1 Beispiel-Bibliothek

```
provided Fire extends Object{}

provided ExtFire extends Fire{}

provided FireState extends Object{
        isActive : boolean
}

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{
        med : Medicine
}

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: Bibliothek *ExampLe* von Typen

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 zu entnehmen.

Regel	Erläuterung
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{MDEL_1, ..., MDEL_k\}$	Ein Proxy wird für ein Typ T mit einer Mengen von *provided Typen* $P = \{P_1, ..., P_n\}$, 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 Methode 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 Target-Referenz, dem Namen der Methode n, dem Rückgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `err`	Die dritte Variante eines Delegationsziels besteht führt zu einem Fehler, da die Delegation innerhalb des Proxies nicht möglich ist.
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: Grammatikregeln mit Erläuterungen für die Definition eines Proxies

Es handelt sich dabei um Produktionsregeln einer Attributgrammatik. Die dazugehörigen Attribute sind der Tabelle 2 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 List verwendet wird.

Weiterhin sei ein Attribut, dass in der Spalte *Attribute* zu einem Nonterminal nicht aufgeführt ist, wird mit dem Wert *none* belegt.

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 2: Grammatikregeln mit Attributen für die Definition eines Proxies

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

Beispiel So definiert 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 vom *Target-Typ* `Injured` angeboten wird, delegiert wird.

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

Die Delegation einer aufgerufenen Methode an eine Delegationsmethode, kann in drei Schritte unterteilt werden.

1. Parameterübergabe
   Dabei werden die Parameter, mit denen die vom Proxy angebotene Methode, aufgerufen wird, an die Delegationsmethode 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. Und 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 Delegationsmethoden mit den übergeben Parametern. 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 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.

Die Delegation aus dem oben genannten Beispiel kann schematisch wie in Abbildung 1 dargestellt werden. Die Übergabe der Parameter- und Rückgabewerte, die in der Proxy-Definition mit ihren Typen angegeben werden, wird durch die gestrichelten Pfeile symbolisiert.

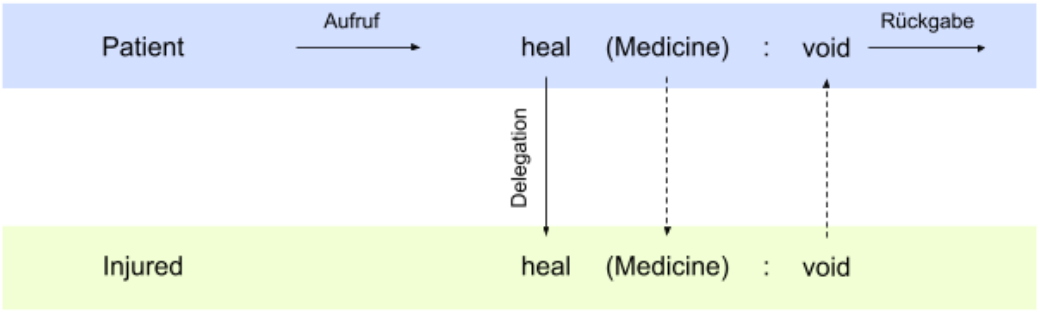

Abbildung 1: Delegation der Methode `heal`

An diesem Beispiel ist leicht zu erkennen, dass die Parameter- und Rückgabe-Typen zueinander passen, da sie jeweils identisch sind. Weiterhin spielt die Reihenfolge der Parameter in diesem Beispiel keine Rolle, da es nur einen Parameter gibt.

Folgendes Beispiel soll zeigen, wie mit unterschiedlichen Parameter-Reihenfolgen umzugehen ist.

Beispiel Die Methoden-Delegation aus Listing 2 ist ein Beispiel für einen solchen Fall. Hier wird die angebotenen 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
```

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 solcher 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 erste Parameter übergeben (siehe auch Abbildung 2).

