FernUniversität Hagen

Masterarbeit

Evaluation von Heuristiken für die testgetriebene Exploration von Enterprise-Java-Beans

Niels Gundermann

Prüfer: Univ. Prof. Dr. Friedrich Steimann
Zweitprüfer : Dr. Daniela Keller
Betreuer: Univ. Prof. Dr. Friedrich Steimann
Lehrgebiet Programmiersysteme
Fachbereich Informatik

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema *Evaluation von Heuristiken für die testgetriebene Exploration von Enterprise-Java-Beans* selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird. Außerdem räume ich dem Lehrgebiet das Recht ein, die Arbeit für eigene Lehr- und Forschungstätigkeiten auszuwerten und unter Angabe des Autors geeignet zu publizieren.

Neubrandenburg, den 20.11.2021

Niels Gundermann

Abstract

Mit dem Verfahren der testgetriebenen Codesuche ist ein*e Software-Entwickler*in in der Lage bestehenden Code in einem Repository nach vorgegebenen Kriterien zu durchsuchen. Die Kriterien beinhalten dabei Testfälle, die auf den bestehenden Code im Repository angewendet werden. Ausgehend davon, dass eine solche Suche während der Laufzeit innerhalb eines Systems möglich ist, wird die Zeit, die dafür zur Verfügung steht zu einem kritischen Aspekt.

Daher zielt diese Arbeit darauf ab, Heuristiken zu evaluieren, durch die die testgetriebene Codesuche beschleunigt werden kann. Dazu wird die Exploration im Kontext der Arbeit formal beschrieben. Aufbauend auf dieser formalen Beschreibung werden drei Heuristiken vorgestellt, die bei der Exploration in einem bestehenden System evaluiert werden. Das Repository bildet dabei ein EJB-Container mit ca. 900 EJBs innerhalb des Systems.

Die Untersuchungsergebnisse zeigen, dass alle drei Heuristiken - wenn auch mit Abstufungen - das Potential haben, die Exploration zu beschleunigen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

In größeren Software-Systemen ist es üblich, dass mehrere Komponenten miteinander über Schnittstellen kommunizieren. In der Regel werden diese Schnittstellen so konzipiert, dass sie Informationen oder Services anbieten, die von anderen Komponenten abgefragt und benutzt werden können. Dabei wird zwischen der Komponente, welche die Schnittstelle implementiert - als angebotene Komponente - und der Komponente, welche die Schnittstelle nutzen soll - als nachfragende Komponente - unterschieden (siehe Abbildung 1.1).

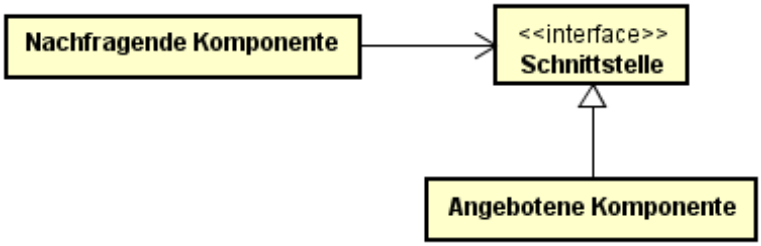


Abbildung 1.1: Abhängigkeiten von nachfragenden und angebotenen Komponenten

Wird von einer nachfragenden Komponente eine Information benötigt, die in dieser Form noch nicht angeboten wird, so wird häufig ein neues Interface für diese benötigte Information erstellt, welches dann passend dazu implementiert wird. Dabei muss neben der Anpassung der nachfragenden Komponente auch eine Anpassung oder Erzeugung der anbietenden Komponente

erfolgen und zusätzlich das neue Interface deklariert werden. Zudem bedingt eine nachträgliche Änderung der neuen Schnittstelle ebenfalls eine Anpassung der drei genannten Artefakte.

In einem großen Software-System mit einer Vielzahl von bestehenden Schnittstellen ist eine gewisse Wahrscheinlichkeit gegeben, dass die Informationen oder Services, die von einer neuen nachfragenden Komponente benötigt werden, in einer ähnlichen Form bereits existieren. Das Problem ist jedoch, dass die manuelle Evaluation der Schnittstellen mitunter sehr aufwendig bis, aufgrund von unzureichender Dokumentation und Kenntnis über die bestehenden Schnittstellen, unmöglich ist.

Weiterhin ist es denkbar, dass ein Software-System auf unterschiedlichen Maschinen verteilt wurde und dadurch Teile des Systems ausfallen können. Das hat zur Folge, dass die Implementierung bestimmter Schnittstellen nicht erreichbar ist. Dadurch, dass eine Schnittstelle durch eine nachfragende Komponente explizit referenziert wird, kann eine solche Komponente nicht korrekt arbeiten, wenn die Implementierung der Schnittstelle nicht erreichbar ist, obwohl die benötigten Informationen und Services vielleicht durch andere Schnittstellen, deren Implementierung durchaus zur Verfügung stehen, bereitgestellt werden könnten.

Dies führt zu der Überlegung, ob eine nachfragende Komponente anstelle der Referenzierung einer Schnittstelle eine Spezifizierung der Schnittstelle vornimmt, anhand derer eine angebotene Komponente, die dieses Spezifikation erfüllt, gefunden werden kann.

Ein solches Vorgehen wird bei der testgetriebene Codesuche (testdriven codesearch - *TDCS*) verfolgt, welche als Basis für diese Arbeit herangezogen wird. Dabei stellt der Entwickler eine Menge von Suchparametern zusammen, die er an eine so genannte Source Engine übergibt. Die Suchparameter sind dabei jedoch stark an dem orientiert, was der Entwickler benötigt und weniger daran, was tatsächlich im Repository vorliegt. Diese Source Engine durchsucht anschließend ein Repository nach Komponenten, die zu den gestellten Suchparametern passen.

Die Suchergebnisse werden aufgelistet und der Entwickler entscheidet letztendlich explizit, welche Komponente verwenden möchte. Die Verwendung der Komponente läuft dann jedoch auf

eine Referenzierung dieser in der nachfragenden Komponente hinaus. Somit arbeiten die Source Engines also nicht zur Laufzeit des Systems, in dem die Komponenten verwendet werden sollen.

In dieser Arbeit soll eine solche Exploration jedoch zur Laufzeit erfolgen, sodass eine explizite Referenzierung der angebotenen Komponente nicht erfolgen muss. Dabei ist die Zeit als Ressource während der Suche nach einer passenden Komponente als knapp anzusehen. Aus diesem Grund werden in dieser Arbeit Heuristiken vorgeschlagen, die ein gezieltes Auffinden einer passenden Komponente ermöglichen und damit die Suche beschleunigen.

1.2 Aufbau dieser Arbeit

Zuerst wird in Kapitel 2 auf den aktuellen Forschungsstand zur *TDCS* eingegangen. Im Anschluss daran wird beschrieben, wie sich die *TDCS* auf einen Ansatz, in dem zur Laufzeit nach Komponenten gesucht wird, eingegangen, um so eine Abgrenzung zu den früheren Arbeiten zu schaffen.

In Kapitel 3 werden die einzelnen Schnritte, die während der Exploration durchgeführt werden, sowie die zu evaluierenden Heuristiken formal beschrieben.

Kapitel 4 gibt einen kurzen Überblick über die Implementierung der in Kapitel 3 genannten Aspekte.

In Kapitel 5 werden die Untersuchungsergebnisse, die unter Anwendung der Heuristiken im Einzelnen und in Kombination zusammengetragen wurden, vorgestellt.

Die Auswertung dieser Ergebnisse erfolgt in Kapitel 6 zusammen mit einer kritischen Betrachtung des in der Arbeit vorgestellten Ansatzes, sowie einer kurzen Betrachtung möglicher Erweiterungen für diesen Ansatz.

Komplettiert wird die Arbeit durch eine kurzen Zusammenfassung der Ergebnisse und einem Ausblick in Kapitel 7.

Kapitel 2

Forschungsziel und Abgrenzung

2.1 Testgetriebene Codesuche

Die Idee der *TDCS* beruht im Grunde auf dem Ziel der Wiederverwendung von Software, welches 1992 von Krueger wie folgt beschrieben wurde: *„Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.“* [Kru92] In der *TDCS* soll dieses Ziel in Verbindung mit dem Prozess der testgetriebenen Software-Entwicklung (testdriven development - *TDD*) erreicht werden. [Hum08]

Bei der *TDCS* werden die jeweiligen Anforderungen an den zu suchenden Source Code durch die Entwickler*innen spezifiziert. Diese werden im Abschluss verwendet, um den relevanten Source Code aus einem Repository zu ermittelt. Darauf aufbauend kann das jeweilige Tool den Entwickler*innen Vorschläge für die Wiederverwendung des bestehenden Codes unterbreiten.

Der Prozess der *TDCS* wurde von Hummel und Janjic grundlegend als Zyklus beschrieben und wie in Abbildung 2.1 dargestellt [HJ13].

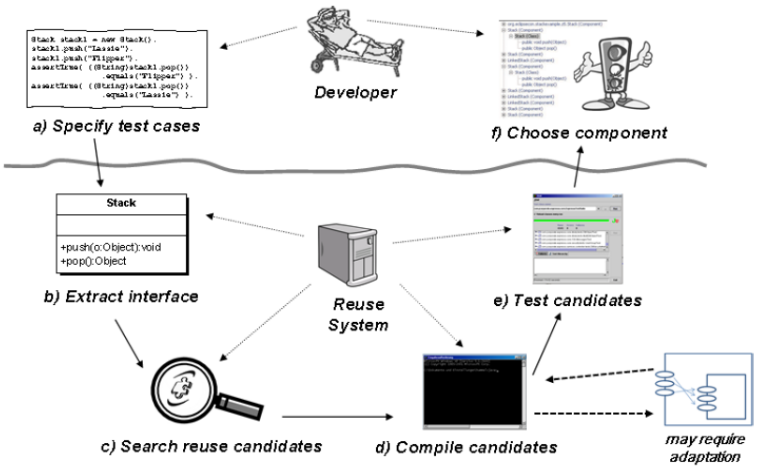


Abbildung 2.1: The testdriven reuse "cycle"[HJ13]

Hier spezifizieren die Entwickler*innen eine Menge von Testfällen (a) aus denen von eine Search Engine (in der Abbildung „Reuse System“ genannt) ein Interface extrahiert wird (b). Dieses Interface wird für die Suche nach Kandidaten, die zu diesem Interface passen, verwendet (c). Diese Kandidaten werden im Anschluss kompiliert, wobei mitunter eine Anpassung (Adaption) erfolgen muss, um das extrahierte Interface vollständig[1] zu erfüllen (d). Die letzte Aufgabe der Search Engine besteht dann im Test der kompilierten und mitunter adaptierten Kandidaten. Hierfür werden die von den Entwickler*innen in Schritt a spezifizierten Testfälle verwendet. Darauf aufbauend wird eine Liste von relevanten Kandidaten erarbeitet, aus der die Entwickler*innen eine Kandidaten zur weiteren Verwendung auswählen können (f).

Zu beachten ist, dass die Entwickler*innen bei diesem Ansatz die zu verwendende Komponente letztendlich selbst auswählen müssen. Der Ansatz eignet sich also nicht zum Einsatz während der Laufzeit des Systems. Weiterhin ist zu erwähnen, dass das Interface, welches für die Suche verwendet wird, aus den von den Entwickler*innen spezifizierten Testfällen extrahiert wird. Im Rahmen dieser Arbeit soll das Interface jedoch vorgegeben werden[2], da so der Explorationsalgorithmus und die beschriebenen Heuristiken besser nachvollzogen werden können.

[1] ein (angepasster) Kandidat erfüllt ein Interface vollständig, wenn er zu jeder Methode des Interfaces eine passende Methode anbietet.

[2] Die Vorgabe erfolgt durch die Entwickler*innen.

Der Ansatz zur *TDCS* wurde bereits in [BNL+06] von Bajaracharya et al. verfolgt. Diese Gruppe entwickelte eine Search Engine namens *Sourcerer*, welche die Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens *CodeGenie* entwickelt, welches Softwareentwickler*innen die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde erstmals der Begriff der *TDCS* etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummels [Hum08] ebenfalls eine Weiterentwicklung von *Sourcerer* veröffentlicht, welche unter dem Namen *Merobase* bekannt ist und ebenfalls das Konzept der *TDCS* verfolgt.

In [Hum08] wurden in Bezug auf die *TDCS* drei weitere Voraussetzungen identifiziert, die in dem oben beschriebenen Zyklus nicht eindeutig erwähnt wurden:

1. Ein Software-Repository, in dem die wiederverwendbaren Softwareteile enthalten sind.

2. Ein Format für die Repräsentation dieser Softwareteile.

3. Ein Mechanismus, welcher in der Lage ist, das Repository zu durchsuchen.

In früheren Arbeiten wurden im Internet bestehende Code-Repositiories als Software-Repository verwendet. Die Repräsentation konnte dabei je nach Repository unterschiedliche Formen haben. Damit ist die Darstellung gemeint, auf deren Basis die Kandidaten aus dem Repository ermittelt werden. Daher muss es möglich sein sowohl die Kandidaten als auch das Interface, welches von den Entwickler*innen spezifiziert oder aus den Testfällen extrahiert wurde, in dieser Form zu beschreiben. Die Mechanismen, die in bestehenden Arbeiten für die Suche verwendet wurden, sind ebenfalls vielfältig. Eine Auflistung der am häufigsten verwendeten Ansätze und eine kurze Erklärung ist in [HJ13] und [Hum08] zu finden.

In den derzeit jüngsten Arbeiten zu diesem Thema wurde versucht, den *TDCS*-Ansatz immer tiefer in den Entwicklungsprozess zu verankern (vgl. [KA18]). Darüber hinaus wurde versucht, die Komponenten, die den Entwickler*innen für die Wiederverwendung angeboten wurden, nach unterschiedlichen Kriterien zu sortieren. Dabei ist eine Vielzahl von Ranking Ansätzen entstanden, in denen die Abdeckung der funktionellen Anforderungen (bspw. in [SED16], [KA15]), oder die Abdeckung der nicht-funktionelle Anforderungen (bspw. in [KA16]) bestimmt und für die

Sortierung verwendet wird. Zu bemerken ist jedoch, dass das Ranking einen nachgelagerten Prozess darstellt[3]. Somit wird die Suche/Exploration der Search Engine durch ein solches Ranking nicht beschleunigt.

2.2 Testgetriebene Exploration von EJBs

Diese Arbeit legt den Fokus auf die Exploration von *Enterprise-Java-Beans* (*EJBs*). Hummel identifizierte *EJBs* bereits in [Hum08] als eine Client-Server-Architektur für Software-Systeme, welche die Kommunikation zwischen Komponenten (so genannten *Beans*), die auf physikalisch unterschiedlichen Maschinen laufen, koordinieren bzw. unterstützen können (vgl. auch [DeM05]). Dazu wird das jeweilige Software-System auf einem Applikationsserver deployed, der die EJB-Spezifikation [DeM05] erfüllt.

Bei einer *Bean* handelt es sich grundlegend um eine Java-Klasse, deren Struktur außerhalb dieser Klasse spezifiziert wurde. Seit der Version 3.0 der EJB-Spezifikation kann die Struktur durch ein Java-Interface spezifiziert werden. In früheren Versionen erfolgt dies in einer XML-Datei. [DeM05]

Die *Beans* können über einen *EJB-Container* abgerufen werden. Zu diesem Zweck publiziert der *EJB-Container* die Interfaces der deployten *Beans*, sodass diese auf den Clients über JNDI oder Dependency Injection zur Verfügung stehen [DeM05].

Bezogen auf die in [Hum08] beschriebenen Voraussetzungen für die *TDCS* wird der *EJB-Container* in dieser Arbeit als Software-Repository angesehen. Die einzelnen Softwareteile werden in Form von Java-Interfaces repräsentiert. Und der Mechanismus zum Durchsuchen des Repositories wird durch die Publikation der Java-Interfaces der *EJBs* durch den *EJB-Container* bereitgestellt.

Der Prozess für die Exploration von *EJBs* unterscheidet sich leicht von dem aus Abbildung 2.1. Während in der Beschreibung von Hummer das Interface aus den Testfällen extrahiert

[3] In Bezug auf Abbildung 2.1 würde das Ranking zwischen Schritt e und f eingeordnet werden.

wird, müssen die Entwickler*innen hier das Interface selbst entwerfen. Dies erfolgt in der Form eines Java-Interfaces. Die Tests werden als Java-Klassen deklariert, die über ihre Methoden eine Validierung der *EJBs* erlauben.

Darüber hinaus muss klargestellt werden, dass die Exploration der *EJBs* zur Laufzeit durchgeführt wird, da anderenfalls der *EJB-Container* nicht zur Verfügung steht. Somit muss die Exploration während der Laufzeit gestartet werden können. Zu diesem Zweck wird eine *Explorationskomponente* in das System integriert.

Der Prozess für den beschriebenen Ansatz kann somit in einen Implementierungsprozess und einen Explorationsprozess, welcher zur Laufzeit durchgeführt wird, eingeteilt werden.

In Abbildung 2.2 ist der Implementierungsprozess aufgezeigt:

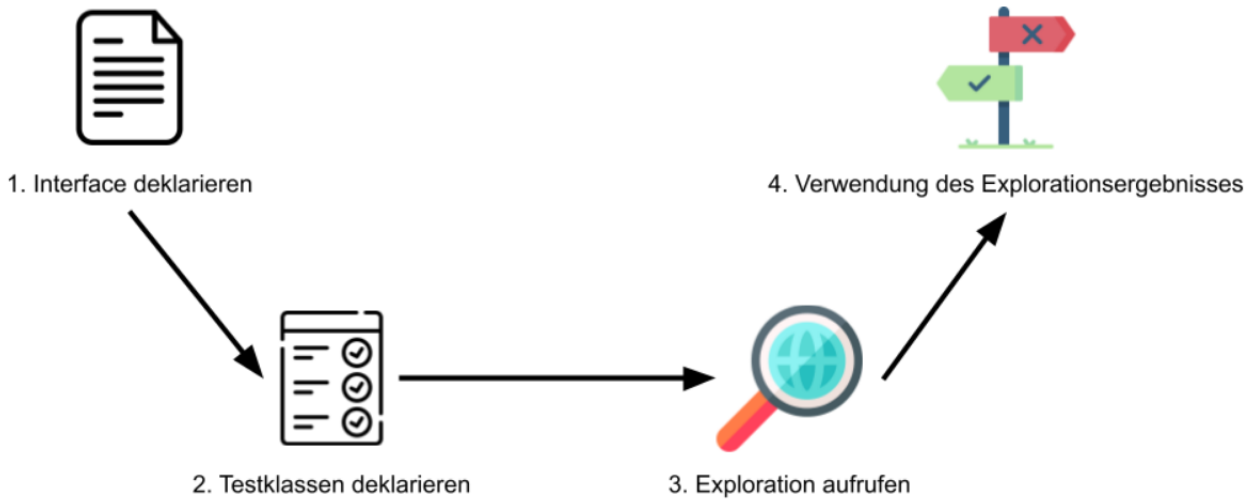


Abbildung 2.2: Implementierungsprozess

Wie bereits erwähnt, deklarieren die Entwickler*innen das Interface (1) und die Testklassen (2). Im dritten Schritt erfolgt der Aufruf der Explorationskomponente, wodurch zur Laufzeit der Explorationsprozess (siehe Abbildung 2.3) gestartet wird. Das Ergebnis des Explorations-

prozesses kann in Form des deklarierten Interfaces weiter verwendet werden. Allerdings muss der Entwickler auch davon ausgehen, dass durch die Explorationskomponente keine passende Komponente ermittelt werden konnte.

Die folgende Abbildung stellt den Explorationsprozess dar:

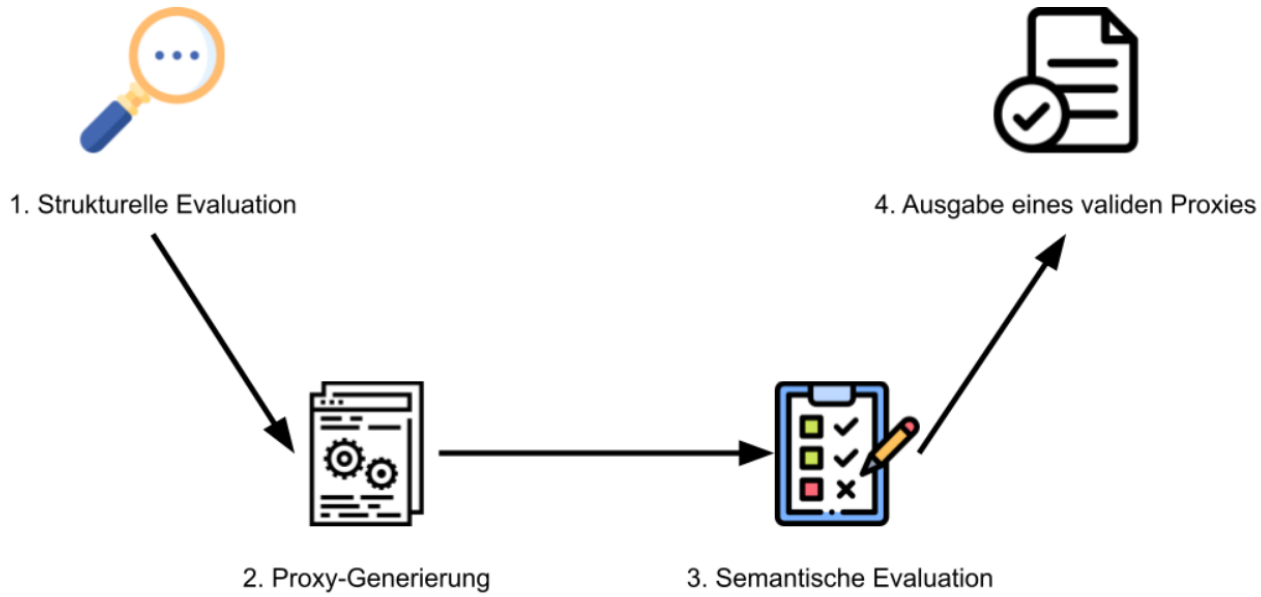


Abbildung 2.3: Explorationsprozess

Hier wird zuerst eine *strukturelle Evaluation* auf Basis des vorgegebenen Interfaces und der vom *EJB-Container* publizierten Interfaces durchgeführt. Dieser Schritt ist mit dem Suchen der Kanditaten aus Abbildung 2.1 vergleichbar. Auf Basis der Kandidaten, die bei der *sturktu-rellen Evaluation* ermittelt wurden, werden im zweiten Schritt *Proxies* generiert, durch die die Methodenaufrufe auf dem vorgegebenen Interface an die jeweiligen Kandidaten delegiert werden. Dies ist mit Schritt d aus Abbildung 2.1 zu vergleichen. Im nächsten Schritt (*semantische Evaluation*) werden - analog zu Schritt e aus Abbildung 2.1 - die vorgegebenen Testklassen verwendet, um eben jene *Proxies* zu validieren. Sofern ein valider *Proxy* gefunden wurden, wird dieser im 4. Schritt von der Explorationskomponente zurückgegeben.

Wie in Kapitel 3 noch beschrieben wird, besteht die Möglichkeit, dass das vorgegebene Interface erst durch eine Kombination einiger *EJBs* gänzlich erfüllt werden kann. Eine solche Kombination der *Beans* soll über ein *Proxy-Objekt* erreicht werden, welches bei der Exploration im Anschluss an die *strukturelle Evaluation* generiert wird. Das *Proxy-Objekt* muss dann zum einen in der Lage sein, die Methodenaufrufe wie in den Methoden-Signaturen den vorgegebenen Interfaces entgegenzunehmen und diese dann zum anderen an die entsprechende *Bean*, die eine dazu passende Methode bereitstellt, delegieren.

Da die Exploration wie oben beschrieben zur Laufzeit durchgeführt wird, sollte die Suche abgebrochen werden, sofern ein *Proxy* erfolgreich validiert wurde. Anderenfalls kann es bspw. zu unnötigen Timeouts laufender Transaktionen kommen. Um darüber hinaus ein schnelles Auffinden eines validen *Proxies* zu gewährleisten, können bei der *semantischen Evaluation* Heuristiken verwendet werden, welche die Generierung von *Proxies* und deren Validierung beschleunigen. Die vorliegende Arbeit dient hauptsächlich der Evaluation solcher Heuristiken.

Kapitel 3

Theoretische Grundlagen

In den folgenden Abschnitten wird der Explorationsprozess und dessen Grundlagen formal beschrieben sowie zum besseren Verständnis mit entsprechenden Beispielen untermalt. Die einzelnen Schritte des Prozesses finden sich damit in den Überschriften der Abschnitte wieder.

3.1 Strukturelle Evaluation

In Anlehnung an [Hum08] werden die *EJBs* auf der Basis des Signature-Matching Ansatzes ermittelt. Dieser Ansatz wurde ursprünglich von Zaremski und Wing [ZW95] beschrieben. Er basiert darauf, dass lediglich die Methoden-Signaturen der Typen (Klassen bzw. Interfaces) miteinander abgeglichen (gematcht) werden.

Zu diesem Zweck wird eine Struktur zur Deklaration von Typen in Abschnitt 3.1.1 vorgegeben, die eine abstrakte Darstellung von Klassen oder Interfaces, darstellen. Darüber hinaus werden in den genannten Abschnitt die Eigenschaften der Typen sowie Funktionen vorgestellt, die für den weiteren Verlauf der Arbeit von Belang sind.

Der Abgleich der Methoden-Signaturen dieser Typen erfolgt in Anlehnung an [ZW95] auf der Basis von Matchern, welche in Abschnitt 3.1.2 genauer beschrieben werden. Einige der dort beschriebenen Matcher basieren auf denen aus [ZW95]. Andere basieren auf Überlegungen aus [Hum08].

3.1.1 Struktur für die Definition von Typen

Die Typen werden in einer Bibliothek L in folgender Form deklariert:

Regel	Erläuterung
$L ::= TD*$	Eine Bibliothek L besteht aus einer Menge von Typdefinitionen.
$TD ::= PD \mid RD$	Eine Typdefinition kann entweder die Definition eines *provided Typen* (PD) oder eines *required Typen* (RD) sein.
$PD ::=$ `provided` T `extends` T' $\{FD*MD*\}$	Die Definition eines *provided Typen* besteht aus dem Namen des Typen T, dem Namen des Super-Typs T' von T sowie mehreren Feld- und Methodendeklarationen.
$RD ::=$ `required` T $\{MD*\}$	Die Definition eines *required Typen* besteht aus dem Namen des Typen T sowie mehreren Methodendeklarationen.
$FD ::= T\ f$	Eine Felddeklaration besteht aus dem Namen des Feldes f und dem Namen seines Typs T.
$MD ::= T'\ m(T_1, ..., T_n)$	Eine Methodendeklaration besteht aus dem Namen der Methode m, den insgesamt n Namen der Parameter-Typen T_1 bis T_n und dem Namen des Rückgabe-Typs T'.

Tabelle 3.1: Struktur für die Definition einer Bibliothek von Typen

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

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

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

Darüber hinaus seien folgende Funktionen definiert:

$$members(T) := \left\{ \; T \;\; f \;\middle|\; T \;\; f \text{ ist Felddeklaration von } T \; \right\}$$

$$memType(f, T) := \;\; T' \;\middle|\; T' \;\; f \in members(T)$$

$$ret(T' \;\; m(T''_1, ... T''_n)) := T'$$

$$params(T'' \;\; m(T'_1, ... T'_n)) := \{T'_1, ..., T'_n\}$$

$$methods(T) := \left\{ \; T'' \; m(T'_1, ..., T'_n) \;\middle|\; T'' \; m(T'_1, ..., T'_n) \text{ ist Methodendeklaration von } T \; \right\}$$

Listing 3.1 zeigt die Deklaration der Bibliothek *ExampLe* als Beispiel für eine Bibilothek mit *required* und *provided Typen*[1].

```
provided Fire extends Object{}

provided FireState extends Object{
 boolean isActive
}

provided Injured extends Object{
 void heal(Medicine med)
}

provided FireFighter extends Object{
 FireState extinguishFire(Fire fire)
}

provided InverseDoctor extends Object{
 void heal( Medicine med, Patient pat )
}

required PatientMedicalFireFighter {
 void heal( Patient patient,
            MedCabinet med )
 boolean extinguishFire( ExtFire fire )
}
```

```
provided ExtFire extends Fire{}

provided Medicine extends Object{
 String getDescription()
}

provided Patient extends Injured{
 String getName()
}

provided Doctor extends Object{
 void heal( Patient pat, Medicine med )
}

provided MedCabinet extends Object{
 Medicine med
}

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

Listing 3.1: Bibliothek *ExampLe* von Typen

[1]Zu beachten ist, dass die Bibliothek auf die im JDK enthaltenen Typen aufbaut. Daher ist davon auszugehen, dass Typen wie `Object` oder `boolean` bereits als *provided Typen* definiert sind.

3.1.2 Definition der Matcher

Ein Matcher definiert das Matching eines Typs T zu einem Typ T' oder einer Methode m zu einer Methode m' durch die asymmetrische Relation $\Rightarrow$ (auch Matchingrealtion genannt)[2]. Im Folgenden werden die Matchingrelationen der spezifischen Matcher über ein Subskript differenziert.

ExactTypeMatcher

Der *ExactTypeMatcher* definiert das Matching von einem Typ T zu sich selbst her (vgl. [ZW95]). Die dazugehörige Matchingrelation $\Rightarrow_{exact}$ wird durch folgende Regel beschrieben:

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

GenTypeMatcher

Der *GenTypeMatcher* definiert das Matching von einem Typ T zu einem Typ T' mit $T > T'$ (vgl. [ZW95]). Die dazugehörige Matchingrelation $\Rightarrow_{gen}$ wird durch folgende Regel beschrieben:

$$\frac{T > T'}{T \Rightarrow_{gen} T'}$$

SpecTypeMatcher

Der *SpecTypeMatcher* definiert im Verhältnis zum *GenTypeMatcher* das Matching in die entgegengesetzte Richtung (vgl. [ZW95]). Die dazugehörige Matchingrelation $\Rightarrow_{spec}$ wird durch folgende Regel beschrieben:

$$\frac{T < T'}{T \Rightarrow_{spec} T'}$$

Die oben genannten Matchingrelationen werden für die Definition weiterer Matcher zusam-

[2] $T \Rightarrow T'$
Gesprochen: T matcht T'

mengefasst, wodurch sich die Matchingrelation $\Rightarrow_{internCont}$ ergibt:

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

Die folgenden Matcher definieren das Matching für so genannte Wrapper-Typen zu den Typen der in ihnen enthaltenen Attribute. Die Idee für solche Matcher fand in [Hum08] zwar Erwähnung, jedoch erfolgte dort keine formale Beschreibung. Das Ziel dieser Matcher ist es bspw. die Typen `boolean` und `FireState` aus der in Listing 3.1 deklarierten Bibliothek *ExampLe* zu matchen.

ContentTypeMatcher

Der *ContentTypeMatcher* definiert das Matching von einem Typ T zu einem Typ T', wobei T' ein Feld enthält, auf dessen Typ T'' der Typ T über die Matchingrelation $\Rightarrow_{internCont}$ gematcht werden kann.

Die dazugehörige Matchingrelation $\Rightarrow_{content}$ wird durch folgende Regel beschrieben:

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

So würde für die Typen `boolean` und `FireState` aus der Bibliothek *ExampLe* (siehe 3.1) gelten:

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

ContainerTypeMatcher

Der *ContainerTypeMatcher* definiert im Verhältnis zum *ContentTypeMatcher* das Matching für die entgegengesetzte Richtung.

Die dazugehörige Matchingrelation $\Rightarrow_{container}$ wird durch folgende Regel beschrieben:

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

So gilt für die Typen `FireState` und `boolean` aus der Bibliothek *ExampLe* (siehe 3.1):

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

Zur Definition des letzten Matchers werden die Matchingrelationen der oben genannten Matcher wiederum zusammengefasst. Dabei entsteht die Matchingrelation $\Rightarrow_{internStruct}$, welche durch folgende Regel beschrieben wird:

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

StructuralTypeMatcher

Der *StructuralTypeMatcher* definiert das Matching von einem *required Typ* R zu einem *provided Typ* T auf der Basis der Methoden-Signaturen der beiden Typen.

Somit soll bspw. ein Matching zwischen dem Typ `MedicalFireFighter` und dem den `FireFighter` aus der Bibliothek *ExampLe* (siehe Listing 3.1) gematcht werden. Als ein weiteres Beispiel, bezogen auf die Typen aus der Bibliothek *ExampLe*, kann das Matching zwischen dem Typ `MedicalFireFighter` und dem Typ `Doctor` angebracht werden.

Damit ein *required Typ* R auf einen *provided Typ* T über den *StrukturalTypeMatcher* gematcht werden kann, muss mindestens eine Methode aus R zu einer Methode aus T gematcht werden (Signature-Matching). Ein Matching der Methoden liegt dann vor, wenn sowohl die Rückgabe- als auch die Parameter-Typen dieser beiden Methoden miteinander gematcht werden können (vgl. [ZW95]).

Wie in [ZW95] soll die Reihenfolge, in der die Parameter in der jeweiligen Methode dekla-

riert wurden, keine Rolle spielen. Ausgehend von den Parameter-Typen der beiden Methoden als Mengen, muss eine der Mengen also so umsortiert werden, dass die Parameter-Typen aus beiden Mengen an der jeweils gleichen Position miteinander gematcht werden können. Die möglichen umsortierte Mengen von Parameter-Typen einer Methode m auf die dies in Bezug auf die Menge der Parameter-Typen einer Methode m' zutrifft, werden über die Funktion *matchingParams* beschrieben:

$$matchingParams(m, m') := \left\{ \{mP_1, ..., mP_n\} \;\middle|\; \begin{array}{l} \{P_1, ..., P_n\} = params(m) \wedge \\ \forall i \in \{1, ..., n\} : mP_i \in params(m') \wedge \\ mP_i \Rightarrow_{internStruct} P_i \end{array} \right\}$$

Das Matching zweier Methoden m und m' wird durch die Relation $\Rightarrow_{method}$ über folgende Regel beschrieben:

$$\frac{ret(m) \Rightarrow_{internStruct} ret(m') \wedge matchingParams(m, m') \neq \emptyset}{m \Rightarrow_{method} m'}$$

Die Menge der gematchten Methoden aus R in T wird darauf aufbauend durch folgende Funktion beschrieben:

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

Die Matchingrelation für den *StructuralTypeMatcher* wird durch folgende Regel beschrieben:

$$\frac{structM_{source}(R, T) \neq \emptyset}{R \Rightarrow_{struct} T}$$

3.1.3 Ergebnis der strukturellen Evaluation

Die Exploration wird für einen *required Typ* durchgeführt. Bei der *strukturellen Evaluation* sollen Mengen von *provided Typen* ermittelt werden, deren Methoden in Kombination zu jeder Methode des *required Typ* ein Matching aufweisen. Die Mengen von *provided Typen* innerhalb einer Bibliothek L für die dies in Bezug auf ein *required Typ* R zutrifft, wird über die Funktion *cover* beschrieben.

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

Die *provided Typen* innerhalb dieser Mengen werden im nächsten Schritt des Explorationsprozesses als *Target-Typen* bezeichnet und als Basis für die Generierung der Proxies für den *required Typ* R verwendet.

Beispiel 1 Sei folgende Bibliothek L gegeben.

