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{}

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. Dabei handelt es sich um Produktionsregeln einer Attributgrammatik. Die dazugehörigen Attribute sind der Tabelle 2 zu entnehmen.[1]

Regel	Erläuterung
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{INJ_1, ..., INJ_m$ $MDEL_1, ..., MDEL_k\}$	Ein Proxy wird für ein Typ T mit einer Mengen von *required Typen* $P = \{P_1, ..., P_n\}$, einer Menge von Injektionen sowie einer Menge von Methoden-Delegationen erzeugt.
INJ ::= `inject` P_i `in` f	Eine Injektion besteht aus einem Typnamen P_i und einem Feldnamen f einer Feld-Deklaration.
$MDEL$ ::= $CALLM \rightarrow DELM$	Eine Methodendelegation besteht aus einer aufgerufenen Methode und aus einem Delegationsziel.
$CALLM$ ::= $m(CP_1, ..., CP_n) : CR$	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 $\{DP_1, ..., DP_n\}$.
$REF ::= P_i$	Die erste Variante einer Refernez besteht lediglich aus einem Typ P_i.
$REF ::= P_i\,.f$	Die zweite Variante einer Refernez besteht aus einem Typ P_i und einem Feldnamen f.

Tabelle 1: Grammatikregeln mit Erläuterungen für die Definition eines Proxies

[1] Die Notation $NT\,.*$ in der Spalte *Attribute* beschreibt eine Key-Value-Liste aller Attribute des Nonterminals NT, wobei der Attributname als Key und dessen Wert als Value innerhalb der List verwendet wird.

Regel	Attribute
$PROXY$ ::= `proxy for` T `with` $[P_1, ..., P_n]$ $\{INJ_1, ..., INJ_m$ $MDEL_1, ..., MDEL_k\}$	`type` $= T$ `targets` $= [P_1, ..., P_n]$ `injections` $= [INJ_1.*, ..., INJ_m.*]$ `delegations` $= [MDEL_1.*, ..., MDEL_k.*]$
INJ ::= `inject` P `in` f	`target` $= P$ `field` $= f$

$MDEL$ ::=
$CALLM \rightarrow DELM$
	`cmName` $= CALLM$`.mName`
	`cPT` $= CALLM$`.paramTypes`
	`cRT` $= CALLM$`.returnType`
	`dmName` $= DELM$`.mName`
	`dPT` $= DELM$`.paramTypes`
	`dRT` $= DELM$`.returnType`
	`posModi` $= DELM$`.posModi`
	`target` $= DELM$`.target`
	`delTyp` $= DELM$`.delTyp`

$CALLM$ ::= $m(CP_1, ..., CP_n) : CR$	`mName` $= m$ `paramTypes` $= [CP_1, ..., CP_n]$ `returnType` $= CR$

$DELM$ ::=
$TREF.n(DP_1, ..., DP_n) : DR$
	`mName` $= n$
	`paramTypes` $= [DP_1, ..., DP_n]$
	`returnType` $= DR$
	`posModi` $= [0, ..., n-1]$
	`target` $= TREF$`.target`
	`delTyp` $= TREF$`.delTyp`

$DELM$ ::= `posModi`$(I_1, ..., I_n)$
$TREF.n(DP_1, ..., DP_n) : DR$
	`mName` $= n$
	`paramTypes` $= [DP_1, ..., DP_n]$
	`returnType` $= DR$
	`posModi` $= [I_1, ..., I_n]$
	`target` $= TREF$`.target`
	`delTyp` $= TREF$`.delTyp`

$TREF$ ::= P	`target` $= P$ `delTyp` $= P$
$TREF$ ::= $P.f$	`target` $= P$ `delTyp` $= feldTyp(f, P)$

Tabelle 2: Grammatikregeln mit Attributen für die Definition eines Proxies

2.1 Erläuterung der Semantik von Proxies

Der Proxy wird zum Zweck der Konvertierung für den Typ T erzeugt. Das bedeutet, dass ein solcher Proxy überall dort verwendet werden kann, wo der Typ T erwartet wird. Der Typ T wird auch als *Source-Typ* bezeichnet. Die Typen aus der Menge der P werden als *Target-Typen* bezeichnet.

