FernUniversität Hagen

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

Tabellenverzeichnis

Listings

Kapitel 1

Theoretische Grundlagen

1.1 Strukturelle Evaluation

1.1.1 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.
$PD ::=$ `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 Rückgabe-Typs T'.

Tabelle 1.1: Struktur für die Definition einer Bibliothek von Typen

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

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

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

Darüber 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 1.1: Bibliothek *ExampLe* von Typen

1.1.2 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 1.2 zu entnehmen.

Regel	Erläuterung
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{MDEL_1, ..., MDEL_k\}$	Ein Proxy wird für 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 Rückgabetyp 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 Rückgabetyp 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 Rückgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `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 1.2: Grammatikregeln mit Erläuterungen für die Definition eines Proxies

Es handelt sich dabei um Produktionsregeln einer Attributgrammatik. Die dazugehörigen Attribute sind der Tabelle 1.3 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 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 1.2: 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**
$PROXY$ ::= `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 1.3: Grammatikregeln mit Attributen für die Definition eines Proxies

Die Delegation aus dem oben genannten Beispiel kann schematisch wie in Abbildung 1.1 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-

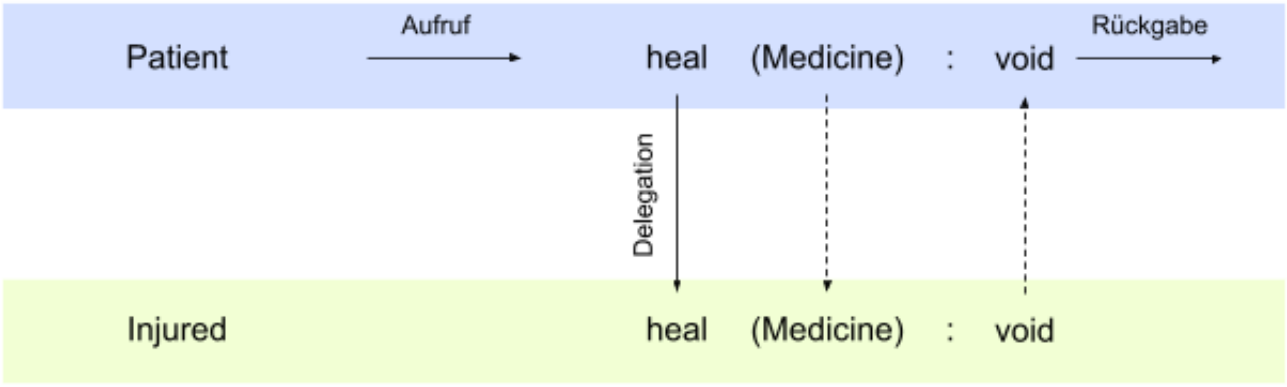

Abbildung 1.1: Delegation der Methode `heal`

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 1.1.2 ist ein Beispiel für einen solchen Fall. Hier wird die aufgerufene Methode `heal` mit den Parametern `Patient` und `MedCabinet` aus dem Typ `PatientMedicalFireFighter` an die gleichnamige Methode aus dem Typ `InverseDoctor` delegiert. Die 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 1.3: 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 1.2).