```
provided Come extends Object{
        String hello()
        String goodMorning()
}

provided Leave extends Object{
        String bye()
}

required Greeting{
        String hello()
        String bye()
}
```

Über die Funktion *cover* werden folgende *Target-Typen* für die Generierung von Proxies für den *required Typ* `Greeting` ermittelt.

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

3.2 Generierung der Proxies auf Basis von Matchern

Ein *Proxy* ist ein Objekt, das stellvertretend für ein anderes Objekt verwendet wird und den Zugang - in diesem Fall den Methodenaufruf - auf dieses Objekt kontrolliert (vgl. [ES13]). Dadurch ist es dem *Proxy* möglich, die Methodenaufrufe an andere Objekte zu delegieren. Diese Eigenschaft wird sich in dieser Arbeit zunutze gemacht, um einen Aufruf einer Methode, die in

einem Typ (*required* oder *provided Typ*) deklariert wurde, an eine Methode zu delegieren, die in einem anderen *provided Typ* deklariert wurde.

Dabei wird zwischen einem *Source-Typen* und einem oder mehrerer *Target-Typen* unterschieden. Als *Source-Typ* wird immer der Typ bezeichnet, für den der *Proxy* generiert und stellvertretend eingesetzt wird. Bei den *Target-Typen* handelt es sich um die *provided Typen* an deren Methoden die Methodenaufrufe delegiert werden.

Zur Beschreibung der Generierung von *Proxies* wird im Folgenden zuerst vorgegeben, wie sich ein *Proxy* deklarieren lässt. Darauf aufbauend werden die Generatoren, die in Abhängigkeit des Matchings zwischen *Source-* und *Target-Typen* Anwendung finden, beschrieben.

3.2.1 Struktur für die Definition von Proxies

Die Grammatikregeln für die Deklaration eines *Proxies* sind Tabelle 3.2 zu entnehmen.

Regel	Erläuterung
$PROXY$ ::= `proxy for` T `with [`$P_1, ..., P_n$`]` $\{MDEL_1, ..., MDEL_k\}$	Ein *Proxy* wird für ein Typ T als *Source-Typ* mit einer Menge von *provided Typen* $P = \{P_1, ..., P_n\}$ als *Target-Typen*, einer Menge von Methoden-Delegationen erzeugt.
$MDEL$ ::= $CALLM \rightarrow DELM$	Eine *Methodendelegation* besteht aus einer *aufgerufenen Methode* und aus einem *Delegationsziel*.
$CALLM$ ::= $REF.m(CP_1, ..., CP_n) : CR$	Eine *aufgerufene Methode* besteht aus dem Namen der Methode m, dem Rückgabetyp CR und einer Menge von Parametertypen $\{CP_1, ..., CP_n\}$.
$DELM$ ::= $REF.n(DP_1, ..., DP_n) : DR$	Die erste Variante eines *Delegationsziels* besteht aus dem Namen der *Delegationsmethode* n, dem Rückgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `posModi(`$I_1, ..., I_n$`)` $REF.n(DP_1, ..., DP_n) : DR$	Die zweite Variante eines *Delegationsziels* besteht aus einer Menge von Indizies $\{I_1, ..., I_n\}$, einer *Referenz*, dem Namen der *Delegationsmethode* n, dem Rückgabetyp DR und einer Menge von Parametertypen $\{DP_1, ..., DP_n\}$.
$DELM$ ::= `err`	Die dritte Variante eines *Delegationsziels* enthält keine weiteren Bestandteile. Das Terminal `err` weist darauf hin, dass die Delegation innerhalb des Proxies nicht möglich ist und zu einem Fehler führt.
$REF ::= P_i$	Die erste Variante einer *Referenz* besteht aus einem Typ P_i .
$REF ::= P_i.f$	Die zweite Variante einer *Referenz* besteht aus einem Typ P_i und einem Feldnamen f.

Tabelle 3.2: Grammatikregeln mit Erläuterungen für die Deklaration eines Proxies

Es handelt sich dabei um Produktionsregeln einer Attributgrammatik. Die dazugehörigen Attribute sind der Tabelle 3.3 zu entnehmen. Dazu sei zusätzlich festgelegt, dass die Notation $NT.*$ in der Spalte *Attribute* eine Key-Value-Liste aller Attribute des Nonterminals NT beschreibt, wobei der Attributname als Key und dessen Wert als Value innerhalb der Liste verwendet wird. Weiterhin sei ein Attribut, dass in der Spalte *Attribute* zu einem Nonterminal nicht aufgeführt ist, 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` $= [MDEL_1.*, ..., MDEL_k.*]$ $MDEL_1$`.call.field` $= ... =$ $MDEL_k$`.call.field` $MDEL_1$`.del.field` $= ... =$ $MDEL_k$`.del.field`
$MDEL$ ::= $CALLM \rightarrow DELM$	`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` $= memType(f, P)$

Tabelle 3.3: Grammatikregeln mit Attributen für die Deklaration eines Proxies

3.2.2 Delegation von Methoden im Proxy

Ein *Proxy* bietet alle Methoden des *Source-Typen* an. Einige dieser Methoden werden an eine Methode delegiert, die von einem *Target-Typ* des *Proxies* angeboten wird. Eine solche Delegation wird durch eine *Methoden-Delegation* (siehe Tabelle 3.3 Nontermial *MDEL*) definiert.

Beispiel So beschreibt die folgende *Methoden-Delegation*, dass die Methode `extinguishFire`, die vom *Source-Typ* `Patient` - und damit auch vom *Proxy* - angeboten wird, an die Methoden `heal`, die der *Target-Typ* `Injured` anbietet, delegiert wird.

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

Listing 3.2: Einfache Methoden-Delegation

Die Delegation einer *aufgerufenen Methode* an ein *Delegationsziel*, erfolgt in drei Schritten.

1. Parameterübergabe
   Dabei werden die Parameter, mit denen die vom *Proxy angebotene Methode* aufgerufen wird, an die *Delegationsmethode* des *Delegationsziels* übergeben. Dabei sind zwei Dinge zu beachten. Zum einen müssen die Typen der übergebenen Parameter zu den Typen der von der *Delegationsmethode* erwarteten Parameter passen. Zum anderen muss die Reihenfolge, in der die Parameter übergeben wurden, an die erwartete Reihenfolge der *Delegationsmethode* angepasst werden. (siehe auch Funktion *matchingParams* aus Abschnitt 3.1.2)

2. Ausführung
   Dieser Schritt meint die Durchführung der *Delegationsmethode* mit den übergeben Parametern aus Schritt 1. Dies schließt auch die Ermittlung des korrekten Rückgabewertes der *Delegationsmethode* ein.

3. Übergabe des Rückgabewertes
   Ähnlich wie bei der Parameterübergabe, muss auch der Rückgabewert, der bei der Ausführung in Schritt 2 ermittelt wurde, an die *aufgerufene 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 3.1 dargestellt werden. Die Übergabe der Parameter- und Rückgabewerte wird durch die gestrichelten Pfeile symbolisiert.

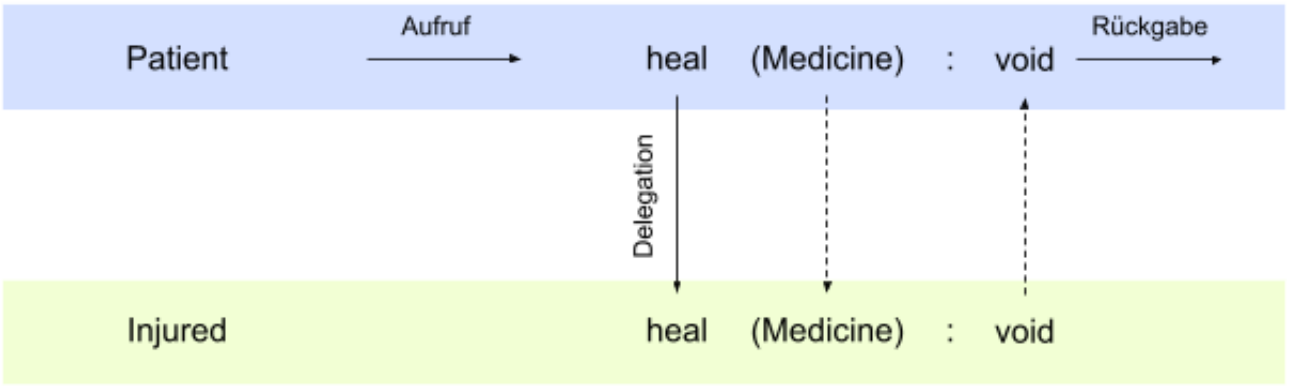


Abbildung 3.1: Delegation der Methode `heal`

In diesem Beispiel sind sowohl die Parameter- als auch die Rückgabe-Typen der *aufgerufenen Methode* und der *Delegationsmethode* identisch. Weiterhin spielt die Reihenfolge der Parameter in diesem Beispiel keine Rolle, da es nur einen Parameter gibt. Daher stellt die Übergabe der Parameter- und Rückgabewerte kein Problem dar.

Folgendes Beispiel soll zeigen, wie mit unterschiedlichen Reihenfolgen bzgl. der Parameter bei einer *Methoden-Delegation* umzugehen ist.

Beispiel Die *Methoden-Delegation* aus Listing 3.2.2 ist ein Beispiel für einen solchen Fall. Hier wird die *aufgerufene Methode* `heal` mit den Parametern `Patient` und `MedCabinet` aus dem Typ `PatientMedicalFireFighter` an die gleichnamige Methode aus dem Typ `InverseDoctor` delegiert. Die *Delegationsmethode* verwendet zwar identische Parameter-Typen, aber die Reihenfolge, in der die Parameter übergeben werden, ist unterschiedlich.

```
PatientMedicalFireFighter.heal(Patient, MedCabinet):void → posModi(1,0)
    InverseDoctor.heal(MedCabinet,Patient):void
```

Listing 3.3: Methoden-Delegation mit Parametern in unterschiedlicher Reihenfolge

Um die Reihenfolge der Parameter aus dem ursprünglichen Aufruf zu variieren, wird das Schlüsselwort `posModi` verwendet. Dort werden eine Reihe von Indizes angegeben. Die Anzahl der angegebenen Indizes muss mit der Anzahl der Parameter übereinstimmen. Ein Index beschreibt die Position des in der *aufgerufenen Methode* angegebenen Parameter. Weiterhin spielt die Reihenfolge der Indizes eine wichtige Rolle. Diese ist mit der Reihenfolge der Parameter der *Delegationsmethoden* gleichzusetzen.

So wird in dem o.g. Beispiel der erste Parameter der *aufgerufenen Methoden* (Index = 0) der *Delegationsmethode* als zweiter Parameter übergeben. Dementsprechend wird der zweite Parameter der *aufgerufenen Methode* (Index = 1) der *Delegationsmethode* als erster Parameter übergeben (siehe Abbildung 3.2).

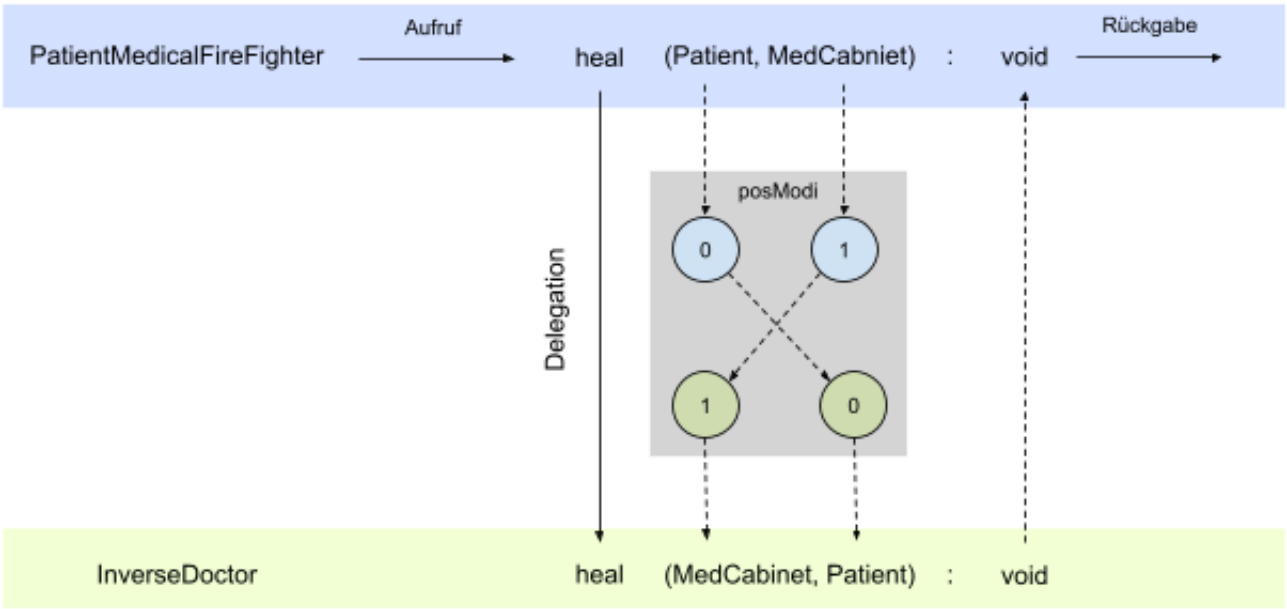


Abbildung 3.2: Delegation der Methode `heal` mit Parametern in unterschiedlicher Reihenfolge

Dass identische Typen keine Probleme bei der Übergabe zwischen *aufgerufener Methode* und *Delegationsmethode* darstellen, wurde in den oben genannten Beispielen gezeigt. Darüber hinaus können Typen aber auch dann ohne Probleme übergeben werden, wenn sie sich aufgrund des Substitutionsprinzips austauschen lassen. Daher kann ein Typ T anstelle eines Typs T'

verwendet werden, sofern $T \leq T'$ gilt.

Folgendes Beispiel soll zeigen, wie mit übergebenen Typen umzugehen ist, die nicht ohne Probleme übergeben werden können.

Beispiel In folgendem Listing ist eine *Methoden-Delegation* aufgerührt, bei der sowohl die Parameter- als auch die Rückgabe-Typen der *aufgerufenen Methode* und der *Delegationsmethode* nicht auf Basis des Substitutionsprinzips übergeben werden können.

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

Listing 3.4: Methoden-Delegation mit Typkonvertierung

In einem solchen Fall müssen die Parameter-Typen der *aufgerufenen Methoden* in die Parameter-Typen der *Delegationsmethode* konvertiert werden. Analog dazu muss der Rückgabetyp der *Delegationsmethode* in den Rückgabetyp der *aufgerufenen Methoden* konvertiert werden. Die Konvertierung wird dabei über die Generierung eines *Proxies* erzielt.

Angenommen, eine Funktion *proxies*(S, T) beschreibt eine Menge von *Proxies*, mit S als *Source-Typ* und T als Menge der *Target-Typen*, dann müssten bezogen auf die *Methoden-Delegation* aus Listing 3.4 für die Parameter einer der *Proxies* aus der Menge *proxies*(`Fire`, {`ExtFire`}) an die *Delegationsmethode* übergeben werden. Nach der Ausführung der *Delegationsmethode* müsste aus dem Rückgabewert ein *Proxy* aus der Menge *proxies*(`boolean`, {`FireState`}) erzeugt werden und an die *aufgerufene Methode* als Rückgabewert übergeben werden. Der Sachverhalt wird in Abbildung 3.3 schematisch dargestellt. Die grauen Kästen symbolisieren die Generierung eines Proxies.

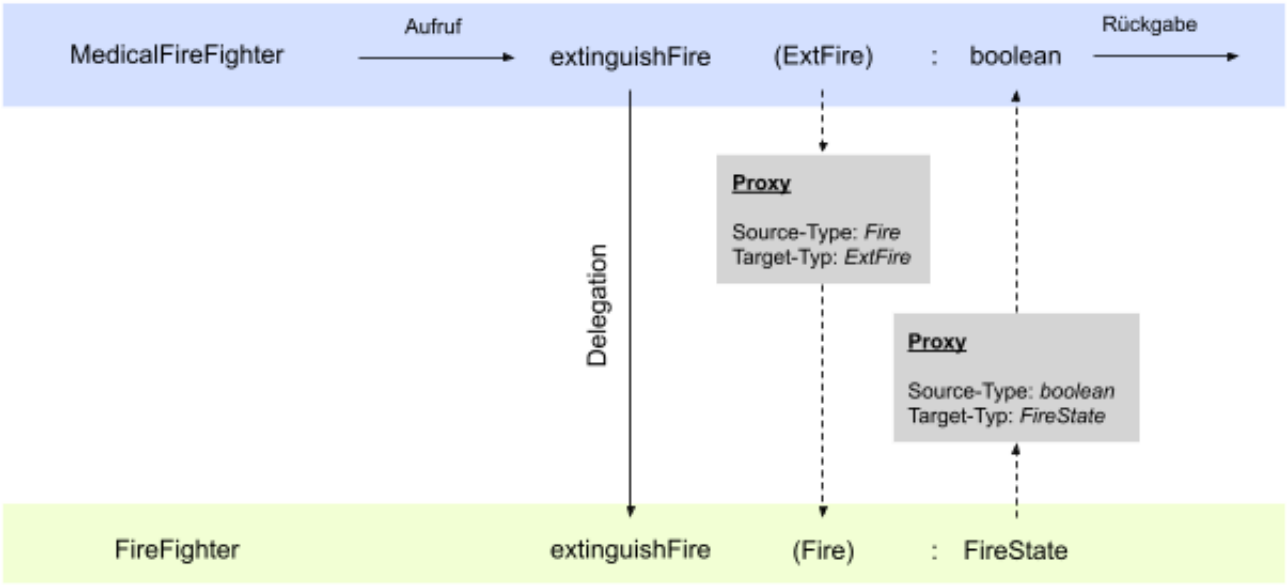


Abbildung 3.3: Delegation der Methode `extinguishFire` mit Typkonvertierungen

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

3.2.3 Generierung von Proxies

Wie im Abschnitt 3.2.1 bereits angedeutet, soll die Menge der *Proxies* für einen *Source-Typ* S und einer Menge von *Target-Typen* TM über die Funktion $proxies(S, TM)$ beschrieben werden.

In Abhängigkeit von dem Matching zwischen dem *Source-Typ* und den *Target-Typen* werden unterschiedliche Arten von *Proxies* generiert. Für die unterschiedlichen *Proxy*-Arten gibt es ebenfalls Funktionen, die eine Menge von *Proxies* zu einem *Source-Typen* S und einer Menge von *Target-Typen* TM beschreiben.

In den folgenden Abschnitten werden diese Funktionen für die einzelnen Proxy-Arten beschrieben. Dabei ist davon auszugehen, dass die *Proxies* eine allgemeine Struktur haben, die in Abschnitt 3.2.1 aufgeführt ist. Um die Regeln für die Generierung der *Proxies* zu beschreiben, soll davon ausgegangen werden, dass jedes Listen-Attribut ($NT.*$) aus Tabelle 3.3 ein Attribut `len` enthält in dem die Anzahl der Elemente abgelegt ist, die sich in dieser Liste befinden.

Sub-Proxy

Die Voraussetzung für die Erzeugung eines *Sub-Proxies* vom Typ S (*Source-Typ*) aus einem *Target-Typ* T ist $S \Rightarrow_{spec} T$. Damit ist der *SpecTypeMatcher* der *Basis-Matcher* für den *Sub-Proxy*.

Beispiel Als Beispiel soll der Typ `Patient` als *Source-Typ* und der Typ `Injured` als *Target-Typ* verwendet werden. Da `Patient` $\Rightarrow_{spec}$ `Injured` gilt, kann ein *Sub-Proxy* für diese Konstellation erzeugt werden. Der resultierende *Sub-Proxy* ist in Listing 3.5 aufgeführt. Der abstrakte Syntaxbaum (AST) mit den dazugehörigen Attributen ist Abbildung 3.4 zu entnehmen. [3]

```
proxy for Patient with [Injured]{
        Patient.heal(Medicine):void → Injured.heal(Medicine):void
        Patient.getName():String → err
}
```

Listing 3.5: Sub-Proxy für Patient

Ein *Proxy* bietet alle Methoden an, die auch von dessen *Source-Typ* angeboten werden. Sofern für eine angebotene Methode keine *Methodendelegationen* innerhalb des *Proxies* existiert, wird diese Methode so ausgeführt, wie sie innerhalb des *Source-Typen* implementiert wurde. Anderenfalls beschreiben die *Methodendelegationen* innerhalb eines *Proxies*, was beim Aufruf der entsprechenden Methode 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 passende *Delegationsmethode* zur Verfügung steht.

Weiterhin beschreibt der *Sub-Proxy* aus dem Beispiel auch, dass der Aufruf der Methode `getName` zu einem Fehlschlag führt. Dies ist auf eine Problematik zurückzuführen, die auch bei einem Downcast auftritt. Hierbei soll ein Objekt eines Super-Typs anstelle eines Objektes eines Sub-Typs verwendet werden. Das Objekt des Sub-Typs bietet jedoch mitunter Methoden an, die im Super-Typ nicht deklariert wurden. Folglich können diese Methoden nicht ausgeführt werden, was jedoch häufig erst zur Laufzeit auffällt.

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

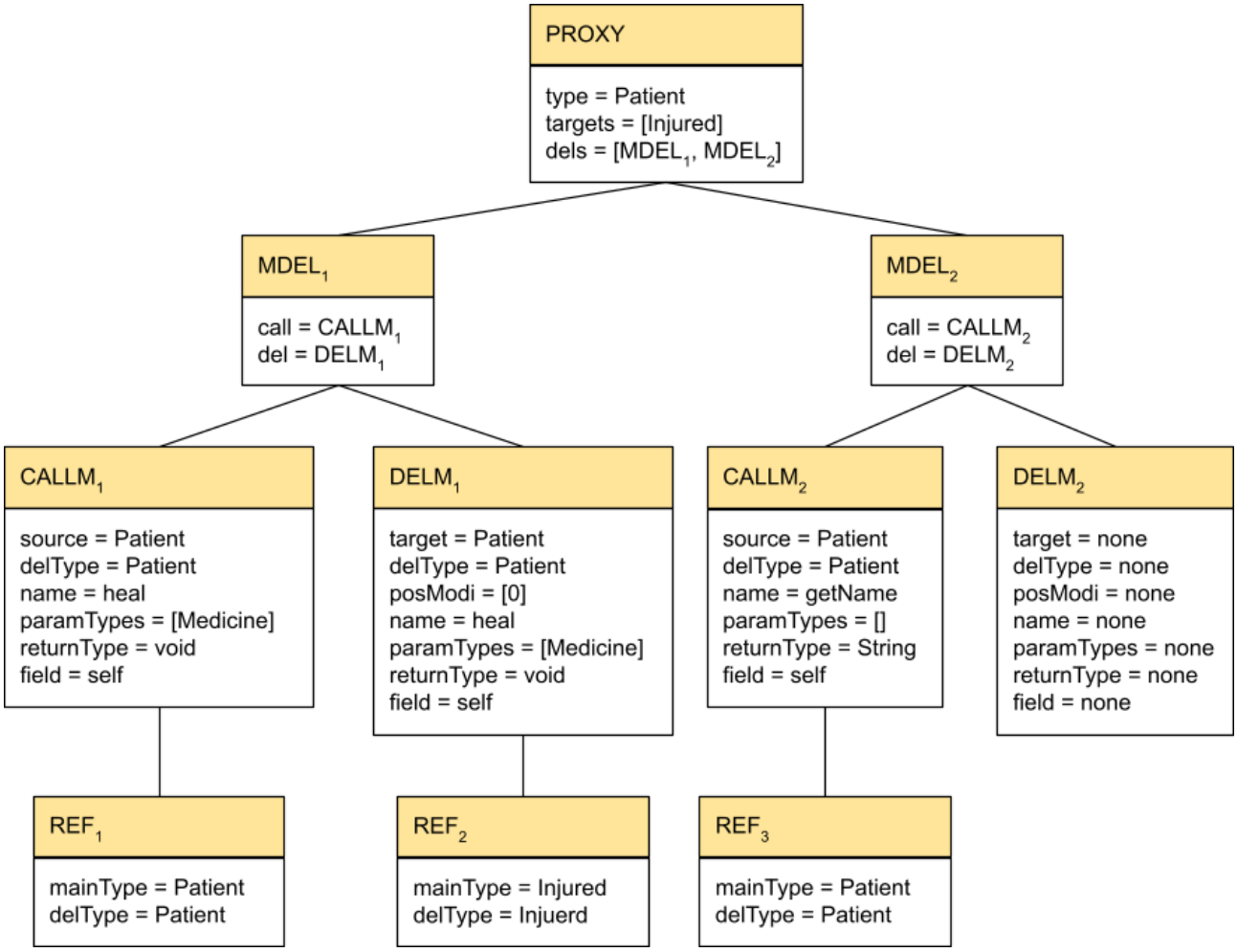


Abbildung 3.4: AST für das Beispiel zum Sub-Proxy

Formalisierung Formal wird ein *Sub-Proxy* für einen *Source-Typ* S auf der Basis von einer Menge von *Target-Typen* TM durch die Regeln beschrieben, die im Folgenden vorgestellt werden.

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

$$\frac{|P.targets| = 1 \wedge \forall T \in TM : T \in P.targets}{targets_{single}(P, TM)}$$

Darüber hinaus kann für jeden *Proxy* P (egal welcher Art) festgehalten werden, dass dessen Attribut `type` immer mit einem bestimmten Typen S übereinstimmt. Damit wird ausgesagt, dass S der *Source-Typ* des *Proxies* P ist.

$$\frac{P.type = S}{proxy(P, S)}$$

Die Unterschiede der einzelnen Proxy-Arten lassen sich in den *Methoden-Delegationen* finden. Eine *Methoden-Delegationen* besteht aus einer linken Seite - die *aufgerufene Methode* - und eines rechten Seite - die *Delegationsmethode*. Bevor die Beziehungen zwischen diesen beiden Seiten beschrieben werden, werden zuerst die separaten Eigenschaften der beiden Seiten beschrieben.

So muss die *aufgerufene Methode* immer im Typ aus dem Attribut `call.delType` deklariert sein.

$$\frac{\exists T'\ m(T) \in methods(MD.call.delType) : MD.call.name = m}{callDecl(MD)}$$

Bezüglich des Feldes `delType` der *Delegationsmethode* gilt ähnliches.

$$\frac{\exists T'\ m(T) \in methods(MD.del.delType) : MD.del.name = m}{delDecl(MD)}$$

Darüber hinaus müssen die Attribute `source` und `delType` der *aufgerufenen Methode* einer *Methoden-Delegation MD* mit dem *Source-Typ* des *Proxies* belegt sein. Dazu müssen die beiden folgenden Regeln gelten.

$$\frac{MD.call.source = MD.call.delType}{callDelegationType_{simple}(MD)}$$

$$\frac{MD.call.source = P.type}{sourceType(MD, P)}$$

Damit ist auch automatisch gewährleistet, dass das Attribut `field` im Attribut `call` der *Methoden-Delegation* mit dem Wert `self` belegt ist (vgl. Tabelle 3.3).

Ähnliches gilt für die Attribute `field` und `mainType` im Attribut `del` der *Methoden-Delegation MD*. Hierbei muss der Wert des Attributs `mainType` jedoch mit dem *Target-Typ* des *Proxies* übereinstimmen.

$$\frac{MD.del.target \in P.targets}{targetType(MD, P)}$$

$$\frac{MD.del.delType = MD.del.target}{delDelegationType_{sub}(MD)}$$

Damit ist wiederum automatisch gewährleistet, dass das Attribut `field` im Attribut `del` der *Methoden-Delegation* mit dem Wert `self` belegt ist (vgl. Tabelle 3.3).

Die Regeln für die linke Seite einer *Methoden-Delegation MD* innerhalb eines *Proxies P* können damit in folgender Regel zusammengefasst werden:

$$\frac{callDecl(MD) \wedge callDelegationType_{simple}(MD, P) \wedge sourceType(MD, P)}{call_{simple}(MD, P)}$$

Dementsprechend dazu können auch die Regeln für die rechte Seite einer *Methoden-Delegation MD* innerhalb eines *Proxies P* zusammengefasst werden:

$$\frac{delDecl(MD) \wedge targetType_{sub}(MD, P) \wedge delDelegationType_{sub}(MD)}{del_{simple}(MD, P)}$$

Die oben genannten Regeln beschreiben die notwendigen Bedingungen der beiden Seiten einer *Methoden-Delegation* innerhalb eines *Sub-Proxies*. Die Bedingungen, die für eine gesamte *Methoden-Delegation MD* eines *Sub-Proxies P* gilt, werden durch die folgenden beiden Regeln beschrieben.

$$\frac{MD.call.name = MD.del.name}{methodMatch_{simple}(MD)}$$

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{simple}}{delegation_{sub}(MD, P)}$$

Zu beachten ist jedoch, dass ein *Sub-Proxy* P auch fehlschlagende *Methoden-Delegationen* MD enthalten kann. Somit gilt ebenfalls:

$$\frac{MD.del.name = \texttt{none}}{methodErr(MD)}$$

$$\frac{call_{simple}(MD, P) \wedge methodErr(MD)}{delegation_{sub}(MD, P)}$$

Die Regel für alle *Methoden-Delegationen* eines *Sub-Proxies* P lässt sich aufbauend auf den oben genannten Regeln, wie folgt darstellen.

$$\frac{\forall MD \in P.dels : delegation_{sub}(MD, P)}{delegations_{sub}(P)}$$

Die Menge der *Sub-Proxies*, die mit dem *Source-Typ* S und der Menge von *Target-Typen* TM erzeugt werden, wird darauf aufbauend durch die folgende Funktion beschrieben.

$$proxies_{sub}(S, TM) := \left\{ P \;\middle|\; \begin{array}{l} proxy(P, S) \wedge \\ targets_{single}(P, TM) \wedge \\ delegations_{sub}(P) \end{array} \right\}$$

Content-Proxy

Die Voraussetzung für die Erzeugung eines *Content-Proxies* vom Typ S aus einem *Target-Typ* T ist $S \Rightarrow_{content} T$. Damit ist der *ContentTypeMatcher* der Basis-Matcher für den *Content-Proxy*.

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

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

