

Masterarbeit

**Evaluation von Heuristiken für die testgetriebene
Exploration von Enterprise-Java-Beans**

Niels Gundermann

Prüfer: Univ. Prof. Dr. Friedrich Steimann

Zweitprüfer : Dr. Daniela Keller

Betreuer: Univ. Prof. Dr. Friedrich Steimann

Lehrgebiet Programmiersysteme

Fachbereich Informatik

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema *Evaluation von Heuristiken für die testgetriebene Exploration von Enterprise-Java-Beans* selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird. Außerdem räume ich dem Lehrgebiet das Recht ein, die Arbeit für eigene Lehr- und Forschungstätigkeiten auszuwerten und unter Angabe des Autors geeignet zu publizieren.

Neubrandenburg, den 20.11.2021

Niels Gundermann

Abstract

Mit dem Verfahren der testgetriebenen Codesuche ist ein Software-Entwickler in der Lage bestehenden Code in einem Repository nach vorgegebenen Kriterien zu durchsuchen. Die Kriterien beinhalten dabei Testfälle, die auf den bestehenden Code im Repository angewendet werden. Ausgehend davon, dass eine solche Suche während der Laufzeit innerhalb eines Systems möglich ist, wird die Zeit, die dafür zur Verfügung steht zu einem kritischen Aspekt.

Daher zielt diese Arbeit darauf ab, Heuristiken zu evaluieren, durch die die testgetriebene Codesuche beschleunigt werden kann. Dazu wird die Exploration im Kontext der Arbeit formal beschrieben. Aufbauend auf dieser formalen Beschreibung werden drei Heuristiken vorgestellt, die bei der Exploration in einem bestehenden System evaluiert werden. Das Repository bildet dabei ein EJB-Container mit ca. 900 EJBs innerhalb des Systems.

Die Untersuchungsergebnisse zeigen, dass alle drei Heuristiken - wenn auch mit Abstufungen - das Potential haben, die Exploration zu beschleunigen.

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
Tabellenverzeichnis	v
Listings	xiv
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau dieser Arbeit	3
2 Forschungsziel und Abgrenzung	5
2.1 Testgetriebene Codesuche	5
2.2 Testgetriebene Exploration von EJBs	8
3 Theoretische Grundlagen	13
3.1 Strukturelle Evaluation	13
3.1.1 Struktur für die Definition von Typen	14
3.1.2 Definition der Matcher	16
3.1.3 Ergebnis der strukturellen Evaluation	19
3.2 Generierung der Proxies auf Basis von Matchern	20
3.2.1 Struktur für die Definition von Proxies	21
3.2.2 Delegation von Methoden im Proxy	24
3.2.3 Generierung von Proxies	28
3.2.4 Anzahl struktureller Proxies innerhalb einer Bibliothek	42
3.3 Semantische Evaluation	45
3.3.1 Besonderheiten der Testfälle	46

3.3.2	Algorithmus für die semantische Evaluation	47
3.4	Heuristiken	49
3.4.1	Beachtung des Matcherratings (LMF)	49
3.4.2	Beachtung positiver Tests (PTTF)	54
3.4.3	Beachtung fehlgeschlagener Methodenaufrufe (BL_NMC)	56
4	Implementierung	59
4.1	Modul: SignatureMatching	60
4.2	Modul: ComponentTester	66
4.3	Modul: DesiredComponentSourcerer	69
5	Untersuchungsergebnisse	73
5.1	Darstellung der Untersuchungsergebnisse	74
5.2	Ausgangspunkt	75
5.3	Ergebnisse für die Heuristik LMF	78
5.4	Ergebnisse für die Heuristik PTTF	80
5.5	Ergebnisse für die Heuristik BL_NMC	83
5.6	Ergebnisse für die Kombination der Heuristiken	86
5.6.1	Kombination: LMF + PTTF	86
5.6.2	Kombination: LMF + BL_NMC	88
5.6.3	Kombination: PTTF + BL_NMC	90
5.6.4	Kombination: LMF + PTTF + BL_NMC	92
6	Diskussion	97
6.1	Auswertung der Untersuchungsergebnisse	97
6.1.1	Einzelbetrachtung	97
6.1.2	Synergien	98
6.1.3	Erhöhte Komplexität	99
6.1.4	Zusammenfassung	100
6.2	Kritik am Ansatz	100
6.2.1	Seiteneffekte durch Testevaluation	100
6.2.2	Auswirkung auf die Verfügbarkeit eines Systems	101

6.2.3	Auswirkung von Änderungen an bestehenden Komponenten	102
6.2.4	Nutzen für den Entwickler	103
6.3	Erweiterungsmöglichkeiten	104
6.3.1	Zusätzliche Matcher	104
6.3.2	Default-Implementierungen in required Typen	105
7	Schlussbemerkung	109
7.1	Zusammenfassung	109
7.2	Ausblick	110
	Glossar	111
A	Kombination von Matchern	113
B	Verwendung aller Heuristiken	117
C	Deklaration der relevanten Typen	119
D	Interfaces und Test-Implementierungen	127
E	Ergebnisse für die Heuristik LMF (Ergänzungen)	137
	Literaturverzeichnis	164

Abbildungsverzeichnis

1.1	Abhängigkeiten von nachfragenden und angebotenen Komponenten	1
2.1	The testdriven reuse ”cycle” [HJ13]	6
2.2	Implementierungsprozess	9
2.3	Explorationsprozess	10

3.1	Delegation der Methode <code>heal</code>	25
3.2	Delegation der Methode <code>heal</code> mit Parametern in unterschiedlicher Reihenfolge	26
3.3	Delegation der Methode <code>extinguishFire</code> mit Typkonvertierungen	28
3.4	AST für das Beispiel zum Sub-Proxy	30
3.5	AST für das Beispiel zum Content-Proxy	34
3.6	AST für das Beispiel zum Container-Proxy	37
3.7	AST für das Beispiel zum strukturellen Proxy	40
4.1	Architektur	59
4.2	Modul: <code>SignatureMatching</code>	61
4.3	Klassendiagramm: <code>StructuralTypeMatcher</code> und <code>MatchingInfos</code>	63
4.4	Klassendiagramm: <code>TypeMatcher</code> und <code>SingleMatchingInfo</code>	64
4.5	Klassendiagramm: <code>MethodMatchingInfo</code>	65
4.6	Klassendiagramm: <code>TypeConverter</code>	67
4.7	Modul: <code>ComponentTester</code>	68
4.8	Modul: <code>DesiredComponentSourcerer</code>	70

Tabellenverzeichnis

3.1	Struktur für die Definition einer Bibliothek von Typen	14
3.2	Grammatikregeln mit Erläuterungen für die Deklaration eines Proxies	22
3.3	Grammatikregeln mit Attributen für die Deklaration eines Proxies	23
3.4	Proxy-Arten mit Matchingrelationen und Proxy-Funktionen	42
3.5	Varianten für die Ermittlung des Matcherratings einer Menge von <i>provided</i> <i>Typen</i>	53
4.1	Zuordnung der Matcher zu den Matcher- und Generator-Implementierungen	61

5.1	Required Typen mit Kürzeln von matchenden Kombinationen von provided Typen für die Evaluation	73
5.2	Beispiel: Vier-Felder-Tafel	75
5.3	Anzahl strukturell gematchten provided Typen für die Evaluation	76
5.4	Ausgangspunkt für TEI1	76
5.5	Ausgangspunkt für TEI2	76
5.6	Ausgangspunkt für TEI3	76
5.7	Ausgangspunkt für TEI4 1. Durchlauf	76
5.8	Ausgangspunkt für TEI4 2. Durchlauf	76
5.9	Ausgangspunkt für TEI5 1. Durchlauf	77
5.10	Ausgangspunkt für TEI5 2. Durchlauf	77
5.11	Ausgangspunkt für TEI6 1. Durchlauf	77
5.12	Ausgangspunkt für TEI6 2. Durchlauf	77
5.13	Ausgangspunkt für TEI7 1. Durchlauf	77
5.14	Ausgangspunkt für TEI7 2. Durchlauf	77
5.15	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI1 1. Durchlauf	78
5.16	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI2 1. Durchlauf	78
5.17	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI3 1. Durchlauf	78
5.18	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI4 1. Durchlauf	78
5.19	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI4 2. Durchlauf	79
5.20	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI5 1. Durchlauf	79
5.21	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI5 2. Durchlauf	79
5.22	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI6 1. Durchlauf	79
5.23	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI6 2. Durchlauf	79
5.24	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI7 1. Durchlauf	79
5.25	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI7 2. Durchlauf	80
5.26	Ergebnisse <i>PTTF</i> für TEI1 1. Durchlauf	80
5.27	Ergebnisse <i>PTTF</i> für TEI2 1. Durchlauf	81
5.28	Ergebnisse <i>PTTF</i> für TEI3 1. Durchlauf	81
5.29	Ergebnisse <i>PTTF</i> für TEI4 1. Durchlauf	82

5.30	Ergebnisse <i>PTTF</i> für TEI4 2. Durchlauf	82
5.31	Ergebnisse <i>PTTF</i> für TEI5 1. Durchlauf	82
5.32	Ergebnisse <i>PTTF</i> für TEI5 2. Durchlauf	82
5.33	Ergebnisse <i>PTTF</i> für TEI6 1. Durchlauf	82
5.34	Ergebnisse <i>PTTF</i> für TEI6 2. Durchlauf	82
5.35	Ergebnisse <i>PTTF</i> für TEI7 1. Durchlauf	83
5.36	Ergebnisse <i>PTTF</i> für TEI7 2. Durchlauf	83
5.37	Ergebnisse <i>BL_NMC</i> für TEI1 1. Durchlauf	83
5.38	Ergebnisse <i>BL_NMC</i> für TEI2 1. Durchlauf	84
5.39	Ergebnisse <i>BL_NMC</i> für TEI3 1. Durchlauf	84
5.40	Ergebnisse <i>BL_NMC</i> für TEI4 1. Durchlauf	84
5.41	Ergebnisse <i>BL_NMC</i> für TEI4 2. Durchlauf	84
5.42	Ergebnisse <i>BL_NMC</i> für TEI5 1. Durchlauf	84
5.43	Ergebnisse <i>BL_NMC</i> für TEI5 2. Durchlauf	84
5.44	Ergebnisse <i>BL_NMC</i> für TEI6 1. Durchlauf	85
5.45	Ergebnisse <i>BL_NMC</i> für TEI6 2. Durchlauf	85
5.46	Ergebnisse <i>BL_NMC</i> für TEI7 1. Durchlauf	85
5.47	Ergebnisse <i>BL_NMC</i> für TEI7 2. Durchlauf	85
5.48	Rangfolge der Heuristiken (Einzelbetrachtung)	86
5.49	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI1	86
5.50	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI2 1. Durchlauf	86
5.51	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI3 1. Durchlauf	86
5.52	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI4 1. Durchlauf	87
5.53	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI4 2. Durchlauf	87
5.54	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI5 1. Durchlauf	87
5.55	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI5 2. Durchlauf	87
5.56	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI6 1. Durchlauf	87
5.57	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI6 2. Durchlauf	87
5.58	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI7 1. Durchlauf	87
5.59	Ergebnisse <i>LMF</i> + <i>PTTF</i> für TEI7 2. Durchlauf	88
5.60	Ergebnisse <i>LMF</i> + <i>BL_NMC</i> für TEI1	88

5.61	Ergebnisse $LMF + BL_NMC$ für TEI2 1. Durchlauf	88
5.62	Ergebnisse $LMF + BL_NMC$ für TEI3 1. Durchlauf	88
5.63	Ergebnisse $LMF + BL_NMC$ für TEI4 1. Durchlauf	89
5.64	Ergebnisse $LMF + BL_NMC$ für TEI4 2. Durchlauf	89
5.65	Ergebnisse $LMF + BL_NMC$ für TEI5 1. Durchlauf	89
5.66	Ergebnisse $LMF + BL_NMC$ für TEI5 2. Durchlauf	89
5.67	Ergebnisse $LMF + PTTF$ für TEI6 1. Durchlauf	89
5.68	Ergebnisse $LMF + PTTF$ für TEI6 2. Durchlauf	89
5.69	Ergebnisse $LMF + BL_NMC$ für TEI7 1. Durchlauf	89
5.70	Ergebnisse $LMF + BL_NMC$ für TEI7 2. Durchlauf	90
5.71	Ergebnisse $PTTF + BL_NMC$ für TEI1	90
5.72	Ergebnisse $PTTF + BL_NMC$ für TEI2 1. Durchlauf	90
5.73	Ergebnisse $PTTF + BL_NMC$ für TEI3 1. Durchlauf	90
5.74	Ergebnisse $PTTF + BL_NMC$ für TEI4 1. Durchlauf	91
5.75	Ergebnisse $PTTF + BL_NMC$ für TEI4 2. Durchlauf	91
5.76	Ergebnisse $PTTF + BL_NMC$ für TEI5 1. Durchlauf	91
5.77	Ergebnisse $PTTF + BL_NMC$ für TEI5 2. Durchlauf	91
5.78	Ergebnisse $PTTF + PTTF$ für TEI6 1. Durchlauf	91
5.79	Ergebnisse $PTTF + PTTF$ für TEI6 2. Durchlauf	91
5.80	Ergebnisse $PTTF + BL_NMC$ für TEI7 1. Durchlauf	91
5.81	Ergebnisse $PTTF + BL_NMC$ für TEI7 2. Durchlauf	92
5.82	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI1	92
5.83	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI2 1. Durchlauf	92
5.84	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI3 1. Durchlauf	93
5.85	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI4 1. Durchlauf	93
5.86	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI4 2. Durchlauf	93
5.87	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI5 1. Durchlauf	93
5.88	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI5 2. Durchlauf	93
5.89	Ergebnisse $LMF + PTTF + PTTF$ für TEI6 1. Durchlauf	93
5.90	Ergebnisse $LMF + PTTF + PTTF$ für TEI6 2. Durchlauf	94
5.91	Ergebnisse $LMF + PTTF + BL_NMC$ für TEI7 1. Durchlauf	94

5.92	Ergebnisse <i>LMF</i> + <i>PTTF</i> + <i>BL_NMC</i> für TEI7 2. Durchlauf	94
5.93	Rangfolge der Heuristiken (Kombinationen)	95
E.1	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI1 1. Durchlauf	138
E.2	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI2 1. Durchlauf	138
E.3	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI3 1. Durchlauf	138
E.4	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI4 1. Durchlauf	138
E.5	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI4 2. Durchlauf	138
E.6	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI5 1. Durchlauf	138
E.7	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI5 2. Durchlauf	138
E.8	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI6 1. Durchlauf	139
E.9	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI6 2. Durchlauf	139
E.10	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI7 1. Durchlauf	139
E.11	Ergebnisse <i>LMF</i> mit Variante 1.1 für TEI7 2. Durchlauf	139
E.12	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI1 1. Durchlauf	139
E.13	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI2 1. Durchlauf	139
E.14	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI3 1. Durchlauf	139
E.15	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI4 1. Durchlauf	140
E.16	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI4 2. Durchlauf	140
E.17	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI5 1. Durchlauf	140
E.18	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI5 2. Durchlauf	140
E.19	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI6 1. Durchlauf	140
E.20	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI6 2. Durchlauf	140
E.21	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI7 1. Durchlauf	141
E.22	Ergebnisse <i>LMF</i> mit Variante 1.2 für TEI7 2. Durchlauf	141
E.23	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI1 1. Durchlauf	141
E.24	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI2 1. Durchlauf	141
E.25	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI3 1. Durchlauf	141
E.26	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI4 1. Durchlauf	141
E.27	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI4 2. Durchlauf	141
E.28	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI5 1. Durchlauf	142
E.29	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI5 2. Durchlauf	142

E.30	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI6 1. Durchlauf	142
E.31	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI6 2. Durchlauf	142
E.32	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI7 1. Durchlauf	142
E.33	Ergebnisse <i>LMF</i> mit Variante 1.3 für TEI7 2. Durchlauf	142
E.34	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI1 1. Durchlauf	143
E.35	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI2 1. Durchlauf	143
E.36	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI3 1. Durchlauf	143
E.37	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI4 1. Durchlauf	143
E.38	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI4 2. Durchlauf	143
E.39	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI5 1. Durchlauf	143
E.40	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI5 2. Durchlauf	143
E.41	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI6 1. Durchlauf	143
E.42	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI6 2. Durchlauf	144
E.43	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI7 1. Durchlauf	144
E.44	Ergebnisse <i>LMF</i> mit Variante 1.4 für TEI7 2. Durchlauf	144
E.45	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI1 1. Durchlauf	144
E.46	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI2 1. Durchlauf	144
E.47	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI3 1. Durchlauf	144
E.48	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI4 1. Durchlauf	145
E.49	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI4 2. Durchlauf	145
E.50	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI5 1. Durchlauf	145
E.51	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI5 2. Durchlauf	145
E.52	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI6 1. Durchlauf	145
E.53	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI6 2. Durchlauf	145
E.54	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI7 1. Durchlauf	145
E.55	Ergebnisse <i>LMF</i> mit Variante 2.1 für TEI7 2. Durchlauf	146
E.56	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI1 1. Durchlauf	146
E.57	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI2 1. Durchlauf	146
E.58	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI3 1. Durchlauf	146
E.59	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI4 1. Durchlauf	146
E.60	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI4 2. Durchlauf	146

E.61	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI5 1. Durchlauf	147
E.62	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI5 2. Durchlauf	147
E.63	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI6 1. Durchlauf	147
E.64	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI6 2. Durchlauf	147
E.65	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI7 1. Durchlauf	147
E.66	Ergebnisse <i>LMF</i> mit Variante 2.2 für TEI7 2. Durchlauf	147
E.67	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI1 1. Durchlauf	148
E.68	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI2 1. Durchlauf	148
E.69	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI3 1. Durchlauf	148
E.70	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI4 1. Durchlauf	148
E.71	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI4 2. Durchlauf	148
E.72	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI5 1. Durchlauf	148
E.73	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI5 2. Durchlauf	148
E.74	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI6 1. Durchlauf	148
E.75	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI6 2. Durchlauf	149
E.76	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI7 1. Durchlauf	149
E.77	Ergebnisse <i>LMF</i> mit Variante 2.3 für TEI7 2. Durchlauf	149
E.78	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI1 1. Durchlauf	149
E.79	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI2 1. Durchlauf	149
E.80	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI3 1. Durchlauf	149
E.81	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI4 1. Durchlauf	150
E.82	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI4 2. Durchlauf	150
E.83	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI5 1. Durchlauf	150
E.84	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI5 2. Durchlauf	150
E.85	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI6 1. Durchlauf	150
E.86	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI6 2. Durchlauf	150
E.87	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI7 1. Durchlauf	150
E.88	Ergebnisse <i>LMF</i> mit Variante 2.4 für TEI7 2. Durchlauf	151
E.89	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI1 1. Durchlauf	151
E.90	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI2 1. Durchlauf	151
E.91	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI3 1. Durchlauf	151

E.92	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI4 1. Durchlauf	151
E.93	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI4 2. Durchlauf	151
E.94	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI5 1. Durchlauf	152
E.95	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI5 2. Durchlauf	152
E.96	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI6 1. Durchlauf	152
E.97	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI6 2. Durchlauf	152
E.98	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI7 1. Durchlauf	152
E.99	Ergebnisse <i>LMF</i> mit Variante 3.1 für TEI7 2. Durchlauf	152
E.100	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI1 1. Durchlauf	153
E.101	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI2 1. Durchlauf	153
E.102	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI3 1. Durchlauf	153
E.103	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI4 1. Durchlauf	153
E.104	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI4 2. Durchlauf	153
E.105	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI5 1. Durchlauf	153
E.106	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI5 2. Durchlauf	153
E.107	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI6 1. Durchlauf	153
E.108	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI6 2. Durchlauf	154
E.109	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI7 1. Durchlauf	154
E.110	Ergebnisse <i>LMF</i> mit Variante 3.2 für TEI7 2. Durchlauf	154
E.111	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI1 1. Durchlauf	154
E.112	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI2 1. Durchlauf	154
E.113	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI3 1. Durchlauf	154
E.114	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI4 1. Durchlauf	155
E.115	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI4 2. Durchlauf	155
E.116	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI5 1. Durchlauf	155
E.117	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI5 2. Durchlauf	155
E.118	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI6 1. Durchlauf	155
E.119	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI6 2. Durchlauf	155
E.120	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI7 1. Durchlauf	155
E.121	Ergebnisse <i>LMF</i> mit Variante 3.3 für TEI7 2. Durchlauf	156
E.122	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI1 1. Durchlauf	156

E.123	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI2 1. Durchlauf	156
E.124	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI3 1. Durchlauf	156
E.125	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI4 1. Durchlauf	156
E.126	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI4 2. Durchlauf	156
E.127	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI5 1. Durchlauf	157
E.128	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI5 2. Durchlauf	157
E.129	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI6 1. Durchlauf	157
E.130	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI6 2. Durchlauf	157
E.131	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI7 1. Durchlauf	157
E.132	Ergebnisse <i>LMF</i> mit Variante 3.4 für TEI7 2. Durchlauf	157
E.133	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI1 1. Durchlauf	158
E.134	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI2 1. Durchlauf	158
E.135	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI3 1. Durchlauf	158
E.136	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI4 1. Durchlauf	158
E.137	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI4 2. Durchlauf	158
E.138	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI5 1. Durchlauf	158
E.139	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI5 2. Durchlauf	158
E.140	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI6 1. Durchlauf	158
E.141	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI6 2. Durchlauf	159
E.142	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI7 1. Durchlauf	159
E.143	Ergebnisse <i>LMF</i> mit Variante 4.1 für TEI7 2. Durchlauf	159
E.144	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI1 1. Durchlauf	159
E.145	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI2 1. Durchlauf	159
E.146	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI3 1. Durchlauf	159
E.147	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI4 1. Durchlauf	160
E.148	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI4 2. Durchlauf	160
E.149	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI5 1. Durchlauf	160
E.150	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI5 2. Durchlauf	160
E.151	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI6 1. Durchlauf	160
E.152	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI6 2. Durchlauf	160
E.153	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI7 1. Durchlauf	160

E.154	Ergebnisse <i>LMF</i> mit Variante 4.2 für TEI7 2. Durchlauf	161
E.155	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI1 1. Durchlauf	161
E.156	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI2 1. Durchlauf	161
E.157	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI3 1. Durchlauf	161
E.158	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI4 1. Durchlauf	161
E.159	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI4 2. Durchlauf	161
E.160	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI5 1. Durchlauf	162
E.161	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI5 2. Durchlauf	162
E.162	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI6 1. Durchlauf	162
E.163	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI6 2. Durchlauf	162
E.164	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI7 1. Durchlauf	162
E.165	Ergebnisse <i>LMF</i> mit Variante 4.3 für TEI7 2. Durchlauf	162
E.166	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI1 1. Durchlauf	163
E.167	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI2 1. Durchlauf	163
E.168	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI3 1. Durchlauf	163
E.169	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI4 1. Durchlauf	163
E.170	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI4 2. Durchlauf	163
E.171	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI5 1. Durchlauf	163
E.172	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI5 2. Durchlauf	163
E.173	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI6 1. Durchlauf	163
E.174	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI6 2. Durchlauf	164
E.175	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI7 1. Durchlauf	164
E.176	Ergebnisse <i>LMF</i> mit Variante 4.4 für TEI7 2. Durchlauf	164

Listings

3.1	Bibliothek <i>Example</i> von Typen	15
3.2	Einfache Methoden-Delegation	24
3.3	Methoden-Delegation mit Parametern in unterschiedlicher Reihenfolge	25
3.4	Methoden-Delegation mit Typkonvertierung	27
3.5	Sub-Proxy für Patient	29
3.6	Content-Proxy für Medicine	33
3.7	Container-Proxy für MedCabniet	36
3.8	Struktureller Proxy für MedicalFireFighter	39
3.9	Beispielhafte Implementierung einer <code>eval</code> -Methode	46
3.10	Semantische Evaluation ohne Heuristiken	48
3.11	Semantische Evaluation mit Heuristik LMF	52
3.12	Semantische Evaluation mit Heuristik PTTF	55
3.13	Evaluierung einzelner Proxies mit BL_MNC	57
3.14	Blacklist-Methode für Heuristik BL_NMC	57
3.15	Evaluation mehrere Proxies mit BL_MNC	58
6.1	Required Typ <i>Calc</i>	106
6.2	Interface Calc	106
6.3	Test CalcTest	107
A.1	Klasse: MatcherCombiner	114
A.2	Default-Instanziierung des StructuralTypeMatchers im DesiredComponentFinder	115
B.1	Kombination aller Heuristiken	117
C.1	Deklaration von ElerFTFoerderprogrammeProvider	119
C.2	Deklaration von FoerderprogrammeProvider	119

C.3	Deklaration von MinimalFoerderprogrammeProvider	119
C.4	Deklaration von IntubatingFireFighter	120
C.5	Deklaration von IntubatingFreeing	120
C.6	Deklaration von IntubatingPatientFireFighter	120
C.7	Deklaration von KOFGPCProvider	120
C.8	Deklaration von ElerFTFoerderprogramm	120
C.9	Deklaration von Foerderprogramm	121
C.10	Deklaration von DvAntragsJahr	121
C.11	Deklaration von DvFoerderprogramm	121
C.12	Deklaration von Injured	122
C.13	Deklaration von Fire	122
C.14	Deklaration von IntubationPatient	122
C.15	Deklaration von ElerFTStammdatenAuskunftService	122
C.16	Deklaration von StammdatenAuskunftService	124
C.17	Deklaration von Doctor	125
C.18	Deklaration von FireFighter	125
D.1	Interface ElerFTFoerderprogrammeProvider	127
D.2	Interface FoerderprogrammeProvider	127
D.3	Interface MinimalFoerderprogrammeProvider	128
D.4	Interface IntubatingFireFighter	128
D.5	Interface IntubatingFreeing	128
D.6	Interface IntubatingPatientFireFighter	128
D.7	Interface KOFGPCProvider	129
D.8	Interface ElerFTFoerderprogrammProviderTest	129
D.9	Interface FoerderprogrammProviderTest	130
D.10	Interface MinimalFoerderprogrammProviderTest	131
D.11	Interface IntubatingFireFighterTest	132
D.12	Interface IntubatingFreeingTest	133
D.13	Interface IntubatingPatientFireFighterTest	134
D.14	Interface KOFGPCProviderTest	135

Kapitel 1

Einleitung

1.1 Motivation

In größeren Software-Systemen ist es üblich, dass mehrere Komponenten miteinander über Schnittstellen kommunizieren. In der Regel werden diese Schnittstellen so konzipiert, dass sie Informationen oder Services anbieten, die von anderen Komponenten abgefragt und benutzt werden können. Dabei wird zwischen der Komponente, welche die Schnittstelle implementiert - als angebotene Komponente - und der Komponente, welche die Schnittstelle nutzen soll - als nachfragende Komponente - unterschieden (siehe Abbildung 1.1).



Abbildung 1.1: Abhängigkeiten von nachfragenden und angebotenen Komponenten

Wird von einer nachfragenden Komponente eine Information benötigt, die in dieser Form noch nicht angeboten wird, so wird häufig ein neues Interface für diese benötigte Information erstellt, welches dann passend dazu implementiert wird. Dabei muss neben der Anpassung der nachfragenden Komponente auch eine Anpassung oder Erzeugung der anbietenden Komponente

erfolgen und zusätzlich das neue Interface deklariert werden. Zudem bedingt eine nachträgliche Änderung der neuen Schnittstelle ebenfalls eine Anpassung der drei genannten Artefakte.

In einem großen Software-System mit einer Vielzahl von bestehenden Schnittstellen ist eine gewisse Wahrscheinlichkeit gegeben, dass die Informationen oder Services, die von einer neuen nachfragenden Komponente benötigt werden, in einer ähnlichen Form bereits existieren. Das Problem ist jedoch, dass die manuelle Evaluation der Schnittstellen mitunter sehr aufwendig bis, aufgrund von unzureichender Dokumentation und Kenntnis über die bestehenden Schnittstellen, unmöglich ist.

Weiterhin ist es denkbar, dass ein Software-System auf unterschiedlichen Maschinen verteilt wurde und dadurch Teile des Systems ausfallen können. Das hat zur Folge, dass die Implementierung bestimmter Schnittstellen nicht erreichbar ist. Dadurch, dass eine Schnittstelle durch eine nachfragende Komponente explizit referenziert wird, kann eine solche Komponente nicht korrekt arbeiten, wenn die Implementierung der Schnittstelle nicht erreichbar ist, obwohl die benötigten Informationen und Services vielleicht durch andere Schnittstellen, deren Implementierung durchaus zur Verfügung stehen, bereitgestellt werden könnten.

Dies führt zu der Überlegung, ob eine nachfragende Komponente anstelle der Referenzierung einer Schnittstelle eine Spezifizierung der Schnittstelle vornimmt, anhand derer eine angebotene Komponente, die diese Spezifikation erfüllt, gefunden werden kann.

Ein solches Vorgehen wird bei der testgetriebene Codesuche (testdriven codesearch - *TDCS*) verfolgt, welche als Basis für diese Arbeit herangezogen wird. Dabei stellt der Entwickler eine Menge von Suchparametern zusammen, die er an eine so genannte Source Engine übergibt. Die Suchparameter sind dabei jedoch stark an dem orientiert, was der Entwickler benötigt und weniger daran, was tatsächlich im Repository vorliegt. Diese Source Engine durchsucht anschließend ein Repository nach Komponenten (im weitesten Sinne), die zu den gestellten Suchparametern passen.

Die Suchergebnisse werden aufgelistet und der Entwickler entscheidet letztendlich explizit, wel-

che Komponente verwenden möchte. Die Verwendung der Komponente läuft dann jedoch auf eine Referenzierung dieser in der nachfragenden Komponente hinaus. Somit arbeiten die Source Engines also nicht zur Laufzeit des Systems, in dem die Komponenten verwendet werden sollen.

In dieser Arbeit soll eine solche Exploration jedoch zur Laufzeit erfolgen, sodass eine explizite Referenzierung der angebotenen Komponente nicht erfolgen muss. Dabei ist die Zeit als Ressource während der Suche nach einer passenden Komponente als knapp anzusehen. Aus diesem Grund werden in dieser Arbeit Heuristiken vorgeschlagen, die ein gezieltes Auffinden einer passenden Komponente ermöglichen und damit die Suche beschleunigen.

1.2 Aufbau dieser Arbeit

Zuerst wird in Kapitel 2 auf den aktuellen Forschungsstand zur *TDCS* eingegangen. Im Anschluss daran wird beschrieben, wie sich die *TDCS* auf einen Ansatz, in dem zur Laufzeit nach Komponenten gesucht wird, eingegangen, um so eine Abgrenzung zu den früheren Arbeiten zu schaffen.

In Kapitel 3 werden die einzelnen Schritte, die während der Exploration durchgeführt werden, sowie die zu evaluierenden Heuristiken formal beschrieben.

Kapitel 4 gibt einen kurzen Überblick über die Implementierung der in Kapitel 3 genannten Aspekte.

In Kapitel 5 werden die Untersuchungsergebnisse, die unter Anwendung der Heuristiken im Einzelnen und in Kombination zusammengetragen wurden, vorgestellt.

Die Auswertung dieser Ergebnisse erfolgt in Kapitel 6 zusammen mit einer kritischen Betrachtung des in der Arbeit vorgestellten Ansatzes, sowie einer kurzen Betrachtung möglicher Erweiterungen für diesen Ansatz.

Komplettiert wird die Arbeit durch eine kurze Zusammenfassung der Ergebnisse und einem

Ausblick in Kapitel 7.

Kapitel 2

Forschungsziel und Abgrenzung

2.1 Testgetriebene Codesuche

Die Idee der *TDCS* beruht im Grunde auf dem Ziel der Wiederverwendung von Software, welches 1992 von Krueger wie folgt beschrieben wurde: „*Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.*“ [Kru92] In der *TDCS* soll dieses Ziel in Verbindung mit dem Prozess der testgetriebenen Software-Entwicklung (testdriven development - *TDD*) erreicht werden. [Hum08]

Bei der *TDCS* werden die jeweiligen Anforderungen an den zu suchenden Source Code durch die Entwickler*innen spezifiziert. Diese werden im Abschluss verwendet, um den relevanten Source Code aus einem Repository zu ermitteln. Darauf aufbauend kann das jeweilige Tool den Entwickler*innen Vorschläge für die Wiederverwendung des bestehenden Codes unterbreiten.

Der Prozess der *TDCS* wurde von Hummel und Janjic grundlegend als Zyklus beschrieben und wie in Abbildung 2.1 dargestellt [HJ13].

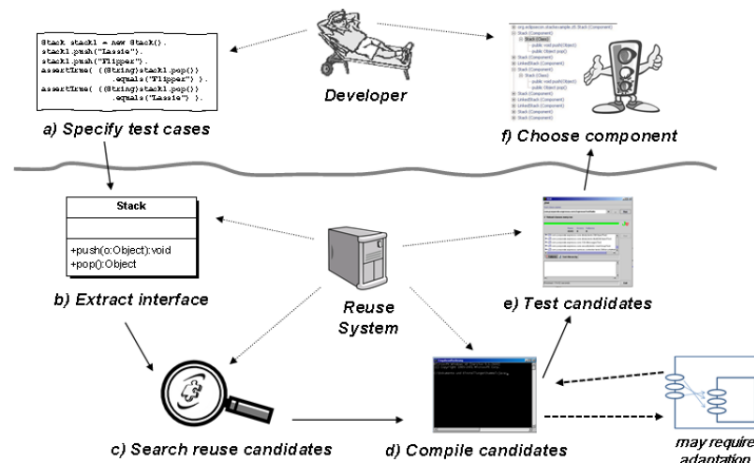


Abbildung 2.1: The testdriven reuse "cycle" [HJ13]

Hier spezifizieren die Entwickler*innen eine Menge von Testfällen (a) aus denen von eine Search Engine (in der Abbildung „Reuse System“ genannt) ein Interface extrahiert wird (b). Dieses Interface wird für die Suche nach Kandidaten, die zu diesem Interface passen, verwendet (c). Diese Kandidaten werden im Anschluss kompiliert, wobei mitunter eine Anpassung (Adaption) erfolgen muss, um das extrahierte Interface vollständig¹ zu erfüllen (d). Die letzte Aufgabe der Search Engine besteht dann im Test der kompilierten und mitunter adaptierten Kandidaten. Hierfür werden die von den Entwickler*innen in Schritt a spezifizierten Testfälle verwendet. Darauf aufbauend wird eine Liste von relevanten Kandidaten erarbeitet, aus der die Entwickler*innen eine Kandidaten zur weiteren Verwendung auswählen können (f).

Zu beachten ist, dass die Entwickler*innen bei diesem Ansatz die zu verwendende Komponente letztendlich selbst auswählen müssen. Der Ansatz eignet sich also nicht zum Einsatz während der Laufzeit des Systems. Weiterhin ist zu erwähnen, dass das Interface, welches für die Suche verwendet wird, aus den von den Entwickler*innen spezifizierten Testfällen extrahiert wird. Im Rahmen dieser Arbeit soll das Interface jedoch vorgegeben werden², da so der Explorationsalgorithmus und die beschriebenen Heuristiken besser nachvollzogen werden können.

¹ein (angepasster) Kandidat erfüllt ein Interface vollständig, wenn er zu jeder Methode des Interfaces eine passende Methode anbietet.

²Die Vorgabe erfolgt durch die Entwickler*innen.

Der Ansatz zur *TDCS* wurde bereits in [BNL⁺06] von Bajaracharya et al. verfolgt. Diese Gruppe entwickelte eine Search Engine namens *Sourcerer*, welche die Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens *CodeGenie* entwickelt, welches Softwareentwickler*innen die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde erstmals der Begriff der *TDCS* etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummels [Hum08] ebenfalls eine Weiterentwicklung von *Sourcerer* veröffentlicht, welche unter dem Namen *Mero-base* bekannt ist und ebenfalls das Konzept der *TDCS* verfolgt.

In [Hum08] wurden in Bezug auf die *TDCS* drei weitere Voraussetzungen identifiziert, die in dem oben beschriebenen Zyklus nicht eindeutig erwähnt wurden:

1. Ein Software-Repository, in dem die wiederverwendbaren Softwareteile enthalten sind.
2. Ein Format für die Repräsentation dieser Softwareteile.
3. Ein Mechanismus, welcher in der Lage ist, das Repository zu durchsuchen.

In früheren Arbeiten wurden im Internet bestehende Code-Repositories als Software-Repository verwendet. Die Repräsentation konnte dabei je nach Repository unterschiedliche Formen haben. Damit ist die Darstellung gemeint, auf deren Basis die Kandidaten aus dem Repository ermittelt werden. Daher muss es möglich sein sowohl die Kandidaten als auch das Interface, welches von den Entwickler*innen spezifiziert oder aus den Testfällen extrahiert wurde, in dieser Form zu beschreiben. Die Mechanismen, die in bestehenden Arbeiten für die Suche verwendet wurden, sind ebenfalls vielfältig. Eine Auflistung der am häufigsten verwendeten Ansätze und eine kurze Erklärung ist in [HJ13] und [Hum08] zu finden.

In den derzeit jüngsten Arbeiten zu diesem Thema wurde versucht, den *TDCS*-Ansatz immer tiefer in den Entwicklungsprozess zu verankern (vgl. [KA18]). Darüber hinaus wurde versucht, die Komponenten, die den Entwickler*innen für die Wiederverwendung angeboten wurden, nach unterschiedlichen Kriterien zu sortieren. Dabei ist eine Vielzahl von Ranking Ansätzen entstanden, in denen die Abdeckung der funktionellen Anforderungen (bspw. in [SED16], [KA15]), oder die Abdeckung der nicht-funktionellen Anforderungen (bspw. in [KA16]) bestimmt und für die

Sortierung verwendet wird. Zu bemerken ist jedoch, dass das Ranking einen nachgelagerten Prozess darstellt³. Somit wird die Suche/Exploration der Search Engine durch ein solches Ranking nicht beschleunigt.

