quelltext von iTrust besitzt der Typ `DrugInteractionAction` ein Attribut vom Typ `DrugInteractionValidator` und nutzt dieses für Validierungszwecke. Darüber hinaus wird der Typ `DrugInteractionValidator` von keinem anderen Typ genutzt. Infolgedessen kann ein semantischer Zusammenhang zwischen diesen Typen angenommen werden, wodurch die vom System durchgeführte Platzierung in einem gemeinsamen Cluster korrekt erscheint.
Für viele der, in der Musterlösung von iTrust enthaltenen, Cluster findet das System weitere sinnvolle Elemente. Die im Vergleich zu Tabelle 7.3 niedrigeren Werte für iTrust sind daher zum Teil dem Umfang der, durch CoEST angebotenen, Musterlösung zuzuschreiben.

Abschließend wird untersucht, wie stark die Parameterbelegungen auf das, bei der Optimierung genutzte, Absichtsmodell angepasst wurden und ob Parameterbelegungen zwischen dem Absichtsmodell auf Typebene und dem Absichtsmodell aus Typen und Methoden übertragbar sind. Hierzu werden die besten Parameterbelegungen für iTrust aus Tabelle 7.3 und Tabelle 7.7 betrachtet. Tabelle 7.8 zeigt die Ergebnisse beider Parameterbelegungen für die verschiedenen Absichtsmodelle.
Die Parameter in opt_{iTrust} wurden auf der Musterlösung von iTrust für Typen und Methoden optimiert. Wird opt_{iTrust} für die Evaluation des Absichtsmodells auf Typebene herangezogen, zeigt sich eine deutliche Verringerung aller Werte. Folglich scheint opt_{iTrust} nicht für den Aufbau eines Absichtsmodells auf Typebene geeignet zu sein. Eine mögliche Erklärung für den schlechten *CPrec*-Wert von opt_{iTrust} auf der Typebene kann der zuvor angesprochene beschränkte Umfang der Musterlösung sein. Allerdings erklärt dies nicht den schwachen *CCov*-Wert von 0,29.
Die Parameterbelegung $type_opt_{iTrust}$ wurde auf der Musterlösung der Typebene für iTrust optimiert. Bei der Verwendung dieser Belegung für das Absichtsmodell aus Typen und Methoden bleibt die Cluster-Präzision unverändert. Lediglich der *CCov*-Wert und der *CHm*-Wert sinken leicht. Im Hinblick auf den niedrigsten *CHm*-Wert für iTrust aus Tabelle 7.3, welcher 0,53 beträgt, ist der, mit $type_opt_{iTrust}$ erreichte, Wert von 0,46 ist jedoch vergleichsweise schwach.
Beim Vergleich der beiden optimierten Parameterbelegungen (Siehe Abschnitt A) zeigen sich einige starke Unterschiede, welche Gründe für die schwache Übertragbarkeit der Parameterbelegungen liefern können. Im Hinblick auf die Parameter der syntaxbasierten Quelltextanalyse ist der Schwellwert für die Berücksichtigung von Zusammenhängen in $type_opt_{iTrust}$ mit 0,503 deutlich höher als der Schwellwert von 0,066 in opt_{iTrust}. Auch bei der lexikalischen Quelltextanalyse ist der Schwellwert in $type_opt_{iTrust}$ mit 0,878 erheblich höher als der Schwellwert von 0,333 in opt_{iTrust}. Zudem wird in $type_opt_{iTrust}$ der Einfluss der Kommentare und Zeichenketten bei der lexikalischen Analyse im Vergleich zu seinem Einfluss in opt_{iTrust} nahezu verdoppelt und besitzt nun ein Gewicht von 0,529.

Der Einfluss der *BA*- und *ZKBA*-Analyse, welche die Bezeichner im Quelltext einbeziehen, verringert sich hingegen. Bei den Parametern der semantischen Clusteranalyse fällt auf, dass sich der Einfluss von syntaktischen und lexikalischen Zusammenhängen tauscht. Wohingegen in opt_{iTrust} die Ergebnisse der syntaxbasierten Quelltextanalyse mit 0,611 berücksichtigt werden, fällt ihr Einfluss in $type_opt_{iTrust}$ auf 0,232 ab. Analog zu den einzelnen Quelltextanalysen steigt auch der Schwellwert für die Berücksichtigung semantischer Zusammenhänge von 0,384 in opt_{iTrust} auf 0,635 in $type_opt_{iTrust}$ an.
Insgesamt deuten die Ergebnisse aus Tabelle 7.8 an, dass Parameterbelegungen, die zu guten Ergebnissen, bezogen auf das Absichtsmodell für Typen oder Typen und Methoden, führen, nicht automatisch für beide Arten von Absichtsmodellen geeignet sind. Die Unterschiede von $type_opt_{iTrust}$ und opt_{iTrust} zeigen, dass lexikalische Zusammenhänge für das Absichtsmodell auf Typebene eine größere Bedeutung haben, als syntaktische Zusammenhänge. Darüber hinaus steigen die Schwellwerte in den beiden Quelltextanalysen und der semantischen Clusteranalyse stark an, was darauf hindeutet, dass viele der Zusammenhänge, die für ein Absichtsmodell der Typen und Methoden von Bedeutung sind, für die Betrachtung auf Typebene keine Relevanz besitzen.