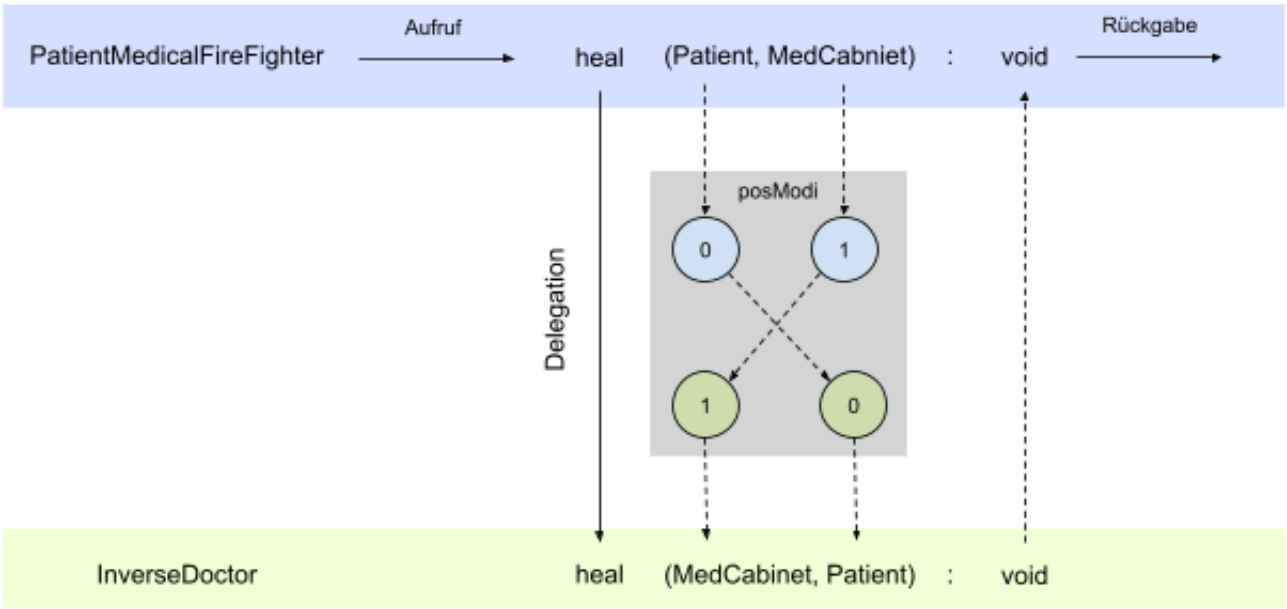

Abbildung 1.2: Delegation der Methode `heal` mit Parametern in unterschiedlicher Reihenfolge

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 aufgerührt, bei der sowohl die Parameter- als auch die Rückgabe-Typen der aufgerufenen Methode und der Delegationsmethode nicht auf Basis des Substitionsprinzips übergeben werden können.

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

Listing 1.4: 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 Rückgabetyp der Delegationsmethode in den Rückgabetyp 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 Rückgabetyp übergeben werden. Der Sachverhalt wird in Abbildung 1.3 schematisch dargestellt.

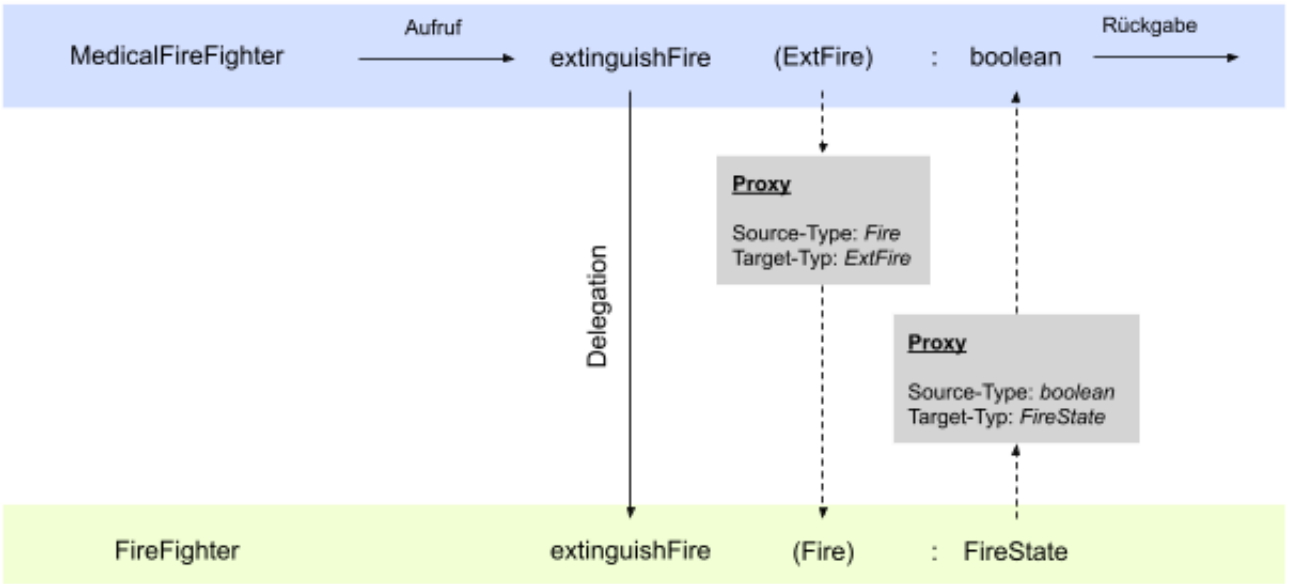

Abbildung 1.3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

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

1.1.3 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.

Matcher

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:

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

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:

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

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 `booealn` 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`:

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

Zur Definition des letzten Matchers werden die Matchingrelationen der oben genannten Mat-

cher 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:

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

Da die Notation es nicht hergibt, ist 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:

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

Generierung von Proxies

Wie im Abschnitt 1.1.2 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 1.1.2 aufgeführt ist. Um die Regeln für die Generierung der Proxies zu beschreiben, soll davon ausgegangen werden, dass jedes Listen-Attribut ($NT.*$) aus Tabelle 1.3 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-Proxies* 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 1.5: Sub-Proxy für Patient

Der abstrakte Syntaxbaum mit den dazugehörigen Attributen ist Abbildung 1.4 zu entnehmen. [1] Der Proxy bietet alle Methoden an, die auch von dessen Source-Typ angeboten werden. Die 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.