2.2 Testgetriebene Exploration von EJBs

Diese Arbeit legt den Fokus auf die Exploration von *Enterprise-Java-Beans* (*EJBs*). Hummel identifizierte *EJBs* bereits in [Hum08] als eine Client-Server-Architektur für Software-Systeme, welche die Kommunikation zwischen Komponenten (so genannten *Beans*), die auf physikalisch unterschiedlichen Maschinen laufen, koordinieren bzw. unterstützen können (vgl. auch [DeM05]). Dazu wird das jeweilige Software-System auf einem Applikationsserver deployed, der die EJB-Spezifikation [DeM05] erfüllt.

Bei einer *Bean* handelt es sich grundlegend um eine Java-Klasse, deren Struktur außerhalb dieser Klasse spezifiziert wurde. Seit der Version 3.0 der EJB-Spezifikation kann die Struktur durch ein Java-Interface spezifiziert werden. In früheren Versionen erfolgt dies in einer XML-Datei. [DeM05]

Die *Beans* können über einen *EJB-Container* abgerufen werden. Zu diesem Zweck publiziert der *EJB-Container* die Interfaces der deployten *Beans*, sodass diese auf den Clients über *JNDI* oder *Dependency Injection* zur Verfügung stehen [DeM05].

Bezogen auf die in [Hum08] beschriebenen Voraussetzungen für die *TDCS* wird der *EJB-Container* in dieser Arbeit als Software-Repository angesehen. Die einzelnen Softwareteile werden in Form von Java-Interfaces repräsentiert. Und der Mechanismus zum Durchsuchen des Repositories wird durch die Publikation der Java-Interfaces der *EJBs* durch den *EJB-Container* bereitgestellt.

Der Prozess für die Exploration von *EJBs* unterscheidet sich leicht von dem aus Abbildung 2.1. Während in der Beschreibung von Hummer das Interface aus den Testfällen extrahiert

³In Bezug auf Abbildung 2.1 würde das Ranking zwischen Schritt e und f eingeordnet werden.

wird, müssen die Entwickler*innen hier das Interface selbst entwerfen. Dies erfolgt in der Form eines Java-Interfaces. Die Tests werden als Java-Klassen deklariert, die über ihre Methoden eine Validierung der *EJBs* erlauben.

Darüber hinaus muss klargestellt werden, dass die Exploration der *EJBs* zur Laufzeit durchgeführt wird, da anderenfalls der *EJB-Container* nicht zur Verfügung steht. Somit muss die Exploration während der Laufzeit gestartet werden können. Zu diesem Zweck wird eine *Explorationskomponente* in das System integriert.

Der Prozess für den beschriebenen Ansatz kann somit in einen Implementierungsprozess und einen Explorationsprozess, welcher zur Laufzeit durchgeführt wird, eingeteilt werden.

In Abbildung 2.2 ist der Implementierungsprozess aufgezeigt:

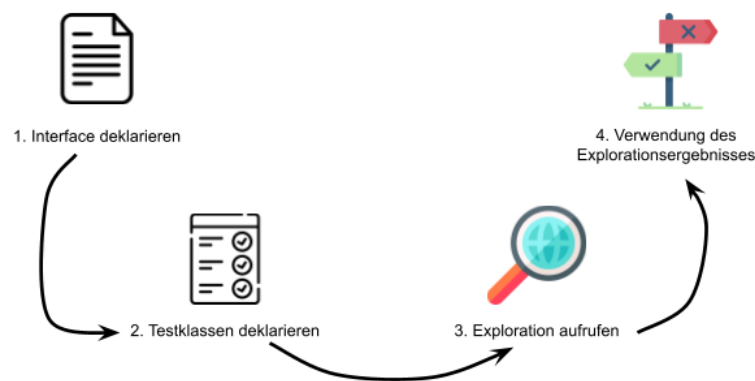


Abbildung 2.2: Implementierungsprozess

Wie bereits erwähnt, deklarieren die Entwickler*innen das Interface (1) und die Testklassen (2). Im dritten Schritt erfolgt der Aufruf der Explorationskomponente, wodurch zur Laufzeit der Explorationsprozess (siehe Abbildung 2.3) gestartet wird. Das Ergebnis des Explorationsprozesses kann in Form des deklarierten Interfaces weiter verwendet werden. Allerdings muss der Entwickler auch davon ausgehen, dass durch die Explorationskomponente keine passende Komponente ermittelt werden konnte.

Die folgende Abbildung stellt den Explorationsprozess dar:

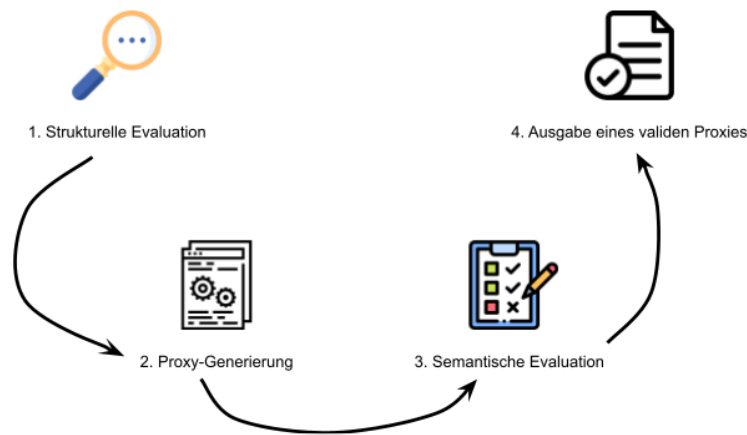


Abbildung 2.3: Explorationsprozess

Hier wird zuerst eine *strukturelle Evaluation* auf Basis des vorgegebenen Interfaces und der vom *EJB-Container* publizierten Interfaces durchgeführt. Dieser Schritt ist mit dem Suchen der Kandidaten aus Abbildung 2.1 vergleichbar. Auf Basis der Kandidaten, die bei der *strukturellen Evaluation* ermittelt wurden, werden im zweiten Schritt *Proxies* generiert, durch die die Methodenaufrufe auf dem vorgegebenen Interface an die jeweiligen Kandidaten delegiert werden. Dies ist mit Schritt d aus Abbildung 2.1 zu vergleichen. Im nächsten Schritt (*semantische Evaluation*) werden - analog zu Schritt e aus Abbildung 2.1 - die vorgegebenen Testklassen verwendet, um eben jene *Proxies* zu validieren. Sofern ein valider *Proxy* gefunden wurde, wird dieser im 4. Schritt von der Explorationskomponente zurückgegeben.

Wie in Kapitel 3 noch beschrieben wird, besteht die Möglichkeit, dass das vorgegebene Interface erst durch eine Kombination einiger *EJBs* gänzlich erfüllt werden kann. Eine solche Kombination der *Beans* soll über ein *Proxy-Objekt* erreicht werden, welches bei der Exploration im Anschluss an die *strukturelle Evaluation* generiert wird. Das Proxy-Objekt muss dann zum einen in der Lage sein, die Methodenaufrufe wie in den Methoden-Signaturen den vorgegebenen Interfaces entgegenzunehmen und diese dann zum anderen an die entsprechende *Bean*, die eine dazu passende Methode bereitstellt, delegieren.

Da die Exploration wie oben beschrieben zur Laufzeit durchgeführt wird, sollte die Suche ab-

gebrochen werden, sofern ein *Proxy* erfolgreich validiert wurde. Anderenfalls kann es bspw. zu unnötigen Timeouts laufender Transaktionen kommen. Um darüber hinaus ein schnelles Auffinden eines validen *Proxies* zu gewährleisten, können bei der *semantischen Evaluation* Heuristiken verwendet werden, welche die Generierung von *Proxies* und deren Validierung beschleunigen. Die vorliegende Arbeit dient hauptsächlich der Evaluation solcher Heuristiken.

Kapitel 3

Theoretische Grundlagen

In den folgenden Abschnitten wird der Explorationsprozess und dessen Grundlagen formal beschrieben sowie zum besseren Verständnis mit entsprechenden Beispielen untermalt. Die einzelnen Schritte des Prozesses finden sich damit in den Überschriften der Abschnitte wieder.

3.1 Strukturelle Evaluation

In Anlehnung an [Hum08] werden die *EJBs* auf der Basis des Signature-Matching Ansatzes ermittelt. Dieser Ansatz wurde ursprünglich von Zaremski und Wing [ZW95] beschrieben. Er basiert darauf, dass lediglich die Methoden-Signaturen der Typen (Klassen bzw. Interfaces) miteinander abgeglichen (gematcht) werden.

Zu diesem Zweck wird eine Struktur zur Deklaration von Typen in Abschnitt 3.1.1 vorgegeben, die eine abstrakte Darstellung von Klassen oder Interfaces, darstellen. Darüber hinaus werden in den genannten Abschnitt die Eigenschaften der Typen sowie Funktionen vorgestellt, die für den weiteren Verlauf der Arbeit von Belang sind.

Der Abgleich der Methoden-Signaturen dieser Typen erfolgt in Anlehnung an [ZW95] auf der Basis von Matchern, welche in Abschnitt 3.1.2 genauer beschrieben werden. Einige der dort beschriebenen Matcher basieren auf denen aus [ZW95]. Andere basieren auf Überlegungen aus [Hum08].

3.1.1 Struktur für die Definition von Typen

Die Typen werden in einer Bibliothek L in folgender Form deklariert:

Regel	Erläuterung
$L ::= TD^*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \mid RD$	Eine Typdefinition kann entweder die Definition eines <i>provided Typen</i> (PD) oder eines <i>required Typen</i> (RD) sein.
$PD ::=$ $\text{provided } T \text{ extends } T'$ $\{FD^*MD^*\}$	Die Definition eines <i>provided Typen</i> besteht aus dem Namen des Typen T , dem Namen des Super-Typs T' von T sowie mehreren Feld- und Methodendeklarationen.
$RD ::= \text{required } T \{MD^*\}$	Die Definition eines <i>required Typen</i> besteht aus dem Namen des Typen T sowie mehreren Methodendeklarationen.
$FD ::= T \ f$	Eine Felddeklaration besteht aus dem Namen des Feldes f und dem Namen seines Typs T .
$MD ::= T' \ m(T_1, \dots, T_n)$	Eine Methodendeklaration besteht aus dem Namen der Methode m , den insgesamt n Namen der Parameter-Typen T_1 bis T_n und dem Namen des Rückgabe-Typs T' .

Tabelle 3.1: Struktur für die Definition einer Bibliothek von Typen

Zudem sei die Relation $<$ auf Typen durch folgende Regeln definiert:

$$\frac{\text{provided } T \text{ extends } T' \in L}{T < T'}$$

$$\frac{\text{provided } T \text{ extends } T'' \in L \wedge T'' < T'}{T < T'}$$

Darüber hinaus seien folgende Funktionen definiert:

$$\begin{aligned}
 felder(T) &:= \left\{ T \ f \mid T \ f \text{ ist Felddeklaration von } T \right\} \\
 feldTyp(f, T) &:= T' \mid T' \ f \in felder(T) \\
 ret(T' \ m(T''_1, \dots, T''_n)) &:= T' \\
 params(T'' \ m(T'_1, \dots, T'_n)) &:= \{T'_1, \dots, T'_n\} \\
 methoden(T) &:= \left\{ T'' \ m(T'_1, \dots, T'_n) \mid T'' \ m(T'_1, \dots, T'_n) \text{ ist Methodendeklaration von } T \right\}
 \end{aligned}$$

Listing 3.1 zeigt die Deklaration der Bibliothek *Example* als Beispiel für eine Bibliothek mit *required* und *provided Typen*¹.

```

provided Fire extends Object{}
provided FireState extends Object{
    boolean isActive
}

provided Injured extends Object{
    void heal(Medicine med)
}

provided FireFighter extends Object{
    FireState extinguishFire(Fire fire)
}

provided InverseDoctor extends Object{
    void heal( Medicine med, Patient pat )
}

required PatientMedicalFireFighter {
    void heal( Patient patient,
              MedCabinet med )
    boolean extinguishFire( ExtFire fire )
}

provided ExtFire extends Fire{}
provided Medicine extends Object{
    String getDescription()
}

provided Patient extends Injured{
    String getName()
}

provided Doctor extends Object{
    void heal( Patient pat, Medicine med )
}

provided MedCabinet extends Object{
    Medicine med
}

required MedicalFireFighter {
    void heal( Injured injured,
              MedCabinet med )
    boolean extinguishFire( ExtFire fire )
}

```

Listing 3.1: Bibliothek *Example* von Typen

¹Zu beachten ist, dass die Bibliothek auf die im JDK enthaltenen Typen aufbaut. Daher ist davon auszugehen, dass Typen wie `Object` oder `boolean` bereits als *provided Typen* definiert sind.

3.1.2 Definition der Matcher

Ein Matcher definiert das Matching eines Typs T zu einem Typ T' oder einer Methode m zu einer Methode m' durch die asymmetrische Relation \Rightarrow (auch Matchingrelation genannt)². Im Folgenden werden die Matchingrelationen der spezifischen Matcher über ein Subskript differenziert.

ExactTypeMatcher

Der *ExactTypeMatcher* definiert das Matching von einem Typ T zu sich selbst her (vgl. [ZW95]). Die dazugehörige Matchingrelation \Rightarrow_{exact} wird durch folgende Regel beschrieben:

$$\overline{T \Rightarrow_{exact} T}$$

GenTypeMatcher

Der *GenTypeMatcher* definiert das Matching von einem Typ T zu einem Typ T' mit $T > T'$ (vgl. [ZW95]). Die dazugehörige Matchingrelation \Rightarrow_{gen} wird durch folgende Regel beschrieben:

$$\frac{T > T'}{T \Rightarrow_{gen} T'}$$

SpecTypeMatcher

Der *SpecTypeMatcher* definiert im Verhältnis zum *GenTypeMatcher* das Matching in die entgegengesetzte Richtung (vgl. [ZW95]). Die dazugehörige Matchingrelation \Rightarrow_{spec} wird durch folgende Regel beschrieben:

$$\frac{T < T'}{T \Rightarrow_{spec} T'}$$

Die oben genannten Matchingrelationen werden für die Definition weiterer Matcher zusam-

² $T \Rightarrow T'$
Gesprochen: T matcht T'

mengefasst, wodurch sich die Matchingrelation $\Rightarrow_{internCont}$ ergibt:

$$\frac{T \Rightarrow_{exact} T' \vee T \Rightarrow_{gen} T' \vee T \Rightarrow_{spec} T'}{T \Rightarrow_{internCont} T'}$$

Die folgenden Matcher definieren das Matching für so genannte Wrapper-Typen zu den Typen der in ihnen enthaltenen Attribute. Die Idee für solche Matcher fand in [Hum08] zwar Erwähnung, jedoch erfolgte dort keine formale Beschreibung. Das Ziel dieser Matcher ist es bspw. die Typen `boolean` und `FireState` aus der in Listing 3.1 deklarierten Bibliothek *ExampleLe* zu matchen.

ContentTypeMatcher

Der *ContentTypeMatcher* definiert das Matching von einem Typ T zu einem Typ T' , wobei T' ein Feld enthält, auf dessen Typ T'' der Typ T über die Matchingrelation $\Rightarrow_{internCont}$ gematcht werden kann.

Die dazugehörige Matchingrelation $\Rightarrow_{content}$ wird durch folgende Regel beschrieben:

$$\frac{\exists T'' f \in felder(T') : T \Rightarrow_{internCont} T''}{T \Rightarrow_{content} T'}$$

So würde für die Typen `boolean` und `FireState` aus der Bibliothek *ExampleLe* (siehe 3.1) gelten:

$$\text{boolean} \Rightarrow_{content} \text{FireState}$$

ContainerTypeMatcher

Der *ContainerTypeMatcher* definiert im Verhältnis zum *ContentTypeMatcher* das Matching für die entgegengesetzte Richtung.

Die dazugehörige Matchingrelation $\Rightarrow_{\text{container}}$ wird durch folgende Regel beschrieben:

$$\frac{\exists T'' f \in \text{felder}(T) : T'' \Rightarrow_{\text{internCont}} T'}{T \Rightarrow_{\text{container}} T'}$$

So gilt für die Typen `FireState` und `boolean` aus der Bibliothek *ExampLe* (siehe 3.1):

$$\text{FireState} \Rightarrow_{\text{container}} \text{boolean}$$

Zur Definition des letzten Matchers werden die Matchingrelationen der oben genannten Matcher wiederum zusammengefasst. Dabei entsteht die Matchingrelation $\Rightarrow_{\text{internStruct}}$, welche durch folgende Regel beschrieben wird:

$$\frac{T \Rightarrow_{\text{internCont}} T' \vee T \Rightarrow_{\text{container}} T' \vee T \Rightarrow_{\text{content}} T'}{T \Rightarrow_{\text{internStruct}} T'}$$

StructuralTypeMatcher

Der *StructuralTypeMatcher* definiert das Matching von einem *required Typ R* zu einem *provided Typ P* auf der Basis der Methoden-Signaturen der beiden Typen.

Somit soll bspw. ein Matching zwischen dem Typ `MedicalFireFighter` und dem den `FireFighter` aus der Bibliothek *ExampLe* (siehe Listing 3.1) gematcht werden. Als ein weiteres Beispiel, bezogen auf die Typen aus der Bibliothek *ExampLe*, kann das Matching zwischen dem Typ `MedicalFireFighter` und dem Typ `Doctor` angebracht werden.

Damit ein *required Typ R* auf einen *provided Typ P* über den *StrukturalTypeMatcher* gematcht werden kann, muss mindestens eine Methode aus *R* zu einer Methode aus *P* gematcht werden (Signature-Matching). Ein Matching der Methoden liegt dann vor, wenn sowohl die Rückgabe- als auch die Parameter-Typen dieser beiden Methoden miteinander gematcht werden können (vgl. [ZW95]).

Wie in [ZW95] soll die Reihenfolge, in der die Parameter in der jeweiligen Methode dekla-

riert wurden, keine Rolle spielen. Ausgehend von den Parameter-Typen der beiden Methoden als Mengen, muss eine der Mengen also so umsortiert werden, dass die Parameter-Typen aus beiden Mengen an der jeweils gleichen Position miteinander gematcht werden können. Die möglichen umsortierte Mengen von Parameter-Typen einer Methode m auf die dies in Bezug auf die Menge der Parameter-Typen einer Methode m' zutrifft, werden über die Funktion *matchingParams* beschrieben:

$$\text{matchingParams}(m, m') := \left\{ \{mP_1, \dots, mP_n\} \left| \begin{array}{l} \{P_1, \dots, P_n\} = \text{params}(m) \wedge \\ \forall i \in \{1, \dots, n\} : mP_i \in \text{params}(m') \wedge \\ mP_i \Rightarrow_{\text{internStruct}} P_i \end{array} \right. \right\}$$

Das Matching zweier Methoden m und m' wird durch die Relation $\Rightarrow_{\text{method}}$ über folgende Regel beschrieben:

$$\frac{\text{ret}(m) \Rightarrow_{\text{internStruct}} \text{ret}(m') \wedge \text{matchingParams}(m, m') \neq \emptyset}{m \Rightarrow_{\text{method}} m'}$$

Die Menge der gematchten Methoden aus R in P wird darauf aufbauend durch folgende Funktion beschrieben:

$$\text{structM}(R, P) := \left\{ m \left| \begin{array}{l} m \in \text{methoden}(R) \wedge \\ \exists m' \in \text{methoden}(P) : m \Rightarrow_{\text{method}} m' \end{array} \right. \right\}$$

Die Matchingrelation für den *StructuralTypeMatcher* wird durch folgende Regel beschrieben:

$$\frac{\text{structM}(R, P) \neq \emptyset}{R \Rightarrow_{\text{struct}} P}$$

3.1.3 Ergebnis der strukturellen Evaluation

Die Exploration wird für einen *required Typ* durchgeführt. Bei der *strukturellen Evaluation* sollen Mengen von *provided Typen* ermittelt werden, deren Methoden in Kombination zu jeder Methode des *required Typ* ein Matching aufweisen. Die Mengen von *provided Typen* innerhalb einer Bibliothek L für die dies in Bezug auf ein *required Typ* R zutrifft, wird über die Funktion *cover* beschrieben.

$$cover(R, L) := \left\{ \begin{array}{l} \{T_1, \dots, T_n\} \mid \begin{array}{l} T_1 \in L \wedge \dots \wedge T_n \in L \wedge \\ methoden(R) = structM(R, T_1) \cup \\ \dots \cup structM(R, T_n) \wedge \\ \forall T \in \{T_1, \dots, T_n\} : structM(R, T) \neq \emptyset \end{array} \end{array} \right\}$$

Die *provided Typen* innerhalb dieser Mengen werden im nächsten Schritt des Explorationsprozesses als *Target-Typen* bezeichnet und als Basis für die Generierung der Proxies für den *required Typ* *R* verwendet.

Beispiel 1 Sei folgende Bibliothek *L* gegeben.

```
provided Come extends Object{
    String hello()
    String goodMorning()
}

provided Leave extends Object{
    String bye()
}

required Greeting{
    String hello()
    String bye()
}
```

Über die Funktion *cover* werden folgende *Target-Typen* für die Generierung von Proxies für den *required Typ* *Greeting* ermittelt.

$$cover(Greeting, L) = \{\{Come\}, \{Leave, Come\}\}$$

3.2 Generierung der Proxies auf Basis von Matchern

Ein *Proxy* ist ein Objekt, das stellvertretend für ein anderes Objekt verwendet wird und den Zugang - in diesem Fall den Methodenaufruf - auf dieses Objekt kontrolliert (vgl. [ES13]). Dadurch ist es dem *Proxy* möglich, die Methodenaufrufe an andere Objekte zu delegieren. Diese Eigenschaft wird sich in dieser Arbeit zunutze gemacht, um einen Aufruf einer Methode, die in

einem Typ (*required* oder *provided Typ*) deklariert wurde, an eine Methode zu delegieren, die in einem anderen *provided Typ* deklariert wurde.

Dabei wird zwischen einem *Source-Typen* und einem oder mehrerer *Target-Typen* unterschieden. Als *Source-Typ* wird immer der Typ bezeichnet, für den der *Proxy* generiert und stellvertretend eingesetzt wird. Bei den *Target-Typen* handelt es sich um die *provided Typen* an deren Methoden die Methodenaufrufe delegiert werden.

Zur Beschreibung der Generierung von *Proxies* wird im Folgenden zuerst vorgegeben, wie sich ein *Proxy* deklarieren lässt. Darauf aufbauend werden die Generatoren, die in Abhängigkeit des Matchings zwischen *Source-* und *Target-Typen* Anwendung finden, beschrieben.

3.2.1 Struktur für die Definition von Proxies

Die Grammatikregeln für die Deklaration eines *Proxies* sind Tabelle 3.2 zu entnehmen.

Regel	Erläuterung
$PROXY ::=$ $\text{proxy for } T$ $\text{with } [P_1, \dots, P_n]$ $\{MDEL_1, \dots, MDEL_k\}$	Ein <i>Proxy</i> wird für ein Typ T als <i>Source-Typ</i> mit einer Menge von <i>provided Typen</i> $P = \{P_1, \dots, P_n\}$ als <i>Target-Typen</i> , einer Menge von Methoden-Delegationen erzeugt.
$MDEL ::=$ $CALLM \rightarrow DELM$	Eine <i>Methodendelegation</i> besteht aus einer <i>aufgerufenen Methode</i> und aus einem <i>Delegationsziel</i> .
$CALLM ::=$ $REF.m(CP_1, \dots, CP_n) : CR$	Eine <i>aufgerufene Methode</i> besteht aus dem Namen der Methode m , dem Rückgabetypp CR und einer Menge von Parametertypen $\{CP_1, \dots, CP_n\}$.
$DELM ::=$ $REF.n(DP_1, \dots, DP_n) : DR$	Die erste Variante eines <i>Delegationsziels</i> besteht aus dem Namen der <i>Delegationsmethode</i> n , dem Rückgabetypp DR und einer Menge von Parametertypen $\{DP_1, \dots, DP_n\}$.
$DELM ::=$ $\text{posModi}(I_1, \dots, I_n)$ $REF.n(DP_1, \dots, DP_n) : DR$	Die zweite Variante eines <i>Delegationsziels</i> besteht aus einer Menge von Indizes $\{I_1, \dots, I_n\}$, einer <i>Referenz</i> , dem Namen der <i>Delegationsmethode</i> n , dem Rückgabetypp DR und einer Menge von Parametertypen $\{DP_1, \dots, DP_n\}$.
$DELM ::= \text{err}$	Die dritte Variante eines <i>Delegationsziels</i> enthält keine weiteren Bestandteile. Das Terminal err weist darauf hin, dass die Delegation innerhalb des Proxies nicht möglich ist und zu einem Fehler führt.
$REF ::= P_i$	Die erste Variante einer <i>Referenz</i> besteht aus einem Typ P_i .
$REF ::= P_i.f$	Die zweite Variante einer <i>Referenz</i> besteht aus einem Typ P_i und einem Feldnamen f .

Tabelle 3.2: Grammatikregeln mit Erläuterungen für die Deklaration eines Proxies

Es handelt sich dabei um Produktionsregeln einer Attributgrammatik. Die dazugehörigen Attribute sind der Tabelle 3.3 zu entnehmen. Dazu sei zusätzlich festgelegt, dass die Notation $NT.*$ in der Spalte *Attribute* eine Key-Value-Liste aller Attribute des Nonterminals NT beschreibt, wobei der Attributname als Key und dessen Wert als Value innerhalb der Liste verwendet wird. Weiterhin sei ein Attribut, dass in der Spalte *Attribute* zu einem Nonterminal nicht aufgeführt ist, mit dem Wert *none* belegt.

Regel	Attribute
$PROXY ::=$ $\text{proxy for } T$ $\text{with } [P_1, \dots, P_n]$ $\{MDEL_1, \dots, MDEL_k\}$	$\text{type} = T$ $\text{targets} = [P_1, \dots, P_n]$ $\text{dels} = [MDEL_1.*, \dots, MDEL_k.*]$ $MDEL_1.\text{call}.\text{field} = \dots = MDEL_k.\text{call}.\text{field}$ $MDEL_1.\text{del}.\text{field} = \dots = MDEL_k.\text{del}.\text{field}$
$MDEL ::=$ $CALLM \rightarrow DELM$	$\text{call} = CALLM.*$ $\text{del} = DELM.*$
$CALLM ::=$ $REF.m(CP_1, \dots, CP_n) : CR$	$\text{source} = REF.\text{mainType}$ $\text{delType} = REF.\text{delType}$ $\text{name} = m$ $\text{paramTypes} = [CP_1, \dots, CP_n]$ $\text{returnType} = CR$ $\text{field} = REF.\text{field}$ $\text{paramCount} = n$
$DELM ::=$ $REF.n(DP_1, \dots, DP_n) : DR$	$\text{target} = REF.\text{mainType}$ $\text{delType} = REF.\text{delType}$ $\text{posModi} = [0, \dots, n - 1]$ $\text{name} = n$ $\text{paramTypes} = [DP_1, \dots, DP_n]$ $\text{returnType} = DR$ $\text{field} = REF.\text{field}$
$DELM ::= \text{posModi}(I_1, \dots, I_n)$ $REF.n(DP_1, \dots, DP_n) : DR$	$\text{target} = REF.\text{mainType}$ $\text{delType} = REF.\text{delType}$ $\text{posModi} = [I_1, \dots, I_n]$ $\text{name} = n$ $\text{paramTypes} = [DP_1, \dots, DP_n]$ $\text{returnType} = DR$ $\text{field} = REF.\text{field}$
$DELM ::= \text{err}$	
$REF ::= P$	$\text{mainType} = P$ $\text{field} = \text{self}$ $\text{delType} = P$
$REF ::= P.f$	$\text{mainType} = P$ $\text{field} = f$ $\text{delType} = \text{feldTyp}(f, P)$

Tabelle 3.3: Grammatikregeln mit Attributen für die Deklaration eines Proxies

3.2.2 Delegation von Methoden im Proxy

Ein *Proxy* bietet alle Methoden des *Source-Typen* an. Einige dieser Methoden werden an eine Methode delegiert, die von einem *Target-Typ* des *Proxies* angeboten wird. Eine solche Delegation wird durch eine *Methoden-Delegation* (siehe Tabelle 3.3 Nonterminal *MDEL*) definiert.

Beispiel So beschreibt die folgende *Methoden-Delegation*, dass die Methode `extinguishFire`, die vom *Source-Typ* `Patient` - und damit auch vom *Proxy* - angeboten wird, an die Methoden `heal`, die der *Target-Typ* `Injured` anbietet, delegiert wird.

```
Patient.heal(Medicine):void → Injured.heal(Medicine):void
```

Listing 3.2: Einfache Methoden-Delegation

Die Delegation einer *aufgerufenen Methode* an ein *Delegationsziel*, erfolgt in drei Schritten.

1. Parameterübergabe

Dabei werden die Parameter, mit denen die vom *Proxy* angebotene Methode aufgerufen wird, an die *Delegationsmethode* des *Delegationsziels* übergeben. Dabei sind zwei Dinge zu beachten. Zum einen müssen die Typen der übergebenen Parameter zu den Typen der von der *Delegationsmethode* erwarteten Parameter passen. Zum anderen muss die Reihenfolge, in der die Parameter übergeben wurden, an die erwartete Reihenfolge der *Delegationsmethode* angepasst werden. (siehe auch Funktion `matchingParams` aus Abschnitt 3.1.2)

2. Ausführung

Dieser Schritt meint die Durchführung der *Delegationsmethode* mit den übergeben Parametern aus Schritt 1. Dies schließt auch die Ermittlung des korrekten Rückgabewertes der *Delegationsmethode* ein.

3. Übergabe des Rückgabewertes

Ähnlich wie bei der Parameterübergabe, muss auch der Rückgabewert, der bei der Ausführung in Schritt 2 ermittelt wurde, an die *aufgerufene Methode*, die vom *Proxy* angeboten wird, übergeben werden. Hier muss ebenfalls sichergestellt werden, dass die beiden Rückgabetypen der beiden Methoden zueinander passen.

Die Delegation aus dem oben genannten Beispiel kann schematisch wie in Abbildung 3.1 dargestellt werden. Die Übergabe der Parameter- und Rückgabewerte wird durch die gestrichelten Pfeile symbolisiert.

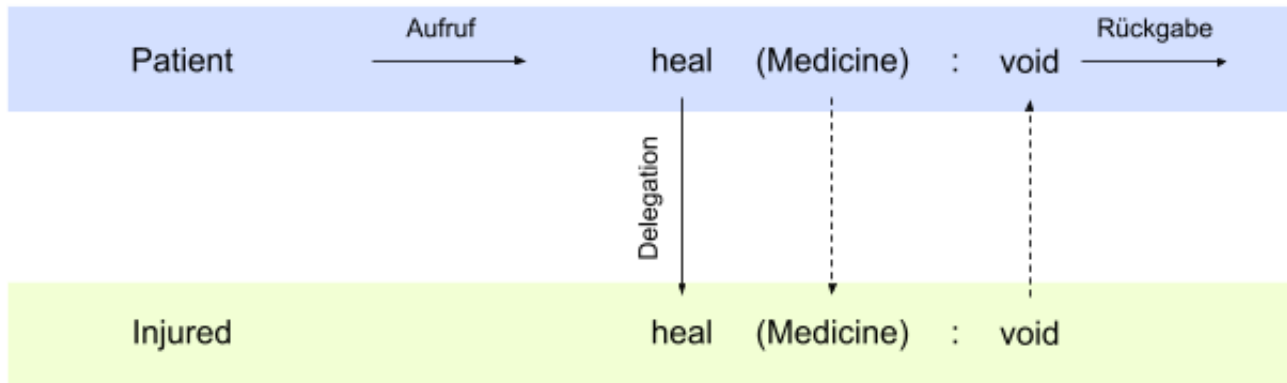


Abbildung 3.1: Delegation der Methode `heal`

In diesem Beispiel sind sowohl die Parameter- als auch die Rückgabe-Typen der *aufgerufenen Methode* und der *Delegationsmethode* identisch. Weiterhin spielt die Reihenfolge der Parameter in diesem Beispiel keine Rolle, da es nur einen Parameter gibt. Daher stellt die Übergabe der Parameter- und Rückgabewerte kein Problem dar.

Folgendes Beispiel soll zeigen, wie mit unterschiedlichen Reihenfolgen bzgl. der Parameter bei einer *Methoden-Delegation* umzugehen ist.

Beispiel Die *Methoden-Delegation* aus Listing 3.2.2 ist ein Beispiel für einen solchen Fall. Hier wird die *aufgerufene Methode* `heal` mit den Parametern `Patient` und `MedCabinet` aus dem Typ `PatientMedicalFireFighter` an die gleichnamige Methode aus dem Typ `InverseDoctor` delegiert. Die *Delegationsmethode* verwendet zwar identische Parameter-Typen, aber die Reihenfolge, in der die Parameter übergeben werden, ist unterschiedlich.

```

PatientMedicalFireFighter.heal(Patient, MedCabinet):void → posModi(1,0)
InverseDoctor.heal(MedCabinet, Patient):void
  
```

Listing 3.3: Methoden-Delegation mit Parametern in unterschiedlicher Reihenfolge

Um die Reihenfolge der Parameter aus dem ursprünglichen Aufruf zu variieren, wird das Schlüsselwort `posModi` verwendet. Dort werden eine Reihe von Indizes angegeben. Die Anzahl der angegebenen Indizes muss mit der Anzahl der Parameter übereinstimmen. Ein Index beschreibt die Position des in der *aufgerufenen Methode* angegebenen Parameter. Weiterhin spielt die Reihenfolge der Indizes eine wichtige Rolle. Diese ist mit der Reihenfolge der Parameter der *Delegationsmethoden* gleichzusetzen.

So wird in dem o.g. Beispiel der erste Parameter der *aufgerufenen Methoden* (Index = 0) der *Delegationsmethode* als zweiter Parameter übergeben. Dementsprechend wird der zweite Parameter der *aufgerufenen Methode* (Index = 1) der *Delegationsmethode* als erster Parameter übergeben (siehe Abbildung 3.2).

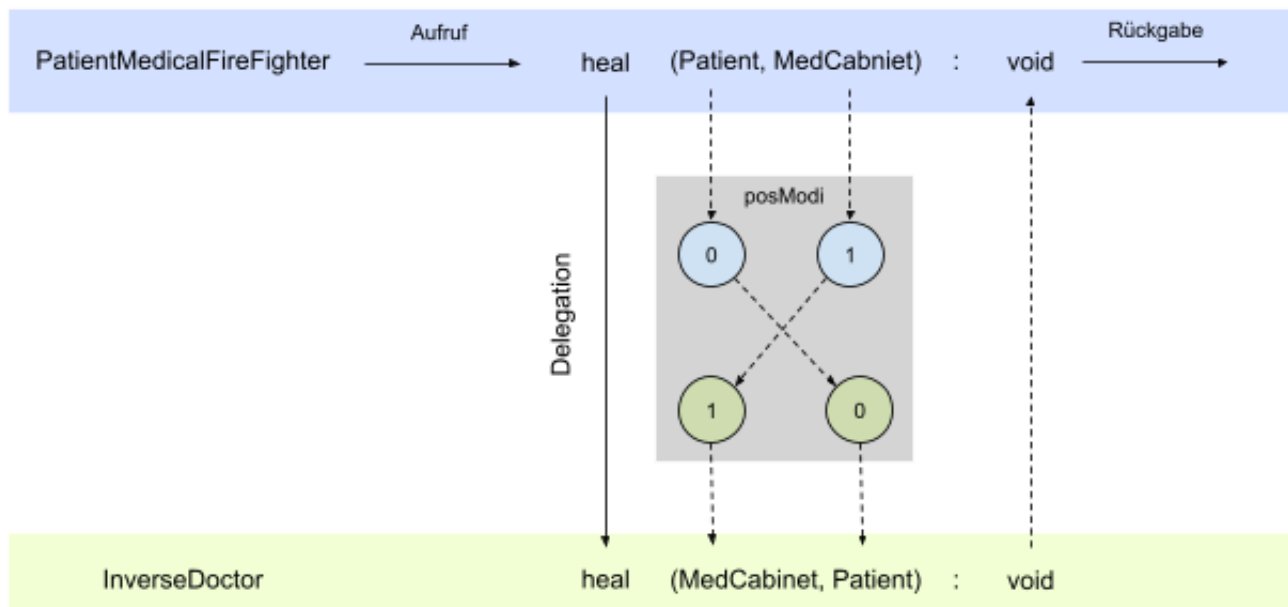


Abbildung 3.2: Delegation der Methode `heal` mit Parametern in unterschiedlicher Reihenfolge

Dass identische Typen keine Probleme bei der Übergabe zwischen *aufgerufener Methode* und *Delegationsmethode* darstellen, wurde in den oben genannten Beispielen gezeigt. Darüber hinaus können Typen aber auch dann ohne Probleme übergeben werden, wenn sie sich aufgrund des Substitutionsprinzips austauschen lassen. Daher kann ein Typ T anstelle eines Typs T'

verwendet werden, sofern $T \leq T'$ gilt.

Folgendes Beispiel soll zeigen, wie mit übergebenen Typen umzugehen ist, die nicht ohne Probleme übergeben werden können.

