

Masterarbeit

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

Niels Gundermann

Themensteller: Univ. Prof. Dr. Friedrich Steimann

Betreuer: Univ. Prof. Dr. Friedrich Steimann

Lehrgebiet Programmiersysteme

Fachbereich Informatik

Inhaltsverzeichnis

Abbildungsverzeichnis	ii
Tabellenverzeichnis	iii
Listings	v
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau dieser Arbeit	2
2 Problemstellung	3
2.1 Testgetriebene Exploration von EJBs	3
2.2 Verwandte Arbeiten	3
3 Theoretische Grundlagen	5
3.1 Strukturelle Evaluation	5
3.1.1 Struktur für die Definition von Typen	5
3.1.2 Definition der Matchern	8
3.1.3 Ergebnis der strukturellen Evaluation	10
3.2 Generierung der Proxies auf Basis von Matchern	11
3.2.1 Struktur für die Definition von Proxies	12
3.2.2 Generierung von Proxies	18
3.2.3 Anzahl möglicher Proxies innerhalb einer Bibliothek	32
3.3 Semantische Evaluation	35
3.3.1 Besonderheiten der Testfälle	35

3.3.2	Algorithmus für die semantische Evaluation	36
3.4	Heuristiken	38
3.4.1	Beachtung des Matcherratings (LMF)	39
3.4.2	Beachtung positiver Tests (PTTF)	43
3.4.3	Beachtung fehlgeschlagener Methodenaufrufe (BL_NMC)	45
4	Implementierung	49
4.1	Modul: SignatureMatching	50
4.2	Modul: ComponentTester	56
4.3	Modul: DesiredComponentSourcerer	59
5	Evaluierung	63
6	Diskussion	67
7	Ausblick	69
8	Schlussbemerkung	71
	Literaturverzeichnis	71
A	Semantische Evaluation mit allen vorgestellten Heuristiken	75

Abbildungsverzeichnis

1.1	Abhängigkeiten von nachfragenden und angebotenen Komponenten	1
3.1	Delegation der Methode <code>heal</code>	15
3.2	Delegation der Methode <code>heal</code> mit Parametern in unterschiedlicher Reihenfolge .	16
3.3	Delegation der Methode <code>extinguishFire</code> mit Typkonvertierungen	17
3.4	AST für das Beispiel zum Sub-Proxy	19
3.5	AST für das Beispiel zum Content-Proxy	24
3.6	AST für das Beispiel zum Container-Proxy	26
3.7	AST für das Beispiel zum strukturellen Proxy	29
4.1	Architektur	49
4.2	Modul: <code>SignatureMatching</code>	51
4.3	Klassendiagramm: <code>StructuralTypeMatcher</code> und <code>MatchingInfos</code>	53
4.4	Klassendiagramm: <code>TypeMatcher</code> und <code>SingleMatchingInfo</code>	54
4.5	Klassendiagramm: <code>MethodMatchingInfo</code>	55
4.6	Klassendiagramm: <code>TypeConverter</code>	57
4.7	Modul: <code>ComponentTester</code>	58
4.8	Modul: <code>DesiredComponentSourcerer</code>	60

Tabellenverzeichnis

3.1	Struktur für die Definition einer Bibliothek von Typen	6
3.2	Grammatikregeln mit Erläuterungen für die Definition eines Proxies	12
3.3	Grammatikregeln mit Attributen für die Definition eines Proxies	14
3.4	Proxy-Arten mit Matchingrelationen und Proxy-Funktionen	31
4.1	Zuordnung der Matcher zu den Matcher- und Generator-Implementierungen . .	51
5.1	Beispiel: Vier-Felder-Tafel	65

Listings

3.1	Bibliothek <i>Example</i> von Typen	7
3.2	Einfache Methoden-Delegation	13
3.3	Methoden-Delegation mit Parametern in unterschiedlicher Reihenfolge	15
3.4	Methoden-Delegation mit Typkonvertierung	17
3.5	Sub-Proxy für Patient	18
3.6	Content-Proxy für Medicine	23
3.7	Container-Proxy für MedCabniet	25
3.8	Struktureller Proxy für MedicalFireFighter	28
3.9	Beispielhafte Implementierung einer eval-Methode	36
3.10	Semantische Evaluation ohne Heuristiken	37
3.11	Semantische Evaluation mit Heuristik LMF	42
3.12	Semantische Evaluation mit Heuristik PTTF	44
3.13	Evaluierung einzelner Proxies mit BL_MNC	46
3.14	Blacklist-Methode für Heuristik BL_NMC	46
3.15	Evaluation mehrere Proxies mit BL_MNC	47
A.1	Kombination aller Heuristiken	75

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 es nicht möglich ist, dass eine nachfragende Komponente einfach selbst spezifizieren kann, welche Informationen oder Services sie erwartet, wodurch auf der Basis dieser Spezifikation eine passende anbietende Komponente gefunden werden kann.

1.2 Aufbau dieser Arbeit

Kapitel 2

Problemstellung

2.1 Testgetriebene Exploration von EJBs

2.2 Verwandte Arbeiten

Ein solcher Ansatz wurde bereits in [BNL⁺06] von Bajaracharya et al. verfolgt. Diese Gruppe entwickelte eine Search Engine namens Sourcerer, welche Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens CodeGenie entwickelt, welches einem Softwareentwickler die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde erstmals der Begriff der Test-Driven Code Search (TDCS) etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummel [Hum08] ebenfalls eine Weiterentwicklung von Sourcerer veröffentlicht, welche unter dem Namen Merobase bekannt ist, welches ebenfalls das Konzept der TDCS verfolgt. TDCS beruht grundlegend darauf, dass der Entwickler Testfälle spezifiziert, die im Anschluss verwendet werden, um relevanten Source Code aus einem Repository hinsichtlich dieser Testfälle zu evaluieren. Damit kann das jeweilige Tool dem Entwickler Vorschläge für die Wiederverwendung bestehenden Codes unterbreiten.

Bezogen auf die am Ende des vorherigen Abschnitts formulierte Überlegung ermöglichen die genannten Search Engines, das Internet nach bestehendem Source Code zu durchsuchen und damit bereits bestehende Implementierungen für eine nachfragende Komponente zu ermitteln.

Kapitel 3

Theoretische Grundlagen

3.1 Strukturelle Evaluation

3.1.1 Struktur für die Definition von Typen

Die Typen seien in einer Bibliothek L in folgender Form zusammengefasst:

Regel	Erläuterung
$L ::= TD^*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \mid RD$	Eine Typdefinition kann entweder die Definition eines provided Typen (PD) oder eines required Typen (RD) sein.
$PD ::=$ $\text{provided } T \text{ extends } T'$ $\{FD^*MD^*\}$	Die Definition eines provided Typen 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 required Typen 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 .
$MD ::= T' \ m(T)$	Eine Methodendeklaration besteht aus dem Namen der Methode m , dem Namen des Parameter-Typs T und dem Namen des Rückgabe-Typs T' .

Tabelle 3.1: Struktur für die Definition einer Bibliothek von Typen

Weiterhin 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:

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

Beispiel-Bibliothek

```

provided Fire extends Object{}

provided ExtFire extends Fire{}

provided FireState extends Object{
    boolean isActive
}

provided Medicine extends Object{
    String getDescription()
}

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

provided Patient extends Injured{
    String getName()
}

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

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

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

provided MedCabinet extends Object{
    Medicine med
}

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

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

```

Listing 3.1: Bibliothek *Example* von Typen

3.1.2 Definition der Matchern

Ein Matcher definiert das Matching eines Typs T zu einem Typ T' durch die asymmetrische Relation $T \Rightarrow T'$.

ExactTypeMatcher

Der *ExactTypeMatcher* stellt ein Matching von einem Typ T zu demselben Typ T her. Die dazugehörige Matchingrelation \Rightarrow_{exact} wird durch folgende Regel beschrieben:

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

GenTypeMatcher

Der *GenTypeMatcher* stellt ein Matching von einem Typ T zu einem Typ T' mit $T > T'$ her. Die dazugehörige Matchingrelation \Rightarrow_{gen} wird durch folgende Regel beschrieben:

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

SpecTypeMatcher Der *SpecTypeMatcher* stellt im Verhältnis zum *GenTypeMatcher* das Matching in die entgegengesetzte Richtung dar. 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 zusammengefasst, wodurch sich die Matchingrelation $\Rightarrow_{internCont}$ ergibt:

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

ContentTypeMatcher

Der *ContentTypeMatcher* matcht einen Typ T auf einen Typ T' , wobei T' ein Feld enthält, auf dessen Typ T'' der Typ T über die Matchingrelation $\Rightarrow_{internCont}$ gematcht werden kann. So kann bspw. der Typ `boolean` aus Listing 1 auf den Typ `FireState` gematcht werden.

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` gelten:

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

ContainerTypeMatcher

Der *ContainerTypeMatcher* stellt im Verhältnis zum *ContentTypeMatcher* das Matching in die entgegengesetzte Richtung dar. So kann bspw. auch der Typ `FireState` auf den Typ `boolean` aus Listing 1 gematcht werden.

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

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

So gilt für die Typen `FireState` und `boolean`:

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

Zur Definition des letzten Matchers werden die Matchingrelationen der oben genannten Matcher noch einmal zusammengefasst. Dabei entsteht die Matchingrelation $\Rightarrow_{internStruct}$, welche

durch folgende Regel beschrieben wird:

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

StructuralTypeMatcher

Der *StructuralTypeMatcher* matcht einen *required Typ* R auf einen *provided Typ* P auf der Basis struktureller Eigenschaften der Methoden, die in den Typen deklariert sind.

Somit soll bspw. der Typ **MedicalFireFighter** auf den Typ **FireFighter** (siehe Listing 1) gematcht werden. Als ein weiteres Beispiel, bezogen auf die Typen aus Listing 1, kann das Matching des Typs **MedicalFireFighter** auf den 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. Die Menge der gematchten Methoden aus R in P wird wie folgt beschrieben:

$$structM(R, P) := \left\{ T' \ m(T) \left| \begin{array}{l} T' \ m(T) \in methoden(R) \wedge \\ \exists S' \ n(S) \in methoden(P) : \\ S \Rightarrow_{internStruct} T \wedge T' \Rightarrow_{internStruct} S' \end{array} \right. \right\}$$

Da die Notation es nicht hergibt, ist zusätzlich zu erwähnen, dass, sofern in m und n mehrere Parameter verwendet werden, deren Reihenfolge irrelevant ist.

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

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

3.1.3 Ergebnis der strukturellen Evaluation

Die gesamte Exploration wird für einen *required Typ* durchgeführt. Bei der strukturellen Evaluation sollen dabei 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.

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

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 folgenden Mengen von Target-Typen für die Bildung von Proxies für den required Typ **Greeting** ermittelt.

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

3.2 Generierung der Proxies auf Basis von Matchern

Ein Proxy wird in Abhängigkeit vom Matching zwischen dem Source- und den Target-Typen erzeugt. Im Folgenden werden zuerst die Matcher beschrieben. Im Anschluss wird auf die Generierung der Proxies eingegangen.

