FernUniversitt Hagen

Masterarbeit

Evaluation von Heuristiken fr 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

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

In greren Software-Systemen ist es blich, dass mehrere Komponenten miteinander ber Schnittstellen kommunizieren. In der Regel werden diese Schnittstellen so konzipiert, dass sie Informationen oder Services anbieten, die von anderen Komponenten abgefragt und benutzt werden knnen. 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).

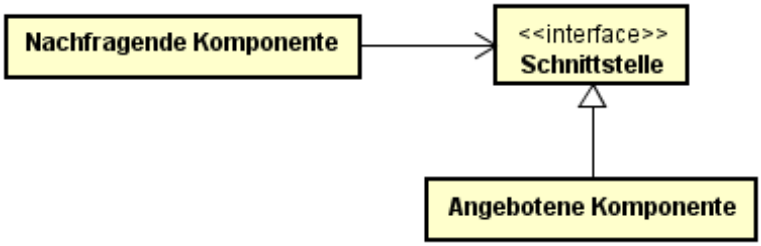

Abbildung 1.1: Abhngigkeiten von nachfragenden und angebotenen Komponenten

Wird von einer nachfragenden Komponente eine Information bentigt, die in dieser Form noch nicht angeboten wird, so wird hufig ein neues Interface fr diese bentigte 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 zustzlich das neue Interface deklariert werden. Zudem bedingt eine nachtrgliche nderung der neuen Schnittstelle ebenfalls eine Anpassung der drei genannten Artefakte.

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

Weiterhin ist es denkbar, dass ein Software-System auf unterschiedlichen Maschinen verteilt wurde und dadurch Teile des Systems ausfallen knnen. 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 bentigten Informationen und Services vielleicht durch andere Schnittstellen, deren Implementierung durchaus zur Verfgung stehen, bereitgestellt werden knnten.

Dies fhrt zu der berlegung, ob es nicht mglich 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 [**?**] von Bajaracharya et al. verfolgt. Diese Gruppe entwickelte eine Search Engine namens Sourcerer, welche Suche von Open Source Code im Internet ermglichte. Darauf aufbauend wurde von derselben Gruppe in [**?**] ein Tool namens CodeGenie entwickelt, welches einem Softwareentwickler die Code Suche ber ein Eclipse-Plugin ermglicht. In diesem Zusammenhang wurde erstmals der Begriff der Test-Driven Code Search (TDCS) etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummel [**?**] ebenfalls eine Weiterentwicklung von Sourcerer verffentlicht, welche unter dem Namen Merobase bekannt ist, welches ebenfalls das Konzept der TDCS verfolgt. TDCS beruht grundlegend darauf, dass der Entwickler Testflle spezifiziert, die im Anschluss verwendet werden, um relevanten Source Code aus einem Repository hinsichtlich dieser Testflle zu evaluieren. Damit kann das jeweilige Tool dem Entwickler Vorschlge fr die Wiederverwendung bestehenden Codes unterbreiten.

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

Kapitel 3

Theoretische Grundlagen

3.1 Strukturelle Evaluation

3.1.1 Struktur fr die Definition von Typen

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

Regel	Erluterung
$L ::= TD*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \vert RD$	Eine Typdefinition kann entweder die Definition eines provided Typen (PD) oder eines required Typen (RD) sein.
$PD ::=$ `provided` T `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.
$RD ::=$ `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 Rckgabe-Typs T'.

Tabelle 3.1: Struktur fr die Definition einer Bibliothek von Typen

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

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

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

Darber hinaus seien folgende Funktionen definiert:

$$\mathit{felder}(T) := \left\{\ T\ f \;\middle|\; T\ f \text{ ist Felddeklaration von } T\ \right\}$$

$$\mathit{methoden}(T) := \left\{\ T''\ m(T') \;\middle|\; T''\ m(T') \text{ ist Methodendeklaration von } T\ \right\}$$

$$\mathit{feldTyp}(f, T) := \ T' \;\middle|\; T'\ f \text{ ist Felddeklatation von } T$$

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 dazugehrige Matchingrelation $\Rightarrow_{exact}$ wird durch folgende Regel beschrieben:

$$\frac{}{T \Rightarrow_{exact} T}$$

GenTypeMatcher

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

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

SpecTypeMatcher Der *SpecTypeMatcher* stellt im Verhltnis zum *GenTypeMatcher* das Matching in die entgegengesetzte Richtung dar. Die dazugehrige Matchingrelation $\Rightarrow_{spec}$ wird durch folgende Regel beschrieben:

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

Die oben genannten Matchingrelationen werden fr 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 enthlt, auf dessen Typ T'' der Typ T ber die Matchingrelation $\Rightarrow_{internCont}$ gematcht werden kann. So kann bspw. der Typ `boolean` aus Listing 1 auf den Typ `FireState` gematcht werden.

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

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

So wrde fr die Typen `boolean` und `FireState` gelten:

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

ContainerTypeMatcher

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

Die dazugehrige 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 fr die Typen `FireState` und `boolean`:

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

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 ber 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:

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

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

Die Matchingrelation fr 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 fr einen required Typ durchgefhrt. 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 fr die dies in Bezug auf ein required Typ R zutrifft, wird ber die Funktion *cover* beschrieben.