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

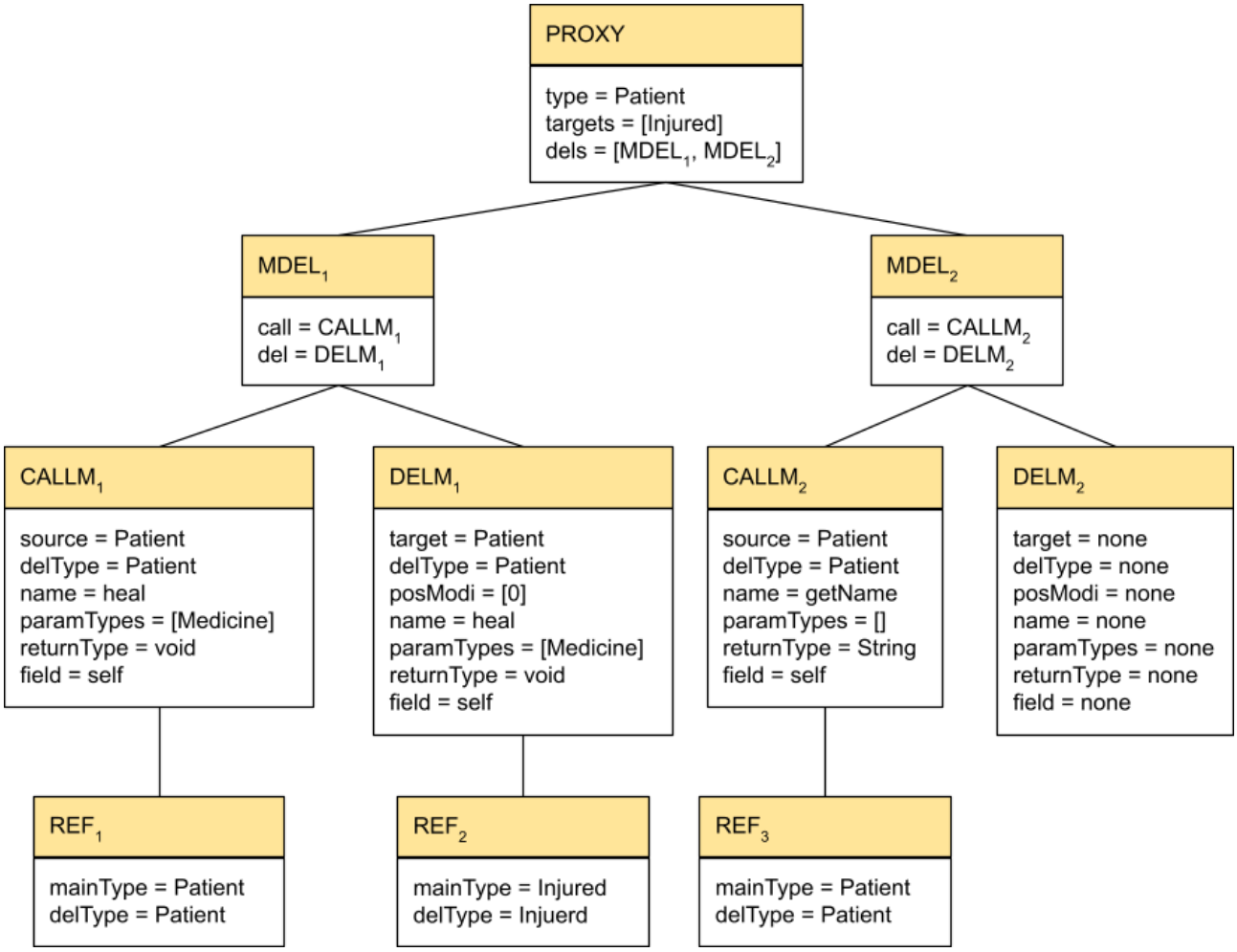

Abbildung 1.4: AST für das Beispiel zum Sub-Proxy

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

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

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 1.3)

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 1.3)

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:

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

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.

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

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.getDesciption():String → MedCabinet.med.getDesciption():String
```