3.2.1 Struktur für die Definition von Proxies

Die Konvertierung eines Typs T aus einer Menge von provided Typen P wird durch *Proxies* beschrieben. Die Grammatikregeln für einen 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 Proxy wird für ein Typ T als Source-Typ mit einer Mengen von provided Typen $P = \{P_1, \dots, P_n\}$ als Target-Typen, einer Menge von Methoden-Delegationen erzeugt.
$MDEL ::=$ $CALLM \rightarrow DELM$	Eine <i>Methodendelelegation</i> besteht aus einer <i>aufgerufenen Methode</i> und aus einem <i>Delegationsziel</i> .
$CALLM ::=$ $REF.m(CP_1, \dots, CP_n) : CR$	Eine aufgerufene Methode besteht aus dem Namen der Methode m , dem Rückgabotyp CR und einer Menge von Parametertypen $\{CP_1, \dots, CP_n\}$.
$DELM ::=$ $REF.n(DP_1, \dots, DP_n) : DR$	Die erste Variante eines Delegationsziels besteht aus dem Namen der <i>Delegationsmethode</i> n , dem Rückgabotyp 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 Delegationsziels besteht aus einer Menge von Indizes $\{I_1, \dots, I_n\}$, einer <i>Referenz</i> , dem Namen der Delegationsmethode n , dem Rückgabotyp DR und einer Menge von Parametertypen $\{DP_1, \dots, DP_n\}$.
$DELM ::= \text{err}$	Die dritte Variante eines Delegationsziels 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 Referenz besteht aus einem Typ P_i .
$REF ::= P_i.f$	Die zweite Variante einer Referenz besteht aus einem Typ P_i und einem Feldnamen f .

Tabelle 3.2: Grammatikregeln mit Erläuterungen für die Definition 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, wird mit dem Wert *none* belegt. Ein Proxy bietet alle Methoden des Source-Typen an. Einige dieser Methoden werden an eine Methode delegiert, die von einem der Target-Typ des Proxies angeboten wird. Eine solche Delegation wird durch eine Methoden-Delegation (siehe 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.

2. Ausführung

Dieser Schritt meint die Durchführung der Delegationsmethode mit den übergeben Parametern aus Schritt 1. Dies schließt auch die Ermittlung des 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 aufgerufenen Methode, die vom Proxy angeboten wird, übergeben werden. Hier muss ebenfalls sichergestellt werden, dass die beiden Rückgabetypen der beiden Methoden zueinander passen.

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

Tabelle 3.3: Grammatikregeln mit Attributen für die Definition eines Proxies

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. An diesem Beispiel sind sowohl die Parameter- als auch die Rückgabe-

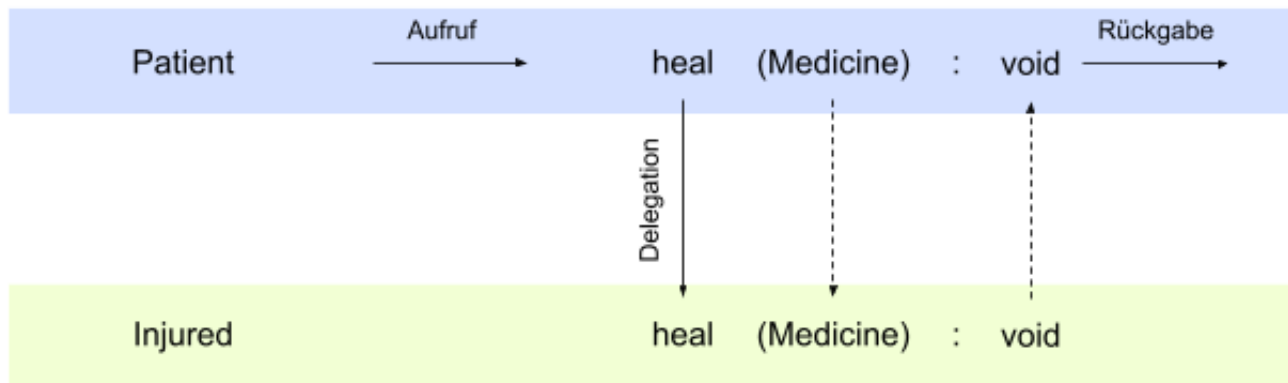


Abbildung 3.1: Delegation der Methode `heal`

Typen der aufgerufenen Methode und der Delegationsmethode identisch sind. 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 3.2.1 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 Delegationsmethoden 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. Dementsprechende wird er zweite Parameter der aufgerufenen Methoden (Index = 1) der Delegationsmethode als erster Parameter übergeben (siehe Abbildung 3.2).

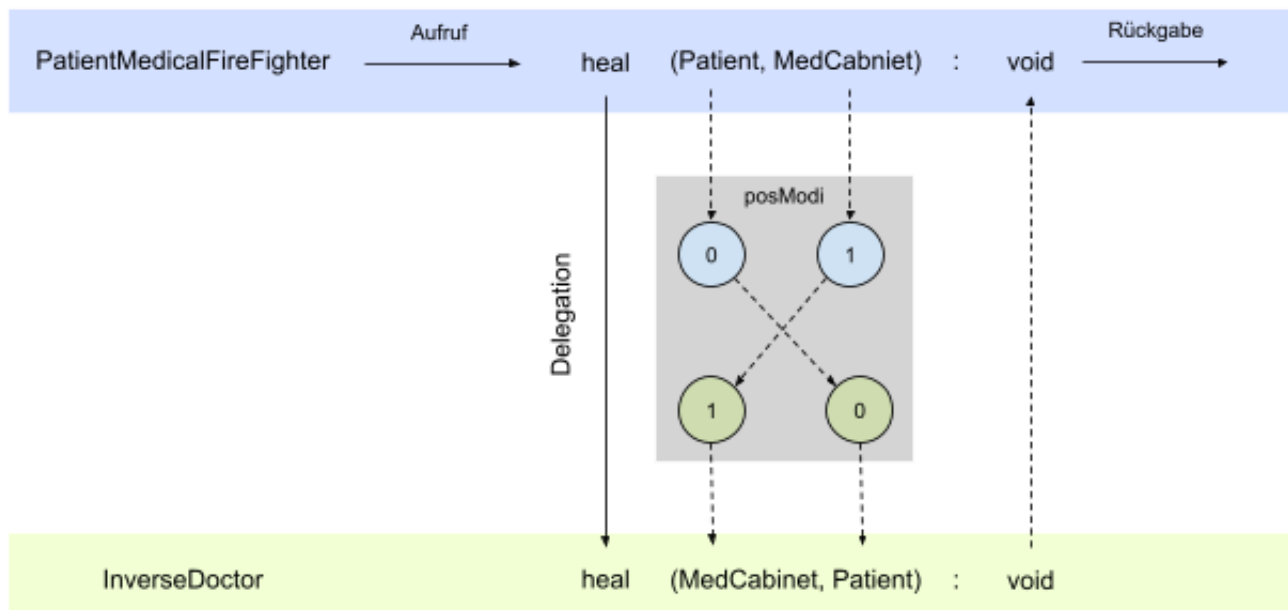


Abbildung 3.2: Delegation der Methode `heal` mit Parametern in unterschiedlicher Reihenfolge

Ein weiteres Beispiel soll zeigen, wie mit übergebenen Typen umzugehen ist, die nicht ohne Probleme übergeben werden können. Dafür ist jedoch vorab zu klären, wann dies der Fall ist.

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.

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.

Angenommen, die 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 4 für die Parameter-Typen einer der Proxies aus der Menge $proxies(Fire, \{ExtFire\})$ an die Delegationsmethode übergeben werden. Nach der Ausführung der Delegationsmethode müsste ein Proxy aus der Menge $proxies(boolean, \{FireState\})$ an die aufgerufenen Methode als Rückgabotyp übergeben werden. Der Sachverhalt wird in Abbildung 3.3 schematisch dargestellt.

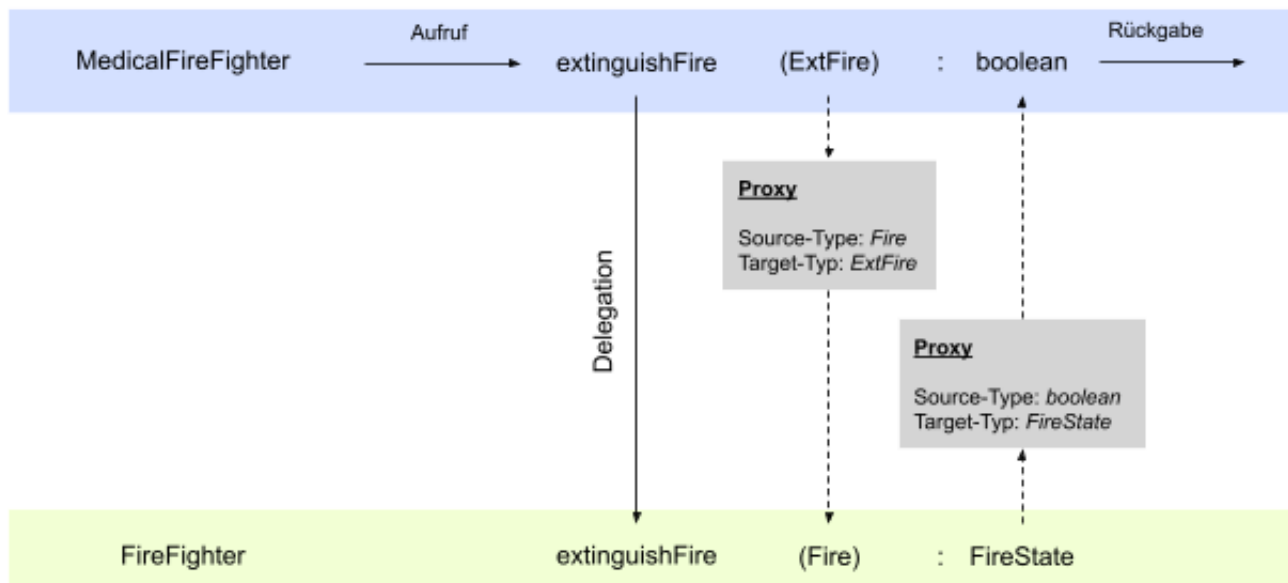


Abbildung 3.3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

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

3.2.2 Generierung von Proxies

Wie im Abschnitt 3.2.1 bereits erwähnt, 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 in der Liste befindlichen Elemente abgelegt ist.

Sub-Proxy

Die Voraussetzung für die Erzeugung eines *Sub-Proxy* vom Typ T aus einem Target-Typ T' ist $T \Rightarrow_{spec} T'$. 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 im folgenden Listing aufgeführt.

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

Listing 3.5: Sub-Proxy für Patient

Der abstrakte Syntaxbaum mit den dazugehörigen Attributen ist Abbildung 3.4 zu entnehmen.