```
}
```

Listing 3.6: Content-Proxy für Medicine

Durch die Methoden-Delegation dieses *Content-Proxies* wird die Methode `getDescription` an das Feld `med` des *Target-Typen* `MedCabniet` delegiert.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.5 zu entnehmen. [4]

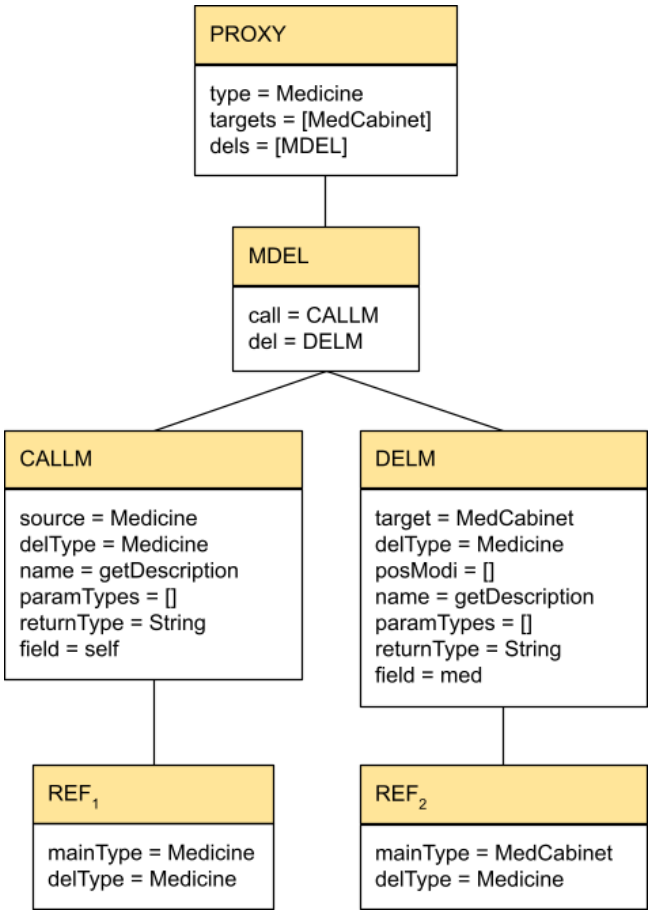


Abbildung 3.5: AST für das Beispiel zum Content-Proxy

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

[4] 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 MD* muss zwar ebenfalls das Attribut `target` in den *Target-Typen* des *Proxies P* enthalten sein ($targetType(MD, P)$), allerdings muss im *Content-Proxy* darüber hinaus jenes Attribut `target` ein Matching zum Typen im Attribut `delType` aufweisen.

$$\frac{P.type \Rightarrow_{internCont} MD.del.delType}{delDelegationType_{content}(MD, P)}$$

Folglich werden auch die Eigenschaften einer *Delegationsmethode* in einer *Methoden-Delegation MD* im *Content-Proxy* durch eine andere Regel beschrieben als beim *Sub-Proxy*:

$$\frac{delDecl(MD) \wedge targetType(MD, P) \wedge delDelegationType_{content}(MD, P)}{del_{content}(MD, P)}$$

Darauf aufbauend ergibt sich wiederum folgende Regel für eine gesamte *Methoden-Delegation MD* innerhalb eines *Content-Proxies P*.

$$\frac{call_{simple}(MD, P) \wedge del_{content}(MD, P) \wedge methodMatch_{simple}}{delegation_{content}(MD, P)}$$

Da auch ein *Content-Proxy* fehlschlagende *Methoden-Delegationen* enthalten kann, muss ebenfalls gelten:

$$\frac{call_{simple}(MD, P) \wedge methodErr(MD)}{delegation_{content}(MD, P)}$$

Für die Gesamtheit der *Methoden-Delegationen* innerhalb eines *Content-Proxies P* muss dann gelten:

$$\frac{\forall MD \in P.dels : delegation_{content}(MD, P)}{delegations_{content}(P)}$$

Die Menge der *Content-Proxies*, die mit dem *Source-Typ* S und der Mengen von *Target-Typen* TM erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{content}(S, TM) := \left\{ P \;\middle|\; \begin{array}{l} proxy(P, S) \wedge \\ targets_{single}(P, TM) \wedge \\ delegations_{content}(P) \end{array} \right\}$$

Container-Proxy

Die Voraussetzung für die Erzeugung eines *Container-Proxies* vom Typ S aus einem *Target-Typ* T ist $S \Rightarrow_{container} T$. Damit ist der *ContainerTypeMatcher* der Basis-Matcher für den *Container-Proxy*.

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

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

Listing 3.7: Container-Proxy für MedCabniet

Durch die *Methoden-Delegation* dieses *Container-Proxies* findet eine Delegation nur dann statt, wenn die Methoden `getDescription` auf dem Feld `med` des *Source-Typ* aufgerufen wird. Diese wird dann an den *Target-Typen* `MedCabniet` delegiert.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.6 zu entnehmen. [5]

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

Ein *Container-Proxy* enthält, wie die vorher beschriebenen *Proxies*, genau einen *Target-Typ*. Die Eigenschaften der *Delegationsmethoden* innerhalb der einzelnen *Methoden-Delegationen*

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

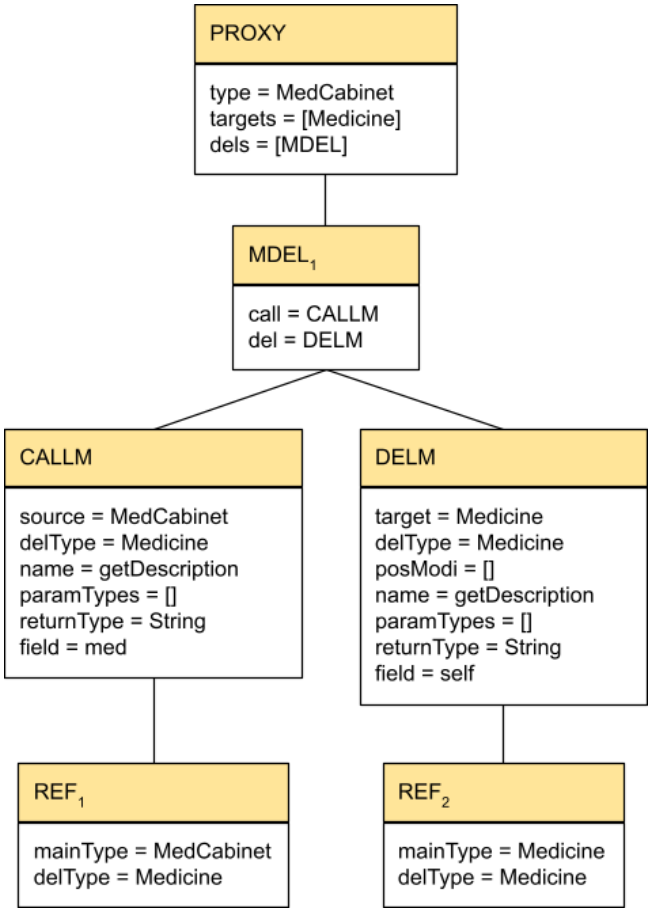


Abbildung 3.6: AST für das Beispiel zum Container-Proxy

gleichen denen aus dem *Sub-Proxy*.

In der *aufgerufenen Methode* einer einzelnen *Methoden-Delegation MD* muss das Attribut `source` ebenfalls wie im *Sub-Proxy* mit dem *Source-Typen* des *Proxies* P übereinstimmen ($sourceType(MD, P)$), allerdings muss das Attribut `delType` der *aufgerufenen Methode* ein Matching zu einem der *Target-Typen* des *Proxies* aufweisen:

$$\frac{\exists T \in P.targets : MD.call.delType \Rightarrow_{internCont} T}{callDelegationType_{container}(MD)}$$

Folglich werden auch die Eigenschaften einer *aufgerufenen Methode* in einer *Methoden-Delegation MD* im *Container-Proxy* durch eine andere Regel beschrieben als beim *Sub-Proxy*:

$$\frac{callDecl(MD) \wedge callDelegationType_{container}(MD, P) \wedge sourceType(MD, P)}{call_{container}(MD, P)}$$

Darauf aufbauend ergibt sich wiederum folgende Regel für eine gesamte *Methoden-Delegation MD* innerhalb eines *Container-Proxies P*.

$$\frac{call_{container}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{simple}}{delegation_{container}(MD, P)}$$

Da auch ein *Container-Proxy* fehlschlagende *Methoden-Delegationen* enthalten kann, muss ebenfalls gelten:

$$\frac{call_{container}(MD, P) \wedge methodErr(MD)}{delegation_{container}(MD, P)}$$

Für die Gesamtheit der *Methoden-Delegationen* innerhalb eines *Container-Proxies P* muss dann gelten:

$$\frac{\forall MD \in P.dels : delegation_{container}(MD, P)}{delegations_{container}(P)}$$

Die Menge der *Container-Proxies*, die mit dem *Source-Typ S* und der Menge von *Target-Typen TM* erzeugt werden, wird durch die folgende Funktion beschrieben.

$$proxies_{container}(S, TM) := \left\{ P \;\middle|\; \begin{array}{l} proxy(P, S) \wedge \\ target_{single}(P, TM) \wedge \\ delegations_{container}(P) \end{array} \right\}$$

Struktureller Proxy

Die Voraussetzung für die Erzeugung eines *strukturellen Proxies* vom *required Typ R* aus einem Target-Typ T ist $R \Rightarrow_{struct} T$. Damit ist der *StructuralTypeMatcher* der Basis-Matcher für den *strukturellen Proxy*.

Der *strukturelle Proxy* ist der einzige *Proxy*, der mit mehreren *Target-Typen* erzeugt werden kann.

Beispiel Als Beispiel werden die Typen `MedicalFireFighter`, `Doctor` und `FireFighter` verwendet. Dabei ist `MedicalFireFighter` der *Source-Typ* des *Proxies* und die Menge der anderen beiden Typen bilden die *Target-Typen* des *Proxies*. Da der *Source-Typ* zu den *Target-Typen* ein Matching der Form `MedicalFireFighter` $\Rightarrow_{struct}$ `FireFighter` bzw. `MedicalFireFighter` $\Rightarrow_{struct}$ `Doctor` aufweist, kann ein *struktureller Proxy* erzeugt werden. Ein solcher ist in folgendem Listing aufgeführt.

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

Listing 3.8: Struktureller Proxy für MedicalFireFighter

In diesem Beispiel wird der Methodenaufruf der Methode `heal` auf dem *Proxy* an die Methode `heal` des Typs `Doctor` delegiert. Analog dazu würde ein Aufruf der Methode `extinguishFire` auf dem Proxy an die Methode `extinguishFire` des Typs `FireFighter` delegiert werden. Die Methoden stimmen jeweils strukturell überein.

Der AST mit den dazugehörigen Attributen ist Abbildung 3.7 zu entnehmen. [6]

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

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

$$\frac{|P.targets| = |TM| \wedge \forall T \in P.targets : T \in TM \wedge \exists MD \in P.dels : MD.del.target = T}{targets_{multi}(P, TM)}$$

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

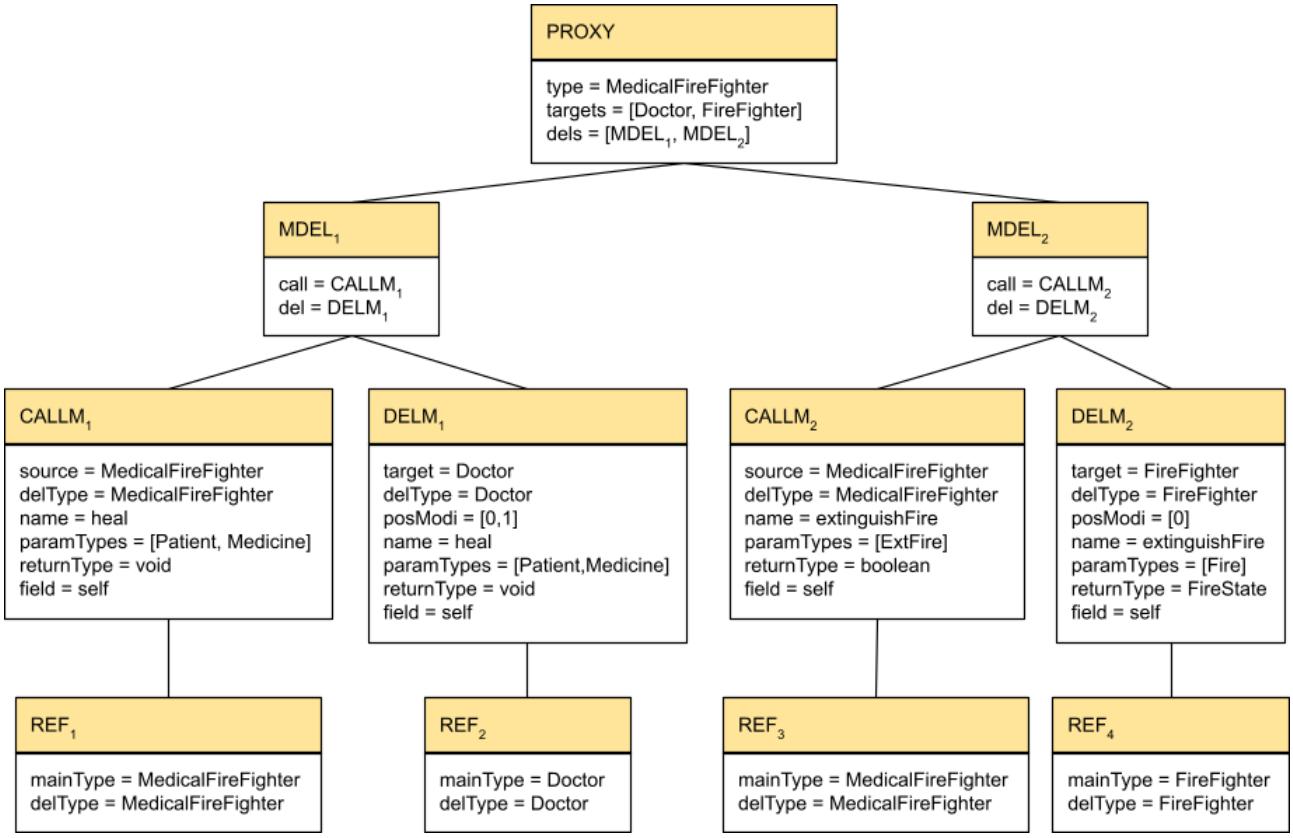


Abbildung 3.7: AST für das Beispiel zum strukturellen Proxy

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

$$\frac{D.returnType \Rightarrow_{internStruct} C.returnType}{returnMatch(C, D)}$$

Weiterhin muss für die Parameter-Typen gelten:

$$\frac{C.paramTypes[i] \Rightarrow_{internStruct} D.paramTypes[D.posModi[i]]}{posModiMatch(C, D, i)}$$

$$\frac{\forall i \in \{0, ..., C.paramCount - 1\} : posModiMatch(C, D, i)}{paramsMatch(C, D)}$$

Das strukturelle Matching zwischen einer einer *aufgerufenen Methode* C und einer *Delegationsmethode* D kann darauf aufbauend wie folgt beschrieben werden.

$$\frac{returnMatch(C, D) \wedge paramsMatch(C, D)}{methodMatch_{struct}(C, D)}$$

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

$$\frac{call_{simple}(MD, P) \wedge del_{simple}(MD, P) \wedge methodMatch_{struct}(MD.call, MD.del)}{delegation_{struct}(MD, P)}$$

In einem *strukturellen Proxy* muss für jede Methode m des *Source-Typen* genau eine *Methoden-Delegation* mit der Methode m als *aufgerufene Methode* existieren:

$$\frac{|methods(P.type)| = P.dels.len}{delegationCount_{struct}(P)}$$

Für die Gesamtheit der *Methoden-Delegationen* aus einem *strukturellen Proxy* P muss dann gelten:

$$\frac{delegationCount_{struct}(P) \wedge \forall MD \in P.dels : delegation_{struct}(MD, P)}{delegations_{struct}(P)}$$

Die Menge der *strukturellen Proxies*, die mit dem *Source-Typ* S und der Menge von *Target-Typen* TM erzeugt werden, wird aufbauend auf den oben genannten Regeln durch die folgende

Funktion beschrieben.

$$proxies_{struct}(S, TM) := \left\{ P \;\middle|\; \begin{array}{l} proxy(P, S) \wedge \\ targets_{multi}(P, TM) \wedge \\ delegations_{struct}(P) \end{array} \right\}$$

Allgemeine Generierung von Proxies

Die Proxy-Funktionen der einzelnen Proxy-Arten werden zur Beschreibung einer allgemeine Funktion für die Generierung der *Proxies* verwendet. Dazu sind die Proxy-Arten zusammen mit den dazugehörigen *Matchingrelationen* und den Namen der Funktionen zur Generierung des jeweiligen *Proxies* in Tabelle 3.4 noch einmal aufgeführt.

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

Tabelle 3.4: Proxy-Arten mit Matchingrelationen und Proxy-Funktionen

Die im Abschnitt 3.2.1 erwähnte Funktion $proxies(S, TM)$ kann darauf aufbauend für einen *Source-Typ* S und eine Menge von *Target-Typen* TM wie folgt beschrieben werden.

$$proxies(S, TM) := \left\{ \begin{array}{ll} proxies_{sub}(S, TM) & \text{wenn } |TM| = 1 \wedge \\ & \forall T \in TM : S \Rightarrow_{spec} T \\ & \\ proxies_{content}(S, TM) & \text{wenn } |TM| = 1 \wedge \\ & \forall T \in TM : S \Rightarrow_{content} T \\ & \\ proxies_{container}(S, TM) & \text{wenn } |TM| = 1 \wedge \\ & \forall T \in TM : S \Rightarrow_{container} T \\ & \\ proxies_{struct}(S, TM) & \text{wenn } |TM| > 0 \wedge \\ & \forall T \in TM : S \Rightarrow_{struct} T \end{array} \right\}$$

3.2.4 Anzahl struktureller Proxies innerhalb einer Bibliothek

Die Generierung der *strukturellen Proxies* für ein *required Typ* R aus der Bibliothek L erfolgt während des *Explorationsprozesses* auf Basis der Mengen von *provided Typen* aus $cover(R, L)$ (siehe Abschnitt 3.1.3). Bezüglich dieser Mengen aus $cover(R, L)$ gilt jedoch folgendes Theorem[7]:

Theorem 1. *Sei R ein* required Typ *innerhalb einer Bibliothek L. Ein* struktureller Proxy *für R lässt sich nur aus den Mengen $TM \in cover(R, L)$ generieren, für die gilt:*

$$|TM| \leq |methods(R)|$$

Mit einer Menge $TM \in cover(R, L)$, für die die Bedingung aus Theorem 1 eingehalten wird, können durchaus mehrere *Proxies* erzeugt werden. Das ist dann der Fall, wenn mehrere der Methoden, die in den *provided Typen* aus TM deklariert wurden, mit einer Methode aus R strukturell übereinstimmen, oder wenn für eine Methode m aus R und eine Methode m' aus einem der *Target-Typen* $|matchingParams(m, m')| > 1$ gilt (siehe Abschnitt 3.1.2).

Die Anzahl der *strukturellen Proxies* für einen *required Typ* R mit einer bestimmten Menge von *Target-Typen* ist somit von der Anzahl der Methoden abhängig, die in einem der *Target-Typen* deklariert wurden und strukturell mit den Methoden aus R übereinstimmen.

Die Menge der Methoden eines *provided Typs* T, die strukturell mit einer Methode m übereinstimmen, wird über die Funktion $structM_{target}$ beschrieben.

$$structM_{target}(m, T) := \left\{ m' \;\middle|\; m' \in methods(T) \wedge m \Rightarrow_{method} m' \right\}$$

Darauf aufbauend wird die Menge der Methoden einer Menge von *provided Typen* TM, die strukturell mit einer Methode m übereinstimmen, über die Funktion $structM_{targetset}$ beschrieben.

$$structM_{targetset}(m, TM) := \left\{ m' \;\middle|\; \exists T \in TM : m' \in structM_{target}(m, T) \right\}$$

[7] Der Beweis ist in Anhang F zu finden.

Beispiel 2 Aufbauend auf dem vorherigen Beispiel 1 ergeben sich für die Menge der *Target-Typen* {`Leave`, `Come`} und die beiden Methoden des *required Typs* `Greeting` folgende Mengen von übereinstimmenden Methoden über die Funktion $structM_{targetset}$:

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

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

Sei R ein *required Typ* und TM eine Menge von *provided Typen* innerhalb einer Bibliothek L mit $TM \in cover(R, L)$. Dann bildet die Funktion *structMSets* die Menge von Mengen der Methoden aus den Elementen aus TM ab, die mit jeweils einer Methode aus R gematcht werden können.

$$structMSets(R, TM) := \left\{ M \;\middle|\; \begin{array}{l} \exists m \in methods(R) : \\ M = structM_{targetset}(m, TM) \end{array} \right\}$$

Für die Bildung eines *Proxies* wird aus jedem Element der Menge $structMSets(R, TM)$ genau ein Element als *Delegationsmethode* verwendet.

Beispiel 3 Ausgehend von Beispiel 2 lassen sich die folgenden vier *Proxies* mit den *Target-Typen* `Leave` und `Come` erzeugen.

```
proxy Greeting with [Come, Leave]{
        Greeting.hello():String → Come.hello():String
        Greeting.bye():String → Leave.bye():String
}

proxy Greeting with [Come, Leave]{
        Greeting.hello():String → Come.goodMorning():String
        Greeting.bye():String → Leave.bye():String
}

proxy Greeting with [Come, Leave]{
        Greeting.hello():String → Leave.bye():String
```

```
        Greeting.bye():String → Come.hello():String
}

proxy Greeting with [Come, Leave]{
        Greeting.hello():String → Leave.bye():String
        Greeting.bye():String → Come.goodMorning():String
}
```

Die Anzahl aller möglichen *strukturellen Proxies* für ein *required Typ* R aus einer Menge von *Target-Typen* TM sei näherungsweise über die Funktion $proxyCount(R, TM)$ beschrieben. Dass es sich hierbei lediglich um eine Annäherung handelt liegt daran, dass eine Methode dm mit $dm \in M_1 \cup ... \cup M_n$ und $\{M_1, ..., M_n\} = structMSets(R, TM)$ innerhalb eines Proxy maximal einmal als *Delegationsmethode* verwendet werden darf. Es ist jedoch möglich, dass es zwischen den Mengen $M_1, ..., M_n$ Überschneidungen gibt (siehe vorheriges Beispiel). Darüber hinaus blenden die oben genannten Funktionen zur Ermittlung der matchenden Methoden die Anzahl der für die *Methoden-Delegation* möglichen Parameter-Sortierungen aus *matchtingParams* aus.

Unter der Annahme, dass es nur eine mögliche Sortierung der Parameter für die matchenden Methoden gibt, gilt:

$$proxyCount(R, TM) \leq \prod_{i=1}^{n} |structM_{targetset}(m_i, TM)| \left| \left\{ \begin{array}{l} m_1, \\ ..., \\ m_n \end{array} \right\} = methods(R) \right.$$

Durch mehrere Möglichkeiten der Sortierung der Parameter würde sich dieser Wert nochmals erhöhen. Auf eine genaue Formel wird an dieser Stelle verzichtet, da hier lediglich eine Vorstellung davon vermittelt werden soll, wie große die Anzahl der möglichen *strukturellen Proxies* mit einer bestimmten Mengen von *Target-Typen* ist.

Da innerhalb einer Bibliothek L mehrere Mengen von *Target-Typen* zur Bildung eines *Proxies* für einen *required Typ* R infrage kommen (siehe Funktion *cover*) muss die Anzahl der *strukturellen Proxies* über die Funktion *proxyCount* für alle Elemente aus $cover(R, L)$ ermittelt und summiert werden. Dabei muss jedoch beachtet werden, dass die Funktion *cover* nicht die Bedingung aus Theorem 1 abbildet. Daher muss die Funktion für diesen Sachverhalt, wie folgt

näherungsweise definiert werden:

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

3.3 Semantische Evaluation

Das Ziel der *semantischen Evaluation* ist es, einen der *Proxies*, die aus den Mengen von *Target-Typen*, die im Rahmen der *strukturellen Evaluation* erzeugt werden können, hinsichtlich der vordefinierten Testfälle zu evaluieren. Da der gesamte *Explorationsprozess* zur Laufzeit des jeweiligen Programms durchgeführt wird, stellt sie hinsichtlich der nicht-funktionalen Anforderungen eine zeitkritische Komponente dar.

Da die Anforderungen an den gesuchten *Proxy* mit Bedacht spezifiziert werden müssen, ist es irrelevant, ob es mehrere *Proxies* gibt, die hinsichtlich der vordefinierten Testfällen positiv geprüft werden können. Es ist ausreichend lediglich ein *Proxy* zu finden, dessen Semantik zu positiven Ergebnissen hinsichtlich aller vordefinierten Testfälle führt.

3.3.1 Besonderheiten der Testfälle

Bei den vordefinierten Tests handelt es sich auf formaler Ebene um Typen, die eine `eval`-Methode mit der Struktur `boolean eval( proxy )` anbieten, welche einen *Proxy* als Parameter erwartet und ein Objekt vom Typ `boolean` zurückgibt. Weiterhin verfügt ein Test über ein Attribut `triedMethodCalls`, in dem eine Liste von Methodennamen, die bei der Durchführung der `eval`-Methode auf den *Proxies* aufgerufen wurden, hinterlegt ist.

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

1. Vor dem Aufruf einer Methode auf dem als Parameter übergebenen *Proxy*, wird der Name dieser Methode in der Liste im Feld `triedMethodCalls` ergänzt.

2. Wenn der *Proxy* den Test besteht, wird der Wert `true` zurückgegeben. Anderenfalls wird der Wert `false` zurückgegeben.

Beispiel 4 In folgendem Listing 3.9 ist eine `eval`-Methode aufgeführt, die die oben genannten Bedingungen erfüllt. Es sei davon auszugehen, dass der als Parameter übergebene *Proxy* eine Methode mit der Struktur *Integer* $add(Integer, Integer)$ anbietet.

```
1 function eval( proxy ){
2   res = 0
3   triedMethodCalls.add( "add" )
4   res = proxy.add(1, 1)
5   return res == 2;
6 }
```

Listing 3.9: Beispielhafte Implementierung einer `eval`-Methode

3.3.2 Algorithmus für die semantische Evaluation

Während des *Explorationsprozesses* soll aus den *provided Typen* in einer Bibliothek L zu einem vorgegebenen *required Type* R ein Proxy generiert und evaluiert werden. Die Mengen der *Target-Typen* auf deren Basis mehrere *Proxies* erzeugt werden können, wurden in Abschnitt 3.2.4 mithilfe der Funktion $cover(R, L)$ beschrieben. In diesem Zusammenhang wurde in Lemma 1 eine Restriktion bzgl. der Anzahl möglicher *Target-Typen* eines *Proxies* beschrieben. Darauf aufbauend, kann die maximale Anzahl von *Target-Typen* eines *Proxies* für R wie folgt definiert werden:

$$maxTargets(R) := |methods(R)|$$

Das in dieser Arbeit beschriebene Konzept basiert auf der Annahme, dass der gesamte Anwendungsfall - oder Teile davon - , der mit der vordefinierten Struktur (*required Typ*) und den vordefinierten Tests abgebildet werden soll, schon einmal genauso oder so ähnlich in dem gesamten System implementiert wurde. Aus diesem Grund kann für die *semantische Evaluation* davon ausgegangen werden, dass die erfolgreiche Durchführung aller relevanten Tests umso wahrscheinlicher ist, je weniger *Target-Typen* im *Proxy* enthalten sind.

Die Mengen innerhalb einer Menge C mit einer Mächtigkeit a seien durch folgende Funkti-

on beschrieben:

$$targetSets(C, a) := \left\{ \; TM \;\middle|\; TM \in C \wedge |TM| = a \; \right\}$$

Ausgehend von einer Bibliothek L kann der Algorithmus für die *semantische Evaluation* der *Proxies*, die für einen *required Typ* R (Parameter `R`) mit den Mengen der *Target-Typen* $cover(R, L)$ (Parameter `T`) erzeugt werden können, und einer Menge von Tests (Parameter `tests`) über die Methode `semanticEval` wie folgt im Pseudo-Code beschrieben werden. Die globale Variable `passedTests` enthält dabei die Anzahl der für den aktuell zu überprüfenden *Proxy* erfolgreich durchgeführten Tests. Außerdem sei davon auszugehen, dass die Funktionen aus Abschnitt 3.2.3 wie beschrieben definiert sind.

```
1  passedTests = 0
2
3  function semanticEval( R, T, tests ){
4    for( anzahl = 1; anzahl <= maxTargets(R); i++ ){
5      for( targets : targetSets(T, anzahl) ){
6        relProxies = proxies(R, targets)
7        proxy = evalProxies( relProxies, tests )
8        if( proxy != null ){
9          // validen Proxy gefunden
10         return proxy
11       }
12     }
13   }
14   // kein validen Proxy gefunden
15   return null;
16 }
17
18 function evalProxies(proxies, tests){
19   for( proxy : proxies ){
20     passedTests = 0
21     evalProxy(proxy, tests)
22     if( passedTests == tests.size ){
23       // validen Proxy gefunden
24       return proxy
25     }
26   }
27   // kein validen Proxy gefunden
28   return null
29 }
30
```

```
31 function evalProxy(proxy, tests){
32   for( test : tests ){
33     if( !test.eval( proxy ) ){
34       // wenn ein Test fehlschlaegt, dann entspricht der
35       // Proxy nicht den semantischen Anforderungen
36       return
37     }
38     passedTests = passedTests + 1
39   }
40 }
```

Listing 3.10: Semantische Evaluation ohne Heuristiken

Die Dauer der Laufzeit der in Listing 3.10 definierten Funktionen hängt maßgeblich von der Anzahl der *Proxies* ab, die für den *required Typ* R in der Bibliothek L erzeugt werden können (siehe auch Abschnitt 3.2.4 Funktion *libProxyCount*). Im schlimmsten Fall müssen alle *Proxies* generiert werden und hinsichtlich der vordefinierten Tests geprüft werden. Um die Anzahl dieser *Proxies* zu reduzieren, werden die im folgenden Abschnitt beschriebenen Heuristiken verwendet.

3.4 Heuristiken

Als Heuristiken werden in dieser Arbeit Verfahren bezeichnet, durch die die Lösung eines Problems beschleunigt werden kann, indem neu gewonnene Erkenntnisse beim Finden der Lösung berücksichtigt werden. Konkret bedeutet dies, dass die oben beschriebene *semantische Evaluation* durch diese Verfahren beschleunigt werden soll.

Die Heuristiken, die in den Abschnitten 3.4.1 und 3.4.2 beschrieben werden, haben zum Ziel, die Reihenfolge, in der die *Proxies* hinsichtlich der vordefinierten Tests geprüft werden, so anzupassen, dass ein valider *Proxy* möglichst früh geprüft wird. Die dritte Heuristik, die im Abschnitt 3.4.3 beschrieben wird, beschreibt ein Ausschlussverfahren.

Für die Verwendung der Heuristiken wird der Algorithmus aus Listing 3.10 erweitert. Diese Erweiterung beinhaltet die Verwaltung der neu gewonnenen Erkenntnisse sowie die Anwendung der Heuristiken auf die zu generierenden bzw. generierten *Proxies*.

In den folgenden Abschnitten werden die Heuristiken und die dafür notwendigen Anpassungen

an den jeweiligen Funktionen beschrieben. Der Pseudo-Code für die *semantische Evaluation* inklusive der Verwendung aller vorgestellten Heuristiken ist im Anhang B zu finden.

3.4.1 Beachtung des Matcherratings (LMF)

Bei dieser Heuristik, welche den Namen *low matcherrating first* (kurz: *LMF*) trägt, werden die Mengen von *Target-Typen*, aus denen die *Proxies* erzeugt werden, auf der Basis eines so genannten *Matcherratings* bewertet. Bei dem *Matcherrating* einer solchen Menge handelt es sich um einen numerischen Wert, auf dessen Basis entschieden werden kann, für welche Menge von *Target-Typ* die Generierung und Prüfung der *Proxies* zuerst vollzogen werden soll.

Um das *Matcherrating* zu ermitteln, wird für jede Matchingrelation bzw. für jeden Matcher aus Abschnitt 3.1.2 ein *Basisrating* vergeben. Folgende Funktion beschreibt das *Basisrating* für das Matching zweier Typen S und T:

$$base(S, T) := \begin{cases} 100 \text{ wenn } S \Rightarrow_{exact} T \\ 200 \text{ wenn } S \Rightarrow_{gen} T \\ 200 \text{ wenn } S \Rightarrow_{spec} T \\ 300 \text{ wenn } S \Rightarrow_{contained} T \\ 300 \text{ wenn } S \Rightarrow_{container} T \end{cases}$$

Dabei ist zu erwähnen, dass einige der oben genannten Matcher über dasselbe *Basisrating* verfügen. Das liegt daran, dass sie technisch jeweils gemeinsam umgesetzt wurden.[8]

Wie an der Funktion *base* zu erkennen ist, wird das *Matcherrating* für Typen, die über den *StructuralTypeMatcher* gematcht wurden, nicht spezifiziert. Dieses muss berechnet werden. Die Basis dafür bildet ein *Matcherrating*, welches für die gematchten Methoden ermittelt wird. Hierzu sei die Funktion $bases_{method}$ für zwei Methoden mR und mT mit $mR \Rightarrow_{method} mT$ wie

[8] Der *GenTypeMatcher* und der *SpecTypeMatcher* wurden gemeinsam in der Klasse `GenSpecTypeMatcher` umgesetzt. Der *ContentTypeMatcher* und der *ContainerTypeMatcher* wurden gemeinsam in der Klasse `WrappedTypeMatcher` umgesetzt. (siehe angehängter Quellcode auf dem beiliegenden Datenträger)

folgt definiert:

$$bases_{method}(mR, mT) := \begin{array}{l|l} base(ret(mR), ret(mT)) \cup & \{pR_1, ..., pR_n\} = params(mR) \wedge \\ \bigcup_{i=1}^{n} base(pR_i, pT_i) & \{pT_1, ..., pT_n\} = params(mT) \end{array}$$

Darauf aufbauend kann die Funktion *mRating* für die beiden Methoden mR und mT definiert werden. Hierzu seien folgende Hilfsfunktionen definiert:

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

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

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

In dieser Arbeit werden vier Varianten für diese Definition von *mRating* vorgeschlagen, die in Abschnitt 5.3 evaluiert werden sollen.

Variante 1: Durchschnitt ($mRating_1$)

$$mRating_1(mR, mT) := \frac{sum(base_{method}(mR, mT))}{|params(mR)| + 1}$$

Variante 2: Maximum ($mRating_2$)

$$mRating_2(mR, mT) := max(bases_{method}(mR, mT))$$

Variante 3: Minimum ($mRating_3$)

$$mRating_3(mR, mT) := min(bases_{method}(mR, mT))$$

Variante 4: Durchschnitt aus Minimum und Maximum ($mRating_4$)

$$mRating_4(mR, mT) := \frac{max(bases_{method}(mR, mT)) + min(bases_{method}(mR, mT))}{2}$$

In einem *provided Typ* T sind mitunter mehrere Methoden deklariert, die ein Matching zu einer Methode m aufweisen. Für die Bestimmung des *Matcherratings* sei hierbei nur das kleinste *Matcherrating* jener Methoden aus P relevant. Das *minimale Matcherrating* einer solchen Methode wird durch folgende Funktion beschrieben[9]

$$minMRating(m, T) := \begin{array}{l|l} min(mRating_*(m'_1), & \{m'_1, ..., m'_n\} = \\ ..., mRating_*(m'_n)) & structM_{target}(m, T) \end{array}$$

Für einen *required Typ* R und einem *provided Typ* T wird die Menge dieser *minimalen Matcherratings* je Methode $m \in structM(R)$ über folgende Funktion definiert:

$$minMRatings(R, T) := \left\{ \; minMRating(m, T) \;\middle|\; m \in structM(R, T) \; \right\}$$

In einer Bibliothek L wird die Ermittlung des *Matcherratings* eines *required Typs* R und einer Menge von *provided Typen* $\{T_1, ..., T_n\}$ mit $\{T_1, ..., T_n\} \in cover(R, L)$ über die Funktion *rating* beschrieben. Auch hierfür werden in dieser Arbeit insgesamt 4 Varianten vorgeschlagen, die in Kapitel 5 evaluiert werden sollen.

Variante 1: Durchschnitt ($rating_1$)

$$rating_1(R, \{T_1, ..., T_n\}) := \frac{sum(minMRatings(R, T_1), ..., minMRatings(R, T_n))}{\sum_{i=1}^{n} |structM(R, T_i)|}$$

Variante 2: Maximum ($rating_2$)

$$rating_2(R, \{T_1, ..., T_n\}) := max(minMRatings(R, T_1), ..., minMRatings(R, T_n))$$

Variante 3: Minimum ($rating_3$)

$$rating_3(R, \{T_1, ..., T_n\}) := min(minMRatings(R, T_1), ..., minMRatings(R, T_n))$$

[9] Da die Varianten der Funktion *mRating* in *minMRating* flexibel verwendet werden können, wurde für *mRating* das Subskript $*$ verwendet.

Variante 4: Durchschnitt aus Minimum und Maximum ($rating_4$)

$$rating_4(R, \{T_1, ..., T_n\}) := \frac{\begin{array}{c} min(minMRatings(R, T_1), ..., minMRatings(R, T_n)) \\ + max(minMRatings(R, T_1), ..., minMRatings(R, T_n)) \end{array}}{2}$$

Da die Funktion *rating* von *mRating* abhängt und für *mRating* 4 Varianten vorgeschlagen wurden, ergeben sich insgesamt 16 Varianten für die Definition von *rating*. Diese Varianten (1.1 - 4.4) sind in der Tabelle 3.5 mit den Kombinationen der Varianten für *mRating* und *rating* aufgeführt.

Variante	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4	4.1	4.2	4.3	4.4
$rating_*$	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
$mRating_*$	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Tabelle 3.5: Varianten für die Ermittlung des Matcherratings einer Menge von *provided Typen*

Zur Anwendung der Heuristik muss das *Matcherrating* bei der Generierung der *Proxies* aus den jeweiligen Mengen von *provided Typen* beachtet werden. Dabei sollte die Liste der Mengen von *provided Typen*, die über die Funktion *targetSets* abgebildet wird (siehe Abschnitt 3.3.2) und über die in der Methode `semanticEval` (siehe Listing 3.10) iteriert wird, entsprechend dem *Matcherrating* sortiert werden. Dadurch werden in der Methode `evalProxies` (siehe Listing 3.10) zuerst die *Proxies* generiert und geprüft, die auf Basis einer Menge von *provided Typen* mit dem kleinsten *Matcherrating* erzeugt wurden.

Listing 3.11 zeigt die Anpassungen der Methode `relevantProxies` auf Basis der Implementierung der *semantischen Evaluation* aus Listing 3.10. Für die Sortierung der Liste von *Proxies* wurde in der Methode `LMF` exemplarisch das Bubble-Sort-Verfahren verwendet.

```
1 function semanticEval( R, T, tests ){
2   for( anzahl = 1; anzahl <= maxTargets(R); i++ ){
3     targetSets = targetSets(T, anzahl)
4     sortedSets = LMF( R, targetSets )
5     for( targets : sorted ){
6       relProxies = proxies(R, targets)
7       proxy = evalProxies( relProxies, tests )
8       if( proxy != null ){
9         // validen Proxy gefunden
```

```
10          return proxy
11        }
12      }
13    }
14    // kein validen Proxy gefunden
15    return null;
16  }
17
18  function LMF( R, targets ){
19    for ( n=targets.size(); n>1; n--){
20      for( i=0; i<n-1; i++){
21        if( rating(R, targets[i] ) < rating(R, targets[i+1] ) ){
22          tmp = targets[i]
23          targets[i] = targets[i+1]
24          targets[i+1] = tmp
25        }
26      }
27    }
28    return targets
29  }
```

Listing 3.11: Semantische Evaluation mit Heuristik LMF

3.4.2 Beachtung positiver Tests (PTTF)

Das Testergebnis, welches bei Applikation eines Testfalls für einen *Proxy* ermittelt wird, ist maßgeblich von den *Methoden-Delegationen* des *Proxies* abhängig. Jede *Methoden-Delegation MD* enthält einen Typ in dem die *Delegationsmethode* deklariert wurde. Dieser Typ befindet sich im Attribut `MD.del.delTyp`. Im Fall der *sturkturellen Proxies*, handelt es sich bei diesem Typ um einen der *Target-Typen* des *Proxies*. Bezüglich der *Target-Typen* möglicher *struktureller Proxies* gilt folgendes Theorem[10]

Theorem 2. *Sei R ein* required Typ *aus einer Bibliothek L. Sei weiterhin $C = cover(R, L)$. Ferner seien $TM \in C$ und $TM' \in C$ mit $proxies_{struct}(R, TM) \neq \emptyset$ sowie $proxies_{struct}(R, TM') \neq \emptyset$ und $|TM| < |TM'|$ gegeben.*

[10]Der Beweis ist in Anhang F zu finden.

Dann gilt:

$$\forall T \in TM : \exists TM'' \in targetSets(C, |TM'|) : proxies_{struct}(R, TM'') \neq \emptyset \wedge T \in TM''$$

In Bezug auf die *Proxies*, die bei der *semantischen Evaluation* in mehreren Durchläufen geprüft werden sollen, bedeutet das Folgendes: Die einzelnen *Target-Typen* der *Proxies*, die innerhalb eines Durchlaufs geprüft wurden, sind auch in den *Target-Typen* der *Proxies* enthalten, die in einem späteren Durchlauf geprüft werden - sofern solche *Proxies* überhaupt existieren.

Für die in diesem Abschnitt beschriebene Heuristik mit dem Namen *positive tested targets first* (kurz: *PTTF*) ist das Ergebnis einzelner Tests in Bezug auf einen *Proxy P* relevant. Wenn ein Testfall mit einem *Proxy P* erfolgreich durchgeführt wurde, dann sollte die Reihenfolge der zu prüfenden *Proxies* späterer Durchläufe so angepasst werden, dass die *Proxies*, die einen *Target-Typen* des *Proxies P* verwenden, im weiteren Verlauf zuerst geprüft werden.

Dafür sind auf Basis von Listing 3.10 mehrere Anpassungen bzgl. der Implementierung der Methode `evalProxies` von Nöten:

1. Die *Target-Typen* der *Proxies*, mit denen mind. ein Testfall erfolgreich durchgeführt werden konnte, müssen in einer globalen Variable (`prioTargets`) hinterlegt werden.

2. Die Liste der *Proxies*, die der Methode `evalProxies` als Parameter übergeben wird, muss so sortiert werden, dass die *Proxies*, mit den *Target-Typen*, die in der globalen Variable (`prioTargets`) hinterlegt wurden, zuerst getestet werden.

3. Die Liste der *Proxies*, über die innerhalb der Methode `evalProxies` iteriert wird, kann bzgl. ihrer Reihenfolge bereits dann optimiert werden, wenn mind. einer der Testfälle für den aktuellen *Proxy* erfolgreich durchgeführt wurde. Dazu müssen jedoch die *Proxies*, die bereits innerhalb der Methode getestet wurden, in einer lokalen Variable (`tested`) hinterlegt werden. Dann kann die Methode rekursiv mit den *Proxies*, die noch nicht getestet wurden, aufgerufen werden. So werden die darin enthaltenen Elemente aufgrund der 2. Anpassung erneut sortiert.

In Listing 3.12 sind die oben genannten Anpassungen im Vergleich zu Listing 3.10 zu entnehmen.

```
1  prioTargets = []
2
3  function evalProxies( proxies, tests ){
4   tested = []
5   sorted = PTTF( proxies )
6   for( proxy : sorted ){
7    passedTests = 0
8    evalProxy( proxy, tests )
9    if( passedTests == tests.size ){
10    // validen Proxy gefunden
11    return proxy
12   }
13   else{
14    tested.add( proxy )
15    if( passedTests > 0 ){
16     prioTargets.addAll( proxy.targets )
17     // noch nicht evaluierte Proxies ermitteln
18     leftProxies = sorted.removeAll( testedProxies )
19     return evalProxies( leftProxies, tests )
20    }
21   }
22  }
23  // kein validen Proxy gefunden
24  return null
25 }
26
27 function PTTF( proxies ){
28  for( n=proxies.size ; n>1; n--){
29   for( i=0; i<n-1; i++){
30    targetsFirst = proxies[i].targets
31    targetsSecond = proxies[i+1].targets
32    if( !prioTargets.contains( targetsFirst )
33         && prioTargets.contains( targetsSecond ) ){
34     tmp = proxies[i]
35     proxies[i] = proxies[i+1]
36     proxies[i+1] = tmp
37    }
38   }
39  }
40  return proxies
41 }
```

Listing 3.12: Semantische Evaluation mit Heuristik PTTF

3.4.3 Beachtung fehlgeschlagener Methodenaufrufe (BL_NMC)

Diese Heuristik mit dem Namen *blacklist negative method calls* (kurz: *BL_NMC*) beschreibt ein Ausschlussverfahren. Das bedeutet, dass bestimmte *Proxies* auf der Basis von Erkenntnissen, die während der *semantischen Evaluation* entstanden sind, für den weiteren Verlauf ausgeschlossen werden. Dadurch soll die Prüfung eines *Proxies*, dessen *Methoden-Delegationen* ohnehin nicht zum gewünschten Ergebnis führen, verhindert werden.

Die Heuristik zielt darauf ab, *Methoden-Delegationen*, die immer fehlschlagen, zu identifizieren. Wurde eine solche *Methoden-Delegation* gefunden, können alle *Proxies*, die diese *Methoden-Delegation* enthalten von der weiteren Exploration ausgeschlossen werden.

Die *Methoden-Delegationen*, die auf der Basis der folgenden Heuristik aussortiert werden sollen, werden zu diesem Zweck in einer globalen Variable (`mdelBlacklist`) gehalten. Aus einer Liste von *Proxies* können darauf aufbauend diejenigen *Proxies* entfernt werden, die eine jener *Methoden-Delegationen* enthalten. Für die Implementierung wird im Folgenden davon ausgegangen, dass die Methoden eines *required Typen* über den Namen identifiziert werden können.

Ausgehend vom Algorithmus der *semantischen Evaluation* (siehe Listing 3.10) muss die Methode `evalProxy` für das Füllen der globalen Variable `mdelBlacklist` angepasst werden. Die Identifikation der *Methoden-Delegationen* über die Methodennamen erfolgt in der Methode `getMethodDelegations`. Beide Methoden sind Listing 3.13 zu entnehmen.