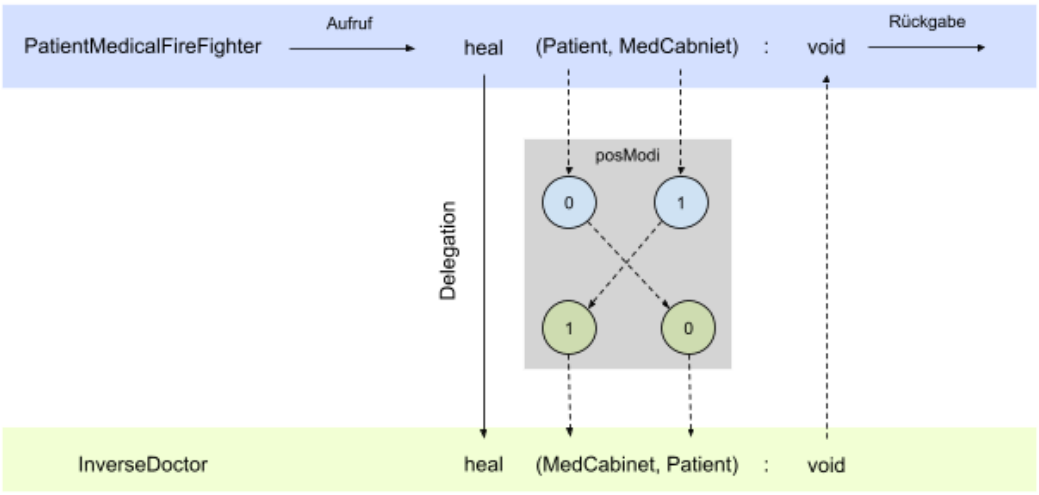

Abbildung 2: Delegation der Methode `heal` mit unterschiedlicher Reihenfolge

Ein weiteres Beispiel soll zeigen, wie mit übergebenen Typen umzugehen ist, die nicht zueinander passen. Dafür ist jedoch vorab zu klären, wann dies der Fall ist.

Dass identische Typen keine Probleme bei der Übergabe zwischen aufgerufener Methoden und Delegationsmethode darstellen, wurde in den oberen Beispielen gezeigt.

Darüber hinaus können aber auch Typen ohne Probleme übergeben werden, wenn sie sich aufgrund des Substitutionsprinzips austauschen lassen. Unter Verwendung der o.g Matcher kann ein Typ T anstelle eines Typs T' verwendet werden, sofern folgende Regel gilt.

$$\frac{T' \Rightarrow_{exact} T \vee T' \Rightarrow_{gen} T}{T' \Rightarrow_{exactGen} T}$$

In folgenden Beispiel ist dies jedoch nicht der Fall.

Beispiel In Listing 2 ist eine Methoden-Delegation aufgerührt, bei der sowohl die Parameter- als auch die Rückgabe-Typen der aufgerufenen Methode und der Delegationsmethoden nicht übereinstimmen.

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

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 das oben genannte Beispiel für die Parameter-Typen einer der Proxies aus der Menge *proxies*(`Fire`, {`ExtFire`}) an die Delegationsmethode übergeben werden. Nach der Ausführung der Delegationsmethode müsste Proxy aus der Menge *proxies*(`boolean`, {`FireState`}) an die aufgerufenen Methode als Rückgabetyp übergeben werden. Der Sachverhalt wird in Abbildung 3 schematisch dargestellt.