2.1.1 Injektion

Die Injektion innerhalb eines Proxies, hat zur Folge, dass der Typ aus der Felddeklaration mit dem Feldnamen f in den Typ P_1 konvertiert wird. Dieser Konverierung erfolgt wiederum über einen Proxy.

```
proxy for FireState with [boolean]{
        inject boolean in isActive
}
```

2.1.2 Methoden-Delegation

Eine Methoden-Delegation beschreibt, wie ein Methodenaufruf delegiert wird. Dabei wird zwischen der aufgerufenen und der delegierten Methoden unterschieden. Die Deklaration einer aufgerufene Methode ist immer im *Source-Typ* zu finden. Die Deklaration der delegierten Methoden ist in dem Typ aus der Target-Referenz zu finden. Hierbei kann es sich um einen *Target-Typen* oder um den Typ eines Feldes aus einem *Target-Typ* handeln.

```
proxy for MedicalFireFighter with [Doctor]{
        heal(Injured, MedCabinet):void → Doctor.heal(Patient,
            Medicine):void
}
```

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

Die Parameter der aufgerufenen Methoden werden dabei in die Parameter der Methode aus dem Delegationsziel konvertiert. Nach der Ausführung der Methode aus dem Delegationsziel wird der Rückgabetyp jener Methode in den Rückgabetyp der aufgerufenen Methode konvertiert.

Bezüglich der Parameter kann es vorkommen, dass neben einer Konvertierung der Typen aus eine Anpassung hinsichtlich der Position vorgenommen

werden muss. Ein Proxy für den Typ `MedicalFireFighter`, der mit dem *Target-Typ* `InverseDoctor` erzeugt wird, gibt ein Beispiel für dieses Szenario.

Im Vergleich zum Typ `Doctor` aus einem der beiden obigen Beispiele, ist im Typ `InverseDoctor` ebenfalls eine Methode `heal` deklariert. Allerdings ist die Reihenfolge der Parameter umgekehrt. Wenn ein Proxy für den Typ `MedicalFireFighter`, der mit dem *Target-Typ* `InverseDoctor` erzeugt wird, muss die Reihenfolge der Parameter aus der aufgerufenen Methode der Reihenfolge der Parameter im Delegationsziel angepasst werden. Dies erfolgt über die nach dem Schlüsselwort `posModi` angegebenen Indizes. Dabei beschreibt der Index die Position des Parameters aus der aufgerufenen Methode. Die Position des Index ist mit der Position des Parameters aus der Methode des Delegationsziels gleichzusetzen.

```
proxy for PatientMedicalFireFighter with [InverseDoctor]{
        heal(Patient, MedCabinet):void → posModi(1,0)
            InverseDoctor.heal(Medicine, Patient):void
}
```

$$\frac{cPT.\texttt{len} = dPT.\texttt{len} \wedge cPT.\texttt{len} = 0}{matchedParams(cPT, dPT, pos, \Rightarrow_{PM})}$$

$$\frac{\forall i \in \{0, ..., pos.\texttt{len} - 1\}.cPT[pos[i]] \Rightarrow_{PM} dPT[i]}{matchedParams(cPT, dPT, pos, \Rightarrow_{PM})}$$

$$\frac{D.\texttt{cRT} \Rightarrow_{RM} D.\texttt{dRT} \wedge matchedParams(D.\texttt{cPT}, D.\texttt{dPT}, D.\texttt{posModi}, \Rightarrow_{PM})}{matchedMDEL(D, \Rightarrow_{RM}, \Rightarrow_{PM})}$$

Eine Methoden-Delegation der Form
$cm(CP_1, ..., CP_n) : CR \rightarrow posModi(I_i, ..., I_n)dm(DP_1, ..., DP_n) : DR$
beschreibt, dass der Aufruf der Methoden cm mit den Parametern $CP_1, ..., CP_n$ an die Methode dm delegiert wird. Das bedeutet, dass anstelle der Methode cm die Methode dm mit den Parametern $CP_1, ..., CP_n$ unter Berücksichtigung der über das Schlüsselwort `posModi` angegebenen Reihenfolge aufgerufen wird. Dies funktioniert aber nur wenn für alle Parameter $CP_k \in \{CP_1, ..., CP_n\}$ mit $1 <= k <= n$ Folgendes gilt:

$CP_k \Rightarrow_{spec} DP_{I_k} \vee CP_k \Rightarrow_{exact} DP_{I_k}$

Ist diese Voraussetzung für eines der Paare CP_k und DP_{I_k} nicht erfüllt, muss der Typ CP_k in den Typ DP_{I_k} konvertiert werden. Diese Konvertierung erfolgt über einen Proxy, der auf der Basis der passenden Matching-Relation zwischen CP_k und DP_{I_k} ($CP_k \Rightarrow DP_{I_k}$) erzeugt wird.