```
1  function evalProxy( proxy, tests ){
2   for( test : T ){
3    if( test.eval( proxy ) ){
4     passedTestcases = passedTestcases + 1
5    }
6    else {
7     triedMethodCalls = test.triedMethodCalls
8     mDel = getMethodDelegations( proxy, triedMethodCalls )
9     mdelBlacklist.add( mDel )
10   }
11  }
12 }
13
14 function getMethodDelegations( proxy, methodNames ){
```

```
15  for( i=0; i < proxy.dels.size; i++ ){
16   methodName = proxy.dels[i].call.name
17   if( methodNames.containsAll( methodName ) ){
18    return proxy.dels[i]
19   }
20  }
21  return null
22 }
```

Listing 3.13: Evaluierung einzelner Proxies mit BL_MNC

Das Ausschließen bestimmter *Proxies* erfolgt, indem Elemente aus einer Liste von *Proxies* entfernt werden. Listing 3.14 zeigt die dafür vorgesehene Methode `BL`, welche die Basis-Liste der *Proxies* im Parameter `proxies` und die Liste der Kombinationen von *Methoden-Delegationen*, die die Grundlage für den Ausschluss einzelner *Proxies* bilden, im Parameter `blacklist` erwartet.

```
1  function BL( proxies, blacklist ){
2    filtered = []
3    for( proxy : proxies ){
4      blacklisted = false
5      for( md : blacklist ){
6        if( proxy.dels.contains( md ) ){
7          blacklisted = true
8          break
9        }
10     }
11     if( !blacklisted ){
12       filtered.add( proxy )
13     }
14   }
15   return filtered
16 }
```

Listing 3.14: Blacklist-Methode für Heuristil BL_NMC

Bei dieser Heuristik ist deren Anwendung nach jedem Evaluationsversuch eines einzelnen *Proxies* sinnvoll. Listing 3.15 zeigt die Anpassungen in der Funktion `evalProxies` aus Listing 3.10 für die Heuristik *BL_NMC*. Dabei sei davon auszugehen, dass die oben beschriebenen Funktionen aus den Listings 3.14 und 3.13 zur Verfügung stehen.

```
1  function evalProxies( proxies, tests ){
2    tested = []
3    filtered = BL( proxies, mdelBlacklist )
```