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

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()
}
```

ber die Funktion *cover* werden folgenden Mengen von Target-Typen fr die Bildung von Proxies fr den required Typ `Greeting` ermittelt.

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

3.2 Generierung der Proxies auf Basis von Matchern

Ein Proxy wird in Abhngigkeit 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 fr die Definition von Proxies

Die Konvertierung eines Typs T aus einer Menge von provided Typen P wird durch *Proxies* beschrieben. Die Grammatikregeln fr einen Proxies sind Tabelle 3.2 zu entnehmen.

Regel	Erluterung
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{MDEL_1, ..., MDEL_k\}$	Ein Proxy wird fr ein Typ T als Source-Typ mit einer Mengen von provided Typen $P = \{P_1, ..., P_n\}$ als Target-Typen, einer Menge von Methoden-Delegationen erzeugt.
$MDEL$ ::= $CALLM \rightarrow DELM$	Eine *Methodendelegation* besteht aus einer *aufgerufenen Methode* und aus einem *Delegationsziel*.
$CALLM$ ::= $REF.m(CP_1, ..., CP_n) : CR$	Eine aufgerufene Methode besteht aus dem Namen der Methode m, dem Rckgabetyp CR und einer Menge von Parametertypen $\{CP_1, ..., CP_n\}$.
$DELM$ ::= $REF.n(DP_1, ..., DP_n) : DR$	Die erste Variante eines Delegationsziels besteht aus dem Namen der *Delegationsmethode* n, dem Rckgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `posModi(`$I_1, ..., I_n$`)` $REF.n(DP_1, ..., DP_n) : DR$	Die zweite Variante eines Delegationsziels besteht aus einer Menge von Indizies $\{I_1, ..., I_n\}$, einer *Referenz*, dem Namen der Delegationsmethode n, dem Rckgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `err`	Die dritte Variante eines Delegationsziels enthlt keine weiteren Bestandteile. Das Terminal `err` weist darauf hin, dass die Delegation innerhalb des Proxies nicht mglich ist und zu einem Fehler fhrt.
$REF ::= P_i$	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 Erluterungen fr die Definition eines Proxies

Es handelt sich dabei um Produktionsregeln einer Attributgrammatik. Die dazugehrigen Attribute sind der Tabelle 3.3 zu entnehmen. Dazu sei zustzlich 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 aufgefhrt ist, wird mit dem Wert *none* belegt. Ein Proxy bietet alle Methoden des Source-Typen an.

Regel	**Attribute**
$PROXY$ ::= `proxy for` T `with` $[P_1, ..., P_n]$ $\{MDEL_1, ..., MDEL_k\}$	`type` $= T$ `targets` $= [P_1, ..., P_n]$ `dels` $= [MDEL_1.*, ..., MDEL_k.*]$
$MDEL$ ::= $CALLM \rightarrow DELM$	`call` $= CALLM.*$ `del` $= DELM.*$

$CALLM$ ::=
$REF.m(CP_1, ..., CP_n) : CR$
	`source` $= REF$.`mainType`
	`delType` $= REF$.`delType`
	`name` $= m$
	`paramTypes` $= [CP_1, ..., CP_n]$
	`returnType` $= CR$
	`field` $= REF$`.field`
	`paramCount` $= n$

$DELM$ ::=
$REF.n(DP_1, ..., DP_n) : DR$
	`target` $= REF$.`mainType`
	`delType` $= REF$.`delType`
	`posModi` $= [0, ..., n-1]$
	`name` $= n$
	`paramTypes` $= [DP_1, ..., DP_n]$
	`returnType` $= DR$
	`field` $= REF$`.field`

$DELM$ ::= `posModi`$(I_1, ..., I_n)$
$REF.n(DP_1, ..., DP_n) : DR$
	`target` $= REF$.`mainType`
	`delType` $= REF$.`delType`
	`posModi` $= [I_1, ..., I_n]$
	`name` $= n$
	`paramTypes` $= [DP_1, ..., DP_n]$
	`returnType` $= DR$
	`field` $= REF$`.field`

$DELM$ ::= `err`	
REF ::= P	`mainType` $= P$ `field` $=$ `self` `delType` $= P$
REF ::= $P.f$	`mainType` $= P$ `field` $= f$ `delType` $= feldTyp(f, P)$

Tabelle 3.3: Grammatikregeln mit Attributen fr die Definition eines Proxies

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 Nontermial *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. Parameterbergabe
   Dabei werden die Parameter, mit denen die vom Proxy angebotene Methode, aufgerufen wird, an die Delegationsmethode des Delegationsziels bergeben. Dabei sind zwei Dinge zu beachten. Zum Einen mssen die Typen der bergebenen Parameter zu den Typen der von der Delegationsmethode erwarteten Parameter passen. Zum Anderen muss die Reihenfolge, in der die Parameter bergeben wurden, an die erwartete Reihenfolge der Delegationsmethode angepasst werden.

2. Ausfhrung
   Dieser Schritt meint die Durchfhrung der Delegationsmethode mit den bergeben Parametern aus Schritt 1. Dies schliet auch die Ermittlung des Rckgabewertes der Delegationsmethode ein.

3. bergabe des Rckgabewertes
   hnlich wie bei der Parameterbergabe, muss auch der Rckgabewert, der bei der Ausfhrung in Schritt 2 ermittelt wurde, an die aufgerufenen Methode, die vom Proxy angeboten wird, bergeben werden. Hier muss ebenfalls sichergestellt werden, dass die beiden Rckgabetypen der beiden Methoden zueinander passen.

Die Delegation aus dem oben genannten Beispiel kann schematisch wie in Abbildung 3.1 dargestellt werden. Die bergabe der Parameter- und Rckgabewerte wird durch die gestrichelten Pfeile symbolisiert. An diesem Beispiel sind sowohl die Parameter- als auch die Rckgabe-Typen

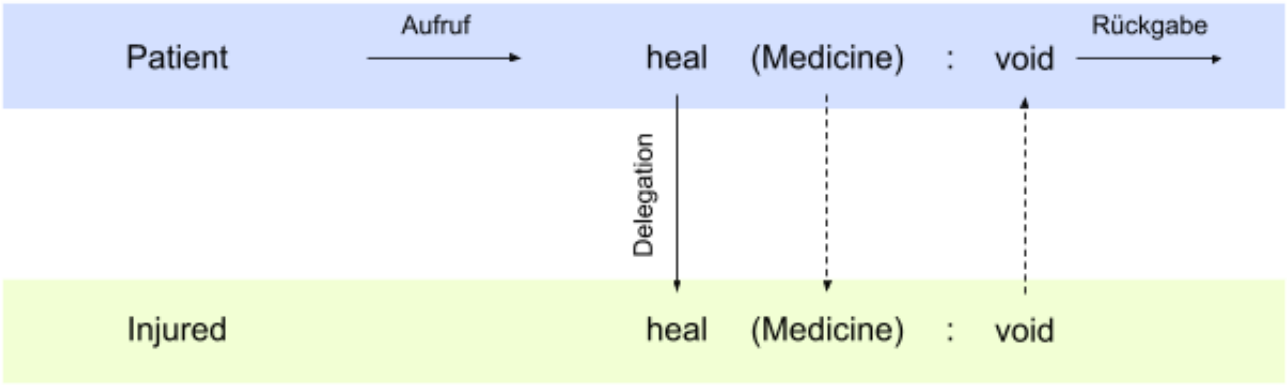

Abbildung 3.1: Delegation der Methode `heal`

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 bergabe der Parameter- und Rckgabewerte 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 fr 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 bergeben 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 ursprnglichen Aufruf zu variieren, wird das Schlsselwort `posModi` verwendet. Dort werden eine Reihe von Indizes angegeben. Die Anzahl der angegebenen Indizes muss mit der Anzahl der Parameter bereinstimmen. 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 bergeben. Dementsprechende wird er zweite Parameter der aufgerufenen Methoden (Index = 1) der Delegationsmethode als erster Parameter bergeben (siehe Abbildung 3.2).

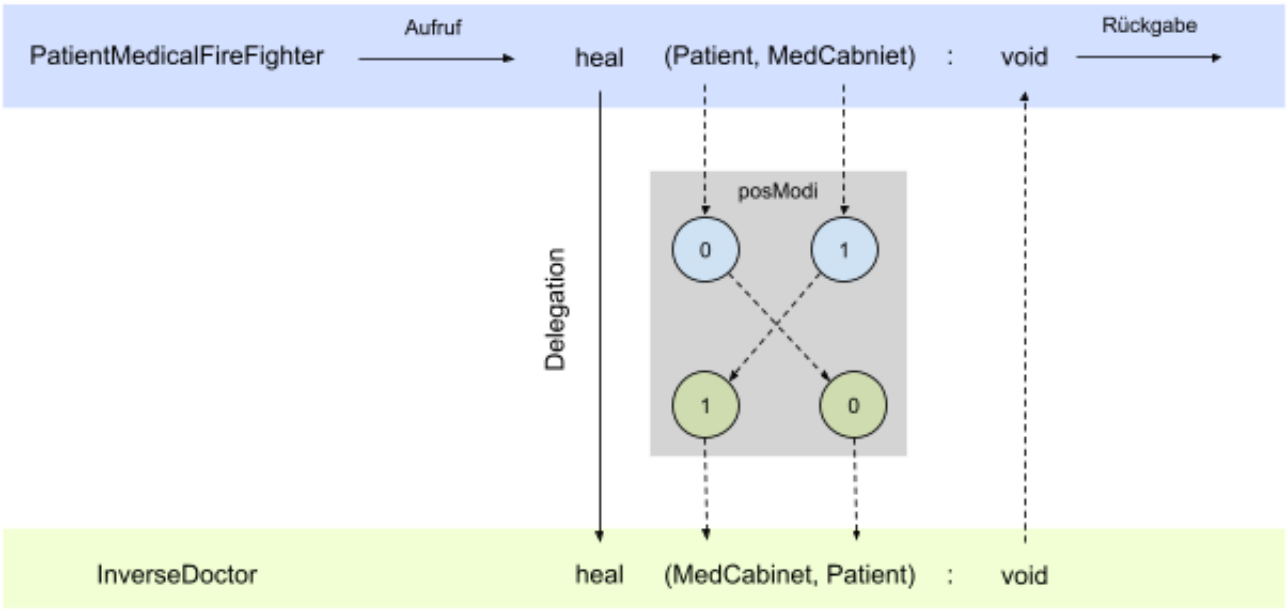

Abbildung 3.2: Delegation der Methode `heal` mit Parametern in unterschiedlicher Reihenfolge

Ein weiteres Beispiel soll zeigen, wie mit bergebenen Typen umzugehen ist, die nicht ohne Probleme bergeben werden knnen. Dafr ist jedoch vorab zu klren, wann dies der Fall ist.

Dass identische Typen keine Probleme bei der bergabe zwischen aufgerufener Methode und Delegationsmethode darstellen, wurde in den oben genannten Beispielen gezeigt.

Darber hinaus knnen Typen aber auch dann ohne Probleme bergeben 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 aufgerhrt, bei der sowohl die Parameter- als auch die Rckgabe-Typen der aufgerufenen Methode und der Delegationsmethode nicht auf Basis des Substitionsprinzips bergeben werden knnen.

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

Listing 3.4: Methoden-Delegation mit Typkonvertierung

In einem solchen Fall mssen die Parameter-Typen der aufgerufenen Methoden in die Parameter-Typen der Delegationsmethode konvertiert werden. Analog dazu muss der Rckgabetyp der Delegationsmethode in den Rckgabetyp 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 mssten bezogen auf die Methoden-Delegation aus Listing 4 fr die Parameter-Typen einer der Proxies aus der Menge *proxies*(`Fire`, {`ExtFire`}) an die Delegationsmethode bergeben werden. Nach der Ausfhrung der Delegationsmethode mssste ein Proxy aus der Menge *proxies*(`boolean`, {`FireState`}) an die aufgerufenen Methode als Rckgabetyp bergeben werden. Der Sachverhalt wird in Abbildung 3.3 schematisch dargestellt.

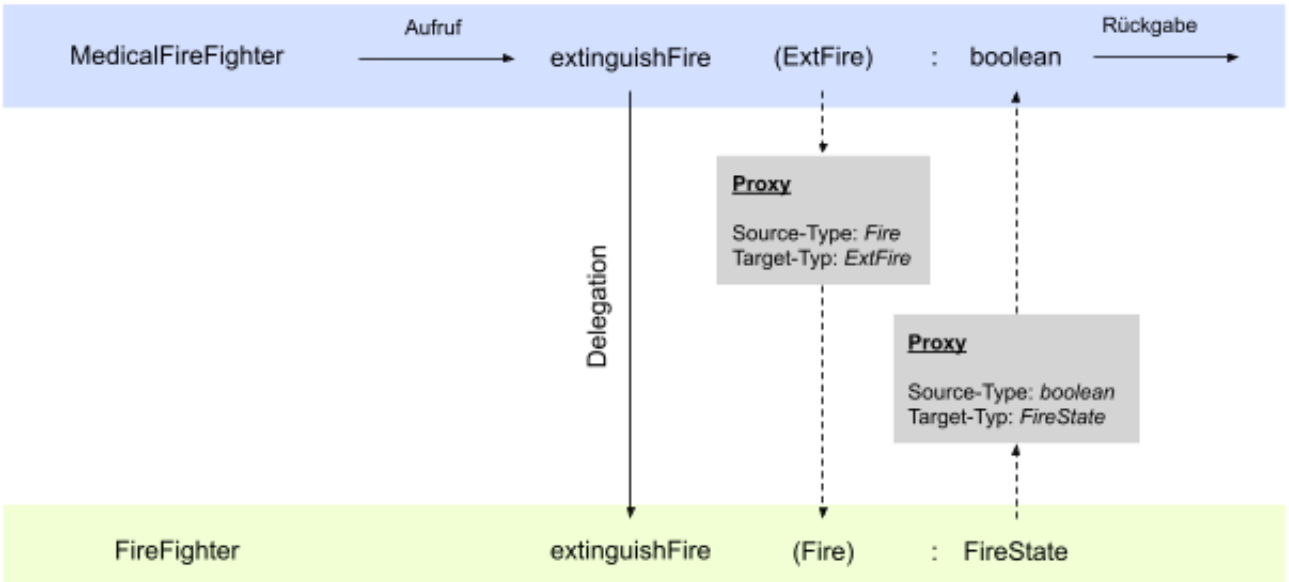