Bezogen auf das Beispiel von oben, kann der Parameter-Typ `Patient` ohne Probleme in die Delegations-Methode für den erwarteten Parameter-Typen `Patient` verwendet werden, da gilt `Patient` $\Rightarrow_{spec}$ `Patient`. Bezüglich der anderen Parameter-Typen `MedCabinet` aus der aufgerufenen Methoden und `Medicine` aus der delegierten Methode, gilt jedoch keine der oben genannten Bedingungen. Allerdings gilt: `MedCabinet` $\Rightarrow_{container}$ `Medicine`. Und da die Matching-Relation $\Rightarrow_{container}$ wird ein *Container-Proxy* erzeugt. gilt

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.

Ein *Sub-Proxy* enthält keinerlei Injektionen, was für einen Proxy P durch durch folgende Regel beschrieben wird.

$$\frac{P.\texttt{injections.len} = 0}{noInjections(P)}$$

Weiterhin enthält ein *Sub-Proxy* genau einen Target-Typ. Für einen Proxy P wird dieser Sachverhalt durch die folgende Regel dargestellt.

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

Darüber hinaus enthält ein *Sub-Proxy* P eine bestimmte Menge von Methoden-Delegationen. Dabei gilt, dass die aufgerufenen Methode und die Delegati-

onsmethode einer Methoden-Delegation denselben Namen haben.

$$\frac{DEL.\texttt{cmName} = DEL.\texttt{dmName}}{nominalDel(DEL)}$$

Die aufgerufene Methode muss dabei im Source-Typ deklariert sein und die Delegationsmethode im Target-Typ.

$$\frac{\exists m(P_1, ..., P_n) : R \in methoden(P.\texttt{type}).DEL.\texttt{cmName} = m}{simpleCallMethod(DEL, P)}$$

$$\frac{\exists m(P_1, ..., P_n) : R \in methoden(P.\texttt{targets}[0]).DEL.\texttt{dmName} = m}{simpleDelMethod(DEL, P)}$$

Zusätzlich müssen die Attribute `target` und `delTyp` der Delegationsmethode mit dem Target-Typ des *Sub-Proxies* identisch sein.

$$\frac{DEL.\texttt{target} = DEL.\texttt{delTyp} \wedge DEL.\texttt{target} = P.\texttt{targets}[0]}{simpleDelTarget(DEL, P)}$$

Die oben genannten Kriterien werden in folgender Regel zusammengefasst:

$$\frac{\begin{array}{c} nominalDel(DEL) \wedge simpleCallMethod(DEL, P) \wedge \\ simpleDelMethod(DEL, P) \wedge simpleDelTarget(DEL, P) \end{array}}{subDelegation(DEL, P)}$$

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

$$\frac{\begin{array}{l} |methoden(P.targets[0])| = |P.delegations| \wedge \\ \quad \forall m(P_1, ..., P_n) : R \in methoden(P.targets[0]).\exists DEL \in P.delegations. \\ \qquad\qquad m = DEL.dmName \wedge subDelegation(DEL, P) \end{array}}{subDelegations(P)}$$

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

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

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*.

Ein *Content-Proxy* enthält wiederum keinerlei Injektionen. Weiterhin enthält ein *Content-Proxy*, wie auch der *Sub-Proxy*, genau einen Target-Typ. Ebenfalls identisch zum *Sub-Proxy* ist, dass es sich in den Methoden-Delegationen bei den aufgerufenen Methoden nur um Methoden handeln darf, die im Typ T oder dessen Super-Typ deklariert wurden.