¹ Der Proxy bietet alle Methoden an, die auch von dessen Source-Typ angeboten werden. Die

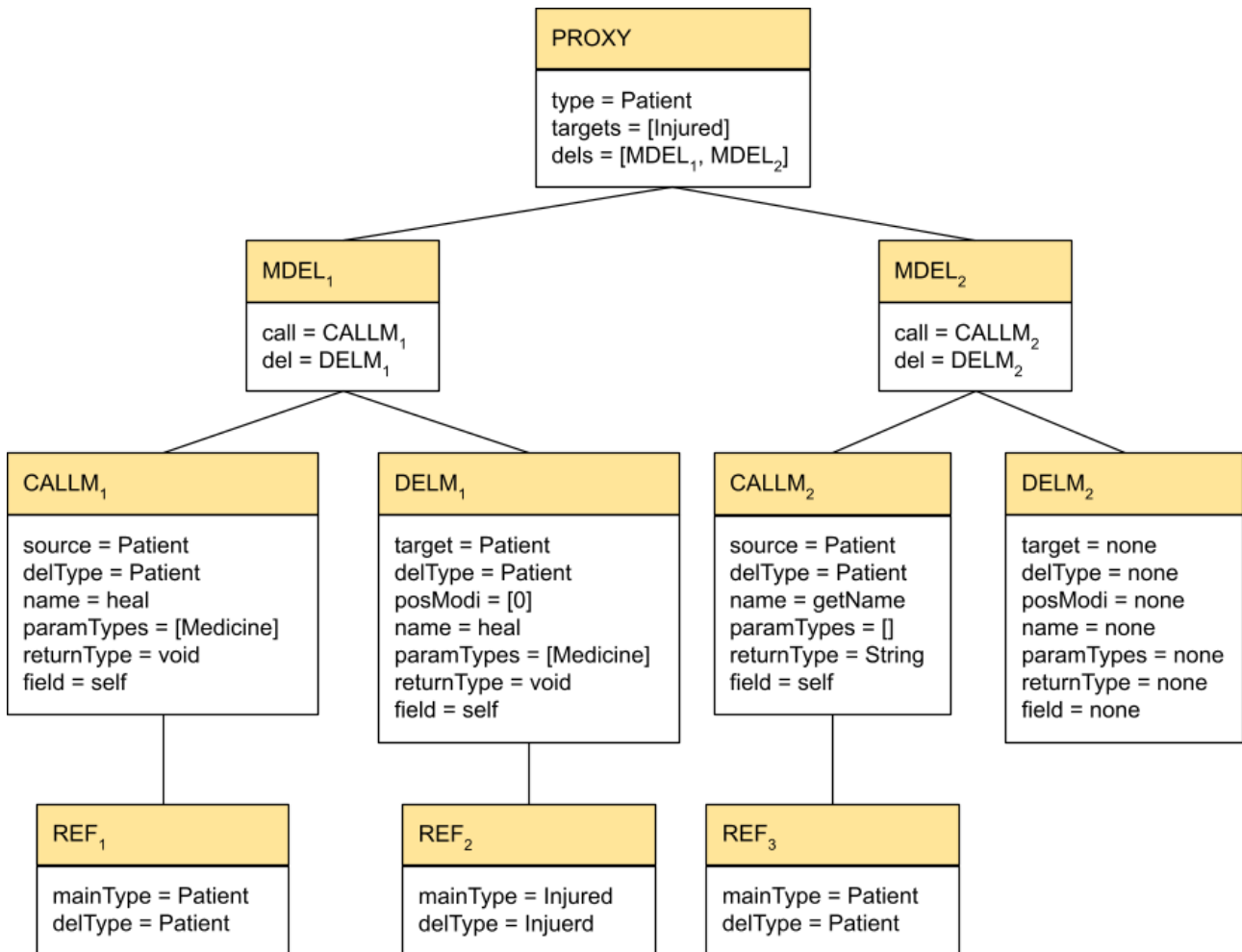


Abbildung 3.4: AST für das Beispiel zum Sub-Proxy

Methodendelegationen innerhalb des Proxies, beschreiben, was beim Aufruf der jeweiligen aufgerufenen Methoden 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 Delegationsmethode zur Verfügung steht.

Im Hinblick darauf, dass eine Konvertierung von einem Super-Typ und einen Sub-Typ (Down-

¹Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

Cast) ebenfalls dazu führt, dass bestimmte Methoden, wie in diesem Fall `getName` nicht ausgeführt werden können, spiegelt der *Sub-Proxy* dieses Verhalten wieder.

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.

$$\frac{|P.targets| = 1 \wedge \forall T' \in P.targets : T = T'}{targets_{single}(P, T)}$$

Darüber hinaus enthält ein *Sub-Proxy* P eine bestimmte Menge von Methoden-Delegationen. Dabei muss in allen Methodendelegationen das Attribut `field` der aufgerufenen Methoden mit dem der Delegationsmethoden übereinstimmen. Folgende Regel stellt diesen Sachverhalt für eine Menge von Methoden-Delegationen $MDList$ dar.

$$\frac{\forall MD_1 \in MDList : \neg(\exists MD_2 \in MDList : MD_1.call.field \neq MD_2.call.field \vee MD_1.del.field \neq MD_2.del.field)}{equalRefs(MDList)}$$

Für jede einzelne Methoden-Delegation MD gilt weiterhin, dass die aufgerufene Methode und die Delegationsmethode denselben Namen haben.

$$\frac{MD.call.name = MD.del.name}{methDel_{nominal}(MD)}$$

Die aufgerufene Methode muss dabei generell im Typ aus dem Attribut `call.delType` deklariert sein und die Delegationsmethode im Typ aus dem Attribut `del.delType`.

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

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

Zusätzlich muss das Attribut `field` im Attribut `call` mit dem Wert `self` belegt und das Attribut `mainType` mit dem Source-Typ des Proxies belegt sein.

$$\frac{MD.call.mainType = P.type \wedge MD.call.field = self}{callMethodDelType_{simple}(MD, P)}$$

Damit ist auch automatisch gewährleistet, dass die Attribute `mainType` und `delType` im Attribut `call` übereinstimmen. (siehe Tabelle 3.3)

Ähnliches gilt für die Attribute `field` und `mainType` im Attribut `del`. Hierbei muss der Wert des Attributs `mainType` jedoch mit dem Target-Typ des Proxies übereinstimmen.

$$\frac{MD.del.field = self \wedge MD.del.mainType \in P.targets}{delMethodDelType_{simple}(MD, P)}$$

Damit ist wiederum automatisch gewährleistet, dass die Attribute `mainType` und `delType` im Attribut `del` übereinstimmen. (siehe Tabelle 3.3)

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

$$\frac{callMethod_{simple}(MD) \wedge callMethodDelType_{simple}(MD, P)}{call_{simple}(MD, P)}$$

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

$$\frac{delMethod_{simple}(MD) \wedge delMethodDelType_{simple}(MD, P)}{del_{simple}(MD, P)}$$

Im *Sub-Proxy* ist darüber hinaus noch die Methoden-Delegation zu beachten, die bei einem Aufruf zu einem Fehler führt. Dieser Fall wird für eine Methoden-Delegation MD wie folgt beschrieben:

$$\frac{MD.del.name = none}{del_{err}(MD)}$$

Die genannten Regeln für eine Methoden-Delegation MD in einem *Sub-Proxy* lassen sich über die beiden folgenden Regeln beschreiben:

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge methDel_{nominal}(MD)}{methDel_{sub}(MD, P)}$$

$$\frac{call_{simple}(MD, P) \wedge del_{err}(MD)}{methDel_{sub}(MD, P)}$$

Innerhalb eines *Sub-Proxies* gibt es für jede Methode m des Source-Typ genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode. Damit lässt sich für einen Proxy P in Bezug auf alle seine Methoden-Delegationen folgende Regeln formulieren:

$$\frac{M = methoden(P.type) \wedge |M| = |P.dels| \wedge \forall T' m(T) \in M : \\ \exists MD \in P.dels : m = MD.call.name \wedge methDel_{sub}(MD, P)}{methDelList_{sub}(P)}$$

Für einen Proxy P kann die Regel $equalRefs(P)$ im Allgemeinen mit der Bedingung zusammengefasst werden, die besagt, dass ein Proxy immer einen bestimmten Source-Typ S haben muss. Die zusammengefasste Regel lautet:

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

Die Menge der *Sub-Proxies*, die mit dem Source-Typ T und dem Target-Typ T' erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{sub}(T, T') := \left\{ P \left| \begin{array}{l} proxy(P, T) \wedge \\ targets_{single}(P, T') \wedge \\ methDelList_{sub}(P) \end{array} \right. \right\}$$

Content-Proxy

Die Voraussetzung für die Erzeugung eines *Content-Proxies* vom Typ T aus einem Target-Typ T' ist $T \Rightarrow_{content} T'$. 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 `MedCabniet` delegiert.

Der abstrakte Syntaxbaum mit den dazugehörigen Attributen ist Abbildung 3.5 zu entnehmen.²

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 Methoden-Delegationen.

In den Delegationsmethoden einer einzelnen Methoden-Delegation MD dürfen die Attribute `mainType` und `delType` im *Content-Proxy* nicht identisch sein. Dementsprechend darf das Attribut `field` nicht mit dem Wert `self` belegt sein. Vielmehr muss für das Attribut `delType` und den Source-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont} MD.del.delType$ gelten. Daher gilt für den *Content-Proxy* die folgende Regel:

$$\frac{P.type \Rightarrow_{internCont} MD.del.delType \wedge MD.del.mainType \in P.targets}{delMethodDelType_{content}(MD, P)}$$

²Es wurden nur die Nonterminale mit den dazugehörigen Attributen aufgeführt.

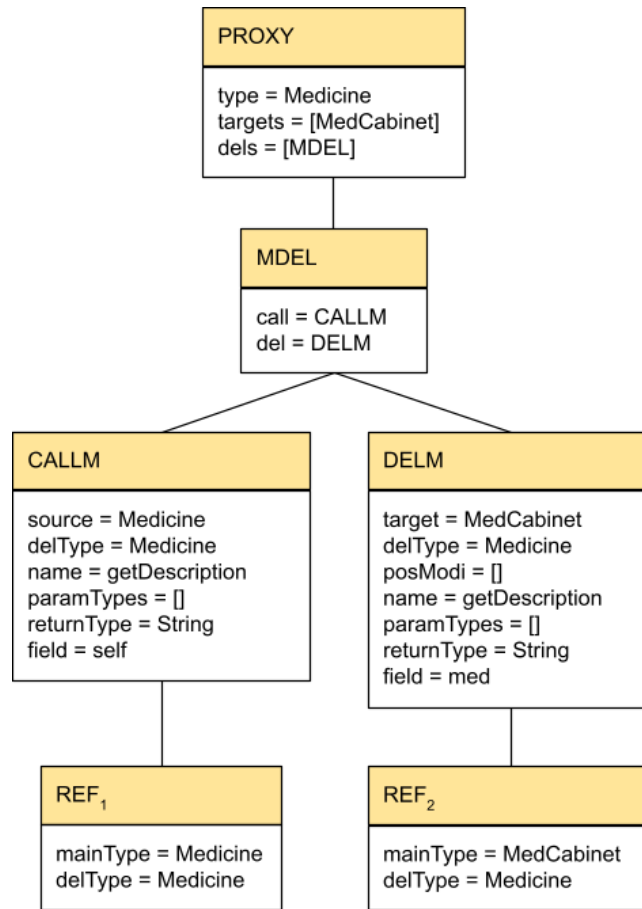


Abbildung 3.5: AST für das Beispiel zum Content-Proxy

Damit kann eine zusammenfassende Regel für die Delegationsmethoden einer Methoden-Delegation MD wie folgt definiert werden:

$$\frac{delMethod_{simple}(MD) \wedge delMethodDelType_{content}(MD, P)}{del_{content}(MD, P)}$$

Die zusammenfassende Regel für eine einzelne Methoden-Delegation MD innerhalb eines *Content-Proxies* hat die folgende Form:

$$\frac{call_{simple}(MD, P) \wedge del_{content}(MD, P) \wedge methDel_{nominal}(MD)}{methDel_{content}(MD, P)}$$

Wie auch im *Sub-Proxy* gibt es im *Content-Proxy* für jede Methode m des Source-Typen genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode. Daraus ergibt sich für alle Methoden-Delegationen aus einem *Content-Proxy* P folgende Regel:

$$\frac{M = \text{methoden}(P.\text{type}) \wedge |M| = |P.\text{dels}| \wedge \forall T' \ m(T) \in M : \\ \exists MD \in P.\text{dels} : m = MD.\text{call.name} \wedge \text{methDel}_{\text{content}}(MD, P)}{\text{methDelList}_{\text{content}}(P)}$$

Die Menge der *Content-Proxies*, die mit dem Source-Typ T und dem Target-Typ T' erzeugt werden, wird durch die folgende Funktion beschrieben.