8 Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war der Entwurf und Aufbau einer semantischen Quelltextrepräsentation. Hierzu wurde eine Graphrepräsentation konzipiert, welche sowohl die syntaktischen als auch semantischen Informationen im Quelltext beschreibt. Neben semantischen Zusammenhängen auf der Methoden- und Typebene enthält die Graphrepräsentation Absichtsknoten. Diese nehmen eine Unterteilung der Programmelemente in semantisch zusammenhängende Gruppen vor und erlauben einen einfachen Zugriff auf Programmelemente die im Zusammenhang mit einer konkreten Absicht stehen.
Für den Aufbau der semantischen Quelltextrepräsentation wurde eine syntaxbasierte und eine lexikalische Quelltextanalyse umgesetzt. Beide Analysen suchen unabhängig voneinander nach zusammenhängenden Programmelementen. Die gefundenen Zusammenhänge der einzelnen Quelltextanalysen werden zu semantischen Zusammenhängen zusammengefasst und als Kanten in den Graphen eingefügt. Anschließend wird eine hierarchische Clusteranalyse auf der Grundlage der gefundenen semantischen Zusammenhänge durchgeführt, um Gruppen semantisch zusammenhängender Programmelemente zu identifizieren. Jeder Absichtsknoten im Graphen repräsentiert eine solche Gruppe von Programmelementen. Um sich den Gegebenheiten eines Softwareprojekts optimal anpassen zu können, verfügt die Implementierung über diverse Parameter, die es dem Anwender erlauben, den Einfluss der verschiedenen Quelltextanalysen sowie der darin genutzten Merkmale individuell einzustellen. Das umgesetzte Verfahren zum Aufbau der semantischen Quelltextrepräsentation erlaubt die Hinzunahme weitere Quelltextanalysen ohne das Änderungen an den bestehenden Quelltextanalysen oder der Clusteranalyse notwendig sind. Die Berücksichtigung weiterer Informationen beim Aufbau der Quelltextrepräsentation ist folglich auf einfache Weise möglich.

Die Ergebnisse der Evaluation zeigen, dass die syntaxbasierte und die lexikalische Quelltextanalyse komplementäre Informationen liefern und deren Kombination, wie in dieser Arbeit umgesetzt, sinnvoll ist. Bei der Betrachtung der semantischen Zusammenhänge zwischen den Typen und Methoden eines Softwaresystems konnte eine Cluster-Präzision von bis zu 0,61 und eine Cluster-Abdeckung von bis zu 0,91 erzielt werden. Nach der Projektion der semantischen Zusammenhänge auf die Typebene wurde eine maximale Cluster-Präzision von 0,9 und eine Cluster-Abdeckung von bis zu 0,79 erreicht. Darüber hinaus deuten die Ergebnisse der Evaluation an, dass sich gut funktionierende Parameterbelegungen zwischen Softwareprojekten mit ähnlicher Charakteristik übertragen lassen.

Obgleich mit dem, in dieser Arbeit umgesetzten, Ansatz bereits gute Ergebnisse erzielt werden konnten, existieren verschiedene Erweiterungen, die das Potential besitzen, die se-

mantische Quelltextrepräsentation weiter zu verbessern. Im Hinblick auf die lexikalische Quelltextanalyse wäre eine Erweiterung der Vorverarbeitung um einen Schritt zur Auflösung von Abkürzungen denkbar, der über die Verwendung eines Abkürzungswörterbuchs hinausgeht. Hierdurch könnten projektspezifische Abkürzungen aufgelöst und die Schwerpunktanalyse weiter verbessert werden.
Darüber hinaus bietet es sich an, neben der, in dieser Arbeit betrachteten, Bedeutung der natürlichen Sprache auf Wortebene eine weitere lexikalische Analyse durchzuführen, um die Semantik auf Satzebene zu berücksichtigen. Insbesondere im Hinblick auf den Fließtext in Dokumentationskommentaren erscheint eine solche Erweiterung sinnvoll.
Neben den, in dieser Arbeit betrachteten, Informationen im Quelltext existieren weitere Möglichkeiten zur Identifikation semantischer Zusammenhänge. Eine dynamische Analyse von Quelltext wäre bspw. in der Lage, Zusammenhänge zwischen Typen und Methoden zu erkennen, die bei der syntaxbasierten Quelltextanalyse, aufgrund von Polymorphie, unentdeckt bleiben. Relevante Informationen über semantische Zusammenhänge können darüber hinaus aus Versionsverwaltungsdaten, durch bspw. die Analyse der getätigten Änderungen und der Beschreibung einer Version, gewonnen werden. Durch die Hinzunahme weiterer Quelltextanalysen kann ein breiteres Spektrum der zur Verfügung stehenden Informationen über den Quelltext abgedeckt werden, wodurch die semantische Quelltextrepräsentation unabhängiger von der Ausprägung einzelner Merkmale oder Informationen im Quelltext wird.
Weiteres Optimierungspotential besteht in der Clusteranalyse. Die, in dieser Arbeit umgesetzte, semantische Clusteranalyse unterteilt die Programmelemente in disjunkte Cluster. Im Allgemeinen kann ein Programmelement jedoch für verschiedene Absichten genutzt werden. Um Mehrfachzuordnungen zwischen Programmelementen und Absichtsknoten zu erlauben wäre eine Erweiterung der Clusteranalyse denkbar, sodass weitere Cluster aus der Clusterhierarchie als Absichtsknoten in die semantische Quelltextrepräsentation eingefügt werden.
Um die Aussagekraft der semantischen Quelltextrepräsentation weiter zu verbessern, können Absichtsknoten mit einer Beschreibung des, von ihnen repräsentierten, Konzepts angereichert werden. Diese Beschreibungen können helfen, den Grund für eine gemeinsame Gruppierung von Programmelementen herauszufinden. Außerdem erlauben sie es, die semantische Quelltextrepräsentation auf der Ebene der Absichtsknoten zu durchsuchen, indem bspw. ein Anfrage mit den Beschreibungen der Absichtsknoten verglichen wird.