Abbildung 3.3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

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

3.2.2 Generierung von Proxies

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

In Abhngigkeit von dem Matching zwischen dem Source-Typ und den Target-Typen werden unterschiedliche Arten von Proxies generiert. Fr 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 fr die einzelnen Proxy-Arten beschrieben. Dabei ist davon auszugehen, dass die Proxies eine allgemeine Struktur haben, die in Abschnitt 3.2.1 aufgefhrt ist. Um die Regeln fr die Generierung der Proxies zu beschreiben, soll davon ausgegangen werden, dass jedes Listen-Attribut ($NT.*$) aus Tabelle 3.3 ein Attribut `len` enthlt in dem die Anzahl der in der Liste befindlichen Elemente abgelegt ist.

Sub-Proxy

Die Voraussetzung fr die Erzeugung eines *Sub-Proxies* vom Typ T aus einem Target-Typ T' ist $T \Rightarrow_{spec} T'$. Damit ist der *SpecTypeMatcher* der Basis-Matcher fr 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* fr diese Konstellation erzeugt werden. Der resultierende *Sub-Proxy* ist im folgenden Listing aufgefhrt.

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

Listing 3.5: Sub-Proxy fr Patient

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

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

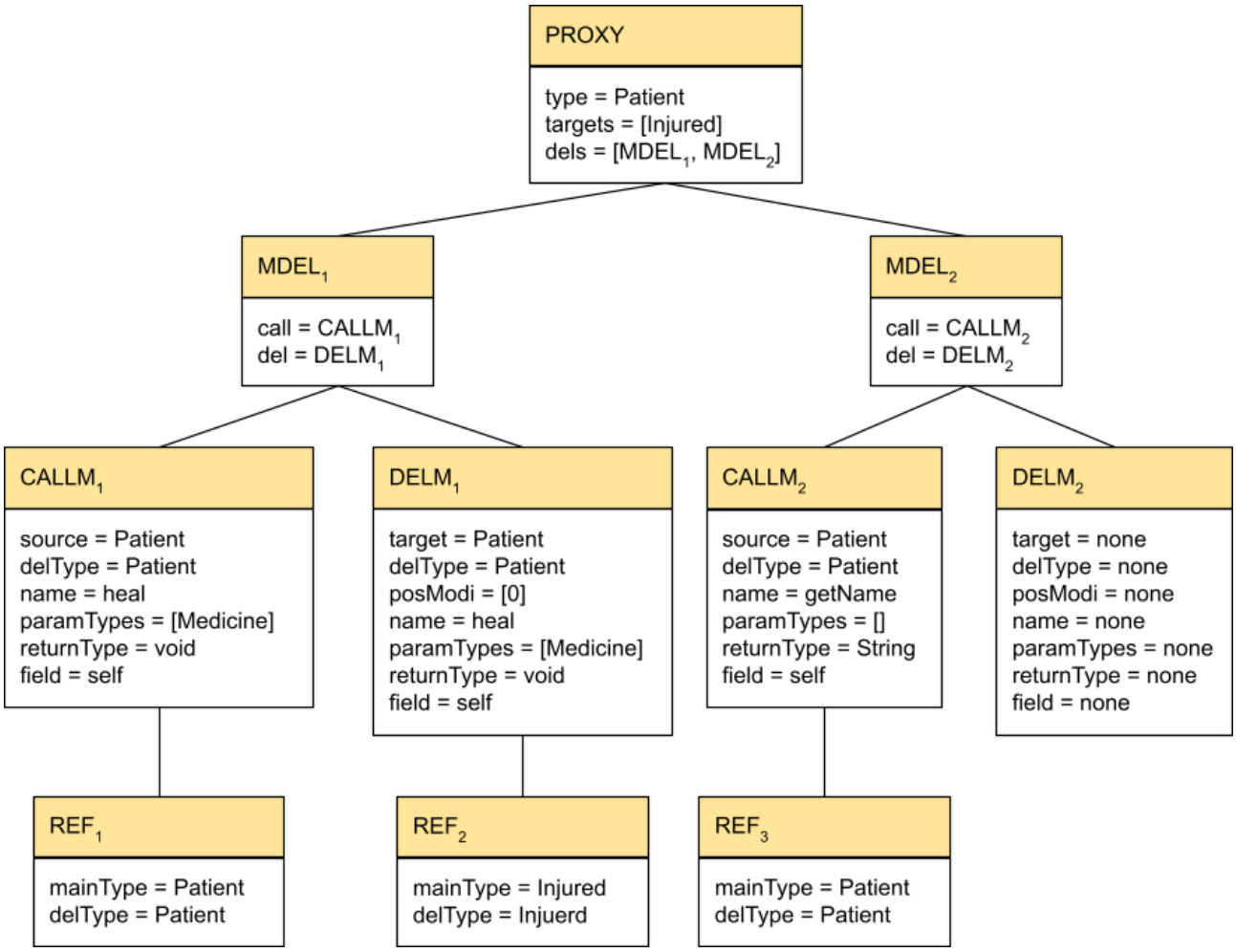

Abbildung 3.4: AST fr das Beispiel zum Sub-Proxy

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 fhrt zu einem Fehler, weil keine Delegationsmethode zur Verfgung steht.

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

[1] Es wurden nur die Nonterminale mit den dazugehrigen Attributen aufgefhrt.

Cast) ebenfalls dazu fhrt, dass bestimmte Methoden, wie in diesem Fall `getName` nicht ausgefhrt werden knnen, 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* enthlt genau einen Target-Typ. Fr einen Proxy P wird dieser Sachverhalt durch die folgende Regel dargestellt.

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

Darber hinaus enthlt 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 bereinstimmen. Folgende Regel stellt diesen Sachverhalt fr eine Menge von Methoden-Delegationen *MDList* dar.

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

Fr 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`.

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

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

Zustzlich 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 gewhrleistet, dass die Attribute `mainType` und `delType` im Attribut `call` bereinstimmen. (siehe Tabelle 3.3)

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

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

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

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

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

Analog dazu knnen auch die Regeln fr 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 darber hinaus noch die Methoden-Delegation zu beachten, die bei einem Aufruf zu einem Fehler fhrt. Dieser Fall wird fr eine Methoden-Delegation *MD* wie folgt beschrieben:

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

Die genannten Regeln fr eine Methoden-Delegation *MD* in einem *Sub-Proxy* lassen sich ber 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 fr jede Methode m des Source-Typ genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode. Damit lsst sich fr einen Proxy P in Bezug auf alle seine Methoden-Delegationen folgende Regeln formulieren:

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

Fr 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.

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

Content-Proxy

Die Voraussetzung fr 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 fr 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* fr diese Konstellation erzeugt werden. Ein resultierender *Content-Proxy* ist in folgendem Listing aufgefhrt.

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

Listing 3.6: Content-Proxy fr 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 dazugehrigen 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* enthlt, 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 drfen 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 fr das Attribut `delTyp` und den Source-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont} MD.del.delTyp$ gelten. Daher gilt fr den *Content-Proxy* die folgende Regel:

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

[2] Es wurden nur die Nonterminale mit den dazugehrigen Attributen aufgefhrt.

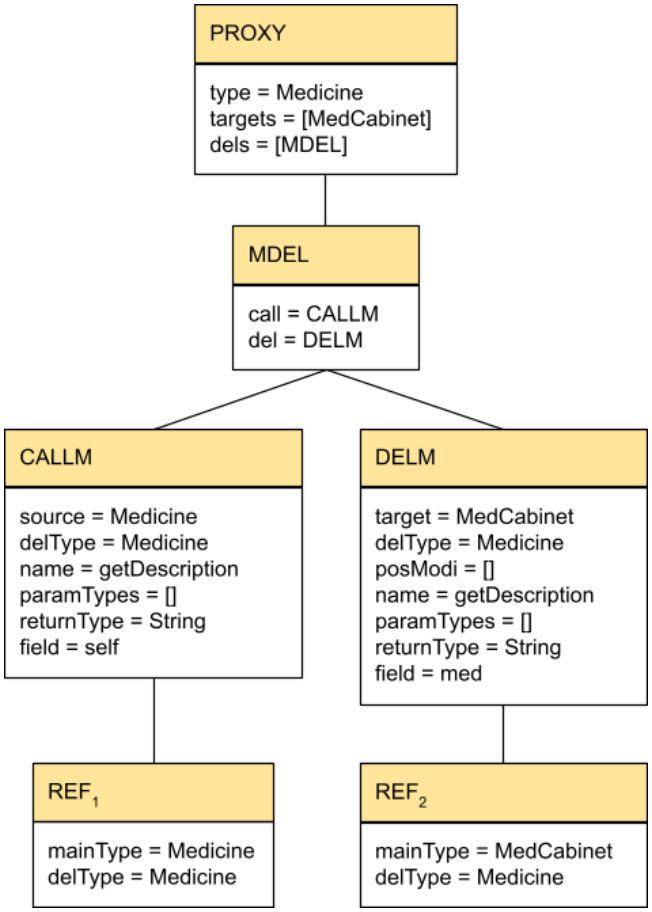

Abbildung 3.5: AST fr das Beispiel zum Content-Proxy

Damit kann eine zusammenfassende Regel fr 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 fr 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* fr jede Methode m des Source-Typen genau eine Methoden-Delegation mit der Methode m als aufgerufene Methode. Daraus ergibt sich fr alle Methoden-Delegationen aus einem *Content-Proxy* P folgende Regel:

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

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

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

Container-Proxy

Die Voraussetzung fr die Erzeugung eines *Container-Proxies* vom Typ T aus einem Target-Typ T' ist $T \Rightarrow_{container} T'$. Damit ist der *ContainerTypeMatcher* der Basis-Matcher fr den *Container-Proxy*.

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

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

Listing 3.7: Container-Proxy fr 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 dazugehrigen Attributen ist Abbildung 3.6 zu entnehmen.

3

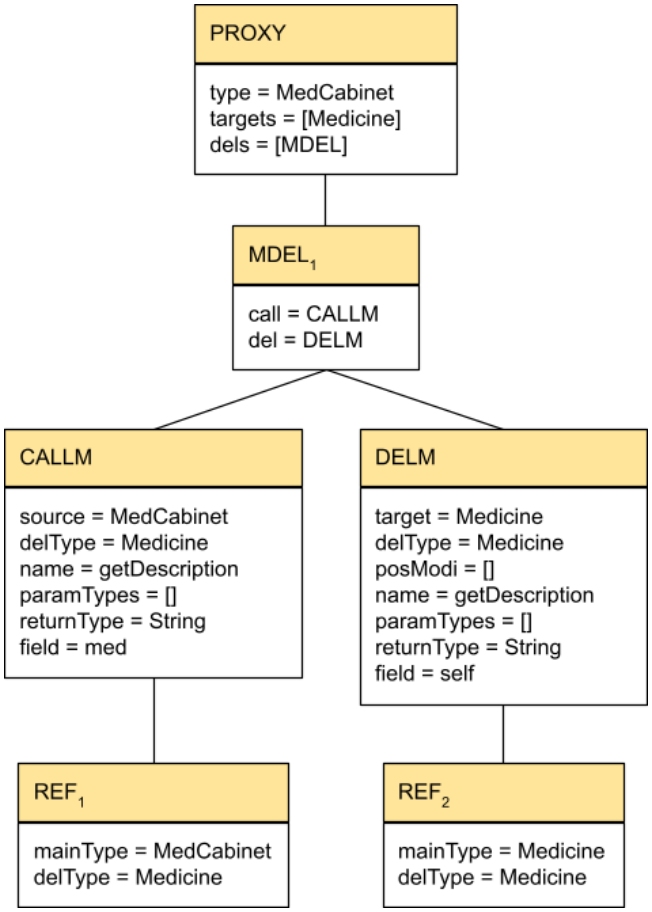

Abbildung 3.6: AST fr das Beispiel zum Container-Proxy

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

Ein *Container-Proxy* enthlt, wie die vorher beschriebenen Proxies, genau einen Target-Typ. Die Eigenschaften der Delegationsmethoden innerhalb der einzelnen Methoden-Delegationen gleichen denen aus dem *Sub-Proxy*.

In den angerufenen Methoden einer einzelnen Methoden-Delegation *MD* drfen die Attribute

[3]Es wurden nur die Nonterminale mit den dazugehrigen Attributen aufgefhrt.

`mainType` und `delType` im *Container-Proxy* nicht bereinstimmen. Dementsprechend darf das Attribut `field` nicht mit dem Wert `self` belegt sein. Vielmehr mssen der Wert des Attributs `delTyp` und der Target-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont}$ `delTyp` ausweisen. Daher gilt fr den *Container-Proxy* P folgende Regel.

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

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

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

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

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

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

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

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

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

Struktureller Proxy

Die Voraussetzung fr 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 fr 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 aufgefhrt.