Beispiel In folgendem Listing ist eine *Methoden-Delegation* aufgeführt, bei der sowohl die Parameter- als auch die Rückgabe-Typen der *aufgerufenen Methode* und der *Delegationsmethode* nicht auf Basis des Substitutionsprinzips übergeben werden können.

```
MedicalFireFighter.extinguishFire(ExtFire):boolean →
FireFigher.extinguishFire(Fire):FireState
```

Listing 3.4: Methoden-Delegation mit Typkonvertierung

In einem solchen Fall müssen die Parameter-Typen der *aufgerufenen Methoden* in die Parameter-Typen der *Delegationsmethode* konvertiert werden. Analog dazu muss der Rückgabotyp der *Delegationsmethode* in den Rückgabotyp der *aufgerufenen Methoden* konvertiert werden. Die Konvertierung wird dabei über die Generierung eines *Proxies* erzielt.

Angenommen, eine Funktion $proxies(S, T)$ beschreibt eine Menge von *Proxies*, mit S als *Source-Typ* und T als Menge der *Target-Typen*, dann müssten bezogen auf die *Methoden-Delegation* aus Listing 3.4 für die Parameter einer der *Proxies* aus der Menge $proxies(\text{Fire}, \{\text{ExtFire}\})$ an die *Delegationsmethode* übergeben werden. Nach der Ausführung der *Delegationsmethode* müsste aus dem Rückgabewert ein *Proxy* aus der Menge $proxies(\text{boolean}, \{\text{FireState}\})$ erzeugt werden und an die *aufgerufene Methode* als Rückgabewert übergeben werden. Der Sachverhalt wird in Abbildung 3.3 schematisch dargestellt. Die grauen Kästen symbolisieren die Generierung eines *Proxies*.

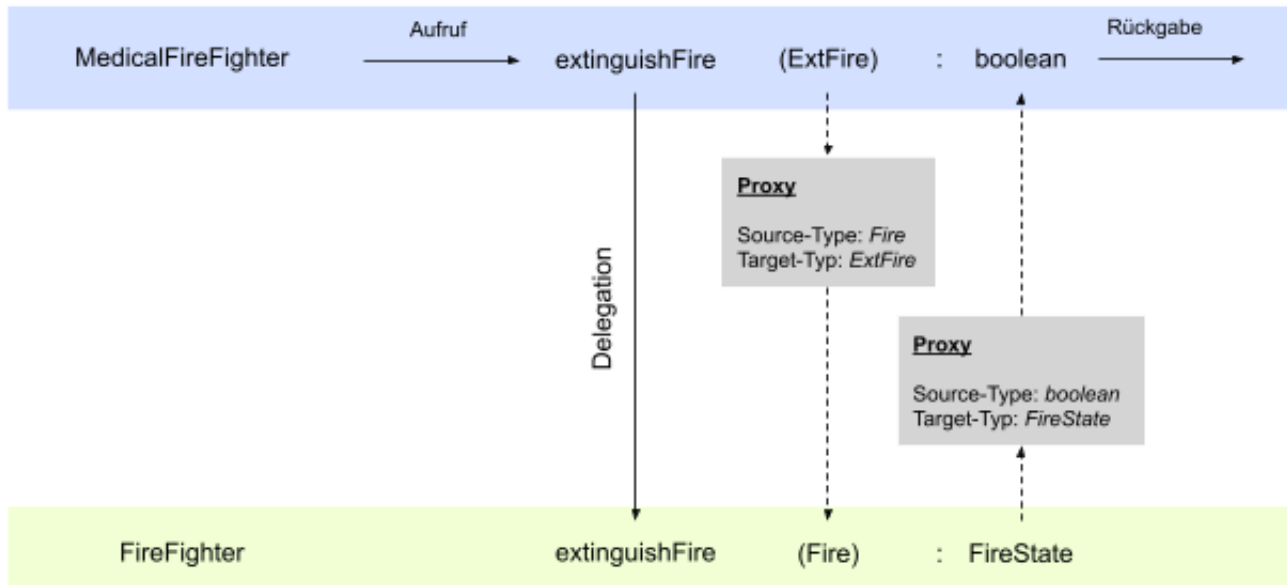


Abbildung 3.3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

Wie die *Proxies* generiert werden, wird im folgenden Abschnitt beschrieben.

3.2.3 Generierung von Proxies

Wie im Abschnitt 3.2.1 bereits angedeutet, soll die Menge der *Proxies* für einen *Source-Typ* S und einer Menge von *Target-Typen* T über die Funktion $proxies(S, T)$ beschrieben werden.

In Abhängigkeit von dem Matching zwischen dem *Source-Typ* und den *Target-Typen* werden unterschiedliche Arten von *Proxies* generiert. Für die unterschiedlichen Proxy-Arten gibt es ebenfalls Funktionen, die eine Menge von *Proxies* zu einem *Source-Typen* S und einer Menge von *Target-Typen* T beschreiben.

In den folgenden Abschnitten werden diese Funktionen für die einzelnen Proxy-Arten beschrieben. Dabei ist davon auszugehen, dass die *Proxies* eine allgemeine Struktur haben, die in Abschnitt 3.2.1 aufgeführt ist. Um die Regeln für die Generierung der *Proxies* zu beschreiben, soll davon ausgegangen werden, dass jedes Listen-Attribut ($NT.*$) aus Tabelle 3.3 ein Attribut `len` enthält in dem die Anzahl der Elemente abgelegt ist, die sich in dieser Liste befinden.

Sub-Proxy

Die Voraussetzung für die Erzeugung eines *Sub-Proxies* vom Typ *S* (*Source-Typ*) aus einem *Target-Typ* *T* ist $S \Rightarrow_{spec} T$. Damit ist der *SpecTypeMatcher* der *Basis-Matcher* für den *Sub-Proxy*.

Beispiel Als Beispiel soll der Typ *Patient* als *Source-Typ* und der Typ *Injured* als *Target-Typ* verwendet werden. Da $Patient \Rightarrow_{spec} Injured$ gilt, kann ein *Sub-Proxy* für diese Konstellation erzeugt werden. Der resultierende *Sub-Proxy* ist in Listing 3.5 aufgeführt. Der abstrakte Syntaxbaum (AST) mit den dazugehörigen Attributen ist Abbildung 3.4 zu entnehmen.³

```
proxy for Patient with [Injured]{
    Patient.heal(Medicine):void → Injured.heal(Medicine):void
    Patient.getName():String → err
}
```

Listing 3.5: Sub-Proxy für Patient

Ein *Proxy* bietet alle Methoden an, die auch von dessen *Source-Typ* angeboten werden. Sofern für eine angebotene Methode keine *Methodendelegationen* innerhalb des *Proxies* existiert, wird diese Methode so ausgeführt, wie sie innerhalb des *Source-Typen* implementiert wurde. Anderenfalls beschreiben die *Methodendelegationen* innerhalb eines *Proxies*, was beim Aufruf der entsprechenden Methode passiert. So wird ein Aufruf der Methode `heal` an die Methode `heal` aus dem *Target-Typ* delegiert. Ein Aufruf der Methode `getName` hingegen führt zu einem Fehler, weil keine passende *Delegationsmethode* zur Verfügung steht.

Weiterhin beschreibt der *Sub-Proxy* aus dem Beispiel auch, dass der Aufruf der Methode `getName` zu einem Fehlschlag führt. Dies ist auf eine Problematik zurückzuführen, die auch bei einem Down-Cast auftritt. Hierbei soll ein Objekt eines Super-Typs anstelle eines Objektes eines Sub-Typs verwendet werden. Das Objekt des Sub-Typs bietet jedoch mitunter Methoden an, die im Super-Typ nicht deklariert wurden. Folglich können diese Methoden nicht ausgeführt werden, was jedoch häufig erst zur Laufzeit auffällt.

³Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

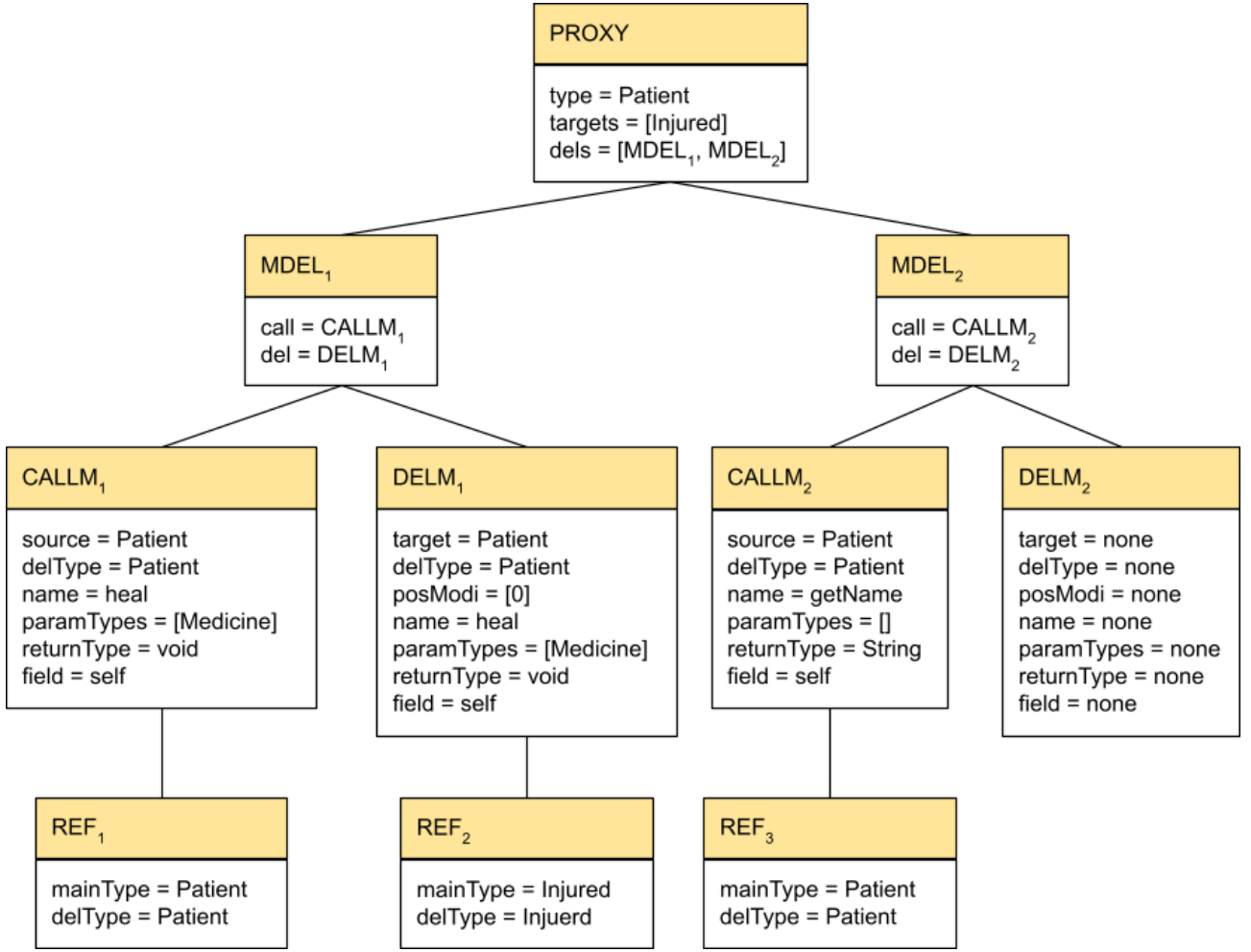


Abbildung 3.4: AST für das Beispiel zum Sub-Proxy

Formalisierung Formal wird ein *Sub-Proxy* durch die Regeln beschrieben, die im Folgenden vorgestellt werden. Ein *Sub-Proxy* enthält genau einen *Target-Typ*. Für einen *Proxy* P wird dieser Sachverhalt durch die folgende Regel dargestellt.

$$\frac{|P.targets| = 1 \wedge T \in P.targets}{targets_{single}(P, T)}$$

Darüber hinaus kann für jeden *Proxy* P (egal welcher Art) festgehalten werden, dass dessen Attribut **type** immer mit einem bestimmten Typen S übereinstimmt. Damit wird ausgesagt,

dass S der *Source-Typ* des *Proxies* P ist.

$$\frac{P.type = S}{proxy(P, S)}$$

Die Unterschiede der einzelnen Proxy-Arten lassen sich in den *Methoden-Delegationen* finden. Eine *Methoden-Delegationen* besteht aus einer linken Seite - die *aufgerufene Methode* - und einer rechten Seite - die *Delegationsmethode*. Bevor die Beziehungen zwischen diesen beiden Seiten beschrieben werden, werden zuerst die separaten Eigenschaften der beiden Seiten beschrieben.

So muss die *aufgerufene Methode* immer im Typ aus dem Attribut `call.delType` deklariert sein.

$$\frac{\exists T' \ m(T) \in methoden(MD.call.delType) : MD.call.name = m}{callDecl(MD)}$$

Bezüglich des Feldes `delType` der *Delegationsmethode* gilt ähnliches.

$$\frac{\exists T' \ m(T) \in methoden(MD.del.delType) : MD.del.name = m}{delDecl(MD)}$$

Darüber hinaus müssen die Attribute `source` und `delType` der *aufgerufenen Methode* einer *Methoden-Delegation* MD mit dem *Source-Typ* des *Proxies* belegt sein. Dazu müssen die beiden folgenden Regeln gelten.

$$\frac{MD.call.source = MD.call.delType}{callDelegationType_{simple}(MD)}$$

$$\frac{MD.call.source = P.type}{sourceType(MD, P)}$$

Damit ist auch automatisch gewährleistet, dass das Attribut `field` im Attribut `call` der *Methoden-Delegation* mit dem Wert `self` belegt ist (vgl. Tabelle 3.3).

Ähnliches gilt für die Attribute `field` und `mainType` im Attribut `del` der *Methoden-Delegation*

MD . Hierbei muss der Wert des Attributs **mainType** jedoch mit dem *Target-Typ* des *Proxies* übereinstimmen.

$$\frac{MD.del.target \in P.targets}{targetType(MD, P)}$$

$$\frac{MD.del.delType = MD.del.target}{delDelegationType_{sub}(MD)}$$

Damit ist wiederum automatisch gewährleistet, dass das Attribut **field** im Attribut **del** der *Methoden-Delegation* mit dem Wert **self** belegt ist (vgl. Tabelle 3.3).

Die Regeln für die linke Seite einer *Methoden-Delegation* MD innerhalb eines *Proxies* P können damit in folgender Regel zusammengefasst werden:

$$\frac{callDecl(MD) \wedge callDelegationType_{simple}(MD, P) \wedge sourceType(MD, P)}{call_{simple}(MD, P)}$$

Dementsprechend dazu können auch die Regeln für die rechte Seite einer *Methoden-Delegation* MD innerhalb eines *Proxies* P zusammengefasst werden:

$$\frac{delDecl(MD) \wedge targetType_{sub}(MD, P) \wedge delDelegationType_{sub}(MD)}{del_{simple}(MD, P)}$$

Die oben genannten Regeln beschreiben die notwendigen Bedingungen der beiden Seiten einer *Methoden-Delegation* innerhalb eines *Sub-Proxies*. Die Bedingungen, die für eine gesamte *Methoden-Delegation* MD eines *Sub-Proxies* P gilt, werden durch die folgenden beiden Regeln beschrieben.

$$\frac{MD.call.name = MD.del.name}{methodMatch_{simple}(MD)}$$

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{simple}}{delegation_{sub}(MD, P)}$$

Zu beachten ist jedoch, dass ein *Sub-Proxy* P auch fehlschlagende *Methoden-Delegationen* MD enthalten kann. Somit gilt ebenfalls:

$$\frac{MD.del.name = \mathbf{none}}{methodErr(MD)}$$

$$\frac{call_{simple}(MD, P) \wedge methodErr(MD)}{delegation_{sub}(MD, P)}$$

Die Regel für alle *Methoden-Delegationen* eines *Sub-Proxies* P lässt sich aufbauend auf den oben genannten Regeln, wie folgt darstellen.

$$\frac{\forall MD \in P.dels : delegation_{sub}(MD, P)}{delegations_{sub}(P)}$$

Die Menge der *Sub-Proxies*, die mit dem *Source-Typ* S und dem *Target-Typ* T erzeugt werden, wird darauf aufbauend durch die folgende Funktion beschrieben.

$$proxies_{sub}(S, T) := \left\{ P \left| \begin{array}{l} proxy(P, S) \wedge \\ targets_{sub}(P, T) \wedge \\ delegations_{sub}(P) \end{array} \right. \right\}$$

Content-Proxy

Die Voraussetzung für die Erzeugung eines *Content-Proxies* vom Typ S aus einem *Target-Typ* T ist $S \Rightarrow_{content} T$. Damit ist der *ContentTypeMatcher* der Basis-Matcher für den *Content-Proxy*.

Beispiel Als Beispiel sollen die Typen **Medicine** und **MedCabinet** verwendet werden, welche ein Matching der Form **Medicine** $\Rightarrow_{content}$ **MedCabinet** aufweisen. Daher kann ein *Content-Proxy* für diese Konstellation erzeugt werden. Ein resultierender *Content-Proxy* ist in folgendem Listing aufgeführt.

```
proxy for Medicine with [MedCabinet]{
    Medicine.getDescription():String → MedCabinet.med.getDescription():String
}
```

Listing 3.6: Content-Proxy für Medicine

Durch die Methoden-Delegation dieses *Content-Proxies* wird die Methode `getDescription` an das Feld `med` des *Target-Typen* `MedCabinet` delegiert.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.5 zu entnehmen.⁴

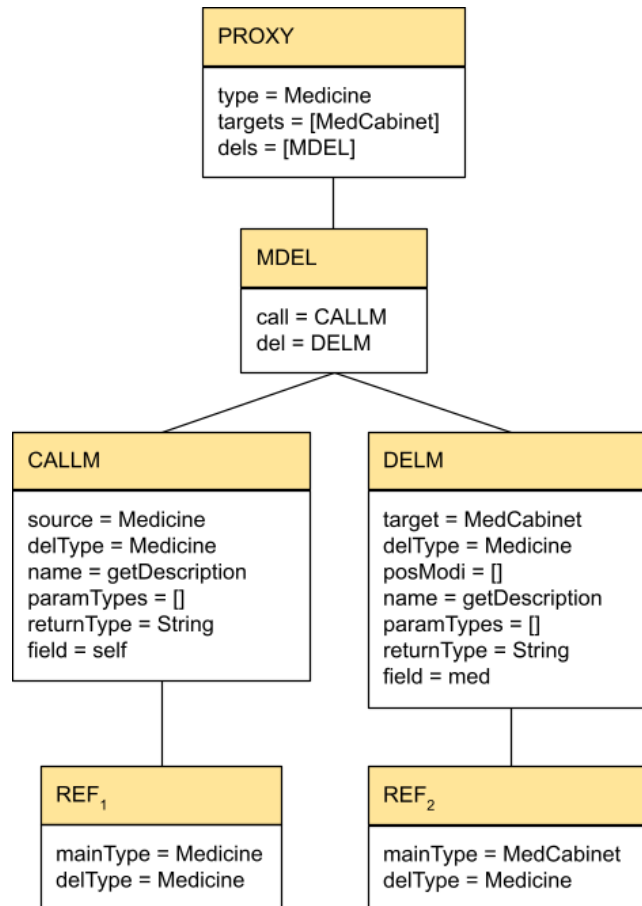


Abbildung 3.5: AST für das Beispiel zum Content-Proxy

Formalisierung Formal wird ein *Content-Proxy* durch die Regeln beschrieben, die im Folgenden vorgestellt werden.

Ein *Content-Proxy* enthält, wie auch der *Sub-Proxy*, genau einen *Target-Typ*. Ebenfalls identisch zum *Sub-Proxy* sind die Bedingungen hinsichtlich der *aufgerufenen Methoden* in den einzelnen

⁴Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

Methoden-Delegationen.

In den *Delegationsmethoden* einer einzelnen *Methoden-Delegation* MD muss zwar ebenfalls das Attribut **target** in den *Target-Typen* des *Proxies* P enthalten sein ($targetType(MD, P)$), allerdings muss im *Content-Proxy* darüber hinaus jenes Attribut **target** ein Matching zum Typen im Attribut **delType** aufweisen.

$$\frac{P.type \Rightarrow_{internCont} MD.del.delType}{delDelegationType_{content}(MD, P)}$$

Folglich werden auch die Eigenschaften einer *Delegationsmethode* in einer *Methoden-Delegation* MD im *Content-Proxy* durch eine andere Regel beschrieben als beim *Sub-Proxy*:

$$\frac{delDecl(MD) \wedge targetType(MD, P) \wedge delDelegationType_{content}(MD, P)}{del_{content}(MD, P)}$$

Darauf aufbauend ergibt sich wiederum folgende Regel für eine gesamte *Methoden-Delegation* MD innerhalb eines *Content-Proxies* P .

$$\frac{call_{simple}(MD, P) \wedge del_{content}(MD, P) \wedge methodMatch_{simple}}{delegation_{content}(MD, P)}$$

Da auch ein *Content-Proxy* fehlschlagende *Methoden-Delegationen* enthalten kann, muss ebenfalls gelten:

$$\frac{call_{simple}(MD, P) \wedge methodErr(MD)}{delegation_{content}(MD, P)}$$

Für die Gesamtheit der *Methoden-Delegationen* innerhalb eines *Content-Proxies* P muss dann gelten:

$$\frac{\forall MD \in P.dels : delegation_{content}(MD, P)}{delegations_{content}(P)}$$

Die Menge der *Content-Proxies*, die mit dem *Source-Typ* S und dem *Target-Typ* T erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{content}(S, T) := \left\{ P \mid \begin{array}{l} proxy(P, S) \wedge \\ targets_{single}(P, T) \wedge \\ delegations_{content}(P) \end{array} \right\}$$

Container-Proxy

Die Voraussetzung für die Erzeugung eines *Container-Proxies* vom Typ S aus einem *Target-Typ* T ist $S \Rightarrow_{container} T$. Damit ist der *ContainerTypeMatcher* der Basis-Matcher für den *Container-Proxy*.

Beispiel Als Beispiel werden wiederum die Typen **Medicine** und **MedCabinet** verwendet, welche ein Matching der Form **MedCabinet** $\Rightarrow_{container}$ **Medicine** aufweisen. Daher kann ein *Content-Proxy* für diese Konstellation erzeugt werden. Ein resultierender *Content-Proxy* ist in folgendem Listing aufgeführt.

```
proxy for MedCabinet with [Medicine]{
    MedCabinet.med.getDescription():String → Medicine.getDescription():String
}
```

Listing 3.7: Container-Proxy für MedCabniet

Durch die *Methoden-Delegation* dieses *Container-Proxies* findet eine Delegation nur dann statt, wenn die Methoden **getDescription** auf dem Feld **med** des *Source-Typ* aufgerufen wird. Diese wird dann an den *Target-Typen* **MedCabniet** delegiert.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.6 zu entnehmen.⁵

Formalisierung Formal wird ein *Container-Proxy* durch die Regeln beschrieben, die im Folgenden vorgestellt werden.

Ein *Container-Proxy* enthält, wie die vorher beschriebenen *Proxies*, genau einen *Target-Typ*. Die Eigenschaften der *Delegationsmethoden* innerhalb der einzelnen *Methoden-Delegationen*

⁵Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

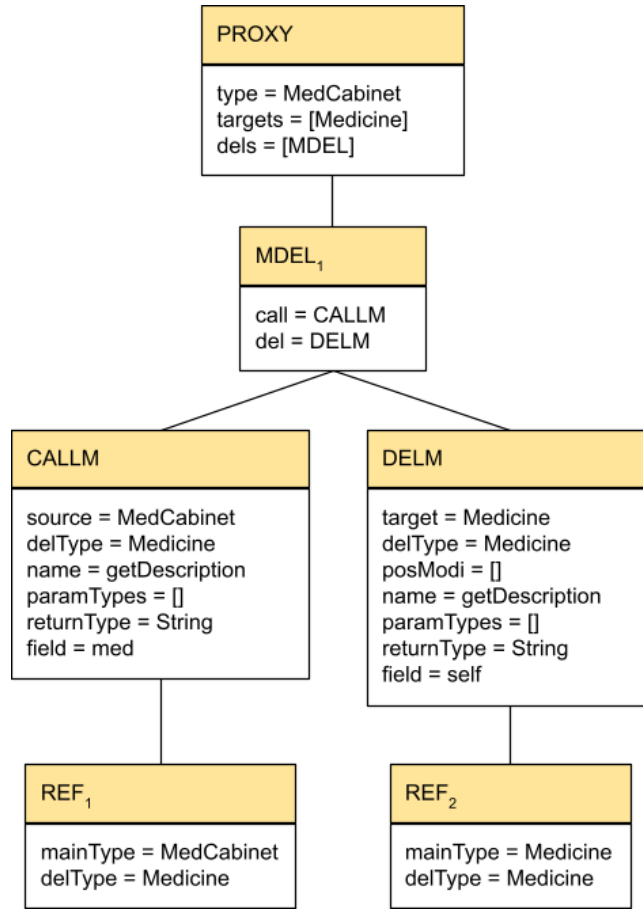


Abbildung 3.6: AST für das Beispiel zum Container-Proxy

gleichen denen aus dem *Sub-Proxy*.

In der *aufgerufenen Methode* einer einzelnen *Methoden-Delegation MD* muss das Attribut **source** ebenfalls wie im *Sub-Proxy* mit dem *Source-Typen* des *Proxies P* übereinstimmen ($sourceType(MD, P)$), allerdings muss das Attribut **delType** der *aufgerufenen Methode* ein Matching zu einem der *Target-Typen* des *Proxies* aufweisen:

$$\frac{\exists T \in P.targets : MD.call.delType \Rightarrow_{internCont} T}{callDelegationType_{container}(MD)}$$

Folglich werden auch die Eigenschaften einer *aufgerufenen Methode* in einer *Methoden-*

Delegation MD im *Container-Proxy* durch eine andere Regel beschrieben als beim *Sub-Proxy*:

$$\frac{callDecl(MD) \wedge callDelegationType_{container}(MD, P) \wedge sourceType(MD, P)}{call_{container}(MD, P)}$$

Darauf aufbauend ergibt sich wiederum folgende Regel für eine gesamte *Methoden-Delegation MD* innerhalb eines *Container-Proxies P*.

$$\frac{call_{container}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{simple}}{delegation_{container}(MD, P)}$$

Da auch ein *Container-Proxy* fehlschlagende *Methoden-Delegationen* enthalten kann, muss ebenfalls gelten:

$$\frac{call_{container}(MD, P) \wedge methodErr(MD)}{delegation_{container}(MD, P)}$$

Für die Gesamtheit der *Methoden-Delegationen* innerhalb eines *Container-Proxies P* muss dann gelten:

$$\frac{\forall MD \in P.dels : delegation_{container}(MD, P)}{delegations_{container}(P)}$$

Die Menge der *Container-Proxies*, die mit dem *Source-Typ S* und dem *Target-Typ T* erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{container}(S, T) := \left\{ P \mid \begin{array}{l} proxy(P, S) \wedge \\ target_{single}(P, T) \wedge \\ delegations_{container}(P) \end{array} \right\}$$

Struktureller Proxy

Die Voraussetzung für die Erzeugung eines *strukturellen Proxies* vom *required Typ R* aus einem *Target-Typ T* ist $R \Rightarrow_{struct} T$. Damit ist der *StructuralTypeMatcher* der Basis-Matcher für den *strukturellen Proxy*.

Der *strukturelle Proxy* ist der einzige *Proxy*, der mit mehreren *Target-Typen* erzeugt werden

kann.

Beispiel Als Beispiel werden die Typen `MedicalFireFighter`, `Doctor` und `FireFighter` verwendet. Dabei ist `MedicalFireFighter` der *Source-Typ* des *Proxies* und die Menge der anderen beiden Typen bilden die *Target-Typen* des *Proxies*. Da der *Source-Typ* zu den *Target-Typen* ein Matching der Form `MedicalFireFighter` \Rightarrow_{struct} `FireFighter` bzw. `MedicalFireFighter` \Rightarrow_{struct} `Doctor` aufweist, kann ein *struktureller Proxy* erzeugt werden. Ein solcher ist in folgendem Listing aufgeführt.

```
proxy for MedicalFireFighter with [Doctor, FireFighter]{
  MedicalFireFighter.heal(Patient, MedCabinet):void →
    Doctor.heal(Patient, Medicine):void
  MedicalFireFighter.extinguishFire(ExtFire):boolean →
    FireFighter.extinguishFire(Fire):FireState
}
```

Listing 3.8: Struktureller Proxy für MedicalFireFighter

In diesem Beispiel wird der Methodenaufruf der Methode `heal` auf dem *Proxy* an die Methode `heal` des Typs `Doctor` delegiert. Analog dazu würde ein Aufruf der Methode `extinguishFire` auf dem *Proxy* an die Methode `extinguishFire` des Typs `FireFighter` delegiert werden. Die Methoden stimmen jeweils strukturell überein.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.7 zu entnehmen.⁶

Formalisierung Ein *struktureller Proxy* wird formal durch die folgenden Regeln beschrieben.

Ein *struktureller Proxy* kann, wie bereits erwähnt, mehrere *Target-Typen* enthalten. Für jeden *Target-Typ* T muss dabei jedoch wenigstens eine *Delegationsmethode* im *Proxy* mit einem Attribut `target = T` existieren. Dadurch gilt die für einen *strukturellen Proxy* P :

$$\frac{\forall T \in P.targets : \exists MD \in P.dels : MD.del.target = T}{targets_{multi}(P, T)}$$

Für die *aufgerufene Methode* und die *Delegationsmethode* einer einzelnen *Methoden-Delegation* gelten im *strukturellen Proxy* dieselben Regeln wie für den *Sub-Proxy*. Die Namen der *auf-*

⁶Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

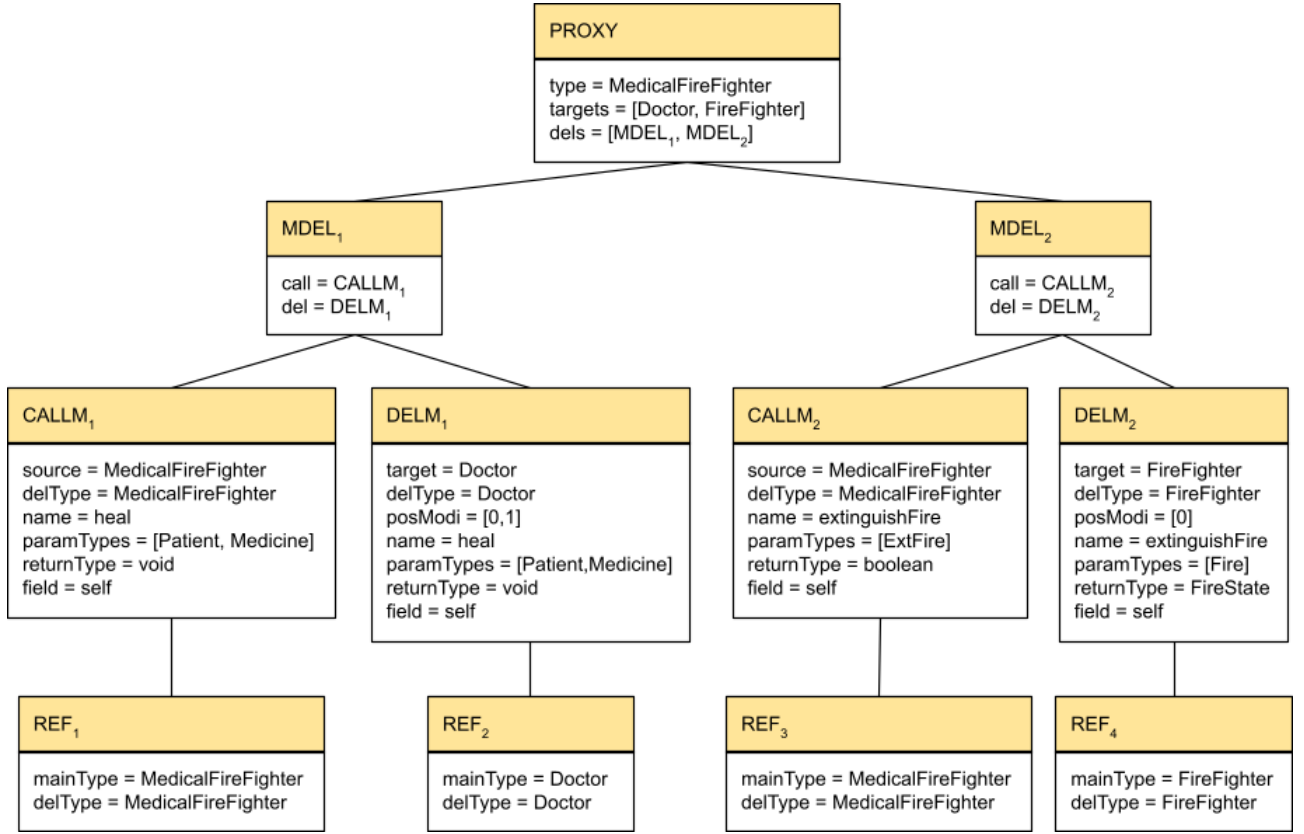


Abbildung 3.7: AST für das Beispiel zum strukturellen Proxy

gerufenen Methoden und der *Delegationsmethode* müssen dabei jedoch nicht übereinstimmen. Dafür müssen diese beiden Methoden jedoch ein strukturelles Matching aufweisen. Bezogen auf die Rückgabe-Typen einer *aufgerufenen Methode C* und einer *Delegationsmethode D* müssen daher folgende Regeln gelten.

$$\frac{D.returnType \Rightarrow_{internStruct} C.returnType}{returnMatch(C, D)}$$

Weiterhin muss für die Parameter-Typen gelten:

$$\frac{C.paramTypes[i] \Rightarrow_{internStruct} D.paramTypes[D.posModi[i]]}{posModiMatch(C, D, i)}$$

$$\frac{\forall i \in \{0, \dots, C.paramCount - 1\} : posModiMatch(C, D, i)}{paramsMatch(C, D)}$$

Das strukturelle Matching zwischen einer *aufgerufenen Methode* C und einer *Delegationsmethode* D kann darauf aufbauend wie folgt beschrieben werden.

$$\frac{returnMatch(C, D) \wedge paramsMatch(C, D)}{methodMatch_{struct}(C, D)}$$

Für eine einzelne *Methoden-Delegation* MD eines *strukturellen Proxies* P kann dann folgende Regel aufgestellt werden.

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{struct}(MD.call, MD.del)}{delegation_{struct}(MD, P)}$$

In einem *strukturellen Proxy* muss für jede Methode m des *Source-Typen* genau eine *Methoden-Delegation* mit der Methode m als *aufgerufene Methode* existieren:

$$\frac{|methoden(P.type)| = P.dels.len}{delegationCount_{struct}(P)}$$

Für die Gesamtheit der *Methoden-Delegationen* aus einem *strukturellen Proxy* P muss dann gelten:

$$\frac{delegationCount_{struct}(P) \wedge \forall MD \in P.dels : delegation_{struct}(MD, P)}{delegations_{struct}(P)}$$

Die Menge der *strukturellen Proxies*, die mit dem *Source-Typ* S und der Menge von *Target-Typen* T erzeugt werden, wird aufbauend auf den oben genannten Regeln durch die folgende Funktion beschrieben.

$$proxies_{struct}(S, T) := \left\{ P \left| \begin{array}{l} proxy(P, S) \wedge \\ targets_{multi}(P, T) \wedge \\ delegations_{struct}(P) \end{array} \right. \right\}$$

Allgemeine Generierung von Proxies

Die Proxy-Funktionen der einzelnen Proxy-Arten werden zur Beschreibung einer allgemeine Funktion für die Generierung der *Proxies* verwendet. Dazu sind die Proxy-Arten zusammen mit den dazugehörigen *Matchingrelationen* und den Namen der Funktionen zur Generierung des jeweiligen *Proxies* in Tabelle 3.4 noch einmal aufgeführt.

Proxy-Art	Matchingrelation	Funktionsname
Sub-Proxy	\Rightarrow_{spec}	$proxies_{sub}$
Content-Proxy	$\Rightarrow_{content}$	$proxies_{content}$
Container-Proxy	$\Rightarrow_{container}$	$proxies_{container}$
Struktureller Proxy	\Rightarrow_{struct}	$proxies_{struct}$

Tabelle 3.4: Proxy-Arten mit Matchingrelationen und Proxy-Funktionen

Die im Abschnitt 3.2.1 erwähnte Funktion $proxies(S, T)$ kann darauf aufbauend für einen *Source-Typ* S und eine Menge von *Target-Typen* T wie folgt beschrieben werden.

$$proxies(S, T) := \left\{ \begin{array}{ll} proxy_{sub}(S, T) & \text{wenn } |T| = 1 \wedge \\ & \forall T' \in T : S \Rightarrow_{spec} T' \\ \\ proxy_{content}(S, T) & \text{wenn } |T| = 1 \wedge \\ & \forall T' \in T : S \Rightarrow_{content} T' \\ \\ proxy_{container}(S, T) & \text{wenn } |T| = 1 \wedge \\ & \forall T' \in T : S \Rightarrow_{container} T' \\ \\ proxy_{struct}(S, T) & \text{wenn } |T| > 0 \wedge \\ & \forall T' \in T : S \Rightarrow_{struct} T' \end{array} \right\}$$

3.2.4 Anzahl struktureller Proxies innerhalb einer Bibliothek

Die Generierung der *strukturellen Proxies* für ein *required Typ* R aus der Bibliothek L erfolgt während des *Explorationsprozesses* auf Basis der Mengen von *provided Typen* aus $cover(R, L)$ (siehe Abschnitt 3.1.3). Mit einer Menge $TM \in cover(R, L)$ können durchaus mehrere *Pro-*

xies erzeugt werden. Das ist dann der Fall, wenn mehrere der Methoden, die in den *provided Typen* aus TM deklariert wurden, mit einer Methode aus R strukturell übereinstimmen, oder wenn für eine Methode m aus R und eine Methode m' aus einem der *Target-Typen* $|matchingParams(m, m')| > 1$ gilt (siehe Abschnitt 3.1.2).