Literaturverzeichnis

[AC17] AMARJEET ; CHHABRA, Jitender K.: Improving modular structure of software system using structural and lexical dependency. In: *Information and Software Technology* 82 (2017), Feb, S. 96–120. http://dx.doi.org/10.1016/j.infsof.2016.09.011. – DOI 10.1016/j.infsof.2016.09.011. – ISSN 0950–5849 (zitiert auf den Seiten 24, 34, 46, 47, 48, 51, 57, 79 und 82).

[ACC+02] ANTONIOL, G. ; CANFORA, G. ; CASAZZA, G. ; LUCIA, A. D. ; MERLO, E.: Recovering traceability links between code and documentation. In: *IEEE Transactions on Software Engineering* 28 (2002), Oct, Nr. 10, S. 970–983. http://dx.doi.org/10.1109/TSE.2002.1041053. – DOI 10.1109/TSE.2002.1041053. – ISSN 0098–5589 (zitiert auf Seite 29).

[Alp19] ALPAYDIN, Ethem: *Maschinelles Lernen*. Walter de Gruyter GmbH & Co KG, 2019. – ISBN 978–3–11–061789–4 (zitiert auf den Seiten 10 und 12).

[ALSU07] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULMAN, Jeffrey D.: *Compilers: principles, techniques, & tools*. 2nd ed. Pearson/Addison Wesley, 2007. – ISBN 978–0–321–48681–3 (zitiert auf Seite 4).

[BGJM17] BOJANOWSKI, Piotr ; GRAVE, Edouard ; JOULIN, Armand ; MIKOLOV, Tomas: Enriching word vectors with subword information. In: *Transactions of the Association for Computational Linguistics* 5 (2017), S. 135–146 (zitiert auf den Seiten 10, 18, 66, 74 und 91).

[BNJ03] BLEI, David M. ; NG, Andrew Y. ; JORDAN, Michael I.: Latent Dirichlet Allocation. In: *Journal of Machine Learning Research* 3 (2003), Nr. Jan, S. 993–1022. – ISSN 1533–7928 (zitiert auf den Seiten 21 und 65).

[Buß02] BUSSMANN, Hadumod: *Lexikon der Sprachwissenschaft*. 3., aktualisierte und erw. Aufl. Stuttgart : Kröner, 2002. – ISBN 3520452030; 9783520452030 (zitiert auf Seite 3).

[CB09] CHANG, Jonathan ; BLEI, David: Relational topic models for document networks. In: *Artificial Intelligence and Statistics*, 2009, S. 81–88 (zitiert auf Seite 17).

[CCZX19] CHEN, Xiang ; CHEN, Chunyang ; ZHANG, Dun ; XING, Zhenchang: SEthesaurus: WordNet in Software Engineering. In: *IEEE Transactions on Software Engineering* (2019), S. 1–1. http://dx.doi.org/10.1109/TSE.2019.2940439. – DOI 10.1109/TSE.2019.2940439. – ISSN 2326–3881 (zitiert auf den Seiten 68 und 90).

[CDMM12] CORAZZA, Anna ; DI MARTINO, Sergio ; MAGGIO, Valerio: LINSEN: An efficient approach to split identifiers and expand abbreviations. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012. – ISSN 1063–6773, S. 233–242 (zitiert auf den Seiten 27, 64 und 68).

[CDMMS16] Corazza, Anna ; Di Martino, Sergio ; Maggio, Valerio ; Scanniello, Giuseppe: Weighing lexical information for software clustering in the context of architecture recovery. In: *Empirical Software Engineering* 21 (2016), Nr. 1, S. 72–103 (zitiert auf den Seiten 23, 25, 63 und 64).

[CEBE09] Cleary, Brendan ; Exton, Chris ; Buckley, Jim ; English, Michael: An empirical analysis of information retrieval based concept location techniques in software comprehension. In: *Empirical Software Engineering* 14 (2009), Feb, Nr. 1, S. 93–130. http://dx.doi.org/10.1007/s10664-008-9095-3. – DOI 10.1007/s10664–008–9095–3. – ISSN 1573–7616 (zitiert auf Seite 64).

[CLRS09] *Kapitel* 26 Maximum Flow. In: Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to algorithms*. MIT press, 2009 (zitiert auf Seite 25).

[CV07] Cilibrasi, Rudi L. ; Vitanyi, Paul M.: The google similarity distance. In: *IEEE Transactions on knowledge and data engineering* 19 (2007), Nr. 3, S. 370–383 (zitiert auf Seite 21).

[CZD+09] Cornelissen, B. ; Zaidman, A. ; Deursen, A. v. ; Moonen, L. ; Koschke, R.: A Systematic Survey of Program Comprehension through Dynamic Analysis. In: *IEEE Transactions on Software Engineering* 35 (2009), Sep, Nr. 5, S. 684–702. http://dx.doi.org/10.1109/TSE.2009.28. – DOI 10.1109/TSE.2009.28. – ISSN 0098–5589 (zitiert auf Seite 1).

[ES19] Efstathiou, Vasiliki ; Spinellis, Diomidis: Semantic Source Code Models Using Identifier Embeddings. In: *Proceedings of the 16th International Conference on Mining Software Repositories*, IEEE Press, 2019 (MSR '19), 29–33 (zitiert auf den Seiten 18, 65 und 91).

[Fer14] Fernández, Maribel: *Programming languages and operational semantics*. Springer, 2014 https://doi.org/10.1007/978-1-4471-6368-8. – DOI 10.1007/978-1-4471-6368-8 (zitiert auf Seite 3).

[GAL18] Gharibi, G. ; Alanazi, R. ; Lee, Y.: Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension. In: *2018 IEEE International Conference on Big Data (Big Data)*, 2018, S. 4016–4025 (zitiert auf den Seiten 17, 20, 31 und 44).