```
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 fr MedicalFireFighter

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

Der abstrakte Syntaxbaum mit den dazugehrigen Attributen ist Abbildung 3.7 zu entnehmen. [4]

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

[4] Es wurden nur die Nonterminale mit den dazugehrigen Attributen aufgefhrt.

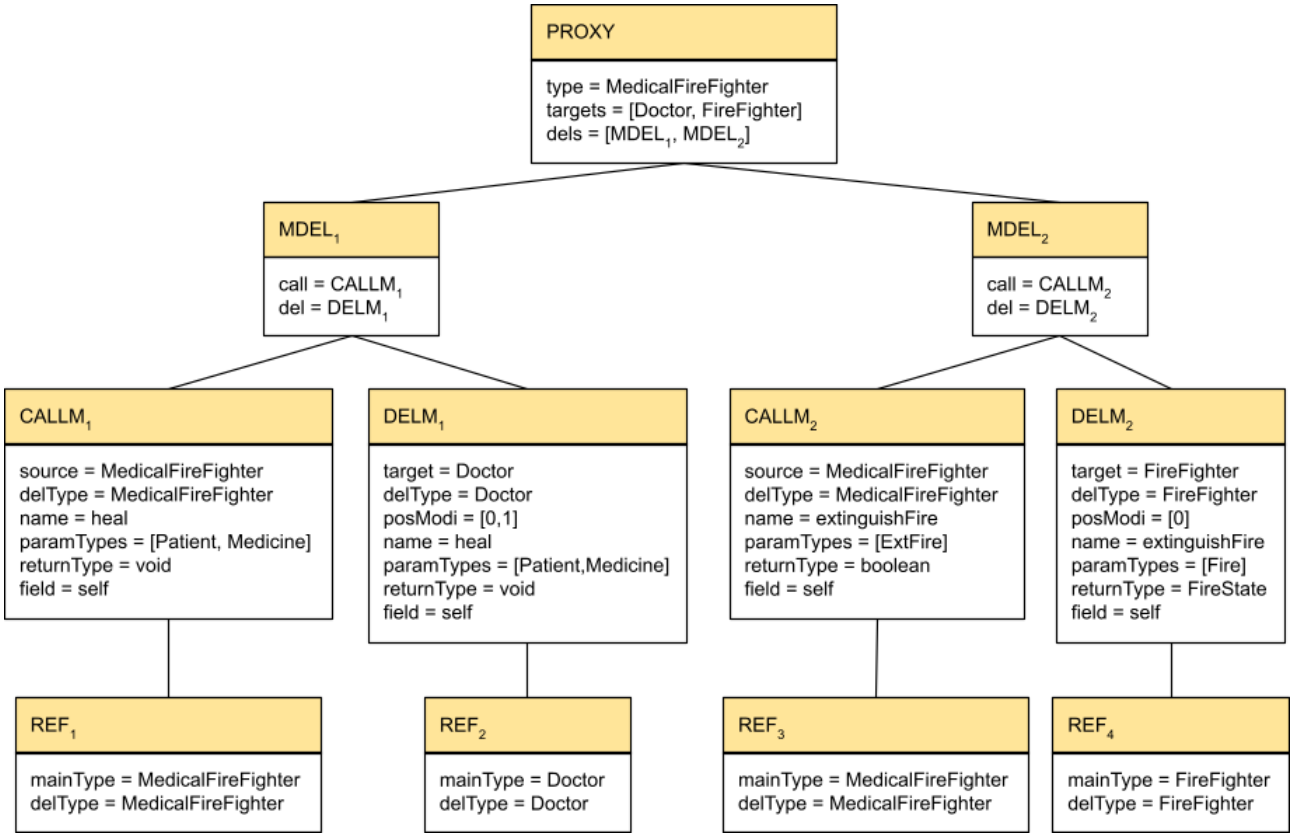

Abbildung 3.7: AST fr das Beispiel zum strukturellen Proxy

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

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

Fr die aufgerufene Methode und die Delegationsmethode einer einzelnen Methoden-Delegation M gelten im *strukturellen Proxy* dieselben Regeln wie fr den *Sub-Proxy*. Die Namen der aufgerufenen Methode und der Delegationsmethode mssen dabei jedoch nicht bereinstimmen. Dafr mssen diese beiden Methode jedoch ein strukturelles Matching aufweisen. Bezogen auf die Rckgabe-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 fr die Parameter-Typen gelten:

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

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

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

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

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

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

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.

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

Allgemeine Generierung von Proxies Die Proxy-Funktion der einzelnen Proxy-Arten werden zur Beschreibung einer allgemeine Funktion fr die Generierung der Proxies verwendet. Dazu sind die Proxy-Arten zusammen mit den dazugehrigen Matchingrelationen und Proxy-Fukntionen in Tabelle 3.4 noch einmal aufgefhrt.

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 erwhnte Funktion $proxies(S, T)$ kann darauf aufbauend fr 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 mglicher Proxies innerhalb einer Bibliothek

Die Generierung der Proxies fr ein required Typ R aus der Bibliothek L erfolgt whrend der Exploration mit den Mengen von provided Typen aus $cover(R, L)$ (siehe Abschnitt 3.1.3). Mit einer Menge $T \in cover(R, L)$ knnen 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 bereinstimmen. Die Anzahl der mglichen Proxies fr ein required Typ R mit einer bestimmten Mengen von Target-Typen $T_1, ..., T_k$ ist somit von der Anzahl der Methoden abhngig, die in einem der Target-Typen des Proxies deklariert wurden und strukturell mit den Methoden aus R bereinstimmen.

