1 Explorationsalgorithmus

Mit diesen Voraussetzungen kann eine Komponente entwickelt werden, welche die Erwartungen der nachfragenden Komponente mit den bestehenden Funktionalitäten der angebotenen Komponenten zusammenbringt. In Abbildung 1 ist dies als Explorationskomponente dargestellt. Die Abhängigkeiten zu der nachfragenden und den angebotenen Komponenten ist nicht direkt vorhanden, da sie lediglich durch reflexive Aufrufe zur Laufzeit zustande kommen.

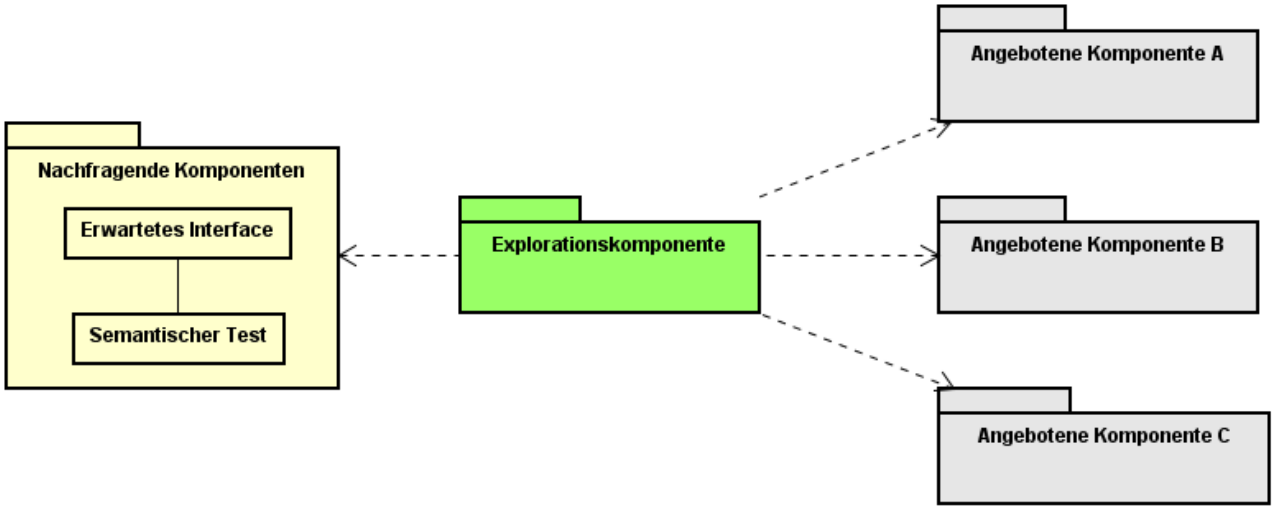

Abbildung 1: Allgemeiner Aufbau des System mit der Explorationskomponente

Um die Explorationskomponente anzusprechen, muss der Entwickler eine Instanz der Klasse DesiredComponentFinder, die von der Explorationskomponente bereitgestellt wird, erzeugen. Dabei müssen dem Konstruktor dieser Klasse zwei Parameter übergeben werden. Der erste Parameter ist eine Liste aller angebotenen Interfaces. Der zweite Parameter ist eine java.util.Function, über die die konkreten Implementierungen der angebotenen Interfaces ermittelt werden können. Die Suche wird mit dem Aufruf der Methode getDesiredComponent gestartet, welcher das erwartete Interface als Parameter übergeben werden muss. Somit kann ein Objekt der Klasse DesiredComponentFinder für mehrere Suchen mit unterschiedlichen erwarteten Interfaces verwendet werden.

Zu erwähnen ist noch, dass die in der nachfragenden Komponente spezifizierten Erwartungen

mitunter nur durch eine Kombination von angebotenen Komponenten erfüllt werden können. Aus diesem Grund wird innerhalb der Explorationskomponente eine so genannte benötigte Komponente erzeugt, in der das Zusammenspiel einer solchen Kombination von angebotenen Komponenten verwaltet wird. Ein solches Szenario ist Abbildung 2 zu entnehmen.