$$\text{proxies}_{\text{content}}(T, T') := \left\{ P \left| \begin{array}{l} \text{proxy}(P, T) \wedge \\ \text{targets}_{\text{single}}(P, T') \wedge \\ \text{methDelList}_{\text{content}}(P) \end{array} \right. \right\}$$

Container-Proxy

Die Voraussetzung für die Erzeugung eines *Container-Proxies* vom Typ T aus einem Target-Typ T' ist $T \Rightarrow_{\text{container}} T'$. 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` $\Rightarrow_{\text{container}}$ `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 abstrakte Syntaxbaum mit den dazugehörigen Attributen ist Abbildung 3.6 zu entnehmen.³

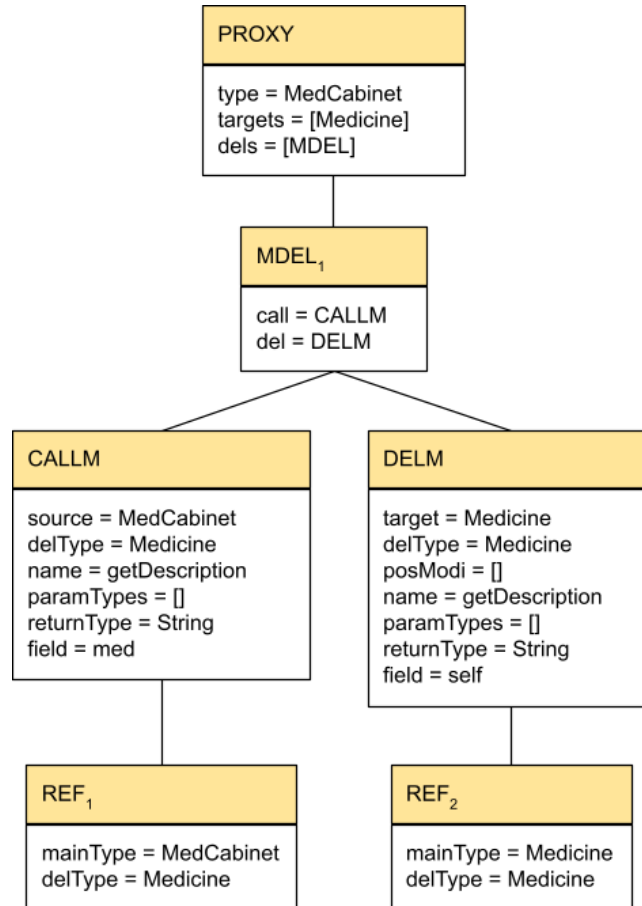


Abbildung 3.6: AST für das Beispiel zum Container-Proxy

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 gleichen denen aus dem *Sub-Proxy*.

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

In den angerufenen Methoden einer einzelnen Methoden-Delegation MD dürfen die Attribute `mainType` und `delType` im *Container-Proxy* nicht übereinstimmen. Dementsprechend darf das Attribut `field` nicht mit dem Wert `self` belegt sein. Vielmehr müssen der Wert des Attributs `delType` und der Target-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont} \text{delType}$ ausweisen. Daher gilt für den *Container-Proxy* P folgende Regel.

$$\frac{MD.call.mainType = P.type \wedge \forall T \in P.targets : T \Rightarrow_{internCont} MD.call.delType}{callMethodDelType_{container}(MD, P)}$$

Damit kann eine zusammenfassende Regel für die aufgerufenen Methoden wie folgt definiert werden:

$$\frac{callMethod_{simple}(MD) \wedge callMethodDelType_{container}(MD, P)}{call_{container}(MD, P)}$$

Die zusammenfassende Regel für eine einzelne Methoden-Delegation MD innerhalb eines *Container-Proxies* hat die folgende Form:

$$\frac{call_{container}(MD, P) \wedge del_{simple}(MD, P) \wedge methDel_{nominal}(MD)}{methDel_{container}(MD, P)}$$

Für einen *Container-Proxy* P gilt ebenfalls die Regel $equalRefs(P.dels)$. Daher müssen die Werte des Attributs `call.delType` aller Methoden-Delegationen des Proxies P übereinstimmen. Ferner muss es für jede Methode m des Typen aus `call.delType` genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode existieren. Daraus ergibt sich für alle Methoden-Delegationen aus einem *Content-Proxy* P folgende Regel:

$$\frac{M = methoden(P.dels[0].call.delType) \wedge |M| = |P.dels| \wedge \forall T' m(T) \in M : \exists MD \in P.dels : m = MD.call.name \wedge methDel_{container}(MD, P)}{methDelList_{container}(P)}$$

Die Menge der *Container-Proxies*, die mit dem Source-Typ T und dem Target-Typ T' erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{container}(T, T') := \left\{ P \left| \begin{array}{l} proxy(P, T) \wedge \\ target_{single}(P, T') \wedge \\ methDelList_{container}(P) \end{array} \right. \right\}$$

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 abstrakte Syntaxbaum mit den dazugehörigen Attributen ist Abbildung 3.7 zu entnehmen.⁴

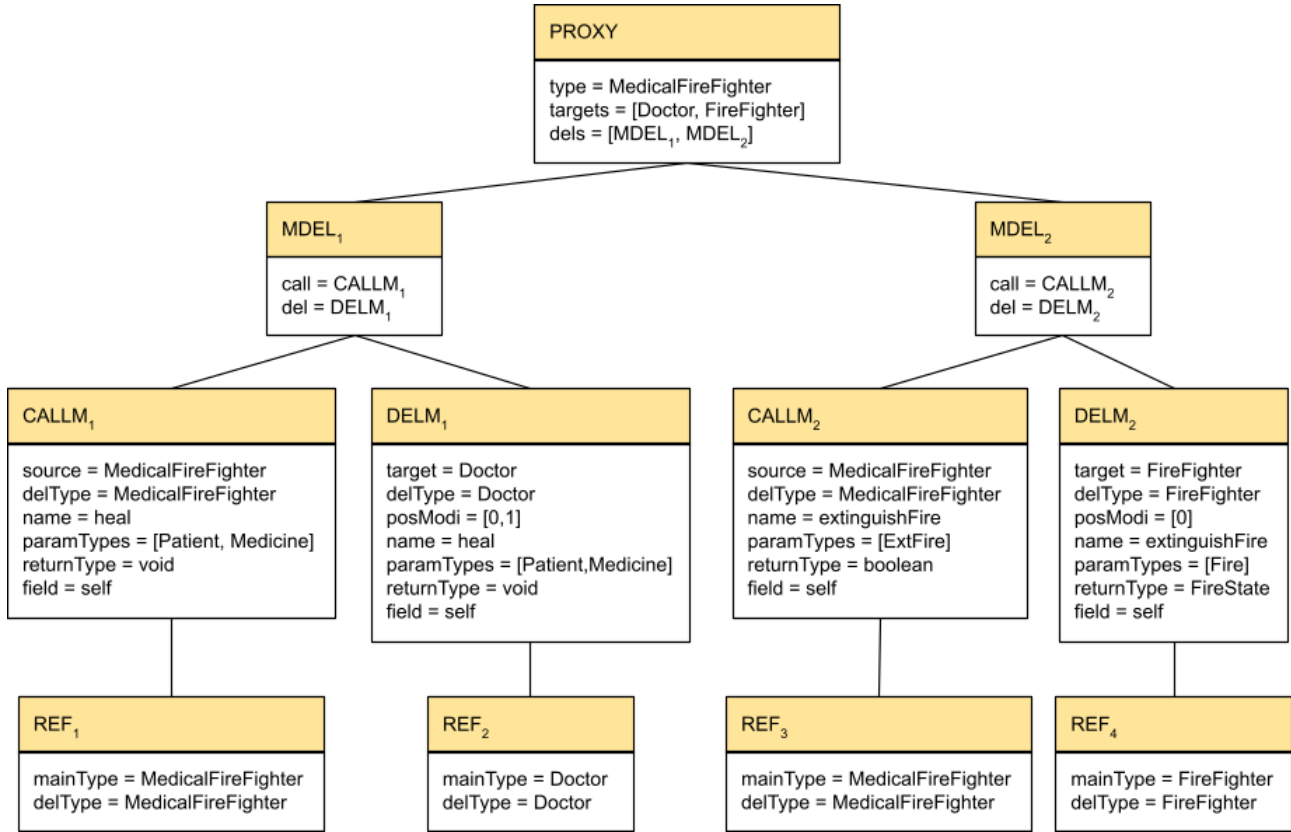


Abbildung 3.7: AST für das Beispiel zum strukturellen Proxy

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` existiert. Dadurch gilt die für einen *strukturellen Proxy* Proxy P :

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

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

Für die aufgerufene Methode und die Delegationsmethode einer einzelnen Methoden-Delegation M gelten im *strukturellen Proxy* dieselben Regeln wie für den *Sub-Proxy*. Die Namen der aufgerufenen Methode und der Delegationsmethode müssen dabei jedoch nicht übereinstimmen. Dafür müssen diese beiden Methode jedoch ein strukturelles Matching aufweisen. Bezogen auf die Rückgabe-Typen einer aufgerufenen Methode C und der Delegationsmethode D aus einer Methoden-Delegation muss daher Folgendes gelten.

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

Weiterhin muss für die Parameter-Typen gelten:

$$\frac{C.paramCount = 0}{params_{struct}(C, D)}$$

$$\frac{\forall i \in \{0, \dots, C.paramCount - 1\} : \quad C.paramTypes[i] \Rightarrow_{internStruct} D.paramTypes[D.posModi[i]]}{params_{struct}(C, D)}$$

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

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge \quad return_{struct}(MD.call, MD.del) \wedge params_{struct}(MD.call, MD.del)}{methDel_{struct}(MD, P)}$$

In einem *strukturellen Proxy* muss für jede Methode m des Source-Typen genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode existieren. Daraus ergibt sich für alle Methoden-Delegationen aus einem *strukturellen Proxy* P folgende Regel:

$$\frac{M = methoden(P.type) \wedge |M| = |P.dels| \wedge \forall T' \quad m(T) \in M : \quad \exists MD \in P.dels : MD.call.name = m \wedge methDel_{struct}(MD, P)}{methDelList_{struct}(P)}$$

Wie in Abschnitt Die Menge der *strukturellen Proxies*, die mit dem Source-Typ R und der Menge von Target-Typen T erzeugt werden, wird durch die folgende Funktion beschrieben.

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

Allgemeine Generierung von Proxies Die Proxy-Funktion 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 Proxy-Fukntionen 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.

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

3.2.3 Anzahl möglicher Proxies innerhalb einer Bibliothek

Die Generierung der Proxies für ein required Typ R aus der Bibliothek L erfolgt während der Exploration mit den Mengen von provided Typen aus $cover(R, L)$ (siehe Abschnitt 3.1.3). Mit einer Menge $T \in cover(R, L)$ können durchaus mehrere Proxies erzeugt werden. Das ist dann der Fall, wenn mehrere der Methoden, die in den provided Typen aus T deklariert wurden, mit einer Methode des required Typs R strukturell übereinstimmen. Die Anzahl der möglichen Proxies für ein required Typ R mit einer bestimmten Mengen von Target-Typen T_1, \dots, T_k ist somit von der Anzahl der Methoden abhängig, die in einem der Target-Typen des Proxies deklariert wurden und strukturell mit den Methoden aus R übereinstimmen.