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

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

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

$$M_1 = structM_{target}(m_1, T)$$
$$...$$
$$M_n = structM_{target}(m_n, T)$$

Fr jede Kombination von jeweils einem Element aus jeder der Mengen $M_1, ...M_n$ kann ein Proxy fr R mit der Menge der Target-Typen T erzeugt werden.

Beispiel 2 Aufbauend auf dem vorherigen Beispiel 1 ergeben sich fr die Menge der Target-Typen {`Leave`, `Come`} und die beiden Methoden des required Typs `Greeting` folgende Menge

von bereinstimmenden Methoden ber die Funktion $structM_{target}$:

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

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

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
}
```

Fr die Bildung eines Proxies wird aus jeder der oben genannten Menge $M_1, ..., M_n$ genau ein Element als Delegationsmethode verwendet werden. Die Anzahl aller mglichen Proxies fr ein required Typ R aus einer Menge von Target-Typen T und unter der Annahme, dass $\{m_1, ..., m_n\} = methoden(R)$, sei ber die Funktion $proxyCount(R, T)$ ausgedrckt. Fr $proxyCount(R, T)$ ist zu beachten, dass es sich dabei lediglich um eine Annherung an die tatschliche Anzahl der Proxies handelt, die unter den oben beschriebenen Bedingungen erzeugt werden knnen. Dies liegt daran, dass eine Delegationsmethoden $dm \in M_1 \cup ... \cup M_n$ innerhalb eines Proxy maximal

einmal verwendet werden darf. Es ist jedoch mglich, dass es zwischen den oben genannten Mengen $M_1, ..., M_n$ berschneidungen gibt (siehe vorheriges Beispiel). Daher gelten fr die Funktion *proxyCount* folgende Regeln unter den oben genannten Modalitten:

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

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

Im Allgemeinen gilt demnach:

$$proxyCount(R,\, T) \leq \prod_{i=1}^{n} |structM_{target}(m_i,\, T)| \;\middle|\; \left\{ \begin{array}{l} m_1, \\ ..., \\ m_n \end{array} \right\} = methoden(R)$$

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

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

3.3 Semantische Evaluation

Das Ziel der semantischen Evaluation ist es, einen der Proxies, die aus den Mengen von Target-Typen, die im Rahmen der strukturellen Evaluation erzeugt werden knnen, hinsichtlich der vordefinierten Testflle zu evaluieren. Da die gesamte Exploration zur Laufzeit des Programms durchgefhrt wird, stellt sie hinsichtlich der nicht-funktionalen Anforderungen eine zeitkritische

Komponente dar.

Da die Anforderungen an die gesuchte Komponente mit bedacht spezifiziert werden mssen, ist es irrelevant, ob es mehrere Proxies gibt, die hinsichtlich der vordefinierten Testfllen positiv evaluatiert werden knnen. Es ist ausreichend lediglich ein Proxy zu finden, dessen Semantik zu positiven Ergebnissen hinsichtlich aller vordefinierten Testflle fhrt.

3.3.1 Besonderheiten der Testflle

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` zurckgibt. Weiterhin verfgt ein Test ber ein Attribut `triedMethodCalls`, in dem eine Liste von Methodennamen des Proxies, die bei der Durchfhrung der eval-Methode aufgerufen wurden, hinterlegt ist.

Die Implementierung der eval-Methode ist an folgende Bedingungen geknpft:

1. Vor dem Aufruf einer Methode auf dem als Parameter bergebenen Proxy-Objekt, wird der Name der dieser Methode in der Liste im Feld `triedMethodCalls` ergnzt.
2. Wenn der Proxy den Test erfllt, wird der Wert `true` zurckgegeben. Anderenfalls wird der Wert `false` zurckgegeben.

Beispiel 3 In folgendem Listing 3.9 ist eine eval-Methode aufgefhrt, die die oben genannten Bedingungen erfllt. Es sei davon auszugehen, dass der als Parameter bergebene Proxy eine Methode mit der Struktur *Integer add*(*Integerx*, *Integery*) anbietet. Der Fehlschlag (`err`) dieser Methode wird ber 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 fr 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 knnen, wurden im Abschnitt **??** ber $cover(R, L)$ beschrieben. Die in $T = cover(R, L)$ befindlichen Mengen knnen eine unterschiedliche Anzahl von Target-Typen enthalten. Die maximale Mchtigkeit 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 hnlich in dem gesamten System implementiert wurde. Aus diesem Grund kann fr die semantische Evaluation davon ausgegangen werden, dass die erfolgreiche Durchfhrung aller relevanten Tests umso wahrscheinlicher ist, je weniger Target-Typen im Proxy verwendet werden.

Sei folgende Funktion fr eine Menge von Target-Typen $T \in cover(R, L)$ und eine ganze Zahl $a > 0$ definiert:

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

Ausgehend von einer Bibliothek L kann der Algorithmus fr die semantische Evaluation der Proxies, die fr einen required Typ R mit den Mengen der Target-Typen $T = cover(R, L)$ erzeugt werden knnen, und der Menge von Tests (Parameter `tests`) wie folgt im Pseudo-Code

beschrieben werden. Die globale Variable `passedTests` enthlt dabei die Anzahl der fr den aktuell zu berprfenden Proxy erfolgreich durchgefhrten Tests. Auerdem 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 hngt mageblich von der Anzahl der Proxies ab, die fr den required Typ R in der Bibliothek L erzeugt werden knnen (siehe auch Abschnitt **??** Funktion *proxyCount*). Im schlimmsten Fall mssen 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 fr die Verwendung einer Heuristik notwendig, weitere Information whrend der semantischen Evaluation zu ermitteln und diese zu speichern. In den folgenden Abschnitten werden die Heuristiken und die dafr notwendigen Anpassungen an den jeweiligen Funktionen beschrieben.

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

3.4.1 Beachtung des Matcherratings (LMF)

Bei dieser Heuristik, welche den Namen*low matcherrating first* (kurz: LMF) trgt, 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 fr jede Matchingrelation bzw. fr jeden Matcher aus Abschnitt 3.1.2 ein Basisrating vergeben. Folgende

Funktion beschreibt das Basisrating fr 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 erwhnen, dass einige der o.g. Matcher ber dasselbe Basisrating erfgen. Das liegt daran, dass sie technisch jeweils gemeinsam umgesetzt wurden.[5]

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

Fr die Definition von Funktionen gelten im weiteren Verlauf folgende verkrzte Schreibweise in Bezug auf eine Methoden-Delegation MD:

$$pc := MD.call.paramCount$$
$$cRT := MD.call.returnType$$
$$dRT := MD.del.returnType$$
$$cPT := MD.call.paramTypes$$
$$dPT := MD.del.paramTypes$$
$$pos := MD.call.posModi$$