```
}
```

Listing 1.6: 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 1.5 zu entnehmen. [2]

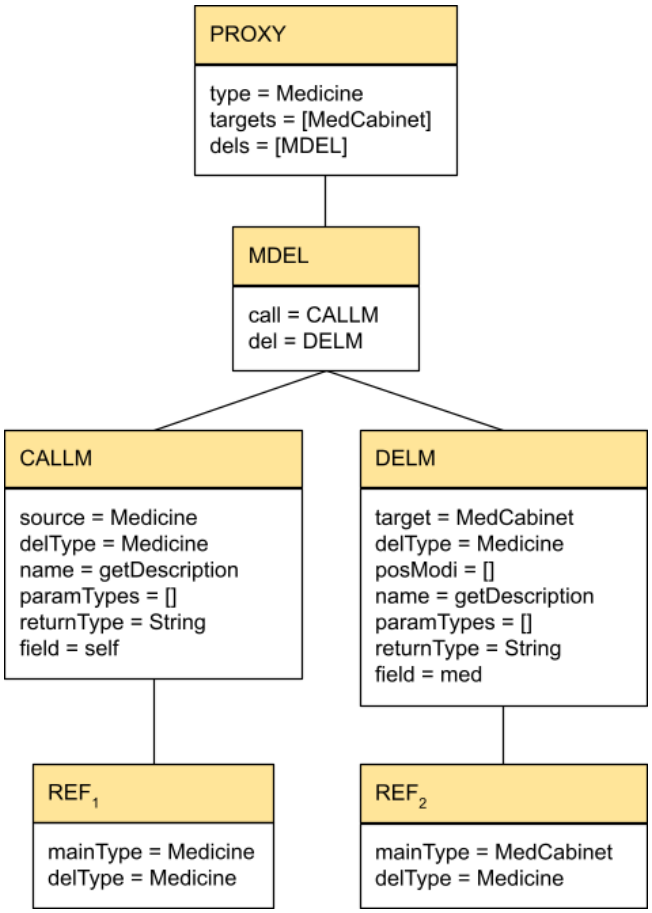

Abbildung 1.5: AST für das Beispiel zum Content-Proxy

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

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 `delTyp` und den Source-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont} MD.del.delTyp$ gelten. Daher gilt für den *Content-Proxy* die folgende Regel:

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

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:

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

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 für 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 für den *Container-Proxy*.

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

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

Listing 1.7: 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 1.6 zu entnehmen. [3]

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

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

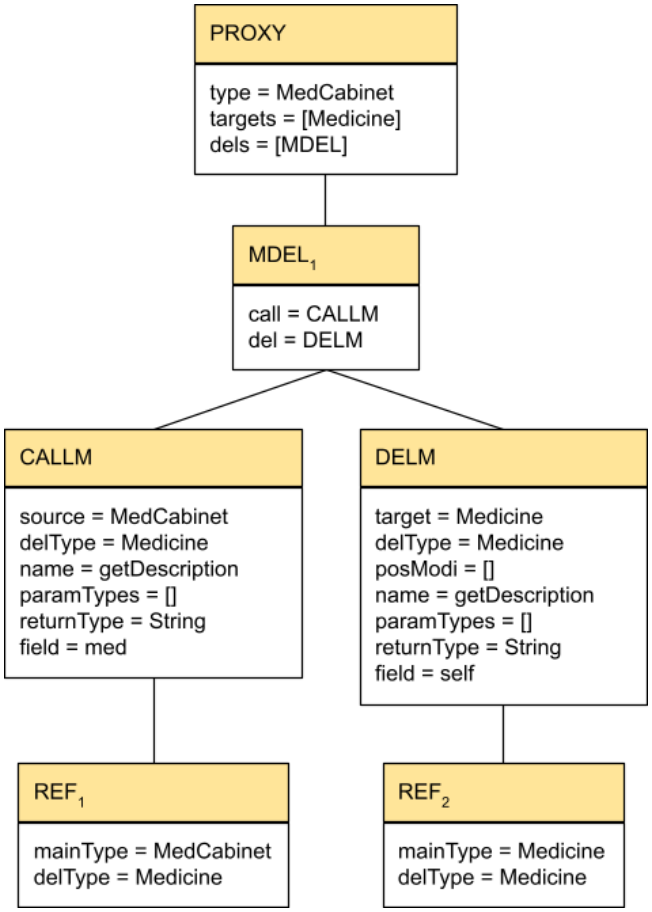

Abbildung 1.6: AST für das Beispiel zum Container-Proxy

gleichen denen aus dem *Sub-Proxy*.

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 `delTyp` und der Target-Typ T des Proxies ein Matching der Form $T \Rightarrow_{internCont}$ `delTyp` 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:

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

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 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 1.8: 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 1.7 zu entnehmen. [4]

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:

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

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.

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

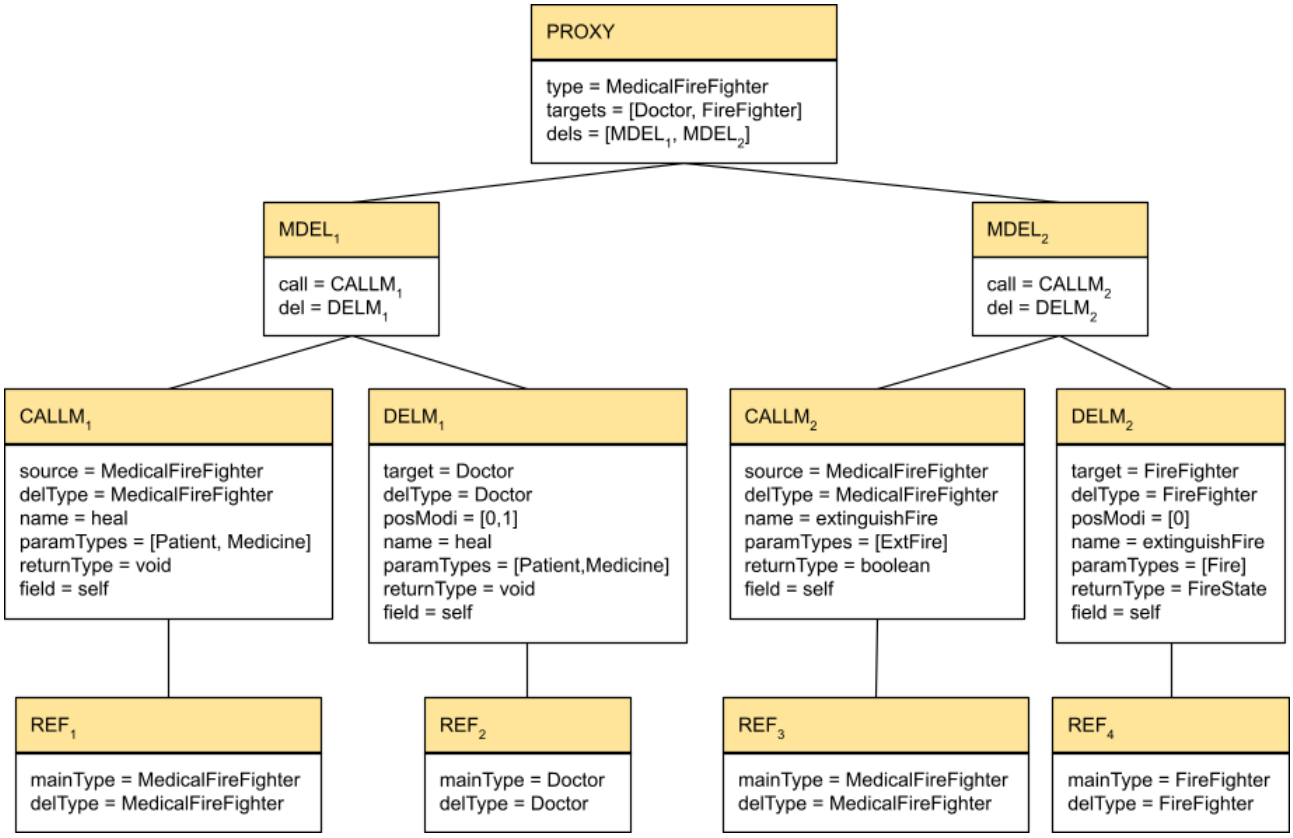

Abbildung 1.7: AST für das Beispiel zum strukturellen Proxy

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:

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

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

Für 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 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:

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

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 für die Generierung der Proxies verwendet. Dazu sind die Proxy-Arten zusammen mit den dazugehörigen Matchingrelationen und Proxy-Fukntionen in Tabelle 1.4 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 1.4: Proxy-Arten mit Matchingrelationen und Proxy-Funktionen

Die im Abschnitt 1.1.2 erwähnte Funktion $proxies(S, T)$ kann darauf aufbauend für einen Source-Typ S und eine Menge von Target-Typen T wie folgt beschrieben werden.

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

Anzahl möglicher Proxies innerhalb einer Bibliothek

Innerhalb einer Bibliothek L kann für einen required Typ R mitunter eine Vielzahl von *Proxies* erzeugt werden. Die folgende Funktion *cover* beschreibt eine Menge von Mengen von provided Typen aus der Bibliothek L, die für die Erzeugung eines Proxies für R verwendet werden können.

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

Beispiel 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.

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

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 $T_1, ..., T_k$ 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 $A\ \ m(P)$ übereinstimmen, wird über 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 übereinstimmen.

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

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

Beispiel Aufbauend auf dem vorherigen Beispiel ergeben sich für die Menge der Target-Typen $\{\texttt{Leave}, \texttt{Come}\}$ und die beiden Methoden des required Typs `Greeting` folgende Menge von übereinstimmenden Methoden über die Funktion $structM_{target}$:

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

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

Darauch 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 $M_1, ..., M_n$ 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 $\{m_1, ..., m_n\} = methoden(R)$, 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 $dm \in M_1 \cup ... \cup M_n$ innerhalb eines Proxy maximal einmal verwendet werden darf. Es ist jedoch möglich, dass es zwischen den oben genannten Mengen $M_1, ..., M_n$ Überschneidungen gibt (siehe vorheriges Beispiel). Daher gelten für die Funktion *proxyCount* folgende Regeln unter den oben genannten Modalitäten:

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

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

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 für einen required Typ R infrage kommen (siehe Funktion *cover*) muss die Anzahl der Proxies über die Funktion *proxyCount* für alle Elemente aus $cover(R, L)$ ermittelt und summiert wer-

den. Die folgende Funktion beschreibt diesen Sachverhalt für einen required Typ R aus einer Bibliothek L.

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

1.2 Semantische Evaluation

Das Ziel der semantischen Evaluation ist es, einen der Proxies, die im Rahmen der 1. Stufe der Exploration erzeugt wurden, 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 den vordefinierten Testfällen standhalten. Vielmehr soll bei der semantischen Evaluation lediglich ein Proxy gefunden werden, dessen Semantik zu positiven Ergebnissen hinsichtlich aller vordefinierten Testfälle führt. Somit wird die semantische Evaluation beendet, sobald ein solcher Proxy gefunden ist.

1.2.1 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 die in Tabelle 1.5 aufgeführten Felder, deren Werte bei der Abarbeitung der Methode `eval` verändert werden.

Feldname	**Erläuterung**
`calledMethods`	Eine Liste von Namen von Methoden des Proxies, die bei der Durchführung der eval-Methode aufgerufen wurden.
`failedMethod`	Der Name einer Methode des Proxies, bei der es während der Ausführung der eval-Methode zu einem Fehler kam.

Tabelle 1.5: Felder der Tests

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

1. Nach einem erfolgreichen Aufruf einer Methode auf dem als Parameter übergebenen Proxy-Objekt, wird der Name der dieser Methode in der Liste im Feld `calledMethods` ergänzt.

2. Nach einem fehlgeschlagenen Aufruf einer Methode auf dem als Parameter übergebenen Proxy-Objekt, wird das Feld `failedMethod` mit dem Namen der fehlgeschlagenen Methode belegt. Zusätzlich wird die eval-Methode direkt danach mit dem Rückgabewert `false` beendet.

3. Wenn der Proxy den Test erfüllt, wird der Wert `true` zurückgegeben. Anderenfalls wird der Wert `false` zurückgegeben.

Beispiel 1 In folgendem Listing 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*(*Integerx*, *Integery*) anbietet. Der Fehlschlag (`err`) dieser Methode wird über einen Try-Catch-Block abgefangen.