```
4     for( proxy : proxies ){
5       passedTestcases = 0
6       evalProxy(proxy, tests)
7       if( passedTestcases == tests.size ){
8         // validen Proxy gefunden
9         return proxy
10      }
11      else{
12        tested.add( proxy )
13          // noch nicht evaluierte Proxies ermitteln
14        leftProxies = proxies.removeAll( tested )
15        return evalProxies( leftProxies, tests )
16      }
17    }
18    // kein validen Proxy gefunden
19    return null
20  }
```

Listing 3.15: Evaluation mehrere Proxies mit BL_MNC

Kapitel 4

Implementierung

Die Implementierung der Explorationskomponente besteht aus drei Teilen, die jeweils als separates Java-Projekt umgesetzt wurden. Im weiteren Verlauf werden diese Java-Projekte als Module bezeichnet.In Abbildung 4.1 ist die Architektur der Explorationskomponente mit diesen drei Modulen aufgeführt.

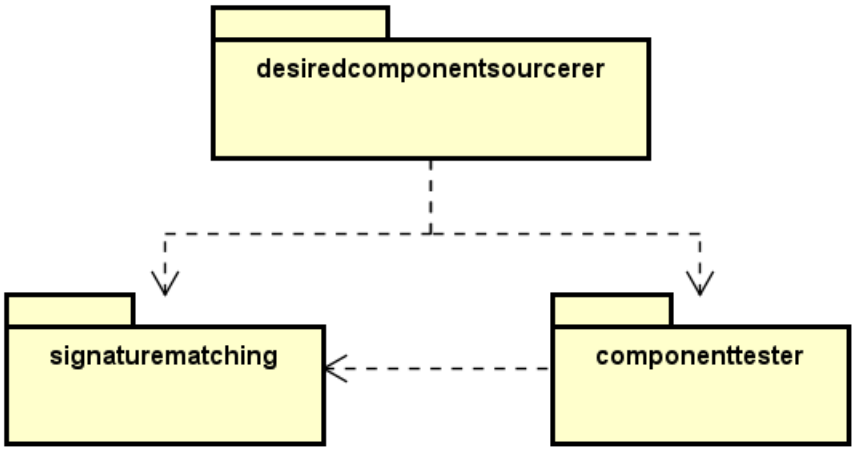


Abbildung 4.1: Architektur

Im weiteren Verlauf dieses Kapitels werden die Module einzeln beschrieben . Das Modul *DesiredComponentSourcerer* ist dabei von den Modulen *ComponentTester* und *SignatureMatching* abhängig, während das Modul *ComponentTester* lediglich vom Modul *SignatureMatching* abhängig ist.

Darüber hinaus, werden folgende externe Bibliotheken verwendet:

- cglib 3.3.0 [Ber19]
- objenesis 3.1 [obj21]
- junit 4.13.0 [jun21a]

Auf die konkrete Verwendung der externen Bibliotheken wird in den detaillierteren Beschreibungen der einzelnen Module eingegangen.

4.1 Modul: SignatureMatching

In diesem Modul befinden sich zum einen die Implementierungen der Matcher, die in Abschnitt 3.1.2 formal beschrieben wurden und zum anderen die Implementierung der Generatoren für die *Proxies*. In Abbildung 4.2 sind die wichtigsten Klassen und Interfaces dieses Moduls mit ihren Abhängigkeiten zueinander aufgeführt. Die Matcher befinden sich dabei im Package *matching* und die Generatoren für die *Proxies* in Form der Implementierungen des Interfaces `ProxyFactory` im Package *glue*.

Die in Abschnitt 3.1.2 beschriebenen Matcher und Generatoren wurden teilweise in einer Klasse zusammengefasst. Tabelle 4.1 zeigt die Zuordnung von Matchern zu den jeweiligen Klassen, die die Implementierung dieser darstellen (Spalte: Matcher-Implementierung). Zudem sind in der Tabelle 4.1 auch die Klassen ausgewiesen, die die Implementierung des Generators für den Proxy, der auf Basis des Matchers Anwendung findet, darstellen (Spalte: Generator-Implementierung).

Die Klasse `StructuralTypeMatcher` nimmt bei den Matcher-Klassen eine Sonderstellung ein. Dies ist daran zu erkennen, dass dieser nicht das Interface `TypeMatcher` implementiert. Das liegt daran, dass es sich bei diesem Matcher um den Einstiegspunkt der *strukturellen Evaluation* handelt. Analog zum *StructuralTypeMatcher* aus Abschnitt 3.1.2 wird in der Klasse `StructuralTypeMatcher` auf die anderen Matcher bzw. Matcher-Klassen zugegriffen, was in

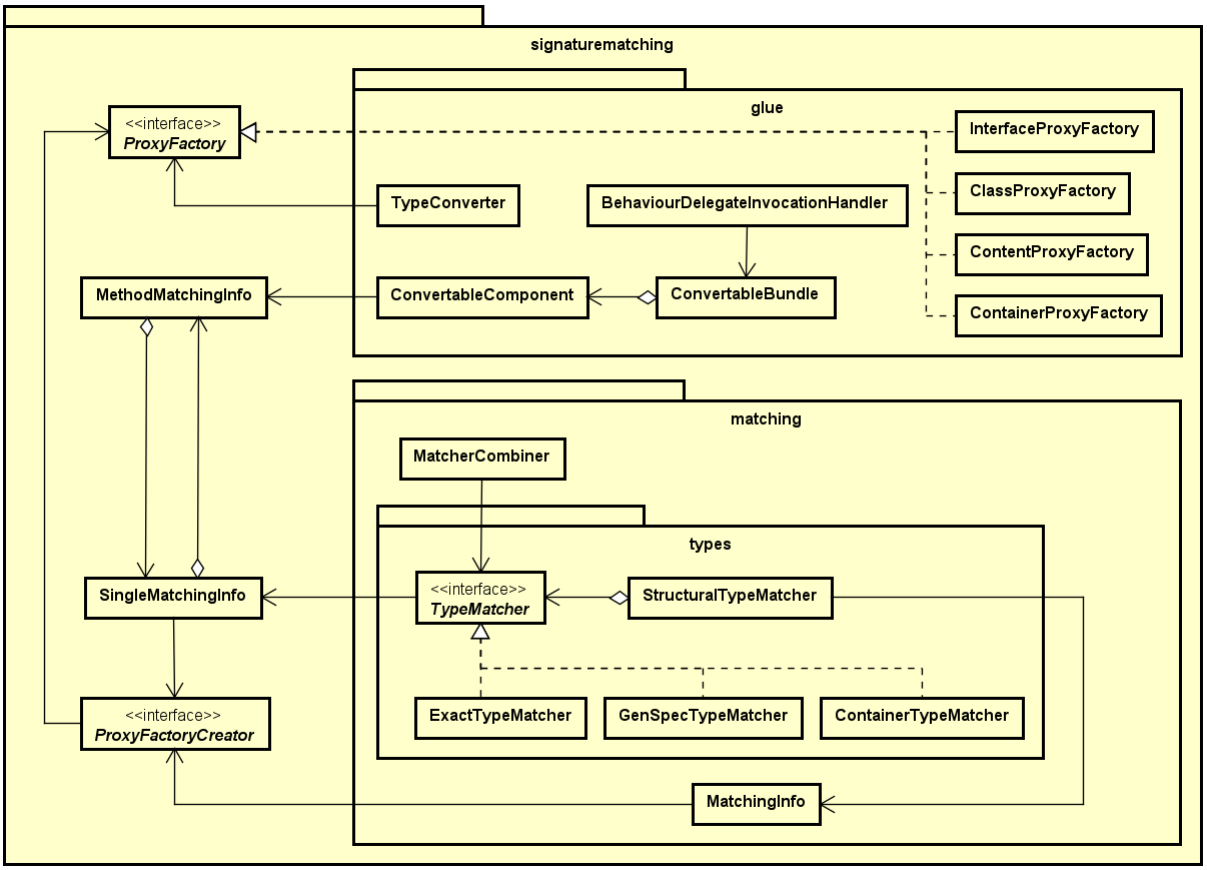


Abbildung 4.2: Modul: SignatureMatching

Matcher	Matcher-Implementierung	Generator-Implementierung
ExactTypeMatcher	`ExactTypeMatcher`	`ClassProxyFactory`
GenTypeMatcher	`GenSpecTypeMatcher`	`ClassProxyFactory`
SpecTypeMatcher	`GenSpecTypeMatcher`	`ClassProxyFactory`
ContentTypeMatcher	`ContainerTypeMatcher`	`ContentProxyFactory`
ContainerTypeMatcher	`ContainerTypeMatcher`	`ContainerProxyFactory`
StructuralTypeMatcher	`StructuralTypeMatcher`	`InterfaceProxyFactory`

Tabelle 4.1: Zuordnung der Matcher zu den Matcher- und Generator-Implementierungen

Abbildung 4.2 durch die Aggregation zwischen der Klasse `StructuralTypeMatcher` und dem Interface `TypeMatcher` angedeutet werden soll.

Die übrigen Matcher-Klassen implementieren das Interface `TypeMatcher` und können über die

Methode `combine` aus der Klasse `MatcherCombinator` miteinander kombiniert werden[1].

So kann eine Kombination mehrerer `TypeMatcher`, die wiederum von Typ `TypeMatcher` ist, in der Klasse `StructuralTypeMatcher` verwendet werden. Die konkrete `TypeMatcher`-Kombination, die im `StructuralTypeMatcher` instanziiert wird, orientiert sich an den Ausführungen in Abschnitt 3.1.2 (siehe auch Anhang A). Es ist aber zu erwähnen, dass die Verwendung weiterer Matcher, die in dieser Arbeit nicht definiert wurden, denkbar ist. Eine solche Erweiterung ließe sich leicht in dieses Modul über die Implementierung des Interfaces `TypeMatcher` und die Verwendung der Klasse `MatcherCombiner` vornehmen.

Alle Matcher-Implementierungen bieten die Möglichkeit, zu ermitteln, ob ein Matching zwischen zwei Typen besteht (siehe Klassendiagramme in Abbildungen 4.3 und 4.4). Dies erfolgt jeweils über die Methode `matchesType`. Über die Methode `calculateMatchingInfos` werden die Informationen bzgl. der Methodendelegationen zwischen den beiden gematchten Typen ermittelt. Diese Informationen werden in einem Objekt der Klasse `SingleMatchingInfo` bzw. `MatchingInfo` zusammengetragen, welche in Abbildung 4.3 und 4.4 detailliert dargestellt werden.

Diese beiden Klassen unterscheiden sich lediglich bzgl. des Attributs in dem die Delegationsmethoden hinterlegt sind. Dabei handelt es sich auf Seiten der `SingleMatchingInfo` um das Attribut `methodMatchingInfos` und auf Seiten der `MatchingInfo` um das Attribut `methodMatchingSupplier`.

Während ein Objekt der Klasse `MatchingInfo` mehrere *Delegationsmethoden* zu einer *aufgerufenen Methoden* enthalten kann, darf ein Objekt der Klasse `SingleMatchingInfo` lediglich eine *Delegationsmethode* zu einer *aufgerufenen Methode* enthalten (vgl. auch Abschnitt 3.1.2). Zusätzlich zu erwähnen ist, dass die Informationen über die *Delegationsmethoden* aus einer `MatchingInfo` über einen `MethodSupplier` überliefert werden.

Eine Instanz der Klasse `MethodSupplier` enthält zum einen ein `MatcherRating` welches Informationen bzgl. des in Abschnitt 3.4.1 beschriebenen *Matcherratings* beinhaltet. Zum anderen

[1] Ein Beispiel für die Kombination von Matchern ist im Anhang A zu finden.

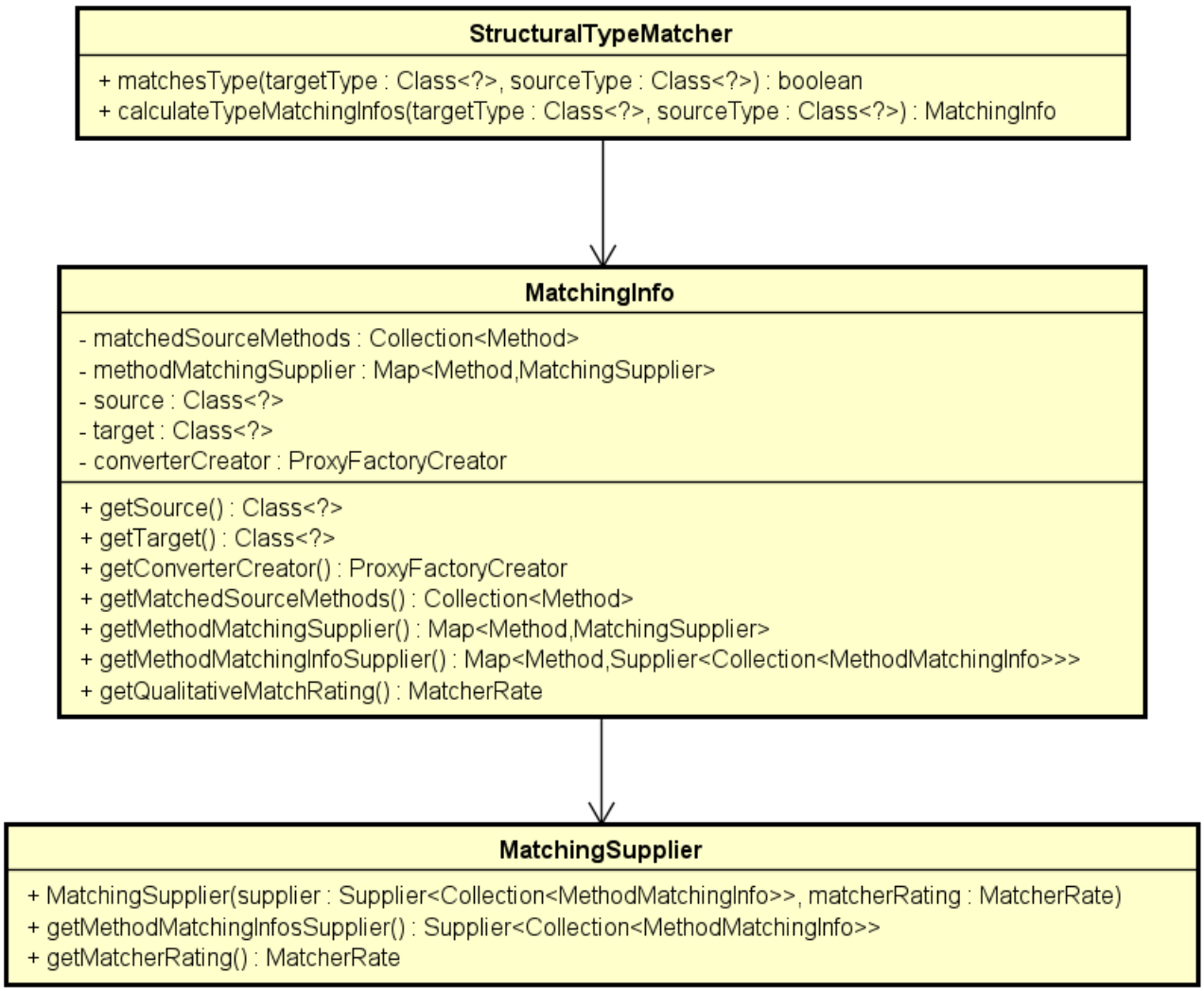


Abbildung 4.3: Klassendiagramm: `StructuralTypeMatcher` und `MatchingInfos`

werden im Attribut `methodMatchingInfo` in einem Objekt der Klasse `MethodMatchingInfo` (siehe Abbildung 4.5) die Informationen bzgl. der Delegation der *aufgerufenen Methode* an die *Delegationsmethode* hinterlegt.

Bezüglich der Klasse `SingleMatchingInfo` ist noch das Attribut `proxyFactoryCreator` zu beschreiben. Darin werden Informationen bzgl. der strukturellen Verbindung zwischen den gematchten Typen gehalten.

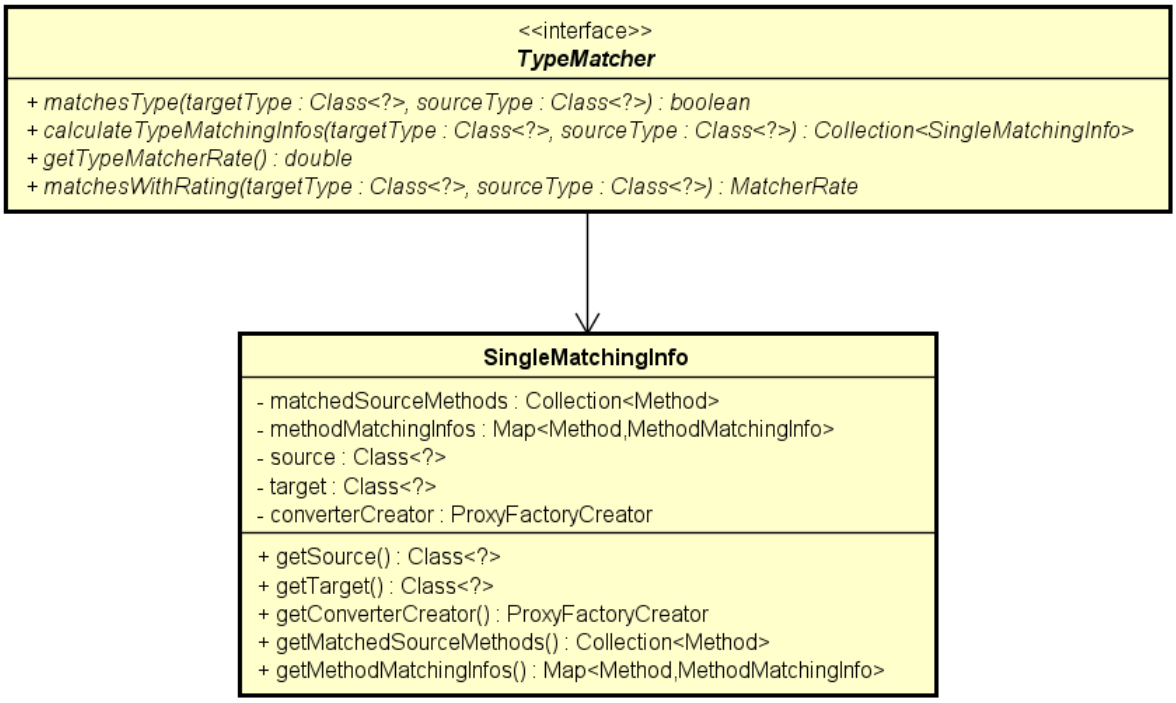


Abbildung 4.4: Klassendiagramm: `TypeMatcher` und `SingleMatchingInfo`

Für den *ExactTypeMatcher*, den *GenTypeMatcher* und den *SpecTypeMatcher* wird dabei ein `ProxyFactoryCreator` erzeugt, der in der Lage ist, eine `ProxyFactory` für Typen zu erzeugen, die in einer nominalen Beziehung [2] stehen.

Für den *ContentTypeMatcher* und den *ContainedTypeMatcher* hingegen, wird ein Objekt vom Typ `ProxyFactoryCreator` erzeugt, der in der Lage ist, eine `ProxyFactory` für Typen zu erzeugen, bei denen der eine Typ ein Attribut vom Typ des anderen enthält (vgl. mit Tabelle 4.1). Die erzeugten Objekte vom Typ `ProxyFactory` werden bei der Generierung der *Proxies* unter der Zuhilfenahme der Bibliotheken *cglib* und *objenesis* verwendet[3].

[2] Identität, Generalisierung, Spezialisierung

[3] Diese beiden Frameworks wurden verwendet, da die Erzeugung der *Proxies* mit ihnen komfortabler ist, als mit den Mitteln die das JKD zur Verfügung stellt. Dies gilt insbesondere für die Erzeugung von *Proxies* für Klassen, die mit dem Schlüsselwort `final` versehen sind. (vgl. [obj21], [Ber19])

Der `ProxyFactoryCreator` stellt damit eines der Bindeglieder zwischen der Package *matching* und dem Package *glue* innerhalb dieses Moduls her. Das zweite Artefakt, welches als Bindeglied fungiert, ist die oben bereits erwähnte Klasse `MethodMatchingInfo`, deren Aufbau dem Klassendiagramm aus Abbildung 4.5 zu entnehmen ist.

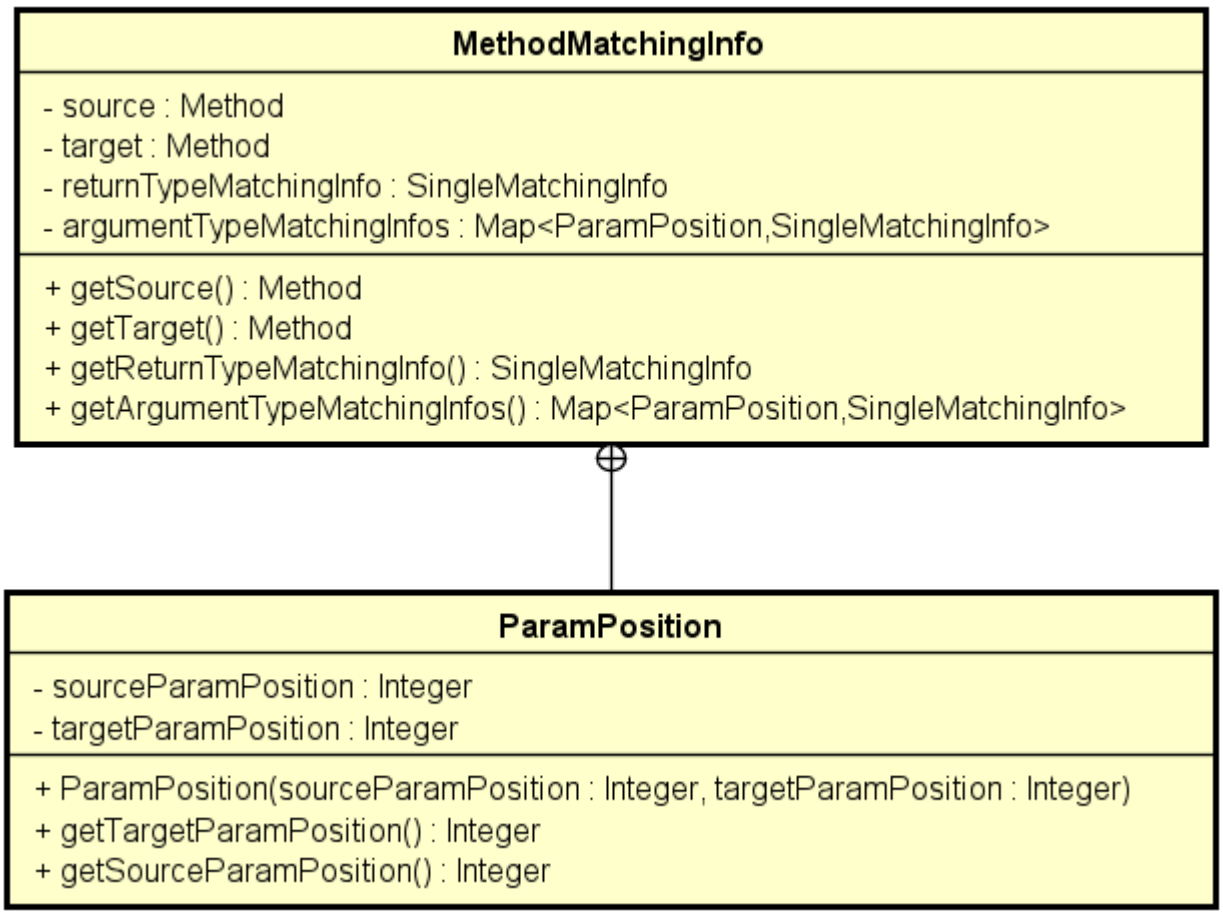


Abbildung 4.5: Klassendiagramm: `MethodMatchingInfo`

Ein Objekt der Klasse `MethodMatchingInfo` enthält in den Attributen `source` und `target` je eine Methode. Dabei ist im Attribut `source` die *aufgerufene Methode* der *Methoden-Delegation* und im Attribut `target` die *Delegationsmethode* hinterlegt. Darüber hinaus wird im Attribut `returnTypeMatchingInfo` ein Objekt der Klasse `SingleMatchingInfo` gehalten, welches alle notwendigen Informationen für das Erzeugen eines *Proxies* des Rückgabetyps der *aufgerufenen*

Methode aus dem Rückgabetyp der *Delegationsmethode* enthält.

Analog dazu wird im Attribut `argumentTypeMatchingInfos` eine Map, bestehend aus weiteren Objekten der Klasse `SingleMatchingInfo` und jeweils einem Objekt der Klasse `ParamPosition`, gehalten. Diese Map enthält alle notwendigen Information für das Erzeugen eines *Proxies* für die Parametertypen der *Delegationsmethoden* aus den Parametertypen der *aufgerufenen Methode*, sowie der Anpassung der Übergabeposition bei der Delegation der *aufgerufenen Methode* (siehe auch Abschnitt 3.2.1).

Um die *Methoden-Delegationen* zu koordinieren, wird bei der Erzeugung des *Proxies* in der jeweiligen `ProxyFactory` für das *Proxy*-Objekt ein `InvocationHandler` instanziiert (vgl. [inv20]). Dieses Interface wird im *glue*-Package durch die Klasse `BehaviourDelegateInvocationHandler` implementiert, in der letztendlich die Koordination der *Methoden-Delegationen* auf Basis der jeweiligen `MethodMatchingInfo` spezifiziert ist.

Um einen *Proxy* basierend auf dem Matching zweier Typen zu erzeugen, steht die Klasse `TypeConverter` zur Verfügung (siehe Abbildung 4.6). Die Zugriffe innerhalb des Packages *glue* als auch die Zugriffe von außerhalb verlangen jeweils ein Objekt der Klasse `ConvertableBundle`. Diese Klasse beschreibt eine Kombination mehrerer Objekte vom Typ `ConvertableComponent`, die als *Target-Typen* des zu erzeugenden Proxy-Objektes fungieren sollen. Ein Objekt der Klasse `ConvertableComponent` enthält eine Liste von Objekten vom Typ `SingleMatchingInfo`, die wie bereits erwähnt beschreiben, am welche Methode die Delegation erfolgen soll. Das Objekt im Attribut `convertableObject` der `ModuleMatchingInfo` beinhaltet das Objekt, auf dem die *Delegationsmethode* aufgerufen werden soll.

4.2 Modul: ComponentTester

Dieses Modul ist für die Ausführung der vordefinierten Tests zuständig. Darüber hinaus bietet es die Möglichkeit, die vordefinierten Tests mit den Interfaces, die den jeweiligen *required Typ* darstellen, zu verbinden. Dabei sei davon auszugehen, dass ein *required Typ* R in Form eines Interfaces existiert. Um die Tests für R zu definieren, können eine oder mehrere Testklassen

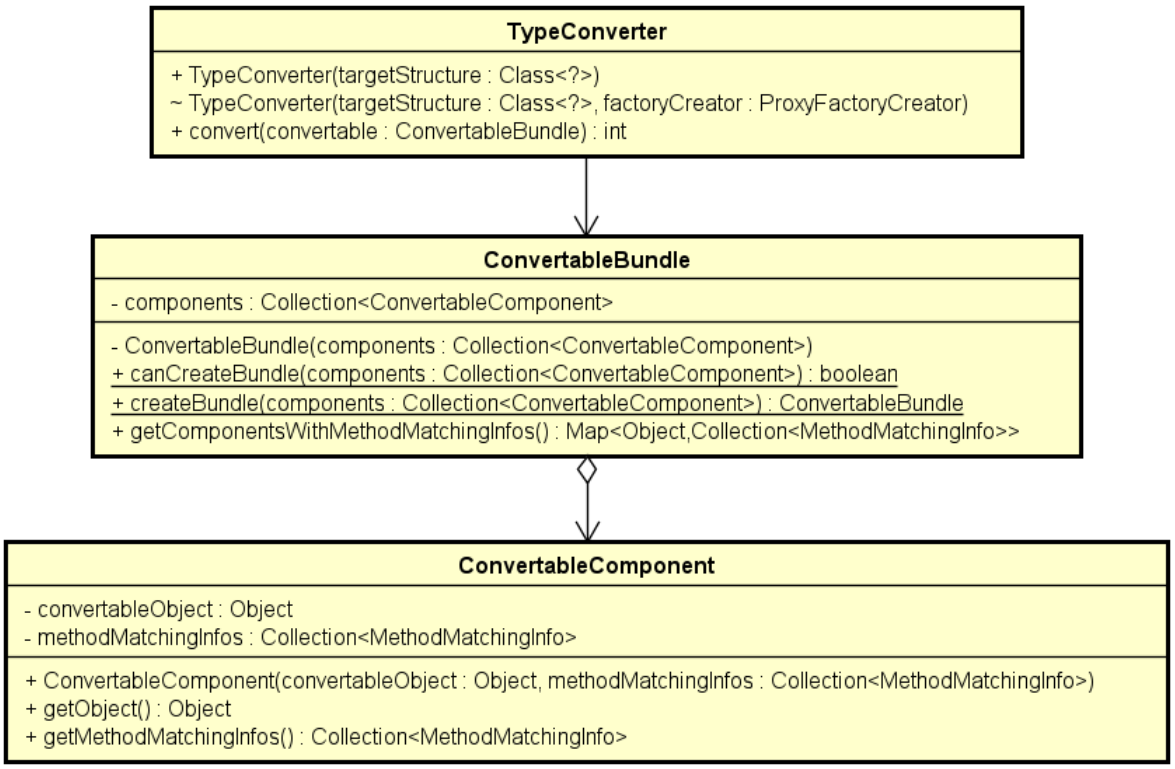


Abbildung 4.6: Klassendiagramm: `TypeConverter`

implementiert werden.

Die Testklassen werden dabei in dem Interface R über das Attribut `testClasses` der Annotation `RequiredTypeTestReference` angegeben (siehe Abbildung 4.7 Package: *API*). Ein Beispiel für die Deklaration eines solchen Interfaces und den dazugehörigen Testklassen ist im Anhang D zu finden.

Damit die Testmethoden in den Testklassen die in Abschnitt 3.3.1 beschriebenen Eigenschaften aufweisen und durch das *ComponentTester*-Modul ausfindig gemacht werden können, stehen mehrere Artefakte in dem *API*- und dem *SPI*-Package des *ComponentTester*-Moduls bereit (siehe Abbildung 4.7).

So muss jede Testklasse eine Methode bereitstellen, über die ein Objekt vom Typ R in die

Instanz der Testklasse injiziert werden kann.[4] Diese Methode wird von dem *ComponentTester*-Modul über die Annotation `RequiredTypeInstanceSetter` gefunden. Von daher muss die Methode mit eben dieser Annotation markiert werden. Die Testmethoden müssen von der

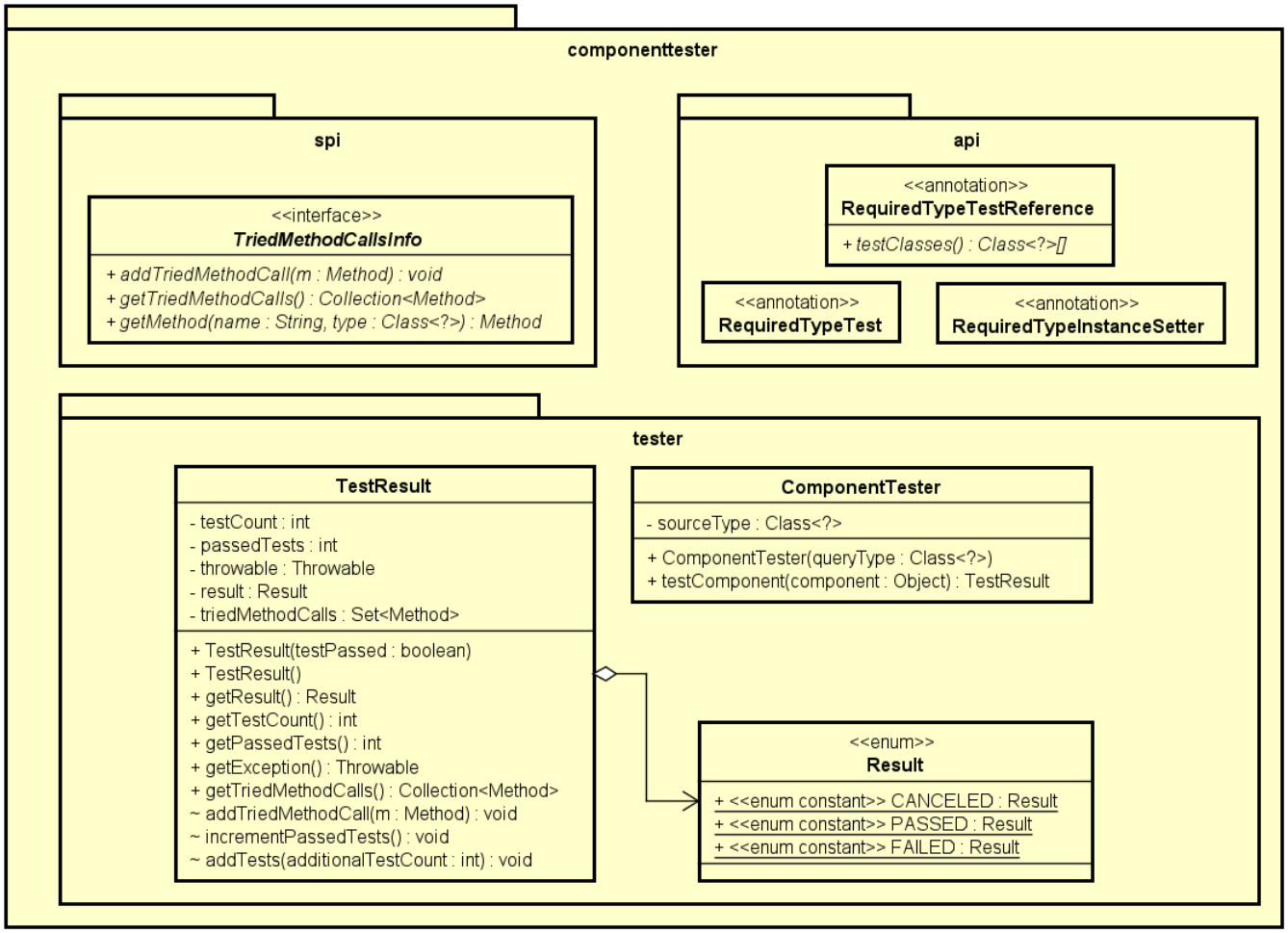


Abbildung 4.7: Modul: ComponentTester

Sichtbarkeit her öffentlich (`public`) sein. Weiterhin dürfen die Testmethoden keine Parameter erwarten und müssen mit der Annotation `RequiredTypeTest` markiert sein. Die Erwartungen innerhalb der Testmethoden müssen über die in JUnit 4 zur Verfügung stehenden Methoden aus der Klasse `Assert` (siehe auch [jun21b]) deklariert werden. Testdaten, die für alle Testmethoden

[4] auch genannt: Setter-Injection (vgl. [Fow04])

innerhalb einer Testklasse zur Verfügung stehen sollen, können über Methoden bereitgestellt werden, die mit den durch die JUnit 4 Bibliothek bereitgestellten Annotationen `Before` und `After` (vgl. [jun21b]) markiert wurden.

Um die Reihenfolge der versuchten Aufrufe der Methoden, die von R angeboten werden, zu verwalten[5], muss die Testklasse das Interface `TriedMethodCallsInfo` implementieren (siehe Abbildung 4.7 Package: *spi*). Dadurch wird die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` erzwungen. Die Methode `getMethod` kann mit der Defaultimplementierung übernommen werden, sofern die in R deklarierten Methoden über den Namen identifiziert werden können.

Die Implementierung der Methoden `addTriedMethodCall` und `getTriedMethodCalls` hat so zu erfolgen, dass bei einem Aufruf der Methode `addTriedMethodCall` der übergebene Parameter an eine Liste angefügt wird. Der Aufruf der Methode `getTriedMethodCalls` liefert eben diese Liste als Rückgabewert. Weiterhin ist sicherzustellen, dass vor dem Aufruf einer Methode m aus R die Methode `addTriedMethodCall` mit m als Parameter aufgerufen wird. Im Anhang D sind mehrere Beispiele für die korrekte Implementierung von Testklassen zu finden (siehe Listings D.8 - D.14).

Die Durchführung der Tests, die für R definiert wurde, wird über eine Instanz der Klasse `ComponentTester` gestartet (siehe Abbildung 4.7 Package: *Tester*). In Abhängigkeit der in R deklarierten Testklassen werden alle darin befindlichen Testmethoden mit einem *Proxy* für R durchgeführt, bis einer dieser Testfälle fehlschlägt. Der Aufrufer der Testdurchführung erhält dabei ein Objekt der Klasse `TestResult` zurück (siehe Abbildung 4.7). In diesem Objekt sind die für die Auswertung des Testergebnisses relevanten Informationen vorhanden, auf die die Heuristiken *PTTF* (siehe Abschnitt 3.4.2) und *BL_NMC* (siehe Abschnitt 3.4.3) angewiesen sind.

[5]Das ist für die Heuristik *BL_NMC* notwendig. (vgl. auch Abschnitt 3.4.3)

4.3 Modul: DesiredComponentSourcerer

In diesem Modul befindet sich die Implementierung für den Einsteigspunkt des *Explorationsprozesses*. Zum Starten des *Explorationsprozesses* für ein *required Typ R* in Form eines Interfaces muss zuerst eine Instanz der Klasse `DesiredComponentFinder` erzeugt werden (genannt: *Finder*). Dies erfolgt über einen Konstruktor, der ein Objekt der Klasse `DesiredComponentFinderConfig` (genannt: *Konfig*) erwartet (siehe Abbildung 4.8).

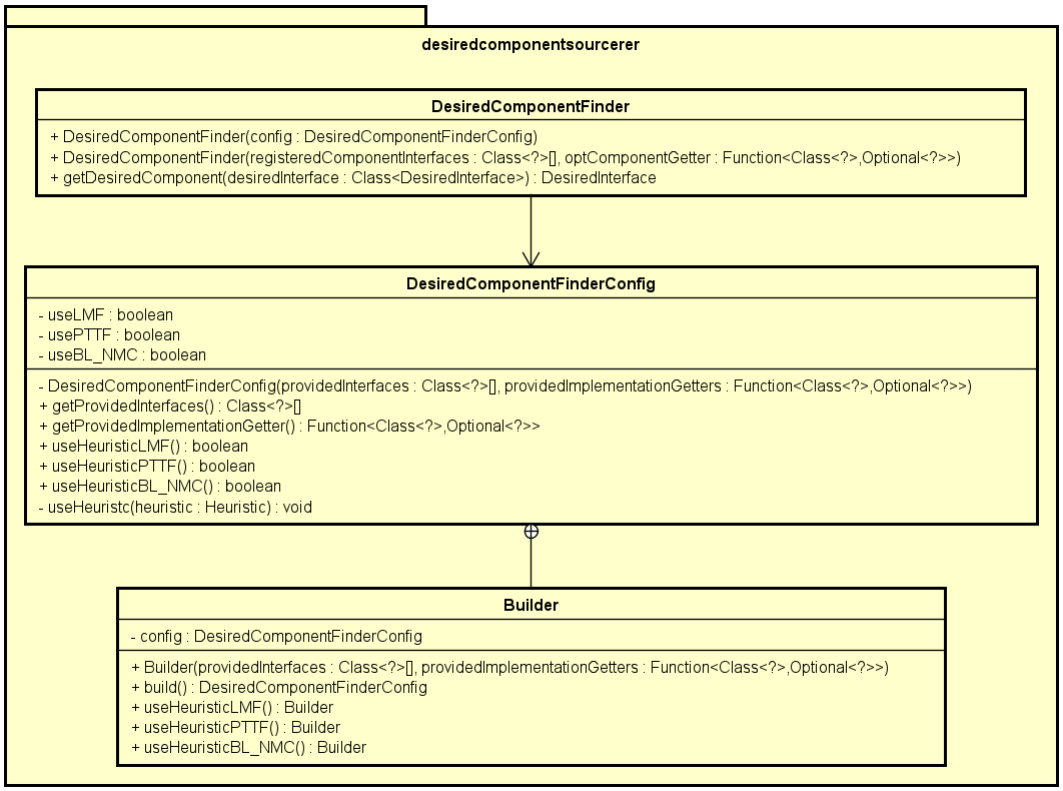


Abbildung 4.8: Modul: DesiredComponentSourcerer

Die Instanziierung einer solchen *Konfig* erfolgt über die Klasse `Builder`[6]. Dabei müssen zum einen alle *provided Typen* in Form einer Liste von Interfaces angegeben werden[7]. Zum anderen wird eine Funktion (`java.util.Function`) gefordert, über die die Implementierungen jener

[6] Builder-Pattern (siehe auch [ES13])
[7] In Bezug auf *EJBs* sind hier also alle *EJB*-Interfaces anzugeben

Interfaces ermittelt werden können.

Zum Zweck der gezielten Evaluation der Heuristiken im folgenden Kapitel kann über die *Konfig* gesteuert werden, welche der in Abschnitt 3.4 beschriebenen Heuristiken während des *Explorationsprozesses* verwendet werden sollen. Dies erfolgt über die in Abbildung 4.8 ersichtlichen Methoden mit den Präfix `useHeuristic`.

Nachdem der *Finder* erzeugt wurde, kann der *Explorationsprozess* über die Methode `getDesiredComponent` mit der Übergabe des Interfaces für den *required Typ* R als Parameter gestartet werden. Im Anschluss wird die *strukturelle Evaluation* für alle *provided Typen* durchgeführt. Hierzu wird ein Objekt vom `StructuralTypeMatcher` aus dem *SignatureMatching*-Modul verwendet[8] und versucht die *provided Typen* mit dem *required Typ* zu matchen.

Auf formaler Ebene gleicht dieser Schritt der Ausführung der Funktion $cover(R, L)$, wobei die in L befindlichen *provided Typen* auf die Typen, die dem *Finder* bei der Instanziierung übergebenen wurden, beschränkt sind.

Nach der *strukturellen Evaluation*, wird gemäß Abschnitt 3.3 die *semantische Evaluation* durchgeführt. Dabei werden zuerst die *Proxies* aus den Kombinationen der gematchten *provided Typen*[9] erzeugt, welche im Anschluss hinsichtlich der vordefinierten Tests bzgl. des *required Typs* R geprüft werden. Dabei werden die Heuristiken, die in der *Konfig* hinterlegt wurden, angewendet. Sofern während des *Explorationsprozesses* ein *Proxy* erfolgreich getestet wurde, wird dieser als Ergebnis des Aufrufs der Methode `getDesiredComponent` zurückgegeben. Falls kein *Proxy* die Tests besteht, wird `null` zurückgegeben.

[8] Dieses Objekt wird beim Instanziieren des *Finders* erzeugt (siehe auch Anhang A: Listing A.2).
[9] Diese Kombinationen sind auf formaler Ebene Äquivalent zu den Elementen der Mengen aus $cover(R, L)$.

Kapitel 5

Untersuchungsergebnisse

In dem System, welches für die Evaluation der Heuristiken verwendet wird, sind insgesamt 891 *provided Typen* (*EJBs*) und 7 *required Typen* enthalten. In Tabelle 5.1 sind die Namen der *required Typen* zusammen mit jeweils einem Kürzel und den Namen der strukturell und semantisch matchenden Kombinationen von *provided Typen* aufgeführt, die während des *Explorationsprozesses* ermittelt werden sollen. Die Kürzel dienen im weiteren Verlauf der Identifizierung der *required Typen*.

required Typ	**Kürzel**	**Kombination von provided Typen**
ElerFTFoerderprogrammeProvider	TEI1	ElerFTStammdatenAuskunftService
FoerderprogrammeProvider	TEI2	StammdatenAuskunftService
MinimalFoerderprogrammeProvider	TEI3	StammdatenAuskunftService
IntubatingFireFighter	TEI4	Doctor, FireFigher
IntubatingFreeing	TEI5	Doctor, FireFigher
IntubatingPatientFireFighter	TEI6	Doctor, FireFigher
KOFGPCProvider	TEI7	ElerFTStammdatenAuskunftService, StammdatenAuskunftService

Tabelle 5.1: Required Typen mit Kürzeln und matchenden Kombinationen von provided Typen

Die Deklarationen der *required Typen* und der *provided Typen* aus Tabelle 5.1 sind im Anhang C zu finden. Aufgrund der Geheimhaltungspflicht bzgl. der Implementierungsdetails kann auf die Deklaration der Java-Interfaces, die sich aus dieser Deklaration der *required* und *provided Typen* ableiten lassen, und deren Implementierungen in dieser Arbeit nicht genauer eingegangen werden.

Um die Ergebnisse nachstellen zu können, kann das Modul, welches im Abschnitt 4.3 beschrieben wurde, mit einer beliebigen Bibliothek, welche sich ebenfalls durch die in Abschnitt 3.1.1 beschriebene Struktur von Typen abbilden lässt, verwendet werden. Zudem befindet sich auf dem beiliegenden Datenträger (siehe Anhang H) das Java-Projekt *DesCoSTests*. Dort sind einige Minimalbeispiele bzgl. der Verwendung des in Abschnitt 4.3 beschriebenen Moduls. In Anhang G sind weitere Beschreibungen zu dem Java-Projekt *DesCoSTests* zu finden.

5.1 Darstellung der Untersuchungsergebnisse

Die Untersuchungsergebnisse werden in der Form von Vier-Felder-Tafeln dargestellt (Beispiel siehe Tabelle 5.2). Für jeden *required Typ* wird eine Vier-Felder-Tafel für jeden Durchlauf der Schleife innerhalb der Methode `semanticEval` der *semantischen Evaluation* (siehe Abschnitt 3.3) aufgezeigt. Aus der jeweiligen Tafel geht hervor, wie viele *Proxies* über die Funktion *targetSets* (vgl. Abschnitt 3.3) in dem aktuellen Iterationsschritt erzeugt werden können. Der Wert, den die Iterationsvariable `i` im betrachteten Durchlauf enthält, wird in der oberen rechten Ecke der Tafel abgebildet.

In der Spalte “positiv” ist die Anzahl der *Proxies* verzeichnet, die innerhalb des Durchlaufs erzeugt und geprüft wurden. Die Zahl in der Spalte “negativ” drückt hingegen aus, wie viele der möglichen *Proxies* aufgrund bestimmter Kriterien (bzw. Heuristiken) nicht erzeugt wurden.

Die Zeile “falsch” beschreibt die Anzahl der möglichen *Proxies*, welche die *semantische Evaluation* nicht bestehen. Dementsprechend stellt die Zeile “richtig” die Anzahl der *Proxies* dar, welche die *semantischen Evaluation* bestehen.

Da der *Explorationsprozess* abgebrochen wird, sofern ein *Proxy* die *semantische Evaluation* besteht, ist in der Zelle “positiv” - “richtig” ein Wert von 0 oder 1 zu erwarten. Dementsprechend ist in der Zelle “negativ” - “richtig” immer der Wert 0 enthalten, denn ein *Proxy*, der nicht erzeugt wurde, kann auch nicht positiv getestet werden.

Aus Abschnitt 3.2.4 geht hervor, dass die Anzahl der *Proxies*, die für einen *required Typ* R mit einer Menge von *provided Typen* T über die Funktion $proxyCount(R, T)$ näherungsweise bestimmt werden kann. Für eine vereinfachte Darstellung der Untersuchungsergebnisse bzgl. eines *required Typs* R aus einer Bibliothek L mit $C = cover(R, L)$ und einem Iterationsschritt i wird die Anzahl der *Proxies* für die Anzahl a von Mengen von *provided Typen*, auf deren Basis die *Proxies* erzeugt werden können, näherungsweise auch wie folgt beschrieben:

$$p_i(a) := \sum_{k=1}^{a} proxyCount(R, TM) \;\Big|\; \{TM_1, ..., TM_a\} = targetSets(C, i)$$

Diese Notation kommt jedoch nur bei der Darstellung der Untersuchungsergebnisse eines Iterationsschrittes zum Einsatz, in dem ein valider *Proxy* gefunden wird. Für alle anderen Durchläufe ist die Anzahl der möglichen *Proxies* bekannt und wird somit auch explizit dargestellt.

Tabelle 5.2 zeigt ein Beispiel für eine solche Vier-Felder-Tafel, in der die Ergebnisse des 1. Iterationsschrittes dargestellt sind. Dabei wurden 11 *Proxies* generiert und getestet. 10 dieser *Proxies* bestanden die *semantische Evaluation* nicht. Da in diesem Beispiel ein *Proxy* die *semantische Evaluation* bestand, und der *Explorationsprozess* anschließend beendet wurde, mussten die übrigen *Proxies*, die auf Basis der insgesamt 20 Kombinationen von *provided Typen* hätten erzeugt werden können, nicht generiert und damit auch nicht getestet werden.

1	**positiv**	**negativ**
falsch	10	$p_1(20) - 11$
richtig	1	0

Tabelle 5.2: Beispiel: Vier-Felder-Tafel

5.2 Ausgangspunkt

Für einen *reqiured Typ* können mehrere *provided Typen* gefunden werden, auf deren Basis ein *Proxy* erzeugt werden kann. Tabelle 5.3 zeigt die Anzahl der *provided Typen*, zu denen der jeweilige *required Typ* über den *StructuralTypeMatcher* gematcht werden kann[1]. Diese kommen einzeln oder in Kombination für die *semantische Evaluation* in Frage.

Required Typ	**Anzahl strukturell übereinstimmender provided Typ**
TEI1	221
TEI2	272
TEI3	268
TEI4	75
TEI5	75
TEI6	53
TEI7	346

Tabelle 5.3: Anzahl strukturell gematchten provided Typen für die Evaluation

Die Tabellen 5.4-5.14 zeigen die Vier-Felder-Tafeln, in denen die Ergebnisse der benötigten Iterationen innerhalb der *semantischen Evaluation* für jeden der *required Typen* aus Tabelle 5.3. Dabei wurden keine Heuristiken verwendet. Somit stellt dies den Ausgangspunkt für die weitere Evaluation der Heuristiken dar.

1	**positiv**	**negativ**
falsch	233	$p_1(44) - 234$
richtig	1	0

Tabelle 5.4: Ausgangspunkt für TEI1

1	**positiv**	**negativ**
falsch	9389	$p_1(55) - 9399$
richtig	1	0

Tabelle 5.5: Ausgangspunkt für TEI2

1	**positiv**	**negativ**
falsch	8364	$p_1(50) - 8365$
richtig	1	0

Tabelle 5.6: Ausgangspunkt für TEI3

[1]Strukturelle Übereinstimmung

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.7: Ausgangspunkt für TEI4 1. Durchlauf

2	positiv	negativ
falsch	56766	$p_2(2247) - 56767$
richtig	1	0

Tabelle 5.8: Ausgangspunkt für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.9: Ausgangspunkt für TEI5 1. Durchlauf

2	positiv	negativ
falsch	244479	$p_2(2775) - 244480$
richtig	1	0

Tabelle 5.10: Ausgangspunkt für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.11: Ausgangspunkt für TEI6 1. Durchlauf

2	positiv	negativ
falsch	43360	$p_2(1323) - 43361$
richtig	1	0

Tabelle 5.12: Ausgangspunkt für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle 5.13: Ausgangspunkt für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	7764501	$p_2(52150) - 7764502$
richtig	1	0

Tabelle 5.14: Ausgangspunkt für TEI7
2. Durchlauf

Für die *required Typen TEI4-TEI7* werden zwei Durchläufe benötigt, da die Tests nur von einem *Proxy* bestanden werden, der aus einer Kombination zweier *provided Typen* erzeugt wurde (siehe auch Tabelle 5.1).

5.3 Ergebnisse für die Heuristik LMF

In Bezug auf die Heuristik *LMF* gilt es nicht nur zu evaluieren, ob die Suche nach einem *Proxy*, der die vordefinierten Tests besteht, beschleunigt werden kann, sondern auch, mit welcher Variante zur Bestimmung des *Matcherratings* (vgl. Abschnitt 3.4.1) die besten Ergebnisse erzielt werden können.

Hierzu wird der *Explorationsprozess* für alle in Tabelle 5.1 genannten *required Typen* mit jede Variante zur Bestimmung der *Matcherratings* durchgeführt (siehe Abschnitt 3.4.1 Tabelle 3.5). Im Folgenden wird lediglich auf die Variante eingegangen, die die besten Ergebnisse hervorgebracht hat. Die Ergebnisse unter Verwendung der übrigen Varianten sind im Anhang E zu finden.

Die Variante *1.1* (vgl. Tabelle 3.5) erbrachte die besten Ergebnisse. Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse mit dieser Variante zur Bestimmung der *Matcherratings* für die *required Typen TEI1-TEI3* auf.

1	positiv	negativ
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle 5.15: Ergebnisse *LMF* mit Variante 1.1 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	1889	$p_1(55) - 1890$
richtig	1	0

Tabelle 5.16: Ergebnisse *LMF* mit Variante 1.1 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1463	$p_1(50) - 1464$
richtig	1	0

Tabelle 5.17: Ergebnisse *LMF* mit Variante 1.1 für TEI3 1. Durchlauf

Die Ergebnisse für die *required Typen TEI4 - TEI7* zeigen die folgenden Vier-Felder-Tafeln.

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.18: Ergebnisse *LMF* mit Variante 1.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(2247) - 3$
richtig	1	0

Tabelle 5.19: Ergebnisse *LMF* mit Variante 1.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.20: Ergebnisse *LMF* mit Variante 1.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	32	$p_2(2775) - 33$
richtig	1	0

Tabelle 5.21: Ergebnisse *LMF* mit Variante 1.1 für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	1051	0
richtig	0	0

Tabelle 5.22: Ergebnisse *LMF* mit Variante 1.1 für TEI6 1. Durchlauf

2	**positiv**	**negativ**
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle 5.23: Ergebnisse *LMF* mit Variante 1.1 für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	161294	0
richtig	0	0

Tabelle 5.24: Ergebnisse *LMF* mit Variante 1.1 für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	7641	$p_2(52150) - 7642$
richtig	1	0

Tabelle 5.25: Ergebnisse *LMF* mit Variante 1.1 für TEI7 2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *LMF* erzielt im Vergleich zum Ausgangspunkt (Abschnitt 5.2) für jeden *required Typ* eine weitere Reduktion der zu prüfenden *Proxies*.

2. Die Heuristik *LMF* hat keine Auswirkung auf einen Durchlauf, in dem kein *Proxy* erzeugt wird, mit dem die vordefinierten Tests erfolgreich durchgeführt werden können. Dies kann durch einen Vergleich des ersten Durchlaufs für die *required Typen TEI4-TEI7* im Ausgangspunkt (Tabellen 5.7, 5.9, 5.11 und 5.11) mit dem ersten Durchlauf unter Anwendung der Heuristik *LMF* (Tabellen 5.18, 5.20, 5.22 und 5.24) festgestellt werden. Aus diesem Grund kommt die in Punkt 1 beschriebene Reduktion erst im jeweils letzten Durchlauf zum Tragen.

5.4 Ergebnisse für die Heuristik PTTF

Für die Heuristik *PTTF* gilt es zu evaluieren, ob die Suche nach einem *Proxy*, der die vordefinierten Tests besteht, beschleunigt werden kann. Hierzu wird der *Explorationsprozess* für alle in Tabelle 5.1 genannten *required Typen* unter der Verwendung der in Abschnitt 3.4.2 beschriebenen Heuristik durchgeführt.

Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse für die *required Typen TEI1-TEI7* auf.

1	**positiv**	**negativ**
falsch	29	$p_1(44) - 30$
richtig	1	0

Tabelle 5.26: Ergebnisse *PTTF* für TEI1 1. Durchlauf

1	**positiv**	**negativ**
falsch	5544	$p_1(55) - 5545$
richtig	1	0

Tabelle 5.27: Ergebnisse *PTTF* für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	4761	$p_1(50) - 4762$
richtig	1	0

Tabelle 5.28: Ergebnisse *PTTF* für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle 5.29: Ergebnisse *PTTF* für TEI4 1. Durchlauf

2	positiv	negativ
falsch	466	$p_2(2247) - 467$
richtig	1	0

Tabelle 5.30: Ergebnisse *PTTF* für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle 5.31: Ergebnisse *PTTF* für TEI5 1. Durchlauf

2	positiv	negativ
falsch	2172	$p_2(2775) - 2173$
richtig	1	0

Tabelle 5.32: Ergebnisse *PTTF* für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle 5.33: Ergebnisse *PTTF* für TEI6 1. Durchlauf

2	positiv	negativ
falsch	13122	$p_2(1323) - 13123$
richtig	1	0

Tabelle 5.34: Ergebnisse *PTTF* für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle 5.35: Ergebnisse *PTTF* für TEI7 1. Durchlauf

2	positiv	negativ
falsch	149961	$p_2(52150) - 149962$
richtig	1	0

Tabelle 5.36: Ergebnisse *PTTF* für TEI7 2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *PTTF* erzielt im Vergleich zum Ausgangspunkt (Abschnitt 5.2) für jeden *required Typ* eine weitere Reduktion der zu prüfenden *Proxies*.

2. Die Heuristik *PTTF* hat keine Auswirkung auf einen Durchlauf, in dem kein *Proxy* erzeugt wird, mit dem die vordefinierten Tests erfolgreich durchgeführt werden können. Dies kann durch einen Vergleich des ersten Durchlaufs für den *required Typ TEI4-TEI7* im Ausgangspunkt (Tabelle 5.7, 5.9, 5.11 und 5.11) mit dem ersten Durchlauf unter Anwendung der Heuristik (Tabellen 5.29, 5.31, 5.33 und 5.35) festgestellt werden. Aus diesem Grund kommt die in Punkt 1 beschriebene Reduktion erst im jeweils letzten Durchlauf zum Tragen.

5.5 Ergebnisse für die Heuristik BL_NMC

Für die Heuristik *BL_NMC* gilt es zu evaluieren, ob die Suche nach einem *Proxy*, der die vordefinierten Tests besteht, beschleunigt werden kann. Hierzu wird der *Explorationsprozess* für alle in Tabelle 5.1genannten *required Typen* unter der Verwendung der in Abschnitt 3.4.3 beschriebenen Heuristik durchgeführt.

Die folgenden Vier-Felder-Tafeln zeigen die Ergebnisse für die *required Typen TEI1-TEI7* auf.

1	positiv	negativ
falsch	105	$p_1(44) - 106$
richtig	1	0

Tabelle 5.37: Ergebnisse *BL_NMC* für TEI1 1. Durchlauf

1	positiv	negativ
falsch	342	$p_1(55) - 343$
richtig	1	0

Tabelle 5.38: Ergebnisse *BL_NMC* für TEI2 1. Durchlauf

1	positiv	negativ
falsch	357	$p_1(50) - 358$
richtig	1	0

Tabelle 5.39: Ergebnisse *BL_NMC* für TEI3 1. Durchlauf

1	positiv	negativ
falsch	120	1054
richtig	0	0

Tabelle 5.40: Ergebnisse *BL_NMC* für TEI4 1. Durchlauf

2	positiv	negativ
falsch	442	$p_2(2247) - 443$
richtig	1	0

Tabelle 5.41: Ergebnisse *BL_NMC* für TEI4 2. Durchlauf

1	positiv	negativ
falsch	550	4434
richtig	0	0

Tabelle 5.42: Ergebnisse *BL_NMC* für TEI5 1. Durchlauf

2	positiv	negativ
falsch	1304	$p_2(2775) - 1305$
richtig	1	0

Tabelle 5.43: Ergebnisse *BL_NMC* für TEI5 2. Durchlauf

1	positiv	negativ
falsch	366	685
richtig	0	0

Tabelle 5.44: Ergebnisse *BL_NMC* für TEI6 1. Durchlauf

2	positiv	negativ
falsch	204	$p_2(1323) - 205$
richtig	1	0

Tabelle 5.45: Ergebnisse *BL_NMC* für TEI6 2. Durchlauf

1	positiv	negativ
falsch	1051	160243
richtig	0	0

Tabelle 5.46: Ergebnisse *BL_NMC* für TEI7 1. Durchlauf

2	positiv	negativ
falsch	135089	$p_2(52150) - 135090$
richtig	1	0

Tabelle 5.47: Ergebnisse *BL_NMC* für TEI7 2. Durchlauf

Folgendes kann aus diesen Ergebnissen abgeleitet werden:

1. Die Heuristik *BL_NMC* erzielt im Vergleich zum Ausgangspunkt (Abschnitt 5.2) für jeden *required Typ* eine weitere Reduktion der zu prüfenden *Proxies*.

2. Die Heuristik *BL_NMC* hat das Potential jeden Durchlauf innerhalb der *semantischen Evaluation* zu beschleunigen. Für den jeweils ersten Durchlauf kann dies durch einen Vergleich der Tabellen 5.4, 5.5, 5.6, 5.7, 5.9, 5.11 und 5.13 zum Ausgangspunkt mit den Tabellen 5.37, 5.38, 5.39, 5.40, 5.42, 5.44 und 5.46 festgestellt werden. Ein Vergleich der Tabelle 5.8, 5.10, 5.12 und 5.14 im Ausgangspunkt mit den Tabellen 5.41, 5.43, 5.45 und 5.47 belegt dies für den zweiten Durchlauf auf.

Aus den Ergebnissen, die in den Abschnitten 5.3 - 5.5 beschrieben wurden, lässt sich je *required Typ* eine Rangfolge der vorgestellten Heuristiken erstellen. Diese Rangfolge kann Tabelle 5.48 entnommen werden. Dabei gilt, dass die Heuristik, mit der am wenigsten *Proxies* generiert und geprüft werden mussten, den ersten Platz einnimmt.

Heuristik/Required Typ	**TEI1**	**TEI2**	**TEI3**	**TEI4**	**TEI5**	**TEI6**	**TEI7**
LMF	1.	2.	2.	2.	2.	2.	2.
PTTF	3.	3.	3.	3.	3.	3.	3.
BL_NMC	2.	1.	1.	1.	1.	1.	1.

Tabelle 5.48: Rangfolge der Heuristiken (Einzelbetrachtung)

5.6 Ergebnisse für die Kombination der Heuristiken

Im vorherigen Abschnitt wurde gezeigt, dass der *Explorationsprozess* durch jede der beschriebenen Heuristiken beschleunigt werden kann. Dabei wurde der *Explorationsprozess* mit jeweils einer der Heuristiken durchgeführt. In den folgenden Abschnitten soll evaluiert werden, ob die Verwendung einer Kombination der einzelnen Heuristiken während des *Explorationsprozesses* einen zusätzlichen Vorteil bringt.

Hierzu werden die Ergebnisse aller Kombinationen der einzelnen Heuristiken aufgeführt und im Anschluss bewertet.

5.6.1 Kombination: LMF + PTTF

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken *LMF* und *PTTF*.

1	**positiv**	**negativ**
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle 5.49: Ergebnisse *LMF + PTTF* für TEI1

1	**positiv**	**negativ**
falsch	1877	$p_1(55) - 1878$
richtig	1	0

Tabelle 5.50: Ergebnisse *LMF + PTTF* für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	1473	$p_1(50) - 1474$
richtig	1	0

Tabelle 5.51: Ergebnisse *LMF + PTTF* für TEI3 1. Durchlauf

1	**positiv**	**negativ**
falsch	1174	0
richtig	0	0

Tabelle 5.52: Ergebnisse *LMF + PTTF* für TEI4 1. Durchlauf

2	**positiv**	**negativ**
falsch	4	$p_2(2247) - 5$
richtig	1	0

Tabelle 5.53: Ergebnisse *LMF + PTTF* für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	4984	0
richtig	0	0

Tabelle 5.54: Ergebnisse *LMF + PTTF* für TEI5 1. Durchlauf

2	**positiv**	**negativ**
falsch	34	$p_2(2346) - 35$
richtig	1	0

Tabelle 5.55: Ergebnisse *LMF + PTTF* für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	1051	0
richtig	0	0

Tabelle 5.56: Ergebnisse *LMF + PTTF* für TEI6 1. Durchlauf

2	**positiv**	**negativ**
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle 5.57: Ergebnisse *LMF* + *PTTF* für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	161294	0
richtig	0	0

Tabelle 5.58: Ergebnisse *LMF* + *PTTF* für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	1076	$p_2(52150) - 1077$
richtig	1	0

Tabelle 5.59: Ergebnisse *LMF* + *PTTF* für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf wirkt sich die Kombination der Heuristiken *LMF* und *PTTF* nicht nennenswert aus. Da in der Einzelbetrachtung mit der Heuristik *LMF* bessere Ergebnisse erzielt wurden als mit der Heuristik *PTTF*, kann dies durch einen Vergleich der Tabellen 5.15, 5.16, 5.17, 5.18, 5.20, 5.22 und 5.24 mit den Tabellen 5.49, 5.50, 5.51, 5.52, 5.54, 5.56 und 5.58 nachvollzogen werden.

2. Für den zweiten Durchlauf ist eine Verbesserung zu festzustellen. Diese bezieht sich jedoch nur auf den *Explorationsprozess* für *TEI7* (vergleiche Tabelle 5.25 aus Abschnitt 5.3 mit Tabelle 5.59).

5.6.2 Kombination: LMF + BL_NMC

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken *LMF* und *BL_NMC*.

1	**positiv**	**negativ**
falsch	0	$p_1(44) - 1$
richtig	1	0

Tabelle 5.60: Ergebnisse *LMF* + *BL_NMC* für TEI1

1	**positiv**	**negativ**
falsch	83	$p_1(55) - 84$
richtig	1	0

Tabelle 5.61: Ergebnisse *LMF* + *BL_NMC* für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	89	$p_1(50) - 90$
richtig	1	0

Tabelle 5.62: Ergebnisse *LMF* + *BL_NMC* für TEI3 1. Durchlauf

1	**positiv**	**negativ**
falsch	120	1054
richtig	0	0

Tabelle 5.63: Ergebnisse *LMF* + *BL_NMC* für TEI4 1. Durchlauf

2	**positiv**	**negativ**
falsch	4	$p_2(2247) - 5$
richtig	1	0

Tabelle 5.64: Ergebnisse *LMF* + *BL_NMC* für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	550	4434
richtig	0	0

Tabelle 5.65: Ergebnisse *LMF* + *BL_NMC* für TEI5 1. Durchlauf

2	**positiv**	**negativ**
falsch	34	$p_2(2346) - 35$
richtig	1	0

Tabelle 5.66: Ergebnisse *LMF* + *BL_NMC* für TEI5 2. Durchlauf

1	positiv	negativ
falsch	115	936
richtig	0	0

Tabelle 5.67: Ergebnisse *LMF + PTTF* für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle 5.68: Ergebnisse *LMF + PTTF* für TEI6 2. Durchlauf

1	positiv	negativ
falsch	2448	158846
richtig	0	0

Tabelle 5.69: Ergebnisse *LMF + BL_NMC* für TEI7 1. Durchlauf

2	positiv	negativ
falsch	954	$p_2(52150) - 955$
richtig	1	0

Tabelle 5.70: Ergebnisse *LMF + BL_NMC* für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf wirkt sich die Kombination der Heuristiken *LMF* und *BL_NMC* positiv aus. Hierzu sind die Tabelle 5.60 mit der Tabelle 5.15 aus Abschnitt 5.3 sowie die Tabellen 5.61, 5.62 und 5.67 mit den Tabellen 5.38, 5.39, 5.39 und 5.44 aus Abschnitt 5.5 zu vergleichen.

2. Für den zweiten Durchlauf ist ebenfalls eine Verbesserung zu erkennen. Diese bezieht sich jedoch nur auf den *Explorationsprozess* für *TEI7* (vergleiche Tabelle 5.25 aus Abschnitt 5.3 mit Tabelle 5.70).

5.6.3 Kombination: PTTF + BL_NMC

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken *PTTF* und *BL_NMC*.

1	**positiv**	**negativ**
falsch	104	$p_1(44) - 105$
richtig	1	0

Tabelle 5.71: Ergebnisse *PTTF + BL_NMC* für TEI1

1	**positiv**	**negativ**
falsch	337	$p_1(55) - 338$
richtig	1	0

Tabelle 5.72: Ergebnisse *PTTF + BL_NMC* für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	357	$p_1(50) - 358$
richtig	1	0

Tabelle 5.73: Ergebnisse *PTTF + BL_NMC* für TEI3 1. Durchlauf

1	**positiv**	**negativ**
falsch	120	1054
richtig	0	0

Tabelle 5.74: Ergebnisse *PTTF + BL_NMC* für TEI4 1. Durchlauf

2	**positiv**	**negativ**
falsch	47	$p_2(2247) - 48$
richtig	1	0

Tabelle 5.75: Ergebnisse *PTTF + BL_NMC* für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	550	4434
richtig	0	0

Tabelle 5.76: Ergebnisse *PTTF + BL_NMC* für TEI5 1. Durchlauf

2	**positiv**	**negativ**
falsch	219	$p_2(2346) - 220$
richtig	1	0

Tabelle 5.77: Ergebnisse *PTTF + BL_NMC* für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	366	685
richtig	0	0

Tabelle 5.78: Ergebnisse *PTTF* + *PTTF* für TEI6 1. Durchlauf

2	**positiv**	**negativ**
falsch	204	$p_2(1323) - 205$
richtig	1	0

Tabelle 5.79: Ergebnisse *PTTF* + *PTTF* für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	1036	160258
richtig	0	0

Tabelle 5.80: Ergebnisse *PTTF* + *BL_NMC* für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	6015	$p_2(52150) - 6016$
richtig	1	0

Tabelle 5.81: Ergebnisse *PTTF* + *BL_NMC* für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf hat die Kombination der Heuristiken *PTTF* und *BL_NMC* keine Auswirkung. Die Ergebnisse sind nahezu identisch mit denen der Durchgeführten *Explorationsprozesse* mit der Heuristik *BL_NMC* aus Abschnitt 5.5. Dies kann durch einen Vergleich der Tabellen 5.71, 5.72, 5.73, 5.74, 5.76, 5.67 und 5.80 mit den Tabellen 5.37, 5.38, 5.39, 5.40, 5.42, 5.44 und 5.46 nachvollzogen werden.

2. Für den zweiten Durchlauf ist eine Verbesserung zu erkennen. Da in der Einzelbetrachtung mit der Heuristik *BL_NMC* bessere Ergebnisse erzielt wurden als mit der Heuristik *PTTF* (vergleiche Ergebnisse aus Abschnitt 5.5 mit den Ergebnissen aus Abschnitt 5.4), kann dies durch einen Vergleich der Tabellen 5.75, 5.77, 5.68 und 5.81 mit den Tabellen 5.41, 5.43, 5.45 und 5.47 nachvollzogen werden.

5.6.4 Kombination: LMF + PTTF + BL_NMC

Die folgenden Vier-Felder Tafeln zeigen die Ergebnisse mit der Kombination der Heuristiken *LMF*, *PTTF* und *BL_NMC*.

1	**positiv**	**negativ**
falsch	2	$p_1(44) - 3$
richtig	1	0

Tabelle 5.82: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI1

1	**positiv**	**negativ**
falsch	79	$p_1(55) - 80$
richtig	1	0

Tabelle 5.83: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	86	$p_1(50) - 87$
richtig	1	0

Tabelle 5.84: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI3 1. Durchlauf

1	**positiv**	**negativ**
falsch	120	1054
richtig	0	0

Tabelle 5.85: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI4 1. Durchlauf

2	**positiv**	**negativ**
falsch	4	$p_2(2247) - 5$
richtig	1	0

Tabelle 5.86: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	550	4434
richtig	0	0

Tabelle 5.87: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI5 1. Durchlauf

2	**positiv**	**negativ**
falsch	34	$p_2(2346) - 35$
richtig	1	0

Tabelle 5.88: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	115	936
richtig	0	0

Tabelle 5.89: Ergebnisse *LMF* + *PTTF* + *PTTF* für TEI6 1. Durchlauf

2	**positiv**	**negativ**
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle 5.90: Ergebnisse *LMF* + *PTTF* + *PTTF* für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	2448	158846
richtig	0	0

Tabelle 5.91: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	12	$p_2(52150) - 13$
richtig	1	0

Tabelle 5.92: Ergebnisse *LMF* + *PTTF* + *BL_NMC* für TEI7 2. Durchlauf

Aus diesen Ergebnissen lässt sich Folgendes ableiten:

1. Auf den ersten Durchlauf wirkt sich die Kombination der Heuristiken *LMF*, *PTTF* und *BL_NMC* nicht stärker aus als die Kombination der Heuristiken *LMF* und *BL_NMC* (siehe Abschnitt 5.6.2). Die Ergebnisse sind nahezu identisch.

2. Für den zweiten Durchlauf gilt zumindest für die *required Typen TEI4 - TEI6* dasselbe, wie für den ersten Durchlauf. Für den *required Typ TEI7* ist hingegen nochmals eine Verbesserung im Vergleich zu den 2er-Kombinationen (siehe Abschnitte 5.6.1-5.6.3) zu erkennen.

Wie bei der Einzelbetrachtung der Heuristiken lässt sich auch eine Rangfolge der Kombinationen von Heuristiken je *required Typ* erstellen. Diese Rangfolge kann Tabelle 5.93 entnommen werden. Dabei gilt wiederum, dass die Kombination von required Typ, mit der am wenigsten *Proxies* generiert und geprüft werden mussten, den ersten Platz einnimmt. Sofern mehrere Kombinationen von Proxies bzgl. dessen gleich aufliegen, wird dies durch eine Doppelplatzierung dargestellt.

Heuristik/Required Typ	**TEI1**	**TEI2**	**TEI3**	**TEI4**	**TEI5**	**TEI6**	**TEI7**
LMF + PTTF	3.	4.	4.	4.	4.	4.	4.
LMF + BL_NMC	1.	2.	2.	1./2.	1./2.	1./2.	2.
PTTF + BL_NMC	4.	3.	3.	3.	3.	3.	3.
LMF + PTTF + BL_NMC	2.	1.	1.	1./2.	1./2.	1./2.	1.

Tabelle 5.93: Rangfolge der Heuristiken (Kombinationen)

Zusammenfassend können die Untersuchungsergebnisse zusammen mit dem Ausgangspunkt nochmals Abbildung 5.1 entnommen werden. Diese zeigt die Anzahl der *Proxies*, die während des *Explorationsprozesses* für den jeweiligen *required Typ* unter der Verwendung der entsprechenden Heuristiken generiert und getestet wurden. Zu beachten ist hierbei, dass die Y-Achse (Anzahl der generierten und getesteten *Proxies*) logarithmisch skaliert ist.

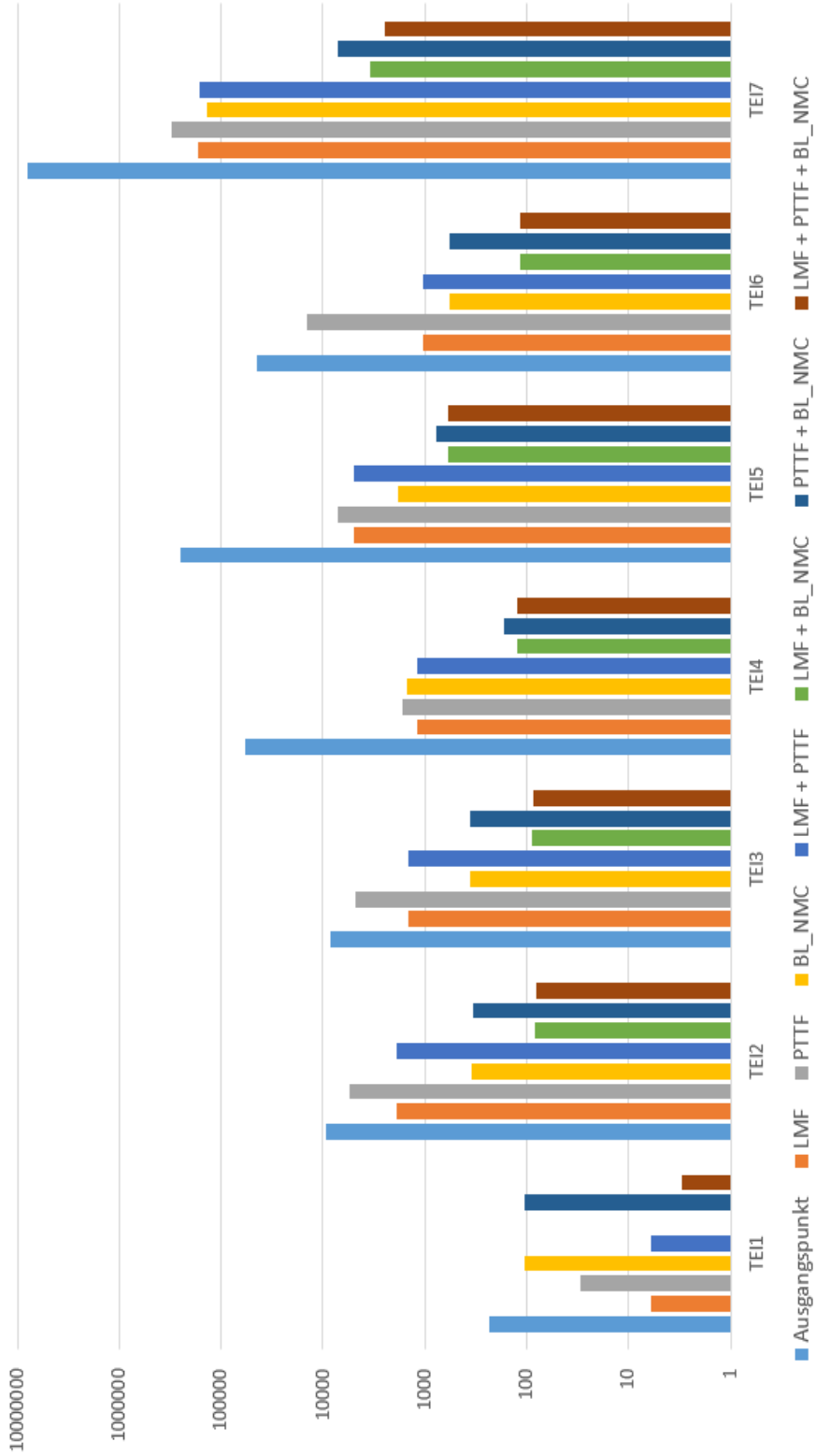


Abbildung 5.1: Gegenüberstellung der Untersuchungsergebniss

Kapitel 6

Diskussion

In den folgenden Abschnitten werden die Untersuchungsergebnisse aus Kapitel 5 ausgewertet und die Vor- und Nachteile des Ansatzes zur Exploration von *EJBs* zur Laufzeit gegenüber gestellt. Aufbauend auf diesen Vor- und Nachteilen werden außerdem Erweiterungsvorschläge des Ansatzes vorgestellt.

6.1 Auswertung der Untersuchungsergebnisse

6.1.1 Einzelbetrachtung

Die in Kapitel 5 beschriebenen Untersuchungsergebnisse zeigen, dass die Heuristiken die Anzahl der zu generierenden und zu prüfenden *Proxies* reduzieren. Dabei zeigt sich, dass sich die Heuristiken nicht auf alle Durchläufe der *semantischen Evaluation* positiv auswirken. So kann für die Heuristiken *LMF* und *PTTF* festgehalten werden, dass diese nur in dem Durchlauf eine positive Wirkung erzielt, in dem auch ein passender *Proxy* gefunden wird.

Die Heuristik *BL_NMC* hingegen wirkt sich auf jeden der durchgeführten Durchläufe aus. Ein Grund dafür ist, dass die Menge der Informationen, auf deren Basis diese Heuristik arbeitet, während eines Durchlaufs anwächst. Bei der Heuristik *LMF* ist dies nicht der Fall. Hier stehen die notwendigen Informationen bereits nach der *strukturelle Evaluation* zur Verfügung und ändern sich nicht mehr. Anders ist es wiederum bei der Heuristik *PTTF*, die ebenfalls mit Informationen arbeitet, die während dem Fortschreiten der *semantischen Evaluation* an-

wachsen. Daher muss die oben genannte Wirkung der Heuristik *BL_NMC* auf etwas anderes zurückzuführen sein.

Ein weiterer Grund dafür ist, dass die Heuristik *BL_NMC* dafür sorgt, dass *Proxies* bei der *semantischen Evaluation* mitunter übersprungen werden, oder diese gar nicht erst generiert werden. Die anderen Heuristiken hingegen sorgen lediglich für eine Umsortierung der zu generierenden und zu prüfenden *Proxies*. Somit müssen unter der Verwendung der Heuristiken *LMF* und *PTTF* im Zweifelsfall alle *Proxies* generiert und geprüft werden, auch wenn kein passender *Proxy* während des aktuellen Durchlaufs ausgemacht werden kann.

Weiterhin ist festzuhalten, dass mit der Heuristik *BL_NMC* scheinbar die besten Ergebnisse erzielt werden. Eine Ausnahme bildet hier lediglich die Exploration zum *required Typ* `ElerFTFoerderprogram` (*TEI1*). Für diesen *required Typ* wurden die besten Ergebnisse mit der Heuristik *LMF* erzielt. Die Ursache dafür liegt darin begründet, dass die in den Methoden von *TEI1* verwendeten Typ mit denen, die innerhalb des erwarteten *provided Typen*, auf dessen Basis ein passender *Proxy* erzeugt wird, exakt übereinstimmen. Damit wird ein vergleichsweise geringes *Matcherrating* für das Matching dieser beiden Typen ermittelt, wodurch der *Proxy* sehr früh während der *semantischen Evaluation* generiert und geprüft wird.

6.1.2 Synergien

Neben der Einzelbetrachtung der Heuristiken wurden in Abschnitt 5.6 auch die Kombinationen der drei Heuristiken untersucht. Aus den Feststellungen in Abschnitt 6.1.1 lässt sich ableiten, dass eine Kombination mit der Heuristik *BL_NMC* durchaus sinnvoll ist, egal ob sie mit der Heuristik *LMF* oder *PTTF* kombiniert wird. Der Grund dafür liegt wiederum in der Tatsache, dass die Heuristiken *LMF* und *PTTF* lediglich auf einen der Durchläufe einen positiven Effekt haben. Aus diesem Grund kann in Kombination mit der Heuristik *BL_NMC* wenigstens in den anderen Durchläufen eine positive Auswirkung festgestellt werden.

Dem entgegen liefert die Kombination der Heuristiken *LMF* und *PTTF* miteinander kaum bessere Ergebnisse als die Heuristik *LMF* alleine. Eine Ausnahme bildet der *required Typ* `KOFGPCProvider` (*TEI7*). Dazu ist jedoch zu sagen, dass gerade zu diesem *required Typ* im

Vergleich zu den anderen *required Typen* die meisten matchenden *provided Typen* existieren. Insofern darf dieser scheinbare Ausreißer nicht unterschätzt werden, weshalb auch die Kombination der oben genannten Heuristiken *LMF* und *PTTF* als sinnvoll anzusehen ist.

Ähnliches gilt für die Kombination aller vorgestellten Heuristiken (*LMF + PTTF + BL_NMC*). Dies ergibt sich ebenfalls aus den vorherigen Auswertungen bzgl. der Synergien in diesem Abschnitt. Bei der Betrachtung der Untersuchungsergebnisse zeigt sich hier ein ähnliches Muster wie zuvor: Die Kombination aller vorgestellten Heuristiken liefert nur für den *required Typ* `KOFGPCProvider` (*TEI7*) bessere Ergebnisse, als die Kombination der Heuristiken *LMF* und *BL_NMC*. Aber auch hier darf dieses Ergebnis aufgrund der Eigenschaften von *TEI7* nicht vernachlässigt werden.

6.1.3 Erhöhte Komplexität

Die vorliegende Untersuchung zeigt, dass die Anzahl der zu generierenden und zu prüfenden *Proxies* in dem verwendeten System mit den vorgeschlagenen Heuristiken reduziert werden kann. Allerdings wurden negative Auswirkungen wie bspw. Speichernutzung (Speicherkomplexität) oder die benötigte Zeit (Zeitkomplexität) für den *Explorationsprozess* nicht untersucht.

Die Anwendung der Heuristiken hängt, wie in Abschnitt 3.4 beschrieben, von Informationen ab, die teilweise aus den für die *Proxies* verwendeten *provided Typen* ermittelt werden müssen (Matcherrating) bzw. nach der Ausführung der Tests über die gesamte restliche Laufzeit des *Explorationsprozesses* verwaltet werden müssen. Von daher ist davon auszugehen, dass sich die Anwendung der Heuristiken durchaus auf den Speicherverbrauch auswirkt.

Da die benötigte Zeit für die Verwaltung von Listen, wie sie bei den Heuristiken vorgenommen wird, mit der Anzahl der zu verwaltenden Elemente wächst, kann davon ausgegangen werden, dass die Anwendung der Heuristiken ebenfalls mehr Zeit in Anspruch nimmt, je weiter fortgeschritten der *Explorationsprozess* ist. Dies gilt insbesondere für die Heuristiken *PTTF* und *BL_NMC*.

6.1.4 Zusammenfassung

Die Ausführungen der Abschnitt 6.1.1 und 6.1.2 lassen vermuten, dass die lediglich die Heuristiken *LMF* und *BL_NMC* eine Daseinsberechtigung haben. Dies ist nicht korrekt. Die Heuristik *PTTF* liefert zwar schlechtere Ergebnisse, dennoch hat sie die zu generierenden und zu prüfenden *Proxies* im Vergleich zum Ausgangspunkt (siehe Abschnitt 5.2) stark reduziert. Zudem haben die Entwickler*innen bei der Verwendung der Heuristik *PTTF* keinen höheren Aufwand bei der Implementierung der Testfälle.

Dies gilt auch für die Heuristik *LMF*. Diese kann aufgrund dessen, dass sie sich lediglich auf den finalen Durchlauf des *semantischen Evaluation* positiv auswirkt, nur in wenigen Fällen mit der Heuristik *BL_NMC* mithalten. Allerdings gilt auch hier, dass keine weiteren Anforderungen an die Arbeit der Entwickler*innen gestellt werden. Dazu kommt noch, dass die Ermittlung der *Matcherratings* quasi bei dem Matching der Typen mit abfällt, wodurch die Verwendung dieser Heuristik kaum eine Auswirkung auf die Komplexität des *Explorationsprozesses* hat.

Die Heuristik *BL_NMC*, welche sich in dieser Untersuchung häufig als diejenige mit den besten Ergebnissen herausgestellt hat, bedarf einer speziellen Implementierung der Testfälle. Weiterhin ist davon auszugehen, dass diese Heuristik von allen in dieser Arbeit beschriebenen Heuristiken aufgrund der Menge an Informationen, die für diese Heuristik gesammelt werden, den größten negativen Einfluss auf die Komplexität des *Explorationsprozesses* hat.

6.2 Kritik am Ansatz

6.2.1 Seiteneffekte durch Testevaluation

Der beschriebene *Explorationsprozess* erfordert die Ausführung der vordefinierten Testfälle zur Laufzeit. Sofern diese Testfälle eine Änderung des Zustands bestimmter Objekte bewirken, kann dies auch Auswirkungen auf die Funktionsweise des Systems haben.

Um dieses Problem zu beheben könnte sichergestellt werden, dass die Generierung der *Proxies* nur auf Basis von *provided Typen* (*EJBs*) erfolgt, die solche Seiteneffekte nicht aufweisen.

Diese Eigenschaft kann jedoch nur durch die Entwickler*innen festgestellt. Solche *EJBs* können dann bspw. über Annotationen markiert werden. Während des *Explorationsprozesses* könnten solche *EJBs* über solche Markierungen erkannt werden. Dieser Ansatz reduziert jedoch die Anzahl der *provided Typen*, die für die Generierung eines *Proxies* verwendet werden können. Dadurch sinkt auch die Wahrscheinlichkeit, dass ein passender *Proxy* gefunden wird.

Um die zu markierenden *EJBs* zu identifizieren ist zu prüfen, wie sich die Ausführung der einzelnen Methoden der *Bean* auf das System auswirken. Es kann festgehalten werden, dass alle Methoden, die den persistenten oder den transienten Zustand von Objekten verändern, das Potential für solche unerwünschten Seiteneffekte besitzen.

Aufbauend auf einer solchen Prüfung einzelner Methoden, kann auch die Markierung von Methoden in Betracht gezogen werden. So dürften markierte Methoden bei der Generierung eines *Proxies* nicht als *Delegationsmethode* verwendet werden.

6.2.2 Auswirkung auf die Verfügbarkeit eines Systems

Die Verfügbarkeit eines Systems bzw. von Systemkomponenten, bezeichnet die Wahrscheinlichkeit, ein System oder Systemkomponenten zu einem vorgegebenen Zeitpunkt in einem funktionsfähigen Zustand anzutreffen. [IT 21] Die Auswirkung des Ansatzes auf die Verfügbarkeit wurde in dieser Arbeit nicht systematisch untersucht. Da der Ansatz jedoch darauf abzielt, bestimmte Komponenten - in diesem Fall *EJBs* - zur Laufzeit zu kombinieren, können Überlegungen bzgl. der Verfügbarkeit durchaus angestellt werden.

Dabei muss allerdings bedacht werden, dass die Betrachtung der Verfügbarkeit einer *EJB* in diesem Zusammenhang nicht ausreichend ist. Immerhin kann ein passender *Proxy* auch auf einer Kombination von *EJBs* erzeugt werden. Insofern bilden eher die Methoden, die von den EJBs angeboten werden, die Komponenten in Bezug auf die oben beschriebene betrachtete Verfügbarkeit.

Ausgehend davon kann die These aufgestellt werden, dass mit diesem Ansatz eine höhere Verfügbarkeit erreicht wird, sofern die Methoden im System redundant vorliegen. Dabei ist

das Vorliegen redundanter Methoden innerhalb einer Systems wahrscheinlicher, als das Vorliegen redundanter Komponenten, die diese Methoden enthalten[1].

6.2.3 Auswirkung von Änderungen an bestehenden Komponenten

Da die *EJBs* bei dem vorgestellten Ansatz nicht explizit adressiert werden, weiß der Entwickler auch nicht, an welche *EJBs* die Methodenaufrufe letztendlich delegiert werden. Somit sind die Auswirkungen von Änderungen an bestehenden Komponenten nicht direkt vorhersehbar, da sich die Menge der matchenden *provided Typen* (*EJBs*) und dementsprechend auch die generierten *Proxies* ändern.

Im Folgenden wird zum einen die Erweiterung um zusätzlichen *provided Typen* und zum anderen die Entfernung von *provided Typen* betrachtet. Dabei sei angenommen, dass die *required Typen*, zu denen ein passender *Proxy* gefunden werden soll, nicht verändert werden.

Erweiterungen um neue Komponenten

Die Erweiterung von Systemen geht in Bezug auf den beschriebenen Ansatz zur testgetriebenen Exploration zur Laufzeit damit einher, dass sich die Anzahl der *provided Typen* verändert. Wie in Abschnitt 3.2.4 beschrieben, besteht damit auch die Gefahr, dass die Anzahl der möglichen *Proxies* steigt. Dazu muss jedoch gelten, dass eine Methode aus einem *required Typ* auf eine der Methode aus dem neuen *provided Typ* gematcht werden kann.

Mehrere mögliche *Proxies* haben wiederum einen Einfluss auf die Laufzeit und das Ergebnis des *Explorationsprozesses*. So kann nicht davon ausgegangen werden, dass ein passender *Proxy* zu einem bestimmten *required Typ* genauso schnell gefunden wird, nachdem ein *provided Typen* im System ergänzt wurde.

Entfernen von bestehenden Komponenten

Ebenso wirkt sich das Entfernen eines *provided Typs*, der während eines früheren *Explorationsprozesses* für die Generierung eines *Proxies* verwendet wurde, auf den *Explorationsprozess*

[1] Dies ist bei *EJBs* der Fall, denn eine *EJB* enthält Methoden und nicht anders herum.

nach einer solchen Änderung aus. Dadurch, dass der früher verwendete *provided Typ* nicht mehr vorhanden ist, muss ein anderer *Proxy*, der auf andere *provided Typen* basiert, erzeugt werden[2].

Da der *Explorationsprozess* beendet wird, sofern ein passender *Proxy* gefunden wurde, kann es auch unter diesen Umständen dazu kommen, dass der *Explorationsprozess* mitunter länger dauert als vorher. Zudem besteht in diesem Fall die Gefahr, dass während des *Explorationsprozesses* kein passender *Proxy* gefunden wird.

6.2.4 Nutzen für den Entwickler

Aus den vorherigen Absätzen ergibt sich, dass die Entwickler*innen bei der Verwendung dieses Ansatzes eine große Verantwortung tragen. Dieser Verantwortung können sie umso besser gerecht werden, je genauer sie das System, in dem der Ansatz verwendet werden soll, kennen.

So kann festgehalten werden, dass Entwickler*innen, die das System gut kennen und somit wissen, welche Komponenten innerhalb dessen verwendet werden, diesen Ansatz wohl kaum benötigen. Vielmehr ist es ihnen möglich die passenden Komponenten aufgrund ihres Wissens explizit zu benennen, wie es im *EJB*-Framework grundlegend der Fall ist.

Entwickler*innen, die das System hingegen weniger kennen, können von diesem Ansatz profitieren, da sie nicht selbst nach einer für ihren Anwendungsfall passenden *EJB* (mitunter auch mehreren) suchen müssen. Diese können sie über die Deklaration eines *required Typen* und der Spezifikation dazugehöriger Tests suchen lassen. Dabei ist jedoch zu erwähnen, dass der *Explorationsprozess* insbesondere mit der vorgestellten Heuristik *LMF* umso schneller ist, je genauer die in den Methoden des *required Typs* verwendeten Typen mit den Typen, die in den Methoden der *provided Typen* übereinstimmen (*Matcherrating*).

Ist den Entwickler*innen das System unbekannt, besteht die Gefahr, dass der *required Typ* so deklariert wird, dass das *Matcherrating* relativ hoch ausfällt und somit der *Explorationsprozess* mehr Zeit in Anspruch nimmt.

[2]sofern dies gelingt, unterstützt dies die These aus Abschnitt 6.2.2

Zusammenfassend kann folgende These formuliert werden: Der Nutzen dieses Ansatzes für Entwickler*innen steht im umgekehrt proportionalen Verhältnis zum Wissen dieser Entwickler*innen über das System, in dem der Ansatz verwendet werden soll.

6.3 Erweiterungsmöglichkeiten

6.3.1 Zusätzliche Matcher

Eine mögliche Erweiterung des Ansatzes wäre die Definition und Implementierung zusätzlicher Matcher. Diese würde es ermöglichen, dass der Abstraktionsgrad zwischen den Typen, die in den Methoden der *required Typen* und *provided Typen* verwendet werden, noch weiter auseinandergeht, als es bei den vorgestellten Matchern in Abschnitt 3.1.2 der Fall ist (Identität, Vererbung, Container).

Die vorgestellten Matcher beachten beispielsweise keine impliziten Typumwandlungen (*Coercions*). Diese können je nach Programmiersprache abweichen, was eine formale und allgemeine Beschreibung wie in Abschnitt 3.1.2 eines solchen Matchers (*CoercionMatcher*) erschwert. So müsste ein *CoercionMatcher* für jede Programmiersprache explizit spezifiziert werden.

Die Programmiersprache Java bietet eine Vielzahl solcher impliziten Typumwandlungen an [GJS+15a]. Dabei ist zu beachten, dass es implizite Typumwandlungen gibt, die ohne Informationsverlust vonstatten gehen[3] und solche, bei denen ein Informationsverlust nicht auszuschließen ist[4].

Implizite Typumwandlungen ohne Informationsverlust sind in Bezug auf die weitere Verwendung innerhalb eines *Proxies* unbedenklich. Diese sind hinsichtlich des Informationsverlustes mit dem *GenTypeMatcher* vergleichbar, welcher in Abschnitt 3.1.2 beschrieben wurde. So wie ein Typ A, der über den *GenTypeMatcher* zu einem Typ B gematcht wird ($B \Rightarrow_{gen} A$), ohne Probleme anstelle des Typen B verwendet werden kann, kann auch ein Typ C, der ohne Informationsverlust implizit aus B umgewandelt wurde, anstelle von B verwendet werden.

[3] bspw. *Identity Conversion* oder *Widening Primitive Conversion* [GJS+15a]
[4] bspw. *Narrowing Primitive Conversion* [GJS+15a]

Anders ist es bei impliziten Typumwandlungen mit Informationsverlust. Diese sind eher mit dem *SpecTypeMatcher* vergleichbar (siehe Abschnitt 3.1.2). In der Spezifikation des darauf aufbauenden *Proxy*-Generators ist zu erkennen, dass durch eine solche Typumwandlung bestimmte *Methodendelegationen* in einen Fehler münden. Da sich der *SpecTypeMatcher* direkt auf die Vererbungsbeziehung der beiden Typen bezieht, kann die Ursache solcher Fehler auf die Methoden zurückgeführt werden, die zwar im Subtyp jedoch nicht im Supertyp implementiert sind. Bei einem *CoercionMatcher*, der in Abhängigkeit der Programmiersprache spezifiziert wird, kann es andere Fehlerursachen geben.

Aus diesem Grund wäre es sinnvoll, nicht einen einzigen Matcher zu spezifizieren, der alle impliziten Typumwandlungen abdeckt. Vielmehr sollten die in der Programmiersprache definierten *Coercions* nach dem möglichem Informationsverlust kategorisiert werden und dann je Kategorie ein Matcher spezifiziert werden.

Darüber hinaus ist zu beachten, dass die Spezifikation eines Matchers alleine nicht ausreicht, um diesen zu integrieren. Da die Heuristik *LMF* auf dem *Matcherrating* aufbaut, ist es ebenso notwendig, den zusätzlichen Matchern ein Basisrating zuzuweisen. Wie in Abschnitt 4.1 beschrieben, wird dieses Basisrating von der Implementierung des Matchers geliefert. Dabei gilt es jedoch zu beachten, dass das Basisrating eines zusätzlichen Matchers im korrekten Verhältnis zu den bestehenden Matchern steht.

In Bezug auf den/die *CoercionMatcher* gibt es hierbei mehrere Möglichkeiten. Beispielsweise könnte für den/die *CoercionMatcher* ein Basisrating zwischen 100 und 200 verwendet. Die untere Schranke von 100 wird dadurch begründet, dass es kein besseres Matching gibt, als die Identität, welche durch den *ExactTypeMatcher* mit einem Basisrating von 100 beschrieben wird. Die obere Schranke von 200 könnte damit begründet werden, dass es sich um Typumwandlungen handelt, die über die Programmiersprache definiert sind und diese somit sicherer sind als ein Downcast, der durch den *SpecTypeMatcher* mit einem Basisrating von 200 abgedeckt werden.

6.3.2 Default-Implementierungen in required Typen

Im Abschnitt 2.2 wurde darauf aufmerksam gemacht, dass der *Explorationsprozess* das Auffinden eines passenden *Proxies* nicht garantiert. Die Entwickler*innen muss also in einem solchen Fall eine alternative Implementierung bereitstellen.

Dass ein passender *Proxy* nicht gefunden wurde, kann allgemein betrachtet zwei Ursachen haben: Entweder konnte kein *Proxy* generiert werden, oder keiner der generierten *Proxies* erfüllt alle vordefinierten Test.

Die Generierung eines *Proxies* hängt von dem Matching der Methoden des *required Typs* und der Methoden der *provided Typen* ab. Aufgrund dessen dass der Entwickler Testfälle für den *required Typ* spezifizieren muss, hat er eine grundlegende Vorstellung von den Ein- und Ausgabewerten der Methoden, sowie der Verarbeitung dieser. Um nun der Gefahr vorzubeugen, dass gar kein *Proxy* generiert werden kann, könnten die Entwickler*innen eine Implementierung, die seine Erwartungen zumindest minimal erfüllt, als Default-Methode in dem Interface zum *required Typ* aufnehmen. Sofern bei der Exploration zu dieser Methode keine passende Methode aus einem *provided Typ* gefunden wird, kann auf die Default-Implementierung zurückgegriffen werden. Der generierte *Proxy*, welcher technisch gesehen das Interface zum *required Typ* implementiert, würde den Methodenaufruf dann an sich selbst bzw. an die Default-Methode delegieren.

Ein Beispiel für eine solche Konstellation zeigen die folgenden Listings. In Listing 6.1 ist der *required Typ Calc* deklariert. Listing 6.2 zeigt das dazugehörige Java-Interface mit der Default-Methode `div`. Die Implementierung wurde so umgesetzt, dass die Testfälle, welche in der Klasse in Listing 6.3 enthalten sind, positiv ausfallen.