Die Anzahl der *strukturellen Proxies* für einen *required Typ* R mit einer bestimmten Menge von *Target-Typen* ist somit von der Anzahl der Methoden abhängig, die in einem der *Target-Typen* deklariert wurden und strukturell mit den Methoden aus R übereinstimmen.

Die Menge der Methoden eines *provided Typs* T , die strukturell mit einer Methode m übereinstimmen, wird über die Funktion $structM_{target}$ beschrieben.

$$structM_{target}(m, T) := \left\{ m' \mid m' \in methoden(T) \wedge m \Rightarrow_{method} m' \right\}$$

Darauf aufbauend wird die Menge der Methoden einer Menge von *provided Typen* TM , die strukturell mit einer Methode m übereinstimmen, über die Funktion $structM_{targetset}$ beschrieben.

$$structM_{targetset}(m, TM) := \left\{ m' \mid \exists T \in TM : m' \in structM_{target}(m, T) \right\}$$

Beispiel 2 Aufbauend auf dem vorherigen Beispiel 1 ergeben sich für die Menge der *Target-Typen* $\{\text{Leave}, \text{Come}\}$ und die beiden Methoden des *required Typs* **Greeting** folgende Mengen von übereinstimmenden Methoden über die Funktion $structM_{targetset}$:

$$\begin{aligned} structM_{targetset}(String \text{ hello}(), \{\text{Leave}, \text{Come}\}) &= \left\{ \begin{array}{l} String \text{ hello}(), \\ String \text{ goodMorning}(), \\ String \text{ bye}() \end{array} \right\} \\ structM_{targetset}(String \text{ bye}(), \{\text{Leave}, \text{Come}\}) &= \left\{ \begin{array}{l} String \text{ hello}(), \\ String \text{ goodMorning}(), \\ String \text{ bye}() \end{array} \right\} \end{aligned}$$

Sei R ein *required Typ* und TM eine Menge von *provided Typen* innerhalb einer Bibliothek

L mit $TM \in \text{cover}(R, L)$. Dann bildet die Funktion structMSets die Menge von Mengen der Methoden aus den Elementen aus TM ab, die mit jeweils einer Methode aus R gematcht werden können.

$$\text{structMSets}(R, TM) := \left\{ M \mid \begin{array}{l} \exists m \in \text{methoden}(R) : \\ M = \text{structM}_{\text{targetset}}(m, TM) \end{array} \right\}$$

Für die Bildung eines *Proxies* wird aus jedem Element der Menge $\text{structMSets}(R, TM)$ genau ein Element als Delegationsmethode verwendet.

Beispiel 3 Ausgehend von Beispiel 2 lassen sich die folgenden vier Proxies mit den *Target-Typen* *Leave* und *Come* erzeugen.

```

proxy Greeting with [Come, Leave]{
    Greeting.hello():String → Come.hello():String
    Greeting.bye():String → Leave.bye():String
}

proxy Greeting with [Come, Leave]{
    Greeting.hello():String → Come.goodMorning():String
    Greeting.bye():String → Leave.bye():String
}

proxy Greeting with [Come, Leave]{
    Greeting.hello():String → Leave.bye():String
    Greeting.bye():String → Come.hello():String
}

proxy Greeting with [Come, Leave]{
    Greeting.hello():String → Leave.bye():String
    Greeting.bye():String → Come.goodMorning():String
}

```

Die Anzahl aller möglichen *strukturellen Proxies* für ein *required Typ* R aus einer Menge von *Target-Typen* TM sei näherungsweise über die Funktion $\text{proxyCount}(R, TM)$ beschrieben. Dass es sich hierbei lediglich um eine Annäherung handelt liegt daran, dass eine Methode dm mit $dm \in M_1 \cup \dots \cup M_n$ und $\{M_1, \dots, M_n\} = \text{structMSets}(R, TM)$ innerhalb eines Proxy maximal einmal als Delegationsmethode verwendet werden darf. Es ist jedoch möglich, dass es zwischen

den Mengen M_1, \dots, M_n Überschneidungen gibt (siehe vorheriges Beispiel). Darüber hinaus blenden die oben genannten Funktionen zur Ermittlung der matchenden Methoden die Anzahl der für die *Methoden-Delegation* möglichen Parameter-Sortierungen aus *matchingParams* aus.

Unter der Annahme, dass es nur eine mögliche Sortierung der Parameter für die matchenden Methoden gibt, gilt:

$$proxyCount(R, TM) \leq \prod_{i=1}^n |structM_{targetset}(m_i, TM)| \left| \left\{ \begin{array}{c} m_1, \\ \dots, \\ m_n \end{array} \right\} \right| = methoden(R)$$

Durch mehrere Möglichkeiten der Sortierung der Parameter würde sich dieser Wert nochmals erhöhen. Auf eine genaue Formel wird an dieser Stelle verzichtet, da hier lediglich eine Vorstellung davon vermittelt werden soll, wie große die Anzahl der möglichen *strukturellen Proxies* mit einer bestimmten Mengen von *Target-Typen* ist.

Da innerhalb einer Bibliothek L mehrere Mengen von *Target-Typen* zur Bildung eines *Proxies* für einen *required Typ* R infrage kommen (siehe Funktion *cover*) muss die Anzahl der strukturellen Proxies über die Funktion *proxyCount* für alle Elemente aus *cover*(R, L) ermittelt und summiert werden. Die folgende Funktion beschreibt diesen Sachverhalt:

$$libProxyCount(R, L) = \sum_{i=1}^n proxyCount(R, c_i) \left| \left\{ \begin{array}{c} c_1, \\ \dots, \\ c_n \end{array} \right\} \right| = cover(R, L)$$

3.3 Semantische Evaluation

Das Ziel der *semantischen Evaluation* ist es, einen der *Proxies*, die aus den Mengen von *Target-Typen*, die im Rahmen der *strukturellen Evaluation* erzeugt werden können, hinsichtlich der vordefinierten Testfälle zu evaluieren. Da der gesamte *Explorationsprozess* zur Laufzeit des jeweiligen Programms durchgeführt wird, stellt sie hinsichtlich der nicht-funktionalen Anforderungen eine zeitkritische Komponente dar.

Da die Anforderungen an den gesuchten *Proxy* mit Bedacht spezifiziert werden müssen, ist es irrelevant, ob es mehrere *Proxies* gibt, die hinsichtlich der vordefinierten Testfällen positiv geprüft werden können. Es ist ausreichend lediglich ein *Proxy* zu finden, dessen Semantik zu positiven Ergebnissen hinsichtlich aller vordefinierten Testfälle führt.

3.3.1 Besonderheiten der Testfälle

Bei den vordefinierten Tests handelt es sich auf formaler Ebene um Typen, die eine `eval`-Methode mit der Struktur `boolean eval(proxy)` anbieten, welche einen *Proxy* als Parameter erwartet und ein Objekt vom Typ `boolean` zurückgibt. Weiterhin verfügt ein Test über ein Attribut `triedMethodCalls`, in dem eine Liste von Methodennamen, die bei der Durchführung der `eval`-Methode auf den *Proxies* aufgerufen wurden, hinterlegt ist.

Die Implementierung der `eval`-Methode ist an folgende Bedingungen geknüpft:

1. Vor dem Aufruf einer Methode auf dem als Parameter übergebenen *Proxy*, wird der Name dieser Methode in der Liste im Feld `triedMethodCalls` ergänzt.
2. Wenn der *Proxy* den Test besteht, wird der Wert `true` zurückgegeben. Anderenfalls wird der Wert `false` zurückgegeben.

Beispiel 4 In folgendem Listing 3.9 ist eine `eval`-Methode aufgeführt, die die oben genannten Bedingungen erfüllt. Es sei davon auszugehen, dass der als Parameter übergebene *Proxy* eine Methode mit der Struktur *Integer add(Integer, Integer)* anbietet.

```
1 function eval( proxy ){
2     res = 0
3     triedMethodCalls.add( "add" )
4     res = proxy.add(1, 1)
5     return res == 2;
6 }
```

Listing 3.9: Beispielhafte Implementierung einer `eval`-Methode

3.3.2 Algorithmus für die semantische Evaluation

Während des *Explorationsprozesses* soll aus den *provided Typen* in einer Bibliothek L zu einem vorgegebenen *required Type* R ein Proxy generiert und evaluiert werden. Die Mengen der *Target-Typen* auf deren Basis mehrere *Proxies* erzeugt werden können, wurden im Abschnitt 3.2.4 über $cover(R, L)$ beschrieben. Die in $cover(R, L)$ befindlichen Mengen können eine unterschiedliche Anzahl von *Target-Typen* enthalten. Die maximale Mächtigkeit einer Menge $T_i \in cover(R, L)$ ist gleich der Anzahl der Methoden in R .

$$maxTargets(R) := |methoden(R)|$$

In Bezug zur Funktion $cover$ gilt:

$$\forall T \in cover(R, L) : |T| \leq maxTargets(R)$$

Das in dieser Arbeit beschriebene Konzept basiert auf der Annahme, dass der gesamte Anwendungsfall - oder Teile davon - , der mit der vordefinierten Struktur (*required Typ*) und den vordefinierten Tests abgebildet werden soll, schon einmal genauso oder so ähnlich in dem gesamten System implementiert wurde. Aus diesem Grund kann für die *semantische Evaluation* davon ausgegangen werden, dass die erfolgreiche Durchführung aller relevanten Tests umso wahrscheinlicher ist, je weniger *Target-Typen* im *Proxy* enthalten sind.

Die Mengen innerhalb einer Menge T mit einer Mächtigkeit $a < 0$ seien durch folgende Funktion beschrieben:

$$targetSets(T, a) := \left\{ T_i \mid T_i \in T \wedge |T_i| = a \right\}$$

Ausgehend von einer Bibliothek L kann der Algorithmus für die *semantische Evaluation* der *Proxies*, die für einen *required Typ* R (Parameter **R**) mit den Mengen der *Target-Typen* $cover(R, L)$ (Parameter **T**) erzeugt werden können, und einer Menge von Tests (Parameter **tests**) über die Methode **semanticEval** wie folgt im Pseudo-Code beschrieben werden. Die globale Variable **passedTests** enthält dabei die Anzahl der für den aktuell zu überprüfenden *Proxy* erfolgreich durchgeführten Tests. Außerdem sei davon auszugehen, dass die Funktionen aus Abschnitt 3.2.3

wie beschrieben definiert sind.

```

1  passedTests = 0
2
3  function semanticEval( R, T, tests ){
4      for( anzahl = 1; anzahl <= maxTargets(R); i++ ){
5          for( targets : targetSets(T, anzahl) ){
6              relProxies = proxies(R, targets)
7              proxy = evalProxies( relProxies, tests )
8              if( proxy != null ){
9                  // passenden Proxy gefunden
10                 return proxy
11             }
12         }
13     }
14     // kein passenden Proxy gefunden
15     return null;
16 }
17
18 function evalProxies(proxies, tests){
19     for( proxy : proxies ){
20         passedTests = 0
21         evalProxy(proxy, tests)
22         if( passedTests == tests.size ){
23             // passenden Proxy gefunden
24             return proxy
25         }
26     }
27     // kein passenden Proxy gefunden
28     return null
29 }
30
31 function evalProxy(proxy, tests){
32     for( test : tests ){
33         if( !test.eval( proxy ) ){
34             // wenn ein Test fehlschlaegt, dann entspricht der
35             // Proxy nicht den semantischen Anforderungen
36             return
37         }
38         passedTests = passedTests + 1
39     }
40 }

```

Listing 3.10: Semantische Evaluation ohne Heuristiken

Die Dauer der Laufzeit der in Listing 3.10 definierten Funktionen hängt maßgeblich von der

Anzahl der *Proxies* ab, die für den *required Typ R* in der Bibliothek *L* erzeugt werden können (siehe auch Abschnitt 3.2.4 Funktion *libProxyCount*). Im schlimmsten Fall müssen alle *Proxies* generiert werden und hinsichtlich der vordefinierten Tests geprüft werden. Um die Anzahl dieser *Proxies* zu reduzieren, werden die im folgenden Abschnitt beschriebenen Heuristiken verwendet.

3.4 Heuristiken

Die Heuristiken werden an unterschiedlichen Stellen des Algorithmus aus Listing 3.10 eingebaut. Teilweise ist es für die Verwendung einer Heuristik notwendig, weitere Information während der semantischen Evaluation zu ermitteln und diese zu speichern. In den folgenden Abschnitten werden die Heuristiken und die dafür notwendigen Anpassungen an den jeweiligen Funktionen beschrieben.

Die folgenden Heuristiken haben zum Ziel, die Reihenfolge, in der die Proxies hinsichtlich der vordefinierten Tests geprüft werden, so anzupassen, dass ein passender Proxy möglichst früh geprüft wird.

3.4.1 Beachtung des Matcherratings (LMF)

Bei dieser Heuristik, welche den Namen *low matcherrating first* (kurz: LMF) trägt, werden die Mengen von Target-Typen, aus denen die Proxies erzeugt werden, auf der Basis eines so genannten Matcherratings bewertet. Bei dem Matcherrating einer solchen Menge handelt es sich um einen numerischen Wert, auf dessen Basis entschieden werden kann, für welche Menge von Target-Typ die Generierung und Evaluation der Proxies vollzogen werden soll.

Um das Matcherrating zu ermitteln, wird für jede Matchingrelation bzw. für jeden Matcher aus Abschnitt 3.1.2 ein Basisrating vergeben. Folgende Funktion beschreibt das Basisrating für

das Matching zweier Typen S und T :

$$base(S, T) := \begin{cases} 100 & \text{wenn } S \Rightarrow_{exact} T \\ 200 & \text{wenn } S \Rightarrow_{gen} T \\ 200 & \text{wenn } S \Rightarrow_{spec} T \\ 300 & \text{wenn } S \Rightarrow_{contained} T \\ 300 & \text{wenn } S \Rightarrow_{container} T \end{cases}$$

Dabei ist zu erwähnen, dass einige der o.g. Matcher über dasselbe Basisrating verfügen. Das liegt daran, dass sie technisch jeweils gemeinsam umgesetzt wurden.⁷

Wie an der Funktion *base* zu erkennen ist, wird das Matcherrating für Typen, die über den *StructuralTypeMatcher* gematcht wurden, nicht spezifiziert. Dieses muss berechnet werden. Die Basis dafür bildet ein Matcherrating, welches für die gematchten Methoden ermittelt wird. Hierzu sei die Funktion *bases_{method}* für zwei Methoden mR und mT mit $mR \Rightarrow_{method} mT$ wie folgt definiert:

$$bases_{method}(mR, mT) := \left. \begin{array}{l} base(ret(mR), ret(mT)) \cup \\ \bigcup_{i=1}^n base(pR_i, pT_i) \end{array} \right| \begin{array}{l} \{pR_1, \dots, pR_n\} = params(mR) \wedge \\ \{pT_1, \dots, pT_n\} = params(mT) \end{array}$$

Darauf aufbauend kann die Funktion *mRating* für die beiden Methoden mR und mT definiert werden. Hierzu seien folgende Hilfsfunktionen definiert:

$$\begin{aligned} sum(\{v_1, \dots, v_n\}) &= \sum_{i=1}^n v_i \\ max(\{v_1, \dots, v_n\}) &= v_m | 1 \leq m \leq n \wedge \forall i \in \{1, \dots, n\} : v_i \leq v_m \\ min(\{v_1, \dots, v_n\}) &= v_m | 1 \leq m \leq n \wedge \forall i \in \{1, \dots, n\} : v_i \geq v_m \end{aligned}$$

In dieser Arbeit werden vier Varianten für diese Definition von *mRating* vorgeschlagen, die in Abschnitt 5.3 evaluiert werden sollen.

⁷Der *GenTypeMatcher* und der *SpecTypeMatcher* wurden gemeinsam in der Klasse `GenSpecTypeMatcher` umgesetzt. Der *ContentTypeMatcher* und der *ContainerTypeMatcher* wurden gemeinsam in der Klasse `WrappedTypeMatcher` umgesetzt. (siehe angehängter Quellcode)

Variante 1: Durchschnitt ($mRating$)

$$mRating(mR, mT) := \frac{sum(base_{method}(mR, mT))}{|params(mR)| + 1}$$

Variante 2: Maximum ($mRating$)

$$mRating(mR, mT) = max(bases_{method}(mR, mT))$$

Variante 3: Minimum ($mRating$)

$$mRating(mR, mT) = min(bases_{method}(mR, mT))$$

Variante 4: Durchschnitt aus Minimum und Maximum ($mRating$)

$$mRating(mR, mT) = \frac{max(bases_{method}(mR, mT)) + min(bases_{method}(mR, mT))}{2}$$

In einem *provided Typ* P sind mitunter mehrere Methoden deklariert, die ein Matching zu einer Methode m aufweisen. Für die Bestimmung des Matcherratings sei hierbei nur das kleinste Matcherrating jener Methoden aus P relevant. Das minimale Matcherrating einer solchen Methode wird durch folgende Funktion beschrieben:

$$minMRating(m, P) := \left. \begin{array}{l} min(mRating(m'_1), \\ \dots, mRating(m'_n)) \end{array} \right| \begin{array}{l} \{m'_1, \dots, m'_n\} = \\ structM_{target}(m, T) \end{array}$$

Für einen *required Typ* R und einem *provided Typ* P wird die Menge dieser minimalen Matcherratings je Methode $m \in structM(R)$ über folgende Funktion definiert:

$$minMRatings(R, P) := \left\{ minMRating(m, P) \mid m \in structM(R, P) \right\}$$

In einer Bibliothek L wird die Ermittlung des Matcherratings eines *required Typs* R und einer Menge von *provided Typen* $\{T_1, \dots, T_n\}$ mit $\{T_1, \dots, T_n\} \in cover(R, L)$ über die Funktion *rating* beschrieben. Auch hierfür werden in dieser Arbeit insgesamt 4 Varianten vorgeschlagen, die in Kapitel 5 evaluiert werden sollen.

Variante 1: Durchschnitt (*rating*)

$$rating(R, \{T_1, \dots, T_n\}) := \frac{sum(minMRatings(R, T_1), \dots, minMRatings(R, T_n))}{\sum_{i=1}^n |structM(R, T_i)|}$$

Variante 2: Maximum (*rating*)

$$rating(R, \{T_1, \dots, T_n\}) := max(minMRatings(R, T_1), \dots, minMRatings(R, T_n))$$

Variante 3: Minimum (*rating*)

$$rating(R, \{T_1, \dots, T_n\}) := min(minMRatings(R, T_1), \dots, minMRatings(R, T_n))$$

Variante 4: Durchschnitt aus Minimum und Maximum (*rating*)

$$rating(R, \{T_1, \dots, T_n\}) := \frac{min(minMRatings(R, T_1), \dots, minMRatings(R, T_n)) + max(minMRatings(R, T_1), \dots, minMRatings(R, T_n))}{2}$$

Da die Funktion *rating* von *mRating* abhängt und für *mRating* 4 Varianten vorgeschlagen wurden, ergeben sich insgesamt 16 Varianten für die Definition von *rating*. Diese Varianten (1.1 - 4.4) sind in der Tabelle 3.5 mit den Kombinationen der Varianten für *mRating* und *rating* aufgeführt.

Zur Anwendung der Heuristik muss das Matcherrating bei der Erzeugung der Proxies aus den jeweiligen Mengen von *provided Typen* beachtet werden. Dabei sollte die Liste der Mengen von *provided Typen*, die über die Funktion *targetSets* abgebildet wird und über die in der Methode **semanticEval** iteriert wird, entsprechend dem Matcherrating sortiert werden. Dadurch werden in der Methode **evalProxies** zuerst die Proxies evaluiert, die auf Basis einer Menge von *provided Typen* mit dem kleinsten Matcherrating erzeugt wurden. Listing 3.11 zeigt die Anpassungen der Methode **relevantProxies** auf Basis der Implementierung der semantischen Evaluation aus Listing 3.10. Für die Sortierung der Liste von Proxies wurde in der Methode **LMF** exemplarisch das Bubble-Sort-Verfahren verwendet.

```

1 function semanticEval( R, T, tests ){
2   for( anzahl = 1; anzahl <= maxTargets(R); i++ ){
3     targetSets = targetSets(T, anzahl)

```

Variante	Variante für <i>rating</i>	Variante für <i>mRating</i>
1.1	1	1
1.2	1	2
1.3	1	3
1.4	1	4
2.1	2	1
2.2	2	2
2.3	2	3
2.4	2	4
3.1	3	1
3.2	3	2
3.3	3	3
3.4	3	4
4.1	4	1
4.2	4	2
4.3	4	3
4.4	4	4

Tabelle 3.5: Varianten für die Ermittlung des Matcherratings einer Menge von *provided Typen*

```

4      sortedSets = LMF( R, targetSets )
5      for( targets : sorted ){
6          relProxies = proxies(R, targets)
7          proxy = evalProxies( relProxies, tests )
8          if( proxy != null ){
9              // passenden Proxy gefunden
10             return proxy
11         }
12     }
13 }
14 // kein passenden Proxy gefunden
15 return null;
16 }
17
18 function LMF( R, targets ){
19     for ( n=targets.size(); n>1; n-- ){
20         for( i=0; i<n-1; i++){
21             if( rating(R, targets[i] ) < rating(R, proxies[i+1] ) ){
22                 tmp = targets[i]
23                 targets[i] = targets[i+1]
24                 targets[i+1] = tmp
25             }
26         }
27     }

```

```

27     }
28     return targets
29 }
```

Listing 3.11: Semantische Evaluation mit Heuristik LMF

3.4.2 Beachtung positiver Tests (PTTF)

Das Testergebnis, welches bei Applikation eines Testfalls für einen Proxy ermittelt wird, ist maßgeblich von den Methoden-Delegationen des Proxies abhängig. Jede Methoden-Delegation MD enthält einen Typ in dem die Delegationsmethode deklariert wurde. Dieser Typ befindet sich im Attribut $MD.del.delTyp$. Im Fall der strukturellen Proxies, handelt es sich bei diesem Typ um einen der Target-Typen des Proxies.

Für einen required Typ R aus einer Bibliothek L , kann ein Target-Typ T in den Mengen der möglichen Mengen von Target-Typen $cover(R, L)$ mehrmals auftreten. Dies gilt insbesondere dann, wenn es in $cover(R, L)$ Mengen gibt, deren Mächtigkeit größer ist, als die Mächtigkeit der Menge, in der T enthalten ist. Daher gilt:

$$\frac{\exists TG, TG' \in cover(R, L) : \wedge T \in TG \wedge |TG| < |TG'|}{\exists TG'' \in cover(R, L) : |TG'| = |TG''| \wedge T \in TG''}$$

Für die in diesem Abschnitt beschriebene Heuristik mit dem Namen *positive tested targets first* (kurz: PTTF) ist das Ergebnis einzelner Tests in Bezug auf einen Proxy P relevant. Wenn ein Testfall mit einem Proxy P erfolgreich durchgeführt wurde, dann sollte die Reihenfolge der zu prüfenden Proxies so angepasst werden, dass die Proxies, die einen *Target-Typen* des Proxies P verwenden, im weiteren Verlauf zuerst geprüft werden.

Dafür sind auf Basis von Listing 3.10 mehrere Anpassungen bzgl. der Implementierung der Methode `evalProxies` von Nöten:

1. Die Target-Typen der Proxies, mit denen mind. ein Testfall erfolgreich durchgeführt werden konnte, müssen in einer globalen Variable (`prioTargets`) hinterlegt werden.
2. Die Liste der Proxies, die der Methode `evalProxies` als Parameter übergeben wird,

muss so sortiert werden, dass die Proxies, mit den Target-Typen, die in der globalen Variable (`prioTargets`) hinterlegt wurden, zuerst getestet werden. Dies erfolgt wiederum exemplarisch über das Bubble-Sort-Verfahren in der Methode `PTTF`.

3. Die Liste der Proxies, über die innerhalb der Methode `evalProxies` iteriert wird, kann bzgl. ihrer Reihenfolge bereits dann optimiert werden, wenn mind. einer der Testfälle für den aktuellen Proxy erfolgreich durchgeführt wurde. Dazu müssen jedoch die Proxies, die bereits innerhalb der Methode getestet wurden, in einer lokalen Variable (`tested`) hinterlegt werden. Dann kann die Methode rekursiv mit den Proxies, die noch nicht getestet wurden, aufgerufen werden. So werden die darin enthaltenen Elemente aufgrund der 2. Anpassung erneut sortiert.

In Listing 3.12 sind die entsprechenden Anpassungen und Ergänzungen im Vergleich zu Listing 3.10 zu entnehmen.

```

1  prioTargets = []
2
3  function evalProxies( proxies, tests ){
4      tested = []
5      sorted = PTTF( proxies )
6      for( proxy : sorted ){
7          passedTests = 0
8          evalProxy( proxy, tests )
9          if( passedTests == tests.size ){
10             // passenden Proxy gefunden
11             return proxy
12         }
13         else{
14             tested.add( proxy )
15             if( passedTests > 0 ){
16                 prioTargets.addAll( proxy.targets )
17                 // noch nicht evaluierte Proxies ermitteln
18                 leftProxies = sorted.removeAll( testedProxies )
19                 return evalProxies( leftProxies, tests )
20             }
21         }
22     }
23     // kein passenden Proxy gefunden
24     return null
25 }
26
27 function PTTF( proxies ){

```

```

28  for ( n=proxies.size ; n>1; n--){
29      for( i=0; i<n-1; i++){
30          targetsFirst = proxies[i].targets
31          targetsSecond = proxies[i+1].targets
32          if( !prioTargets.contains( targetsFirst ) && prioTargets.contains(
              targetsSecond ) ){
33              tmp = proxies[i]
34              proxies[i] = proxies[i+1]
35              proxies[i+1] = tmp
36          }
37      }
38  }
39  return proxies
40 }

```

Listing 3.12: Semantische Evaluation mit Heuristik PTTF

3.4.3 Beachtung fehlgeschlagener Methodenaufrufe (BL_NMC)

Diese Heuristik mit dem Namen *blacklist negative method calls* (kurz: BL_NMC) beschreibt ein Ausschlussverfahren. Das bedeutet, dass bestimmte Proxies auf der Basis von Erkenntnissen, die während der laufenden semantischen Evaluation entstanden sind, für den weiteren Verlauf ausgeschlossen werden. Dadurch soll die erneute Prüfung eines Proxies, dessen Target-Typen ohnehin nicht zum gewünschten Ergebnis führt, verhindert werden.

Die Heuristik zielt darauf ab, Methoden-Delegationen, die immer fehlschlagen, zu identifizieren. Wurde eine solche Methoden-Delegation gefunden, können alle Proxies, die diese Methoden-Delegation enthalten von der weiteren Exploration ausgeschlossen werden.

Die Methoden-Delegationen, die auf der Basis der beiden folgenden Heuristiken aussortiert werden sollen, werden zu diesem Zweck in einer globalen Variable (`mdelBlacklist`) gehalten. Aus einer Liste von Proxies können darauf aufbauend diejenigen Proxies entfernt werden, die eine jener Methoden-Delegationen enthalten. Dabei wird davon ausgegangen, dass die Methoden eines required Typen über den Namen identifiziert werden können.

Das Füllen der globalen Variable `mdelBlacklist` erfolgt in der Methoden `evalProxy`. Die

Identifikation der Methoden-Delegationen über die Methodennamen erfolgt in der Methode `getMethodDelegations`. Beide Methoden sind Listing 3.13 zu entnehmen.

```

1  function evalProxy( proxy, tests ){
2      for( test : T ){
3          if( test.eval( proxy ) ){
4              passedTestcases = passedTestcases + 1
5          }
6          else {
7              triedMethodCalls = test.triedMethodCalls
8              mDel = getMethodDelegations( proxy, triedMethodCalls )
9              mdelBlacklist.add( mDel )
10         }
11     }
12 }
13
14 function getMethodDelegations( proxy, methodNames ){
15     for( i=0; i < proxy.dels.size; i++ ){
16         methodName = proxy.dels[i].call.name
17         if( methodNames.containsAll( methodName ) ){
18             return proxy.dels[i]
19         }
20     }
21     return null
22 }
```

Listing 3.13: Evaluierung einzelner Proxies mit BL_MNC

Das Ausschließen bestimmter Proxies erfolgt, indem Elemente aus einer Liste von Proxies entfernt werden. Listing 3.14 zeigt die dafür vorgesehene Methode `BL`, welche die Basis-Liste der Proxies im Parameter `proxies` und die Liste der Kombinationen von Methoden-Delegationen, die die Grundlage für den Ausschluss einzelner Proxies bilden, im Parameter `blacklist` erwartet.

```

1  function BL( proxies, blacklist ){
2      filtered = []
3      for( proxy : proxies ){
4          blacklisted = false
5          for( md : blacklist ){
6              if( proxy.dels.contains( md ) ){
7                  blacklisted = true
8                  break
9              }
10         }
11         if( !blacklisted ){
```

```

12     filtered.add( proxy )
13 }
14 }
15 return filtered
16 }

```

Listing 3.14: Blacklist-Methode für Heuristik BL_NMC

Bei dieser Heuristik ist deren Anwendung nach jedem Evaluationsversuch eines einzelnen Proxies sinnvoll. Listing 3.15 zeigt die Anpassungen in der Funktion *evalProxies* aus Listing 3.10 für die Heuristik BL_NMC. Dabei sei davon auszugehen, dass die oben beschriebene Funktion aus den Listings 3.14 und 3.13 zur Verfügung steht.

```

1 function evalProxies( proxies, tests ){
2     tested = []
3     filtered = BL( proxies, mdelBlacklist )
4     for( proxy : proxies ){
5         passedTestcases = 0
6         evalProxy(proxy, tests)
7         if( passedTestcases == tests.size ){
8             // passenden Proxy gefunden
9             return proxy
10        }
11        else{
12            tested.add( proxy )
13            // noch nicht evaluierte Proxies ermitteln
14            leftProxies = proxies.removeAll( tested )
15            return evalProxies( leftProxies, tests )
16        }
17    }
18    // kein passenden Proxy gefunden
19    return null
20 }

```

Listing 3.15: Evaluation mehrere Proxies mit BL_NMC

Der Pseudo-Code für die semantische Evaluation mit der Kombination aller genannten Heuristiken ist im Anhang B zu finden.

Kapitel 4

Implementierung

Die Implementierung der Explorationskomponente besteht aus drei Hauptbestandteilen, die jeweils als separates Java-Projekt umgesetzt wurden. Im weiteren Verlauf werden diese Java-Projekte als Module bezeichnet. In Abbildung 4.1 ist die Architektur der Explorationskom-

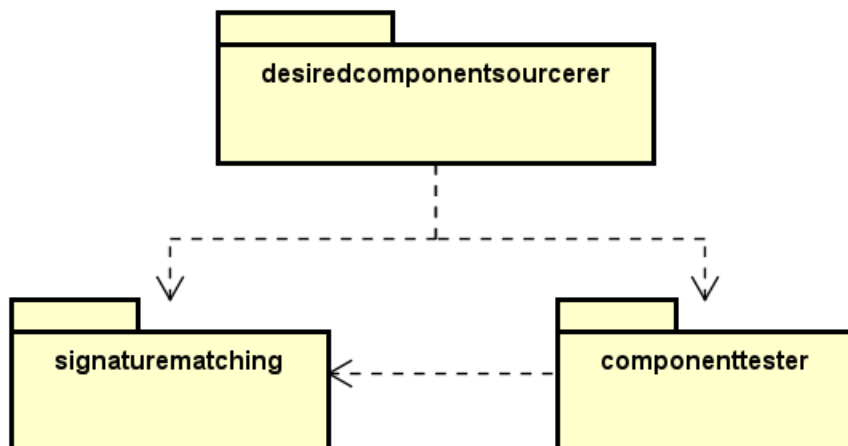


Abbildung 4.1: Architektur

ponente aufgeführt. Dieser ist zu entnehmen, dass die Explorationskomponente aus drei Modulen besteht, die im weiteren Verlauf dieses Kapitels beschrieben werden. Das Modul *DesiredComponentSourcerer* ist dabei von den Modulen *ComponentTester* und *SignatureMatching* abhängig, während das Modul *ComponentTester* lediglich vom Modul *SignatureMatching* abhängig ist.

Darüber hinaus, werden folgende externe Bibliotheken verwendet:

- easymock 3.0 [Tre15]
- cglib 3.3.0 [Ber19]
- objenesis 3.1 [obj21]
- junit 4.13.0 [jun21a]

Auf die konkrete Verwendung der externen Bibliotheken wird in den detaillierteren Beschreibungen der einzelnen Module in den folgenden Abschnitten eingegangen.

4.1 Modul: SignatureMatching

In diesem Modul befinden sich zum Einen die Implementierungen der Matcher, die in Abschnitt 3.1.2 formal beschrieben wurden und zum Anderen die Implementierung der Generatoren für die Proxies. In Abbildung 4.2 sind die wichtigsten Klassen und Interfaces dieses Moduls mit ihren Abhängigkeiten zueinander aufgeführt. Die Matcher befinden sich dabei im Package *matching* und die Generatoren für die Proxies in Form der Implementierungen des Interfaces `ProxyFactory` im Package *glue*.

Die in Abschnitt 3.1.2 beschriebenen Matcher und Generatoren wurden teilweise in einer Klasse zusammengefasst. Tabelle 4.1 zeigt die Zuordnung von Matchern zu den jeweiligen Klassen, die die Implementierung dieser darstellen (Spalte: Matcher-Implementierung). Zudem sind in der Tabelle 4.1 auch die Klassen ausgewiesen, die die Implementierung des Generators für den Proxy, der auf Basis des Matchers Anwendung findet (Spalte: Generator-Implementierung).

Die Klasse `StructuralTypeMatcher` nimmt dabei eine Sonderstellung ein. Dies ist daran zu erkennen, dass dieser nicht das Interface `TypeMatcher` implementiert. Das liegt daran, dass es sich bei diesem Matcher um den Einstiegspunkt der strukturellen Evaluation handelt. Analog zum `StructuralTypeMatcher` aus Abschnitt 3.1.2 wird in der Klasse `StructuralTypeMatcher` auf die anderen Matcher bzw. Matcher-Implementierungen zugegriffen, was in Abbildung 4.2 durch

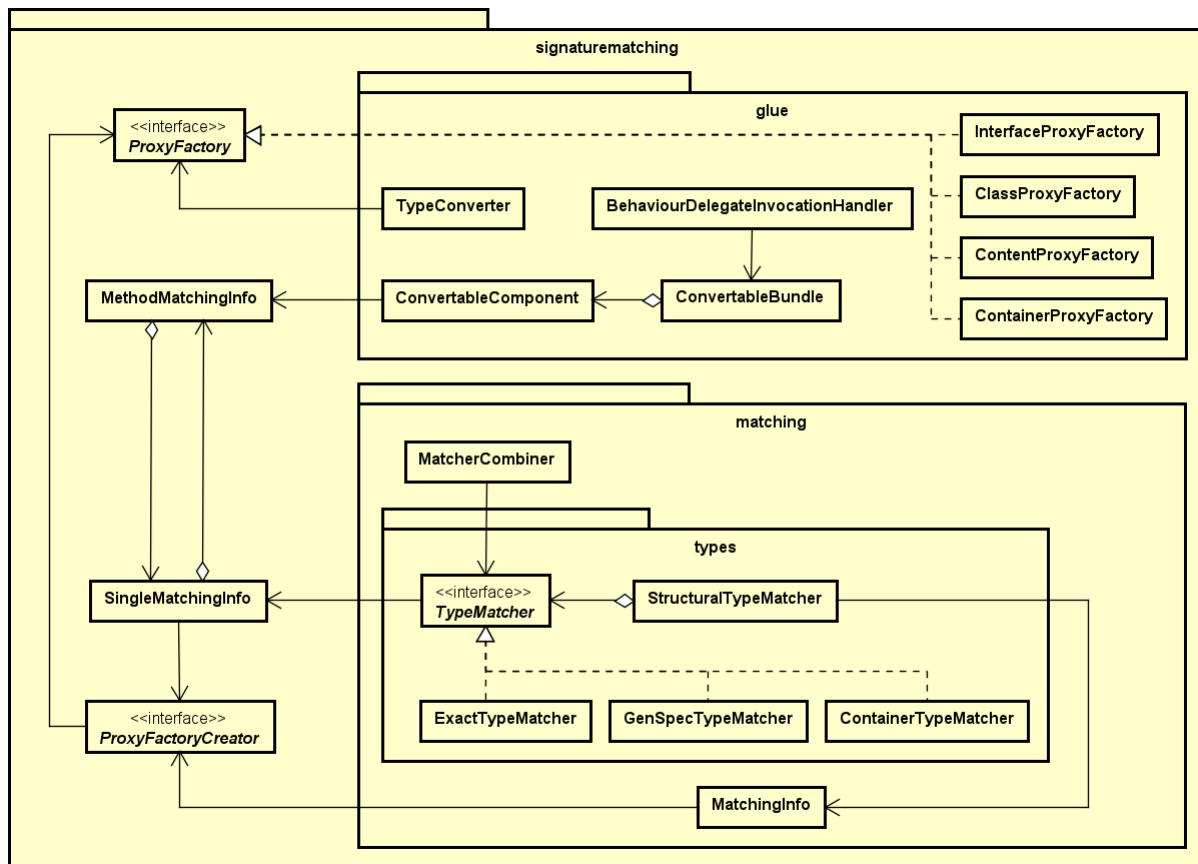


Abbildung 4.2: Modul: SignatureMatching

Matcher	Matcher-Implementierung	Generator-Implementierung
ExactTypeMatcher	ExactTypeMatcher	ClassProxyFactory
GenTypeMatcher	GenSpecTypeMatcher	ClassProxyFactory
SpecTypeMatcher	GenSpecTypeMatcher	ClassProxyFactory
ContentTypeMatcher	ContainerTypeMatcher	ContentProxyFactory
ContainerTypeMatcher	ContainerTypeMatcher	ContainerProxyFactory
StructuralTypeMatcher	StructuralTypeMatcher	InterfaceProxyFactory

Tabelle 4.1: Zuordnung der Matcher zu den Matcher- und Generator-Implementierungen

die Aggregation zwischen der Klasse **StructuralTypeMatcher** und dem Interface **TypeMatcher** angedeutet wird.