Die Menge der Methoden der provided Typen aus einer Menge T , die strukturell mit einer Methoden mit der Struktur $A \rightarrow m(P)$ übereinstimmen, wird über die Funktion $structM_{target}$ beschrieben.

$$structM_{target}(A \rightarrow m(P), T) := \left\{ A' \rightarrow n(P') \left| \begin{array}{l} \exists T_i \in T : \\ A' \rightarrow n(P') \in methoden(T_i) \wedge \\ P' \Rightarrow_{internStruct} P \wedge \\ A \Rightarrow_{internStruct} A' \end{array} \right. \right\}$$

Sei R ein required Typ und T eine Menge von provided Typen innerhalb einer Bibliothek L mit $T \in cover(R, L)$. Sei weiterhin $\{m_1, \dots, m_n\} = methoden(R)$. Dann bilden M_1, \dots, M_n wie folgt die Mengen der Methoden der Target-Typen in T , die mit jeweils einer Methode $m_i \in methoden(R)$ strukturell übereinstimmen.

$$\begin{aligned} M_1 &= structM_{target}(m_1, T) \\ &\dots \\ M_n &= structM_{target}(m_n, T) \end{aligned}$$

Für jede Kombination von jeweils einem Element aus jeder der Mengen M_1, \dots, M_n kann ein Proxy für R mit der Menge der Target-Typen T erzeugt werden.

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 Menge von übereinstimmenden Methoden über die Funktion $structM_{target}$:

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

Darauf aufbauend 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
}
```

Für die Bildung eines Proxies wird aus jeder der oben genannten Menge M_1, \dots, M_n genau ein Element als Delegationsmethode verwendet werden. Die Anzahl aller möglichen Proxies für ein required Typ R aus einer Menge von Target-Typen T und unter der Annahme, dass $\{m_1, \dots, m_n\} = \text{methoden}(R)$, sei über die Funktion $proxyCount(R, T)$ ausgedrückt. Für $proxyCount(R, T)$ ist zu beachten, dass es sich dabei lediglich um eine Annäherung an die

tatsächliche Anzahl der Proxies handelt, die unter den oben beschriebenen Bedingungen erzeugt werden können. Dies liegt daran, dass eine Delegationsmethoden $dm \in M_1 \cup \dots \cup M_n$ innerhalb eines Proxy maximal einmal verwendet werden darf. Es ist jedoch möglich, dass es zwischen den oben genannten Mengen M_1, \dots, M_n Überschneidungen gibt (siehe vorheriges Beispiel). Daher gelten für die Funktion *proxyCount* folgende Regeln unter den oben genannten Modalitäten:

$$\frac{M_1 \cap \dots \cap M_n = \emptyset}{\text{proxyCount}(R, T) = \prod_{i=1}^n |M_i|}$$

$$\frac{M_1 \cap \dots \cap M_n \neq \emptyset}{\text{proxyCount}(R, T) < \prod_{i=1}^n |M_i|}$$

Im Allgemeinen gilt demnach:

$$\text{proxyCount}(R, T) \leq \prod_{i=1}^n |\text{struct}M_{\text{target}}(m_i, T)| \left| \left\{ \begin{array}{c} m_1, \\ \dots, \\ m_n \end{array} \right\} = \text{methoden}(R) \right.$$

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 Proxies über die Funktion *proxyCount* für alle Elemente aus $\text{cover}(R, L)$ ermittelt und summiert werden. Die folgende Funktion beschreibt diesen Sachverhalt für einen required Typ R aus einer Bibliothek L .

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

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 die gesamte Exploration zur Laufzeit des Programms durchgeführt wird, stellt sie hinsichtlich der nicht-funktionalen Anforderungen eine zeitkritische Komponente dar.

Da die Anforderungen an die gesuchte Komponente mit bedacht spezifiziert werden müssen, ist es irrelevant, ob es mehrere Proxies gibt, die hinsichtlich der vordefinierten Testfällen positiv evaluiert 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 des Proxies, die bei der Durchführung der eval-Methode 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-Objekt, wird der Name der dieser Methode in der Liste im Feld `triedMethodCalls` ergänzt.
2. Wenn der Proxy den Test erfüllt, wird der Wert `true` zurückgegeben. Anderenfalls wird der Wert `false` zurückgegeben.

Beispiel 3 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 x, Integer y)` anbietet. Der Fehlschlag (`err`) dieser Methode wird über einen Try-Catch-Block abgefangen.

```

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

Bei der Exploration soll letztendlich in einer Bibliothek L zu einem vorgegebenen required Type R ein Proxy gefunden werden. Die Mengen der Target-Typen auf deren Basis mehrere Proxies erzeugt werden können, wurden im Abschnitt ?? über $cover(R, L)$ beschrieben. Die in $T = cover(R, L)$ befindlichen Mengen können eine unterschiedliche Anzahl von Target-Typen enthalten. Die maximale Mächtigkeit einer Menge $T_i \in T$ 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 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 verwendet werden.

Sei folgende Funktion für eine Menge von Target-Typen $T \in cover(R, L)$ und eine ganze Zahl

$a > 0$ definiert:

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

Ausgehend von einer Bibliothek L kann der Algorithmus für die semantische Evaluation der Proxies, die für einen required Typ R mit den Mengen der Target-Typen $T = \text{cover}(R, L)$ erzeugt werden können, und der Menge von Tests (Parameter **tests**) 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 ?? wie beschrieben definiert sind.

```

1  passedTests = 0
2
3  function semanticEval( R, T, tests ){
4      for( i = 1; i <= maxTargets(R); i++ ){
5          relProxies = relevantProxies( R, T, i )
6          proxy = evalProxies( relProxies, tests )
7          if( proxy != null ){
8              // passenden Proxy gefunden
9              return proxy
10         }
11     }
12     // kein passenden Proxy gefunden
13     return null;
14 }
15
16 function relevantProxies(R, T, anzahl){
17     proxies = []
18     targetSets = targetSets(T, anzahl)
19     for( targets : targetSets ){
20         proxies.addAll( proxies(R, targets) )
21     }
22     return proxies;
23 }
24
25 function evalProxies(proxies, tests){
26     for( proxy : proxies ){
27         passedTests = 0
28         evalProxy(proxy, tests)
29         if( passedTests == tests.size ){
30             // passenden Proxy gefunden
31             return proxy

```

```
32     }
33 }
34 // kein passenden Proxy gefunden
35 return null
36 }
37
38 function evalProxy(proxy, tests){
39     for( test : tests ){
40         if( !test.eval( proxy ) ){
41             \\ wenn ein Test fehlschlaegt, dann entspricht der
42             \\ Proxy nicht den semantischen Anforderungen
43             return
44         }
45         passedTests = passedTests + 1
46     }
47 }
```

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 ?? Funktion *proxyCount*). Im schlimmsten Fall müssen alle Proxies hinsichtlich der vordefinierten Tests erzeugt und evaluiert 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 Proxies auf der Basis eines so genannten Matcherratings bewertet. Bei dem Matcherrating eines Proxies handelt es sich um einen numerischen Wert. Um diesen Wert 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 :

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

Dabei ist zu erwähnen, dass einige der o.g. Matcher über dasselbe Basisrating verfügen. Das liegt daran, dass sie technisch jeweils gemeinsam umgesetzt wurden.⁵

Das Matcherrating eines Proxies P wird über die Funktion *rating* beschrieben. Dieses ist von dem Matcherrating der Methoden-Delegation innerhalb von P abhängig. Das Matcherrating einer Methoden-Delegation ist von den Basisratings der Matcher abhängig, über die die Parameter- und Rückgabe-Typen der aufgerufenen Methode und der Delegationsmethoden gematcht werden können.

Für die Definition von Funktionen gelten im weiteren Verlauf folgende verkürzte Schreibweise

⁵Der *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)

in Bezug auf eine Methoden-Delegation MD :

$$\begin{aligned}
 pc &:= MD.call.paramCount \\
 cRT &:= MD.call.returnType \\
 dRT &:= MD.del.returnType \\
 cPT &:= MD.call.paramTypes \\
 dPT &:= MD.del.paramTypes \\
 pos &:= MD.call.posModi
 \end{aligned}$$

Darauf aufbauend sei die Menge der Matcherratings der Paare von Parameter- und Rückgabetypen aus der aufgerufenen Methode und den Delegationsmethode einer Methoden-Delegation MD wie folgt definiert:

$$bases_{MD}(MD) := base(dRT, cRT) \cup \bigcup_{i=0}^{pc-1} base(cPT[i], dPT[pos[i]])$$

Das Matcherrating einer Methoden-Delegation MD sei über die Funktion $mdRating$ beschrieben. Für die Definition der beiden Funktionen $rating$ und $mdRating$ gibt es unterschiedliche Möglichkeiten. In dieser Arbeit werden 4 Varianten als Definitionen vorgeschlagen, die in Kapitel 5 untersucht werden.

Dazu seien die folgenden Hilfsfunktionen definiert:

$$sum(v_1, \dots, v_n) = \sum_{i=1}^n v_i$$

$$max(v_1, \dots, v_n) = v_m | 1 \leq m \leq n \wedge \forall i \in \{1, \dots, n\} : v_i \leq v_m$$

$$min(v_1, \dots, v_n) = v_m | 1 \leq m \leq n \wedge \forall i \in \{1, \dots, n\} : v_i \geq v_m$$

Für die folgenden Vorschläge zur Definition von *rating* und *mdRating* sei P ein struktureller Proxy mit n Methoden-Delegation.

Variante 1: Durchschnitt

$$mdRating(MD) = \frac{sum(bases_{MD}(MD))}{pc + 1}$$

$$rating(P) = \frac{sum(mdRating(P.dels[0]), \dots, mdRating(P.dels[n - 1]))}{n}$$

Variante 2: Maximum

$$mdRating(MD) = max(bases_{MD}(MD))$$

$$rating(P) = \frac{max(mdRating(P.dels[0]), \dots, mdRating(P.dels[n - 1]))}{n}$$

Variante 3: Minimum

$$mdRating(MD) = min(bases_{MD}(MD))$$

$$rating(P) = \frac{min(mdRating(P.dels[0]), \dots, mdRating(P.dels[n - 1]))}{n}$$

Variante 4: Durchschnitt aus Minimum und Maximum

$$mdRating(MD) = \frac{max(bases_{MD}(MD)) + min(bases_{MD}(MD))}{2}$$

$$rating(P) = \frac{max(mdRating(P.dels[0]), \dots, mdRating(P.dels[n - 1])) + min(mdRating(P.dels[0]), \dots, mdRating(P.dels[n - 1]))}{2}$$

Da die Funktion *rating* von *mdRating* abhängt und für *mdRating* 4 Variante vorgeschlagen wurden, ergeben sich für jede vorgeschlagene Variante für die Definition von *rating* weitere 4

Varianten. Dadurch sind insgesamt 16 Varianten für die Definition von *rating* gegeben.