```
1  function eval( proxy ){
2    res = 0
3    try{
4      res = proxy.add(1, 1)
5      calledMethods.add( "add" )
6    } catch( err ) {
7      failesMethods.add( "add" )
8      return false;
9    }
10   return res == 2;
11 }
```

1.2.2 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:

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

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:

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

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 $T = cover(R, L)$ 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 1.9: Semantische Evaluation ohne Heuristiken

Die Dauer der Laufzeit der in Listing 1.9 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.

1.3 Heuristiken

Die Heuristiken werden an unterschiedlichen Stellen des Algorithmus aus Listing 1.9 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.

1.3.1 Heuristiken für die Optimierung der Reihenfolge

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.

Heuristik LMF: Beachtung des Matcherratings

Bei dieser Heuristik 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 **??** 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 erfügen. 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 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 Rückgabetypen aus der aufgerufenen Methode und den Delegationsmethode einer Methoden-Delegation MD

[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 angehängter Quellcode)

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 **??** untersucht werden.

Dazu seien die folgenden Hilfsfunktionen definiert:

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

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

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

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

Variante 1: Durchschnitt

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

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

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* abhängt und für *mdrating* 4 Variante gegeben sind, ergeben sich für jede gegebene 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 qualitative Rating bei der Auswahl der Proxies in der semantischen Evaluation beachtet werden. Die erfolgt innerhalb der Methode `applyHeuristic(proxies)`. Für diese Heuristik sei dazu eine Methode `sort(proxies, rateFunc)` angenommen, die eine Liste zurückgibt, in der die Elemente in der übergebenen Liste `proxies` aufsteigend nach den Werten sortiert, die durch die Applikation der im Parameter `rateFunc` übergebenen Funktion auf ein einzelnes Element aus der Liste `proxies` ermittelt werden. Darauf aufbauend wird die Methode `applyHeuristic(proxies)` für diese Heuristik in Pseudo-Code wie folgt definiert:

```
1  function relevantProxies( proxies, anzahl ){
2    relProxies = proxiesMitTargets( proxies, anzahl );
3    return LMF( relProxies )
4  }
5
6  function LMF( proxies ){
7    for ( n=proxies.size(); n>1; n--){
8      for( i=0; i<n-1; i++){
9        if( rating( proxies[i] ) < rating( proxies[i+1] ) ){
10         tmp = proxies[i]
11         proxies[i] = proxies[i+1]
12         proxies[i+1] = tmp
13       }
14     }
15   }
16   return proxies
17 }
```

Heuristik PTTF: Beachtung bestandener Tests

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 sturkturellen 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 $TG \in cover(R, L)$.

Wie bereits erwähnt, ist das Ergebnis der semantischen Tests ausschlaggebend für diese Heuris-

tik. Es wird davon ausgegangen, dass wenn ein Teil der Testfälle durch einen Proxy P erfolgreich durchgeführt werden, sollte die Reihenfolge der zu prüfenden Proxies so angepasst werden, dass die Proxies, die einen Target-Typen des Proxies P verwenden, zuerst geprüft werden.

Dafür sind mehrere Anpassungen bzgl. der Implementierung von Nöten.

Für die Methoden `evalProxiesMitTarget(P,anzahl,T)` ergeben sich darüber hinaus mehrere Änderungen. Die Implementierung mit allen Anpassungen ist Listing 1.10 zu entnehmen. Die einzelnen Änderungen werden im Folgenden erläutert.

Merken der priorisierten Target-Typen

Um die Optimierungen auf der Basis dieser Heuristik vornehmen zu können, wird von einer globalen Variable `priorityTargets` ausgegangen. In dieser Variablen wird eine Liste von Target-Typen der Proxies gehalten, für die wenigsten ein Testfall erfolgreich durchgeführt wurde (siehe Listing 1.10 Zeile 14).

Aktualisierung der Proxy-Liste aus der aktuellen Iteration

Im Vergleich zu der Heuristik LMF aus dem vorherigen Abschnitt bietet die Heuristik PTTF die Möglichkeit auch die Reihenfolge der Proxies aus der aktuellen Iteration zu optimieren. Dazu muss die Heuristik PTTF auf die Proxies, die in dieser Iterationsstufe noch nicht evaluiert wurden, angewandt werden (siehe Listing 1.10 Zeile 17). Zu diesem Zweck werden die in dieser Iterationsstufe bereits evaluierten Proxies in einer Liste die in der Variablen `testedProxies` gespeichert (siehe Listing 1.10 Zeile 11). Diese Liste dient dann zur Reduktion der Proxy-Liste, über die in dieser Methode iteriert wird (siehe Listing 1.10 Zeile 16).