In Bezug auf das Attribut `target` und das Attribut `delTyp` innerhalb einer solchen Methoden-Delegation dürfen diese beiden Typen im *Content-Proxy* nicht identisch sein. Vielmehr muss für das Attribut `delTyp` und den Source-Typ T ein Matching der Form $T \Rightarrow_{internCont} \texttt{delTyp}$ gelten. Daher gilt für den *Content-Proxy* folgende Regel.

$$\frac{\begin{array}{l}\forall i \in \{0, ..., P.\texttt{delegations.len} - 1\}. \\ \quad P.\texttt{delegations}[i].\texttt{target} \neq P.\texttt{delegations}[i].\texttt{delTyp} \wedge \\ \qquad\qquad P.\texttt{type} \Rightarrow_{internCont} P.\texttt{delegations}[i].\texttt{delTyp}\end{array}}{contentTarget(P)}$$

Für welche

Zuletzt ist zu beachten, dass es nicht mehr Methoden-Delegationen als Methoden im Target-Typ des Proxies gibt. All diese Regeln bzgl. der Methoden-Delegationen eines *Sub-Proxies* werden in folgender Regel zusammengefasst.

$$\frac{\begin{array}{l}P.\texttt{delegations.len} = |methoden(P.\texttt{targets}[0])| \wedge \\ \qquad subDelMeth(P) \wedge simpleTarget(P) \wedge callMeth(P)\end{array}}{subDelegations(P)}$$

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

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

3.0.3 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. Dazu werden die Grammatikregeln aus Abschnitt 2, um Nebenbedingungen erweitert, die Tabelle 4 zu entnehmen sind.

Regel	**Nebenbedingungen**
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{INJECTION_1 ... INJECTION_m\ MDEL_1 ... MDEL_k\}$	$\{P_1, ..., P_n\} \in cover(T, L)$ $\{INJECTION_1, ..., INJECTION_m\} = \emptyset$ $\forall MD \in \{MDEL_1, ..., MDEL_k\}.MD$`.target` $\in \{P_1, ..., P_n\} \wedge MD$`.cmethode` $\in methoden(T)$ $\forall M \in methoden(T).\exists MD \in \{MDEL_1, ..., MDEL_k\}.MD$`.cmethode` $= M$
$INJECTION$ ::= `inject` P_i `in` f	

$MDEL$ ::= $CALLM \rightarrow DELM$	`cmethode` $= CALLM$`.methode`
	`dmethode` $= DELM$`.methode`
	`target` $= DELM$`.target`

$CALLM$ ::= $m(CP_1, ..., CP_n) : CR$	`methode` $= m(CP_1, ..., CP_n) : CR$ `paramTypen` $= \{CP_1, ..., CP_n\}$ `returnTyp` $= CR$
$DELM$ ::= $TREF$`.` $n(DP_1, ..., DP_n) : DR$	`methode` $= n(DP_1, ..., DP_n) : DR$ `paramTypen` $= \{DP_1, ..., DP_n\}$ `returnTyp` $= DR$ `target` $= TREF$`.typ`

Tabelle 4: Grammatikregeln und Nebenbedingungen für die Definition eines strukturellen Proxies

Beispiel: Bezogen auf das Beispiel aus Abschnitt **??** soll ein *struktureller Proxy* für den *required Typ* `MedicalFireFighter` aus den *provided Typen*

`FireFighter` und `Doctor` generiert werden. Dazu gelte folgende Bedingung:

$$\{\texttt{FireFighter}, \texttt{Doctor}\} \in \mathit{cover}(\texttt{MedicalFireFighter}, \mathit{ExampLe})$$

Die Generierung der *strukturellen Proxies* erfolgt dann in folgenden Schritten, beginnend mit dem Nichtterminal $\mathit{STRUCTPROXY}$. Bei jedem Schritt werden ein oder mehrere Nichtterminal ersetzt und die Attribute aus den Nebenbedingungen soweit wie möglich zugewiesen.

$$\mathit{STRUCTPROXY} \mid \mathit{typ}(\mathit{STRUCTPROXY}) = \texttt{MedicalFireFighter}$$

$$\begin{array}{l|l} \texttt{structproxy for MedicalFireFighter} & \mathit{typ}(\mathit{TARGET}_1) = \texttt{FireFighter} \\ \{\mathit{TARGET}_1 & \mathit{typ}(\mathit{TARGET}_2) = \texttt{Doctor} \\ \mathit{TARGET}_2\} & \end{array}$$

$$\begin{array}{l|} \texttt{structproxy for MedicalFireFighter} \\ \texttt{\{FireFighter\{} \\ \mathit{MDEL}_1\texttt{\}} \\ \texttt{Doctor\{} \\ \mathit{MDEL}_2\texttt{\}\}} \end{array}$$

$$\begin{array}{l|} \texttt{structproxy for MedicalFireFighter} \\ \texttt{\{FireFighter\{} \\ \mathit{CALLM}_1 \rightarrow \mathit{DELM}_1\texttt{\}} \\ \texttt{Doctor\{} \\ \mathit{CALLM}_2 \rightarrow \mathit{DELM}_2\texttt{\}\}} \end{array}$$

