Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Neubrandenburg, 13. Februar 2015

Niels Gundermann

Inhaltsverzeichnis

Abbildungsverzeichnis

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 bzw. 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 Abb. 1).

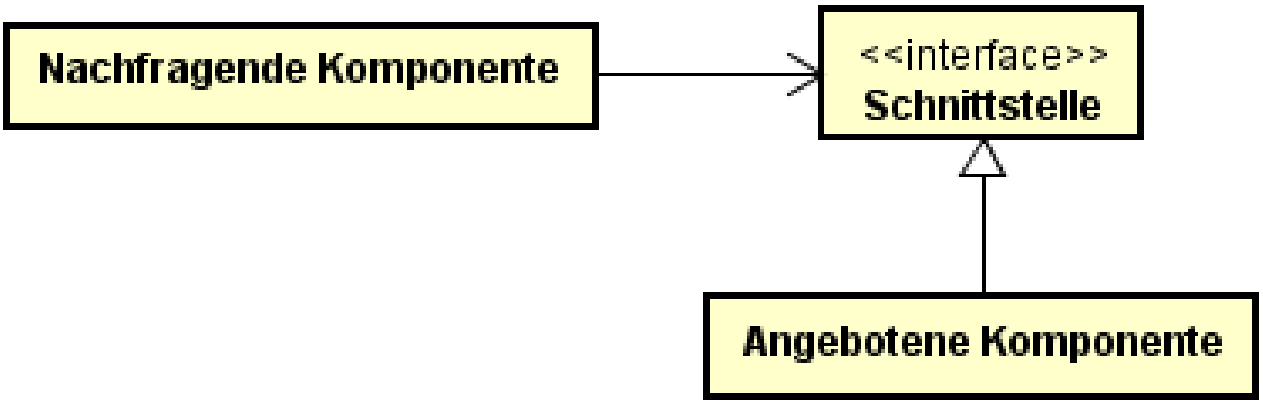

Abbildung 1: Abhängigkeiten von nachfragenden und angebotenen Komponenten

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 es nicht möglich ist, dass eine nachfragende Komponente einfach selbst spezifizieren kann, welche Informationen oder Services sie erwartet, wodurch auf der Basis dieser Spezifikation eine passende anbietende Komponente gefunden werden kann.

2 Verwandte Arbeiten

Ein solcher Ansatz wurde bereits in [BNL+06] von Bajaracharya et al. verfolgt. Bajaracharya et al. entwickelten eine Source Engine namens Sourcerer, welche Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens Code-Genie entwickelt, welches einem Softwareentwickler die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde offenbar erstmals der Begriff der Test-Driven Code Search (TDCS) etabliert. Parallel dazu wurde in Verbindung mit der Dissertation Oliver Hummel [Hum08] ebenfalls eine Weiterentwicklung von Sourcerer veröffentlicht, welche unter dem Namen Merobase bekannt ist. Das Verfahren, auf dem beide Search Engines grundlegend aufbauen wird, wie bereits angedeutet, als TDCS bezeichnet. Dieses Verfahren beruht grundlegend darauf, dass der Entwickler Testfälle spezifiziert, die im Anschluss verwendet werden, um relevanten Source Code aus einem Repository hinsichtlich dieser Testfälle zu evaluieren. Damit kann das jeweilige Tool dem Entwickler Vorschläge für die Wiederverwendung bestehenden Codes unterbreiten.

Bezogen auf die am Ende des vorherigen Abschnitts formulierte Überlegung ermöglichen die genannten Source Engines, das Internet nach bestehendem Source Code zu durchsuchen und damit bereits bestehende Implementierungen für eine nachfragende Komponente zu ermitteln.

3 Gegenstand dieser Arbeit

In dieser Arbeit soll jedoch nicht das gesamte Internet als Quelle oder Repository für die Codesuche dienen. Vielmehr wird der Suchbereich weiter eingeschränkt.

Es wird von einem System ausgegangen, in dem ein EJB-Container zur Verfügung steht. Die Suche soll sich auf die Menge der angemeldeten Bean-Implementierungen beschränken. Die angemeldeten Bean-Implementierungen stellen damit die Menge der angebotenen Komponenten dar. Dabei wird eine angebotenen Komponente als Kombination eines Interfaces, welches die Schnittstelle für die Aufrufer definiert, und einer Implementierung des Interfaces. Das Interfaces einer angebotenen Komponenten wird im Folgenden auch als angebotenes Interfaces bezeichnet. Die Beans werden bspw. als Provider für Informationen oder im weitesten Sinne auch als Services verwenden, die von unterschiedlichen Komponenten des Systems verwendet werden. Bei der Entwicklung bzw. Weiterentwicklung einer Komponente kann es zu folgenden Szenario kommen, welches durch die Erfüllung der unten aufgeführten Annahmen charakterisiert wird:

- Es werden Informationen und Services benötigt, bei denen der Entwickler davon ausgehen kann, dass es innerhalb des Systems angebotene Komponenten gibt, die diese Informationen liefern können bzw. die Services erfüllen.
- Der Entwickler weiß nicht, über welche konkreten angebotenen Komponenten er die Informationen abfragen bzw. die Services in Anspruch nehmen kann.