Die übrigen Matcher-Klassen implementieren das Interface **TypeMatcher** und können über die

Methode `combine` aus der Klasse `MatcherCombinator` miteinander kombiniert werden¹.

So kann eine Kombination mehrerer `TypeMatcher`, die wiederum von `TypeMatcher` ist, in der Klasse `StructuralTypeMatcher` verwendet werden. Die konkrete `TypeMatcher`-Kombination, die im `StructuralTypeMatcher` instanziiert wird, orientiert sich an den Ausführungen in Abschnitt 3.1.2. Es ist aber zu erwähnen, dass die Verwendung weiterer `Matcher`, die in dieser Arbeit nicht definiert wurden, denkbar ist. Eine solche Erweiterung ließe sich leicht in dieses Modul über die Implementierung des Interfaces `TypeMatcher` und die Verwendung der Klasse `MatcherCombiner` vornehmen.

Alle `Matcher`-Implementierungen bieten die Möglichkeit, zu ermitteln, ob ein `Matching` zwischen zwei Typen besteht (siehe Klassendiagramme in Abbildungen 4.3 und 4.4). Dies erfolgt jeweils über die Methode `matchesType`. Über die Methoden `calculateMatchingInfos` bzw. `calculateMatchingInfo` werden die Informationen bzgl. der Methodendelegationen zwischen den beiden gematchten Typen ermittelt. Diese Informationen werden in einem Objekt der Klasse `SingleMatchingInfo` bzw. `MatchingInfo` zusammengetragen, welche in Abbildung 4.3 und 4.3 detailliert dargestellt werden. Diese beiden Klassen unterscheiden sich lediglich bzgl. des Attributs in dem die Delegationsmethoden hinterlegt sind. Dabei handelt es sich auf Seiten der `SingleMatchingInfo` um das Attribut `methodMatchingInfos` und auf Seiten der `MatchingInfo` um das Attribut `methodMatchingSupplier`.

Während ein Objekt der Klasse `MatchingInfo` mehrere Delegationsmethoden zu einer aufgerufenen Methoden enthalten kann, darf ein Objekt der Klasse `SingleMatchingInfo` lediglich eine Delegationsmethode zu einer aufgerufenen Methode enthalten (vgl. auch Abschnitt 3.1.2). Zusätzlich zu erwähnen ist, dass die Informationen über die Delegationsmethoden aus einer `MatchingInfo` über einen `MethodSupplier` überliefert werden.

Eine Instanz der Klasse `MethodSupplier` enthält zum Einen ein `MatcherRating` welches Informationen bzgl. des in Abschnitt 3.4.1 beschriebenen `Matcher-Ratings` beinhaltet. Zum Anderen werden im Attribut `methodMatchingInfo` in einem Objekt der Klasse `MethodMatchingInfo`

¹Ein Beispiel für die Kombination von `Matchern` ist im Anhang A zu finden.

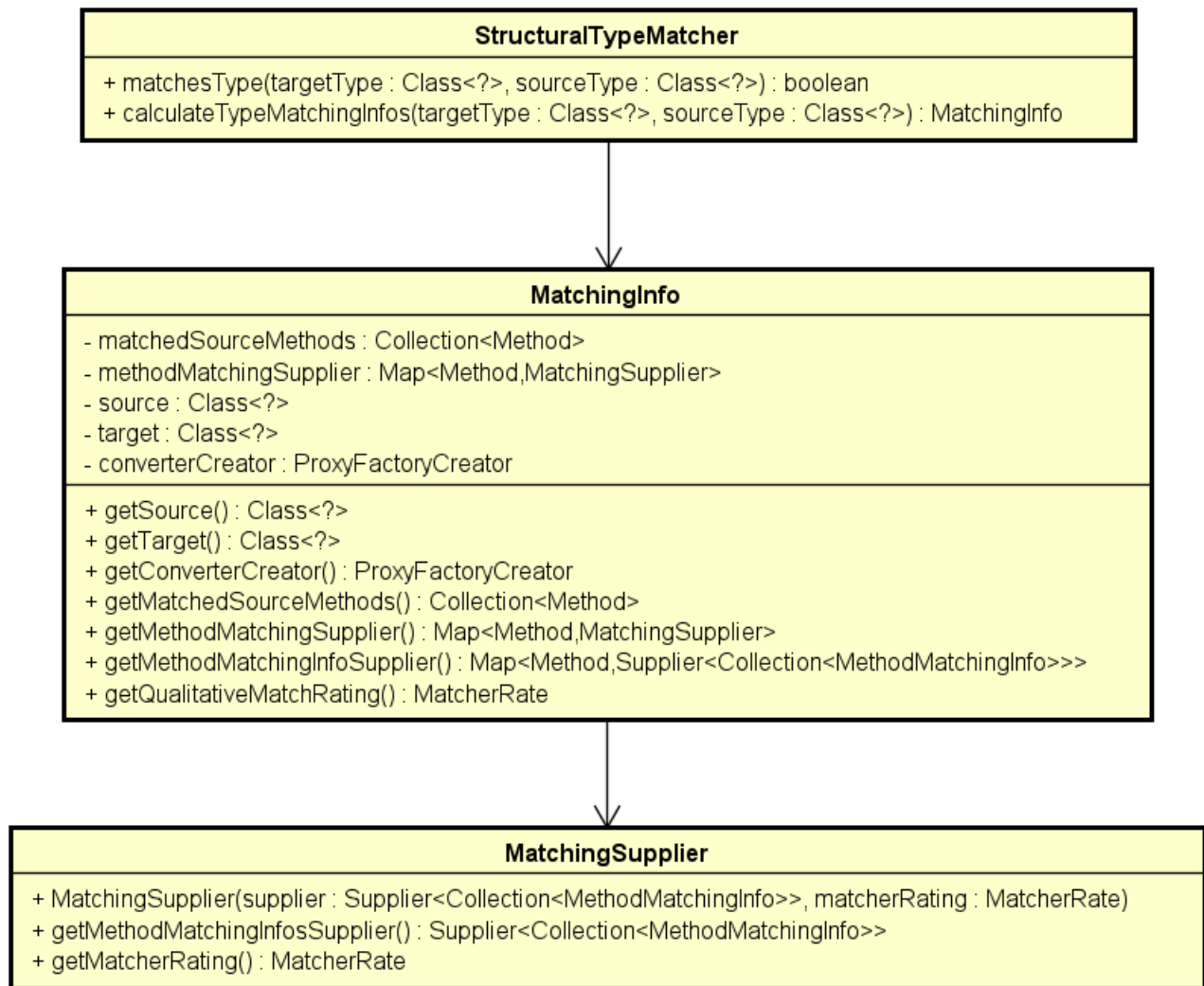
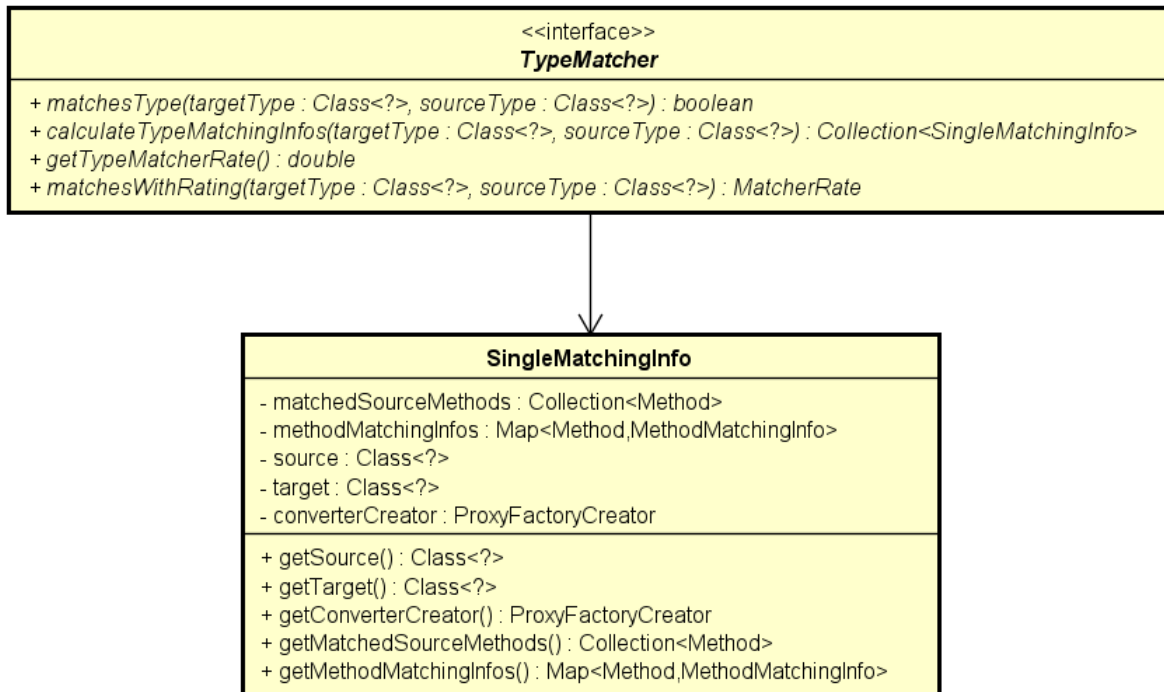


Abbildung 4.3: Klassendiagramm: StructuralTypeMatcher und MatchingInfos

(siehe Abbildung 4.5) die Informationen bzgl. der Delegation der aufgerufenen Methode an die Delegationsmethode hinterlegt.

Bezüglich der Klasse `SingleMatchingInfo` ist noch das Attribut `proxyFactoryCreator` zu beschreiben. Darin werden Informationen bzgl. der strukturellen Verbindung zwischen den gematchten Typen gehalten. Für den *ExactTypeMatcher*, den *GenTypeMatcher* und den *SpecTypeMatcher* wird dabei ein `ProxyFactoryCreator` erzeugt, der in der Lage ist, eine `ProxyFactory`

Abbildung 4.4: Klassendiagramm: **TypeMatcher** und **SingleMatchingInfo**

für Typen zu erzeugen, die in einer nominalen Beziehung² stehen. Für den *ContentTypeMatcher* und den *ContainedTypeMatcher* hingegen, wird ein **ProxyFactoryCreator** erzeugt, der in der Lage ist, eine **ProxyFactory** für Typen zu erzeugen, bei denen der eine Typ ein Attribut vom Typ des anderen enthält (vgl. mit Tabelle 4.1). Die erzeugten Objekte vom Typ **ProxyFactory** werden bei der Generierung der Proxies unter der Zuhilfenahme der Bibliotheken *cglib* und *objenesis* verwendet³.

Der **ProxyFactoryCreator** stellt damit eines der Bindeglieder zwischen der Package *matching* und dem Package *glue* innerhalb des Moduls her. Das zweite Artefakt, welches als Bindeglied fungiert, ist die oben bereits erwähnte Klasse **MethodMatchingInfo**, deren Aufbau dem Klassendiagramm aus Abbildung 4.5 zu entnehmen ist.

²Identität, Generalisierung, Spezialisierung

³Diese beiden Frameworks wurden verwendet, da die Erzeugung der Proxies mit ihnen komfortabler ist, als mit den Mitteln die das JKD zur Verfügung stellt. Dies gilt insbesondere für die Erzeugung von Proxies für Klassen, die mit dem Schlüsselwort `final` versehen sind.

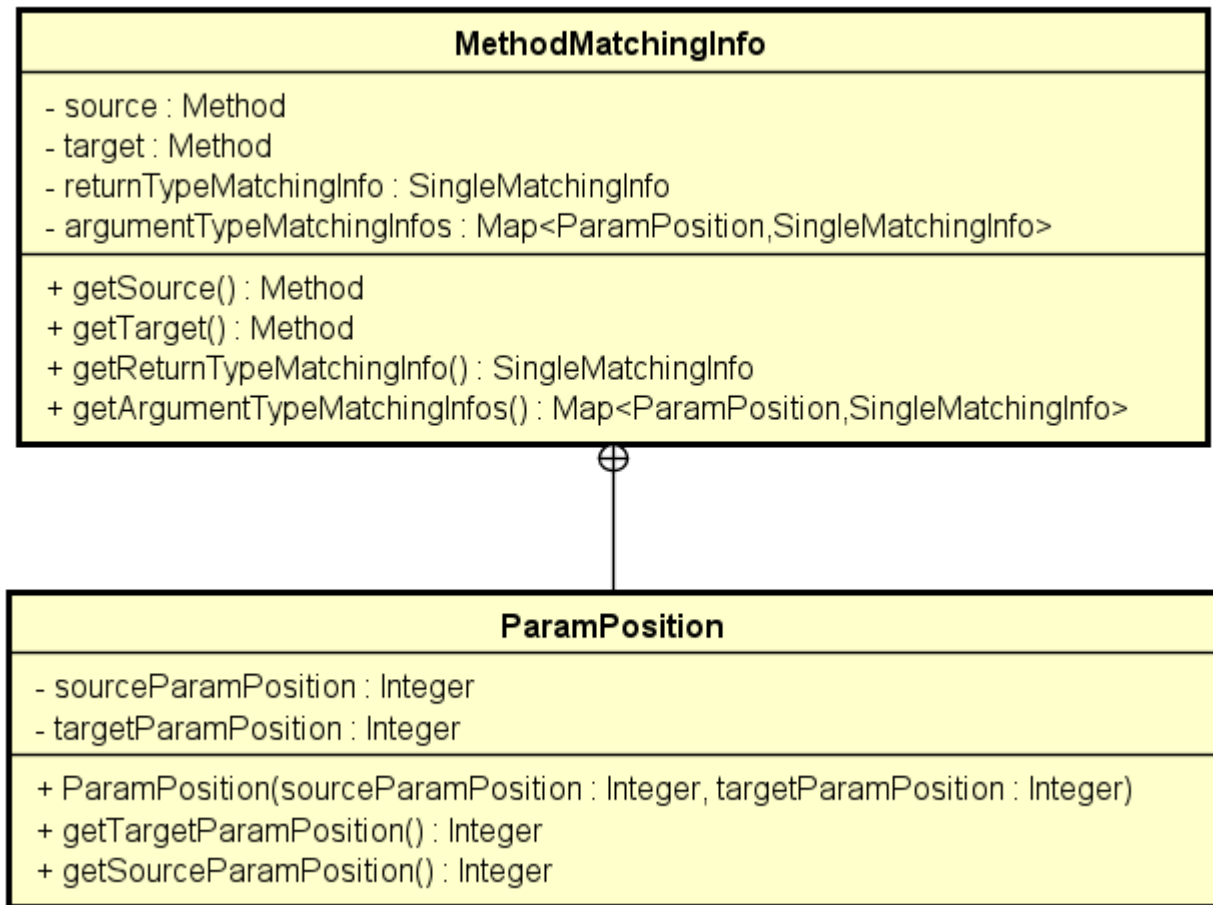


Abbildung 4.5: Klassendiagramm: MethodMatchingInfo

Ein Objekt der Klasse `MethodMatchingInfo` enthält in den Attributen `source` und `target` je eine Methode. Dabei ist im Attribut `source` die aufgerufene Methode der Methoden-Delegation und im Attribut `target` die Delegationsmethode enthalten. Darüber hinaus wird im Attribut `returnTypeMatchingInfo` ein Objekt der Klasse `SingleMatchingInfo` gehalten, welches alle notwendigen Informationen für das Erzeugen eines Proxies des Rückgabetyps der aufgerufenen Methode aus dem Rückgabetypp der Delegationsmethode enthält.

Analog dazu wird im Attribut `argumentTypeMatchingInfos` eine Map, bestehend aus weiteren Objekten der Klasse `SingleMatchingInfo` und jeweils einem Objekt der Klasse `ParamPosition`,

gehalten. Diese Map enthält alle notwendigen Information für das Erzeugen eines Proxies für die Parametertypen der Delegationsmethoden aus den Parametertypen der aufgerufenen Methode, sowie der Anpassung der Übergabeposition bei der Delegation der aufgerufenen Methode (siehe auch Abschnitt 3.2.1).

Um die Methoden-Delegationen zu koordinieren, wird bei der Erzeugung des Proxies in der jeweiligen `ProxyFactory` für das Proxy-Objekt ein `InvocationHandler` instanziiert (vgl. [inv20]). Dieses Interface wird im *glue*-Package durch die Klasse `BehaviourDelegateInvocationHandler` implementiert, in der letztendlich die Koordination der Methoden-Delegationen auf Basis der jeweiligen `MethodMatchingInfo` spezifiziert ist.

Um einen Proxy basierend auf dem Matching zweier Typen zu erzeugen steht die Klasse `TypeConverter` zur Verfügung (siehe Abbildung 4.6). Die Zugriffe innerhalb des Packages *glue* als auch die Zugriffe von außerhalb benötigen jeweils ein Objekt der Klasse `ConvertibleBundle`. Diese Klasse beschreibt eine Kombination mehrerer Objekte vom Typ `ConvertibleComponent`, die als Delegationsziele des zu erzeugenden Proxy-Objektes fungieren sollen. Ein Objekt der Klasse `ConvertibleComponent` enthält eine Liste von Objekten vom Typ `SingleMatchingInfo`, die wie bereits erwähnt beschreiben, am welche Methode die Delegation erfolgen soll. Das Objekt im Attribut `convertableObject` der `ModuleMatchingInfo` beinhaltet das Objekt, auf dem die Delegationsmethode aufgerufen werden soll.

4.2 Modul: ComponentTester

Dieses Modul ist für die Ausführung der vordefinierten Tests zuständig. Darüber hinaus bietet es die Möglichkeit, die vordefinierten Tests mit den Interfaces, die den dazugehörigen required Typ darstellen, zu verbinden. Dabei sei davon auszugehen, dass ein required Typ *R* in Form eines Interfaces existiert. Um Tests für *R* zu definieren, können eine oder mehrere Testklassen implementiert werden.

Die Testklassen werden dabei in dem Interface *R* über das Attribut `testClasses` der Annotation `RequiredTypeTestReference` angegeben (siehe Abbildung 4.7 Package: *API*). Ein

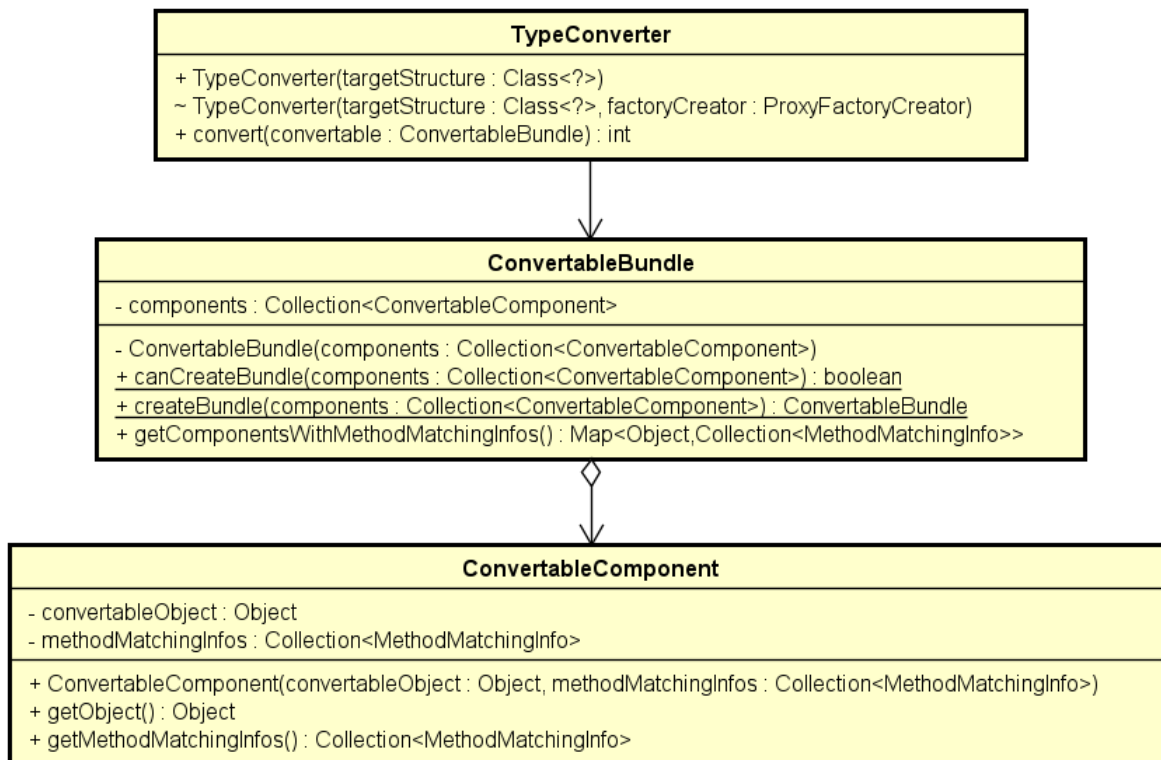


Abbildung 4.6: Klassendiagramm: TypeConverter

Beispiel für die Deklaration eines required Typ in Form eines Java-Interfaces und den dazu gehörigen Testklassen ist im Anhang D zu finden.

Damit die Testmethoden in den Testklassen die in Abschnitt 3.3.1 beschriebenen Eigenschaften aufweisen und durch das *ComponentTester*-Modul ausfindig gemacht werden können, stehen mehrere Artefakte in dem *API*- und dem *SPI*-Package des *ComponentTester*-Moduls bereit (siehe Abbildung 4.7).

So muss jede Testklasse eine Methode bereitstellen, über die ein Objekt vom Typ *R* in die Instanz der Testklasse injiziert werden kann.⁴ Diese Methode wird von dem *ComponentTester*-Modul über die Annotation `RequiredTypeInstanceSetter` gefunden. Von daher muss die Methode mit eben dieser Annotation markiert werden. Die Testmethoden müssen von der

⁴auch genannt: Setter-Injection (vgl. [Fow04])

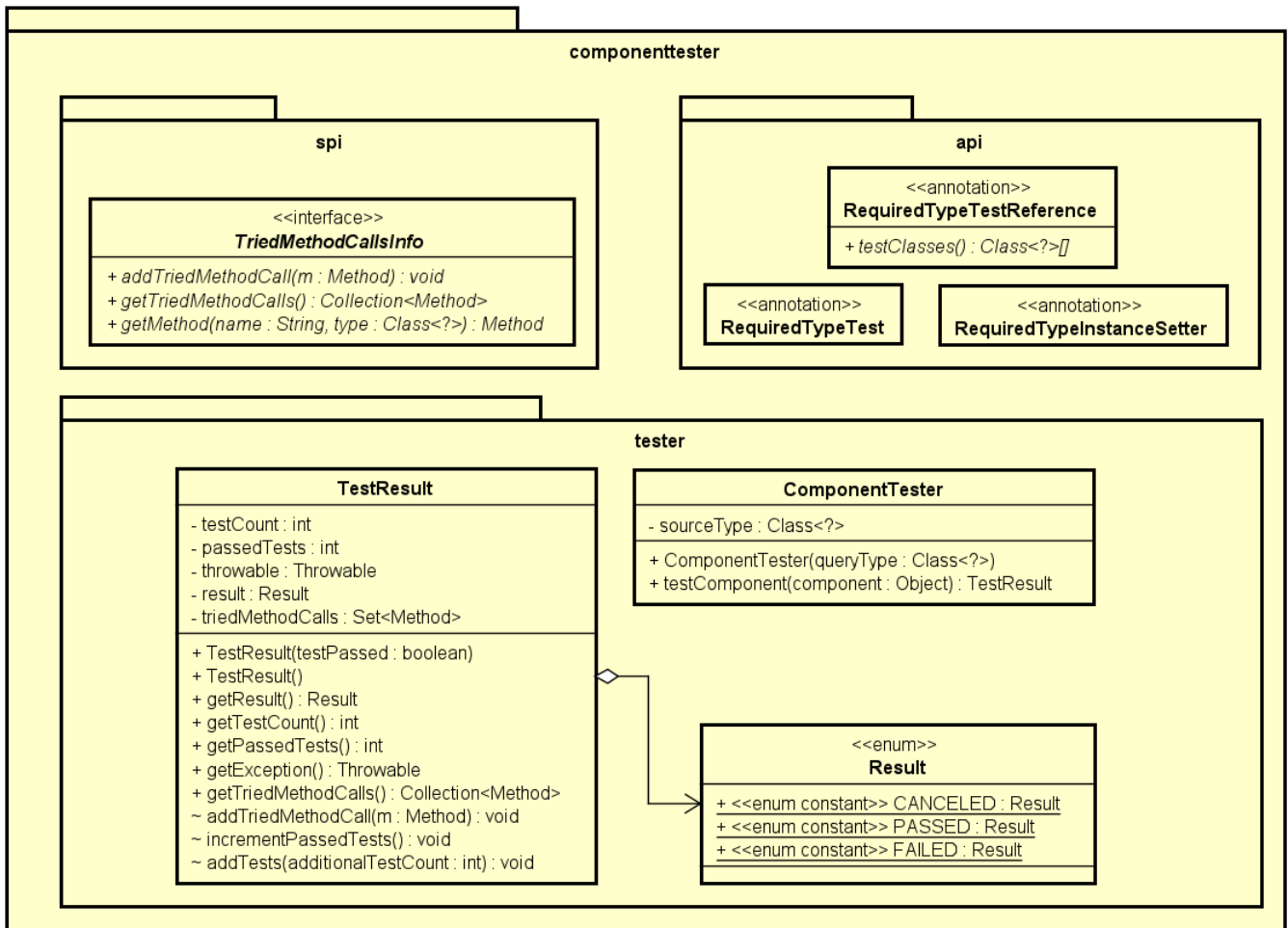


Abbildung 4.7: Modul: ComponentTester

Sichtbarkeit her öffentlich (**public**) sein. Weiterhin dürfen die Testmethoden keine Parameter erwarten und müssen mit der Annotation **RequiredTypeTest** markiert sein. Die Erwartungen innerhalb der Testmethoden müssen über die in JUnit 4 zur Verfügung stehenden Methoden aus der Klasse **Assert** (vgl. [jun21b]) deklariert werden. Testdaten, die für alle Testmethoden innerhalb einer Testklasse zur Verfügung stehen sollen, können über Methoden bereitgestellt werden, die mit den durch die JUnit 4 Bibliothek bereitgestellten Annotationen **Before** und **After** (vgl. [jun21b]) markiert wurden.

Um die Reihenfolge der versuchten Aufrufe der Methoden, die von R angeboten werden, zu verwalten, muss die Testklasse das Interface `TriedMethodCallsInfo` implementieren (siehe Abbildung 4.7 Package: *spi*). Dadurch wird die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` erzwungen. Die Methode `getMethod` kann mit der Defaultimplementierung übernommen werden, sofern die in R deklarierten Methoden über den Namen identifiziert werden können.

Die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` hat so zu erfolgen, dass bei einem Aufruf der Methode `addTriedMethodCall` der übergebene Parameter an eine Liste angefügt wird. Der Aufruf der Methode `getTriedMethodCalls` liefert eben diese Liste als Rückgabewert. Weiterhin ist sicherzustellen, dass vor dem Aufruf einer Methode m aus R die Methode `addTriedMethodCall` mit m als Parameter aufgerufen wird. Im Anhang D ist ein Beispiel für die korrekte Implementierung von Testklassen zu finden (siehe Listings D.8 - D.14).

Der Test eines Proxies für R wird über eine Instanz der Klasse `ComponentTester` gestartet (siehe Abbildung 4.7 Package: *Tester*). In Abhängigkeit der in R deklarierten Testklassen werden alle darin befindlichen Testmethoden durchgeführt, bis einer dieser Testfälle fehlschlägt. Der Aufrufer erhält dabei ein Objekt der Klasse `TestResult` zurück (siehe Abbildung 4.7). In diesem Objekt sind die für die Auswertung des Testergebnisses relevanten Informationen vorhanden, auf die die Heuristiken *PTTF* (siehe Abschnitt 3.4.2) und *BL_NMC* (siehe Abschnitt 3.4.3) angewiesen sind.

4.3 Modul: DesiredComponentSourcerer

In diesem Modul ist die Implementierung der Exploration zu finden. Zum Starten der Exploration für ein *required Typ* R in Form eines Interfaces muss zuerst eine Instanz der Klasse `DesiredComponentFinder` erzeugt werden (genannt: *Finder*). Dies erfolgt über einen Konstruktor, der ein Objekt der Klasse `DesiredComponentFinderConfig` (genannt: *Konfig*) erwartet (siehe Abbildung 4.8).

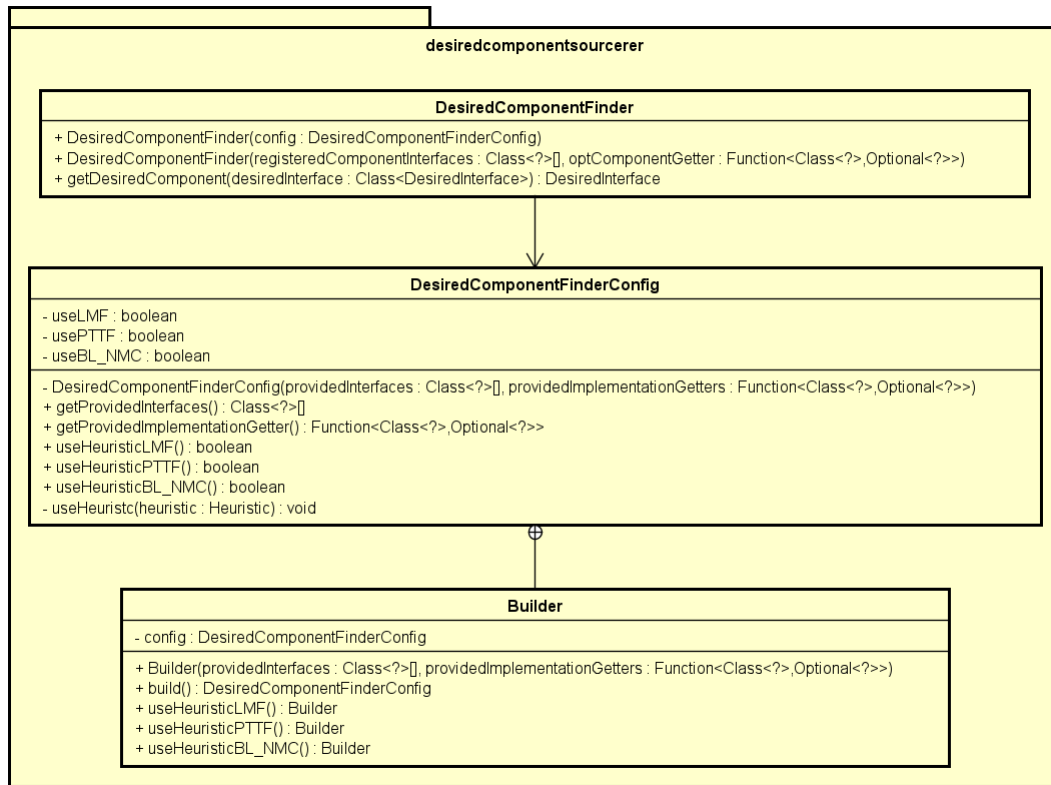


Abbildung 4.8: Modul: DesiredComponentSourcerer

Die Erzeugung einer solchen *Konfig* erfolgt über einen Builder (Builder-Pattern). Dabei müssen zum Einen alle *provided Typen* in Form einer Liste von Interfaces angegeben werden. Zum Anderen wird eine Funktion (`java.util.Function`) gefordert, über die die Implementierungen der im Parameter übergebenen Interfaces ermittelt werden können.

Zum Zweck der gezielten Evaluation der Heuristiken in Kapitel 5 kann über die *Konfig* gesteuert werden, welche der in Abschnitt 3.4 beschriebenen Heuristiken bei der Exploration verwendet werden sollen. Dies erfolgt über die in Abbildung 4.8 ersichtlichen Methoden mit den Präfix `useHeuristic*`.

Nach der Erzeugung des *Finders* kann die Exploration über die Methode `getDesiredComponent` mit der Übergabe des *desired Interface* *R* als Parameter gestartet werden. Im Anschluss wird die syntaktische Evaluation für alle *provided Interfaces* durchgeführt. Hierzu wird ein Objekt

vom `StructuralTypeMatcher` aus dem *SignatureMatching*-Modul verwendet⁵ und versucht die *provided Typen* mit dem *required Typ* zu matchen.

Auf formaler Ebene gleicht dieser Schritt der Ausführung der Funktion $cover(R, L)$, wobei die in L befindlichen *provided Typen* auf die Typen, die dem *Finder* bei der Instanziierung übergebenen wurden, beschränkt sind.

Nach der strukturellen Evaluation, wird gemäß Abschnitt 3.3 die semantische Evaluation durchgeführt. Dabei werden zuerst die Proxies aus den Kombinationen der gematchten *provided Typen*⁶ erzeugt, welche im Anschluss hinsichtlich der vordefinierten Tests zum *required Typ* evaluiert werden. Dabei werden die Heuristiken, die in der *Konfig* hinterlegt wurden, angewendet. Sofern bei der Exploration ein Proxy erfolgreich evaluiert wurde, wird dieser als Ergebnis des Aufrufs der Methode `getDesiredComponent` zurückgegeben.

⁵Dieses Objekt wird beim Instanzieren des *Finders* erzeugt (siehe auch Anhang A: Listing A.2).

⁶Diese Kombinationen sind auf formaler Ebene Äquivalent zu den Elementen der Mengen aus $cover(R, L)$.

Kapitel 5

Untersuchungsergebnisse

In dem System, welches für die Evaluation der Heuristiken verwendet wird, sind insgesamt 891 *provided Typen* und 7 *required Typen* enthalten. In Tabelle 5.1 sind die Namen der *required Typen* zusammen mit jeweils einem Kürzel und den Namen der strukturell und semantisch matchenden Kombinationen von *provided Typen* aufgeführt, die bei der Exploration ermittelt werden sollen. Die Kürzel dienen im weiteren Verlauf der Identifizierung der *required Typen*.

required Typ	Kürzel	Kombination von provided Typen
ElerFTFoerderprogrammeProvider	TEI1	ElerFTStammdatenAuskunftService
FoerderprogrammeProvider	TEI2	StammdatenAuskunftService
MinimalFoerderprogrammeProvider	TEI3	StammdatenAuskunftService
IntubatingFireFighter	TEI4	Doctor, FireFigher
IntubatingFreeing	TEI5	Doctor, FireFigher
IntubatingPatientFireFighter	TEI6	Doctor, FireFigher
KOFGPCProvider	TEI7	ElerFTStammdatenAuskunftService, StammdatenAuskunftService

Tabelle 5.1: Required Typen mit Kürzeln von matchenden Kombinationen von provided Typen für die Evaluation

Die Deklaration der *required Typen* und der *provided Typen* aus Tabelle 5.1 ist im Anhang C zu finden. Aufgrund der Geheimhaltungspflicht bzgl. der Implementierungsdetails kann auf die Deklaration der Java-Interfaces, die sich aus dieser Deklaration der *required* und *provided Typen* ableiten lassen, und deren Implementierungen in dieser Arbeit nicht genauer eingegangen werden.

Um die Ergebnisse nachstellen zu können, kann die Implementierung, welche im Abschnitt 4.3 beschrieben wurde, mit einer beliebigen Bibliothek, welche sich ebenfalls durch die in Abschnitt 3.1.1 beschriebene Struktur von Typen abbilden lässt, verwendet werden.

5.1 Darstellung der Untersuchungsergebnisse

Die Untersuchungsergebnisse werden in der Form von Vier-Felder-Tafeln dargestellt (Beispiel siehe Tabelle 5.2). Für jedes required Interface wird eine Vier-Felder-Tafel für jeden Durchlauf der Schleife innerhalb der Methode `semanticEval` des Explorationsalgorithmus (siehe Abschnitt 3.3) aufgezeigt. Aus der jeweiligen Tafel geht hervor, wie viele Proxies über die Funktion *targetSets* (vgl. Abschnitt 3.3) in dem aktuellen Iterationsschritt erzeugt werden. Der Wert, den die Iterationsvariable *i* im betrachteten Durchlauf enthält, wird in der oberen rechten Ecke der Tafel abgebildet. In der Spalte “positiv” ist die Anzahl der Proxies verzeichnet, die innerhalb des Durchlaufs im schlimmsten Fall evaluiert werden. Die Zahl in der Spalte “negativ” drückt hingegen aus, wie viele der Proxies aufgrund bestimmter Kriterien (bzw. Heuristiken) nicht evaluiert wurden. Die Zeile “falsch” beschreibt die Anzahl der relevanten Proxies, welche die semantische Evaluation nicht bestehen. Dementsprechend stellt die Zeile “richtig” die Anzahl der Proxies dar, welche die semantischen Evaluation bestehen.

Aus Abschnitt 3.2.4 geht hervor, dass die Anzahl der Proxies, die für ein desired Interface *R* mit einer Menge von provided Typen *T* über die Funktion *proxyCount*(*R*, *T*) näherungsweise bestimmt werden kann. Für eine vereinfachte Darstellung der Untersuchungsergebnisse bzgl. eines required Interfaces *R* aus einer Bibliothek *L* und einem Iterationsschritt *i* wird die Anzahl der Proxies für die Anzahl von Mengen von provided Typen *A* auch wie folgt beschrieben:

$$p(A) = \text{proxyCount}(R, \text{targetSets}(T, i) \mid |T| = A \wedge T \in \text{cover}(R, L)$$

Diese Notation kommt jedoch nur bei der Darstellung der Untersuchungsergebnisse eines Iterationsschrittes zum Einsatz, in dem ein passender Proxy gefunden wird. Für alle anderen Durchläufe ist die Anzahl der möglichen Proxies bekannt und wird somit auch dargestellt.