3.0.4 Container-Proxy

3.0.5 Content-Proxy

$$\frac{\mathit{INJS}\,.\texttt{len} = 0}{\mathit{contentInjections}(\mathit{INJS})}$$

$$\frac{\mathit{TARS}\,.\texttt{len} = 1 \wedge T' = \mathit{TARS}[0]}{\mathit{contentTargets}(\mathit{TARS},\, T,\, T')}$$

$$\frac{\mathit{dTypes}\,.\texttt{len} = 0 \wedge \mathit{cTypes}\,.\texttt{len} = 0}{\mathit{contentParamTypes}(\mathit{dTypes}, \mathit{cTypes})}$$

$$\frac{\forall i \in \{0, ..., dTypes.\texttt{len} - 1\}.match_{simple}(dTypes[i], cTypes[i])}{contentParamTypes(dTypes, cTypes)}$$

$$\frac{DELS.\texttt{len} > 0 \wedge \forall i \in \{0, ..., DELS.\texttt{len} - 1\}.contentDelegation(DELS[i], T, T')}{contentDelegations(DELS, T, T')}$$

$$\frac{\begin{array}{l}\exists m(P_1, ..., P_n) : R \in methoden(T).DEL.\texttt{cmName} = m \wedge \\ \quad DELS[i].\texttt{dmName} = m \wedge DELS[i].\texttt{delTyp} = DELS[i].\texttt{target} \wedge DELS[i].\texttt{target} = T' \wedge \\ \qquad match_{simple}(DELS[i].\texttt{creturnType}, DELS[i].\texttt{dreturnType}) \wedge \\ \qquad\qquad contentParamTypes(d.\texttt{dParamTypes}, d.\texttt{cParamTypes})\end{array}}{contentDelegation(DEL, T, T')}$$

$$proxy_{content}(T, T') := \left\{ P \;\middle|\; \begin{array}{l} match_{content}(T, T') \wedge \\ P.\texttt{type} = T \wedge \\ contentTargets(P.\texttt{targets}, T, T') \\ contentInjections(P.\texttt{injections}) \wedge \\ contentDelegations(P.\texttt{delegations}, T, T') \end{array} \right\}$$

3.0.6 Sub-Proxy

$$\frac{INJS.\texttt{len} = 0}{subInjections(INJS)}$$

$$\frac{TARS.\texttt{len} = 1 \wedge T' = TARS[0]}{subTargets(TARS, T, T')}$$

$$\frac{dTypes.\texttt{len} = 0 \wedge cTypes.\texttt{len} = 0}{subParamTypes(dTypes, cTypes)}$$

$$\frac{\forall i \in \{0, ..., dTypes.\texttt{len} - 1\}.match_{simple}(dTypes[i], cTypes[i])}{subParamTypes(dTypes, cTypes)}$$

$$\frac{\begin{array}{l}\forall m(P_1, ..., P_n) : R \in methoden(T').\exists d \in DELS. \\ \quad d.\texttt{cmName} = m \wedge d.\texttt{dmName} = m \wedge d.\texttt{delTyp} = d.\texttt{target} \wedge d.\texttt{target} = T' \wedge \\ \qquad match_{simple}(d.\texttt{creturnType}, d.\texttt{dreturnType}) \wedge \\ \qquad\quad subParamTypes(d.\texttt{dParamTypes}, d.\texttt{cParamTypes})\end{array}}{subDelegations(DELS, T, T')}$$

$$proxy_{sub}(T, T') := \left\{ P \;\middle|\; \begin{array}{l} match_{spec}(T,\, T') \wedge \\ P\,.\texttt{type} = T \wedge \\ subTargets(P\,.\texttt{targets},\, T,\, T') \\ subInjections(P\,.\texttt{injections}) \wedge \\ subDelegations(P\,.\texttt{delegations},\, T,\, T') \end{array} \right\}$$