Darauf aufbauend sei die Menge der Matcherratings der Paare von Parameter- und Rckgabetypen aus der aufgerufenen Methode und den Delegationsmethode einer Methoden-Delegation

[5] Der *GenTypeMatcher* und der *SpecTypeMatcher* wurden gemeinsam in der Klasse `GenSpecTypeMatcher` umgesetzt. Der *ContentTypeMatcher* und der *ContainerTypeMatcher* wurden gemeinsam in der Klasse `WrappedTypeMatcher` umgesetzt. (siehe angehngter Quellcode)

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 ber die Funktion *mdRating* beschrieben. Fr die Definition der beiden Funktionen *rating* und *mdRating* gibt es unterschiedliche Mglichkeiten. In dieser Arbeit werden 4 Varianten als Definitionen vorgeschlagen, die in Kapitel 5 untersucht werden.

Dazu seien die folgenden Hilfsfunktionen definiert:

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

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

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

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

Variante 1: Durchschnitt

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

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

Variante 2: Maximum

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

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

Variante 3: Minimum

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

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

Variante 4: Durchschnitt aus Minimum und Maximum

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

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

Da die Funktion *rating* von *mdRating* abhngt und fr *mdRating* 4 Variante vorgeschlagen wurden, ergeben sich fr jede vorgeschlagene Variante fr die Definition von *rating* weitere 4 Varianten. Dadurch sind insgesamt 16 Varianten fr die Definition von *rating* gegeben.

Zur Anwendung der Heuristik muss das Matcherrating bei der Iteration ber die erzeugten Proxies beachtet werden. Dabei sollte die Liste der Proxies, ber 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. Fr 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 fr einen Proxy ermittelt wird, ist mageblich von den Methoden-Delegationen des Proxies abhngig. Jede Methoden-Delegation MD enthlt ein Typ in dem die Delegationsmethode spezifiziert ist. Dieser Typ befindet sich im Attribut $MD.del.delTyp$. Im Fall der sturkturellen Proxies, handelt es sich bei diesem Typ um einen der Target-Typen des Proxies.

Fr einen required Typ R aus einer Bibliothek L, kann ein Target-Typ T in den Mengen der mglichen Mengen von Target-Typen $cover(R, L)$ mehrmals auftreten. Die gilt insbesondere dann, wenn es in $cover(R, L)$ Mengen gibt, deren Mchtigkeit grer ist, als die Mchtigkeit 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 $TG \in cover(R, L)$.

Fr 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 durchgefhrt wird, sollte die Reihenfolge der zu prfenden Proxies so angepasst werden, dass die Proxies, die einen Target-Typen des Proxies P verwenden, im weiteren Verlauf zuerst geprft werden.

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

1. Die Target-Typen der Proxies, mit denen mind. ein Testfall erfolgreich durchgefhrt werden konnte, mssen in einer globalen Variable (`prioTargets`) hinterlegt werden.

2. Die Liste der Proxies, die der Methode `evalProxies` als Parameter bergeben 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 ber das Bubble-Sort-Verfahren in der Methode `PTTF`.

3. Die Liste der Proxies, ber die innerhalb der Methode `evalProxies` iteriert wird, kann bzgl. ihrer Reihenfolge bereits dann optimiert werden, wenn mind. einer der Testflle fr den aktuellen Proxy erfolgreich durchgefhrt wurde. Dazu mssen 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 Ergnzungen 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 whrend der laufenden semantischen Evaluation entstanden sind, fr den weiteren Verlauf ausgeschlossen werden. Dadurch soll die erneute Prfung eines Proxies, der ohnehin nicht zum gewnschten Ergebnis fhrt, verhindert werden.

Die Heuristik zielt darauf ab, Methoden-Delegationen, die immer fehlschlagen, zu identifizieren. Wurde eine solche Methoden-Delegation gefunden, knnen 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 knnen darauf aufbauend diejenigen Proxies entfernt werden, die eine jener Methoden-Delegationen enthalten. Dabei wird davon ausgegangen, dass die Methoden eines required Typen ber den Namen identifiziert werden knnen.

Das Fllen der globalen Variable `mdelBlacklist` erfolgt in der Methoden `evalProxy`. Die Identifikation der Methoden-Delegationen ber 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 Ausschlieen bestimmter Proxies erfolgt, indem Elemente aus einer Liste von Proxies entfernt werden. Listing 3.14 zeigt die dafr vorgesehene Methode `BL`, welche die Basis-Liste der Proxies im Parameter `proxies` und die Liste der Kombinationen von Methoden-Delegationen, die die Grundlage fr 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 fr Heuristil BL_NMC

Bei dieser Heuristik ist deren Anwendung nach jedem Evaluationsversuch eines einzelnen Proxies sinnvoll. Listing 3.15 zeigt die Anpassungen fr 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 Verfgung 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 fr 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 **??** ist die Architektur der Explorationskomponente aufgefhrt. Die das Modul *DesiredComponentSourcerer* stellt eine Schnittstelle nach Auen bereit, ber die die Explorationskomponenten in ein beliebiges Projekt eingebunden werden kann. Weiterhin ist das Modul *DesiredComponentSourcerer* von den Modulen *ComponentTester* und *SignatureMatching* abhngig, die selbst keine Abhngigkeiten zueinander haben.

Darber hinaus, werden folgende externe Bibliotheken verwendet:

- easymock 3.0 [**?**]
- cglib 3.3.0 [**?**]
- objenesis 3.1 [**?**]
- junit 4.13.0 [**?**]

Auf die konkrete Verwendung der externen Bibliotheken wird in den detaillierteren Beschreibungen der einzelnen Module in den folgenden Abschnitten eingegangen. Im Anschluss an die Beschreibung der Module wird auf die Nutzung der Schnittstelle zur Einbindung der Explorationskomponente in beliebige Java-Projekt eingegangen.

4.1 Modul: SignatureMatching2