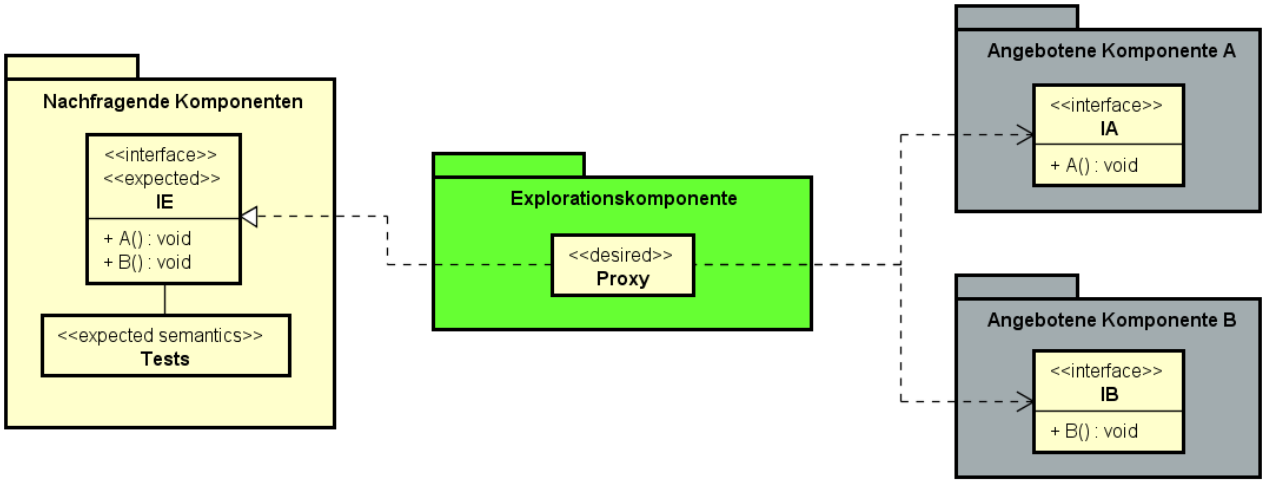

Abbildung 2: Kombination von angebotenen Komponenten

Die Suche nach einer benötigten Komponente innerhalb der Explorationskomponente erfolgt in zwei Schritten. Im ersten Schritt werden die angebotenen Interfaces hinsichtlich ihrer Struktur mit dem erwarteten Interface abgeglichen. Im zweiten Schritt werden die Ergebnisse aus dem ersten Schritt hinsichtlich der semantischen Tests überprüft. Dieser mehrstufige Ansatz baut auf der Arbeit von Hummel [**?**] auf.

1.1 1. Stufe - Strukturelle Übereinstimmung

Wie in [**?**] wird in der ersten Stufe der Suche versucht die angebotenen Interfaces herauszusuchen, die strukturell mit dem erwarteten Interface übereinstimmen. Zu diesem Zweck wird ein Structural-Type-Matcher verwendet, der in Abschnitt 1.1.1 beschrieben wird. Darüber hinaus werden weitere Type-Matcher verwendet (siehe Abschnitte 1.1.2-**??**), die das Matching zweier Typen auf der Basis der Beziehung, in der diese beiden Typen zueinander stehen, feststellen. Allgemein beschrieben, kann durch jeden dieser Type-Matcher festgestellt werden, ob sich ein

Typ in einen anderen Typ konvertieren lässt.

Die Konvertierung erfolgt zur Laufzeit über die Erzeugung von Proxies, die ihre Methodenaufrufe delegieren. So wird bspw. bei der Konvertierung eines Objektes von TypA in ein Objekt von TypB ein Proxy-Objekt für TypB erzeugt, welches die Methodenaufrufe auf dem Objekt von TypA delegiert (vgl. Abbildung 2).

Hummel hatte hierzu bereits auf einige Matcher von Zaremski und Wing [**?**] zurückgegriffen, die in dieser Arbeit ebenfalls zum Einsatz kommen (siehe Abschnitte 1.1.2-1.1.4). Weiterhin wurde in [**?**] ein Anwendungsfall für einen Matcher skizziert, der in der Lage ist Container-Typen zu ihren enthaltenen Typen zu matchen. Auf diese Idee wird in den Abschnitten **??** und **??** weiter eingegangen. Die Definitionen der Matcher beziehen sich vorrangig auf die Programmiersprache Java, weshalb grundlegend von einer nominalen Typkonformität auszugehen ist.