Tabelle 5.2 zeigt ein Beispiel für eine solche Vier-Felder-Tafel, in der die Ergebnisse des 1. Iterationsschrittes dargestellt sind. Dabei wurden 11 Proxies generiert und getestet. 10 dieser Proxies bestanden die Evaluation nicht. Insgesamt konnten Proxies aus 20 Kombinationen von *provided Typen* erzeugt werden. Alle diese abzüglich der 11 generierten Proxies wurden im Vorfeld (bspw. durch Heuristiken) aussortiert. Weiterhin zeigt das Beispiel, dass es einen Proxy gab, der die semantische Evaluation bestand.

1	positiv	negativ
falsch	10	$p(20) - 11$
richtig	1	0

Tabelle 5.2: Beispiel: Vier-Felder-Tafel

5.2 Ausgangspunkt

Für einen *required Typ* können mehrere *provided Typen* gefunden werden, die eine strukturelle Übereinstimmung aufwiesen. Tabelle 5.3 zeigt die Anzahl der strukturell gematchten *provided Typen* zum jeweiligen *required Typ*. Diese kommen einzeln oder in Kombination für die semantische Evaluation in Frage.

required Interface	Anzahl strukturell übereinstimmender provided Interfaces
TEI1	221
TEI2	272
TEI3	268
TEI4	75
TEI5	75
TEI6	53
TEI7	346

Tabelle 5.3: Anzahl strukturell gematchten provided Typen für die Evaluation

Die Tabellen 5.4-5.14 zeigen die Vier-Felder-Tafeln, in denen die Ergebnisse der benötigten Iterationen innerhalb des Explorationsalgorithmus für jeden der *required Typen* aus Tabelle 5.3. Dabei wurden keine Heuristiken verwendet. Somit stellt dies den Ausgangspunkt für die weitere Evaluation dar.

1	positiv	negativ
falsch	233	$p(44) - 234$
richtig	1	0

1	positiv	negativ
falsch	9389	$p(55) - 9399$
richtig	1	0

1	positiv	negativ
falsch	8364	$p(50) - 8365$
richtig	1	0

Tabelle 5.4: Ausgangspunkt für TEI1

Tabelle 5.5: Ausgangspunkt für TEI2

Tabelle 5.6: Ausgangspunkt für TEI3

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	56766	$p(2247) - 56767$
richtig	1	0

Tabelle 5.7: Ausgangspunkt für TEI4 1. Durchlauf

Tabelle 5.8: Ausgangspunkt für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.9: Ausgangspunkt für TEI5 1. Durchlauf

2	positiv	negativ
falsch	244479	$p(2775) - 244480$
richtig	1	0

Tabelle 5.10: Ausgangspunkt für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.11: Ausgangspunkt für TEI6 1. Durchlauf

2	positiv	negativ
falsch	43360	$p(1323) - 43361$
richtig	1	0

Tabelle 5.12: Ausgangspunkt für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle 5.13: Ausgangspunkt für TEI7 1. Durchlauf

2	positiv	negativ
falsch	7764501	$p(52150) - 7764502$
richtig	1	0

Tabelle 5.14: Ausgangspunkt für TEI7 2. Durchlauf

Für die *required Typen* *TEI4-TEI7* werden zwei Durchläufe benötigt, da die semantischen Tests nur von einem Proxy bestanden werden, der aus einer Kombination zweier *provided Typen* erzeugt wurde (siehe auch Tabelle 5.1).

5.3 Ergebnisse für die Heuristik LMF

In Bezug auf die Heuristik *LMF* gilt es nicht nur zu evaluieren, ob die Suche nach einem Proxy, der die vordefinierten Tests besteht, beschleunigt werden kann, sondern auch, mit welcher Variante zur Bestimmung des Matcherratings (vgl. Abschnitt 3.4.1) die besten Ergebnisse erzielt werden können.

Hierzu wird die Exploration für alle der oben genannten *required Typen* für jede Variante zur Bestimmung der Matcherratings durchgeführt (siehe Abschnitt 3.4.1 Tabelle 3.5). Im folgenden Verlauf wird lediglich auf die Variante eingegangen, die die besten Ergebnisse hervorgebracht hat. Die Ergebnisse unter Verwendung der übrigen Varianten sind im Anhang E zu finden.

Die Variante 1.1 (vgl. Tabelle 3.5) erbrachte die besten Ergebnisse. Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse mit dieser Variante zur Bestimmung der Matcherratings für die *required Typen* *TEI1-TEI3* auf.

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	1889	$p(55) - 1890$
richtig	1	0

1	positiv	negativ
falsch	1463	$p(50) - 1464$
richtig	1	0

Tabelle 5.15: Ergebnisse *LMF* mit Variante 1.1 für TEI1
1. Durchlauf

Tabelle 5.16: Ergebnisse *LMF* mit Variante 1.1 für TEI2
1. Durchlauf

Tabelle 5.17: Ergebnisse *LMF* mit Variante 1.1 für TEI3
1. Durchlauf

Die Ergebnisse für die *required Typen* *TEI4-TEI7* zeigen die folgenden Vier-Felder-Tafeln.

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.18: Ergebnisse *LMF* mit Variante 1.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p(2247) - 3$
richtig	1	0

Tabelle 5.19: Ergebnisse *LMF* mit Variante 1.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.20: Ergebnisse *LMF* mit Variante 1.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	32	$p(2775) - 33$
richtig	1	0

Tabelle 5.21: Ergebnisse *LMF* mit Variante 1.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.22: Ergebnisse *LMF* mit Variante 1.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle 5.23: Ergebnisse *LMF* mit Variante 1.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle 5.24: Ergebnisse *LMF* mit Variante 1.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	7641	$p(52150) - 7642$
richtig	1	0

Tabelle 5.25: Ergebnisse *LMF* mit Variante 1.1 für TEI7 2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *LMF* erzielt eine Reduktion der zu erzeugenden Proxies. Dies wird durch einen Vergleich der Spalte “positiv” innerhalb der Vier-Felder-Tafeln zum jeweiligen *required Typ* belegt.
2. Die Heuristik *LMF* hat keine Auswirkung auf einen Durchlauf, in dem kein Proxy erzeugt wird, mit dem die semantischen Tests erfolgreich durchgeführt werden können. Dies kann durch einen Vergleich des ersten Durchlaufs für die *required Typen TEI4-TEI7* im Ausgangspunkt (Tabellen 5.7, 5.9, 5.11 und 5.11) mit dem ersten Durchlauf unter Anwendung der Heuristik (Tabellen 5.18, 5.20, 5.22 und 5.24) festgestellt werden.

5.4 Ergebnisse für die Heuristik PTTF

Für die Heuristik PTTF gilt es zu evaluieren, ob die Suche nach einem Proxy, der die vordefinierten Tests besteht, beschleunigt werden kann. Hierzu wird die Exploration für alle der oben genannten *required Typen* unter der Verwendung der in Abschnitt 3.4.2 beschriebenen Heuristik durchgeführt.

Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse für die *required Typen* TEI1-TEI7 auf.

1	positiv	negativ
falsch	29	$p(44) - 30$
richtig	1	0

Tabelle 5.26: Ergebnisse *PTTF* für TEI1 1. Durchlauf

1	positiv	negativ
falsch	5544	$p(55) - 5545$
richtig	1	0

1	positiv	negativ
falsch	4761	$p(50) - 4762$
richtig	1	0

Tabelle 5.27: Ergebnisse *PTTF* für TEI2 1. Durchlauf

Tabelle 5.28: Ergebnisse *PTTF* für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.29: Ergebnisse *PTTF* für TEI4
1. Durchlauf

2	positiv	negativ
falsch	466	$p(2247) - 467$
richtig	1	0

Tabelle 5.30: Ergebnisse *PTTF* für TEI4
2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.31: Ergebnisse *PTTF* für TEI5
1. Durchlauf

2	positiv	negativ
falsch	2172	$p(2775) - 2173$
richtig	1	0

Tabelle 5.32: Ergebnisse *PTTF* für TEI5
2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.33: Ergebnisse *PTTF* für TEI6
1. Durchlauf

2	positiv	negativ
falsch	13122	$p(1323) - 13123$
richtig	1	0

Tabelle 5.34: Ergebnisse *PTTF* für TEI6
2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	149961	$p(52150) - 149962$
richtig	1	0

Tabelle 5.35: Ergebnisse *PTTF* für TEI7 1. Durchlauf

Tabelle 5.36: Ergebnisse *PTTF* für TEI7 2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *PTTF* erzielt eine Reduktion der zu evaluierenden Proxies. Dies wird durch einen Vergleich der Spalte “positiv” innerhalb der Vier-Felder-Tafeln zum jeweiligen *required Typ* belegt.
2. Die Heuristik *PTTF* hat keine Auswirkung auf einen Durchlauf, in dem kein Proxy erzeugt wird, mit dem die semantischen Tests erfolgreich durchgeführt werden können. Dies kann durch einen Vergleich des ersten Durchlaufs für den *required Typ* *TEI4-TEI7* im Ausgangspunkt (Tabelle 5.7, 5.9, 5.11 und 5.11) mit dem ersten Durchlauf unter Anwendung der Heuristik (Tabellen 5.29, 5.31, 5.33 und 5.35) festgestellt werden.

5.5 Ergebnisse für die Heuristik BL_NMC

Für die Heuristik *BL_NMC* gilt es zu evaluieren, ob die Suche nach einem Proxy, der die vordefinierten Tests besteht, beschleunigt werden kann. Hierzu wird die Exploration für alle der oben genannten *required Typen* unter der Verwendung der in Abschnitt 3.4.3 beschriebenen Heuristik durchgeführt.

Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse für die *required Typen* *TEI1-TEI7* auf.

1	positiv	negativ
falsch	105	$p(44) - 106$
richtig	1	0

Tabelle 5.37: Ergebnisse *BL_NMC* für TEI1 1. Durchlauf

1	positiv	negativ
falsch	342	$p(55) - 343$
richtig	1	0

1	positiv	negativ
falsch	357	$p(50) - 358$
richtig	1	0

Tabelle 5.38: Ergebnisse BL_NMC für TEI2 1. Durchlauf

Tabelle 5.39: Ergebnisse BL_NMC für TEI3 1. Durchlauf

1	positiv	negativ
falsch	120	1054
richtig	0	0

2	positiv	negativ
falsch	442	$p(2247) - 443$
richtig	1	0

Tabelle 5.40: Ergebnisse BL_NMC für TEI4 1. Durchlauf

Tabelle 5.41: Ergebnisse BL_NMC für TEI4 2. Durchlauf

1	positiv	negativ
falsch	550	4434
richtig	0	0

2	positiv	negativ
falsch	1304	$p(2775) - 1305$
richtig	1	0

Tabelle 5.42: Ergebnisse BL_NMC für TEI5 1. Durchlauf

Tabelle 5.43: Ergebnisse BL_NMC für TEI5 2. Durchlauf

1	positiv	negativ
falsch	366	685
richtig	0	0

2	positiv	negativ
falsch	204	$p(1323) - 205$
richtig	1	0

Tabelle 5.44: Ergebnisse *BL_NMC* für TEI6
1. DurchlaufTabelle 5.45: Ergebnisse *BL_NMC* für TEI6
2. Durchlauf

1	positiv	negativ
falsch	1051	160243
richtig	0	0

2	positiv	negativ
falsch	135089	$p(52150) - 135090$
richtig	1	0

Tabelle 5.46: Ergebnisse *BL_NMC* für TEI7
1. DurchlaufTabelle 5.47: Ergebnisse *BL_NMC* für TEI7
2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *BL_NMC* erzielt eine Reduktion der zu evaluierenden Proxies. Dies wird durch einen Vergleich der Spalte “positiv” innerhalb der Vier-Felder-Tafeln zum jeweiligen *required Typ* belegt.
2. Die Heuristik *BL_NMC* hat das Potential jeden Durchlauf innerhalb der semantischen Evaluation zu beschleunigen. Für den jeweils ersten Durchlauf kann dies durch einen Vergleich der Tabellen 5.4, 5.5, 5.6, 5.7, 5.9, 5.11 und 5.13 im Ausgangspunkt mit den Tabellen 5.37, 5.38, 5.39, 5.40, 5.42, 5.44 und 5.46 festgestellt werden. Ein Vergleich der Tabelle 5.8, 5.10, 5.12 und 5.14 im Ausgangspunkt mit den Tabellen 5.41, 5.43, 5.45 und 5.47 zeigt diesen Fakt für den zweiten Durchlauf auf.

Aus den Ergebnissen, die in den Abschnitten 5.3 - 5.5 beschrieben wurden, lässt sich je required Typ eine Rangfolge der vorgestellten Heuristiken erstellen. Diese Rangfolge kann Tabelle 5.48 entnommen werden. Dabei gilt, dass die Heuristik, mit der am wenigsten Proxies generiert und evaluiert werden mussten, den ersten Platz einnimmt.

Heuristik/Required Typ	TEI1	TEI2	TEI3	TEI4	TEI5	TEI6	TEI7
LMF	1.	2.	2.	2.	2.	2.	2.
PTTF	3.	3.	3.	3.	3.	3.	3.
BLNMC	2.	1.	1.	1.	1.	1.	1.

Tabelle 5.48: Rangfolge der Heuristiken (Einzelbetrachtung)

5.6 Ergebnisse für die Kombination der Heuristiken

Im vorherigen Abschnitt wurde gezeigt, dass die Exploration durch jede der beschriebenen Heuristiken beschleunigt werden kann. Dabei wurden Exploration mit jeweils einer der Heuristiken durchgeführt. In den folgenden Abschnitten soll evaluiert werden, ob die Verwendung einer Kombination der einzelnen Heuristiken bei der Exploration einen zusätzlichen Vorteil bringt.

Hierzu werden die Ergebnisse aller Kombinationen der einzelnen Heuristiken aufgeführt und im Anschluss bewertet.

5.6.1 Kombination: LMF + PTTF

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken *LMF* und *PTTF*.

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	1877	$p(55) - 1878$
richtig	1	0

1	positiv	negativ
falsch	1473	$p(50) - 1474$
richtig	1	0

Tabelle 5.49: Ergebnisse *LMF* + *PTTF* für TEI1

Tabelle 5.50: Ergebnisse *LMF* + *PTTF* für TEI2 1. Durchlauf

Tabelle 5.51: Ergebnisse *LMF* + *PTTF* für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.52: Ergebnisse $LMF + PTTF$ für TEI4 1. Durchlauf

2	positiv	negativ
falsch	4	$p(2247) - 5$
richtig	1	0

Tabelle 5.53: Ergebnisse $LMF + PTTF$ für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.54: Ergebnisse $LMF + PTTF$ für TEI5 1. Durchlauf

2	positiv	negativ
falsch	34	$p(2346) - 35$
richtig	1	0

Tabelle 5.55: Ergebnisse $LMF + PTTF$ für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.56: Ergebnisse $LMF + PTTF$ für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle 5.57: Ergebnisse $LMF + PTTF$ für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle 5.58: Ergebnisse $LMF + PTTF$ für TEI7 1. Durchlauf

2	positiv	negativ
falsch	1076	$p(52150) - 1077$
richtig	1	0

Tabelle 5.59: Ergebnisse $LMF + PTTF$ für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf wirkt sich die Kombination der Heuristiken LMF und $PTTF$ nicht nennenswert während der Exploration aus. Da mit der Heuristik LMF alleine bessere Ergebnisse erzielt wurden als mit der Heuristik $PTTF$, müssen hierfür die Tabellen 5.15, 5.16, 5.17, 5.18, 5.20, 5.22 und 5.24 mit den Tabellen 5.49, 5.50, 5.51, 5.52, 5.54, 5.56 und 5.58 verglichen werden.
2. Für den zweiten Durchlauf während der Exploration ist eine Verbesserung zu erkennen. Diese bezieht sich jedoch nur auf die Exploration für $TEI7$ (vergleiche Tabelle 5.25 aus Abschnitt 5.3 mit Tabelle 5.59).

5.6.2 Kombination: $LMF + BL_NMC$

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken LMF und BL_NMC .

1	positiv	negativ
falsch	0	$p(44) - 1$
richtig	1	0

1	positiv	negativ
falsch	83	$p(55) - 84$
richtig	1	0

1	positiv	negativ
falsch	89	$p(50) - 90$
richtig	1	0

Tabelle 5.60: Ergebnisse $LMF + BL_NMC$ für TEI1

Tabelle 5.61: Ergebnisse $LMF + BL_NMC$ für TEI2 1. Durchlauf

Tabelle 5.62: Ergebnisse $LMF + BL_NMC$ für TEI3 1. Durchlauf

1	positiv	negativ
falsch	120	1054
richtig	0	0

Tabelle 5.63: Ergebnisse $LMF + BL_NMC$ für TEI4 1. Durchlauf

2	positiv	negativ
falsch	4	$p(2247) - 5$
richtig	1	0

Tabelle 5.64: Ergebnisse $LMF + BL_NMC$ für TEI4 2. Durchlauf

1	positiv	negativ
falsch	550	4434
richtig	0	0

Tabelle 5.65: Ergebnisse $LMF + BL_NMC$ für TEI5 1. Durchlauf

2	positiv	negativ
falsch	34	$p(2346) - 35$
richtig	1	0

Tabelle 5.66: Ergebnisse $LMF + BL_NMC$ für TEI5 2. Durchlauf

1	positiv	negativ
falsch	115	936
richtig	0	0

Tabelle 5.67: Ergebnisse $LMF + PTTF$ für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle 5.68: Ergebnisse $LMF + PTTF$ für TEI6 2. Durchlauf

1	positiv	negativ
falsch	2448	158846
richtig	0	0

Tabelle 5.69: Ergebnisse $LMF + BL_NMC$ für TEI7 1. Durchlauf

2	positiv	negativ
falsch	954	$p(52150) - 955$
richtig	1	0

Tabelle 5.70: Ergebnisse $LMF + BL_NMC$ für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf während der Exploration wirkt sich die Kombination der Heuristiken LMF und BL_NMC positiv aus. Hierzu sind die Tabelle 5.60 mit der Tabelle 5.15 aus Abschnitt 5.3 sowie die Tabellen 5.61, 5.62 und 5.67 mit den Tabellen 5.38, 5.39, 5.39 und 5.44 aus Abschnitt 5.5 zu vergleichen.
2. Für den zweiten Durchlauf während der Exploration ist ebenfalls eine Verbesserung zu erkennen. Diese bezieht sich jedoch nur auf die Exploration für $TEI7$ (vergleiche Tabelle 5.25 aus Abschnitt 5.3 mit Tabelle 5.70).

5.6.3 Kombination: PTTF + BL_NMC

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken $PTTF$ und BL_NMC .

1	positiv	negativ
falsch	104	$p(44) - 105$
richtig	1	0

1	positiv	negativ
falsch	337	$p(55) - 338$
richtig	1	0

1	positiv	negativ
falsch	357	$p(50) - 358$
richtig	1	0

Tabelle 5.71: Ergebnisse $PTTF + BL_NMC$ für TEI1

Tabelle 5.72: Ergebnisse $PTTF + BL_NMC$ für TEI2
1. Durchlauf

Tabelle 5.73: Ergebnisse $PTTF + BL_NMC$ für TEI3
1. Durchlauf

1	positiv	negativ
falsch	120	1054
richtig	0	0

2	positiv	negativ
falsch	47	$p(2247) - 48$
richtig	1	0

Tabelle 5.74: Ergebnisse $PTTF + BL_NMC$ für TEI4 1. DurchlaufTabelle 5.75: Ergebnisse $PTTF + BL_NMC$ für TEI4 2. Durchlauf

1	positiv	negativ
falsch	550	4434
richtig	0	0

2	positiv	negativ
falsch	219	$p(2346) - 220$
richtig	1	0

Tabelle 5.76: Ergebnisse $PTTF + BL_NMC$ für TEI5 1. DurchlaufTabelle 5.77: Ergebnisse $PTTF + BL_NMC$ für TEI5 2. Durchlauf

1	positiv	negativ
falsch	366	685
richtig	0	0

2	positiv	negativ
falsch	204	$p(1323) - 205$
richtig	1	0

Tabelle 5.78: Ergebnisse $PTTF + PTTF$ für TEI6 1. DurchlaufTabelle 5.79: Ergebnisse $PTTF + PTTF$ für TEI6 2. Durchlauf

1	positiv	negativ
falsch	1036	160258
richtig	0	0

Tabelle 5.80: Ergebnisse $PTTF + BL_NMC$ für TEI7 1. Durchlauf

2	positiv	negativ
falsch	6015	$p(52150) - 6016$
richtig	1	0

Tabelle 5.81: Ergebnisse $PTTF + BL_NMC$
für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf während der Exploration hat die Kombination der Heuristiken $PTTF$ und BL_NMC keine Auswirkung. Die Ergebnisse sind nahezu identisch mit denen der Exploration mit der Heuristik BL_NMC aus Abschnitt 5.5. (Vergleiche Tabellen 5.71, 5.72, 5.73, 5.74, 5.76, 5.67 und 5.80 mit den Tabellen 5.37, 5.38, 5.39, 5.40, 5.42, 5.44 und 5.46.
2. Für den zweiten Durchlauf während der Exploration ist eine Verbesserung zu erkennen. Da mit der Heuristik BL_NMC bessere Ergebnisse erzielt wurden als mit der Heuristik $PTTF$ (vergleiche Ergebnisse aus Abschnitt 5.5 mit den Ergebnissen aus Abschnitt 5.4), kann dies durch einen Vergleich der Tabellen 5.75, 5.77, 5.68 und 5.81 mit den Tabellen 5.41, 5.43, 5.45 und 5.47 belegt werden.

5.6.4 Kombination: LMF + PTTF + BL_NMC

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken LMF , $PTTF$ und BL_NMC .

1	positiv	negativ
falsch	2	$p(44) - 3$
richtig	1	0

1	positiv	negativ
falsch	79	$p(55) - 80$
richtig	1	0

Tabelle 5.82: Ergebnisse $LMF + PTTF + BL_NMC$ für TEI1

Tabelle 5.83: Ergebnisse $LMF + PTTF + BL_NMC$ für TEI2 1. Durchlauf

1	positiv	negativ
falsch	86	$p(50) - 87$
richtig	1	0

Tabelle 5.84: Ergebnisse LMF + $PTTF$ + BL_NMC für TEI3 1. Durchlauf

1	positiv	negativ
falsch	120	1054
richtig	0	0

2	positiv	negativ
falsch	4	$p(2247) - 5$
richtig	1	0

Tabelle 5.85: Ergebnisse LMF + $PTTF$ + BL_NMC für TEI4 1. Durchlauf

Tabelle 5.86: Ergebnisse LMF + $PTTF$ + BL_NMC für TEI4 2. Durchlauf

1	positiv	negativ
falsch	550	4434
richtig	0	0

2	positiv	negativ
falsch	34	$p(2346) - 35$
richtig	1	0

Tabelle 5.87: Ergebnisse LMF + $PTTF$ + BL_NMC für TEI5 1. Durchlauf

Tabelle 5.88: Ergebnisse LMF + $PTTF$ + BL_NMC für TEI5 2. Durchlauf

1	positiv	negativ
falsch	115	936
richtig	0	0

Tabelle 5.89: Ergebnisse LMF + $PTTF$ + $PTTF$ für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle 5.90: Ergebnisse $LMF + PTTF + PTTF$ für TEI6 2. Durchlauf

1	positiv	negativ
falsch	2448	158846
richtig	0	0

2	positiv	negativ
falsch	12	$p(52150) - 13$
richtig	1	0

Tabelle 5.91: Ergebnisse $LMF + PTTF + BL_NMC$ für TEI7 1. Durchlauf

Tabelle 5.92: Ergebnisse $LMF + PTTF + BL_NMC$ für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf während der Exploration wirkt sich die Kombination der Heuristiken LMF , $PTTF$ und BL_NMC nicht besser aus als die Kombination der Heuristiken LMF und BL_NMC (siehe Abschnitt 5.6.2). Die Ergebnisse sind nahezu identisch.
2. Für den zweiten Durchlauf während der Exploration gilt zumindest für die *required Typen* TEI_4 - TEI_6 dasselbe, wie für den ersten Durchlauf. Für den *required Typ* TEI_7 ist hingegen nochmals eine Verbesserung im Vergleich zu den 2er-Kombinationen (siehe Abschnitte 5.6.1-5.6.3) zu erkennen.

Wie bei der Einzelbetrachtung der Heuristiken lässt sich auch eine Rangfolge der Kombinationen

von Heuristiken je required Typ erstellen. Diese Rangfolge kann Tabelle 5.93 entnommen werden. Dabei gilt wiederum, dass die Kombination von Heuristiken, mit der am wenigsten Proxies generiert und evaluiert werden mussten, den ersten Platz einnimmt. Sofern mehrere Kombinationen von Heuristiken bzgl. dessen gleich aufliegen, wird dies durch eine Doppelplatzierung dargestellt.

Heuristik/Required Typ	TEI1	TEI2	TEI3	TEI4	TEI5	TEI6	TEI7
LMF + PTTF	3.	4.	4.	4.	4.	4.	4.
LMF + BL_NMC	1.	2.	2.	1./2.	1./2.	1./2.	2.
PTTF + BL_NMC	4.	3.	3.	3.	3.	3.	3.
LMF + PTTF + BL_NMC	2.	1.	1.	1./2.	1./2.	1./2.	1.

Tabelle 5.93: Rangfolge der Heuristiken (Kombinationen)

Kapitel 6

Diskussion

In den folgenden Abschnitten werden die Untersuchungsergebnisse aus Kapitel 5 ausgewertet und die Vor- und Nachteile des Ansatzes zur Exploration von EJBs zur Laufzeit gegenüber gestellt. Darüber hinaus werden Erweiterungsmöglichkeiten bzgl. der Deklaration von required Typen und der Matcher, sowie deren zu erwartende Auswirkung auf die Exploration beschrieben. Aufbauend auf den Vor- und Nachteilen des beschriebenen Ansatzes zur testgetriebenen Exploration von EJBs zur Laufzeit werden außerdem Erweiterungsvorschläge des Ansatzes vorgestellt.

6.1 Auswertung der Untersuchungsergebnisse

6.1.1 Einzelbetrachtung

Die in Kapitel 5 beschriebenen Untersuchungsergebnisse zeigen, dass die Heuristiken die Anzahl der zu generierenden und zu evaluierenden Proxies reduzieren. Dabei zeigt sich, dass sich die Heuristiken nicht auf alle Explorationsdurchläufe positiv auswirken. So kann für die Heuristiken *LMF* und *PTTF* festgehalten werden, dass diese nur in dem Durchlauf eine positive Wirkung erzielt, in dem ein passender Proxy auch gefunden wird.

Die Heuristik *BL-NMC* hingegen wirkt sich auf jeden der durchgeführten Durchläufe aus. Dies liegt zum einen daran, dass die Menge der Informationen, auf deren Basis sie arbeitet, während eines Durchlaufs anwächst. Bei der Heuristik *LMF* ist dies nicht der Fall. Allerdings weist die

Heuristik *PTTF* ebenfalls dieses Merkmal auf.

Ein weiterer Grund ist, dass die Heuristik *BL_NMC* dafür sorgt, dass Proxies bei der Evaluierung mitunter übersprungen werden, oder diese gar nicht erst generiert werden. Die anderen Heuristiken hingegen sorgen lediglich für eine Umsortierung der zu generierenden bzw. zu evaluierenden Proxies. Somit müssen unter der Verwendung der Heuristiken *LMF* und *PTTF* im Zweifelsfall alle Proxies generiert und erzeugt werden, auch wenn kein passender Proxy ausgemacht werden kann.

Weiterhin ist festzuhalten, dass mit der Heuristik *BL_NMC* scheinbar die besten Ergebnisse erzielt werden. Eine Ausnahme bildet hier lediglich die Exploration zum required Typ *ElerFTFoerderprogrammeProvider* (*TEI1*). Die Ursache dafür liegt darin begründet, dass die in den Methoden von *TEI1* verwendeten *provided Typen* mit denen des erwarteten *provided Typen*, auf dessen Basis ein passender Proxy erzeugt wird, genau übereinstimmen. Damit wird ein vergleichsweise geringes Matcherrating für das Matching dieser beiden Typen ermittelt, wodurch der Proxy sehr früh während der Exploration generiert und evaluiert wird.

6.1.2 Synergien

Neben der Einzelbetrachtung der Heuristiken wurden in Abschnitt 5.6 auch die Kombinationen der drei Heuristiken untersucht. Aus den Feststellungen in Abschnitt 6.1.1 lässt sich ableiten, dass eine Kombination mit der Heuristik *BL_NMC* durchaus sinnvoll ist, egal ob sie mit der Heuristik *LMF* oder *PTTF* kombiniert wird. Der Grund dafür liegt wiederum in der Tatsache, dass die Heuristiken *LMF* und *PTTF* lediglich auf einen der Explorationsdurchläufe einen positiven Effekt haben. Aus diesem Grund kann in Kombination mit der Heuristik *BL_NMC* wenigstens in den anderen Durchläufen eine positive Auswirkung festgestellt werden.

Dementgegen liefert die Kombination der Heuristiken *LMF* und *PTTF* miteinander kaum bessere Ergebnisse als die Heuristik *LMF* alleine. Eine Ausnahme bildet der required Typ *KOFGPCProvider* (*TEI7*). Dazu ist jedoch zu sagen, dass gerade zu diesem *required Typ* im Vergleich zu den anderen required Typen die meisten matchenden *provided Typen* existieren. Insofern darf dieser scheinbare Ausreißer nicht unterschätzt werden, weshalb auch die Kombi-

nation der oben genannten Heuristiken sinnvoll ist.

Ähnliches gilt für die Kombination aller vorgestellten Heuristiken ($LMF + PTTF + BL_NMC$). Dies ergibt sich ebenfalls aus den vorherigen Auswertungen bzgl. der Synergien in diesem Abschnitt. Bei der Betrachtung der Untersuchungsergebnisse zeigt sich hier ein ähnliches Muster wie zuvor: Die Kombination aller vorgestellten Heuristiken liefert nur für den *required Typ* `KOFGPCProvider` (*TEI7*) bessere Ergebnisse, als die Kombination der Heuristiken *BL_NMC* und *LMF*. Aber auch hier darf dieses Ergebnis aufgrund der Eigenschaften von *TEI7* nicht vernachlässigt werden.

6.1.3 Erhöhte Komplexität

Die vorliegende Untersuchung zeigt, dass die Anzahl der zu evaluierenden Proxies in dem verwendeten System mit den vorgeschlagenen Heuristiken reduziert werden können. Allerdings wurden negative Auswirkungen wie bspw. Speichernutzung (Speicherkomplexität) oder die benötigte Zeit (Zeitkomplexität) für die Exploration nicht untersucht.

Die Anwendung der Heuristiken hängt, wie in Abschnitt 3.4 beschrieben, von Informationen ab, die teilweise aus den für die Proxies verwendeten *provided Typen* ermittelt werden müssen (Matcherrating) bzw. nach der Ausführung der Tests über die gesamte restliche Laufzeit der Exploration verwaltet werden müssen. Von daher ist davon auszugehen, dass sich die Anwendung der Heuristiken durchaus auf den Speicherverbrauch auswirkt.

Da die benötigte Zeit für die Verwaltung von Listen, wie sie bei den Heuristiken vorgenommen wird, mit der Anzahl der zu verwaltenden Elemente wächst, kann davon ausgegangen werden, dass die Anwendung der Heuristiken ebenfalls mehr Zeit in Anspruch nimmt, je weiter fortgeschritten die Exploration ist. Dies gilt insbesondere für die Heuristiken *PTTF* und *BL_NMC*.

Aufgrund dessen, dass in dieser Arbeit lediglich die Anzahl der zu evaluierenden Proxies während der Exploration untersucht wurden, ist es auch nicht auszuschließen, dass die verwendete Implementierung kein Optimierungspotential besitzt.

6.1.4 Zusammenfassung

Die Ausführungen der Abschnitte 6.1.1 und 6.1.2 lassen vermuten, dass lediglich die Heuristiken *LMF* und *BL_NMC* eine Daseinsberechtigung haben. Dies ist nicht korrekt. Die Heuristik *PTTF* liefert zwar schlechtere Ergebnisse, dennoch hat sie die zu generierenden und zu prüfenden Proxies im Vergleich zum schlimmsten Fall ohne Heuristiken stark reduziert. Allerdings hat der Entwickler keinen höheren Aufwand bei der Implementierung der Testfälle. Die Heuristik *BL_NMC*, welche sich in dieser Untersuchung häufig als diejenige mit den besten Ergebnissen herausgestellt hat, bedarf einer speziellen Implementierung der Testfälle.

Dasselbe gilt für die Heuristik *LMF*. Diese liefert zwar bessere Ergebnisse als die Heuristik *PTTF*, kann aber aufgrund dessen, dass sie sich lediglich auf den finalen Explorationsdurchlauf positiv auswirkt, nur in wenigen Fällen mit der Heuristik *BL_NMC* mithalten. Allerdings gilt auch hier, dass keine weiteren Anforderungen an die Arbeit des Entwicklers gestellt werden. Dazu kommt noch, dass die Ermittlung der Matcherratings quasi bei dem Matching der Typen mit abfällt, wodurch die Verwendung dieser Heuristik kaum eine Auswirkung auf die Komplexität der Exploration hat.

6.2 Kritik am Ansatz

6.2.1 Seiteneffekte durch Testevaluation

Die Exploration erfordert die Ausführung der vordefinierten Testfälle zur Laufzeit. Sofern diese Testfälle eine Änderung des Zustands bestimmter Objekte bewirken, kann dies auch Auswirkungen auf die Funktionsweise des Systems haben.

Um dieses Problem zu beheben könnte man sicherstellen, dass die Generierung der Proxies nur auf Basis von *provided Typen* erfolgt, die solche Seiteneffekte nicht aufweisen. Diese Eigenschaft kann jedoch nur durch den Entwickler festgestellt und entsprechend markiert werden (bspw. über Annotationen). Während der Exploration könnten solche *provided Typen* über solche Markierungen erkannt werden. Dieser Ansatz reduziert jedoch die Anzahl der *provided Typen*, die für die Generierung eines Proxies verwendet werden können. Dadurch sinkt auch die

Wahrscheinlichkeit, dass ein passender Proxy gefunden wird.

Um die zu markierenden *EJBs* zu identifizieren ist zu prüfen, wie sich die Ausführung der einzelnen Methoden der Bean auf das System auswirken. Es kann festgehalten werden, dass alle Methoden, die den persistenten oder den transienten Zustand von Objekten verändern, das Potential für solche unerwünschten Seiteneffekte besitzen.

Aufbauend auf der Prüfung einzelner Methoden, kann auch die Markierung von Methoden in Betracht gezogen werden. So dürften markierte Methoden bei der Generierung eines Proxies nicht als Delegationsmethode verwendet werden.

6.2.2 Auswirkung auf die Verfügbarkeit eines Systems

Die Verfügbarkeit eines Systems bzw. von Systemkomponenten, bezeichnet die Wahrscheinlichkeit, ein System oder Systemkomponenten zu einem vorgegebenen Zeitpunkt in einem funktionsfähigen Zustand anzutreffen. [Adm21] Die Auswirkung des Ansatzes auf die Verfügbarkeit wurde in dieser Arbeit nicht systematisch untersucht. Da der Ansatz jedoch darauf abzielt, bestimmte Komponenten (EJBs) zur Laufzeit zu kombinieren, können Überlegungen bzgl. der Verfügbarkeit durchaus angestellt werden.

Dabei muss allerdings bedacht werden, dass die Verfügbarkeit der Komponente (also der EJB) in diesem Zusammenhang nicht ausschlaggebend ist. Immerhin wird sie nicht direkt adressiert, sondern auf Basis struktureller und semantischer Vorgaben ermittelt. Insofern bilden eher die Methoden, die von den EJBs angeboten werden, die Komponenten in Bezug auf die hier betrachtete Verfügbarkeit.

Ausgehend davon kann die These aufgestellt werden, dass mit diesem Ansatz eine höhere Verfügbarkeit erreicht wird, sofern die Methoden im System redundant vorliegen.

6.2.3 Auswirkung von Änderungen an bestehenden Komponenten

Da die EJBs bei dem vorgestellten Ansatz nicht explizit adressiert werden, weiß der Entwickler auch nicht, an welche EJBs die Methodenaufrufe letztendlich delegiert werden. Somit sind die Auswirkungen von Änderungen an bestehenden Komponenten nicht direkt vorhersehbar, da sich die Menge der matchenden provided Typen (EJBs) und dementsprechend auch die generierten Proxies ändern.

Im Folgenden wird zum Einen die Erweiterung von zusätzlichen provided Typen und zum Anderen die Entfernung von provided Typen betrachtet. Dabei sei angenommen, dass die required Typen, zu denen ein passender Proxy gefunden werden soll, nicht verändert werden.

Erweiterungen um neue Komponenten

Die Erweiterung von Systemen geht in Bezug auf den beschriebenen Ansatz zur testgetriebenen Exploration zur Laufzeit damit einher, dass sich die Anzahl der provided Typen verändert. Wie in Abschnitt 3.2.4 beschrieben, besteht damit auch die Gefahr, dass die Anzahl der möglichen Proxies steigt. Dazu muss jedoch gelten, dass eine Methode im neuen provided Typ mit einer Methode eines required Typ gematcht werden kann.