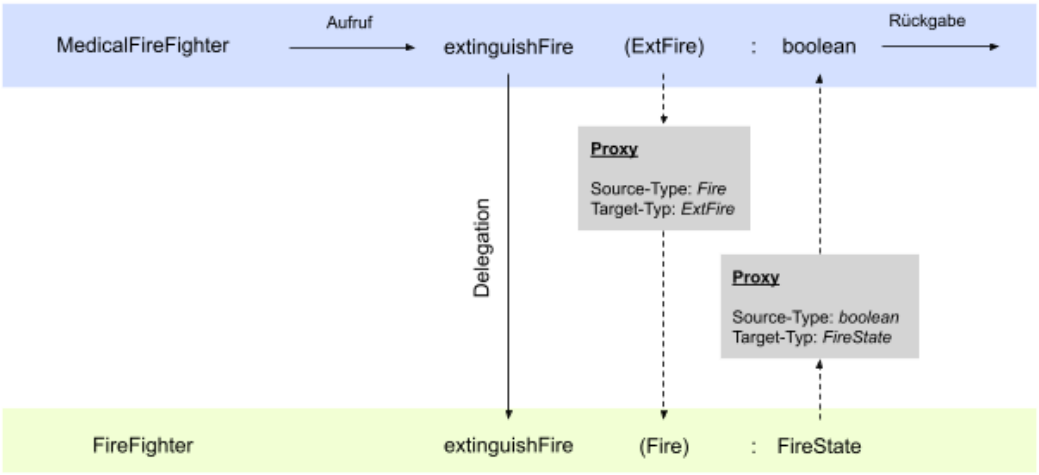

Abbildung 3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

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

3 Generierung der Proxies auf Basis von Matchern

Die Matcher beinhalten die Definition der jeweiligen Matchingrelation ($\Rightarrow$). Auf deren Basis werden Proxies für bestimmte Typen erzeugt. Dabei gibt es unterschiedliche Arten von Proxies. Jede Proxy-Art basiert auf einem anderen Matcher.

Wie im vorherigen Abschnitt bereits erwähnt, wird die Menge der Proxies für einen Source-Typ S und einer Menge von Target-Typen T über die Funktion *proxies*(S, T) beschrieben. 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. Die Namen dieser Funktionen sind zusammen mit den Proxy-Arten und den dazugehörigen Matchingrelationen in Tabelle 3 aufgeführt.

Proxy-Art	**Matchingrelation**	**Funktionsname**
Sub-Proxy	$\Rightarrow_{spec}$	$proxy_{sub}$
Content-Proxy	$\Rightarrow_{content}$	$proxy_{content}$
Container-Proxy	$\Rightarrow_{container}$	$proxy_{container}$
struktureller Proxy	$\Rightarrow_{struct}$	$proxy_{struct}$

Tabelle 3: Proxy-Arten mit Matchingrelationen und Funktionsnamen

Die im vorherigen Abschnitt erwähnt Funktion $proxy(S, T)$ kann darauf aufbauend wie folgt beschrieben werden.

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

Die Proxies haben eine allgemeine Struktur, die in Abschnitt 2 aufgeführt ist. Um die Regeln für die Generierung der Proxies zu beschreiben, soll davon ausgegangen werden, dass jedes Listen-Attribut aus Tabelle 2 ein Attribut `len` enthält in dem die Anzahl der in der Liste befindlichen Elemente abgelegt ist.