[GHJV04] Gamma, E ; Helm, R ; Johnson, R ; Vlissides, J: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, 2004. – ISBN 978–3–8273–2199–2 (zitiert auf den Seiten 5 und 51).

[Har54] Harris, Zellig S.: Distributional Structure. In: *WORD* 10 (1954), Aug, Nr. 2–3, S. 146–162. http://dx.doi.org/10.1080/00437956.1954.11659520. – DOI 10.1080/00437956.1954.11659520. – ISSN 0043–7956, 2373–5112 (zitiert auf Seite 10).

[Hey19] Hey, Tobias: INDIRECT: intent-driven requirements-to-code traceability. In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings* IEEE Press, 2019, S. 190–191 (zitiert auf Seite 15).

[JLZZ18] Jiang, Y. ; Liu, H. ; Zhu, J. Q. ; Zhang, L.: Automatic and Accurate Expansion of Abbreviations in Parameters. In: *IEEE Transactions on Software Engineering* (2018), S. 1–1. http://dx.doi.org/10.1109/TSE.2018.2868762. – DOI 10.1109/TSE.2018.2868762. – ISSN 0098–5589 (zitiert auf den Seiten 26 und 64).

[JM08] JURAFSKY, Daniel ; MARTIN, James H.: *Speech and Language Processing.* Bd. 2. Pearson Education India, 2008 (zitiert auf den Seiten 8 und 9).

[KB09] KORNER, S. J. ; BRUMM, T.: RESI - A Natural Language Specification Improver. In: *2009 IEEE International Conference on Semantic Computing*, 2009, S. 1–8 (zitiert auf Seite 7).

[KDG07] KUHN, Adrian ; DUCASSE, Stéphane ; GÎRBA, Tudor: Semantic clustering: Identifying topics in source code. In: *Information and Software Technology* 49 (2007), Mar, Nr. 3, S. 230–243. http://dx.doi.org/10.1016/j.infsof.2006.10.017. – DOI 10.1016/j.infsof.2006.10.017. – ISSN 0950–5849 (zitiert auf den Seiten 7, 20, 21, 32 und 64).

[KGH+19] KUANG, Hongyu ; GAO, Hui ; HU, Hao ; MA, Xiaoxing ; LÜ, Jian ; MÄDER, Patrick ; EGYED, Alexander: Using frugal user feedback with closeness analysis on code to improve IR-based traceability recovery. In: *Proceedings of the 27th International Conference on Program Comprehension* IEEE Press, 2019, S. 369–379 (zitiert auf den Seiten 17 und 31).

[KNH+17] KUANG, H. ; NIE, J. ; HU, H. ; REMPEL, P. ; LÜ, J. ; EGYED, A. ; MÄDER, P.: Analyzing closeness of code dependencies for improving IR-based Traceability Recovery. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, S. 68–78 (zitiert auf den Seiten 23, 48, 50, 51, 56, 63 und 64).

[KQ05] KOSCHKE, Rainer ; QUANTE, Jochen: On Dynamic Feature Location. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2005 (ASE '05). – ISBN 978–1–58113–993–8, 86–95 (zitiert auf Seite 35).

[KR09] KAUFMAN, Leonard ; ROUSSEEUW, Peter J.: *Finding groups in data: an introduction to cluster analysis.* Bd. 344. John Wiley & Sons, 2009 (zitiert auf Seite 24).

[KSKW15] KUSNER, Matt ; SUN, Yu ; KOLKIN, Nicholas ; WEINBERGER, Kilian: From word embeddings to document distances. In: *International conference on machine learning*, 2015, S. 957–966 (zitiert auf Seite 19).

[Lev66] LEVENSHTEIN, Vladimir I.: Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet physics doklady* Bd. 10, 1966, S. 707–710 (zitiert auf Seite 10).

[LOV08] LUCIA, A. D. ; OLIVETO, R. ; VORRARO, L.: Using structural and semantic metrics to improve class cohesion. In: *2008 IEEE International Conference on Software Maintenance*, 2008, S. 27–36 (zitiert auf den Seiten 25, 79 und 82).

[LR03] LEVI, Paul ; REMBOLD, Ulrich: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure.* 4., aktualisierte und überarb. Aufl. Hanser Verlag, 2003. – ISBN 3–446–21932–3 (zitiert auf Seite 3).

[LSLZ14] LIU, X. ; SUN, X. ; LI, B. ; ZHU, J.: PFN: A novel program feature network for program comprehension. In: *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, 2014, S. 349–354 (zitiert auf den Seiten 17, 63 und 65).

[MAK+12] MISRA, J. ; ANNERVAZ, K. M. ; KAULGUD, V. ; SENGUPTA, S. ; TITUS, G.: Software Clustering: Unifying Syntactic and Semantic Features. In: *2012*

19th Working Conference on Reverse Engineering, 2012, S. 113–122 (zitiert auf den Seiten 17, 22, 26, 43, 45 und 46).

[MB17] Mahmoud, Anas ; Bradshaw, Gary: Semantic topic models for source code analysis. In: *Empirical Software Engineering* 22 (2017), Aug, Nr. 4, S. 1965–2000. http://dx.doi.org/10.1007/s10664-016-9473-1. – DOI 10.1007/s10664–016–9473–1. – ISSN 1573–7616 (zitiert auf den Seiten 20, 63 und 65).

[MCCD13] Mikolov, Tomas ; Chen, Kai ; Corrado, Greg ; Dean, Jeffrey: Efficient estimation of word representations in vector space. In: *arXiv preprint arXiv:1301.3781* (2013) (zitiert auf Seite 10).