```
required Calc {
 Float div( int a, int b )
}
```

Listing 6.1: Required Typ *Calc*

```
@RequiredTypeTestReference( testClasses = CalcTest.class )
public interface Calc {
```

```
 default Float div(int a, int b){
  if(b == 0)
   return null;
  return Float.valueOf(a/b)
 }

}
```

Listing 6.2: Interface Calc

```
public class CalcTest {

 private Calc calc;

 @RequiredTypeInstanceSetter
 public void setProvider( Calc calc ) {
  this.calc = calc;
 }

 @RequiredTypeTest
 public void testDivByZero() {
  assertThat( calc.dev(1,0), nullValue() );
 }

 @RequiredTypeTest
 public void testDiv() {
  assertThat( calc.dev(4,2), equalTo(2) );
 }

}
```

Listing 6.3: Test CalcTest

Dadurch ist zwar immer noch nicht sichergestellt, dass ein passender *Proxy* in jedem Fall gefunden wird, aber den Entwickler*innen kann ein alternatives Verhalten direkt im Interface zum *required Typ* implementieren, wodurch diese Implementierung einen sehr engen Bezug zum *required Typ* hat.

Kapitel 7

Schlussbemerkung

7.1 Zusammenfassung

Zusammenfassend ist zu sagen, dass die vorgestellten Heuristiken ihren Zweck erfüllen und gemessen an der Anzahl der zu generierenden und zu prüfenden *Proxies* eine schnellere Exploration nach einem passenden *Proxy* ermöglichen. Dabei konnten auch Synergieeffekte zwischen den einzelnen Heuristiken festgestellt werden.

Weiterhin wurde gezeigt, dass die testgetriebene Exploration von *EJBs* zur Laufzeit grundlegend funktioniert. Dennoch gibt es Szenarien, in denen von diesem Verfahren eher abzuraten ist. Das betrifft insbesondere solche *EJBs*, durch deren Methodenaufrufe eine Änderung an ihrem inneren Zustand bezweckt wird. Es wurden jedoch Möglichkeiten aufgezeigt, wie mit solchen Fällen umgegangen werden kann.

Ob der Ansatz der testgetriebenen Exploration zur Laufzeit im Allgemeinen einen Nutzen verspricht wurde nicht geklärt. Wenn dies überhaupt der Fall ist, dann hängt der Nutzen vermutlich mit dem Wissen der Entwickler*innen zusammen, welches sie über das vorliegende System aufweisen können.

Unabhängig davon wurde in dieser Arbeit eine allgemeine formale Beschreibung für Matcher von Wrapper-Typen gegeben (*ContentTypeMatcher* und *ContainerTypeMatcher*).

Zudem können die entwickelten Module, welche in Kapitel 4 beschrieben wurden, in unterschiedlichen Systemen verwendet werden. Hinsichtlich des Repositories haben die Entwickler*innen sehr viel Freiraum und sind nicht auf einen *EJB-Container* beschränkt. Weiterhin können neue Matcher durch die Implementierung der dafür vorgesehenen Interfaces in die Module integriert werden, was den Nutzen des Ansatzes in einem System individuell steigern kann.

7.2 Ausblick

Die Heuristiken wurden für die Exploration zur Laufzeit entworfen. In einem nächsten Schritt könnte versucht werden, diese Heuristiken in bestehende Search Engines wie *Merobase* oder *CodeGenie* zu integrieren und deren Nutzen in diesem Kontext zu untersuchen.

Weiterhin wäre es interessant zu untersuchen, ob und wie dieser Ansatz der Exploration von Komponenten zur Laufzeit in anderen Systemtypen wie bspw. Self-Contained-Systems funktioniert. Mitunter ergeben sich bei diesen Untersuchungen weitere Vorteile oder Probleme dieses Ansatzes.

Darüber hinaus bieten die in Abschnitt 6.2 aufgestellten Thesen bzgl. der höheren Verfügbarkeit (Abschnitt 6.2.1) und dem Nutzen des Ansatzes für die Entwickler*innen im Verhältnis zu deren Wissen über das System das Potential für weitere Untersuchungen.

Anhang A

Kombination von Matchern

Wie aus der formalen Beschreibung zum *StructuralTypeMatcher* im Abschnitt 3.1.2 hervorgeht, ist dieser von den übrigen Matchern abhängig. Die Implementierung der dazugehörigen Klasse `StructuralTypeMatcher` verlangt zur Erzeugung eines Objektes dieser Klasse eine `TypeMatcher`. Dieser `TypeMatcher` muss laut der formalen Beschreibung die Implementierung der Matchingrelation $\Rightarrow_{internStruct}$ darstellen.

Zu diesem Zweck müssen die übrigen Matcher bzw. die dafür implementierten Klassen miteinander kombiniert werden, wie es in der Definition zur Matchingrelation $\Rightarrow_{internStruct}$ der Fall ist. Für solche Kombinationen steht die Klasse `MatcherCombiner` im Modul *SignatureMatching* bereit (siehe Listing A.1).

Diese Klasse erlaubt die Kombination von Objekten vom Typ `TypeMatcher`. Die Matcher-Klassen `ExactTypeMatcher`, `GenSpecTypeMatcher` und `WrappedTypeMatcher` implementieren alle dieses Interface.

Über die Methode `combine` in der `MatcherCombiner` wird bei der Kombination ein Supplier-Objekt erzeuge, welches über die `get`-Methode ein Objekt vom Typ `TypeMatcher` liefern kann. Dieses `TypeMatcher`-Objekt versucht beim Aufruf der Methode `matchesType(S,T)` die beiden Typen S und T über einen der kombinierten Matcher zu matchen (siehe Listing A.1). Dabei liefert die Methode `getSortedMatcher` eine sortiert Liste der kombinierten Matcher. Die

Sortierung wird aufsteigend entsprechend dem Basisrating (siehe auch Abschnitt 3.4.1) der kombinierten Matcher vorgenommen .

```
package de.fernuni.hagen.ma.gundermann.signaturematching.matching;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.function.Supplier;

import de.fernuni.hagen.ma.gundermann.signaturematching.SingleMatchingInfo;
import de.fernuni.hagen.ma.gundermann.signaturematching.matching.types.TypeMatcher;

public final class MatcherCombiner {
 private MatcherCombiner() {
 }

 public static Supplier<TypeMatcher> combine(TypeMatcher... matcher) {
  return () -> new TypeMatcher() {

   @Override
   public boolean matchesType(Class<?> checkType, Class<?> queryType) {
        for (TypeMatcher m : getSortedMatcher()) {
         if (m.matchesType(checkType, queryType)) {
          return true;
         }
        }
        return false;
   }

   @Override
   public Collection<SingleMatchingInfo> calculateTypeMatchingInfos(Class<?>
       checkType, Class<?> queryType) {
        for (TypeMatcher m : getSortedMatcher()) {
         if (m.matchesType(checkType, queryType)) {
          return m.calculateTypeMatchingInfos(checkType, queryType);
         }
        }
        return new ArrayList<>();
   }

   @Override
   public MatcherRate matchesWithRating(Class<?> checkType, Class<?> queryType) {
```

```
        for (TypeMatcher m : getSortedMatcher()) {
         MatcherRate rating = m.matchesWithRating(checkType, queryType);
         if (rating != null) {
          return rating;
         }
        }
        return null;
    }

    @Override
    public double getTypeMatcherRate() {
        // irrelevant, weil matchesWithRating ueberschrieben wurde.
        return -0;
    }

    private Collection<TypeMatcher> getSortedMatcher() {
        List<TypeMatcher> matcherList = Arrays.asList(matcher);
        Collections.sort(matcherList,(l1, l2) ->
            Double.compare(l1.getTypeMatcherRate(), l2.getTypeMatcherRate()));
        return matcherList;
    }

  };
 }
}
```

Listing A.1: Klasse: MatcherCombiner

Bei der Exploration wird letztendlich immer ein Objekt des der Klasse `StructuralTypeMatcher` zur Ermittlung des Matchings verwendet. Listing A.2 zeigt die Instanziierung dieses Objektes unter Verwendung der Klasse `MatcherCombiner`.

```
TypeMatcher exactTM = new ExactTypeMatcher();
TypeMatcher genSpecTM = new GenSpecTypeMatcher();
TypeMatcher combinedGenSpecExactTM = MatcherCombiner.combine(genSpecTM,
    exactTM).get();
TypeMatcher wrappedTM = new WrappedTypeMatcher(() -> combinedGenSpecExactTM);
TypeMatcher combinedWrappedGenSpecExact = MatcherCombiner.combine(genSpecTM,
    exactTM, wrappedTM).get();
StructuralTypeMatcher structWrappedGenSpecExactTM = new StructuralTypeMatcher(
                                () -> combinedWrappedGenSpecExact);
```

Listing A.2: Default-Instanziierung des StructuralTypeMatchers im DesiredComponentFinder

Anhang B

Verwendung aller Heuristiken

Die in den Abschnitten 3.4.1 - 3.4.3 vorgestellten Heuristiken können miteinander kombiniert werden. Listing B.1 zeigt die Implementierung der Funktionen, die für diese Kombination auf der Basis von Listing 3.10 angepasst oder ergänzt werden müssen.