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

Regel	**Erläuterung**
$L ::= TD^*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \vert RD$	Eine Typdefinition kann entweder die Definition eines provided Typen (PD) oder eines required Typen (RD) sein.
PD ::= provided T extends T' $\{FD^*MD^*\}$	Die Definition eines provided Typen besteht aus dem Namen des Typen T, dem Namen des Super-Typs T' von T sowie mehreren Feld- und Methodendeklarationen.
RD ::= required T $\{MD^*\}$	Die Definition eines required Typen besteht aus dem Namen des Typen T sowie mehreren Methodendeklarationen.
FD ::= f : T	Eine Felddeklaration besteht aus dem Namen des Feldes f und dem Namen seines Typs T.
MD ::= m(T):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: Grammatik für die Definition einer Bibliothek von Typen

Weiterhin sei die Relation $<$ auf Typen durch folgenden Regel definiert:

$$T < T' := \text{provided } T \text{ extends } T\text{'} \in L \vee (\text{provided } T \text{ extends } T\text{''} \in L \wedge T'' < T')$$

Darüber hinaus seien folgende Funktionen definiert:

$$felder(T) := \left\{ f : T' \;\middle|\; f : T\text{'} \text{ ist Felddeklaration von } T \right\}$$

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

$$vererbteMethoden(T, T') := \left\{ m(P) : R \;\middle|\; \begin{array}{l} T < T' \wedge m(P) : R \in methoden(T) \wedge \\ \exists m(P') : R' \in methoden(T'). \\ (P <= P') \wedge (R >= R') \end{array} \right\}$$

Das Matching eines Typs A zu einem Typ B wird durch die asymmetrische Relation $A \Rrightarrow B$ beschrieben. Dabei wird A auch als *Source-Typ* und B als *Target-Typ* bezeichnet.

Ein Proxy beschreibt die Konvertierung einer Menge von Target-Typen $P = \{T_1, ..., T_n\}$ in einen Proxy-Typen X. Die Definition eines Proxies hat dabei folgende Form:

Regel	Erläuterung
$PROXY$ ::= proxy X for PR {$TARGET$*}	Eine Proxy-Definition besteht aus dem Namen des Proxy-Typs X, dem Namen des Typs PR für den der Proxy erzeugt wird. PR kann dabei der Name eines *required Interfaces* oder der eines *provides Interfaces sein.* Weiterhin besteht ein Proxy aus einer Mengen von Targets, die die Basis für den Proxy bilden.
$TARGET$ ::= T target {$MDEL$*AZ*}	Die Definition einer Targets besteht aus dem Namen des Target-Typs T, dem Namen einer Variablen *target* zur Referenzierung des Target-Objektes, sowie einer Menge aus Methodendelegationen und Attributzuweisungen.
AZ ::= f = TR	Eine Attributzuweisung besteht aus dem Namen des Attributfeldes des Proxies f und einer Targetreferenz.
$MDEL ::= CALLM \rightarrow DELT$	Eine Methodendelegation besteht aus einer Methode, die auf dem Proxy aufgerufen wird (CALLM) und einem Delegationsziel (DELT), an das der Aufruf delegiert wird.
$CALLM$ ::= $m(P) : TYPECONV$	Eine aufgerufene Methode besteht aus ihrem Namen m, dem Namen des Parametertyps P und einem Konverter für den Rückgabetyp.
$DELT$ ::= $TR.n(TYPECONV) : R$	Ein Delegationsziel besteht aus einer Referenz auf das Delegationsobjekt (TR), dem Methodennamen n der am Delegationsobjekt aufzurufenden Methoden, sowie einem Konverter Parametertyp und dem Namen des Rückgabetype R der Methode n.
$TYPECONV ::= PROXY \vert T$	Ein Konverter ist entweder wiederum ein Proxy, oder ein Typ T, sofern keine Konvertierung vorgenommen wird.
TR ::= $target \vert target.f$	Eine Targetreferenz ist entweder die Variable für das Target-Objekt (*target*) oder ein Feld mit dem Namen f des Target.Objektes.

Tabelle 2: Grammatik für die Definition eines Proxies

Zusätzlich seien folgenden Funktionen definiert:

$$targets(X) := \left\{ T \;\middle|\; \begin{array}{l} T \text{ ist der Name des Typs einer} \\ \text{Targets von } X \end{array} \right\}$$

$$delegationen(X,T) := \left\{ \begin{array}{l|l} m(SP) : SR & m(SP) : SR \rightarrow target.n(TP) : TR \\ \rightarrow & \text{ist eine Methodendelegation eines} \\ target.n(TP) : TR & \text{Targets } T \text{ mit } T \in targets(X) \end{array} \right\}$$

$$zuweisungen(X,T) := \left\{ f = V \;\middle|\; \begin{array}{l} f = V \text{ ist eine Attributzuweisung eines} \\ \text{Targets } T \text{ mit } T \in targets(X) \end{array} \right\}$$

1.1.1 StructuralTypeMatcher

Ein Ziel dieser Arbeit ist es Typen, die keinerlei Assoziationen zueinander habe, miteinander zu matchen und so zu konvertieren, dass darauf aufbauend die erwartete Semantik überprüfen werden kann. Hierfür soll wie auch in [**?**] die strukturelle Übereinstimmung der beiden Typen genutzt werden. Diesem Zweck dient der StructuralTypeMatcher.