[MK07] McLachlan, Geoffrey ; Krishnan, Thriyambakam: *The EM algorithm and extensions*. Bd. 382. John Wiley & Sons, 2007 (zitiert auf Seite 24).

[MPR09] McMillan, Collin ; Poshyvanyk, Denys ; Revelle, Meghan: Combining textual and structural analysis of software artifacts for traceability link recovery. In: *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering* IEEE Computer Society, 2009, S. 41–48 (zitiert auf Seite 23).

[MR17] Mayvan, Bahareh B. ; Rasoolzadegan, Abbas: Design pattern detection based on the graph theory. In: *Knowledge-Based Systems* 120 (2017), S. 211–225 (zitiert auf den Seiten 26 und 52).

[MRS08] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *Introduction to Information Retrieval*. Cambridge University Press, 2008. – ISBN 978–0–521–86571–5 (zitiert auf den Seiten 8, 9, 13, 20, 64, 82 und 95).

[MSC+13] Mikolov, Tomas ; Sutskever, Ilya ; Chen, Kai ; Corrado, Greg S. ; Dean, Jeff: Distributed representations of words and phrases and their compositionality. In: *Advances in neural information processing systems*, 2013, S. 3111–3119 (zitiert auf Seite 66).

[NS09] Ngomo, Axel-Cyrille N. ; Schumacher, Frank: Borderflow: A local graph clustering algorithm for natural language processing. In: *International Conference on Intelligent Text Processing and Computational Linguistics*, 2009, S. 547–558 (zitiert auf Seite 21).

[OAS16] Oruc, Murat ; Akal, Fuat ; Sever, Hayri: Detecting design patterns in object-oriented design models by using a graph mining approach. In: *2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT)* IEEE, 2016, S. 115–121 (zitiert auf den Seiten 26 und 52).

[OCP+16] Ou, Mingdong ; Cui, Peng ; Pei, Jian ; Zhang, Ziwei ; Zhu, Wenwu: Asymmetric transitivity preserving graph embedding. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2016, S. 1105–1114 (zitiert auf Seite 18).

[Pep07] Pepper, Peter: *Programmieren lernen: Eine grundlegende Einführung mit Java*. Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–72552–7 (zitiert auf Seite 6).

[PMP+16] 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 (2016), Nr. 9, S. 1155–1179. http://dx.doi.org/10.1002/spe.2346. – DOI 10.1002/spe.2346 (zitiert auf Seite 85).

[RCS10] Ratanotayanon, Sukanya ; Choi, Hye J. ; Sim, Susan E.: Using Transitive Changesets to Support Feature Location. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2010 (ASE '10). – ISBN 978–1–4503–0116–9, 341–344 (zitiert auf Seite 35).

[RM02] Robillard, Martin P. ; Murphy, Gail C.: Concern graphs: finding and describing concerns using structural program dependencies. In: *Proceedings of the 24th international conference on Software engineering* ACM, 2002, S. 406–416 (zitiert auf den Seiten 17 und 19).

[Rob08] Robillard, Martin P.: Topology analysis of software dependencies. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17 (2008), Nr. 4, S. 18 (zitiert auf den Seiten 17, 19 und 45).

[RPP06] Rechenberg, Peter ; Pomberger, Gustav ; Pirklbauer, Klaus: *Informatik-Handbuch.* 4., aktualisierte u. erw. Aufl. München; Wien : Hanser, 2006. – ISBN 3446401857; 9783446401853 (zitiert auf den Seiten 3 und 4).

[RW02] Rajlich, V. ; Wilde, N.: The role of concepts in program comprehension. In: *Proceedings 10th International Workshop on Program Comprehension*, 2002, S. 271–278 (zitiert auf Seite 1).

[SAO05] Souza, Sergio Cozzetti B. ; Anquetil, Nicolas ; Oliveira, Káthia M.: A Study of the Documentation Essential to Software Maintenance. In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, ACM, 2005 (SIGDOC '05). – ISBN 978–1–59593–175–7, 68–75. – event-place: Coventry, United Kingdom (zitiert auf Seite 61).

[SM11] Scanniello, G. ; Marcus, A.: Clustering Support for Static Concept Location in Source Code. In: *2011 IEEE 19th International Conference on Program Comprehension*, 2011, S. 1–10 (zitiert auf den Seiten 17, 21, 31 und 44).

[SMP15] Scanniello, Giuseppe ; Marcus, Andrian ; Pascale, Daniele: Link analysis algorithms for static concept location: an empirical assessment. In: *Empirical Software Engineering* 20 (2015), Nr. 6, S. 1666–1720 (zitiert auf Seite 23).

[Tin16] Ting, Kai M.: Precision and Recall. In: Sammut, Claude (Hrsg.) ; Webb, Geoffrey I. (Hrsg.): *Encyclopedia of Machine Learning and Data Mining.* Springer US, 2016, S. 1–1 (zitiert auf Seite 18).

[TWB+18] Tufano, M. ; Watson, C. ; Bavota, G. ; Penta, M. D. ; White, M. ; Poshyvanyk, D.: Deep Learning Similarities from Different Representations of Source Code. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, S. 542–553 (zitiert auf den Seiten 18, 52 und 65).

[Ull04] Ullenboom, Christian: *Java ist auch eine Insel.* Bd. 8. Galileo Press, 2004 (zitiert auf Seite 7).

[Win90] Winkler, William E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990) (zitiert auf Seite 11).

[WT04] Wen, Zhihua ; Tzerpos, V.: An effectiveness measure for software clustering algorithms. In: *Proceedings. 12th IEEE International Workshop on Program*

Comprehension, 2004., 2004. – ISSN 1092–8138, S. 194–203 (zitiert auf Seite 24).