```
1  function evalProxiesMitTarget( proxies, tests ){
2    testedProxies = []
3    for( proxy : proxies ){
4      passedTestcases = 0
5      blacklistChanged = false
6      evalProxy(proxy, tests)
7      if( passedTests == T.size ){
8        // passenden Proxy gefunden
9        return proxy
10     }
11     else{
12       testedProxies.add(proxy)
13       if( passedTests > 0 || blacklistChanged ){
14         // noch nicht evaluierte Proxies ermitteln
15         optmizedProxies = proxies.removeAll( testedProxies )
16         // Heuristik PTTF
17         if( passedTests > 0 ){
18           priorityTargets.addAll( proxy.targets )
19           optmizedProxies = PTTF( optmizedProxies )
20         }
21         // Heuristik BL_FFMD und BL_FSMT
22         if( blacklistChanged ){
23           optmizedProxies = BL( optmizedProxies )
24         }
25         return evalProxiesMitTarget( optmizedProxies, tests )
```

```
26        }
27      }
28    }
29    // kein passenden Proxy gefunden
30    return null
31  }
32
33  function evalProxy(proxy, tests){
34    for( test : tests ){
35      //alle Tests werden durchgefuehrt
36      try{
37        if( test.eval( proxy ) ){
38          passedTestcases = passedTestcases + 1
39        }elseif( test.isSingleMethodTest ){
40          methodName = test.testedSingleMethodName
41          mDel = getMethodDelegation( proxy, methodName )
42          methodDelegationBlacklist.add( mDel )
43          blacklistChanged = true
44          return
45        }
46      }
47      catch (SigMaGlueException e){
48        mDel = e.failedMethodDelegation
49        methodDelegationBlacklist.add( mDel )
50        blacklistChanged = true
51        return
52      }
53    }
54  }
55
56  function relevantProxies( proxies, anzahl ){
57    relProxies = proxiesMitTargets( proxies, anzahl );
58    optimizedLMF = LMF( relProxies )
59    optimizedPTTF = PTTF( optimizedLMF )
60    return BL( optimizedPTTF )
61  }
```

Listing B.1: Kombination aller Heuristiken

Anhang C

Deklaration der relevanten Typen

Im Folgenden erfolgt die Deklaration der *required Typen*, mit denen die Evaluation der Heuristiken in Kapitel 5 durchgeführt wird, sowie die Deklaration der *provided Typen*, die als Ergebnis der jeweiligen Exploration für einen *required Typ* einzeln oder in Kombination erwartet, oder innerhalb einer der Deklarationen eines *required Typ* verwendet werden. Dabei ist davon auszugehen, dass diese Typen aus dem JDK als Bibliothek aufbauen.

Die Listings C.1 - C.7 zeigen die Deklarationen für die *required Typen*.

```
required ElerFTFoerderprogrammeProvider{
        Collection getAlleFreigegebenenFPs()
        ElerFTFoerderprogramm getElerFTFoerderprogramm(DvAntragsJahr,
            DvFoerderprogramm, Date)
}
```

Listing C.1: Deklaration von ElerFTFoerderprogrammeProvider

```
required FoerderprogrammeProvider{
        Collection getAlleFreigegebenenFPs()
        Foerderprogramm getFoerderprogramm(DvAntragsJahr, DvFoerderprogramm, Date)
}
```

Listing C.2: Deklaration von FoerderprogrammeProvider

```
required MinimalFoerderprogrammeProvider{
        Collection getAlleFreigegebenenFPs()
        Foerderprogramm getFoerderprogramm(String, int, Date)
```

```
}
```

Listing C.3: Deklaration von MinimalFoerderprogrammeProvider

```
required IntubatingFireFighter{
        void intubate(Injured)
        FireState extinguishFire(Fire)
}
```

Listing C.4: Deklaration von IntubatingFireFighter

```
required IntubatingFreeing{
        void intubate(Injured)
        void free(Injured)
}
```

Listing C.5: Deklaration von IntubatingFreeing

```
required IntubatingFreeing{
        void intubate(IntubationPatient)
        FireState extinguishFire(Fire)
}
```

Listing C.6: Deklaration von IntubatingPatientFireFighter

```
required KOFGPCProvider{
        Collection getKOFGsVonFP(DvFoerderprogramm)
        Collection getPCsZuKOFG(DvFoerdergegenstand, DvAntragsJahr)
}
```

Listing C.7: Deklaration von KOFGPCProvider

Die Listings C.8 - C.14 zeigen die *provided Typen*, die in den Deklarationen der *required Typen* verwendet wurden und nicht Teil des JDKs sind.

```
provided ElerFTFoerderprogramm extends Foerderprogramm{
  DvFlaeche mindestParzellenGroesse
  DvFlaeche maximaleParzellenGroesse
  int differenzKassenjahrAntragsjahr
  boolean isMehrjaehrig

  DvFlaeche getMaximaleParzellengroesse()
  DvFlaeche getMindestParzellenGroesse()
  int getDifferenzKassenjahrAntragsjahr()
  boolean isMehrjaehrig()
}
```

Listing C.8: Deklaration von ElerFTFoerderprogramm

```
provided Foerderprogramm extends Object{
  Long id
  STDGueltigkeit gueltigkeit
  Long fpId
  BigDecimal bagatellbetrag
  BigDecimal bagatellmenge
  List vorgaengeAm15
  Set landesmassnahmen

  Long getId()
  boolean isTechnischGueltig(Date)
  DvFoerderprogramm getFoerderprogramm()
  BigDecimal getBagatellmengeFoerd()
  BigDecimal getBagatellbetragFoerd()
  boolean isFachlichGueltig(DvAntragsJahr)
  STDGueltigkeit getGueltigkeit()
  Long getFpId()
}
```

Listing C.9: Deklaration von Foerderprogramm

```
provided DvAntragsJahr extends AbstractDomainValue{
  int antragsJahr

  DvAntragsJahr add(int)
  int compareTo(Object)
  int intValue()
  Object readResolve()
  DvAntragsJahr getVorjahr()
  int differenz(DvAntragsJahr)
  DvAntragsJahr sub(int)
  String toStringImpl()
}
```

Listing C.10: Deklaration von DvAntragsJahr

```
provided DvFoerderprogramm extends DvEnumerable{
  long id
  String code
  String fpGruppe
  String bezeichnung
  String bezeichnungLang
  String getName()

  Long getId()
  Long getNummer()
```

```
  void validateCode(String)
  String getFpGruppe()
  String getBezeichnung()
  String toStringImpl()
  String getCode()
  String getFPNummerExtern()
  String getBezeichnungLang()
}
```

Listing C.11: Deklaration von DvFoerderprogramm

```
provided Injured extends Object{
  Collection suffers

  Collection getSuffers()
  void healSuffer(Suffer)
  boolean isStabilized()
}
```

Listing C.12: Deklaration von Injured

```
provided Fire extends Object{
  boolean active

  void extinguish()
  boolean isActive()
}
```

Listing C.13: Deklaration von Fire

```
provided IntubationPatient extends Object{
  boolean isIntubated

  boolean isIntubated()
  void setIntubated(boolean)
}
```

Listing C.14: Deklaration von IntubationPatient

Die Listings C.15 - C.18 zeigen die Deklarationen der *provided Typen*, aus denen bei der Exploration ein passender Proxy erzeugt werden soll.

```
provided ElerFTStammdatenAuskunftService extends Object{
        Collection getAlleElerFTKombiKzFpFoerdergegenstaende()
        Collection getAlleElerFTKoFoerdergegenstaende()
        Collection
            getFeststellungscodeVerpflichtungList(FeststellungscodeVerpflichtungImplQuery)
```

```
FeststellungscodeVerpflichtungImpl
    getFeststellungscodeVerpflichtungImpl(FeststellungscodeVerpflichtungImplQuery)
Collection getAlleElerFTTierFoerdergegenstaende(DvFoerderprogramm,
    DvAntragsJahr, AntragsVorgangsTyp)
Collection getAlleFreigegebenenFoerderprogramme(AntragsVorgangsTyp)
Collection getAlleFreigegebenenFoerderprogramme()
ElerFTKzFpFoerdergegenstand2Foerderfaehigkeit
    getElerFTKzFpFoerdergegenstand2Foerderfaehigkeit(DvFoerdergegenstand,
    DvAntragsJahr)
FeststellungsCodeVerpflichtung2FP
    getFeststellungsCodeVerpflichtung2FP(FeststellungsCodeVerpflichtung2FPQuery)
DvEftOekoFoerdergegenstandGruppe
    getOekoFgGruppe2Foerdergegenstand(DvFoerdergegenstand)
Collection getAlleElerFTKzFpFoerdergegenstaende()
VerpflichtungsGegenstandImpl
    getVerpflichtungsGegenstandImpl(VerpflichtungsGegenstandImplQuery)
ElerFTVorhaben getVorhaben2Foerdergegenstand(DvFoerdergegenstand,
    DvAntragsJahr)
Verpflichtungszeitraum getVerpflichtungszeitraum(DvFoerderprogramm,
    DvAntragsJahr)
int getMaxStandardAnzahlZahlungen(DvFoerderprogramm, DvAntragsJahr)
DvZusatzInfoTyp getZusatzInfo2Foerdergegenstand(DvFoerdergegenstand,
    DvAntragsJahr)
int getStandardAnzahlZahlungen(DvUntermassnahme, DvAntragsJahr)
int getStandardAnzahlZahlungen(Landesmassnahme, DvAntragsJahr)
Collection getElerFTKoFoerdergegenstaende(DvFoerderprogramm,
    DvUntermassnahme, DvAntragsJahr)
Collection getElerFTKoFoerdergegenstaende(DvFoerderprogramm)
Collection getAlleFg2ZusatzInfo(DvZusatzInfoTyp, DvAntragsJahr)
int getDifferenzJahrVerpflbeginnEAJ(DvFoerderprogramm, DvAntragsJahr)
Collection getVerpflichtungsGegenstandList(VerpflichtungsGegenstandImplQuery)
Collection getAenderungscodePropertiesList(AenderungscodePropertiesQuery)
Collection getAlleFg2OekoFgGruppe(DvEftOekoFoerdergegenstandGruppe)
ElerFTFoerderprogramm getFoerderprogramm(ElerFTFoerderprogrammQuery)
ElerFTFoerderprogramm getFoerderprogramm(DvAntragsJahr, DvFoerderprogramm,
    Date)
Collection getElerFTAenderung2ElerFTFP(DvFoerderprogramm)
Collection getElerFTAenderung2ElerFTFP(ElerFTAenderung)
ElerFTAenderung2ElerFTFP getElerFTAenderung2ElerFTFP(ElerFTAenderung,
    DvFoerderprogramm)
Collection getFoerdergegenstaende(AbstractElerFTFoerdergegenstandQuery)
Collection getElerFTTierFoerdergegenstaende(DvFoerderprogramm,
    DvUntermassnahme, DvAntragsJahr)
Collection getFoerderprogramme(ElerFTFoerderprogrammQuery)
Collection getFoerderprogramme(Date)
```

```
        Collection getAlleFoerderprogramme()
        Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm,
            DvUntermassnahme, DvAntragsJahr)
        Collection getElerFTKzFpFoerdergegenstaende(ElerFTKombiKzFpFoerdergegenstand)
        Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm,
            Finanzierungsschluessel, DvAntragsJahr)
        Collection getElerFTKzFpFoerdergegenstaende(DvFoerderprogramm, DvAntragsJahr)
        Collection getAlleFg2Vorhaben(ElerFTVorhaben, DvAntragsJahr)
        Map getKzFpJeFg(Collection, DvAntragsJahr)
}
```

Listing C.15: Deklaration von ElerFTStammdatenAuskunftService

```
provided StammdatenAuskunftService extends Object{
  Collection
      getLandesmassnahmen2Foerdergegenstaende(Landesmassnahme2FoerdergegenstandQuery)
  Collection getFoerdergegenstaendeZuFinanzierungsschluessel(DvFoerderprogramm,
      Finanzierungsschluessel, DvAntragsJahr)
  Landesmassnahme getLandesmassnahme(Long)
  Map getOberFgJeUnterFg(DvAntragsJahr)
  Collection getFoerderprogramme(Date)
  Foerdergegenstand getFoerdergegenstand(FoerdergegenstandQuery)
  Collection getFoerdergegenstaende(DvFoerderprogramm)
  Collection getFoerdergegenstaende(FoerdergegenstandQuery)
  Collection getFoerdergegenstaende(Landesmassnahme)
  Collection getFinanzierungsschluessel(FinanzierungsschluesselQuery)
  Collection getFinanzierungskonfigurationen(FinanzierungskonfigurationQuery)
  Collection getFinanzierungskonfigurationen(Collection, DvAntragsJahr)
  Collection getFinanzierungskonfigurationen(DvAntragsJahr, DvFoerderprogramm, Long)
  Finanzierungskonfiguration getFinanzierungskonfigurationen(DvAntragsJahr,
      DvFoerderprogramm, DvFoerdergegenstand)
  Map getProduktcodesJeFg(DvFoerderprogramm, DvAntragsJahr, Collection,
      ProduktcodeArt, Finanzierungsschluessel)
  Foerderprogramm getFoerderprogramm(Foerdergegenstand)
  Foerderprogramm getFoerderprogramm(DvAntragsJahr, DvFoerderprogramm, Date)
  Collection getAblehnungsgrundCodes(Foerderprogramm, DvAntragsJahr,
      KuerzungsgrundCode)
  Collection getUnterFoerdergegenstaende(DvAntragsJahr, Collection)
  Collection getFoerdergegenstandGruppenZuFgs(DvAntragsJahr, Collection)
  Collection getLandesmassnahmen(DvAntragsJahr, DvFoerderprogramm)
  Collection getLandesmassnahmen(DvAntragsJahr, Foerdergegenstand)
  Collection getLandesmassnahmen(LandesmassnahmeQuery)
  Produktcode getProduktcode(ProduktcodeQuery)
  Produktcode getProduktcode(DvAntragsJahr, DvFoerdergegenstand, ProduktcodeArt)
  Produktcode getProduktcode(DvAntragsJahr, DvFoerdergegenstand, ProduktcodeArt,
      Finanzierungsschluessel)
```

```
  BigDecimal getBeihilfesatz(DvAntragsJahr, DvFoerdergegenstand, Integer)
  Collection getProduktcodes(DvAntragsJahr, Finanzierungsschluessel)
  Collection getProduktcodes(DvAntragsJahr, DvFoerdergegenstand,
      Finanzierungsschluessel)
  Collection getProduktcodes(ProduktcodeQuery)
  Collection getProduktcodes(DvAntragsJahr, DvFoerderprogramm)
  Collection getProduktcodes(DvAntragsJahr, DvFoerdergegenstand)
  Collection getProduktcodes(Collection)
  BigDecimal getKappungBetrag(DvFoerdergegenstand, DvAntragsJahr)
  Collection getVorgaenge(Date, DvFoerderprogramm)
  Collection getVorgaenge(AntragsVorgangsTyp)
  Collection getVorgaenge(Date, AntragsVorgangsTyp)
  Collection getVorgaenge()
  Collection getVorgaenge(DvFoerderprogramm, Date, AntragsVorgangsTyp)
  BigDecimal getKappungMenge(DvFoerdergegenstand, DvAntragsJahr)
  Vorgang getVorgang(DvAntragsJahr, DvFoerderprogramm, Date, AntragsVorgangsTyp,
      DvAntragsJahr)
  Vorgang getVorgang(DvFoerderprogramm, Date, AntragsVorgangsTyp, DvAntragsJahr)
}
```

Listing C.16: Deklaration von StammdatenAuskunftService

```
provided Doctor extends Object{
  void provideHeartbeatMassage(Injured)
  void stablilizeBrokenBones(Injured)
  void healWithMed(Injured, Medicine)
  void placeInfusion(Injured)
  void nurseWounds(Injured)
  void intubate(Injured)
}
```

Listing C.17: Deklaration von Doctor

```
provided FireFighter extends Object{
  void stabilizeBrokenBones(Injured)
  void provideHeartbeatMassage(Injured)
  FireState extinguishFire(Fire)
  void free(Injured)
  void nurseWounds(Injured)
}
```

Listing C.18: Deklaration von FireFighter

Anhang D

Interfaces und Test-Implementierungen

Im Folgenden werden zum einen die Interfaces, die sich aus den Deklarationen der *required Typen* aus dem Anhang C ableiten lassen, aufgeführt. Zum anderen werden die Implementierungen der Testklassen, auf die die oben genannten Interfaces über die Annotation `RequiredTypeTestReference` verweisen, dargelegt.

Die Listings D.1 - D.7 zeigen dabei die Deklarationen der Java-Interfaces[1] für die *required Typen* aus Tabelle 5.1 aus Kapitel 5.

```
@RequiredTypeTestReference( testClasses = ElerFTFoerderprogrammProviderTest.class )
public interface ElerFTFoerderprogrammeProvider {

  Collection<ElerFTFoerderprogramm> getAlleFreigegebenenFPs();

  ElerFTFoerderprogramm getElerFTFoerderprogramm( DvAntragsJahr jahr,
      DvFoerderprogramm fp, Date date );

}
```

Listing D.1: Interface ElerFTFoerderprogrammeProvider

```
@RequiredTypeTestReference( testClasses = FoerderprogrammProviderTest.class )
public interface FoerderprogrammeProvider {

 Collection<Foerderprogramm> getAlleFreigegebenenFPs();
```

[1] Auf die Import-Anweisungen wurde verzichtet.

```
    Foerderprogramm getFoerderprogramm( DvFoerderprogramm fp, DvAntragsJahr jahr,
        Date date );

}
```

Listing D.2: Interface FoerderprogrammeProvider

```
@RequiredTypeTestReference( testClasses = MinimalFoerderprogrammProviderTest.class )
public interface MinimalFoerderprogrammeProvider {

  Collection<String> getAlleFreigegebenenFPs();

  Foerderprogramm getFoerderprogramm( String fp, int jahr, Date date );

}
```

Listing D.3: Interface MinimalFoerderprogrammeProvider

```
@RequiredTypeTestReference( testClasses = IntubatingFireFighterTest.class )
public interface IntubatingFireFighter {

  public void intubate( Injured injured );

  public FireState extinguishFire( Fire fire );

}
```

Listing D.4: Interface IntubatingFireFighter

```
@RequiredTypeTestReference( testClasses = IntubatingFreeingTest.class )
public interface IntubatingFreeing {

  public void intubate( Injured injured );

  public void free( Injured injured );

}
```

Listing D.5: Interface IntubatingFreeing

```
@RequiredTypeTestReference( testClasses = IntubatingPatientFireFighterTest.class )
public interface IntubatingPatientFireFighter {

  public void intubate( IntubationPartient patient );

  public FireState extinguishFire( Fire fire );
```

```
}
```

Listing D.6: Interface IntubatingPatientFireFighter

```
@RequiredTypeTestReference( testClasses = KOFGPCProviderTest.class )
public interface KOFGPCProvider {

  Collection<ElerFTKoFoerdergegenstand> getKOFGsVonFP( DvFoerderprogramm fp );

  Collection<Produktcode> getPCsZuKOFG( DvFoerdergegenstand fg, DvAntragsJahr aj );

}
```

Listing D.7: Interface KOFGPCProvider

Zu erkennen ist, dass jedes Interfaces, wie in Abschnitt 4.2 beschrieben, mit der Annotation `RequiredTypeTestReference` versehen ist, über die auf eine Java-Klasse verwiesen wird, in der die Tests zu dem jeweiligen *required Typ* implementiert sind.

Die Listings D.8 - D.14 zeigen die Implementierungen dieser Testklassen[2].

```
public class ElerFTFoerderprogrammProviderTest implements TriedMethodCallsInfo {

  private ElerFTFoerderprogrammeProvider provider;
  private Collection<Method> calledMethods = new ArrayList<Method>();

  @RequiredTypeInstanceSetter
  public void setProvider( ElerFTFoerderprogrammeProvider provider ) {
    this.provider = provider;
  }

  @RequiredTypeTest
  public void testEmptyCollection() {
    addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
        ElerFTFoerderprogrammeProvider.class ) );
    Collection<ElerFTFoerderprogramm> alleFreigegebenenFPs =
        provider.getAlleFreigegebenenFPs();
    assertThat( alleFreigegebenenFPs, notNullValue() );
  }

  @RequiredTypeTest
  public void testMockedFPCollection() {
```

[2]Auf die Import-Anweisungen wurde verzichtet.

```
    DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
        DvFoerderprogramm.FP215 );
    addTriedMethodCall( getMethod( "getElerFTFoerderprogramm",
        ElerFTFoerderprogrammeProvider.class ) );
    ElerFTFoerderprogramm alleFreigegebenenFPs = provider.getElerFTFoerderprogramm(
        DvAntragsJahr.AJ2020,
        fp, new Date() );
    assertThat( alleFreigegebenenFPs, nullValue() );
  }

  @Override
  public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
  }

  @Override
  public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
  }

}
```

Listing D.8: Interface ElerFTFoerderprogrammProviderTest

```
public class FoerderprogrammProviderTest implements TriedMethodCallsInfo {

  private FoerderprogrammeProvider provider;

  private Collection<Method> calledMethods = new ArrayList<Method>();

  @RequiredTypeInstanceSetter
  public void setProvider( FoerderprogrammeProvider provider ) {
    this.provider = provider;
  }

  @RequiredTypeTest
  public void testEmptyCollection() {
    addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
        FoerderprogrammeProvider.class ) );
    Collection<Foerderprogramm> alleFreigegebenenFPs =
        provider.getAlleFreigegebenenFPs();
    assertThat( alleFreigegebenenFPs, notNullValue() );
  }

  @RequiredTypeTest
  public void testMockedFPCollection() {
```

```
    DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
        DvFoerderprogramm.FP508 );
    addTriedMethodCall( getMethod( "getFoerderprogramm",
        FoerderprogrammeProvider.class ) );
    Foerderprogramm relevantFP = provider.getFoerderprogramm( fp,
        DvAntragsJahr.AJ2020,
        new Date() );
    assertThat( relevantFP, notNullValue() );
  }

  @RequiredTypeTest
  public void testDZFPCollection() {
    DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
        DvFoerderprogramm.FP215 );
    addTriedMethodCall( getMethod( "getFoerderprogramm",
        FoerderprogrammeProvider.class ) );
    Foerderprogramm relevantFP = provider.getFoerderprogramm( fp,
        DvAntragsJahr.AJ2020,
        new Date() );
    assertThat( relevantFP, notNullValue() );
  }

  @Override
  public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
  }

  @Override
  public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
  }

}
```

Listing D.9: Interface FoerderprogrammProviderTest

```
public class MinimalFoerderprogrammProviderTest implements TriedMethodCallsInfo {

  private MinimalFoerderprogrammeProvider provider;
  private Collection<Method> calledMethods = new ArrayList<Method>();

  @RequiredTypeInstanceSetter
  public void setProvider( MinimalFoerderprogrammeProvider provider ) {
    this.provider = provider;
  }
```

```
  @RequiredTypeTest
  public void testEmptyCollection() {
    addTriedMethodCall( getMethod( "getAlleFreigegebenenFPs",
        MinimalFoerderprogrammeProvider.class) );
    Collection<String> alleFreigegebenenFPs = provider.getAlleFreigegebenenFPs();
    assertThat( alleFreigegebenenFPs, notNullValue() );
  }

  @RequiredTypeTest
  public void testGetFoerderprogramm() {
    String fpCode = "215";
    addTriedMethodCall( getMethod( "getFoerderprogramm",
        MinimalFoerderprogrammeProvider.class) );
    Foerderprogramm fp = provider.getFoerderprogramm( fpCode, 2015, new Date() );
    assertThat( fp, notNullValue() );
    DvFoerderprogramm dvFP = fp.getFoerderprogramm();
    assertThat( dvFP, notNullValue() );

    String code = dvFP.getCode();
    assertThat( fpCode, equalTo( code ) );

  }

  @Override
  public void addTriedMethodCall( Method method ) {
    calledMethods.add( method );
  }

  @Override
  public Collection<Method> getTriedMethodCalls() {
    return calledMethods ;
  }

}
```

Listing D.10: Interface MinimalFoerderprogrammProviderTest

```
public class IntubatingFireFighterTest implements TriedMethodCallsInfo {

        private IntubatingFireFighter intubatingFireFighter;
        private Collection<Method> calledMethods = new ArrayList<Method>();

        @RequiredTypeInstanceSetter
        public void setProvider(IntubatingFireFighter intubatingFireFighter) {
                this.intubatingFireFighter = intubatingFireFighter;
        }
```

```
        @RequiredTypeTest
        public void free() {
                Fire fire = new Fire();
                addTriedMethodCall(getMethod("extinguishFire",
                    IntubatingFireFighter.class));
                FireState fireState = intubatingFireFighter.extinguishFire(fire);
                assertTrue(Objects.equals(fireState.isActive(), fire.isActive()));
                assertFalse(fire.isActive());
        }

        @RequiredTypeTest
        public void intubate() {
                Collection<Suffer> suffer = Arrays.asList(Suffer.BREATH_PROBLEMS);
                Injured patient = new Injured(suffer);
                addTriedMethodCall(getMethod("intubate",
                    IntubatingFireFighter.class));
                intubatingFireFighter.intubate(patient);
                assertTrue(patient.isStabilized());
        }

        @Override
        public void addTriedMethodCall(Method m) {
                calledMethods.add(m);
        }

        @Override
        public Collection<Method> getTriedMethodCalls() {
                return calledMethods;
        }

}
```

Listing D.11: Interface IntubatingFireFighterTest

```
public class IntubatingFreeingTest implements TriedMethodCallsInfo {

        private IntubatingFreeing intubatingFreeing;
        private Collection<Method> calledMethods = new ArrayList<Method>();

        @RequiredTypeInstanceSetter
        public void setProvider(IntubatingFreeing intubatingFireFighter) {
                this.intubatingFreeing = intubatingFireFighter;
        }

        @RequiredTypeTest
```

```
        public void free() {
                Collection<Suffer> suffer = Arrays.asList(Suffer.LOCKED);
                Injured patient = new Injured(suffer);
                addTriedMethodCall(getMethod("free", IntubatingFreeing.class));
                intubatingFreeing.free(patient);
                assertTrue(patient.isStabilized());
        }

        @RequiredTypeTest
        public void intubate() {
                Collection<Suffer> suffer = Arrays.asList(Suffer.BREATH_PROBLEMS);
                Injured patient = new Injured(suffer);
                addTriedMethodCall(getMethod("intubate", IntubatingFreeing.class));
                intubatingFreeing.intubate(patient);
                assertTrue(patient.isStabilized());
        }

        @Override
        public void addTriedMethodCall(Method m) {
                calledMethods.add(m);
        }

        @Override
        public Collection<Method> getTriedMethodCalls() {
                return calledMethods;
        }

}
```

Listing D.12: Interface IntubatingFreeingTest

```
public class IntubatingPatientFireFighterTest implements TriedMethodCallsInfo {

        private IntubatingPatientFireFighter intubatingPatientFireFighter;
        private Collection<Method> calledMethods = new ArrayList<Method>();

        @RequiredTypeInstanceSetter
        public void setProvider(IntubatingPatientFireFighter intubatingFireFighter) {
                this.intubatingPatientFireFighter = intubatingFireFighter;
        }

        @RequiredTypeTest
        public void extinguishFire() {
                Fire fire = new Fire();
                addTriedMethodCall(getMethod("extinguishFire",
                    IntubatingPatientFireFighter.class));
```

```
                FireState fireState =
                    intubatingPatientFireFighter.extinguishFire(fire);
                assertTrue(Objects.equals(fireState.isActive(), fire.isActive()));
                assertFalse(fire.isActive());
        }

        @RequiredTypeTest
        public void intubate() {
                IntubationPartient patient = new IntubationPartient();
                addTriedMethodCall(getMethod("intubate",
                    IntubatingPatientFireFighter.class));
                intubatingPatientFireFighter.intubate(patient);
                assertTrue(patient.isIntubated());
        }

        @Override
        public void addTriedMethodCall(Method m) {
                calledMethods.add(m);
        }

        @Override
        public Collection<Method> getTriedMethodCalls() {
                return calledMethods;
        }

}
```

Listing D.13: Interface IntubatingPatientFireFighterTest

```
public class KOFGPCProviderTest implements TriedMethodCallsInfo {

  private KOFGPCProvider provider;

  private Collection<Method> calledMethods = new ArrayList<Method>();

  @RequiredTypeInstanceSetter
  public void setProvider( KOFGPCProvider provider ) {
    this.provider = provider;
  }

  @RequiredTypeTest
  public void testKOFGsCollection() {
    DvFoerderprogramm fp = DvFoerderprogramm.Factory.valueOf(
        DvFoerderprogramm.FP508 );
    addTriedMethodCall( getMethod( "getKOFGsVonFP", KOFGPCProvider.class ) );
    Collection<ElerFTKoFoerdergegenstand> kofGsVonFP = provider.getKOFGsVonFP( fp );
```

```
    assertThat( kofGsVonFP, notNullValue() );
    assertThat( kofGsVonFP.isEmpty(), equalTo( false ) );
    assertThat( kofGsVonFP.stream().anyMatch( fg -> fg.getCode().equals( "KO508" )
        ), equalTo( true ) );
  }

  @RequiredTypeTest
  public void testPCsCollection() {
    DvFoerdergegenstand fg = DvFoerdergegenstand.Factory.valueOf( 20155080025L );
    addTriedMethodCall( getMethod( "getPCsZuKOFG", KOFGPCProvider.class ) );
    Collection<Produktcode> pcs = provider.getPCsZuKOFG( fg, DvAntragsJahr.AJ2020 );
    assertThat( pcs, notNullValue() );
    assertThat( pcs.isEmpty(), equalTo( false ) );
  }

  @Override
  public void addTriedMethodCall( Method m ) {
    this.calledMethods.add( m );
  }

  @Override
  public Collection<Method> getTriedMethodCalls() {
    return calledMethods;
  }

