# 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

Für die Konvertierung eines Typs $T$ aus einer Menge von *provided Typen* $P$ wird durch Proxies beschrieben. Die Struktur eines Proxies ist 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

Dabei handelt es sich 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` $= T$<br>`targets` $= [P_1, ..., P_n]$<br>`dels` $= [MDEL_1.*, ..., MDEL_k.*]$ |
| $MDEL$ ::= $CALLM \rightarrow DELM$ | `call` $= CALLM.*$<br>`del` $= DELM.*$ |
| $CALLM$ ::= $REF.m(CP_1, ..., CP_n) : CR$ | `source` $= REF$.`mainType`<br>`delType` $= REF$.`delType`<br>`name` $= m$<br>`paramTypes` $= [CP_1, ..., CP_n]$<br>`returnType` $= CR$<br>`field` $= REF$`.field` |
| $DELM$ ::= $REF.n(DP_1, ..., DP_n) : DR$ | `target` $= REF$.`mainType`<br>`delType` $= REF$.`delType`<br>`posModi` $= [0, ..., n-1]$<br>`name` $= n$<br>`paramTypes` $= [DP_1, ..., DP_n]$<br>`returnType` $= DR$<br>`field` $= REF$`.field` |
| $DELM$ ::= `posModi`$(I_1, ..., I_n)$ $REF.n(DP_1, ..., DP_n) : DR$ | `target` $= REF$.`mainType`<br>`delType` $= REF$.`delType`<br>`posModi` $= [I_1, ..., I_n]$<br>`name` $= n$<br>`paramTypes` $= [DP_1, ..., DP_n]$<br>`returnType` $= DR$<br>`field` $= REF$`.field` |
| $DELM$ ::= `err` | |
| $REF$ ::= $P$ | `mainType` $= P$<br>`field` $=$ `self`<br>`delType` $= P$ |
| $REF$ ::= $P.f$ | `mainType` $= P$<br>`field` $= f$<br>`delType` $= feldTyp(f, P)$ |

Tabelle 2: Grammatikregeln mit Attributen für die Definition eines Proxies

# 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.

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.

| **Proxy-Art** | **Basis-Matchingrelation** |
|---|---|
| Sub-Proxy | $\Rightarrow_{spec}$ |
| Content-Proxy | $\Rightarrow_{content}$ |
| Container-Proxy | $\Rightarrow_{container}$ |
| struktureller Proxy | $\Rightarrow_{struct}$ |

Tabelle 3: Proxy-Arten und die dazugehörigen Basis-Matchingrelationen

### 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.

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 zu entnehmen. [1]

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

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.

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.

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

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*.

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 zu entnehmen. [2]

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

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

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 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*.

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 zu entnehmen. [3]

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 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

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

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

Ein struktureller Proxy wird für einen *required Typ* $T$ erzeugt. Als Basis für diesen Proxy-Generator fungiert der *StructuralTypeMatcher*.

Für die Generierung eines solchen Proxies vom Typ $T$ muss sichergestellt werden, dass alle in $T$ enthaltenen Methoden durch ein oder mehrere *provided Typen* innerhalb der gesamten Bibliothek $L$ gematcht werden. Die folgende Funktion *cover* beschreibt daher eine Menge von Mengen von *provided Typen*, die für die Erzeugung eines *strukturellen Proxies* für $T$ verwendet werden können.

$$cover(T, L) := \left\{ \{P_1, ..., P_n\} \;\middle|\; \begin{array}{l} P_1 \in L \wedge ... \wedge P_n \in L \wedge \\ methoden(T) = structM(T, P_1) \cup \\ \ldots \cup structM(T, P_n) \wedge \\ structM(T, P_1) \neq \emptyset \wedge \\ \ldots \wedge structM(T, P_n) \neq \emptyset \end{array} \right\}$$

Ausgehend von einer Bibliothek $L$ kann ein *strukureller Proxy* zum *require Typ* $T$ aus einer Menge von *provided Typen* $P$ mit $P \in cover(T, L)$ generiert werden.