[WT15] WEIGELT, S. ; TICHY, W. F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* Bd. 2, 2015, S. 819–820 (zitiert auf den Seiten 7, 15 und 85).

[ZWZ+19] ZHANG, Jian ; WANG, Xu ; ZHANG, Hongyu ; SUN, Hailong ; WANG, Kaixuan ; LIU, Xudong: A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In: *Proceedings of the 41st International Conference on Software Engineering*, IEEE Press, 2019 (ICSE '19), 783–794. – event-place: Montreal, Quebec, Canada (zitiert auf den Seiten 17, 18, 35 und 36).

Anhang

A Parameterkonfigurationen

Tabelle A.1: Auf iTrust optimierte Parameterkonfiguration für das Absichtsmodell aus Typen und Methoden.

Agent	Parameter	Wert
SDA	MM_controlFlow	0,003
	MM_override	0,551
	MM_methodCall	0,446
	MT_parameterType	0,217
	MT_instantiationType	0,185
	MT_controlFlow	0,105
	MT_explicitTypeAccess	0,017
	MT_throws	0,06
	MT_variableOrTypeAccess	0,018
	MT_methodParent	0,184
	MT_returnType	0,214
	TM_methodDefinition	0,184
	TM_controlFlow	0,118
	TM_methodCall	0,698
	TT_parentType	0,003
	TT_throws	0,091
	TT_instantiationType	0,002
	TT_controlFlow	0,007
	TT_variableOrTypeAccess	0,402
	TT_package	0,138
	TT_parameterType	0,025
	TT_typeHierarchy	0,018
	TT_innerType	0,111
	TT_explicitTypeAccess	0,123
	TT_attributeType	0,08
	SDA_THRESHOLD	0,066
LA	ANALYZER_COMMENT_WEIGHT	0,26
	ANALYZER_IDENTIFIER_WEIGHT	0,294
	ANALYZER_COMMENT_IDENTIFIER_WEIGHT	0,446
	IDENTIFIER_SIGNATURE_WEIGHT	0,594
	IDENTIFIER_STATEMENT_WEIGHT	0,406
	IDENTIFIER_CENTROID_WEIGHT	0,704
	IDENTIFIER_STRING_WEIGHT	0,296
	COMMENT_SIGNATURE_WEIGHT	0,77
	COMMENT_STATEMENT_WEIGHT	0,23
	COMMENT_CENTROID_WEIGHT	0,081
	COMMENT_STRING_WEIGHT	0,919
	CROSS_SIGNATURE_WEIGHT	0,446
	CROSS_STATEMENT_WEIGHT	0,554
	CROSS_CENTROID_WEIGHT	0,211
	CROSS_STRING_WEIGHT	0,789
	LA_THRESHOLD	0,333
Clusteranalyse	SDA_WEIGHT	0,611
	LA_WEIGHT	0,389
	SEMANTIC_RELATION_THRESHOLD	0,384
	CLUSTER_THRESHOLD	0,604

Tabelle A.2: Auf Easymock optimierte Parameterkonfiguration für das Absichtsmodell aus Typen und Methoden.

Agent	Parameter	Wert
SDA	MM_controlFlow	0,761
	MM_override	0,027
	MM_methodCall	0,212
	MT_parameterType	0,201
	MT_instantiationType	0,224
	MT_controlFlow	0,303
	MT_explicitTypeAccess	0,077
	MT_throws	0,052
	MT_variableOrTypeAccess	0,056
	MT_methodParent	0,046
	MT_returnType	0,04
	TM_methodDefinition	0,024
	TM_controlFlow	0,487
	TM_methodCall	0,489
	TT_parentType	0,064
	TT_throws	0,124
	TT_instantiationType	0,113
	TT_controlFlow	0,09
	TT_variableOrTypeAccess	0,134
	TT_package	0,117
	TT_parameterType	0,035
	TT_typeHierarchy	0,069
	TT_innerType	0,114
	TT_explicitTypeAccess	0,041
	TT_attributeType	0,1
	SDA_THRESHOLD	0,528
LA	ANALYZER_COMMENT_WEIGHT	0,388
	ANALYZER_IDENTIFIER_WEIGHT	0,602
	ANALYZER_COMMENT_IDENTIFIER_WEIGHT	0,01
	IDENTIFIER_SIGNATURE_WEIGHT	0,206
	IDENTIFIER_STATEMENT_WEIGHT	0,794
	IDENTIFIER_CENTROID_WEIGHT	0,601
	IDENTIFIER_STRING_WEIGHT	0,399
	COMMENT_SIGNATURE_WEIGHT	0,663
	COMMENT_STATEMENT_WEIGHT	0,334
	COMMENT_CENTROID_WEIGHT	0,479
	COMMENT_STRING_WEIGHT	0,521
	CROSS_SIGNATURE_WEIGHT	0,504
	CROSS_STATEMENT_WEIGHT	0,496
	CROSS_CENTROID_WEIGHT	0,458
	CROSS_STRING_WEIGHT	0,542
	LA_THRESHOLD	0,313
Clusteranalyse	SDA_WEIGHT	0,669
	LA_WEIGHT	0,331
	SEMANTIC_RELATION_THRESHOLD	0,38
	CLUSTER_THRESHOLD	0,64

Tabelle A.3: Auf Dronology optimierte Parameterkonfiguration für das Absichtsmodell aus Typen und Methoden.