Zur Anwendung der Heuristik muss das Matcherrating bei der Iteration über die erzeugten Proxies beachtet werden. Dabei sollte die Liste der Proxies, über die in der Methode `evalProxies` iteriert wird, entsprechend dem Matcherrating sortiert werden. Eine Sortierung ist nur vor dem Beginn der Iteration in der Methode `evalProxies` sinnvoll. Listing 3.11 zeigt die Anpassungen der Methode `evalProxies` 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 evalProxies(proxies, tests){
2      sorted = LMF( proxies )
3      for( proxy : sorted ){
4          passedTests = 0
5          evalProxy(proxy, tests)
6          if( passedTests == tests.size ){
7              // passenden Proxy gefunden
8              return proxy
9          }
10     }
11     // kein passenden Proxy gefunden
12     return null
13 }
14
15 function LMF( proxies ){
16     for ( n=proxies.size(); n>1; n--){
17         for( i=0; i<n-1; i++){
18             if( rating( proxies[i] ) < rating( proxies[i+1] ) ){
19                 tmp = proxies[i]
20                 proxies[i] = proxies[i+1]
21                 proxies[i+1] = tmp
22             }
23         }
24     }
25     return proxies
26 }

```

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 ein Typ in dem die Delegationsmethode spezifiziert ist. 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. Die 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:

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

Beweis: Sei R ein required Typ aus der Bibliothek L . Sei weiterhin $T \in TG$ und $TG \in cover(R, L)$.

Für die in diesem Abschnitt beschriebene Heuristik mit dem Namen *positiv tested targets first* (kurz: PTTF) ist das Ergebnis einzelner Tests in Bezug auf einen Proxy P relevant. Es wird davon ausgegangen, dass wenn ein Testfall durch einen Proxy P erfolgreich durchgeführt wird, 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. Die 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 entsprechend 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, der 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 Methoden `getMethodDelegations`. Beide Methode 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 für die Heuristik BL_NMC basieren auf den Funktionen aus Listing ???. 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_MNC

Der Pseudo-Code für die semantische Evaluation mit der Kombination aller genannten Heuristiken ist im Anhang A 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-

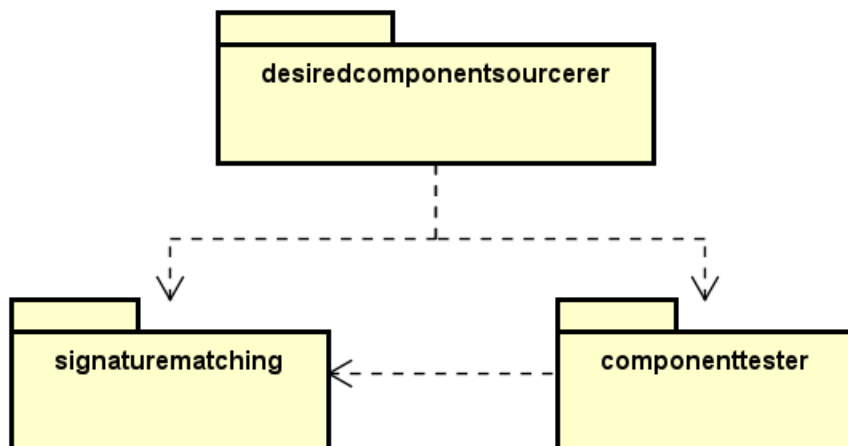


Abbildung 4.1: Architektur

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 [Tac21]
- 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, und den Klassen, die die Implementierung des Generators für den Proxy, der auf Basis des Matchers Anwendung findet, dargestellt. Die Klasse `StructuralTypeMatcher` nimmt dabei eine Sonderstellung ein. Dies ist daran zu erkennen, dass dieser nicht das Interface `TypeMatcher` implementiert. Dies wird damit begründet, 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 die Aggregation zwischen der Klasse `StructuralTypeMatcher` und dem Interface `TypeMatcher` angedeutet wird.

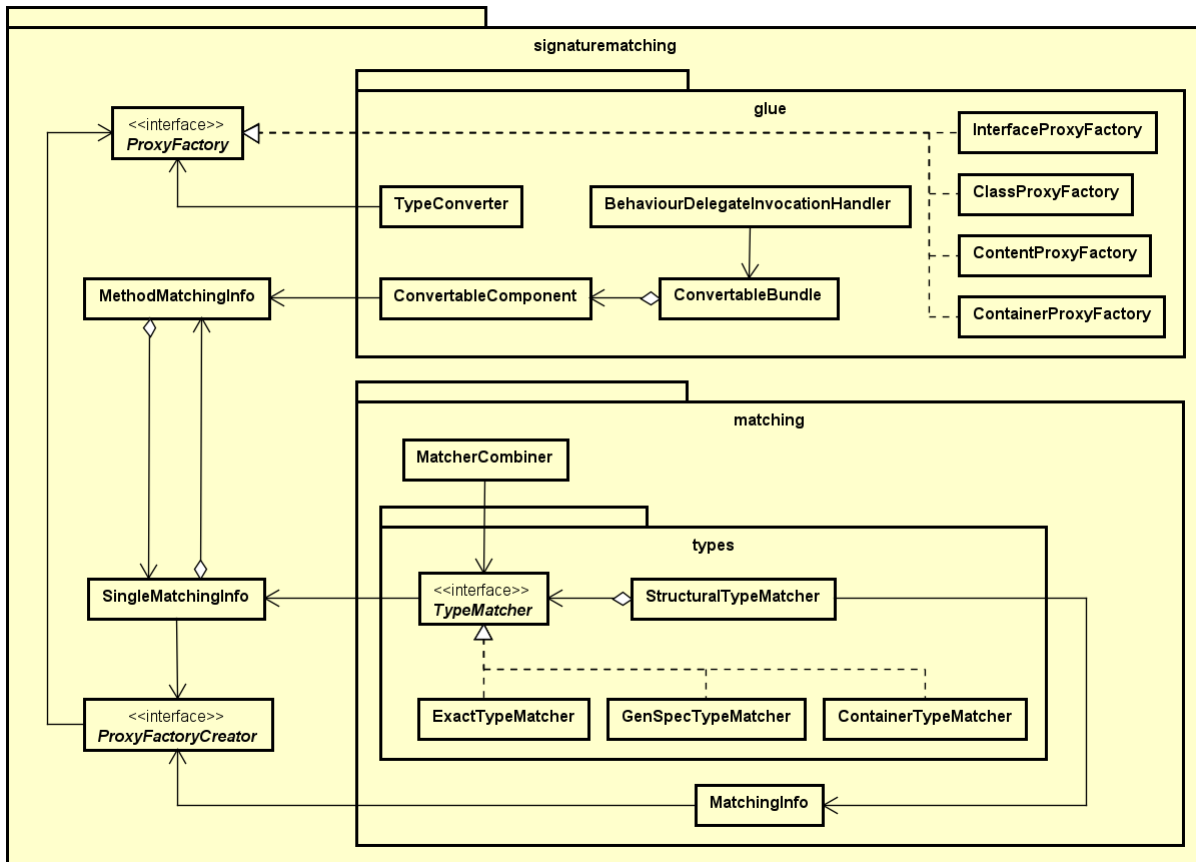


Abbildung 4.2: Modul: SignatureMatching

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 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 Typ `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 weitere `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` bewerkstelligen.

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 gemachten 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 in einem `MethodSupplier` überliefert wird.

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

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

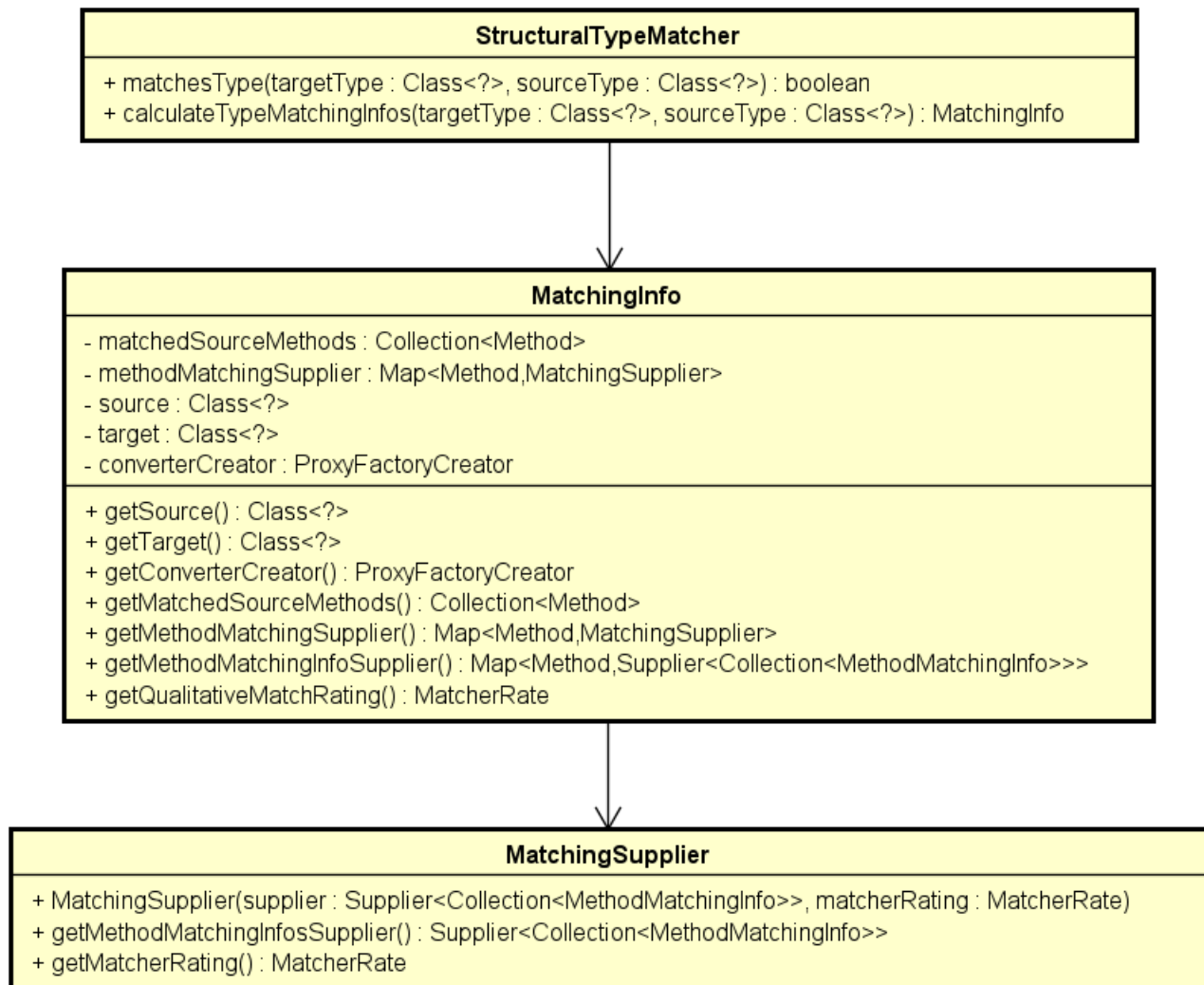
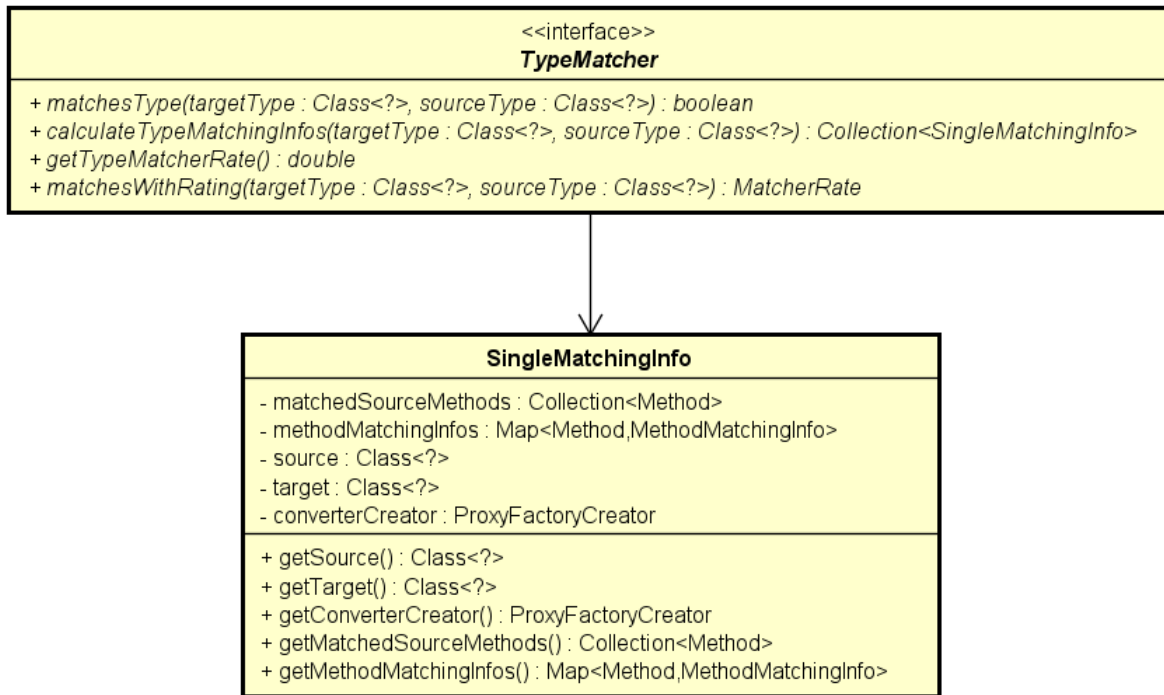


Abbildung 4.3: Klassendiagramm: StructuralTypeMatcher und MatchingInfos

Delegationsmethode hinterlegt.

Bezüglich der Klasse `SingleMatchingInfo` ist noch das Attribut `proxyFactoryCreator` zu beschreiben. Darin werden Informationen bzgl. der strukturellen Verbindung von 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 von 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 erwähnt 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 steht. Dies gilt insbesondere für die Erzeugung von Proxies für Klassen, die mit dem Schlüsselwort `final` versehen sind.