```
1  function evalProxiesMitTarget(proxies, tests){
2    testedProxies = []
3    for( proxy : proxies ){
4      passedTestcases = 0
5      evalProxy(proxy, tests)
6      if( passedTestcases == T.size ){
7        // passenden Proxy gefunden
8        return proxy
9      }
10     else{
11       testedProxies.add(proxy)
12       if( passedTests > 0 ){
```

```
13          priorityTargets.addAll( proxy.targets )
14          // noch nicht evaluierte Proxies ermitteln
15          leftProxies = proxies.removeAll( testedProxies )
16          optmizedProxies = PTTF( leftProxies )
17          return evalProxiesMitTarget( optmizedProxies, tests )
18        }
19      }
20    }
21    // kein passenden Proxy gefunden
22    return null
23  }
24
25  function relevantProxies( proxies, anzahl ){
26    relProxies = proxiesMitTargets( proxies, anzahl );
27    return PTTF( relProxies )
28  }
29
30  function PTTF(proxies){
31    for ( n=proxies.size ; n>1; n--){
32      for( i=0; i<n-1; i++){
33        targetsFirst = proxies[i].targets
34        targetsFirst = proxies[i+1].targets
35        if( !priorityTargets.contains(targetsFirst) &&
              priorityTargets.contains(targetsSecond) ){
36          tmp = proxies[i]
37          proxies[i] = proxies[i+1]
38          proxies[i+1] = tmp
39        }
40      }
41    }
42    return proxies
43  }
```

Listing 1.10: Auswertung des Testergebnisses mit Heuristik PTTF

1.3.2 Heuristiken für den Ausschluss von Methodendelegationen

Bei den folgenden Heuristiken handelt es sich um 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 Heuristiken zielen darauf ab, Methodendelegationen, die immer fehlschlagen, zu identifizieren. Wurde eine solche Methodendelegation gefunden, können alle Proxies, die diese Methodendelegation enthalten von der weiteren Exploration ausgeschlossen werden.

Um eine solche Methodendelegation identifizieren zu können, müssen die Testfälle weitere Besonderheiten erfüllen und Informationen bereitstellen. Zum Einen ist es notwendig, dass spezielle Tests verwendet werden, in denen lediglich eine Methode getestet wird. So kann auf der Basis eines Fehlgeschlagenen Tests geschlussfolgert werden, dass die Methodendelegation, die für die getestete Methode verwendet wurde, ebenfalls in anderen Proxies zu einem Fehlschlag des Testfalls führt.

Zu diesem Zweck sei angenommen, dass ein einzelner Test über ein Attribut `isSingleMethodTest` und `singleMethodName`. Die beiden Attribute sind durch den Entwickler bei der Implementierung dieser Tests zu füllen. Dabei ist vorgesehen, dass das Attribut `isSingleMethodTest` mit dem Wert `true` belegt ist, wenn es sich um einen Test handelt, in dem lediglich eine Methode des Proxies aufgerufen wird. Darüber hinaus ist die Attribut `singleMethodName` mit dem Namen der in dem Testfall aufgerufene Methode des Proxies zu belegen.

Basierend auf diesen Werten der beiden Attribute, können die Tests in folgende Kategorien unterteilt werden:

Test-Kategorie	**Eigenschaften**
Single-Method-Test	`isSingleMethodTest` = `true` `singleMethodName` $\neq$ `null`
Multi-Method-Test	`isSingleMethodTest` = `false`

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

Zusätzlich ist zu erwähnen, dass die folgenden Heuristiken, wie auch die Heuristik PTTF, eine Optimierung der zu testenden Proxies nach jeder fehlgeschlagenden Evaluation eines Proxies erlauben. Ob eine Optimierung an diesem Punkt möglich ist, wird durch eine neue globale Variable `blacklistChanged` gesteuert.

Listing 1.3.2 zeigt die allgemeinen Anpassungen für die folgenden Heuristiken basieren auf den Funktionen aus Listing 1.3.2.