Agent	Parameter	Wert
SDA	MM_controlFlow	0,675
	MM_override	0,083
	MM_methodCall	0,242
	MT_parameterType	0,244
	MT_instantiationType	0,235
	MT_controlFlow	0,284
	MT_explicitTypeAccess	0,072
	MT_throws	0,039
	MT_variableOrTypeAccess	0,078
	MT_methodParent	0,042
	MT_returnType	0,006
	TM_methodDefinition	0,164
	TM_controlFlow	0,214
	TM_methodCall	0,622
	TT_parentType	0,005
	TT_throws	0,148
	TT_instantiationType	0,004
	TT_controlFlow	0,011
	TT_variableOrTypeAccess	0,026
	TT_package	0,225
	TT_parameterType	0,041
	TT_typeHierarchy	0,029
	TT_innerType	0,018
	TT_explicitTypeAccess	0,201
	TT_attributeType	0,13
	SDA_THRESHOLD	0,066
LA	ANALYZER_COMMENT_WEIGHT	0,26
	ANALYZER_IDENTIFIER_WEIGHT	0,294
	ANALYZER_COMMENT_IDENTIFIER_WEIGHT	0,446
	IDENTIFIER_SIGNATURE_WEIGHT	0,594
	IDENTIFIER_STATEMENT_WEIGHT	0,406
	IDENTIFIER_CENTROID_WEIGHT	0,704
	IDENTIFIER_STRING_WEIGHT	0,296
	COMMENT_SIGNATURE_WEIGHT	0,77
	COMMENT_STATEMENT_WEIGHT	0,23
	COMMENT_CENTROID_WEIGHT	0,081
	COMMENT_STRING_WEIGHT	0,919
	CROSS_SIGNATURE_WEIGHT	0,446
	CROSS_STATEMENT_WEIGHT	0,554
	CROSS_CENTROID_WEIGHT	0,211
	CROSS_STRING_WEIGHT	0,789
	LA_THRESHOLD	0,333
Clusteranalyse	SDA_WEIGHT	0,611
	LA_WEIGHT	0,389
	SEMANTIC_RELATION_THRESHOLD	0,384
	CLUSTER_THRESHOLD	0,604

Tabelle A.4: Auf iTrust optimierte Parameterkonfiguration für das Absichtsmodell auf Typebene.

Agent	Parameter	Wert
SDA	MM_controlFlow	0,022
	MM_override	0,336
	MM_methodCall	0,642
	MT_parameterType	0,122
	MT_instantiationType	0,179
	MT_controlFlow	0,165
	MT_explicitTypeAccess	0,093
	MT_throws	0,173
	MT_variableOrTypeAccess	0,082
	MT_methodParent	0,097
	MT_returnType	0,089
	TM_methodDefinition	0,376
	TM_controlFlow	0,509
	TM_methodCall	0,115
	TT_parentType	0,214
	TT_throws	0,011
	TT_instantiationType	0,066
	TT_controlFlow	0,071
	TT_variableOrTypeAccess	0,161
	TT_package	0,023
	TT_parameterType	0,034
	TT_typeHierarchy	0,1
	TT_innerType	0,034
	TT_explicitTypeAccess	0,059
	TT_attributeType	0,227
	SDA_THRESHOLD	0,503
LA	ANALYZER_COMMENT_WEIGHT	0,529
	ANALYZER_IDENTIFIER_WEIGHT	0,151
	ANALYZER_COMMENT_IDENTIFIER_WEIGHT	0,32
	IDENTIFIER_SIGNATURE_WEIGHT	0,768
	IDENTIFIER_STATEMENT_WEIGHT	0,232
	IDENTIFIER_CENTROID_WEIGHT	0,457
	IDENTIFIER_STRING_WEIGHT	0,543
	COMMENT_SIGNATURE_WEIGHT	0,315
	COMMENT_STATEMENT_WEIGHT	0,685
	COMMENT_CENTROID_WEIGHT	0,701
	COMMENT_STRING_WEIGHT	0,299
	CROSS_SIGNATURE_WEIGHT	0,523
	CROSS_STATEMENT_WEIGHT	0,477
	CROSS_CENTROID_WEIGHT	0,666
	CROSS_STRING_WEIGHT	0,334
	LA_THRESHOLD	0,878
Clusteranalyse	SDA_WEIGHT	0,232
	LA_WEIGHT	0,768
	SEMANTIC_RELATION_THRESHOLD	0,635
	CLUSTER_THRESHOLD	0,639

Tabelle A.5: Auf Easymock optimierte Parameterkonfiguration für das Absichtsmodell auf Typebene.

Agent	Parameter	Wert
SDA	MM_controlFlow	0,474
	MM_override	0,394
	MM_methodCall	0,132
	MT_parameterType	0,017
	MT_instantiationType	0,295
	MT_controlFlow	0,147
	MT_explicitTypeAccess	0,047
	MT_throws	0,032
	MT_variableOrTypeAccess	0,034
	MT_methodParent	0,028
	MT_returnType	0,401
	TM_methodDefinition	0,02
	TM_controlFlow	0,415
	TM_methodCall	0,564
	TT_parentType	0,187
	TT_throws	0,034
	TT_instantiationType	0,209
	TT_controlFlow	0,261
	TT_variableOrTypeAccess	0,037
	TT_package	0,032
	TT_parameterType	0,01
	TT_typeHierarchy	0,019
	TT_innerType	0,031
	TT_explicitTypeAccess	0,152
	TT_attributeType	0,027
	SDA_THRESHOLD	0,528
LA	ANALYZER_COMMENT_WEIGHT	0,56
	ANALYZER_IDENTIFIER_WEIGHT	0,262
	ANALYZER_COMMENT_IDENTIFIER_WEIGHT	0,179
	IDENTIFIER_SIGNATURE_WEIGHT	0,212
	IDENTIFIER_STATEMENT_WEIGHT	0,788
	IDENTIFIER_CENTROID_WEIGHT	0,194
	IDENTIFIER_STRING_WEIGHT	0,806
	COMMENT_SIGNATURE_WEIGHT	0,408
	COMMENT_STATEMENT_WEIGHT	0,592
	COMMENT_CENTROID_WEIGHT	0,469
	COMMENT_STRING_WEIGHT	0,531
	CROSS_SIGNATURE_WEIGHT	0,805
	CROSS_STATEMENT_WEIGHT	0,195
	CROSS_CENTROID_WEIGHT	0,051
	CROSS_STRING_WEIGHT	0,949
	LA_THRESHOLD	0,003
Clusteranalyse	SDA_WEIGHT	0,669
	LA_WEIGHT	0,331
	SEMANTIC_RELATION_THRESHOLD	0,38
	CLUSTER_THRESHOLD	0,64