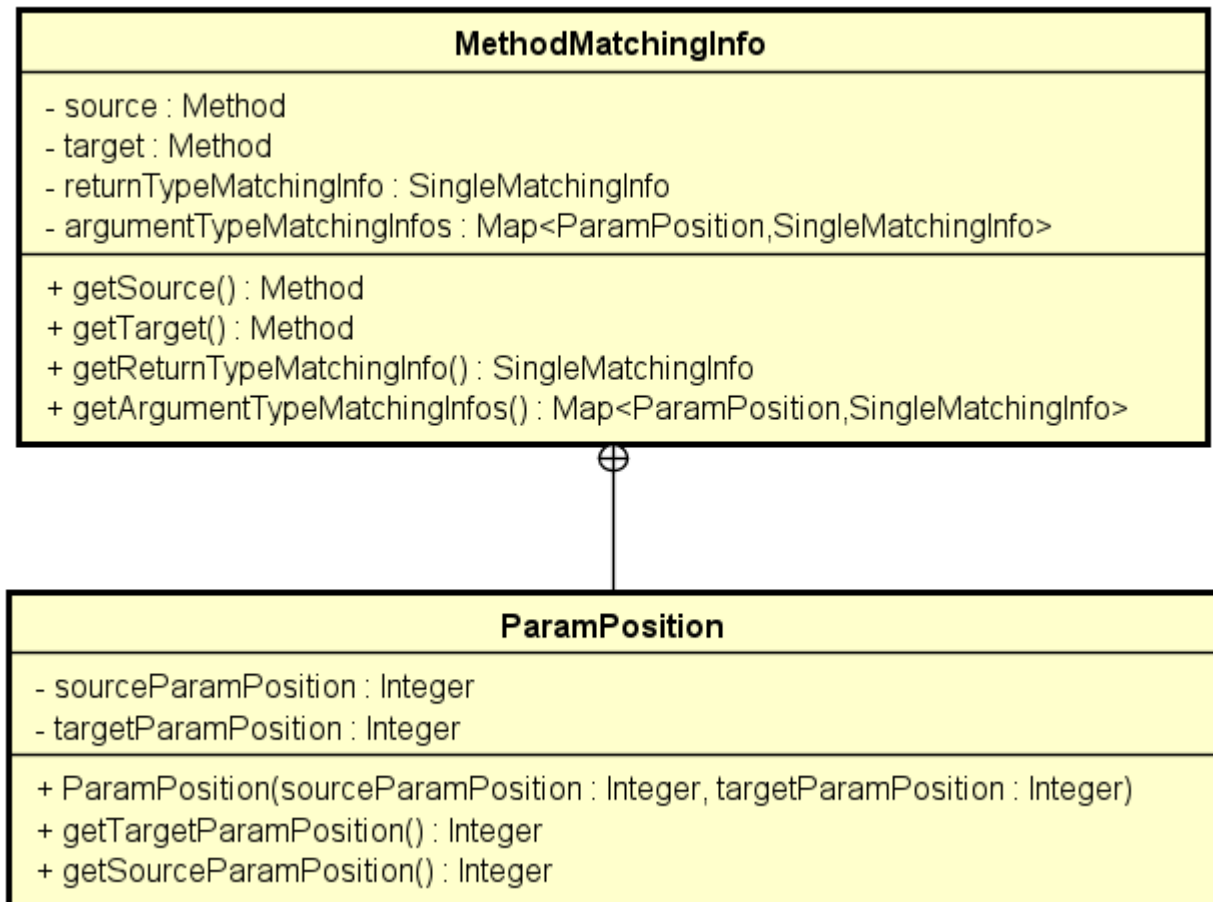


Abbildung 4.5: Klassendiagramm: MethodMatchingInfo

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.

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

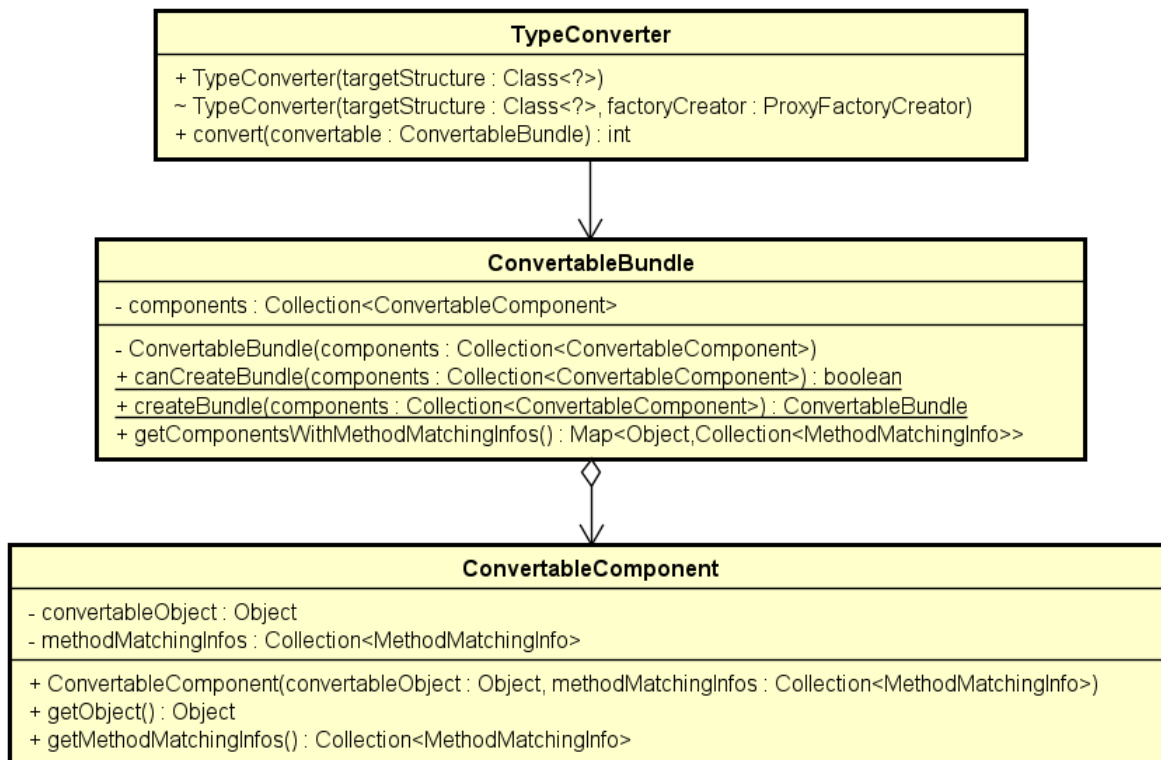
gehalten. Diese Map enthält alle notwendigen Information für das Erzeugen eines Proxies für die Parametertypen der Delegationsmethoden aus den Parametertypen der aufgerufenen Methode, sowie der Anpassung der Übergabeposition bei der Delegation der aufgerufenen Methode (siehe auch Abschnitt 3.2.1).

Um die Methoden-Delegationen zu koordinieren, wird bei der Erzeugung des Proxies in der jeweiligen `ProxyFactory` für das Proxy-Objekt ein `InvocationHandler` instanziiert (vgl. [?]). 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 Zugriff 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 Typdarstellen, 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 Beispiel für die Deklaration eines required Typ in Form eines Java-Interfaces und den dazugehörigen Testklassen ist im Anhang zu finden.

Abbildung 4.6: Klassendiagramm: *TypeConverter*

Damit die Testmethoden in den Testklassen die in Abschnitt 3.3.1 beschriebenen Eigenschaften aufweisen und durch das *ComponentTester*-Modul auffindig 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 Sichtbarkeit her öffentlich (`public`) sein. Weiterhin dürfen die Testmethoden keine Parameter erwarten und müssen mit der Annotation `RequiredTypeTest` markiert sein. Die Erwartungen

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

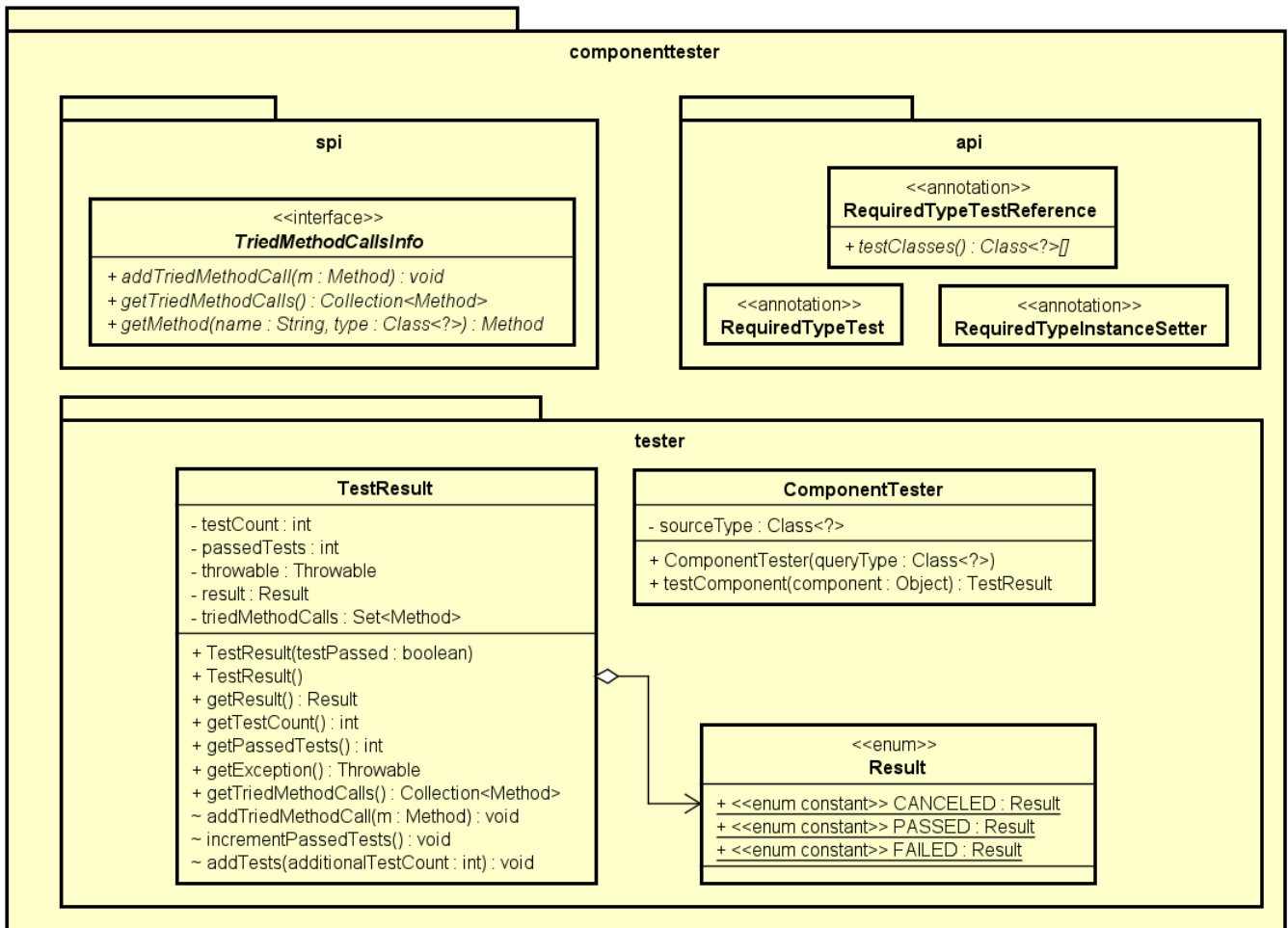


Abbildung 4.7: Modul: ComponentTester

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 diese innerhalb von Methoden erzeugt werden, die mit den in JUnit 4 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 Abbil-

dung 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 ist ein Beispiel für die korrekte Implementierung einer Testklasse zu finden.

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

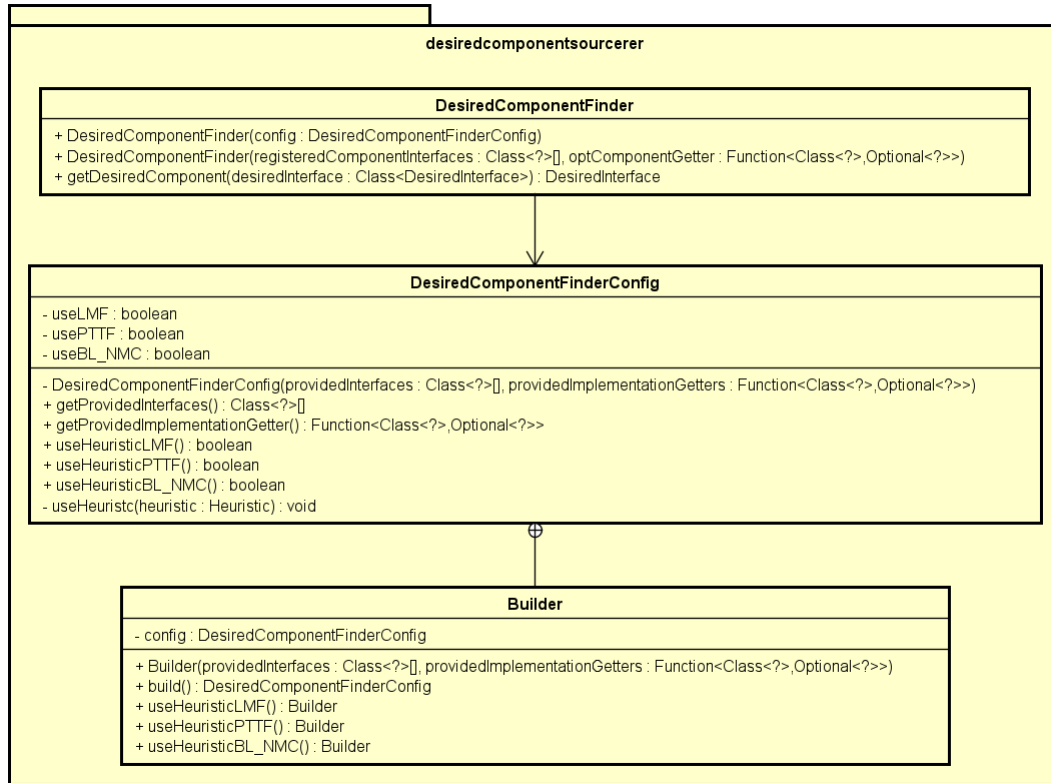


Abbildung 4.8: Modul: DesiredComponentSourcerer

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

Hierzu wird ein Objekt vom `StructuralTypeMatcher` aus dem *SignatureMatching*-Modul ver-

wendet⁵ und versucht die *provided Interfaces* mit dem *desiredInterface* zu matchen (siehe Abbildung ??).

Nach der syntaktischen Evaluation, wird gemäß Abschnitt ?? die semantische Evaluation durchgeführt. Dabei werden zuerst die Proxies aus den Kombinationen der gematchten *provided Interfaces*⁶ erzeugt, welche im Anschluss hinsichtlich der vordefinierten Tests zum *desired Interface* 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.

⁶Diese Kombinationen sind mit den Elementen der Mengen aus $cover(R, L)$ gleichzusetzen.

Kapitel 5

Evaluierung

Die Evaluierung erfolgt innerhalb von Systemen, in denen mindestens 889 angebotene Interfaces existieren. Es wird zwischen einem Test-System und einem Heiß-System unterschieden.

Das Test-System wurde vorrangig für die Evaluation der Type-Matcher Rating basierten Heuristiken verwendet, da für diese Heuristiken keine Implementierungen der angebotenen Interfaces vorliegen müssen.

Das Heiß-System wurde vorrangig für die Evaluation der testergebnis basierten Heuristiken verwendet, da hier zu jedem der 889 angebotenen Interfaces eine Implementierung existiert. Die angebotenen Komponenten wurden im Heiß-System als Java Enterprise Beans umgesetzt.

Darstellung der Evaluationsergebnisse

Die Evaluationsergebnisse werden in der Form von Vier-Felder-Tafeln dargestellt (Beispiel siehe Tabelle 5.1). Für jedes erwartete Interface wird eine Vier-Felder-Tafel für jeden Durchlauf des Explorationsalgorithmus aufgezeigt. Aus der jeweiligen Tafel geht hervor, wie viele Kombinationen von Methoden-Konvertierungsvarianten aus den Kombinationen der ermittelten Typ-Konvertierungsvarianten innerhalb des Durchlaufs erzeugt werden könnten. Die Nummer des Durchlaufs wird in der oberen rechten Ecke der Tafel abgebildet. In der Spalte positiv ist die Anzahl der Kombinationen von Methoden-Konvertierungsvarianten verzeichnet, die innerhalb des Durchlaufs tatsächlich erzeugt wurden. Die Zahl in der Spalte “negativ” drückt hingegen