3.0.1 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 hierfür der Typ `Patient` als Source-Typ der Proxies 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
}
```

Der abstrakte Syntaxbaum mit den dazugehörigen Attributen ist folgender Abbildung 4 zu entnehmen. [1]

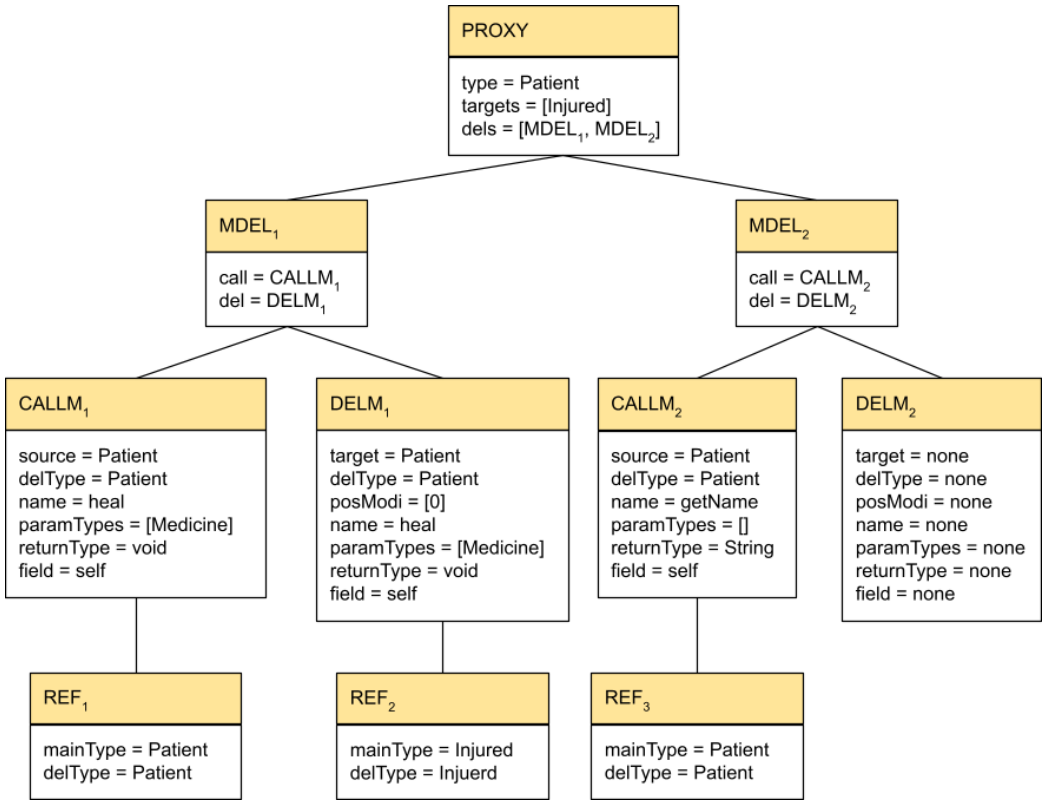

Abbildung 4: AST für das Beispiel zum Sub-Proxy

Formalisierung Wird der Proxy als Typ verwendet, so stehen darin alle Methoden zur Verfügung, die auch im Typ `Patient` zur Verfügung stehen. Die Methodendelegationen innerhalb dieses 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.

In 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 kann, spiegelt der *Sub-Proxy* dieses Verhalten wieder.

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.

$$\frac{|P.targets| = 1 \wedge P.targets[0] = T'}{singleTarget(T')}$$

Darüber hinaus enthält ein *Sub-Proxy* P eine bestimmte Menge von Methoden-Delegationen. Dabei muss das Attribut `field` sowohl in den aufgerufenen Methoden und in den Delegationsmethoden aller Methodendelegationen jeweils übereinstimmen. Folgende Regel stellt diesen Sachverhalt für eine Menge von Methodendelegationen *MDEL* dar.

$$\frac{\begin{array}{r}\forall DEL_1 \in MDEL.\neg(\exists DEL_2 \in MDEL.DEL_1.call.field \neq DEL_2.call.field \\ \vee\ DEL_1.del.field \neq DEL_2.del.field)\end{array}}{equalRefs(MDEL)}$$

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

$$\frac{DEL.call.name = DEL.del.name}{nominalDel(DEL)}$$

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

$$\frac{\exists m(P_1, ..., P_n) : R \in methoden(DEL.call.declType).DEL.call.name = m}{simpleCallMethod(DEL, P)}$$

$$\frac{\exists m(P_1, ..., P_n) : R \in methoden(DEL.del.declType).DEL.del.name = m}{simpleDelMethod(DEL, P)}$$

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

$$\frac{DEL.call.mainType = P.type \wedge DEL.call.field = self}{simpleDelSource(DEL, P)}$$

Damit ist auch gewährleistet, dass die Attribute `mainType` und `delType` im Attribut `call` übereinstimmen.

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.

$$\frac{DEL.del.mainType = P.targets[0] \wedge DEL.del.field = self}{simpleDelTarget(DEL, P)}$$

Damit ist wiederum gewährleistet, dass die Attribute `mainType` und `delType` im Attribut `del` übereinstimmen.

Die Regeln bzgl. der linken Seite einer Methoden-Delegation innerhalb eines *Sub-Proxies* können damit in folgender Regel zusammengefasst werden:

$$\frac{simpleCallMethod(DEL, P) \wedge simpleDelSource(DEL, P)}{simpleCall(DEL, P)}$$

Analog dazu können auch die Regeln bzgl. der rechten Seite einer Methoden-Delegation innerhalb eines *Sub-Proxies* zusammengefasst werden:

$$\frac{simpleDelMethod(DEL, P) \wedge simpleDelTarget(DEL, P)}{simpleDel(DEL, P)}$$

Jedoch ist im *Sub-Proxy* die Ausnahme zu beachten:

$$\frac{DEL.del.name = none}{errDel(DEL)}$$

In diesem Fall würden die o.g. Kriterien nicht gelten. Die genannten Regeln bzgl. einer Methoden-Delegation in einem *Sub-Proxy* lassen sich über beiden folgenden Regeln beschreiben:

$$\frac{simpleCall(DEL, P) \wedge simpleDel(DEL, P) \wedge nominalDel(DEL)}{subDelegation(DEL, P)}$$

$$\frac{simpleCall(DEL, P) \wedge errDel(DEL)}{subMDEL(DEL, P)}$$

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 seine Methoden-Delegationen folgende Regeln formulieren:

$$\frac{\begin{array}{l}|methoden(P.type)| = |P.dels| \wedge \\ \quad \forall m(P_1, ..., P_n) : R \in methoden(P.type). \exists DEL \in P.dels. \\ \qquad\qquad m = DEL.call.name \wedge subMDEL(DEL, P)\end{array}}{subMDELList(P)}$$

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

$$proxies_{sub}(T, T') := \left\{ P \;\middle|\; \begin{array}{l} P.\texttt{type} = T \wedge singleTarget(T') \wedge \\ equalRefs(P.dels) \wedge subMDELList(P) \end{array} \right\}$$

3.0.2 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 können hierfür die Typen `Medicine` und `MedCabinet` verwendet werden. Diese weisen ein Matching der Form $texttt Medicine \Rightarrow_{content} texttt MedCabinet$ auf. 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
}
```