Szenario

Um die grundlegenden Eigenschaften des StructuralTypeMatchers darzustellen, wird von einem Szenario ausgegangen, in dem zwei *provided Interfaces* in Kombination ein *required Interface* erfüllen. Dabei wird von einer Bibliothek L ausgegangen, die zum einen eine Erweiterung der Typen aus dem JDK um die in Listing **??** definierten Typen darstellt.

```
provided Fire extends Object{}

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 MedCabinet extends Object{
        med : Medicine
}
```

```
required MedicalFireFigther {
        void heal( Injured injured, MedCabinet med )
        boolean extinguishFire( Fire fire )
}
```

Listing 1: Bibliothek von Typen

Durch den StructuralTypeMatcher soll zum einen ein Matching der Form $IntubatingFireFighter \Rrightarrow FireFighter$ und $IntubatingFireFighter \Rrightarrow Doctor$ ermittelt werden. Darüber hinaus soll die Kombination aus $FireFighter$ und $Doctor$ in den Typ $IntubatingFireFither$ konvertiert werden. Ein Proxy PTF der aus der Konvertierung der Typen $FireFighter$ und $Doctor$ in den Typ $IntubatingFireFighter$ ist in Listing **??** beschrieben.

```
proxy PTF for MedicalFireFither{
        FireFighter fireFighter {
                extinguishFire(Fire):proxy PFireState for FireState{
                        boolean target{
                                isActive = target
                        }
                } → fireFighter.extinguishFire(Fire):boolean
        }

        Doctor doctor {
                heal(Injured, MedCabinet):void↦ doctor.heal(
                        proxy PPatient for Patient{
                                Injured injured{
                                        heal():void → injured.heal():void

                                }
                        }, proxy PMedicine for Medicine{
                                MedCabinet cab{
```

```
                                        getDescription():String →
                                            cab.med.getDescription():String

                                    }
                            }):void
        }
}
```

Dabei wird beim Aufruf der Methoden *extinguishFire(Fire)* an die Methoden *extinguishFire(Fire)* des Typs $FireFighter$ delegiert. Dabei wird der Parameter vom Typ $Fire$ einfach weitergereicht, ohne dass eine Konvertierung des Parameters ist in diesem Fall nicht notwendig, da der Parameter-Typ der aufgerufenen Methode und der Methode, an die der Aufruf delegiert wird, identisch sind. Dasselbe gilt für die Rückgabewerte der beiden Methoden.

Der Aufruf der Methode *intubate(Injured)* wird an die Methode *intubate(Patient)* des Typs $Doctor$ delegiert. Dabei erfolgt eine weitere Konvertierung des Typs $Injured$ in den Typ $Patient$.

Definition
Das strukturelle Matching zwischen einem *required Interface* R und einem *provided Interface* P ist gegeben, sofern eine Methode aus R zu einer Methode aus P gematcht werden kann. Die Menge der aus R in P gematchten Methoden wird wie folgt beschrieben:

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

Da die Notation es nicht hergibt, ist zusätzlich zu erwähnen, dass die Reihenfolge der Parameter in m und n irrelevant ist.

Die Relation $\Rrightarrow_{egsc}$ wird durch die übrigen Matcher in folgender Form beschrieben:

$$\frac{\begin{array}{r} A \Rrightarrow_{exact} B \wedge A \Rrightarrow_{spec} B \wedge A \Rrightarrow_{gen} B \\ \wedge\, A \Rrightarrow_{container} B \wedge A \Rrightarrow_{content} B \end{array}}{A \Rrightarrow_{egsc} B}$$

Das strukturelle Matching von R und P wird dann durch folgende Regel beschrieben.

Matching (StructuralTypeMatcher)

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

Für die Verwendung von R muss jedoch sichergestellt werden, dass alle darin enthaltenen Methoden durch ein oder mehrere *required Interfaces* innerhalb der gesamten Bibliothek L gematcht werden. Folgende Funktion beschreibt daher eine Menge von *provided Interfaces*, die in Kombination zu allen Methoden von R eine übereinstimmende Methode enthalten.