Mehrere mögliche Proxies haben wiederum einen Einfluss auf die Laufzeit und das Ergebnis der Exploration. So kann nicht davon ausgegangen werden, dass ein passender Proxy zu einem bestimmten required Typ genauso schnell gefunden wird, nachdem sich Änderungen an den provided Typen im System ergeben haben.

Entfernen von bestehenden Komponenten

Ebenso wirkt sich das Entfernen eines provided Typs, der bei einer früheren Exploration für die Generierung eines Proxies verwendet wurde, auf die Exploraiton nach einer solchen Änderung aus. Dadurch, dass der früher verwendete provided Typ nicht mehr vorhanden ist, muss ein

anderer Proxy, der auf anderen provided Typen basiert, erzeugt werden¹.

Da die Exploration beendet wird, sofern ein passender Proxy gefunden wurde, kann es auch unter diesen Umständen dazu kommen, dass die Exploration mitunter länger dauert als vorher. Zudem besteht in diesem Fall die Gefahr, dass die Exploration fehlschlägt.

6.2.4 Nutzen für den Entwickler

Aus den vorherigen Absätzen ergibt sich, dass der Entwickler bei der Verwendung dieses Ansatzes eine große Verantwortung trägt. Dieser Verantwortung kann er umso besser gerecht werden, je genauer er das System, in dem der Ansatz verwendet werden soll, kennt.

So kann festgehalten werden, dass ein Entwickler, der das System gut kennt und somit weiß, welche Komponenten innerhalb dessen verwendet werden, diesen Ansatz wohl kaum benötigt. Vielmehr ist es ihm möglich die passenden Komponenten aufgrund seines Wissens explizit zu benennen, wie es im EJB-Framework grundlegend der Fall ist.

Ein Entwickler, der das System hingegen weniger kennt, kann von diesem Ansatz profitieren, da er nicht selbst nach einer für ihn passenden EJB (mitunter auch mehreren) suchen muss. Diese kann er über die Deklaration eines required Typen und der Spezifikation dazugehöriger Tests suchen lassen. Dabei ist jedoch zu erwähnen, dass die Exploration insbesondere mit der vorgestellten Heuristik LMF umso schneller ist, je genauer die in den Methoden des required Typs verwendeten Typen mit den Typen, die in den Methoden der provided Typen übereinstimmen (Matcherrating).

Ist dem Entwickler das System unbekannt, wird es schwerfallen einen required Typ so zu deklarieren, dass die Anzahl der möglichen Proxies nicht zu hoch wird.

Zusammenfassend kann folgende These formuliert werden: Der Nutzen dieses Ansatzes für einen Entwickler in Bezug auf ein System steht im umgekehrt proportionalen Verhältnis zum Wissen dieses Entwicklers über das System.

¹sofern dies gelingt, unterstützt dies die These aus Abschnitt 6.2.2

6.3 Erweiterungsmöglichkeiten

6.3.1 Zusätzliche Matcher

Eine mögliche Erweiterung des Ansatzes wäre die Definition und Implementierung zusätzlicher Matcher. Diese würde es ermöglichen, dass der Abstraktionsgrad zwischen den Typen, die in den Methoden der `required` und `provided` Typen verwendet werden, noch weiter auseinandergeht, als es bei den vorgestellten Matchern in Abschnitt 3.1.2 der Fall ist (Identität, Vererbung, Container).

Die vorgestellten Matcher beachten beispielsweise keine impliziten Typumwandlungen (Coercions). Diese können je nach Programmiersprache abweichen, was eine formale und allgemeine Beschreibung wie in Abschnitt 3.1.2 eines solchen Matchers (CoercionMatcher) erschwert. So müsste ein CoercionMatcher für jede Programmiersprache explizit spezifiziert werden.

Die Programmiersprache Java bietet eine Vielzahl solcher impliziten Typumwandlungen an [GJS⁺13]. Dabei ist zu beachten, dass es implizite Typumwandlungen gibt, die ohne Informationsverlust vonstatten gehen² und solche, bei denen ein Informationsverlust nicht auszuschließen ist³.

Typumwandlungen ohne Informationsverlust sind in Bezug auf die weitere Verwendung innerhalb eines Proxies unbedenklich. Diese sind hinsichtlich des Informationsverlustes mit dem GenTypeMatcher vergleichbar, welcher in Abschnitt 3.1.2 beschrieben wurde. In der Spezifikation des darauf aufbauenden Proxy-Generators sind dementsprechend keine Methodendelegationen zu finden, die zu einem Fehler führen.

Anders ist es bei Typumwandlungen mit Informationsverlust. Diese sind mit dem SpecTypeMatcher vergleichbar (siehe Abschnitt 3.1.2). In der Spezifikation des darauf aufbauenden Proxy-Generators ist zu erkennen, dass durch eine solche Typumwandlung bestimmte Methodendelegationen in einen Fehler münden. Da sich der SpecTypeMatcher direkt auf die Verer-

²bspw. *Identity Conversion* oder *Widening Primitive Conversion* [GJS⁺13]

³bspw. *Narrowing Primitive Conversion* [GJS⁺13]

bungsbeziehung der beiden Typen bezieht, kann die Ursache solcher Fehler auf die Methoden zurückgeführt werden, die zwar im Subtyp jedoch nicht im Supertyp implementiert sind. Bei einem CoercionMatcher, der in Abhängigkeit der Programmiersprache spezifiziert wird, kann es weitere Fehlerursachen geben.

Aus diesem Grund wäre es sinnvoll, nicht einen einzigen Matcher zu spezifizieren, der alle impliziten Typumwandlungen abdeckt. Vielmehr sollten die in der Programmiersprache definierten Coercions nach dem möglichem Informationsverlust kategorisiert werden und dann je Kategorie ein Matcher spezifiziert werden.

Darüber hinaus ist zu beachten, dass die Spezifikation eines Matchers alleine nicht ausreicht, um diesen zu integrieren. Da die Heuristik LMF auf dem Matcherrating aufbaut, ist es ebenso notwendig, den zusätzlichen Matchern ein Basisrating zuzuweisen. Wie in Abschnitt 4.1 beschrieben, wird dieses Basisrating von der Implementierung des Matchers geliefert. Dabei gilt es jedoch zu beachten, dass das Basisrating eines zusätzlichen Matchers im korrekten Verhältnis zu den bestehenden Matchern steht.

In Bezug auf den/die CoercionMatcher gibt es hierbei mehrere sinnvolle Möglichkeiten. Beispielsweise könnte man begründen, dass für den/die CoercionMatcher ein Basisrating zwischen 100 und 200 verwendet werden muss. Die untere Schranke von 100 wird dadurch begründet, dass es kein besseres Matching gibt, als die Identität, welche durch den ExactTypeMatcher mit einem Basisrating von 100 beschrieben wird. Die obere Schranke von 200 könnte damit begründet werden, dass es sich um Typumwandlungen handelt, die über die Programmiersprache definiert sind und diese somit sicherer sind als Upcasts, die durch den SpecTypeMatcher mit einem Basisrating von 200 abgedeckt werden.

6.3.2 Default-Implementierungen in required Typen

Im Abschnitt 2.2 wurde darauf aufmerksam gemacht, dass die Exploration das Auffinden eines passenden Proxies nicht garantiert. Der Entwickler muss also in einem solchen Fall eine alternative Implementierung bereitstellen.

Dass ein passender Proxy nicht gefunden wurde, kann allgemein betrachtet zwei Ursachen haben: Entweder konnte kein Proxy generiert werden, oder keiner der generierten Proxies erfüllt alle vordefinierten Test.

Die Generierung eines Proxies hängt von dem Matching der Methoden des required Typs und der Methoden der provided Typen ab. Aufgrund dessen dass der Entwickler Testfälle für den required Typ spezifizieren muss, hat er eine grundlegende Vorstellung von den Ein- und Ausgabewerten der Methoden, sowie der Verarbeitung dieser. Um nun der Gefahr vorzubeugen, dass gar kein Proxy generiert werden kann, könnte der Entwickler eine Implementierung, die seine Erwartungen zumindest minimal erfüllt, als default-Methoden in dem Interface zum required Typ aufnehmen. Sofern bei der Exploration zu dieser Methode keine passende Methode aus einem provided Typ gefunden wird, kann auf die Default-Implementierung zurückgegriffen werden. Der generierte Proxy, welcher technisch gesehen das Interface zum required Typ implementiert, würde den Methodenaufruf dann an sich selbst bzw. an die default-Methode delegieren.

Ein Beispiel für eine solche Konstellation zeigen die folgenden Listings. In Listing 6.1 ist der required Typ *Calc* deklariert. Listing 6.2 zeigt das dazugehörige Java-Interface mit der default-Implementierung der Methode *div*. Die Implementierung wurde so umgesetzt, dass die Testfälle, welche in der Klasse in Listing 6.3 enthalten sind, positiv ausfallen.

```
required Calc {
    Float div( int a, int b )
}
```

Listing 6.1: Required Typ *Calc*

```
@RequiredTypeTestReference( testClasses = CalcTest.class )
public interface Calc {

    default Float div(int a, int b){
        if(b == 0)
            return null;
        return Float.valueOf(a/b)
    }
}
```

Listing 6.2: Interface *Calc*

```
public class CalcTest {

    private Calc calc;

    @RequiredTypeInstanceSetter
    public void setProvider( Calc calc ) {
        this.calc = calc;
    }

    @RequiredTypeTest
    public void testDivByZero() {
        assertThat( calc.dev(1,0), nullValue() );
    }

    @RequiredTypeTest
    public void testDiv() {
        assertThat( calc.dev(4,2), equalTo(2) );
    }
}
```

Listing 6.3: Test CalcTest

Dadurch ist zwar immer noch nicht sichergestellt, dass ein passender Proxy in jedem Fall gefunden wird, aber der Entwickler kann ein alternatives Verhalten direkt im Interface zum *required Typ* implementieren, wodurch diese Implementierung einen sehr engen Bezug zum *required Typ* hat.

Kapitel 7

Schlussbemerkung

7.1 Zusammenfassung

Zusammenfassend ist zu sagen, dass die vorgestellten Heuristiken ihren Zweck erfüllen und gemessen an der Anzahl der zu generierenden und zu evaluierenden Proxies eine schnellere Exploration nach einem passenden Proxy ermöglichen. Dabei konnten auch Synergieeffekte zwischen den einzelnen Heuristiken festgestellt werden.

Weiterhin wurde gezeigt, dass die testgetriebene Exploration von EJBs zur Laufzeit grundlegend funktioniert. Dennoch gibt es Szenarien, in denen von diesem Verfahren eher abzuraten ist. Das betrifft insbesondere solche EJBs, durch deren Methodenaufrufe eine Änderung an ihrem inneren Zustand bezweckt wird. Es wurden jedoch Möglichkeiten aufgezeigt, wie mit solchen Konstellationen umgegangen werden kann.

Ob der Ansatz der testgetriebenen Exploration zur Laufzeit im Allgemeinen einen Nutzen verspricht wurde nicht geklärt. Wenn dies überhaupt der Fall ist, dann hängt der Nutzen vermutlich mit dem Wissen des jeweiligen Entwicklers zusammen, welches er über das vorliegende System aufweisen kann.

Unabhängig davon wurde in dieser Arbeit eine allgemeine formale Beschreibung für Typen gegeben, die in anderen Typen enthalten sind (ContentTypeMatcher bzw. ContainerTypeMat-

cher).

Zudem können die entwickelten Module, welche in Kapitel 4 beschrieben wurden, in unterschiedlichen Systemen verwendet werden. Hinsichtlich des Repositories hat der Entwickler sehr viel Freiraum und ist nicht auf einen EJB-Container beschränkt. Weiterhin können neue Matcher durch die Implementierung der dafür vorgesehenen Interfaces in die Module integriert werden, was den Nutzen für ein System individuell steigern kann.

7.2 Ausblick

Die Heuristiken wurden zwar im Rahmen der Exploration zur Laufzeit entworfen. In einem nächsten Schritt könnte versucht werden, diese Heuristiken in bestehende Search Engines wie Merobase oder CodeGenie zu integrieren, um so den Nutzen der Heuristiken für diese Engines zu untersuchen.

Weiterhin wäre es interessant zu untersuchen, ob und wie dieser Ansatz der Exploration von Komponenten zur Laufzeit in anderen Systemtypen wie bspw. Self-Contained-Systems funktioniert. Mitunter ergeben sich bei diesen Untersuchungen weitere Vorteile oder Probleme dieses Ansatzes.

Darüber hinaus bieten die in Abschnitt 6.2 aufgestellten Thesen bzgl. der höheren Verfügbarkeit (Abschnitt 6.2.1) und dem Nutzen des Ansatzes für den Entwickler im Verhältnis zu dessen Wissen über das System das Potential für weitere Untersuchungen.

Glossar

Abstrakter Syntaxbaum . 111

Artefakt Ein Artefakt beschreibt in der Software-Entwicklung die Spezifikation einer physischen Informationseinheit als Ergebnis des Software-Entwicklungsprozesses oder dem Deployment bzw. der Ausführung eines Systems. In der UML Spezifikation 2.1.2 [Obj07] werden u.a. folgende konkrete Beispiele für Artefakte genannt:

- Dateien in denen Source Code enthalten ist
- Skripte
- Datenbanktabellen

Im Kontext dieser Arbeit sind insbesondere die Dateien, in denen Source Code enthalten ist, allgemein als Artefakt bezeichnet. 2

AST Abstrakter Syntaxbaum. 29

Attributgrammatik . 22

Engine Eine Engine beschreibt eine Software oder einen Teil einer Software, der für eine spezifische Aufgabe verantwortlich ist (vgl. [PCM]). Die Aufgabe, die die in der Arbeit beschriebenen Source Engines erfüllen, wird in Abschnitt 2.1 beschrieben. 2, 3, 6–8, 110

Interface Ein Interface hat im Allgemeinen eine Übersetzungs- oder Vermittlungsfunktion zwischen gekoppelten Systemen (vgl. [Hal94]). Die Bedeutung des Begriffs in dieser Arbeit bezieht sich jedoch auf den Kontext der objektorientierten Programmierung. In diesem Zusammenhang beschreibt ein Interface die Methoden, die in den Klassen, die dieses

Interface erfüllen, vorhanden sein müssen. 1, 2, 6–10, 13, 60–62, 66, 69, 70, 106, 107, 113, 127, 129, 136

Komponente Eine Komponente beschreibt in der Softwarearchitektur im Allgemeinen ein Teil eines Softwaresystems. Die Definition dieses Begriffs wird in speziellen Frameworks weiter spezifiziert. Bezogen auf das in der Arbeit verwendete EJB-Framework, werden bspw. die Beans als Komponenten betrachtet (vgl. [DeM05]). 1–3

Substitutionsprinzip . 26, 27

Wrapper-Typ . 17

Anhang A

Kombination von Matchern

Wie aus der formalen Beschreibung zum *StructuralTypeMatcher* im Abschnitt 3.1.2 hervorgeht, ist dieser von den übrigen Matchern abhängig. Die Implementierung der dazugehörigen Klasse **StructuralTypeMatcher** verlangt zur Erzeugung eines Objektes dieser Klasse eine **TypeMatcher**. Dieser **TypeMatcher** muss laut der formalen Beschreibung die Implementierung der Matchingrelation $\Rightarrow_{internStruct}$ darstellen.

Zu diesem Zweck müssen die übrigen Matcher bzw. die dafür implementierten Klassen miteinander kombiniert werden, wie es in der Definition zur Matchingrelation $\Rightarrow_{internStruct}$ der Fall ist. Für solche Kombinationen steht die Klasse **MatcherCombiner** im Modul *SignatureMatching* bereit (siehe Listing A.1).

Diese Klasse erlaubt die Kombination von Objekten vom Typ **TypeMatcher**. Die Matcher-Klassen **ExactTypeMatcher**, **GenSpecTypeMatcher** und **WrappedTypeMatcher** implementieren alle dieses Interface.

Über die Methode **combine** in der **MatcherCombiner** wird bei der Kombination ein Supplier-Objekt erzeugt, welches über die **get**-Methode ein Objekt vom Typ **TypeMatcher** liefern kann. Dieses **TypeMatcher**-Objekt versucht beim Aufruf der Methode **matchesType(S,T)** die beiden Typen *S* und *T* über einen der kombinierten Matcher zu matchen (siehe Listing A.1). Dabei liefert die Methode **getSortedMatcher** eine sortierte Liste der kombinierten Matcher. Die

Sortierung wird aufsteigend entsprechend dem Basisrating (siehe auch Abschnitt 3.4.1) der kombinierten Matcher vorgenommen .

```
package de.fernuni.hagen.ma.gundermann.signaturematching.matching;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.function.Supplier;

import de.fernuni.hagen.ma.gundermann.signaturematching.SingleMatchingInfo;
import de.fernuni.hagen.ma.gundermann.signaturematching.matching.types.TypeMatcher;

public final class MatcherCombiner {
    private MatcherCombiner() {}

    public static Supplier<TypeMatcher> combine(TypeMatcher... matcher) {
        return () -> new TypeMatcher() {

            @Override
            public boolean matchesType(Class<?> checkType, Class<?> queryType) {
                for (TypeMatcher m : getSortedMatcher()) {
                    if (m.matchesType(checkType, queryType)) {
                        return true;
                    }
                }
                return false;
            }

            @Override
            public Collection<SingleMatchingInfo> calculateTypeMatchingInfos(Class<?>
                checkType, Class<?> queryType) {
                for (TypeMatcher m : getSortedMatcher()) {
                    if (m.matchesType(checkType, queryType)) {
                        return m.calculateTypeMatchingInfos(checkType, queryType);
                    }
                }
                return new ArrayList<>();
            }

            @Override
            public MatcherRate matchesWithRating(Class<?> checkType, Class<?> queryType) {
```

```

    for (TypeMatcher m : getSortedMatcher()) {
        MatcherRate rating = m.matchesWithRating(checkType, queryType);
        if (rating != null) {
            return rating;
        }
    }
    return null;
}

@Override
public double getTypeMatcherRate() {
    // irrelevant, weil matchesWithRating ueberschrieben wurde.
    return -0;
}

private Collection<TypeMatcher> getSortedMatcher() {
    List<TypeMatcher> matcherList = Arrays.asList(matcher);
    Collections.sort(matcherList, (l1, l2) ->
        Double.compare(l1.getTypeMatcherRate(), l2.getTypeMatcherRate()));
    return matcherList;
}

};
}
}

```

Listing A.1: Klasse: MatcherCombiner

Bei der Exploration wird letztendlich immer ein Objekt der Klasse **StructuralTypeMatcher** zur Ermittlung des Matchings verwendet. Listing A.2 zeigt die Instanziierung dieses Objektes unter Verwendung der Klasse **MatcherCombiner**.

```

TypeMatcher exactTM = new ExactTypeMatcher();
TypeMatcher genSpecTM = new GenSpecTypeMatcher();
TypeMatcher combinedGenSpecExactTM = MatcherCombiner.combine(genSpecTM,
    exactTM).get();
TypeMatcher wrappedTM = new WrappedTypeMatcher(() -> combinedGenSpecExactTM);
TypeMatcher combinedWrappedGenSpecExact = MatcherCombiner.combine(genSpecTM,
    exactTM, wrappedTM).get();
StructuralTypeMatcher structWrappedGenSpecExactTM = new StructuralTypeMatcher(
    () -> combinedWrappedGenSpecExact);

```

Listing A.2: Default-Instanziierung des StructuralTypeMatchers im DesiredComponentFinder

Anhang B

Verwendung aller Heuristiken

Die in den Abschnitten 3.4.1 - 3.4.3 vorgestellten Heuristiken können miteinander kombiniert werden. Listing B.1 zeigt die Implementierung der Funktionen, die für diese Kombination auf der Basis von Listing 3.10 angepasst oder ergänzt werden müssen.

```
1 function evalProxiesMitTarget( proxies, tests ){
2     testedProxies = []
3     for( proxy : proxies ){
4         passedTestcases = 0
5         blacklistChanged = false
6         evalProxy(proxy, tests)
7         if( passedTests == T.size ){
8             // passenden Proxy gefunden
9             return proxy
10        }
11        else{
12            testedProxies.add(proxy)
13            if( passedTests > 0 || blacklistChanged ){
14                // noch nicht evaluierte Proxies ermitteln
15                optimizedProxies = proxies.removeAll( testedProxies )
16                // Heuristik PTF
17                if( passedTests > 0 ){
18                    priorityTargets.addAll( proxy.targets )
19                    optimizedProxies = PTF( optimizedProxies )
20                }
21                // Heuristik BL_FFMD und BL_FSM
22                if( blacklistChanged ){
23                    optimizedProxies = BL( optimizedProxies )
24                }
25                return evalProxiesMitTarget( optimizedProxies, tests )

```

```
26     }
27 }
28 }
29 // kein passenden Proxy gefunden
30 return null
31 }
32
33 function evalProxy(proxy, tests){
34   for( test : tests ){
35     //alle Tests werden durchgefuehrt
36     try{
37       if( test.eval( proxy ) ){
38         passedTestcases = passedTestcases + 1
39       }elseif( test.isSingleMethodTest ){
40         methodName = test.testedSingleMethodName
41         mDel = getMethodDelegation( proxy, methodName )
42         methodDelegationBlacklist.add( mDel )
43         blacklistChanged = true
44         return
45       }
46     }
47     catch (SigMaGlueException e){
48       mDel = e.failedMethodDelegation
49       methodDelegationBlacklist.add( mDel )
50       blacklistChanged = true
51       return
52     }
53   }
54 }
55
56 function relevantProxies( proxies, anzahl ){
57   relProxies = proxiesMitTargets( proxies, anzahl );
58   optimizedLMF = LMF( relProxies )
59   optimizedPTTF = PTTF( optimizedLMF )
60   return BL( optimizedPTTF )
61 }
```

Listing B.1: Kombination aller Heuristiken

Anhang C

Deklaration der relevanten Typen

Im Folgenden erfolgt die Deklaration der *required Typen*, mit denen die Evaluation der Heuristiken in Kapitel 5 durchgeführt wird, sowie die Deklaration der *provided Typen*, die als Ergebnis der jeweiligen Exploration für einen *required Typ* einzeln oder in Kombination erwartet, oder innerhalb einer der Deklarationen eines *required Typ* verwendet werden. Dabei ist davon auszugehen, dass diese Typen aus dem JDK als Bibliothek aufbauen.

Die Listings C.1 - C.7 zeigen die Deklarationen für die *required Typen*.

```
required ElerFTFoerderprogrammeProvider{
    Collection getAlleFreigegebenenFPs()
    ElerFTFoerderprogramm getElerFTFoerderprogramm(DvAntragsJahr ,
        DvFoerderprogramm , Date)
}
```

Listing C.1: Deklaration von ElerFTFoerderprogrammeProvider

```
required FoerderprogrammeProvider{
    Collection getAlleFreigegebenenFPs()
    Foerderprogramm getFoerderprogramm(DvAntragsJahr , DvFoerderprogramm , Date)
}
```

Listing C.2: Deklaration von FoerderprogrammeProvider

```
required MinimalFoerderprogrammeProvider{
    Collection getAlleFreigegebenenFPs()
    Foerderprogramm getFoerderprogramm(String , int , Date)
```

```
}
```

Listing C.3: Deklaration von MinimalFoerderprogrammeProvider

```
required IntubatingFireFighter{
    void intubate(Injured)
    FireState extinguishFire(Fire)
}
```

Listing C.4: Deklaration von IntubatingFireFighter

```
required IntubatingFreeing{
    void intubate(Injured)
    void free(Injured)
}
```

Listing C.5: Deklaration von IntubatingFreeing

```
required IntubatingFreeing{
    void intubate(IntubationPatient)
    FireState extinguishFire(Fire)
}
```

Listing C.6: Deklaration von IntubatingPatientFireFighter

```
required KOFGPCProvider{
    Collection getKOFGsVonFP(DvFoerderprogramm)
    Collection getPCsZuKOFG(DvFoerdergegenstand, DvAntragsJahr)
}
```

Listing C.7: Deklaration von KOFGPCProvider

Die Listings C.8 - C.14 zeigen die *provided Typen*, die in den Deklarationen der *required Typen* verwendet wurden und nicht Teil des JDKs sind.

```
provided ElerFTFoerderprogramm extends Foerderprogramm{
    DvFlaeche mindestParzellenGroesse
    DvFlaeche maximaleParzellenGroesse
    int differenzKassenjahrAntragsjahr
    boolean isMehrjaehrig

    DvFlaeche getMaximaleParzellengroesse()
    DvFlaeche getMindestParzellenGroesse()
    int getDifferenzKassenjahrAntragsjahr()
    boolean isMehrjaehrig()
}
```

Listing C.8: Deklaration von ElerFTFoerderprogramm

```

provided Foerderprogramm extends Object{
    Long id
    STDGueltigkeit gueltigkeit
    Long fpId
    BigDecimal bagatellbetrag
    BigDecimal bagatellmenge
    List vorgaengeAm15
    Set landesmassnahmen

    Long getId()
    boolean isTechnischGueltig(Date)
    DvFoerderprogramm getFoerderprogramm()
    BigDecimal getBagatellmengeFoerd()
    BigDecimal getBagatellbetragFoerd()
    boolean isFachlichGueltig(DvAntragsJahr)
    STDGueltigkeit getGueltigkeit()
    Long getFpId()
}

```

Listing C.9: Deklaration von Foerderprogramm

```

provided DvAntragsJahr extends AbstractDomainValue{
    int antragsJahr

    DvAntragsJahr add(int)
    int compareTo(Object)
    int intValue()
    Object readResolve()
    DvAntragsJahr getVorjahr()
    int differenz(DvAntragsJahr)
    DvAntragsJahr sub(int)
    String toStringImpl()
}

```

Listing C.10: Deklaration von DvAntragsJahr

```

provided DvFoerderprogramm extends DvEnumerable{
    long id
    String code
    String fpGruppe
    String bezeichnung
    String bezeichnungLang
    String getName()

    Long getId()
    Long getNummer()
}

```

```

    void validateCode(String)
    String getFpGruppe()
    String getBezeichnung()
    String toStringImpl()
    String getCode()
    String getFPNummerExtern()
    String getBezeichnungLang()
}

```

Listing C.11: Deklaration von DvFoerderprogramm

```

provided Injured extends Object{
    Collection suffers

    Collection getSuffers()
    void healSuffer(Suffer)
    boolean isStabilized()
}

```

Listing C.12: Deklaration von Injured

```

provided Fire extends Object{
    boolean active

    void extinguish()
    boolean isActive()
}

```

Listing C.13: Deklaration von Fire

```

provided IntubationPatient extends Object{
    boolean isIntubated

    boolean isIntubated()
    void setIntubated(boolean)
}

```

Listing C.14: Deklaration von IntubationPatient

Die Listings C.15 - C.18 zeigen die Deklarationen der *provided Typen*, aus denen bei der Exploration ein passender Proxy erzeugt werden soll.

```

provided ElerFTStammdatenAuskunftService extends Object{
    Collection getAlleElerFTKombiKzFpFoerdergegenstaende()
    Collection getAlleElerFTKoFoerdergegenstaende()
    Collection
        getFeststellungscodeVerpflichtungList(FeststellungscodeVerpflichtungImplQuery)
}

```

```

FeststellungscodeVerpflichtungImpl
    getFeststellungscodeVerpflichtungImpl(FeststellungscodeVerpflichtungImplQuery)
Collection getAlleElerFTTierFoerdergegenstaende(DvFoerderprogramm,
    DvAntragsJahr, AntragsVorgangsTyp)
Collection getAlleFreigegebenenFoerderprogramme(AntragsVorgangsTyp)
Collection getAlleFreigegebenenFoerderprogramme()
ElerFTKzFpFoerdergegenstand2Foerderfaehigkeit
    getElerFTKzFpFoerdergegenstand2Foerderfaehigkeit(DvFoerdergegenstand,
    DvAntragsJahr)
FeststellungsCodeVerpflichtung2FP
    getFeststellungsCodeVerpflichtung2FP(FeststellungsCodeVerpflichtung2FPQuery)
DvEftOekoFoerdergegenstandGruppe
    getOekoFgGruppe2Foerdergegenstand(DvFoerdergegenstand)
Collection getAlleElerFTKzFpFoerdergegenstaende()
VerpflichtungsGegenstandImpl
    getVerpflichtungsGegenstandImpl(VerpflichtungsGegenstandImplQuery)
ElerFTVorhaben getVorhaben2Foerdergegenstand(DvFoerdergegenstand,
    DvAntragsJahr)
Verpflichtungszeitraum getVerpflichtungszeitraum(DvFoerderprogramm,
    DvAntragsJahr)
int getMaxStandardAnzahlZahlungen(DvFoerderprogramm, DvAntragsJahr)
DvZusatzInfoTyp getZusatzInfo2Foerdergegenstand(DvFoerdergegenstand,
    DvAntragsJahr)
int getStandardAnzahlZahlungen(DvUntermassnahme, DvAntragsJahr)
int getStandardAnzahlZahlungen(Landesmassnahme, DvAntragsJahr)
Collection getElerFTKoFoerdergegenstaende(DvFoerderprogramm,
    DvUntermassnahme, DvAntragsJahr)
Collection getElerFTKoFoerdergegenstaende(DvFoerderprogramm)
Collection getAlleFg2ZusatzInfo(DvZusatzInfoTyp, DvAntragsJahr)
int getDifferenzJahrVerpflbeginnEAJ(DvFoerderprogramm, DvAntragsJahr)
Collection getVerpflichtungsGegenstandList(VerpflichtungsGegenstandImplQuery)
Collection getAenderungscodPropertiesList(AenderungscodPropertiesQuery)
Collection getAlleFg2OekoFgGruppe(DvEftOekoFoerdergegenstandGruppe)
ElerFTFoerderprogramm getFoerderprogramm(ElerFTFoerderprogrammQuery)
ElerFTFoerderprogramm getFoerderprogramm(DvAntragsJahr, DvFoerderprogramm,
    Date)
Collection getElerFTAenderung2ElerFTFP(DvFoerderprogramm)
Collection getElerFTAenderung2ElerFTFP(ElerFTAenderung)
ElerFTAenderung2ElerFTFP getElerFTAenderung2ElerFTFP(ElerFTAenderung,
    DvFoerderprogramm)
Collection getFoerdergegenstaende(AbstractElerFTFoerdergegenstandQuery)
Collection getElerFTTierFoerdergegenstaende(DvFoerderprogramm,
    DvUntermassnahme, DvAntragsJahr)
Collection getFoerderprogramme(ElerFTFoerderprogrammQuery)
Collection getFoerderprogramme(Date)

```

```

    Collection getAlleFoerderprogramme()
    Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm,
        DvUntermassnahme, DvAntragsJahr)
    Collection getElerFTKzFpFoerdergegenstaende(ElerFTKombiKzFpFoerdergegenstand)
    Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm,
        Finanzierungsschluessel, DvAntragsJahr)
    Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm, DvAntragsJahr)
    Collection getAlleFg2Vorhaben(ElerFTVorhaben, DvAntragsJahr)
    Map getKzFpJeFg(Collection, DvAntragsJahr)
}

```

Listing C.15: Deklaration von ElerFTStammdatenAuskunftService

```

provided StammdatenAuskunftService extends Object{
    Collection
        getLandesmassnahmen2Foerdergegenstaende(Landesmassnahme2FoerdergegenstandQuery)
    Collection getFoerdergegenstaendeZuFinanzierungsschluessel(DvFoerderprogramm,
        Finanzierungsschluessel, DvAntragsJahr)
    Landesmassnahme getLandesmassnahme(Long)
    Map getOberFgJeUnterFg(DvAntragsJahr)
    Collection getFoerderprogramme(Date)
    Foerdergegenstand getFoerdergegenstand(FoerdergegenstandQuery)
    Collection getFoerdergegenstaende(DvFoerderprogramm)
    Collection getFoerdergegenstaende(FoerdergegenstandQuery)
    Collection getFoerdergegenstaende(Landesmassnahme)
    Collection getFinanzierungsschluessel(FinanzierungsschluesselQuery)
    Collection getFinanzierungskonfigurationen(FinanzierungskonfigurationQuery)
    Collection getFinanzierungskonfigurationen(Collection, DvAntragsJahr)
    Collection getFinanzierungskonfigurationen(DvAntragsJahr, DvFoerderprogramm, Long)
    Finanzierungskonfiguration getFinanzierungskonfigurationen(DvAntragsJahr,
        DvFoerderprogramm, DvFoerdergegenstand)
    Map getProduktcodesJeFg(DvFoerderprogramm, DvAntragsJahr, Collection,
        ProduktcodeArt, Finanzierungsschluessel)
    Foerderprogramm getFoerderprogramm(Foerdergegenstand)
    Foerderprogramm getFoerderprogramm(DvAntragsJahr, DvFoerderprogramm, Date)
    Collection getAblehnungsgrundCodes(Foerderprogramm, DvAntragsJahr,
        KuerzungsgrundCode)
    Collection getUnterFoerdergegenstaende(DvAntragsJahr, Collection)
    Collection getFoerdergegenstandGruppenZuFgs(DvAntragsJahr, Collection)
    Collection getLandesmassnahmen(DvAntragsJahr, DvFoerderprogramm)
    Collection getLandesmassnahmen(DvAntragsJahr, Foerdergegenstand)
    Collection getLandesmassnahmen(LandesmassnahmeQuery)
    Produktcode getProduktcode(ProduktcodeQuery)
    Produktcode getProduktcode(DvAntragsJahr, DvFoerdergegenstand, ProduktcodeArt)
    Produktcode getProduktcode(DvAntragsJahr, DvFoerdergegenstand, ProduktcodeArt,
        Finanzierungsschluessel)
}

```

```

BigDecimal getBeihilfesatz(DvAntragsJahr, DvFoerdergegenstand, Integer)
Collection getProduktcodes(DvAntragsJahr, Finanzierungsschlüssel)
Collection getProduktcodes(DvAntragsJahr, DvFoerdergegenstand,
    Finanzierungsschlüssel)
Collection getProduktcodes(ProduktcodeQuery)
Collection getProduktcodes(DvAntragsJahr, DvFoerderprogramm)
Collection getProduktcodes(DvAntragsJahr, DvFoerdergegenstand)
Collection getProduktcodes(Collection)
BigDecimal getKappungBetrag(DvFoerdergegenstand, DvAntragsJahr)
Collection getVorgaenge(Date, DvFoerderprogramm)
Collection getVorgaenge(AntragsVorgangsTyp)
Collection getVorgaenge(Date, AntragsVorgangsTyp)
Collection getVorgaenge()
Collection getVorgaenge(DvFoerderprogramm, Date, AntragsVorgangsTyp)
BigDecimal getKappungMenge(DvFoerdergegenstand, DvAntragsJahr)
Vorgang getVorgang(DvAntragsJahr, DvFoerderprogramm, Date, AntragsVorgangsTyp,
    DvAntragsJahr)
Vorgang getVorgang(DvFoerderprogramm, Date, AntragsVorgangsTyp, DvAntragsJahr)
}

```

Listing C.16: Deklaration von StammdatenAuskunftService

```

provided Doctor extends Object{
    void provideHeartbeatMessage(Injured)
    void stabililizeBrokenBones(Injured)
    void healWithMed(Injured, Medicine)
    void placeInfusion(Injured)
    void nurseWounds(Injured)
    void intubate(Injured)
}

```

Listing C.17: Deklaration von Doctor

```

provided FireFighter extends Object{
    void stabilizeBrokenBones(Injured)
    void provideHeartbeatMessage(Injured)
    FireState extinguishFire(Fire)
    void free(Injured)
    void nurseWounds(Injured)
}

```

Listing C.18: Deklaration von FireFighter

Anhang D

Interfaces und Test-Implementierungen

Im Folgenden werden zum einen die Interfaces, die sich aus den Deklarationen der *required Typen* aus dem Anhang C ableiten lassen, aufgeführt. Zum anderen werden die Implementierungen der Testklassen, auf die die oben genannten Interfaces über die Annotation `RequiredTypeTestReference` verweisen, dargelegt.

Die Listings D.1 - D.7 zeigen dabei die Deklarationen der Java-Interfaces¹ für die *required Typen* aus Tabelle 5.1 aus Kapitel 5.

```
@RequiredTypeTestReference( testClasses = ElerFTFoerderprogrammProviderTest.class )
public interface ElerFTFoerderprogrammeProvider {

    Collection<ElerFTFoerderprogramm> getAlleFreigegebenenFPs();

    ElerFTFoerderprogramm getElerFTFoerderprogramm( DvAntragsJahr jahr,
        DvFoerderprogramm fp, Date date );

}
```

Listing D.1: Interface ElerFTFoerderprogrammeProvider

```
@RequiredTypeTestReference( testClasses = FoerderprogrammProviderTest.class )
public interface FoerderprogrammeProvider {

    Collection<Foerderprogramm> getAlleFreigegebenenFPs();

}
```

¹Auf die Import-Anweisungen wurde verzichtet.

```

Foerderprogramm getFoerderprogramm( DvFoerderprogramm fp, DvAntragsJahr jahr,
    Date date );

}

```

Listing D.2: Interface FoerderprogrammeProvider

```

@RequiredTypeTestReference( testClasses = MinimalFoerderprogrammProviderTest.class )
public interface MinimalFoerderprogrammeProvider {

    Collection<String> getAlleFreigegebenenFPs();

    Foerderprogramm getFoerderprogramm( String fp, int jahr, Date date );

}

```

Listing D.3: Interface MinimalFoerderprogrammeProvider

```

@RequiredTypeTestReference( testClasses = IntubatingFireFighterTest.class )
public interface IntubatingFireFighter {

    public void intubate( Injured injured );

    public FireState extinguishFire( Fire fire );

}

```

Listing D.4: Interface IntubatingFireFighter

```

@RequiredTypeTestReference( testClasses = IntubatingFreeingTest.class )
public interface IntubatingFreeing {

    public void intubate( Injured injured );

    public void free( Injured injured );

}

```

Listing D.5: Interface IntubatingFreeing

```

@RequiredTypeTestReference( testClasses = IntubatingPatientFireFighterTest.class )
public interface IntubatingPatientFireFighter {

    public void intubate( IntubationPartient patient );

    public FireState extinguishFire( Fire fire );

}

```

```
}
```

Listing D.6: Interface IntubatingPatientFireFighter

```
@RequiredTypeTestReference( testClasses = KOFGPCProviderTest.class )
public interface KOFGPCProvider {

    Collection<ElerFTKoFoerdergegenstand> getKOFGsVonFP( DvFoerderprogramm fp );

    Collection<Produktcode> getPCsZuKOFG( DvFoerdergegenstand fg, DvAntragsJahr aj );

}
```

Listing D.7: Interface KOFGPCProvider

Zu erkennen ist, dass jedes Interfaces, wie in Abschnitt 4.2 beschrieben, mit der Annotation `RequiredTypeTestReference` versehen ist, über die auf eine Java-Klasse verwiesen wird, in der die Tests zu dem jeweiligen *required Typ* implementiert sind.