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 folgender Abbildung 5 zu entnehmen. [2]

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

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

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

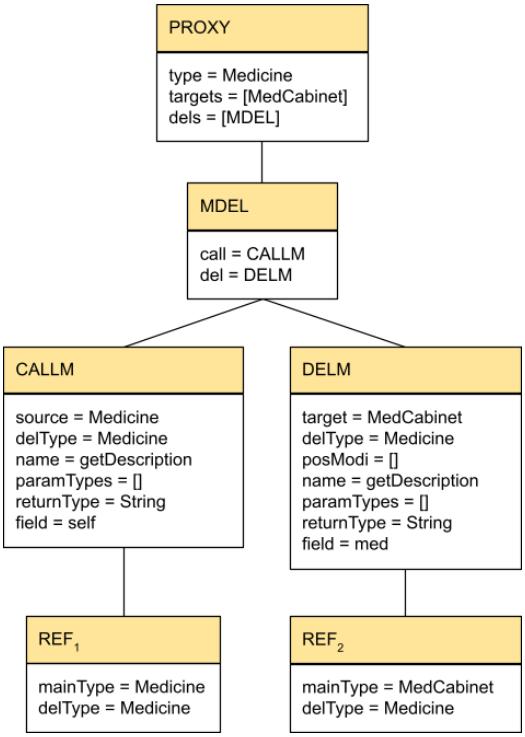

Abbildung 5: AST für das Beispiel zum Content-Proxy

In den Delegationsmethoden einer einzelnen Methoden-Delegation darf das Attribut `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 im Attribut `type` des Proxies ein Matching der Form $T \Rightarrow_{internCont}$ `delTyp` gelten. Daher gilt für den *Content-Proxy* folgende Regel.

$$\frac{DEL.del.mainType = P.targets[0] \wedge P.type \Rightarrow_{internCont} DEL.del.delType}{contentDelTarget(DEL, P)}$$

Damit ist auch die zusammenfassende Regel für die Delegationsmethoden eine andere:

$$\frac{simpleDelMethod(DEL, P) \wedge contentDelTarget(DEL, P)}{contentDel(DEL, P)}$$

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

$$\frac{simpleCall(DEL, P) \wedge contentDel(DEL, P) \wedge nominalDel(DEL)}{contentMDEL(DEL, P)}$$

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:

$$\frac{\begin{array}{l}|methoden(P.type)| = |P.dels| \wedge \\ \quad \forall m(P_1, ..., P_n) : R \in methoden(P.type).\exists DEL \in P.dels. \\ \qquad\qquad m = DEL.call.name \wedge contentMDEL(DEL, P)\end{array}}{contentMDELList(P)}$$

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

$$proxies_{content}(T, T') := \left\{ P \;\middle|\; \begin{array}{l} P.\texttt{type} = T \wedge singleTarget(T') \wedge \\ equalRefs(P.dels) \wedge contentMDELList(P) \end{array} \right\}$$

3.0.3 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 *ContainerType-Matcher* der Basis-Matcher für den *Container-Proxy*.