aus, wie viele der Kombinationen aufgrund bestimmter Kriterien (bzw. Heuristiken) gar nicht erst erzeugt wurden. Die Zeile “falsch” beschreibt die Anzahl der relevanten Kombinationen, aus denen benötigte Komponenten erzeugt werden, welche die semantischen Tests nicht bestehen. Dementsprechend stellt die Zeile “richtig” die Anzahl der Kombinationen dar, aus denen sich benötigte Komponenten erzeugen lassen, welche die semantischen Test bestehen. Der Fall, in dem eine Kombination nicht erzeugt wurde, aber dennoch für die Erstellung einer benötigten Komponente genutzt wurde und die semantischen Tests besteht, (negativ und richtig) kann nicht auftreten.

Für die Anzahl der zu kombinierenden Methoden-Konvertierungsvarianten MK wird der höchste mögliche Wert angenommen. Dieser ist von der Anzahl der angebotenen Methoden am sowie der Anzahl der erwarteten Methoden em abhängig und wird wie folgt berechnet:

$$MK = \frac{am!}{(am - em)! * em!}$$

Die Anzahl der angebotenen Methoden am ist wiederum abhängig von den angebotenen Interfaces deren Typ-Konvertierungsvarianten im jeweiligen Durchlauf miteinander kombiniert wurden. Die Anzahl der Kombinationen von Typ-Konvertierungsvarianten innerhalb des Durchlaufs sei mit TK beschrieben. Der Wert für TK berechnet sich in Abhängigkeit von der Nummer des Durchlaufs d und der Anzahl der strukturell passenden angebotenen Interfaces n (siehe auch Abschnitt Explorationskomponente, 2. Stufe, 2. Kombination von Typ-Konvertierungsvarianten).

$$TK = \frac{n!}{(n - d)! * d!}$$

Da die Anzahl der angebotenen Methoden von System zu System schwanken kann, sei die Funktion $am(TK)$ eine näherungsweise Darstellung von am , in Abhängigkeit von der Anzahl der kombinierten Typ-Konvertierungsvarianten TK .

Da durch die Heuristiken letztendlich Methoden-Konvertierungsvarianten aus der Suche herausfallen, wird die Anzahl der entsprechenden Methoden-Konvertierungsvarianten in dem jeweiligen Feld der Vier-Felder-Tafeln als Funktion $mk(TK)$ dargestellt, die wie folgt definiert

wird:

$$mk(TK) = \frac{am(TK)!}{(am(TK) - em)! * em!}$$

Tabelle 5.1 zeigt ein Beispiel für eine solche Vier-Felder-Tafel, in der die Ergebnisse des 1. Durchlauf des Explorationsalgorithmus dargestellt sind. Dabei wurden Methoden-Konvertierungsvarianten aus 10 Kombinationen von Typ-Konvertierungsvarianten erzeugt. Den Methoden-Konvertierungsvarianten, die nicht beachtet wurden, lagen insgesamt 20 Typ-Konvertierungsvarianten zugrunde. Weiterhin zeigt das Beispiel, dass es eine Kombination von Methoden-Konvertierungsvarianten gibt, aus der eine passende benötigte Komponente erzeugt werden konnte.

1	positiv	negativ
falsch	$mk(10)$	$mk(20)$
richtig	1	0

Tabelle 5.1: Beispiel: Vier-Felder-Tafel

Kapitel 6

Diskussion

Kapitel 7

Ausblick

Kapitel 8

Schlussbemerkung

Literaturverzeichnis

- [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. Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 681–682, New York, NY, USA, 2006. Association for Computing Machinery.
- [Hum08] HUMMEL, OLIVER: *Semantic Component Retrieval in Software Engineering*. , April 2008.
- [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].
- [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.
- [Tac21] TEAM, OBJENESIS ALL CONTRIBUTORS: *Objenesis Release notes*. <http://objenesis.org/notes.html>, 2021. [Online; letzter Zugriff 26.06.2021].

- [Tre15] TREMBLAY, HENRI: *easymock-3.0*. <https://github.com/easymock/easymock/releases/tag/easymock-3.0>, 2015. [Online; letzter Zugriff 26.06.2021].

Anhang A

Semantische Evaluation mit allen vorgestellten Heuristiken

Die in den Abschnitten 3.4.1 - ?? vorgestellten Heuristiken können miteinander Kombiniert werden. Listing A.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 PTTF
17            if( passedTests > 0 ){
18                priorityTargets.addAll( proxy.targets )
19                optimizedProxies = PTTF( optimizedProxies )
20            }
21            // Heuristik BL_FFMD und BL_FSMT
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 A.1: Kombination aller Heuristiken