3.0.7 Simple-Proxy

$$\frac{T \Rightarrow_{exact} T' \vee T \Rightarrow_{gen} T'}{match_{genExact}(T,\, T')}$$

$$\frac{T \Rightarrow_{exact} T' \vee T \Rightarrow_{spec} T'}{match_{specExact}(T,\, T')}$$

$$\frac{INJS\,.\texttt{len} = 0}{noInjections(INJS)}$$

$$\frac{TARS\,.\texttt{len} = 1 \wedge T' = TARS[0]}{singleTarget(TARS,\, T')}$$

$$\frac{superTypes\,.\texttt{len} = 0 \wedge subTypes\,.\texttt{len} = 0}{simpleParamTypes(superTypes, subTypes)}$$

$$\frac{\begin{array}{l}\forall i \in \{0, ..., D\,.\texttt{cParamTypes.len} - 1\}. \\ \quad match_{specExact}(D\,.\texttt{cParamTypes}[i], D\,.\texttt{dParamTypes}[i])\end{array}}{simpleParamTypes(D)}$$

$$\frac{match_{genExact}(D\,.\texttt{creturnType}, D\,.\texttt{dreturnType})}{simpleReturnTypes(D)}$$

$$\frac{\begin{array}{r} D\,.\texttt{cmName} = D\,.\texttt{dmName} \wedge D\,.\texttt{delTyp} = D\,.\texttt{target} \wedge D\,.\texttt{target} = T' \wedge \\ simpleReturnTypes(D) \wedge simpleParamTypes(D) \end{array}}{simpleDelegation(D,\, T,\, T')}$$

$$\frac{\begin{array}{l}\forall m(P_{c1},...,P_{cn}) : R_c \in methoden(T).\exists d \in DELS. \\ \quad callMethod(m,[P_{c1},...,P_{cn}],R_c,d)\wedge \\ \qquad \exists m(P_{d1},...P_{dn}) : R_d \in methoden(T').delMethod(m,[P_{d1},...,P_{dn}],R_d,d)\end{array}}{simpleDelegations(DELS,\,T,\,T')}$$

$$\frac{\begin{array}{l}\forall m(P_1,...,P_n) : R \in methoden(T).\exists d \in DELS. \\ \quad d.\texttt{cmName} = m \wedge d.\texttt{dmName} = m \wedge d.\texttt{delTyp} = d.\texttt{target} \wedge d.\texttt{target} = T'\wedge \\ \qquad\qquad match_{simple}(d.\texttt{creturnType}, d.\texttt{dreturnType})\wedge \\ \qquad\qquad\quad simpleParamTypes(d.\texttt{dParamTypes}, d.\texttt{cParamTypes})\end{array}}{simpleDelegations(DELS,\,T,\,T')}$$

$$proxy_{simple}(T,T') := \left\{ P \;\middle|\; \begin{array}{l} match_{genExact}(T,\,T')\wedge \\ P.\texttt{type} = T\wedge \\ singleTarget(P.\texttt{targets},\,T') \\ noInjections(P.\texttt{injections})\wedge \\ simpleDelegations(P.\texttt{delegations},\,T,\,T') \end{array} \right\}$$

Ein *Simple-Proxy* kann sowohl auf der Basis des *ExactTypeMatchers* als auch auf Basis des *GenTypMatchers* generiert werden. Die für den *Simple-Proxy* relevanten Grammatikregeln aus Abschnitt 2 werden für die Generierung noch um Nebenbedingungen erweitert. Die vollständige Liste der Regeln und Nebenbedingungen für den *Simple-Proxy* ist Tabelle 5 zu entnehmen.

Regel	**Nebenbedingungen**
$STPROXY ::= NPX$	$typ(STPROXY) = typ(NPX)$ $targetTyp(STPROXY) = targetTyp(NPX)$
$NPX ::=$ `simpleproxy for` P	$targetTyp(NPX) \Rightarrow_{exact} P$ $typ(NPX) = P$ $methoden(NPX) = methoden(P)$
$NPX ::=$ `simpleproxy for` P	$targetTyp(NPX) \Rightarrow_{gen} P$ $typ(NPX) = P$ $methoden(NPX) = methoden(P)$