Beispiel Als Beispiel können hierfür wiederum die Typen `Medicine` und `MedCabinet` verwendet werden. Diese weisen ein Matching der Form `MedCabinet` $\Rightarrow_{container}$ `Medicine` auf. 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
}
```

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 folgender Abbildung 6 zu entnehmen. [3]

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

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

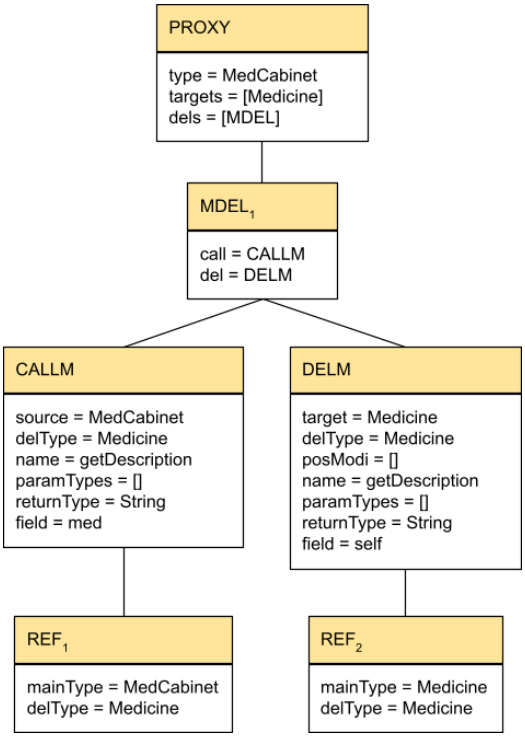

Abbildung 6: AST für das Beispiel zum Container-Proxy

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

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

$$\frac{DEL.call.mainType = P.type \wedge P.targets[0] \Rightarrow_{internCont} DEL.call.delType}{containerDelSource(DEL, P)}$$

Damit ist auch die zusammenfassende Regel für die aufgerufenen Methoden eine andere:

$$\frac{simpleCallMethod(DEL, P) \wedge containerDelSource(DEL, P)}{containerCall(DEL, P)}$$

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

$$\frac{containerCall(DEL, P) \wedge simpleDel(DEL, P) \wedge nominalDel(DEL)}{containerMDEL(DEL, P)}$$

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. Daraus ergibt sich für alle Methoden-Delegationen aus einem *Content-Proxy* P folgende Regel:

$$\frac{\begin{array}{l}|methoden(P.dels[0].call.delType)| = |P.dels| \wedge \\ \quad \forall m(P_1, ..., P_n) : R \in methoden(P.dels[0].call.delType). \\ \quad\quad \exists DEL \in P.dels.m = DEL.call.name \wedge containerMDEL(DEL, P)\end{array}}{containerMDELList(P)}$$

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

$$proxies_{container}(T, T') := \left\{ P \;\middle|\; \begin{array}{l} P.\texttt{type} = T \wedge singleTarget(T') \wedge \\ equalRefs(P.dels) \wedge containerMDELList(P) \end{array} \right\}$$

3.0.4 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 hierfür können die Typen $MedicalFireFighter$, $Doctor$ und $FireFighter$ verwendet werden. Dabei ist $MedicalFireFighter$ der Source-Typ des Proxies und die Menge der anderen beiden Typen bilden die Target-Typen des Proxies.

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

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 folgender Abbildung 7 zu entnehmen. [4]

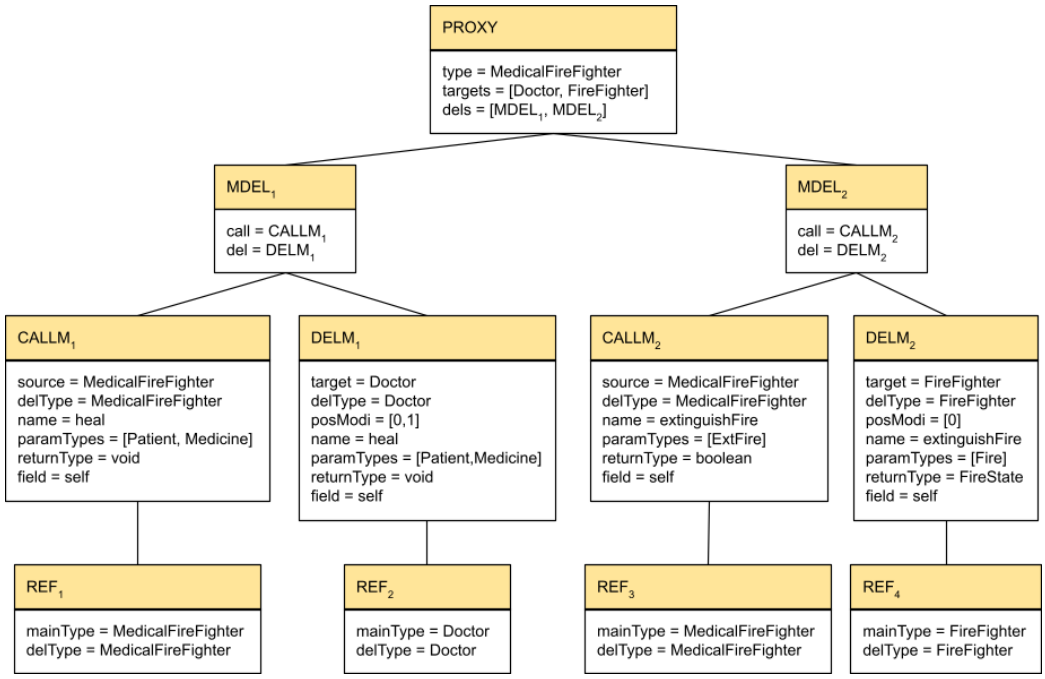

Abbildung 7: AST für das Beispiel zum strukturellen Proxy

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

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