In diesem Modul sind die Implementierungen der Matcher, die in Abschnitt 3.1.2 formal beschrieben wurden, untergebracht. So befinden sich, wie in dem Klassendiagramm in Abbildung 4.1 zu erkennen ist, mehrere Klassen, die das Interface *TypeMatcher* implementieren. Dieses Interface bietet die Methode `matchesType` an (siehe Abbildung **??**), ber die die jeweilige Matchingrelation eines formal definierten Matchers implementiert wird. Bei der Implementierung wurden einige der in Abschnitt 3.1.2 formal beschriebenen Matcher gemeinsam in einer Klasse umgesetzt. Die unten stehende Tabelle 4.1 zeigt die Zuordnung von Matchern zu den jeweiligen Klassen.

Matcher	**Implementierung (Klasse)**
ExactTypeMatcher	`ExactTypeMatcher`
GenTypeMatcher	`GenSpecTypeMatcher`
SpecTypeMatcher	`GenSpecTypeMatcher`
ContentTypeMatcher	`WrappedTypeMatcher`
ContainerTypeMatcher	`WrappedTypeMatcher`
StructuralTypeMatcher	`StructuralTypeMatcher`

Tabelle 4.1: Zuordnung der Matcher zu den Klassen, in denen sie implementiert sind

Neben der Methode `matchesType` bietet das Interface `TypeMatcher` noch die Methode `matchesWithRating` an. ber diese Methode wird das in Abschnitt 3.4.1 beschriebene Matcherrating bzgl. der beiden gematchten Typen zurckgegeben.

Die dritte der Methode, die von dem Interface `TypeMatcher` angeboten wird, ist `calculateTypeMatchingIn`
Diese Methode erzeugt fr einen Source- und einen Target-Typ eine Liste von Objekte der Klasse `ModuleMatchingInfo` (siehe Abbildung 4.1). Die Objekte dieser Klasse enthalten smtliche Informationen, die fr die Generierung eines Proxies fr den Source-Typen mit dem Target-Typ relevant sind. So enthlt ein solches Objekt in den Attributen `source` bzw. `target` den Source- bzw. Target-Typ fr die es erzeugt wurde. Darber hinaus ist fr jede Methode, die im Source-Typ enthalten ist, ein Objekt vom Typ `MethodMatchingInfo` in dem Attribut `methodMatchingInfos` hinterlegt. Zustzlich befindet sich im Attribut `converterCreator` ein Objekt dessen Klasse das Interface `ProxyFactoryCreator` hinterlegt. Auf die Klassen `MethodMatchingInfo` und `ProxyFactoryCreator` wird zu einem spteren Zeitpunkt weiter eingegangen.

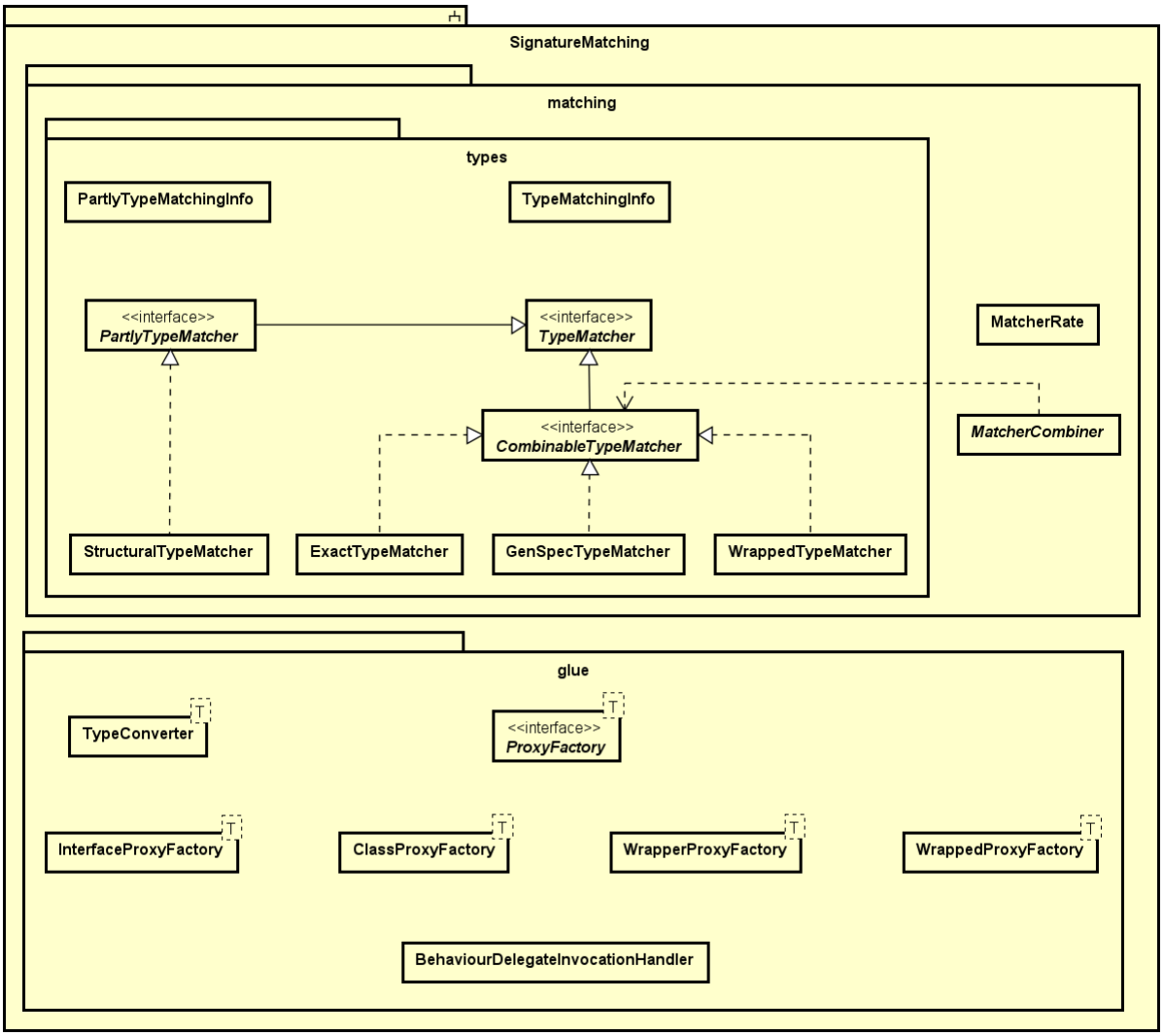

Abbildung 4.1: Modul: SignatureMatching

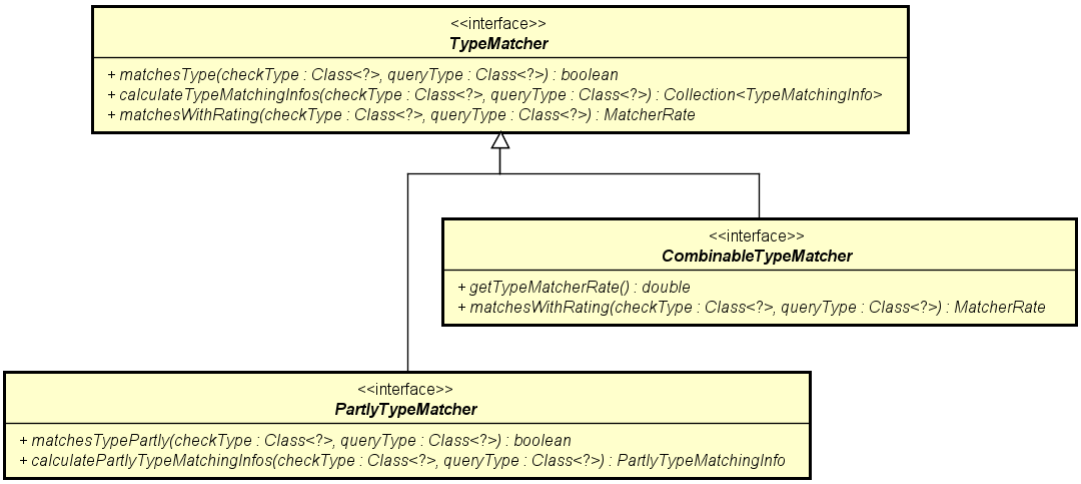

Abbildung 4.2: TypeMatcher-Interfaces

Die Matcher-Klassen `ExactTypeMatcher`, `GenSpecTypeMatcher` und `WrappedTypeMatcher` implementieren auch das von `TypeMatcher` erbende Interfaces `CombinalbeTypeMatcher`. Klassen, die dieses Interface implementieren knnen ber die Klasse `MatcherCombiner` zu einem neuen `TypeMatcher`-Objekt kombiniert werden. Ein solcher kombinierte `TypeMatcher` versucht beim Aufruf der Methode `matchesType(S,T)` die beiden Typen S und T ber einen der kombinierten Matcher zu matchen. Abbildung 4.1 zeigt das Sequenzdiagramm fr diesen Aufruf. Dabei liefert die Methode `getSortedMatcher` eine sortiert Liste der kombinierten Matcher. Die Sortierung wird aufsteigend entsprechend dem Matcherrating der kombinierten Matcher vorgenommen.

Darber hinaus gibt es noch das von `TypeMatcher` erbende Interface `PartlyTypeMatcher`. Dieses Interface wird nur von dem `StructuralTypeMatcher` implementiert, welcher u.a. als Schnittstelle zwischen dem Modul *SignatureMatching* und *DesiredComponentSourcerer* fungiert. Wie der Name des Interfaces bereits impliziert, bieten die Implementierungen des Interfaces `PartlyTypeMatcher` die Mglichkeit, zwei Typen nur teilweise zu Matchen. Das bildet die Grundlage fr die Ermittlung der Typen, aus denen die Proxies fr die semantische Evaluation erzeugt werden knnen (vgl. Abschnitt 3.1.3). So stellen die Objekte, die ber die Methode `calculatePartlyTypeMatchingInfos` erzeugten wurden, auf formaler Ebene die Elemente der

Mengen, die in Abschnitt 3.1.3 ber Funktion `cover` beschrieben wurden, dar.

Neben den Implementierungen der Matcher sind in dem Modul *SignatureMatching* auch die Generatoren fr die Proxies untergebracht. Als Schnittstelle fr die Erzeugung der Proxies dient die Klasse `TypeConverter`. Die Proxies werden unter der Zuhilfenahme der Bibliotheken *cglib* und *objenesis* erzeugt. Das Erzeugen eines Proxies erfolgt ber die Factory-Klassen, die Abbildung 4.1 entnommen werden knnen. Eine Proxy-Factory wird in Abhngigkeit von Source-Typ verwendet. Die Delegation der auf den Source-Typ aufgerufenen Methoden erfolgt auf Basis einer `MethodMatchingInfo` (siehe Klassendiagramm in Abbildung 4.1). Ein Objekt der Klasse `MethodMatchingInfo` enthlt 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. Darber hinaus wird im Attribut `returnTypeMatchingInfo` ein Objekt der Klasse `ModuleMatchingInfo` gehalten , welches alle notwendigen Informationen fr das Erzeugen eines Proxies des Rckgabetyp der aufgerufenen Methode aus dem Rckgabetyp der Delegationsmethode. Analog dazu wird im Attribut `argumentTypeMatchingInfos` eine Map, bestehend aus weiteren Objekten der Klasse `ModuleMatchingInfo` und jeweils einem Objekt der Klasse `ParamPosition`, gehalten. Diese Map enthlt alle notwendigen Information fr das Erzeugen eines Proxies fr die Parametertypen der Delegationsmethoden aus den Parametertypen der aufgerufenen Methode, sowie der Anpassung der bergabeposition bei der Delegation der aufgerufenen Methode (siehe auch Abschnitt 3.2.1).

Die Koordination der Methoden-Delegationen ist in der Klasse `BehaviourDelegateInvocationHandler` implementiert (siehe 4.1). Eine Instanz dieser Klasse wird dem Proxy-Objekt bei der Erzeugung ber einen `MethodInterceptor` (siehe [**?**]) mitgegeben.

4.2 Modul: ComponentTester

Dieses Modul ist fr die Definition und Ausfhrung der vordefinierten Tests zustndig. Dabei sei davon auszugehen, dass ein required Typ R in Form eines Interfaces existiert. Um Tests fr R zu de-

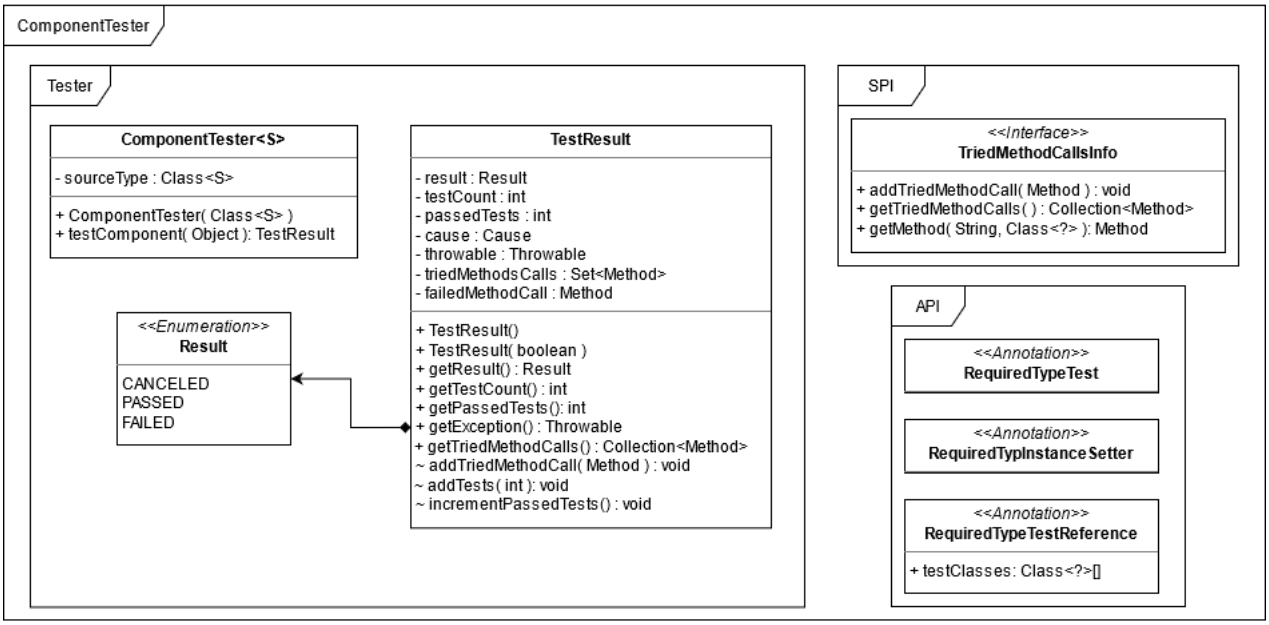

Abbildung 4.3: Modul: ComponentTester

finieren, knnen eine oder mehrere Testklassen implementiert werden. Die Testklassen werden dabei in dem Interface R ber das Attribut `testClasses` der Annotation `RequiredTypeTestReference` angegeben (siehe Abbildung 4.3 Package: *API*). Ein Beispiel fr die Deklaration eines required Typ in Form eines Java-Interfaces und den dazugehrigen Testklassen ist im Anhang zu finden.

Damit die Testmethoden in den Testklassen, den in Abschnitt 3.3.1 beschriebenen Eigenschaften gengen und durch das ComponentTester-Modul ausfindig gemacht werden knnen, stehen mehrere Artefakte in dem API- und dem SPI-Package des ComponentTester-Moduls bereit (siehe Abbildung 4.3).

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

Die Testmethoden mssen von der Sichtbarkeit her ffentlich (`public`) sein. Weiterhin drfen die