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

Für R kann die Exploration abgebrochen werden, wenn $cover(R,L) = \emptyset$ gilt.

Die Menge aller Konvertierungsmöglichkeiten einer Menge von *provided Interfaces* $P = \{P_1,...,P_n\}$ in ein *required Interface* R über den StrucutralTypeMatcher wird durch die Funktion $Proxy_{struct}(R,P)$ beschrieben. Dazu sei $singleMDEL(X, MDEL)$ für einen Proxy X und eine Methodendelegation $MDEL$ durch folgende Regel beschrieben.

$$\frac{\begin{array}{l} \forall T \in targets(X).[\exists MDEL \rightarrow N \in delegationen(X,T) \wedge \\ \quad \forall T' \in targets(X).(MDEL \rightarrow N) \notin delegationen(X,T') \vee T' = T] \end{array}}{singleMDEL(X,MDEL)}$$

Konvertierung (StructuralTypeMatcher)

Die Menge von Proxies, die eine Konvertierung einer Menge von *provided Interfaces* $P = \{P_1,...,P_n\}$ in ein *required Interface* R beschreiben, wird durch die folgenden Funktion definiert:

$$Proxy_{struct}(R,P) := \left\{ X \;\middle|\; \begin{array}{l} \forall T \in targets(X).[zuweisungen(R,T) = \emptyset \wedge T \in P \wedge \\ \forall (m(SP) : SR \rightarrow target.n(TP) : TR) \in delegationen(X,T).[\\ SR \in Proxy_{esgc}(SR,TR) \wedge TP \in Proxy_{esgc}(TP,SP) \\ \wedge m(SP) : SR \in methoden(R) \wedge n(TP) : TR \in methoden(T) \\ \wedge singleMDEL(X, m(SP) : SR)]] \end{array} \right\}$$

Für einen Proxy $X \in Proxy_{esgc}(A, B)$ gilt:

$$Proxy_{esgc}(A, B) = \begin{array}{l} Proxy_{exact}(A, B) \cup Proxy_{spec}(A, B) \cup Proxy_{gen}(A, B) \\ \cup Proxy_{container}(A, B) \cup Proxy_{content}(A, B) \end{array}$$

1.1.2 ExactTypeMatcher

Szenario

Dieser Matcher stellt das Matching zweier identischer Typen fest. Das Matching kann erfolgt somit auf nominaler Ebene erfolgen.

Definition

Matching (ExactTypeMatcher)

$$\frac{}{T \Rrightarrow_{exact} T}$$

Konvertierung (ExactTypeMatcher)

Eine Konvertierung eines Typs T in denselben Typ ist praktisch nicht notwendig. Daher gilt:

$$Proxy_{exact}(T, T) = \{T\}$$

1.1.3 GenTypeMatcher

Szenario

Dieser Matcher stellt das Matching zwischen zwei Typen her, die in einer Vererbungsbeziehung stehen. Speziell erlaubt dieser Matcher das Matching eines Supertyps als *Source-Typ* mit einem Subtyp als *Target-Typ*. Ausgehend von den Typen aus Listing **??** wird für dieses Szenario auf

die Typen $Injured$ als Supertyp und $Patient$ als Subtyp zurückgegriffen. Der GenTypeMatcher soll in diesem Fall ein Matching der Form $Injured \Rrightarrow_{gen} Patient$ festellen. In Abbildung **??** ist schematisch dargestellt, wie eine Methode, die auf dem Supertyp $Injured$ aufgerufen wird, an eine Methode des Subtyps $Patient$ delegiert wird. Eine solche Delegation wird aufgrund der Regeln für die Methoden-Deklarationen innerhalb von Sub- und Supertypen ohne Zuhilfenahme eines Proxies erreicht.[1] Daher muss keine Konvertierung des *Target-Typs* in den *Source-Typ* erfolgen. Vielmehr kann der *Target-Typ* überall dort eingesetzt werden, wo der *Super-Typ erwartet wird.*

Definition

Matching (GenTypeMatcher)

$$\frac{B < A}{A \Rrightarrow_{gen} B}$$

Konvertierung (GenTypeMatcher)
Für zwei Typen A und B für die $A \Rrightarrow_{gen} B$ gilt, ist keine Konvertierung von B in A notwendig. Somit gilt:

$$Proxy_{gen}(A, B) = \{B\}$$

1.1.4 SpecTypeMatcher

Szenario
Analog zum GenTypeMatcher stellt der SpecTypeMatcher ebenfalls das Matching zwischen Typen fest, die in einer Vererbungsbeziehung stehen. Allerdings soll durch diesen Matcher der umgekehrte Fall abgebildet werden. Demnach soll ausgehend von den Typen $Injured$ als Supertyp und $Patient$ als Subtyp aus Listing **??** ein Matching der Form $AccidentVictim \Rrightarrow_{spec}$

[1] Die Parameter-Typen müssen Kovarianz und die Rückgabe-Typen Kontravarianz aufweisen. Folglich ist eine Delegation wie in Abbildung **??** aufgrund des Substitutionsprinzips möglich.