$$\frac{\forall T \in P.targets. \exists MDEL \in \texttt{P.dels}. MDEL.del.target = T}{structTargets(P)}$$

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

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

$$\frac{DELM.returnType \Rightarrow_{internStruct} CALL.returnType}{structRT(CALL, DELM)}$$

Weiterhin muss für die Parameter-Typen gelten:

$$\frac{CALL.paramCount = 0}{structParams(CALL, DELM)}$$

$$\frac{\begin{array}{l}\forall i \in \{0, ..., CALL.paramCount - 1\}. \\ \quad CALL.paramTypes[i] \Rightarrow_{internStruct} DELM.paramTypes[DELM.posModi[i]]\end{array}}{structParams(CALL, DELM)}$$

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

$$\frac{\begin{array}{l}simpleCall(MDEL, P) \wedge simpleDel(MDEL, P) \wedge \\ \quad structRT(MDEL.call, MDEL.del) \wedge structParams(MDEL.call, MDEL.del)\end{array}}{structMDEL(MDEL, P)}$$

Für einen *strukturellen Proxy* P gilt ebenfalls die Regel $equalRefs(P.dels)$. Daher müssen die Werte des Attributs `call.delType` aller Methoden-Delegationen des Proxies P übereinstimmen.

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 *strukturellen Proxy* P folgende Regel:

$$\frac{\begin{array}{l}|methoden(P.type)| = |P.dels| \wedge \\ \quad \forall m(P_1, ..., P_n) : R \in methoden(P.type). \\ \quad\quad \exists DEL \in P.dels.m = DEL.call.name \wedge structMDEL(DEL, P)\end{array}}{structMDELList(P)}$$

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.

$$proxies_{struct}(R, T) := \left\{ P \;\middle|\; \begin{array}{l} P.\texttt{type} = R \wedge structTargets(P) \wedge \\ equalRefs(P.dels) \wedge structMDELList(P) \end{array} \right\}$$

3.0.5 Mögliche Proxies in eine Bibliothek

Innerhalb einer Bibliothek L können für einen *required Typ* R mitunter eine Vielzahl von *strukturellen Proxies* erzeugt werden. Die Anzahl hängt zum einen von der Anzahl der Mengen von *provided Typen* T ab, mit denen ein solcher *struktureller Proxie* erzeugt werden kann.

Die folgende Funktion *cover* beschreibt die eine Menge von Mengen von *provided Typen* aus einer Bibliothek L, die für die Erzeugung eines *strukturellen Proxies* für R verwendet werden können.

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

Darüber hinaus können zu einer Menge aus $cover(R, L)$ durchaus mehrere *strukturelle Proxies* erzeugt werden. Das ist dann der Fall, wenn mehrere der Methoden aus den *provided Typen* mit einer Methode aus dem *required Typ* strukturell übereinstimmen.