Testmethoden keine Parameter erwarten und mssen mit der Annotation `RequiredTypeTest` markiert sein. Die Erwartungen innerhalb der Testmethoden mssen ber die in JUnit 4 zur Verfgung stehenden Methoden aus der Klasse `Assert` (vgl. [**?**]) deklariert werden. Testdaten, die fr alle Testmethoden innerhalb einer Testklasse zur Verfgung stehen sollen, knnen diese innerhalb von Methoden erzeugt werden, die ber die in JUnit 4 bereitgestellten Annotationen `Before` und `After` (vgl. [**?**]) versehen werden.

Um die Reihenfolge der versuchten Aufrufe der Methoden, die von R angeboten werden, zu verwalten, muss die Testklasse das Interface `TriedMethodCallsInfo` implementieren (siehe Abbildung 4.3 Package: *SPI*). Dadurch wird die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` erzwungen. Die Methode `getMethod` kann mit der default-Implementierung bernommen werden, sofern die in R deklarierten Methoden ber den Namen identifiziert werden knnen.

Die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` hat so zu erfolgen, dass bei einem Aufruf der Methode `addTriedMethodCall` der bergebene Parameter an eine Liste angefgt wird. Der Aufruf der Methode `getTriedMethodCalls` liefert eben diese Liste als Rckgabewert. 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 fr die korrekte Implementierung einer Testklasse zu finden.

Der Test eines Proxies fr R wird ber eine Instanz der Klasse `ComponentTester` angestoen (siehe Abbildung 4.3 Package: *Tester*). In Abhngigkeit der in R deklarierten Testklassen werden alle darin befindlichen Testmethoden durchgefhrt, bis einer dieser Testflle fehlschlgt. Der Aufrufer erhlt dabei ein der Klasse `TestResult` zurck (siehe Abbildung 4.3). In diesem Objekt sind die fr 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 wird die Exploration koordiniert. Zum Starten der Exploration fr ein *desired Interface* muss zuerst eine Instanz der Klasse `DesiredComponentFinder` erzeugt werden (genannt: *Finder*). Dies erfolgt ber einen Konstruktor, der ein Objekt der Klasse `DesiredComponentFinderCon` (genannt: *Konfig*) erwartet (siehe Abbildung 4.4). Die Erzeugung einer solchen *Konfig* erfolgt

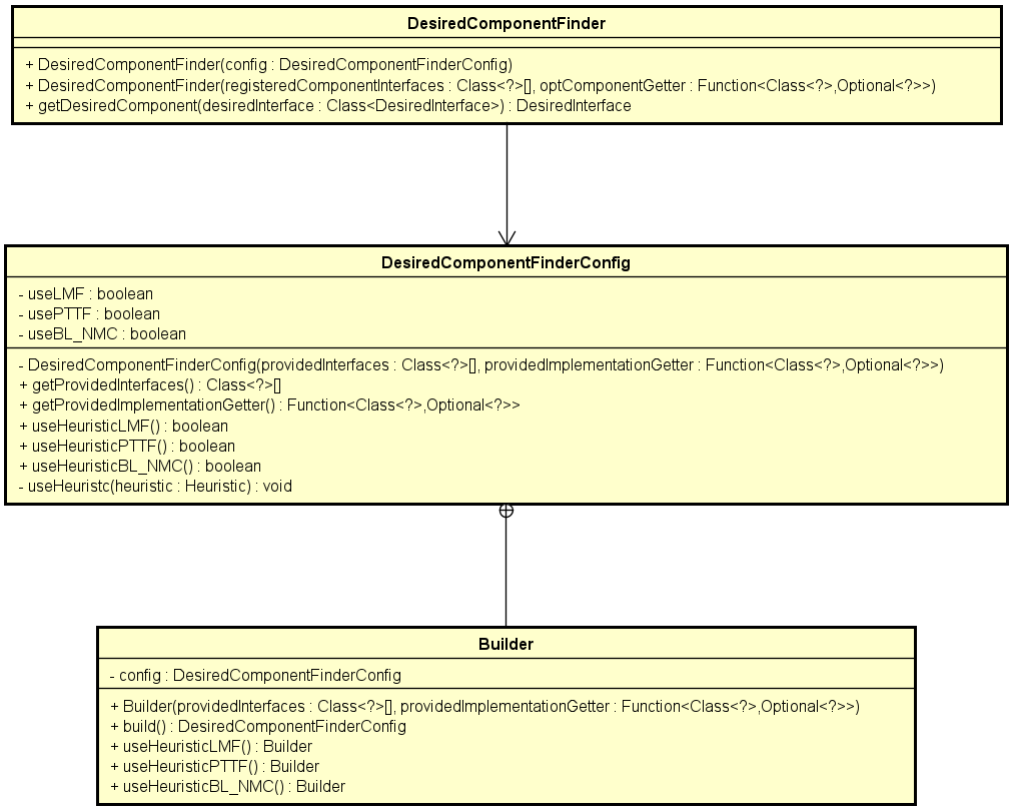

Abbildung 4.4: Schnittstellen des Moduls: DesiredComponentSourcerer

ber einen Builder. Dabei mssen zum Einen die Angabe aller provided Typen in Form einer Liste von Interfaces. Zum Anderen wird eine Funktion (`java.util.Function`) gefordert, ber die die Implementierungen der im Parameter bergebenen Interfaces ermittelt werden knnen.

Zum Zweck der gezielten Evaluation der Heuristiken in Kapitel 5 kann ber die *Konfig* gesteuert werden, welche der in Abschnitt **??** beschriebenen Heuristiken bei der Exploration verwendet werden sollen. Dies erfolgt ber die in Abbildung 4.4 ersichtlichen Methoden mit den Prfix

Abbildung 4.5: Koordination der strukturellen Evaluation

`useHeuristic*`.

Die Exploration wird nach der Erzeugung des *Finders* ber den Aufruf der Methode `getDesiredComponent` mit der bergabe des *desired Interface* R als Parameter. Im Anschluss wird die syntaktische Evaluation fr alle *provided Interfaces* durchgefhrt. Auf formaler ebene gleicht dieser Schritt der Ausfhrung der Funktion $cover(R, L)$, wobei die in L befindlichen *provided Typen* auf die an der *Finder* bergebenen *provided Interfaces* beschrnkt sind.

Hierzu wird ein Objekt vom `StructuralTypeMatcher` aus dem *SignatureMatching*-Modul verwendet[1] und versucht die *provided Interfaces* mit dem *desiredInterface* zu matchen (siehe Abbildung 4.5).

Nach der syntaktischen Evaluation, wird gem Abschnitt **??** die semantische Evaluation durchgefhrt. Dabei werden zuerst die Proxies aus den Kombinationen der gematchten *provided Interfaces*[2] 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` zurckgegeben.

Das Sequenzdiagramm in Abbildung 4.3 zeigt dabei die Interaktionen der drei vorgestellten Module bei der Durchfhrung der Exploration.

[1] Dieses Objekt wird beim Instanziieren des *Finders* erzeugt.
[2] Diese 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 Hei-System unterschieden.

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

Das Hei-System wurde vorrangig fr die Evaluation der testergebnis basierten Heuristiken verwendet, da hier zu jedem der 889 angebotenen Interfaces eine Implementierung existiert. Die angebotenen Komponenten wurden im Hei-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). Fr jedes erwartete Interface wird eine Vier-Felder-Tafel fr 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 knnten. 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 tatschlich erzeugt wurden. Die Zahl in der Spalte “negativ” drckt 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 bentigte Komponenten erzeugt werden, welche die semantischen Tests nicht bestehen. Dementsprechend stellt die Zeile “richtig” die Anzahl der Kombinationen dar, aus denen sich bentigte Komponenten erzeugen lassen, welche die semantischen Test bestehen. Der Fall, in dem eine Kombination nicht erzeugt wurde, aber dennoch fr die Erstellung einer bentigten Komponente genutzt wurde und die semantischen Tests besteht, (negativ und richtig) kann nicht auftreten.

Fr die Anzahl der zu kombinierenden Methoden-Konvertierungsvarianten MK wird der hchste mgliche Wert angenommen. Dieser ist von der Anzahl der angebotenen Methoden am sowie der Anzahl der erwarteten Methoden em abhngig und wird wie folgt berechnet:

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

Die Anzahl der angebotenen Methoden am ist wiederum abhngig 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 fr TK berechnet sich in Abhngigkeit 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 nherungsweise Darstellung von am, in Abhngigkeit 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 fr 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 bentigte 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

Anhang A

Semantische Evaluation mit allen vorgestellten Heuristiken

Die in den Abschnitten 3.4.1 - **??** vorgestellten Heuristiken knnen miteinander Kombiniert werden. Listing A.1 zeigt die Implementierung der Funktionen, die fr diese Kombination auf der Basis von Listing 3.10 angepasst oder ergnzt werden mssen.

```
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         optmizedProxies = proxies.removeAll( testedProxies )
16         // Heuristik PTTF
17         if( passedTests > 0 ){
18           priorityTargets.addAll( proxy.targets )
19           optmizedProxies = PTTF( optmizedProxies )
20         }
21         // Heuristik BL_FFMD und BL_FSMT
22         if( blacklistChanged ){
```

```
23            optmizedProxies = BL( optmizedProxies )
24          }
25          return evalProxiesMitTarget( optmizedProxies, 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