3.1 Funktionale Anforderungen

In dieser Arbeit soll ein Konzept entwickelt werden, welches dem entwickler ermöglicht die Erwartungen an die angebotenen Komponenten zu spezifizieren. Darauf aufbauend soll ein Algorithmus vorgeschlagen werden, welcher die angebotenen Komponenten zur Laufzeit hinsichtlich der spezifizierten Erwartungen des Entwicklers evaluiert und eine Auswahl derer trifft, die diese Erwartungen erfüllen. Da die Evaluation zur Laufzeit durchgeführt wird, kann der Entwickler anders als bei den oben genannten Arbeiten nicht aus einer Liste von Vorschlägen auswählen, welche der evaluierten Komponenten letztendlich verwendet werden soll. Diese Entscheidung ist durch den Algorithmus zu treffen.

3.2 Nichtfunktionale Anforderungen

Aufgrund bestimmter Konfigurationen des Gesamtsystem gibt es folgende weitere nichtfunktionale Anforderungen:

- Die Suche muss innerhalb des Transaktionstimeouts von 5 Minuten zu einem Ergebnis führen.
- Die Suche soll hinsichtlich der Besonderheiten des System, in dem sie verwendet wird, angepasst werden können. (Bspw. bei der Verwendung bestimmter Typen, deren Fachlogik bei der Suche nicht untergraben werden darf.)
- Bei einem Fehlschlag der Suche, sollen dem Entwickler Informationen zur Verfügung gestellt werden, die eine zielgerichtete Anpassung seiner spezifizierten Erwartungen erlauben.

4 Entwurf des Explorationsalgorithmus

4.1 Voraussetzungen

Die erste Voraussetzung bezieht sich auf die Spezifikation der Erwartungen einer nachfragenden Komponente. Diese soll aus zwei Teilen bestehen.

Der erste Teil soll die Struktur der erwarteten Informationen bzw. Services beschreiben. Der Entwickler soll hierzu die Schnittstelle, die von ihm erwartet wird in der Form beschreiben, wie es in dem vorliegenden System üblich ist. In dem konkreten System, von dem in dieser Arbeit ausgegangen wird, handelt es sich dabei um Java-Interfaces. Die Struktur der erwarteten Informationen bzw. Services wird demnach innerhalb der nachfragenden Komponente durch ein Interface dargestellt, in dem die erwarteten Methoden, die innerhalb der nachfragenden Komponente verwendet werden sollen, deklariert wurden. Ein solches Interface wird im Folgenden auch als erwartetes Interface bezeichnet.

Der zweite Teil besteht aus einer Menge von Testfällen, durch die ein Teil der erwarteten Semantik spezifiziert wird. Hierzu können Testfälle in Methoden mehrerer Klassen implementiert werden. Zur Referenzierung der Testklassen wird eine Annotation @QueryTypeTestReference im erwarteten Interface verwendet. Dort können über den Parameter testClasses mehrere Testklassen angegeben werden. Die Testklassen müssen über den Default-Konstruktor verfügen. Innerhalb der Testklassen werden die Testmethoden mit der Annotation @QueryTest markiert. Weiterhin ist es notwendig, die zu testende Instanz zur Laufzeit in ein Objekt einer Testklasse zu injizieren. Dies erfolgt durch Setter-Injection. Aus diesem Grund muss in jeder Testklasse ein Setter für ein Objekt vom Typ des erwarteten Interfaces implementiert werden und mit der Annotation @QueryTypeInstanceSetter markiert werden.

Listing 1 zeigt ein Beispiel für ein erwartetes Interface, welches eine Testklasse referenziert. Listing 2 hingegen zeigt diese referenzierte Testklasse mit den bereits erwähnten Annotationen für den Setter des zu testenden Objektes und den Testmethoden.