Injured ermittelt werden. Eine Verwendung des Typen *Injured* anstelle von *Patient* ist nicht ohne Konvertierung möglich. Daher als Resultat der Konvertierung über diesen Matcher ein Proxytyp *PPatient* erwartet, der Listing **??** entnommen werden kann.

```
proxy PPatient extends Patient{
        Injured injured {
                heal():void → injured.heal():void
        }
}
```

Bei genauerer Betrachtung des *provided Interfaces* *Patient* und *Injured* fällt auf, dass der Subtyp *Patient* eine eigenen Methode deklariert. Bei einer Konvertierung kann diese Methode nicht delegiert werden. Der Aufruf würde dementsprechend fehlschlagen.[2] In Abbildung **??** und Abbildung **??** sind die Methodenaufrufe der beiden angebotenen Methoden des Typs *PPatient* mit ihrer Delegation bzw. Fehlschlag aufgeführt.

Definition

Matching (SpecTypeMatcher)

$$\frac{A < B}{A \Rrightarrow_{spec} B}$$

Konvertierung (SpecTypeMatcher)

Für zwei Typen A und B für die $A \Rrightarrow_{spec} B$ ist die Menge an möglichen Proxytypen wie folgt definiert:

$$Proxy_{spec}(A,B) := \left\{ X \;\middle|\; \begin{array}{l} targets(X) = \{B\} \wedge zuweisungen(X,B) = \emptyset \wedge \\ \forall m(P) : R \in vererbteMethoden(A,B). \\ \exists m(P) : R \rightarrow target.m(P) : R \in delegationen(X,B) \end{array} \right\}$$

[2] Downcast

1.1.5 ContentTypeMatcher

Szenario
In **??**

Definition

Matching (ContentTypeMatcher)

$$\frac{\exists f : T' \in felder(B).A \Rrightarrow_{esg} B}{A \Rrightarrow_{content} B}$$

Die Relation $\Rrightarrow_{esg}$ wird durch die drei zuvor definierten Matcher beschrieben:

$$\frac{A \Rrightarrow_{exact} B \vee A \Rrightarrow_{spec} B \vee A \Rrightarrow_{gen} B}{A \Rrightarrow_{esg} B}$$

Konvertierung (WrappedTypeMatcher)
Für zwei Typen A und B für die $A \Rrightarrow_{content} B$ ist die Menge an möglichen Proxytypen wie folgt definiert:

$$Proxy_{content}(A,B) := \left\{ X \;\middle|\; \begin{array}{l} targets(X) = \{B\} \wedge zuweisung(X,B) = \emptyset \wedge \\ \forall m(SP) : SR \rightarrow target.f.n(TP) : TR \in delegationen(X,B). \\ f \in felder(B) \wedge SR \in Proxy_{esg}(SR,TR) \wedge TP \in Proxy_{esg}(TP,SP) \end{array} \right\}$$

1.1.6 ContainerTypeMatcher

Szenario

Matching (WrapperTypeMatcher)

$$\frac{\exists f : T' \in felder(A).T' \Rrightarrow_{esg} B}{A \Rrightarrow_{container} B}$$

Konvertierung (WrappedTypeMatcher)

Für zwei Typen A und B für die $A \Rrightarrow_{container} B$ ist die Menge an möglichen Proxytypen wie folgt definiert:

$$Proxy_{content}(A,B) := \left\{ X \;\middle|\; \begin{array}{l} targets(X) = \{B\} \wedge delegationen(X,B) = \emptyset \wedge \\ \forall f = T \in zuweisung(X,B). \\ \exists f : T' \in felder(A). T \in Proxy_{esg}(T',B) \end{array} \right\}$$

Ein Beispiel für die Verwendung des Matchers in Bezug auf das o.g. Szenario ist in Anhang **??** zu finden. Außerdem sind dort auch weitere Szenarien aufgeführt, in denen der GenTypeMatcher oder der SpecTypeMatcher als interner Matcher zur Anwendung kommen.