Die Listings D.8 - D.14 zeigen die Implementierungen dieser Testklassen².

```
public class ElerFTFoerderprogrammProviderTest implements TriedMethodCallsInfo {

    private ElerFTFoerderprogrammeProvider provider;
    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider( ElerFTFoerderprogrammeProvider provider ) {
        this.provider = provider;
    }

    @RequiredTypeTest
    public void testEmptyCollection() {
        addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
            ElerFTFoerderprogrammeProvider.class ) );
        Collection<ElerFTFoerderprogramm> alleFreigegebenenFPs =
            provider.getAlleFreigegebenenFPs();
        assertThat( alleFreigegebenenFPs, notNullValue() );
    }

    @RequiredTypeTest
    public void testMockedFPCollection() {
```

²Auf die Import-Anweisungen wurde verzichtet.

```

DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
    DvFoerderprogramm.FP215 );
addTriedMethodCall( getMethod( "getElerFTFoerderprogramm",
    ElerFTFoerderprogrammeProvider.class ) );
ElerFTFoerderprogramm alleFreigegebenenFPs = provider.getElerFTFoerderprogramm(
    DvAntragsJahr.AJ2020,
    fp, new Date() );
assertThat( alleFreigegebenenFPs, nullValue() );
}

@Override
public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
}

@Override
public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
}
}

```

Listing D.8: Interface ElerFTFoerderprogrammProviderTest

```

public class FoerderprogrammProviderTest implements TriedMethodCallsInfo {

    private FoerderprogrammeProvider provider;

    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider( FoerderprogrammeProvider provider ) {
        this.provider = provider;
    }

    @RequiredTypeTest
    public void testEmptyCollection() {
        addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
            FoerderprogrammeProvider.class ) );
        Collection<Foerderprogramm> alleFreigegebenenFPs =
            provider.getAlleFreigegebenenFPs();
        assertThat( alleFreigegebenenFPs, notNullValue() );
    }

    @RequiredTypeTest
    public void testMockedFPCollection() {

```

```

DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
    DvFoerderprogramm.FP508 );
addTriedMethodCall( getMethod( "getFoerderprogramm",
    FoerderprogrammeProvider.class ) );
Foerderprogramm relevantFP = provider.getFoerderprogramm( fp,
    DvAntragsJahr.AJ2020,
    new Date() );
assertThat( relevantFP, notNullValue() );
}

@RequiredTypeTest
public void testDZFPCollection() {
    DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
        DvFoerderprogramm.FP215 );
    addTriedMethodCall( getMethod( "getFoerderprogramm",
        FoerderprogrammeProvider.class ) );
    Foerderprogramm relevantFP = provider.getFoerderprogramm( fp,
        DvAntragsJahr.AJ2020,
        new Date() );
    assertThat( relevantFP, notNullValue() );
}

@Override
public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
}

@Override
public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
}
}

```

Listing D.9: Interface FoerderprogrammProviderTest

```

public class MinimalFoerderprogrammProviderTest implements TriedMethodCallsInfo {

    private MinimalFoerderprogrammeProvider provider;
    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider( MinimalFoerderprogrammeProvider provider ) {
        this.provider = provider;
    }
}

```

```

@RequiredTypeTest
public void testEmptyCollection() {
    addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
        MinimalFoerderprogrammeProvider.class ) );
    Collection<String> alleFreigegebenenFPs = provider.getAlleFreigegebenenFPs();
    assertThat( alleFreigegebenenFPs, notNullValue() );
}

@RequiredTypeTest
public void testGetFoerderprogramm() {
    String fpCode = "215";
    addTriedMethodCall( getMethod( "getFoerderprogramm",
        MinimalFoerderprogrammeProvider.class ) );
    Foerderprogramm fp = provider.getFoerderprogramm( fpCode, 2015, new Date() );
    assertThat( fp, notNullValue() );
    DvFoerderprogramm dvFP = fp.getFoerderprogramm();
    assertThat( dvFP, notNullValue() );

    String code = dvFP.getCode();
    assertThat( fpCode, equalTo( code ) );
}

@Override
public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
}

@Override
public Collection<Method> getTriedMethodCalls() {
    return calledMethods ;
}
}

```

Listing D.10: Interface MinimalFoerderprogrammProviderTest

```

public class IntubatingFireFighterTest implements TriedMethodCallsInfo {

    private IntubatingFireFighter intubatingFireFighter;
    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider(IntubatingFireFighter intubatingFireFighter) {
        this.intubatingFireFighter = intubatingFireFighter;
    }
}

```

```

@RequiredTypeTest
public void free() {
    Fire fire = new Fire();
    addTriedMethodCall(getMethod("extinguishFire",
        IntubatingFireFighter.class));
    FireState fireState = intubatingFireFighter.extinguishFire(fire);
    assertTrue(Objects.equals(fireState.isActive(), fire.isActive()));
    assertFalse(fire.isActive());
}

@RequiredTypeTest
public void intubate() {
    Collection<Suffer> suffer = Arrays.asList(Suffer.BREATH_PROBLEMS);
    Injured patient = new Injured(suffer);
    addTriedMethodCall(getMethod("intubate",
        IntubatingFireFighter.class));
    intubatingFireFighter.intubate(patient);
    assertTrue(patient.isStabilized());
}

@Override
public void addTriedMethodCall(Method m) {
    calledMethods.add(m);
}

@Override
public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
}
}

```

Listing D.11: Interface IntubatingFireFighterTest

```

public class IntubatingFreeingTest implements TriedMethodCallsInfo {

    private IntubatingFreeing intubatingFreeing;
    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider(IntubatingFreeing intubatingFireFighter) {
        this.intubatingFreeing = intubatingFireFighter;
    }

    @RequiredTypeTest

```

```

public void free() {
    Collection<Suffer> suffer = Arrays.asList(Suffer.LOCKED);
    Injured patient = new Injured(suffer);
    addTriedMethodCall(getMethod("free", IntubatingFreeing.class));
    intubatingFreeing.free(patient);
    assertTrue(patient.isStabilized());
}

@RequiredTypeTest
public void intubate() {
    Collection<Suffer> suffer = Arrays.asList(Suffer.BREATH_PROBLEMS);
    Injured patient = new Injured(suffer);
    addTriedMethodCall(getMethod("intubate", IntubatingFreeing.class));
    intubatingFreeing.intubate(patient);
    assertTrue(patient.isStabilized());
}

@Override
public void addTriedMethodCall(Method m) {
    calledMethods.add(m);
}

@Override
public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
}
}

```

Listing D.12: Interface IntubatingFreeingTest

```

public class IntubatingPatientFireFighterTest implements TriedMethodCallsInfo {

    private IntubatingPatientFireFighter intubatingPatientFireFighter;
    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider(IntubatingPatientFireFighter intubatingFireFighter) {
        this.intubatingPatientFireFighter = intubatingFireFighter;
    }

    @RequiredTypeTest
    public void extinguishFire() {
        Fire fire = new Fire();
        addTriedMethodCall(getMethod("extinguishFire",
            IntubatingPatientFireFighter.class));
    }
}

```



```

        FireState fireState =
            intubatingPatientFireFighter.extinguishFire(fire);
        assertTrue(Objects.equals(fireState.isActive(), fire.isActive()));
        assertFalse(fire.isActive());
    }

    @RequiredTypeTest
    public void intubate() {
        IntubationPartient patient = new IntubationPartient();
        addTriedMethodCall(getMethod("intubate",
            IntubatingPatientFireFighter.class));
        intubatingPatientFireFighter.intubate(patient);
        assertTrue(patient.isIntubated());
    }

    @Override
    public void addTriedMethodCall(Method m) {
        calledMethods.add(m);
    }

    @Override
    public Collection<Method> getTriedMethodCalls() {
        return calledMethods;
    }
}

```

Listing D.13: Interface IntubatingPatientFireFighterTest

```

public class KOFGPCProviderTest implements TriedMethodCallsInfo {

    private KOFGPCProvider provider;

    private Collection<Method> calledMethods = new ArrayList<Method>();

    @RequiredTypeInstanceSetter
    public void setProvider( KOFGPCProvider provider ) {
        this.provider = provider;
    }

    @RequiredTypeTest
    public void testKOFGsCollection() {
        DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
            DvFoerderprogramm.FP508 );
        addTriedMethodCall( getMethod( "getKOFGsVonFP", KOFGPCProvider.class ) );
        Collection<ElerFTKoFoerdergegenstand> kofGsVonFP = provider.getKOFGsVonFP( fp );
    }
}

```

```

        assertThat( kofGsVonFP, notNullValue() );
        assertThat( kofGsVonFP.isEmpty(), equalTo( false ) );
        assertThat( kofGsVonFP.stream().anyMatch( fg -> fg.getCode().equals( "K0508" )
            ), equalTo( true ) );
    }

    @RequiredTypeTest
    public void testPCsCollection() {
        DvFoerdergegenstand fg = DvFoerdergegenstand.Factory.valueOf( 20155080025L );
        addTriedMethodCall( getMethod( "getPCsZuK0FG", K0FGPCProvider.class ) );
        Collection<Produktcode> pcs = provider.getPCsZuK0FG( fg, DvAntragsJahr.AJ2020 );
        assertThat( pcs, notNullValue() );
        assertThat( pcs.isEmpty(), equalTo( false ) );
    }

    @Override
    public void addTriedMethodCall( Method m ) {
        this.calledMethods.add( m );
    }

    @Override
    public Collection<Method> getTriedMethodCalls() {
        return calledMethods;
    }
}

```

Listing D.14: Interface K0FGPCProviderTest

Hier ist zu erkennen, dass die Testklassen alle das Interfaces `TriedMethodCallsInfo` implementieren, über das die für die Heuristik *BL_NMC* benötigten Informationen (siehe Abschnitt 3.4.3) ermittelt werden. Ebenso ist die Implementierung dieses Interfaces in den oben genannten Listings zu erkennen.

Anhang E

Ergebnisse für die Heuristik LMF (Ergänzungen)

In diesem Abschnitt werden die Untersuchungsergebnisse der Heuristik *LMF* mit allen Varianten zur Bestimmung des Matcherratings aus Abschnitt 3.4.1 dargelegt. Dieses Kapitel bildet somit eine Ergänzung zu Abschnitt 5.3. Die darin beschriebenen Ergebnisse der Variante *1.1* werden der Vollständigkeit halber in dem vorliegenden Kapitel nochmals aufgeführt.

Die folgenden Ergebnisse beziehen sich auf die in Kapitel 5 vorgestellten *required Typen TEI1-TEI7*.

Ergebnisse für Variante 1.1

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	1889	$p(55) - 1890$
richtig	1	0

1	positiv	negativ
falsch	1463	$p(50) - 1464$
richtig	1	0

Tabelle E.1: Ergebnisse *LMF* mit Variante 1.1 für TEI1 1. Durchlauf
 Tabelle E.2: Ergebnisse *LMF* mit Variante 1.1 für TEI2 1. Durchlauf
 Tabelle E.3: Ergebnisse *LMF* mit Variante 1.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	2	$p(2247) - 3$
richtig	1	0

Tabelle E.4: Ergebnisse *LMF* mit Variante 1.1 für TEI4 1. Durchlauf
 Tabelle E.5: Ergebnisse *LMF* mit Variante 1.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	32	$p(2775) - 33$
richtig	1	0

Tabelle E.6: Ergebnisse *LMF* mit Variante 1.1 für TEI5 1. Durchlauf
 Tabelle E.7: Ergebnisse *LMF* mit Variante 1.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.8: Ergebnisse *LMF* mit Variante 1.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.9: Ergebnisse *LMF* mit Variante 1.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.10: Ergebnisse *LMF* mit Variante 1.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	7641	$p(52150) - 7642$
richtig	1	0

Tabelle E.11: Ergebnisse *LMF* mit Variante 1.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.2

1	positiv	negativ
falsch	1	$p(44) - 2$
richtig	1	0

Tabelle E.12: Ergebnisse *LMF* mit Variante 1.2 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	2783	$p(55) - 2784$
richtig	1	0

Tabelle E.13: Ergebnisse *LMF* mit Variante 1.2 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1830	$p(50) - 1831$
richtig	1	0

Tabelle E.14: Ergebnisse *LMF* mit Variante 1.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.15: Ergebnisse *LMF* mit Variante 1.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	3	$p(2247) - 4$
richtig	1	0

Tabelle E.16: Ergebnisse *LMF* mit Variante 1.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.17: Ergebnisse *LMF* mit Variante 1.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	3	$p(2775) - 4$
richtig	1	0

Tabelle E.18: Ergebnisse *LMF* mit Variante 1.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.19: Ergebnisse *LMF* mit Variante 1.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.20: Ergebnisse *LMF* mit Variante 1.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	161298	$p(52150) - 161299$
richtig	1	0

Tabelle E.21: Ergebnisse *LMF* mit Variante 1.2 für TEI7 1. Durchlauf

Tabelle E.22: Ergebnisse *LMF* mit Variante 1.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.3

1	positiv	negativ
falsch	50	$p(44) - 51$
richtig	1	0

1	positiv	negativ
falsch	20	$p(55) - 21$
richtig	1	0

1	positiv	negativ
falsch	121	$p(50) - 122$
richtig	1	0

Tabelle E.23: Ergebnisse *LMF* mit Variante 1.3 für TEI1 1. Durchlauf

Tabelle E.24: Ergebnisse *LMF* mit Variante 1.3 für TEI2 1. Durchlauf

Tabelle E.25: Ergebnisse *LMF* mit Variante 1.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	57	$p(2247) - 58$
richtig	1	0

Tabelle E.26: Ergebnisse *LMF* mit Variante 1.3 für TEI4 1. Durchlauf

Tabelle E.27: Ergebnisse *LMF* mit Variante 1.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	6246	$p(2775) - 6247$
richtig	1	0

Tabelle E.28: Ergebnisse *LMF* mit Variante 1.3 für TEI5 1. DurchlaufTabelle E.29: Ergebnisse *LMF* mit Variante 1.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

2	positiv	negativ
falsch	5	$p(1323) - 6$
richtig	1	0

Tabelle E.30: Ergebnisse *LMF* mit Variante 1.3 für TEI6 1. DurchlaufTabelle E.31: Ergebnisse *LMF* mit Variante 1.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	121074	$p(52150) - 121075$
richtig	1	0

Tabelle E.32: Ergebnisse *LMF* mit Variante 1.3 für TEI7 1. DurchlaufTabelle E.33: Ergebnisse *LMF* mit Variante 1.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.4

1	positiv	negativ
falsch	45	$p(44) - 46$
richtig	1	0

1	positiv	negativ
falsch	2025	$p(55) - 2026$
richtig	1	0

1	positiv	negativ
falsch	1517	$p(50) - 1518$
richtig	1	0

Tabelle E.34: Ergebnisse *LMF* mit Variante 1.4 für TEI1 1. Durchlauf

Tabelle E.35: Ergebnisse *LMF* mit Variante 1.4 für TEI2 1. Durchlauf

Tabelle E.36: Ergebnisse *LMF* mit Variante 1.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	4	$p(2247) - 5$
richtig	1	0

Tabelle E.37: Ergebnisse *LMF* mit Variante 1.4 für TEI4 1. Durchlauf

Tabelle E.38: Ergebnisse *LMF* mit Variante 1.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	34	$p(2775) - 35$
richtig	1	0

Tabelle E.39: Ergebnisse *LMF* mit Variante 1.4 für TEI5 1. Durchlauf

Tabelle E.40: Ergebnisse *LMF* mit Variante 1.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.41: Ergebnisse *LMF* mit Variante 1.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.42: Ergebnisse *LMF* mit Variante 1.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	21068	$p(52150) - 21069$
richtig	1	0

Tabelle E.43: Ergebnisse *LMF* mit Variante 1.4 für TEI7 1. Durchlauf

Tabelle E.44: Ergebnisse *LMF* mit Variante 1.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.1

1	positiv	negativ
falsch	8	$p(44) - 9$
richtig	1	0

1	positiv	negativ
falsch	3975	$p(55) - 3976$
richtig	1	0

1	positiv	negativ
falsch	2933	$p(50) - 2934$
richtig	1	0

Tabelle E.45: Ergebnisse *LMF* mit Variante 2.1 für TEI1 1. Durchlauf

Tabelle E.46: Ergebnisse *LMF* mit Variante 2.1 für TEI2 1. Durchlauf

Tabelle E.47: Ergebnisse *LMF* mit Variante 2.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.48: Ergebnisse *LMF* mit Variante 2.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p(2247) - 3$
richtig	1	0

Tabelle E.49: Ergebnisse *LMF* mit Variante 2.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.50: Ergebnisse *LMF* mit Variante 2.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	32	$p(2775) - 33$
richtig	1	0

Tabelle E.51: Ergebnisse *LMF* mit Variante 2.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.52: Ergebnisse *LMF* mit Variante 2.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.53: Ergebnisse *LMF* mit Variante 2.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.54: Ergebnisse *LMF* mit Variante 2.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	32018037	$p(52150) - 32018038$
richtig	1	0

Tabelle E.55: Ergebnisse *LMF* mit Variante 2.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.2

1	positiv	negativ
falsch	0	$p(44) - 1$
richtig	1	0

1	positiv	negativ
falsch	8007	$p(55) - 8008$
richtig	1	0

1	positiv	negativ
falsch	7104	$p(50) - 7105$
richtig	1	0

Tabelle E.56: Ergebnisse *LMF* mit Variante 2.2 für TEI1 1. Durchlauf

Tabelle E.57: Ergebnisse *LMF* mit Variante 2.2 für TEI2 1. Durchlauf

Tabelle E.58: Ergebnisse *LMF* mit Variante 2.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	0	$p(2247) - 1$
richtig	1	0

Tabelle E.59: Ergebnisse *LMF* mit Variante 2.2 für TEI4 1. Durchlauf

Tabelle E.60: Ergebnisse *LMF* mit Variante 2.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.61: Ergebnisse *LMF* mit Variante 2.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	0	$p(2775) - 1$
richtig	1	0

Tabelle E.62: Ergebnisse *LMF* mit Variante 2.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.63: Ergebnisse *LMF* mit Variante 2.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.64: Ergebnisse *LMF* mit Variante 2.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.65: Ergebnisse *LMF* mit Variante 2.2 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	2840500	$p(52150) - 2840501$
richtig	1	0

Tabelle E.66: Ergebnisse *LMF* mit Variante 2.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.3

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	2642	$p(55) - 2643$
richtig	1	0

1	positiv	negativ
falsch	1686	$p(50) - 1687$
richtig	1	0

Tabelle E.67: Ergebnisse *LMF* mit Variante 2.3 für TEI1 1. Durchlauf

Tabelle E.68: Ergebnisse *LMF* mit Variante 2.3 für TEI2 1. Durchlauf

Tabelle E.69: Ergebnisse *LMF* mit Variante 2.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	67	$p(2247) - 68$
richtig	1	0

Tabelle E.70: Ergebnisse *LMF* mit Variante 2.3 für TEI4 1. Durchlauf

Tabelle E.71: Ergebnisse *LMF* mit Variante 2.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	5413	$p(2775) - 5414$
richtig	1	0

Tabelle E.72: Ergebnisse *LMF* mit Variante 2.3 für TEI5 1. Durchlauf

Tabelle E.73: Ergebnisse *LMF* mit Variante 2.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.74: Ergebnisse *LMF* mit Variante 2.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	11	$p(1323) - 12$
richtig	1	0

Tabelle E.75: Ergebnisse *LMF* mit Variante 2.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	8084753	$p(52150) - 8084754$
richtig	1	0

Tabelle E.76: Ergebnisse *LMF* mit Variante 2.3 für TEI7 1. Durchlauf

Tabelle E.77: Ergebnisse *LMF* mit Variante 2.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.4

1	positiv	negativ
falsch	20	$p(44) - 21$
richtig	1	0

1	positiv	negativ
falsch	3928	$p(55) - 3929$
richtig	1	0

1	positiv	negativ
falsch	3117	$p(50) - 3118$
richtig	1	0

Tabelle E.78: Ergebnisse *LMF* mit Variante 2.4 für TEI1 1. Durchlauf

Tabelle E.79: Ergebnisse *LMF* mit Variante 2.4 für TEI2 1. Durchlauf

Tabelle E.80: Ergebnisse *LMF* mit Variante 2.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.81: Ergebnisse *LMF* mit Variante 2.4 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	3	$p(2247) - 4$
richtig	1	0

Tabelle E.82: Ergebnisse *LMF* mit Variante 2.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.83: Ergebnisse *LMF* mit Variante 2.4 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	33	$p(2775) - 34$
richtig	1	0

Tabelle E.84: Ergebnisse *LMF* mit Variante 2.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.85: Ergebnisse *LMF* mit Variante 2.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.86: Ergebnisse *LMF* mit Variante 2.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.87: Ergebnisse *LMF* mit Variante 2.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	10899025	$p(52150) - 10899026$
richtig	1	0

Tabelle E.88: Ergebnisse *LMF* mit Variante 2.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.1

1	positiv	negativ
falsch	1037	$p(44) - 1038$
richtig	1	0

1	positiv	negativ
falsch	3956	$p(55) - 3957$
richtig	1	0

1	positiv	negativ
falsch	3851	$p(50) - 3852$
richtig	1	0

Tabelle E.89: Ergebnisse *LMF* mit Variante 3.1 für TEI1 1. Durchlauf

Tabelle E.90: Ergebnisse *LMF* mit Variante 3.1 für TEI2 1. Durchlauf

Tabelle E.91: Ergebnisse *LMF* mit Variante 3.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	191	$p(2247) - 192$
richtig	1	0

Tabelle E.92: Ergebnisse *LMF* mit Variante 3.1 für TEI4 1. Durchlauf

Tabelle E.93: Ergebnisse *LMF* mit Variante 3.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.94: Ergebnisse *LMF* mit Variante 3.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	1608	$p(2775) - 1609$
richtig	1	0

Tabelle E.95: Ergebnisse *LMF* mit Variante 3.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.96: Ergebnisse *LMF* mit Variante 3.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	37	$p(1323) - 38$
richtig	1	0

Tabelle E.97: Ergebnisse *LMF* mit Variante 3.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.98: Ergebnisse *LMF* mit Variante 3.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	758477	$p(52150) - 758478$
richtig	1	0

Tabelle E.99: Ergebnisse *LMF* mit Variante 3.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.2

1	positiv	negativ
falsch	1097	$p(44) - 1098$
richtig	1	0

1	positiv	negativ
falsch	386	$p(55) - 387$
richtig	1	0

1	positiv	negativ
falsch	121	$p(50) - 122$
richtig	1	0

Tabelle E.100: Ergebnisse LMF mit Variante 3.2 für TEI1 1. Durchlauf

Tabelle E.101: Ergebnisse LMF mit Variante 3.2 für TEI2 1. Durchlauf

Tabelle E.102: Ergebnisse LMF mit Variante 3.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	524	$p(2247) - 525$
richtig	1	0

Tabelle E.103: Ergebnisse LMF mit Variante 3.2 für TEI4 1. Durchlauf

Tabelle E.104: Ergebnisse LMF mit Variante 3.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	3402	$p(2775) - 3403$
richtig	1	0

Tabelle E.105: Ergebnisse LMF mit Variante 3.2 für TEI5 1. Durchlauf

Tabelle E.106: Ergebnisse LMF mit Variante 3.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.107: Ergebnisse LMF mit Variante 3.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	115	$p(1323) - 116$
richtig	1	0

Tabelle E.108: Ergebnisse *LMF* mit Variante 3.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	379600	$p(52150) - 379601$
richtig	1	0

Tabelle E.109: Ergebnisse *LMF* mit Variante 3.2 für TEI7 1. Durchlauf

Tabelle E.110: Ergebnisse *LMF* mit Variante 3.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.3

1	positiv	negativ
falsch	4088	$p(44) - 4089$
richtig	1	0

1	positiv	negativ
falsch	2005	$p(55) - 2006$
richtig	1	0

1	positiv	negativ
falsch	1776	$p(50) - 1777$
richtig	1	0

Tabelle E.111: Ergebnisse *LMF* mit Variante 3.3 für TEI1 1. Durchlauf

Tabelle E.112: Ergebnisse *LMF* mit Variante 3.3 für TEI2 1. Durchlauf

Tabelle E.113: Ergebnisse *LMF* mit Variante 3.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.114: Ergebnisse *LMF* mit Variante 3.3 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	55881	$p(2247) - 55882$
richtig	1	0

Tabelle E.115: Ergebnisse *LMF* mit Variante 3.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.116: Ergebnisse *LMF* mit Variante 3.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	239768	$p(2775) - 239769$
richtig	1	0

Tabelle E.117: Ergebnisse *LMF* mit Variante 3.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.118: Ergebnisse *LMF* mit Variante 3.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	42748	$p(1323) - 42749$
richtig	1	0

Tabelle E.119: Ergebnisse *LMF* mit Variante 3.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.120: Ergebnisse *LMF* mit Variante 3.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	4912200	$p(52150) - 4912201$
richtig	1	0

Tabelle E.121: Ergebnisse *LMF* mit Variante 3.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.4

1	positiv	negativ
falsch	5105	$p(44) - 5106$
richtig	1	0

1	positiv	negativ
falsch	3598	$p(55) - 3599$
richtig	1	0

1	positiv	negativ
falsch	3421	$p(50) - 3422$
richtig	1	0

Tabelle E.122: Ergebnisse *LMF* mit Variante 3.4 für TEI1 1. Durchlauf

Tabelle E.123: Ergebnisse *LMF* mit Variante 3.4 für TEI2 1. Durchlauf

Tabelle E.124: Ergebnisse *LMF* mit Variante 3.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	762	$p(2247) - 763$
richtig	1	0

Tabelle E.125: Ergebnisse *LMF* mit Variante 3.4 für TEI4 1. Durchlauf

Tabelle E.126: Ergebnisse *LMF* mit Variante 3.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.127: Ergebnisse *LMF* mit Variante 3.4 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	6130	$p(2775) - 6131$
richtig	1	0

Tabelle E.128: Ergebnisse *LMF* mit Variante 3.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.129: Ergebnisse *LMF* mit Variante 3.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	141	$p(1323) - 142$
richtig	1	0

Tabelle E.130: Ergebnisse *LMF* mit Variante 3.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.131: Ergebnisse *LMF* mit Variante 3.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	788327	$p(52150) - 788328$
richtig	1	0

Tabelle E.132: Ergebnisse *LMF* mit Variante 3.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.1

1	positiv	negativ
falsch	0	$p(44) - 1$
richtig	1	0

1	positiv	negativ
falsch	516	$p(55) - 517$
richtig	1	0

1	positiv	negativ
falsch	185	$p(50) - 186$
richtig	1	0

Tabelle E.133: Ergebnisse *LMF* mit Variante 4.1 für TEI1 1. DurchlaufTabelle E.134: Ergebnisse *LMF* mit Variante 4.1 für TEI2 1. DurchlaufTabelle E.135: Ergebnisse *LMF* mit Variante 4.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	2	$p(2247) - 3$
richtig	1	0

Tabelle E.136: Ergebnisse *LMF* mit Variante 4.1 für TEI4 1. DurchlaufTabelle E.137: Ergebnisse *LMF* mit Variante 4.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	2	$p(2775) - 3$
richtig	1	0

Tabelle E.138: Ergebnisse *LMF* mit Variante 4.1 für TEI5 1. DurchlaufTabelle E.139: Ergebnisse *LMF* mit Variante 4.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.140: Ergebnisse *LMF* mit Variante 4.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.141: Ergebnisse *LMF* mit Variante 4.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

2	positiv	negativ
falsch	314549	$p(52150) - 314550$
richtig	1	0

Tabelle E.142: Ergebnisse *LMF* mit Variante 4.1 für TEI7 1. Durchlauf

Tabelle E.143: Ergebnisse *LMF* mit Variante 4.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.2

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	4132	$p(55) - 4133$
richtig	1	0

1	positiv	negativ
falsch	3847	$p(50) - 3848$
richtig	1	0

Tabelle E.144: Ergebnisse *LMF* mit Variante 4.2 für TEI1 1. Durchlauf

Tabelle E.145: Ergebnisse *LMF* mit Variante 4.2 für TEI2 1. Durchlauf

Tabelle E.146: Ergebnisse *LMF* mit Variante 4.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.147: Ergebnisse *LMF* mit Variante 4.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	0	$p(2247) - 1$
richtig	1	0

Tabelle E.148: Ergebnisse *LMF* mit Variante 4.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.149: Ergebnisse *LMF* mit Variante 4.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	0	$p(2775) - 1$
richtig	1	0

Tabelle E.150: Ergebnisse *LMF* mit Variante 4.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.151: Ergebnisse *LMF* mit Variante 4.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.152: Ergebnisse *LMF* mit Variante 4.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.153: Ergebnisse *LMF* mit Variante 4.2 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	445110	$p(52150) - 445111$
richtig	1	0

Tabelle E.154: Ergebnisse *LMF* mit Variante 4.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.3

1	positiv	negativ
falsch	5	$p(44) - 6$
richtig	1	0

1	positiv	negativ
falsch	6015	$p(55) - 6016$
richtig	1	0

1	positiv	negativ
falsch	6353	$p(50) - 6354$
richtig	1	0

Tabelle E.155: Ergebnisse *LMF* mit Variante 4.3 für TEI1 1. Durchlauf

Tabelle E.156: Ergebnisse *LMF* mit Variante 4.3 für TEI2 1. Durchlauf

Tabelle E.157: Ergebnisse *LMF* mit Variante 4.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	37	$p(2247) - 38$
richtig	1	0

Tabelle E.158: Ergebnisse *LMF* mit Variante 4.3 für TEI4 1. Durchlauf

Tabelle E.159: Ergebnisse *LMF* mit Variante 4.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.160: Ergebnisse *LMF* mit Variante 4.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	4006	$p(2775) - 4007$
richtig	1	0

Tabelle E.161: Ergebnisse *LMF* mit Variante 4.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.162: Ergebnisse *LMF* mit Variante 4.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	2	$p(1323) - 3$
richtig	1	0

Tabelle E.163: Ergebnisse *LMF* mit Variante 4.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.164: Ergebnisse *LMF* mit Variante 4.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	5433499	$p(52150) - 5433500$
richtig	1	0

Tabelle E.165: Ergebnisse *LMF* mit Variante 4.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.4

1	positiv	negativ
falsch	25	$p(44) - 26$
richtig	1	0

1	positiv	negativ
falsch	1286	$p(55) - 1287$
richtig	1	0

1	positiv	negativ
falsch	981	$p(50) - 982$
richtig	1	0

Tabelle E.166: Ergebnisse *LMF* mit Variante 4.4 für TEI1 1. Durchlauf

Tabelle E.167: Ergebnisse *LMF* mit Variante 4.4 für TEI2 1. Durchlauf

Tabelle E.168: Ergebnisse *LMF* mit Variante 4.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

2	positiv	negativ
falsch	1	$p(2247) - 2$
richtig	1	0

Tabelle E.169: Ergebnisse *LMF* mit Variante 4.4 für TEI4 1. Durchlauf

Tabelle E.170: Ergebnisse *LMF* mit Variante 4.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

2	positiv	negativ
falsch	31	$p(2775) - 32$
richtig	1	0

Tabelle E.171: Ergebnisse *LMF* mit Variante 4.4 für TEI5 1. Durchlauf

Tabelle E.172: Ergebnisse *LMF* mit Variante 4.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.173: Ergebnisse *LMF* mit Variante 4.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p(1323) - 1$
richtig	1	0

Tabelle E.174: Ergebnisse *LMF* mit Variante 4.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.175: Ergebnisse *LMF* mit Variante 4.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	500063	$p(52150) - 500064$
richtig	1	0

Tabelle E.176: Ergebnisse *LMF* mit Variante 4.4 für TEI7 2. Durchlauf

Literaturverzeichnis

- [Adm21] ADMINISTRATOR, IT: *Verfügbarkeit*. <https://www.it-administrator.de/lexikon/verfuegbarkeit.html>, 2021. [Online; letzter Zugriff 22.09.2021].
- [Ber19] BERLIN, SAM: *cglib 3.3.0*. https://github.com/cglib/cglib/releases/tag/RELEASE_3_3_0, 2019. [Online; letzter Zugriff 26.06.2021].
- [BNL⁺06] BAJRACHARYA, SUSHIL, TRUNG NGO, ERIK LINSTAD, YIMENG DOU, PAUL RIGOR, PIERRE BALDI CRISTINA LOPES: *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. OOPSLA '06, 681–682, New York, NY, USA, 2006. Association for Computing Machinery.
- [DeM05] DEMICHEL, LINDA: *EJB Core Contracts and Requirements*. https://download.oracle.com/otndocs/jcp/ejb-3_0-pr-spec-oth-JSpec/, 2005. [Online; letzter Zugriff 29.09.2021].
- [ES13] EILEBRECHT, KARL GERNOT STARKE: *Patterns kompakt*. Springer Verlag Berlin Heidelberg, 2013.
- [Fow04] FOWLER, MARTIN: *Inversion of Control Containers and the Dependency Injection pattern*. <https://martinfowler.com/articles/injection.html>, 2004. [Online; letzter Zugriff 13.10.2021].
- [GJS⁺13] GOSLING, JAMES, BILL JOY, GUY STEELE, GILAD BRACHA ALEX BUCKLEY: *The Java Language Specification*. <https://docs.oracle.com/javase/specs/jls/se7/html/index.html>, 2013. [Online; letzter Zugriff 14.09.2021].

- [Hal94] HALBACH, WULF R.: *Interfaces: medien- und kommunikationstheoretische Elemente einer Interface-Theorie*. Wilhelm Fink Verlag, München, 1994.
- [HJ13] HUMMEL, OLIVER WERNER JANJIC: *Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse*, 227–250. Springer New York, New York, NY, 2013.
- [Hum08] HUMMEL, OLIVER: *Semantic Component Retrieval in Software Engineering*. , April 2008.
- [inv20] *Java Plattform - Interface InvocationHandler*. <https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/InvocationHandler.html>, 2020. [Online; letzter Zugriff 27.08.2021].
- [jun21a] *JUnit 4*. <https://junit.org/junit4/>, 2021. [Online; letzter Zugriff 01.07.2021].
- [jun21b] *JUnit 4.13.2 API*. <https://junit.org/junit4/javadoc/latest/index.html>, 2021. [Online; letzter Zugriff 01.07.2021].
- [KA15] KESSEL, MARCUS COLIN ATKINSON: *Measuring the Superfluous Functionality in Software Components. Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE '15, 11–20, New York, NY, USA, 2015. Association for Computing Machinery.
- [KA16] KESSEL, MARCUS COLIN ATKINSON: *Ranking software components for reuse based on non-functional properties*. Inf. Syst. Frontiers, 18(5):825–853, 2016.
- [KA18] KESSEL, MARCUS COLIN ATKINSON: *Integrating reuse into the rapid, continuous software engineering cycle through test-driven search*. BOSCH, JAN, BRIAN FITZGERALD, MICHAEL GOEDICKE, HELENA HOLMSTRÖM OLSSON, MARCO KONERSMANN STEPHAN KRUSCHE (): *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering, RCoSE@ICSE 2018, Gothenburg, Sweden, May 29, 2018*, 8–11. ACM, 2018.
- [Kru92] KRUEGER, CHARLES W.: *Software Reuse*. ACM Comput. Surv., 24(2):131–183, 1992.

- [LLBO07] LAZZARINI LEMOS, OTAVIO AUGUSTO, SUSHIL KRISHNA BAJRACHARYA JOEL OSSHER: *CodeGenie: A Tool for Test-Driven Source Code Search. Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, 917?918, New York, NY, USA, 2007. Association for Computing Machinery.
- [Obj07] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. <https://www.omg.org/spec/UML/2.1.2/Superstructure/PDF>, 2007. [Online; letzter Zugriff 21.10.2021].
- [obj21] *Objenesis Release notes*. <http://objenesis.org/notes.html>, 2021. [Online; letzter Zugriff 26.06.2021].
- [PCM] PCMAG ENCYCLOPEDIA: *engine*. <https://www.pcmag.com/encyclopedia/term/engine>. [Online; letzter Zugriff 22.10.2021].
- [SED16] STOLEE, KATHRYN T., SEBASTIAN ELBAUM MATTHEW B. DWYER: *Code search with input/output queries: Generalizing, ranking, and assessment*. *Journal of Systems and Software*, 116:35–48, 2016.
- [Tre15] TREMBLAY, HENRI: *easymock-3.0*. <https://github.com/easymock/easymock/releases/tag/easymock-3.0>, 2015. [Online; letzter Zugriff 26.06.2021].
- [ZW95] ZAREMSKI, AMY MOORMANN JEANNETTE M. WING: *Signature Matching: A Tool for Using Software Libraries*. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146?170, 1995.