```
1  methodDelegationBlacklist = []
2  blacklistChanged = false
3
4  function evalProxiesMitTarget(proxies, tests){
5    testedProxies = []
6    for( proxy : proxies ){
7      passedTestcases = 0
8      evalProxy(proxy, tests)
9      if( passedTestcases == tests.size ){
10       // passenden Proxy gefunden
11       return proxy
12     }
13     else{
14       testedProxies.add(proxy)
15       if( blacklistChanged ){
16         // noch nicht evaluierte Proxies ermitteln
17         leftProxies = proxies.removeAll(testedProxies)
18         optimizedProxies = BL( leftProxies )
19         return evalProxiesMitTarget( optimizedProxies, tests )
20       }
21     }
22   }
23   // kein passenden Proxy gefunden
24   return null
25 }
26
27 function relevantProxies( proxies, anzahl ){
28   relProxies = proxiesMitTargets( proxies, anzahl );
29   return BL( optimizedFSMT )
30 }
31
32 function BL( proxies ){
33   optimizedProxies = []
34   for( proxy : proxies ){
35     blacklisted = false
```

```
36      for( md : methodDelegationBlacklist ){
37        if( proxy.dels.contains( md ) ){
38          blacklisted = true
39          break
40        }
41      }
42      if( !blacklisted ){
43        optimizedProxies.add( proxy )
44      }
45    }
46    return optimizedProxies
47  }
```

Die folgenden Heuristiken erfordern jeweils eine Anpassung der Funktion `evalProxy`.

Heuristik BL_FSMT: Beachtung fehlgeschlagener Single-Method-Test

Basierend darauf, dass ein

```
1
2
3  function evalProxy(proxy, T){
4    for( test : T ){
5      if( test.eval( proxy ) ){
6        passedTestcases = passedTestcases + 1
7      }elseif( test.isSingleMethodTest ){
8        methodName = test.singleMethodName
9        mDel = getMethodDelegation(proxy, methodName)
10       blacklistChanged = true
11     }
12   }
13 }
14
15 function getMethodDelegation( proxy, methodName ){
16   for( i=0; i < proxy.dels.size; i++ ){
17     if( proxy.dels[i].call.name == methodName ){
18       return proxy.dels[i]
19     }
20   }
21   return null
22 }
```

Listing 1.11: Semantische Evaluation mit Heuristik SMTE

Heuristik BL_FFMD: Beachtung fehlgeschlagener Methoden-Delegationen

```
failedMethodDelegation = []

function evalProxy(proxy, T){
  for( test : T ){
    //alle Tests werden durchgefuehrt
    try{
      if( !test.eval( proxy ) ){
        return
      }
      passedTestcases = passedTestcases + 1
    }
    catch (SigMaGlueException e){
      mDel = e.failedMethodDelegation
      if( test.isSingleMethodTest &&
        mDel.call.name == test.singleMethodName){
        failedMethodDelegation.add(mDel)
        blacklistChanged = true
      }
      return
    }
  }
}
```

Listing 1.12: Abfangen der SigMaGlueException beim Testen eines Proxies

1.3.3 Kombination der Heuristiken

Die oben genannten Heuristiken können miteinander Kombiniert werden. Listing 1.13 zeigt die Implementierung der Funktionen, die für diese Kombination auf der Basis von Listing 1.9 angepasst werden müssen. Dabei ist davon auszugehen, dass die Funktionen `LMF`, `PTTF`, `FSMT` und `FFMD` definiert sind.

```
function evalProxiesMitTarget( proxies, tests ){
  testedProxies = []
  for( proxy : proxies ){
    passedTestcases = 0
    blacklistChanged = false
    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 1.13: Kombination aller Heuristiken

6