Tabelle 5: Regeln und Nebenbedingungen für Simple-Proxies

Als Beispiel wird von dem Matching ausgegangen, welches auch bei der Beschreibung des *ExactTypeMatchers* (Abschnitt **??**) angeführt wurde:

$$\texttt{Fire} \Rightarrow_{exact} \texttt{Fire}$$

Der *Simple-Proxy* basiert auf zwei Matchern, von denen Die Basis für den *Simple-Proxy* sind sowohl der •

4 alter kram

4.1 StructuralTypeMatcher

Ein struktureller Proxy für ein *required Interface* R aus einer Menge von *provided Typen* P wird durch folgende Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$STRUCTPROXY ::=$ `structproxy for` R $\{TARGET_1 \ \ldots \ TARGET_n\}$	$typ(STRUCTPROXY) = R$ $methoden(STRUCTPROXY) =$ $cmethoden(TARGET_1) \cup \ldots \cup cmethoden(TARGET_n)$ $methoden(R) = methoden(STRUCTPROXY)$
$TARGET ::=$ $P \ \{MDEL_1 \ \ldots \ MDEL_n\}$	$typ(TARGET) = P$ $cmethoden(TARGET) =$ $cmethode(MDEL_1) \cup \ldots \cup cmethode(MDEL_n)$ $dmethoden(TARGET) =$ $dmethode(MDEL_1) \cup \ldots \cup dmethode(MDEL_n)$ $dmethoden(TARGET) \subseteq methoden(P)$
$MDEL ::=$ $CALLM \rightarrow DELM$	$cmethode(MDEL) = methode(CALLM)$ $dmethode(MDEL) = methode(DELM)$ $paramTargetTyp(DELM) = paramTyp(CALLM)$ $returnTargetTyp(CALLM) = returnTyp(DELM)$
$CALLM ::=$ $m(SP) : STPROXY$	$SR = typ(STPROXY)$ $methode(CALLM) = m(SP) : SR$ $paramTyp(CALLM) = SP$ $targetTyp(STPROXY) = returnTargetTyp(CALLM)$
$DELM ::=$ $n(STPROXY) : R$	$DP = typ(STPROXY)$ $methode(DELM) = n(DP) : R$ $returnTyp(DELM) = R$ $targetTyp(STPROXY) = paramTargetTyp(DELM)$

Tabelle 6: Grammatik für die Definition eines Proxies

Regeln für das Nonterminal $STPROXY$ unterliegen Nebenbedingungen, die teilweise erst unter Zuhilfenahme der folgenden Matcher erfüllt werden können.

4.2 SpecTypeMatcher

Ein Proxy für einen Typ T, der mit einem Target-Typ T' mit $T \Rightarrow_{spec} T'$ erzeugt werden soll, ist ein *Sub-Proxy* und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$NPX ::=$ `subproxy for` P `with` P' $\{NOMMDEL_1$ $\ldots$ $NOMMDEL_n\}$	$targetTyp(NPX) = P'$ $typ(NPX) = P$ $P \Rightarrow_{spec} P'$ $methoden(NPX) = cmethode(NOMMDEL_1) \cup$ $\ldots \cup cmethode(NOMMDEL_n)$ $methoden(NPX) \subseteq methoden(P)$ $methoden(P') \supseteq dmethode(NOMMDEL_1) \cup$ $\ldots \cup dmethode(NOMMDEL_n)$
$NOMMDEL ::=$ $m(SP) : SR \rightarrow$ $m(TP) : TR$	$SP >= TP$ $SR <= TR$ $cmethode(MOMMDEL) = m(SP) : SR$ $dmethode(MOMMDEL) = m(TP) : TR$

Tabelle 7: Regeln und Nebenbedingungen für Sub-Proxies

4.3 ContentTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

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

Für die Relation $\Rightarrow_{internCont}$ gilt dabei:

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

Ein Proxy für einen Typ P, der mit einem Target-Typ P' mit $P \Rightarrow_{content} P'$ erzeugt werden soll, ist ein *Content-Proxy* und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$STPROXY ::=$ `contentproxy for` P `with` P' $\{CEMDEL_1$ $\ldots$ $CEMDEL_n\}$	$typ(STPROXY) = P$ $targetTyp(STPROXY) = P'$ $P \Rightarrow_{content} P'$ $methoden(STPROXY) = cmethode(CEMDEL_1) \cup$ $\ldots \cup cmethode(CEMDEL_n)$ $methoden(STPROXY) \subseteq methoden(P)$ $containerType(CEMDEL_1) = P'$ $\ldots$ $containerType(CEMDEL_n) = P'$
$CEMDEL ::=$ $CECALLM \rightarrow$ $f.CEDELM$	$f : FT \in felder(containerType(CEMDEL))$ $methode(CEDELM) \in methoden(FT)$ $paramTargetTyp(CEDELM) = paramTyp(CECALLM)$ $returnTargetTyp(CECALLM) = returnTyp(CEDELM)$
$CECALLM ::=$ $m(SP) : NPX$	$paramTyp(CECALLM) = SP$ $SR = typ(NPX)$ $targetTyp(NPX) = returnTargetTyp(CECALLM)$ $methode(CECALLM) = m(SP) : SR$
$CEDELM ::=$ $m(NPX) : TR$	$returnTyp(CEDELM) = TR$ $TP = typ(NPX)$ $targetTyp(NPX) = paramTargetTyp(CEDELM)$ $methode(CEDELM) = m(TP) : TR$

Tabelle 8: Regeln und Nebenbedingungen für Contentproxies

4.4 ContainerTypeMatcher

Die Matchingrelation für diesen Matcher wird durch folgende Regel beschrieben:

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

Ein Proxy für einen Typ P, der mit einem Target-Typ P' mit $P \Rightarrow_{container} P'$ erzeugt werden soll, ist ein *Container-Proxy* und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	Nebenbedingungen
$STPROXY$ ::= `containerproxy for` P `with` P' $\{f = NPX\}$	$targetTyp(STPROXY) = P'$ $typ(STPROXY) = P$ $P \Rightarrow_{container} P'$ $f : FT \in felder(P)$ $targetTyp(NPX) = P'$ $typ(NPX) = FT$

Tabelle 9: Regeln und Nebenbedingungen für Container-Proxies

5 Erweiterung um einen DVMatcher

Die o.g. Struktur für die Definition von Typen wird die Definition von *provided Typen* erweitert.

Regel	Erläuterung
PD ::= `provided` T `extends` T' $\{FD^*MD^*FCD?\}$	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 und einer optionalen Definition eines factory Typen.
FCD ::= `factory` T $\{$ $FD^*MD^*\}$	Die Definition eines factory Typen besteht aus dem Namen des Typen T sowie mehreren Feld- und Methodendeklarationen.

Tabelle 10: Erweiterte Struktur für die Definition einer Bibliothek von Typen

Darüber hinaus wird folgende Funktion definiert:

$$fabriken(T) := \{\ F \mid F \text{ ist ein } \textit{factory Typ}, \text{ der in } T \text{ definiert wurde}\ \}$$

Weiterhin muss die Struktur für die Definition von Proxies um eine weitere Definition für einen *Single-Target-Proxy* erweitert werden.

Regel	Erläuterung
$STPROXY$ ::= `dvproxy for` P `with` F `on m(`P'`):`P	Ein *DV-Proxy* ist ein Single-Target-Proxy, der für ein *provided Typ* P erzeugt wird. Die Methodenaufrufe auf diesem Proxy werden an das Objekt delegiert, welches über die Methode m des Factory-Typen F aus dem Target-Typen P' erzeugt wird.

Tabelle 11: Erweiterung der Struktur für die Definition eines Proxies

Die Matchingrelation $\Rightarrow_{dv}$ wird über folgende Regel beschrieben:

$$\frac{\exists F \in fabriken(T).\exists m(T') : T}{T \Rightarrow_{dv} T'}$$

Darüber hinaus müssen einige der oben beschriebenen Regeln angepasst werden, damit der *ContainerTypeMatcher*, der *ContentTypeMatcher* und der *StructuralTypeMatcher* den *DVMatcher* verwenden:

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

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

Ein Proxy für einen Typ P, der mit einem *Target-Typ* P' mit $P \Rightarrow_{dv} P'$ erzeugt werden soll, ist ein DV-Proxy und wird durch die folgenden Regeln und Nebenbedingungen beschrieben:

Regel	**Nebenbedingungen**
$STPROXY$ ::= `dvproxy for` P `with` F `on m(`P'`):`P	$targetTyp(STPROXY) = P'$ $typ(STPROXY) = P$ $P \Rightarrow_{dv} P'$ $F \in fabriken(P)$

Tabelle 12: Regeln und Nebenbedingungen für DV-Proxies