B Elemente der Graphrepräsentation

Tabelle B.6: Knoten der semantischen Quelltextrepräsentation.

Knoten	Beschreibung	Attribute
package	Paket	id, qualifiedName, simpleName
class	Klasse	id, qualifiedName, simpleName, final, static, abstract, visibility, comments, javaDocs, start, end
interface	Schnittstelle	id, qualifiedName, simpleName, visibility, comments, javaDocs, start, end
enum	Aufzählungstyp	id, qualifiedName, simpleName, visibility, comments, javaDocs, start, end
method	Methode	id, qualifiedName, simpleName, final, static, abstract, visibility, comments, javaDocs, start, end
variable	Variable	id, variableName, final, start, end
attribute	Attribut eines Typs	id, variableName, static, final, visibility, comments, javaDocs, start, end
codeBlock	Eine Folge von Anweisungen	id, comments, javaDocs, start, end
condition	Verzweigung	id, comments, javaDocs, start, end
try	`try-catch`-Anweisung	id, comments, javaDocs, start, end
switch	`switch-case`-Anweisung	id, comments, javaDocs, start, end
conditionalLoop	`while`-, `do-while`- oder `for`-Schleife	id, comments, javaDocs, start, end
forEachLoop	`foreach`-Schleife	id, comments, javaDocs, start, end
statement	Quelltextzeile z.B. Zuweisung	id, constantValues, comments, javaDocs, start, end
expression	Ausdruck z.B. Methodenaufruf	id, constantValues, comments, javaDocs, start, end
intent	Absichtsknoten	coherence

Tabelle B.7: Kanten der semantischen Quelltextrepräsentation.

Typ	Kantenbeschreibung	Quelle	Ziel	Attribute
parent	Definitionsursprung	*	*	id
typeDefinition	Typdefinition	package	class, interface, enum	id
innerType	Innerer Typ	class, interface, enum	class, interface, enum	id
attribute	Typattribut	class, interface, enum	attribute	id
initialization	Attributinitialisierung	attribute	expression	id
varDeclaration	Variablendeklaration	statement, forEachLoop	variable	id
declarationType	Deklarationstyp	attribute, variable	class, interface, enum	id
methodDef	Methodendefinition	class, interface, enum	method	id
parameter	Methodenparameter	method	variable	id
returnType	Rückgabetyp	method	class, interface, enum	id
override	Überschriebene Methode	method	method	id
throws	Ausnahme	method	class	id, position
body	Methodenrumpf	method	codeBlock	id
staticBlock	Statischer Quelltextblock	class, enum	codeBlock	id
constructor	Konstruktor	class, enum	method	id
statement	Anweisung	codeBlock	statement, condition, try, switch, conditionalLoop, forEachLoop	id
expression	Ausdruck	condition, conditionalLoop, forEachLoop	expression	id
enumValExpr	Enum-Wert	enum	expression	id
initStatement	Schleifeninitialisierung	forLoop	codeBlock	id
updateStatement	Schleifenaktualisierung	forLoop	codeBlock	id
then	`then`-Block einer Verzweigung	condition	codeBlock	id
else	`else`-Block einer Verzweigung	condition	codeBlock	id
switchSelector	`switch`-Ausdruck	switch	expression	id
case	`case`-Block	switch	codeBlock	id
caseExpression	`case`-Ausdruck	codeBlock	expression	id

Tabelle B.7: Kanten der semantischen Quelltextrepräsentation.

Typ	Kantenbeschreibung	Quelle	Ziel	Attribute
catchBlock	`catch`-Block	try	codeBlock	id
finalize	`finalize`-Block	try	codeBlock	id
catchVariable	Variablendeklaration	codeBlock	variable	id
catchedException	Behandelter Ausnahmetyp	codeBlock	class	id, position
extends	Erweiterter Typ	class, interface	class, interface	id, position
implement	Implementierte Schnittstelle	class, enum	interface	id, position
methodCall	Methodenaufruf	statement, expression	method	id, position
readVariable	Lesender Variablenzugriff	statement, expression	variable, attribute	id, position
writeVariable	Schreibender Variablenzugriff	statement, expression	variable, attribute	id, position
instantiation	Instanziierung	statement, expression	class, interface, enum	id, position
explTypeRef	Typreferenz	statement, expression	class, interface, enum	id, position
typeAccess	Typzugriff	statement, expression	class, interface, enum	id, position
typeArgument	Typargument	class, interface, enum, method, variable, attribute, statement, expression	class, interface, enum	id, context, typeArcId, position
sdaRelation	Syntaktischer Zusammenhang	class, interface, enum, method	class, interface, enum, method	weight
laRelation	Lexikalischer Zusammenhang	class, interface, enum, method	class, interface, enum, method	weight
semanticRelation	Semantischer Zusammenhang	class, interface, enum, method	class, interface, enum, method	weight
member	Zugehörigkeit zu einer Absicht	class, interface, enum, method	intent	weight