}
```

Listing D.14: Interface KOFGPCProviderTest

Hier ist zu erkennen, dass die Testklassen alle das Interfaces `TriedMethodCallsInfo` implementieren, über das die für die Heuristik *BL_NMC* benötigten Informationen (siehe Abschnitt 3.4.3) ermittelt werden. Ebenso ist die Implementierung dieses Interfaces in den oben genannten Listings zu erkennen.

Anhang E

Ergebnisse für die Heuristik LMF (Ergänzungen)

In diesem Abschnitt werden die Untersuchungsergebnisse der Heuristik *LMF* mit allen Varianten zur Bestimmung des Matcherratings aus Abschnitt 3.4.1 dargelegt. Dieses Kapitel bildet somit eine Ergänzung zu Abschnitt 5.3. Die darin beschriebenen Ergebnisse der Variante *1.1* werden der Vollständigkeit halber in dem vorliegenden Kapitel nochmals aufgeführt.

Die folgenden Ergebnisse beziehen sich auf die in Kapitel 5 vorgestellten *required Typen TEI1-TEI7*.

Ergebnisse für Variante 1.1

1	positiv	negativ
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle E.1: Ergebnisse *LMF* mit Variante 1.1 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	1889	$p_1(55) - 1890$
richtig	1	0

Tabelle E.2: Ergebnisse *LMF* mit Variante 1.1 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1463	$p_1(50) - 1464$
richtig	1	0

Tabelle E.3: Ergebnisse *LMF* mit Variante 1.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.4: Ergebnisse *LMF* mit Variante 1.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(2247) - 3$
richtig	1	0

Tabelle E.5: Ergebnisse *LMF* mit Variante 1.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.6: Ergebnisse *LMF* mit Variante 1.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	32	$p_2(2775) - 33$
richtig	1	0

Tabelle E.7: Ergebnisse *LMF* mit Variante 1.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.8: Ergebnisse *LMF* mit Variante 1.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.9: Ergebnisse *LMF* mit Variante 1.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.10: Ergebnisse *LMF* mit Variante 1.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	7641	$p_2(52150) - 7642$
richtig	1	0

Tabelle E.11: Ergebnisse *LMF* mit Variante 1.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.2

1	positiv	negativ
falsch	1	$p_1(44) - 2$
richtig	1	0

Tabelle E.12: Ergebnisse *LMF* mit Variante 1.2 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	2783	$p_1(55) - 2784$
richtig	1	0

Tabelle E.13: Ergebnisse *LMF* mit Variante 1.2 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1830	$p_1(50) - 1831$
richtig	1	0

Tabelle E.14: Ergebnisse *LMF* mit Variante 1.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.15: Ergebnisse *LMF* mit Variante 1.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	3	$p_2(2247) - 4$
richtig	1	0

Tabelle E.16: Ergebnisse *LMF* mit Variante 1.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.17: Ergebnisse *LMF* mit Variante 1.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	3	$p_2(2775) - 4$
richtig	1	0

Tabelle E.18: Ergebnisse *LMF* mit Variante 1.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.19: Ergebnisse *LMF* mit Variante 1.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.20: Ergebnisse *LMF* mit Variante 1.2 für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	161294	0
richtig	0	0

Tabelle E.21: Ergebnisse *LMF* mit Variante 1.2 für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	161298	$p_2(52150) - 161299$
richtig	1	0

Tabelle E.22: Ergebnisse *LMF* mit Variante 1.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.3

1	**positiv**	**negativ**
falsch	50	$p_1(44) - 51$
richtig	1	0

Tabelle E.23: Ergebnisse *LMF* mit Variante 1.3 für TEI1 1. Durchlauf

1	**positiv**	**negativ**
falsch	20	$p_1(55) - 21$
richtig	1	0

Tabelle E.24: Ergebnisse *LMF* mit Variante 1.3 für TEI2 1. Durchlauf

1	**positiv**	**negativ**
falsch	121	$p_1(50) - 122$
richtig	1	0

Tabelle E.25: Ergebnisse *LMF* mit Variante 1.3 für TEI3 1. Durchlauf

1	**positiv**	**negativ**
falsch	1174	0
richtig	0	0

Tabelle E.26: Ergebnisse *LMF* mit Variante 1.3 für TEI4 1. Durchlauf

2	**positiv**	**negativ**
falsch	57	$p_2(2247) - 58$
richtig	1	0

Tabelle E.27: Ergebnisse *LMF* mit Variante 1.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.28: Ergebnisse *LMF* mit Variante 1.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	6246	$p_2(2775) - 6247$
richtig	1	0

Tabelle E.29: Ergebnisse *LMF* mit Variante 1.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.30: Ergebnisse *LMF* mit Variante 1.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	5	$p_2(1323) - 6$
richtig	1	0

Tabelle E.31: Ergebnisse *LMF* mit Variante 1.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.32: Ergebnisse *LMF* mit Variante 1.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	121074	$p_2(52150) - 121075$
richtig	1	0

Tabelle E.33: Ergebnisse *LMF* mit Variante 1.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 1.4

1	positiv	negativ
falsch	45	$p_1(44) - 46$
richtig	1	0

Tabelle E.34: Ergebnisse *LMF* mit Variante 1.4 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	2025	$p_1(55) - 2026$
richtig	1	0

Tabelle E.35: Ergebnisse *LMF* mit Variante 1.4 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1517	$p_1(50) - 1518$
richtig	1	0

Tabelle E.36: Ergebnisse *LMF* mit Variante 1.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.37: Ergebnisse *LMF* mit Variante 1.4 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	4	$p_2(2247) - 5$
richtig	1	0

Tabelle E.38: Ergebnisse *LMF* mit Variante 1.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.39: Ergebnisse *LMF* mit Variante 1.4 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	34	$p_2(2775) - 35$
richtig	1	0

Tabelle E.40: Ergebnisse *LMF* mit Variante 1.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.41: Ergebnisse *LMF* mit Variante 1.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.42: Ergebnisse *LMF* mit Variante 1.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.43: Ergebnisse *LMF* mit Variante 1.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	21068	$p_2(52150) - 21069$
richtig	1	0

Tabelle E.44: Ergebnisse *LMF* mit Variante 1.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.1

1	positiv	negativ
falsch	8	$p_1(44) - 9$
richtig	1	0

Tabelle E.45: Ergebnisse *LMF* mit Variante 2.1 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	3975	$p_1(55) - 3976$
richtig	1	0

Tabelle E.46: Ergebnisse *LMF* mit Variante 2.1 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	2933	$p_1(50) - 2934$
richtig	1	0

Tabelle E.47: Ergebnisse *LMF* mit Variante 2.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.48: Ergebnisse *LMF* mit Variante 2.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(2247) - 3$
richtig	1	0

Tabelle E.49: Ergebnisse *LMF* mit Variante 2.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.50: Ergebnisse *LMF* mit Variante 2.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	32	$p_2(2775) - 33$
richtig	1	0

Tabelle E.51: Ergebnisse *LMF* mit Variante 2.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.52: Ergebnisse *LMF* mit Variante 2.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.53: Ergebnisse *LMF* mit Variante 2.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.54: Ergebnisse *LMF* mit Variante 2.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	32018037	$p_2(52150) - 32018038$
richtig	1	0

Tabelle E.55: Ergebnisse *LMF* mit Variante 2.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.2

1	positiv	negativ
falsch	0	$p_1(44) - 1$
richtig	1	0

Tabelle E.56: Ergebnisse *LMF* mit Variante 2.2 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	8007	$p_1(55) - 8008$
richtig	1	0

Tabelle E.57: Ergebnisse *LMF* mit Variante 2.2 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	7104	$p_1(50) - 7105$
richtig	1	0

Tabelle E.58: Ergebnisse *LMF* mit Variante 2.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.59: Ergebnisse *LMF* mit Variante 2.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(2247) - 1$
richtig	1	0

Tabelle E.60: Ergebnisse *LMF* mit Variante 2.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.61: Ergebnisse *LMF* mit Variante 2.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(2775) - 1$
richtig	1	0

Tabelle E.62: Ergebnisse *LMF* mit Variante 2.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.63: Ergebnisse *LMF* mit Variante 2.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.64: Ergebnisse *LMF* mit Variante 2.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.65: Ergebnisse *LMF* mit Variante 2.2 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	2840500	$p_2(52150) - 2840501$
richtig	1	0

Tabelle E.66: Ergebnisse *LMF* mit Variante 2.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.3

1	positiv	negativ
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle E.67: Ergebnisse *LMF* mit Variante 2.3 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	2642	$p_1(55) - 2643$
richtig	1	0

Tabelle E.68: Ergebnisse *LMF* mit Variante 2.3 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1686	$p_1(50) - 1687$
richtig	1	0

Tabelle E.69: Ergebnisse *LMF* mit Variante 2.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.70: Ergebnisse *LMF* mit Variante 2.3 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	67	$p_2(2247) - 68$
richtig	1	0

Tabelle E.71: Ergebnisse *LMF* mit Variante 2.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.72: Ergebnisse *LMF* mit Variante 2.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	5413	$p_2(2775) - 5414$
richtig	1	0

Tabelle E.73: Ergebnisse *LMF* mit Variante 2.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.74: Ergebnisse *LMF* mit Variante 2.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	11	$p_2(1323) - 12$
richtig	1	0

Tabelle E.75: Ergebnisse *LMF* mit Variante 2.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.76: Ergebnisse *LMF* mit Variante 2.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	8084753	$p_2(52150) - 8084754$
richtig	1	0

Tabelle E.77: Ergebnisse *LMF* mit Variante 2.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 2.4

1	positiv	negativ
falsch	20	$p_1(44) - 21$
richtig	1	0

Tabelle E.78: Ergebnisse *LMF* mit Variante 2.4 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	3928	$p_1(55) - 3929$
richtig	1	0

Tabelle E.79: Ergebnisse *LMF* mit Variante 2.4 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	3117	$p_1(50) - 3118$
richtig	1	0

Tabelle E.80: Ergebnisse *LMF* mit Variante 2.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.81: Ergebnisse *LMF* mit Variante 2.4 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	3	$p_2(2247) - 4$
richtig	1	0

Tabelle E.82: Ergebnisse *LMF* mit Variante 2.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.83: Ergebnisse *LMF* mit Variante 2.4 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	33	$p_2(2775) - 34$
richtig	1	0

Tabelle E.84: Ergebnisse *LMF* mit Variante 2.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.85: Ergebnisse *LMF* mit Variante 2.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.86: Ergebnisse *LMF* mit Variante 2.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.87: Ergebnisse *LMF* mit Variante 2.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	10899025	$p_2(52150) - 10899026$
richtig	1	0

Tabelle E.88: Ergebnisse *LMF* mit Variante 2.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.1

1	positiv	negativ
falsch	1037	$p_1(44) - 1038$
richtig	1	0

Tabelle E.89: Ergebnisse *LMF* mit Variante 3.1 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	3956	$p_1(55) - 3957$
richtig	1	0

Tabelle E.90: Ergebnisse *LMF* mit Variante 3.1 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	3851	$p_1(50) - 3852$
richtig	1	0

Tabelle E.91: Ergebnisse *LMF* mit Variante 3.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.92: Ergebnisse *LMF* mit Variante 3.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	191	$p_2(2247) - 192$
richtig	1	0

Tabelle E.93: Ergebnisse *LMF* mit Variante 3.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.94: Ergebnisse *LMF* mit Variante 3.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	1608	$p_2(2775) - 1609$
richtig	1	0

Tabelle E.95: Ergebnisse *LMF* mit Variante 3.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.96: Ergebnisse *LMF* mit Variante 3.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	37	$p_2(1323) - 38$
richtig	1	0

Tabelle E.97: Ergebnisse *LMF* mit Variante 3.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.98: Ergebnisse *LMF* mit Variante 3.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	758477	$p_2(52150) - 758478$
richtig	1	0

Tabelle E.99: Ergebnisse *LMF* mit Variante 3.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.2

1	positiv	negativ
falsch	1097	$p_1(44) - 1098$
richtig	1	0

Tabelle E.100: Ergebnisse *LMF* mit Variante 3.2 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	386	$p_1(55) - 387$
richtig	1	0

Tabelle E.101: Ergebnisse *LMF* mit Variante 3.2 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	121	$p_1(50) - 122$
richtig	1	0

Tabelle E.102: Ergebnisse *LMF* mit Variante 3.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.103: Ergebnisse *LMF* mit Variante 3.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	524	$p_2(2247) - 525$
richtig	1	0

Tabelle E.104: Ergebnisse *LMF* mit Variante 3.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.105: Ergebnisse *LMF* mit Variante 3.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	3402	$p_2(2775) - 3403$
richtig	1	0

Tabelle E.106: Ergebnisse *LMF* mit Variante 3.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.107: Ergebnisse *LMF* mit Variante 3.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	115	$p_2(1323) - 116$
richtig	1	0

Tabelle E.108: Ergebnisse *LMF* mit Variante 3.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.109: Ergebnisse *LMF* mit Variante 3.2 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	379600	$p_2(52150) - 379601$
richtig	1	0

Tabelle E.110: Ergebnisse *LMF* mit Variante 3.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.3

1	positiv	negativ
falsch	4088	$p_1(44) - 4089$
richtig	1	0

Tabelle E.111: Ergebnisse *LMF* mit Variante 3.3 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	2005	$p_1(55) - 2006$
richtig	1	0

Tabelle E.112: Ergebnisse *LMF* mit Variante 3.3 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	1776	$p_1(50) - 1777$
richtig	1	0

Tabelle E.113: Ergebnisse *LMF* mit Variante 3.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.114: Ergebnisse *LMF* mit Variante 3.3 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	55881	$p_2(2247) - 55882$
richtig	1	0

Tabelle E.115: Ergebnisse *LMF* mit Variante 3.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.116: Ergebnisse *LMF* mit Variante 3.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	239768	$p_2(2775) - 239769$
richtig	1	0

Tabelle E.117: Ergebnisse *LMF* mit Variante 3.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.118: Ergebnisse *LMF* mit Variante 3.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	42748	$p_2(1323) - 42749$
richtig	1	0

Tabelle E.119: Ergebnisse *LMF* mit Variante 3.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.120: Ergebnisse *LMF* mit Variante 3.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	4912200	$p_2(52150) - 4912201$
richtig	1	0

Tabelle E.121: Ergebnisse *LMF* mit Variante 3.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 3.4

1	positiv	negativ
falsch	5105	$p_1(44) - 5106$
richtig	1	0

Tabelle E.122: Ergebnisse *LMF* mit Variante 3.4 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	3598	$p_1(55) - 3599$
richtig	1	0

Tabelle E.123: Ergebnisse *LMF* mit Variante 3.4 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	3421	$p_1(50) - 3422$
richtig	1	0

Tabelle E.124: Ergebnisse *LMF* mit Variante 3.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.125: Ergebnisse *LMF* mit Variante 3.4 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	762	$p_2(2247) - 763$
richtig	1	0

Tabelle E.126: Ergebnisse *LMF* mit Variante 3.4 für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	4984	0
richtig	0	0

Tabelle E.127: Ergebnisse *LMF* mit Variante 3.4 für TEI5 1. Durchlauf

2	**positiv**	**negativ**
falsch	6130	$p_2(2775) - 6131$
richtig	1	0

Tabelle E.128: Ergebnisse *LMF* mit Variante 3.4 für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	1051	0
richtig	0	0

Tabelle E.129: Ergebnisse *LMF* mit Variante 3.4 für TEI6 1. Durchlauf

2	**positiv**	**negativ**
falsch	141	$p_2(1323) - 142$
richtig	1	0

Tabelle E.130: Ergebnisse *LMF* mit Variante 3.4 für TEI6 2. Durchlauf

1	**positiv**	**negativ**
falsch	161294	0
richtig	0	0

Tabelle E.131: Ergebnisse *LMF* mit Variante 3.4 für TEI7 1. Durchlauf

2	**positiv**	**negativ**
falsch	788327	$p_2(52150) - 788328$
richtig	1	0

Tabelle E.132: Ergebnisse *LMF* mit Variante 3.4 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.1

1	positiv	negativ
falsch	0	$p_1(44) - 1$
richtig	1	0

Tabelle E.133: Ergebnisse *LMF* mit Variante 4.1 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	516	$p_1(55) - 517$
richtig	1	0

Tabelle E.134: Ergebnisse *LMF* mit Variante 4.1 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	185	$p_1(50) - 186$
richtig	1	0

Tabelle E.135: Ergebnisse *LMF* mit Variante 4.1 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.136: Ergebnisse *LMF* mit Variante 4.1 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(2247) - 3$
richtig	1	0

Tabelle E.137: Ergebnisse *LMF* mit Variante 4.1 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.138: Ergebnisse *LMF* mit Variante 4.1 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(2775) - 3$
richtig	1	0

Tabelle E.139: Ergebnisse *LMF* mit Variante 4.1 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.140: Ergebnisse *LMF* mit Variante 4.1 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.141: Ergebnisse *LMF* mit Variante 4.1 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.142: Ergebnisse *LMF* mit Variante 4.1 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	314549	$p_2(52150) - 314550$
richtig	1	0

Tabelle E.143: Ergebnisse *LMF* mit Variante 4.1 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.2

1	positiv	negativ
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle E.144: Ergebnisse *LMF* mit Variante 4.2 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	4132	$p_1(55) - 4133$
richtig	1	0

Tabelle E.145: Ergebnisse *LMF* mit Variante 4.2 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	3847	$p_1(50) - 3848$
richtig	1	0

Tabelle E.146: Ergebnisse *LMF* mit Variante 4.2 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.147: Ergebnisse *LMF* mit Variante 4.2 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(2247) - 1$
richtig	1	0

Tabelle E.148: Ergebnisse *LMF* mit Variante 4.2 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.149: Ergebnisse *LMF* mit Variante 4.2 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(2775) - 1$
richtig	1	0

Tabelle E.150: Ergebnisse *LMF* mit Variante 4.2 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.151: Ergebnisse *LMF* mit Variante 4.2 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.152: Ergebnisse *LMF* mit Variante 4.2 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.153: Ergebnisse *LMF* mit Variante 4.2 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	445110	$p_2(52150) - 445111$
richtig	1	0

Tabelle E.154: Ergebnisse *LMF* mit Variante 4.2 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.3

1	positiv	negativ
falsch	5	$p_1(44) - 6$
richtig	1	0

Tabelle E.155: Ergebnisse *LMF* mit Variante 4.3 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	6015	$p_1(55) - 6016$
richtig	1	0

Tabelle E.156: Ergebnisse *LMF* mit Variante 4.3 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	6353	$p_1(50) - 6354$
richtig	1	0

Tabelle E.157: Ergebnisse *LMF* mit Variante 4.3 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.158: Ergebnisse *LMF* mit Variante 4.3 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	37	$p_2(2247) - 38$
richtig	1	0

Tabelle E.159: Ergebnisse *LMF* mit Variante 4.3 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.160: Ergebnisse *LMF* mit Variante 4.3 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	4006	$p_2(2775) - 4007$
richtig	1	0

Tabelle E.161: Ergebnisse *LMF* mit Variante 4.3 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.162: Ergebnisse *LMF* mit Variante 4.3 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	2	$p_2(1323) - 3$
richtig	1	0

Tabelle E.163: Ergebnisse *LMF* mit Variante 4.3 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.164: Ergebnisse *LMF* mit Variante 4.3 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	5433499	$p_2(52150) - 5433500$
richtig	1	0

Tabelle E.165: Ergebnisse *LMF* mit Variante 4.3 für TEI7 2. Durchlauf

Ergebnisse für Variante 4.4

1	positiv	negativ
falsch	25	$p_1(44) - 26$
richtig	1	0

Tabelle E.166: Ergebnisse *LMF* mit Variante 4.4 für TEI1 1. Durchlauf

1	positiv	negativ
falsch	1286	$p_1(55) - 1287$
richtig	1	0

Tabelle E.167: Ergebnisse *LMF* mit Variante 4.4 für TEI2 1. Durchlauf

1	positiv	negativ
falsch	981	$p_1(50) - 982$
richtig	1	0

Tabelle E.168: Ergebnisse *LMF* mit Variante 4.4 für TEI3 1. Durchlauf

1	positiv	negativ
falsch	1174	0
richtig	0	0

Tabelle E.169: Ergebnisse *LMF* mit Variante 4.4 für TEI4 1. Durchlauf

2	positiv	negativ
falsch	1	$p_2(2247) - 2$
richtig	1	0

Tabelle E.170: Ergebnisse *LMF* mit Variante 4.4 für TEI4 2. Durchlauf

1	positiv	negativ
falsch	4984	0
richtig	0	0

Tabelle E.171: Ergebnisse *LMF* mit Variante 4.4 für TEI5 1. Durchlauf

2	positiv	negativ
falsch	31	$p_2(2775) - 32$
richtig	1	0

Tabelle E.172: Ergebnisse *LMF* mit Variante 4.4 für TEI5 2. Durchlauf

1	positiv	negativ
falsch	1051	0
richtig	0	0

Tabelle E.173: Ergebnisse *LMF* mit Variante 4.4 für TEI6 1. Durchlauf

2	positiv	negativ
falsch	0	$p_2(1323) - 1$
richtig	1	0

Tabelle E.174: Ergebnisse *LMF* mit Variante 4.4 für TEI6 2. Durchlauf

1	positiv	negativ
falsch	161294	0
richtig	0	0

Tabelle E.175: Ergebnisse *LMF* mit Variante 4.4 für TEI7 1. Durchlauf

2	positiv	negativ
falsch	500063	$p_2(52150) - 500064$
richtig	1	0

Tabelle E.176: Ergebnisse *LMF* mit Variante 4.4 für TEI7 2. Durchlauf

Anhang F

Beweise

Theorem 1

Theorem. *Sei R ein* required Typ *innerhalb einer Bibliothek L. Ein* struktureller Proxy *für R lässt sich nur aus den Mengen $TM \in cover(R, L)$ generieren, für die gilt:*

$$|TM| \leq |methods(R)|$$

Beweis. In Bezug auf alle *strukturellen Proxies* $P \in proxies_{struct}(R, TM)$, drückt das Theorem folgendes aus:

$$\forall P \in proxies_{struct}(R, TM) : TM| \leq |methods(R)|$$

Da ein *struktureller Proxy* $P \in proxies_{struct}(R, TM)$ der Bedingung $targets_{multi}(P, TM)$ unterliegt (siehe Abschnitt 3.2.3), muss gelten:

$$|P.targets| = |TM|$$

Weiterhin gilt aufgrund von $targets_{multi}(P, TM)$, dass für jeden *Target-Typ* eine *Methoden-Delegation* existiert, die diesen *Target-Typ* im Attribut `target` enthält:

$$\forall T \in P.targets : \exists MD \in P.dels : MD.del.target = T$$

Daraus folgt für die Mächtigkeit der *Methoden-Delegationen*:

$$P.dels.len \geq |P.targets|$$
$$P.dels.len \geq |TM|$$

Zusätzlich gilt aufgrund der Regel $delegationCount_{struct}(P)$ (siehe Abschnitt 3.2.3):

$$|methods(R)| = P.dels.len$$

Daraus folgt direkt:

$$\forall P \in proxies_{struct}(R, TM) : TM| \leq |methods(R)|$$

QED

Theorem 2

Theorem. *Sei R ein* required Typ *aus einer Bibliothek L. Sei weiterhin $C = cover(R, L)$. Ferner seien $TM \in C$ und $TM' \in C$ mit $proxies_{struct}(R, TM) \neq \emptyset$ sowie $proxies_{struct}(R, TM') \neq \emptyset$ und $|TM| < |TM'|$ gegeben.*

Dann gilt:

$$\forall T \in TM : \exists TM'' \in targetSets(C, |TM'|) : proxies_{struct}(R, TM'') \neq \emptyset \wedge T \in TM''$$

Beweis. Sofern es zwei Mengen von *Target-Typen* eines *strukturellen Proxies* unterschiedlicher Mächtigkeit gibt - so wie es bei TM und TM' der Fall ist, gibt in der Menge mit der geringeren Mächtigkeit auch immer einen Typ der mind. zwei Methoden enthält, die zu den Methoden des *required Typ* R gematcht werden können. Anderenfalls würde Theorem 1 nicht gelten. Darauf aufbauend kann jede Konstellation von Mengen von *Target-Typen*, auf die die oben genannten Voraussetzungen zutreffen, auf das folgende Szenario reduziert werden.

Angenommen R enthält zwei Methoden m_1 und m_2 und $|TM| = 1$. Dann sind in $A \in TM$ zwei Methoden a_1 und a_2 deklariert, sodass $m_1 \Rightarrow_{method} a_1$ und $m_2 \Rightarrow_{method} a_2$ oder $m_1 \Rightarrow_{method} a_2$ und $m_2 \Rightarrow_{method} a_1$. Dann wäre aufgrund von Theorem 1 $|TM'| = 2$. Somit ist in $B \in TM'$ eine Methode b_1 deklariert mit $m_1 \Rightarrow_{method} b_1$ oder $m_2 \Rightarrow_{method} b_1$. Somit sind vier Fälle zu unterscheiden:

1. Wenn gilt:

$$m_1 \Rightarrow_{method} a_1$$
$$m_2 \Rightarrow_{method} a_2$$
$$m_1 \Rightarrow_{method} b_1$$

   Dann gilt $\{A, B\} \in targetSets(C, 2)$ und $proxy_{struct}(R, \{A, B\}) \neq \emptyset$. Ein beispielhafter *struktureller Proxy* wäre:

```
proxy for R with [A, B]{
 R.m1 →TM.b1
 R.m2 →TM.a2
}
```

2. Wenn gilt:

$$m_1 \Rightarrow_{method} a_1$$
$$m_2 \Rightarrow_{method} a_2$$
$$m_2 \Rightarrow_{method} b_1$$

   Dann gilt $\{A, B\} \in targetSets(C, 2)$ und $proxy_{struct}(R, \{A, B\}) \neq \emptyset$. Ein beispielhafter *struktureller Proxy* wäre:

```
proxy for R with [A, B]{
 R.m1 →TM.a1
 R.m2 →TM.b1
}
```

3. Wenn gilt:

$$m_1 \Rightarrow_{method} a_2$$
$$m_2 \Rightarrow_{method} a_1$$
$$m_1 \Rightarrow_{method} b_1$$

Dann gilt $\{A, B\} \in targetSets(C, 2)$ und $proxy_{struct}(R, \{A, B\}) \neq \emptyset$. Ein beispielhafter *struktureller Proxy* wäre:

```
proxy for R with [A, B]{
 R.m1 →TM.b1
 R.m2 →TM.a1
}
```

4. Wenn gilt:

$$m_1 \Rightarrow_{method} a_2$$
$$m_2 \Rightarrow_{method} a_1$$
$$m_2 \Rightarrow_{method} b_1$$

Dann gilt $\{A, B\} \in targetSets(C, 2)$ und $proxy_{struct}(R, \{A, B\}) \neq \emptyset$. Ein beispielhafter *struktureller Proxy* wäre:

```
proxy for R with [A, B]{
 R.m1 →TM.a2
 R.m2 →TM.b1
}
```

Damit kann in jedem Fall ein *struktureller Proxy* generiert werden, in dem der *Target-Typ* aus *TM* ebenfalls als *Target-Typ* verwendet wird.

QED

Anhang G

DesCoSTests

Das Java-Projekt *DesCoSTests* bietet einen kleinen Einblick in die Verwendung der Module, die in Kapitel 4 vorgestellt wurden. Die Jar-Dateien, von denen das Projekt abhängt, befinden sich im Verzeichnis *./lib* innerhalb des Java-Projektes.

Das Projekt enthält zwei Source-Verzeichnisse:

- **./src**:
  Hier befindet sich lediglich eine Enum (`ComponentContainer`) wodurch ein Container mit mehreren Dienste abgebildet werden soll (siehe auch Abbildung G.1). Die Dienste können über die Methode `registerComponent` im Container hinterlegt werden. Über die Methode `getRegisteredComponentInterfaces` werden alle im Container hinterlegten Interfaces der Komponenten zurückgegeben, während über die Methoden `getOptComponent` ein `java.lang.Optional` zurückgegeben wird, in dem die Komponente zu dem übergebenen Interface enthalten ist.

- **./test**:
  Hier sind Testklassen hinterlegt, mit denen der *Explorationsprozess* für bestimmte *required Typen* gestartet werden kann. In dem Sub-Package `ma_scenarios` befinden sich drei Testklassen, die über das JUnit-Plugin in Eclipse gestartet werden können (JUnit4). Die Implementierung dieser Testklassen verwenden jeweils einen anderen *required Typ* (siehe jeweils die Methode `findCombined`). Dabei wird in diesen Methoden jeweils zuerst der Container gefüllt. Im Anschluss wird die *Config* für den *Finder* erzeugt (vgl. auch

Abschnitt 4.3). Und zuletzt wird der *Explorationsprozess* über den *Finder* mit dem jeweiligen *required Typ* gestartet. Aus der Log-Datei, die bei der Ausführung der Tests erzeugt wird, lässt sich ermittelt, wie viele *Proxies* in wie vielen Durchläufen erzeugt wurden. Die Log-Dateien sind nach der Ausführung im Verzeichnis *./log* zu finden.

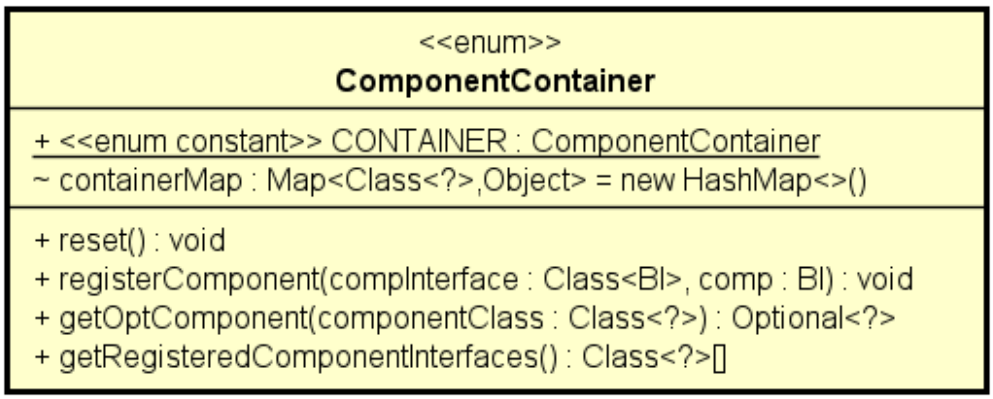


Abbildung G.1: ComponentContainer

Anhang H

Inhalt des beiliegenden Datenträgers

Inhalt	**Pfad auf dem Datenträger**
Masterthesis	Thesis/MT.pdf
Projekt *SignatureMatching*	Implementierung/Signature Matching
Projekt *ComponentTester*	Implementierung/ComponentTester
Projekt *DesiredComponentSourcerer*	Implementierung/DesiredComponentSourcerer
Projekt *DesCoSTests*	Implementierung/DesCoSTests
Referenzierte PDF-Dokumente	PDF/
Referenzierte Internetquellen	Internet/

Glossar

Abstrakter Syntaxbaum Ein abstrakter Syntaxbaum wird für die Darstellung der *abstrakten Syntax* eine Programms verwendet. Diese *abstrakte Syntax* ist eine Datenstruktur, welche die Kerninformationen eines Programms beschreibt, Sie enthält keinerlei Informationen über Details bzgl. der Notation (*konkrete Syntax*). (vgl. [VBK+13]). lix

Artefakt Ein Artefakt beschreibt in der Software-Entwicklung die Spezifikation einer physischen Informationseinheit als Ergebnis des Software-Entwicklungsprozesses oder dem Deployment bzw. der Ausführung eines Systems. In der UML Spezifikation 2.1.2 [Obj07] werden u.a. folgende konkrete Beispiele für Artefakte genannt:

- Dateien in denen Source Code enthalten ist
- Skripte
- Datenbanktabellen

Im Kontext dieser Arbeit sind insbesondere die Dateien, in denen Source Code enthalten ist, allgemein als Artefakt bezeichnet. 2, 67, 69

AST Abstrakter Syntaxbaum. 29, 34, 36, 39

Attributgrammatik Eine Attributgrammatik ist eine kontextfreie Grammatik, in der die Nonterminale Attribute enthalten können. Dadurch werden die Regeln für die Grammatik um semantische Regeln erweitert, die bestimmen, wie die Attribute der Nonterminale belegt werden müssen. (vgl. [Knu68]). 22

Bubble-Sort Unter dem Bubble-Sort-Verfahren versteht man ein Sortierverfahren, bei dem die Elemente einer Liste, die größer als ihr Nachfolger sind, mit ihrem Nachfolger vertauscht

werden. Sofern in der Liste keine Elemente mehr vertauscht werden, gilt die Liste als sortiert . 53

Default-Methode Eine Default-Methode ist eine Methode, die innerhalb eines Java-Interfaces implementiert wurde. Sofern diese Methode von den implementierenden Klassen dieses Interfaces nicht überschrieben werden, wird beim Aufruf dieser Methode die Implementierung der Default-Methode verwendet (vgl. [GJS+15b]) . 108

Dependency Injection Durch Dependency Injection soll die Entkopplung konkreter Implementierungen erreicht werden. Angenommen eine Klasse A hängt von einer Klasse B ab, da sie bspw. ein Attribut vom Typ B enthält. Um die Entkopplung dieser Klassen zu erreichen, wird ein Interface IB geschaffen, welches von der Klasse B implementiert wird. Die Abhängigkeit der zwischen den Klassen A und B wird dann dadurch aufgelöst, dass die Klasse A lediglich vom Interface BI. Dieses Vorgehen entspringt dem Paradigma *Inversion of Control* (vgl. [ES13]). Zur Laufzeit muss jedoch dafür gesorgt werden, dass die in einem Objekt der Klasse A ein Objekt injiziert wird, dessen Klasse das Interface IB implementiert - vorzugsweise also ein Objekt der Klasse B. Es gibt unterschiedliche Vorgehensweisen, um diese Injektion vorzunehmen (vgl. [Fow04]). 8

Downcast Ein Downcast beschreibt eine Typumwandlung eines Typs T in einen von T abgeleiteten Typen T' ($T > T'$). Zur Laufzeit ergibt sich dabei das Problem, dass in T' eine Methode m deklariert wurde, die jedoch nicht in T bekannt ist. Sofern also ein Objekt vom Typ T dort verwendet wird, wo ein Objekt vom Typ T' erwartet wird, führt der Aufruf der Methode m zwangsweise zu einem Fehler, da die Methode in dem verwendeten Objekt unbekannt ist. 29, 107

Engine Eine Engine beschreibt eine Software oder einen Teil einer Software, der für eine spezifische Aufgabe verantwortlich ist (vgl. [PCM]). Die Aufgabe, die die in der Arbeit beschriebenen Source Engines erfüllen, wird in Abschnitt 2.1 beschrieben. 2, 3, 6–8, 112

Heuristik Als Heuristik werden in dieser Arbeit Verfahren bezeichnet, durch die die Lösung eines Problems beschleunigt werden kann, indem neu gewonnene Erkenntnisse bei der Lösungsfindung berücksichtigt werden. 49, 50, 53, 55, 57, 58, 73, 75, 76, 78, 80, 82, 85, 87, 88, 90–97, 99–102, 105, 107, 111, 112

Interface Ein Interface hat im Allgemeinen eine Übersetzungs- oder Vermittlungsfunktion zwischen gekoppelten Systemen (vgl. [Hal94]). Die Bedeutung des Begriffs in dieser Arbeit bezieht sich jedoch auf den Kontext der objektorientierten Programmierung. In diesem Zusammenhang beschreibt ein Interface die Methoden, die in den Klassen, die dieses Interface erfüllen, vorhanden sein müssen. i, xv, xvii, xxiv, lx, 1, 2, 6–11, 13, 62–64, 68, 69, 71–73, 108, 109, 112

JNDI Java Naming and Directory Interface (JNDI) ist ein API, welches den Entwickler*innen bei der Verwendung der Programmiersprache Java erlaubt, Referenzen von Objekten anhand eines Namens abzulegen und diese Objekte somit auch über den jeweiligen Namen zu adressieren. (vgl. [Ora21]). 8

Komplexität Komplexität wird in dieser Arbeit als Oberbegriff für Speicherkomplexität und Zeitkomplexität verwendet . 102

Komponente Eine Komponente beschreibt in der Softwarearchitektur im Allgemeinen ein Teil eines Softwaresystems. Die Definition dieses Begriffs wird in speziellen Frameworks weiter spezifiziert. Bezogen auf das in der Arbeit verwendete EJB-Framework, werden bspw. die Beans als Komponenten betrachtet (vgl. [DeM05]). iv, 1–3, 103–105, 112

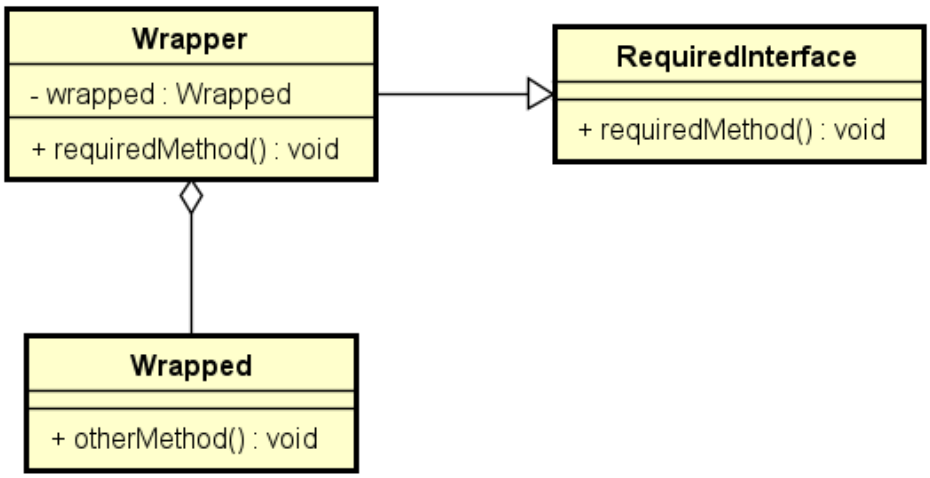
Modul Ein Modul ist in der Software-Entwicklung ein Teil eines Softwaresystems, der eine funktional geschlossene Einheit darstellt und einen bestimmten dienst bereitstellt (vgl. [LS18]). In dieser Arbeit werden einzelne Java-Projekte als Module bezeichnet. 61, 62, 67–70, 72, 73, 112

Speicherkomplexität Unter Speicherkomplexität versteht man i.d.R. den Speicherbedarf, den ein Algorithmus benötigt, um ein bestimmtes Problem zu lösen. 101

Substitutionsprinzip Das Listkov'sche Substitutionsprinzipt ist ein Entwurfsprinzip der objektorientierten Programmierung. Es besagt, dass ein Unterklasse überall dort einsetzbar sein muss, wo die Oberklasse verlangt wird (vgl. [ES13]). 26, 27

Wrapper-Typ Ein Wrapper-Typ wird in der Programmierung auch als Hüllenklasse oder Adapter bezeichnet und entstammen dem *Adapter-Muster* - einem Strukturmuster aus

[GHJV14]. Eine solche Hüllenklasse beschreibt eine Klasse, mit der die Schnittstelle einer anderen Klasse angepasst werden kann. So ist es einem Klienten, welcher eine bestimmte Schnittstelle erwartet, möglich über die Hüllenklassen mit einer Klasse zusammenzuarbeiten, die die erwartete Schnittstelle nicht erfüllt. Die in dieser Arbeit beschriebenen Wrapper-Typen setzen dabei auf die Delegation der Schnittstellen-Anfragen. Das folgende Klassendiagramm zeigt die grundlegende Struktur solcher Wrapper-Typen . 17, 111



Zeitkomplexität Unter Zeitkomplexität versteht man i.d.R. den Bedarf an Rechenschritten, den ein Algorithmus benötigt, um ein bestimmtes Problem zu lösen. 101

Literaturverzeichnis

[Ber19] Berlin, Sam: *cglib 3.3.0*. `https://github.com/cglib/cglib/wiki`, 2019. [Online; letzter Zugriff 16.11.2021].

[BNL+06] Bajracharya, Sushil, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi Cristina Lopes: *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. OOPSLA '06, 681–682, New York, NY, USA, 2006. Association for Computing Machinery.

[DeM05] DeMichiel, Linda: *EJB Core Contracts and Requirements*. `https://download.oracle.com/otndocs/jcp/ejb-3_0-pr-spec-oth-JSpec/`, 2005. [Online; letzter Zugriff 16.11.2021].

[ES13] Eilebrecht, Karl Gernot Starke: *Patterns kompakt*. Springer Verlag Berlin Heidelberg, 2013.

[Fow04] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. `https://martinfowler.com/articles/injection.html`, 2004. [Online; letzter Zugriff 16.11.2021].

[GHJV14] Gamma, Erich, Richard Helm, Ralph Johnson John Vlissides: *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp, 2014.

[GJS+15a] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha Alex Buckley: *The Java Language Specification - Java SE 8 Edition: Chapter 5. Conversions and Contexts*. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html`, 2015. [Online; letzter Zugriff 16.11.2021].

[GJS+15b] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha Alex Buckley: *The Java Language Specification - Java SE 8 Edition: Chapter 9.4. Method Declarations.* https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.4, 2015. [Online; letzter Zugriff 16.11.2021].

[Hal94] Halbach, Wulf R.: *Interfaces: medien- und kommunikationstheoretische Elemente einer Interface-Theorie.* Wilhelm Fink Verlag, München, 1994.

[HJ13] Hummel, Oliver Werner Janjic: *Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse*, 227–250. Springer New York, New York, NY, 2013.

[Hum08] Hummel, Oliver: *Semantic Component Retrieval in Software Engineering.* , April 2008.

[inv20] *Java Plattform - Interface InvocationHandler.* https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/InvocationHandler.html, 2020. [Online; letzter Zugriff 16.11.2021].

[IT 21] IT Administrator: *Verfügbarkeit.* https://www.it-administrator.de/lexikon/verfuegbarkeit.html, 2021. [Online; letzter Zugriff 16.11.2021].

[jun21a] *JUnit 4.* https://junit.org/junit4/, 2021. [Online; letzter Zugriff 16.11.2021].

[jun21b] *JUnit 4.13.2 API.* https://junit.org/junit4/javadoc/latest/index.html, 2021. [Online; letzter Zugriff 16.11.2021].

[KA15] Kessel, Marcus Colin Atkinson: *Measuring the Superfluous Functionality in Software Components. Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE '15, 11–20, New York, NY, USA, 2015. Association for Computing Machinery.

[KA16] Kessel, Marcus Colin Atkinson: *Ranking software components for reuse based on non-functional properties.* Inf. Syst. Frontiers, 18(5):825–853, 2016.

[KA18] Kessel, Marcus Colin Atkinson: *Integrating reuse into the rapid, continuous software engineering cycle through test-driven search*. Bosch, Jan, Brian Fitzgerald, Michael Goedicke, Helena Holmström Olsson, Marco Konersmann Stephan Krusche (): *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering, RCoSE@ICSE 2018, Gothenburg, Sweden, May 29, 2018*, 8–11. ACM, 2018.

[Knu68] Knuth, Donald E.: *Semantics of Context-Free Languages*. *In Mathematical Systems Theory*, 127–145, 1968.

[Kru92] Krueger, Charles W.: *Software Reuse*. ACM Comput. Surv., 24(2):131–183, 1992.

[LLBO07] Lazzarini Lemos, Otavio Augusto, Sushil Krishna Bajracharya Joel Ossher: *CodeGenie: A Tool for Test-Driven Source Code Search*. *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, 917–918, New York, NY, USA, 2007. Association for Computing Machinery.

[LS18] Lackes, Richard Markus Siepermann: *Modul*. `https://wirtschaftslexikon.gabler.de/definition/modul-40077/version-263472`, 2018. [Online; letzter Zugriff 16.11.2021].

[Obj07] Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. `https://www.omg.org/spec/UML/2.1.2/Superstructure/PDF`, 2007. [Online; letzter Zugriff 16.11.2021].

[obj21] *Objenesis*. `http://objenesis.org/index.html`, 2021. [Online; letzter Zugriff 16.11.2021].

[Ora21] Oracle: *Lesson: Overview of JNDI*. `https://docs.oracle.com/javase/tutorial/jndi/overview/index.html`, 2014-2021. [Online; letzter Zugriff 16.11.2021].

[PCM] PCMag Encyclopedia: *engine*. `https://www.pcmag.com/encyclopedia/term/engine`. [Online; letzter Zugriff 16.11.2021].

[SED16] Stolee, Kathryn T., Sebastian Elbaum Matthew B. Dwyer: *Code search with input/output queries: Generalizing, ranking, and assessment.* Journal of Systems and Software, 116:35–48, 2016.

[VBK+13] Völter, Markus, Sebastian Benz, Lennart Kats, Mats Helander, Eelco Visser Guido Wachsmuth: *DSL Engineering.* CreateSpace Independent Publishing Platform, 2013.

[ZW95] Zaremski, Amy Moormann Jeannette M. Wing: *Signature Matching: A Tool for Using Software Libraries.* ACM Trans. Softw. Eng. Methodol., 4(2):146–170, 1995.