Listing 1: Erwartetes Interface IntubatingFireFighter

```
1 @QueryTypeTestReference( testClasses = IntubatingFireFighterTest.class )
2 public interface IntubatingFireFighter {
3
4   public void intubate( AccidentParticipant injured );
5
6   public void extinguishFire( Fire fire );
7 }
```

Listing 2: Testklasse des erwarteten Interfaces IntubatingFireFighter

```
1 public class IntubatingFireFighterTest {
2
3   private IntubatingFireFighter intubatingFireFighter;
4
5   @QueryTypeInstanceSetter
6   public void setProvider( IntubatingFireFighter intubatingFireFighter ) {
7     this.intubatingFireFighter = intubatingFireFighter;
8   }
9
10  @QueryTypeTest
11  public void free() {
12    Fire fire = new Fire();
13    intubatingFireFighter.extinguishFire( fire );
14    assertFalse( fire.isActive() );
15  }
16
17  @QueryTypeTest
18  public void intubate() {
19    Collection<Suffer> suffer = Arrays.asList( Suffer.BREATH_PROBLEMS );
20    AccidentParticipant patient = new AccidentParticipant( suffer );
21    intubatingFireFighter.intubate( patient );
22    assertTrue( patient.isStabilized() );
23  }
24 }
```

Die zweite Voraussetzung betrifft den Zugang zu den bestehenden angebotenen Schnittstellen und deren Implementierungen in dem bestehenden System. Um in der Menge aller angebotenen Komponenten eine passende Komponente finden zu können, muss diese Menge bekannt sein oder ermittelt werden können. Wie oben beschrieben, wird in dieser Arbeit von einem System ausgegangen, in dem die angebotenen Komponenten als Java Enterprise Beans umgesetzt wurden. So wird dementsprechend eine Möglichkeit geschaffen, sämtliche der angemeldeten JNDI-Namen und die dazugehörigen Bean-Interfaces abzufragen. Die Abfrage der angemeldeten Bean-Implementierungen zu einem JNDI-Namen ist durch den EJB-Container bei Vorliegen des entsprechenden Interfaces und des JNDI-Namens bereits gegeben.

4.2 Explorationskomponente

Mit diesen Voraussetzungen kann eine Komponente entwickelt werden, der die Erwartungen der nachfragenden Komponente mit den bestehenden Funktionalitäten der angebotenen Komponenten zusammenbringt. In Abb. 2 ist dies als Explorationskomponente dargestellt. Die Abhängigkeiten zu der nachfragenden und den angebotenen Komponenten ist nicht direkt vorhanden, da sie lediglich durch reflexive Aufrufe zur Laufzeit zustande kommen.

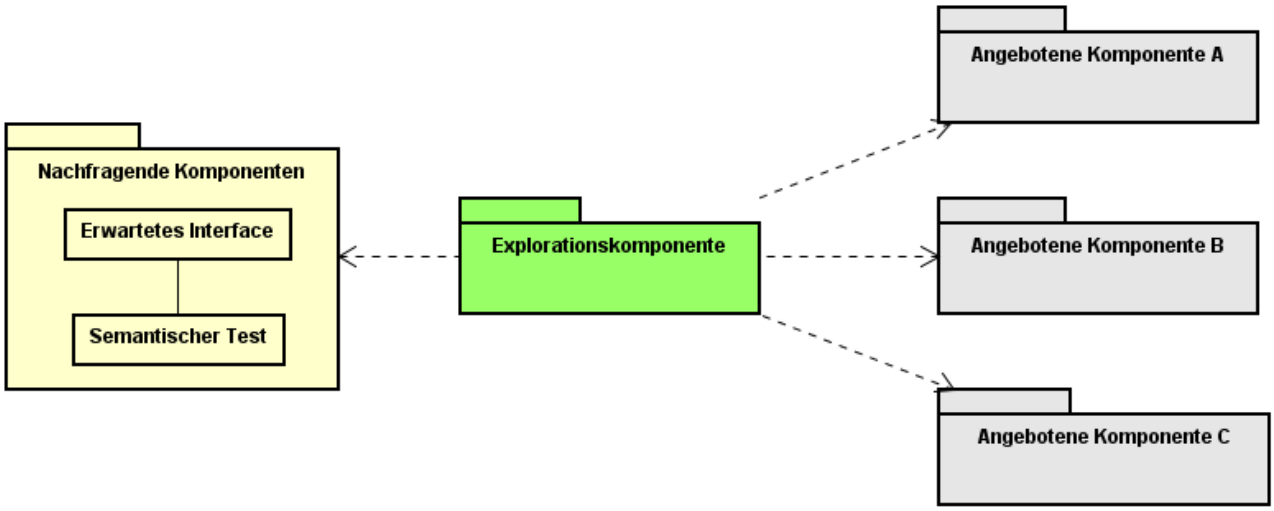

Abbildung 2: Allgemeiner Aufbau des System mit der Explorationskomponente

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

Zu erwähnen ist noch, dass ein die in der nachfragenden Komponente spezifizierten Erwartungen mitunter nur durch eine Kombination von angebotenen Komponenten erfüllt werden können. Aus diesem Grund wird innerhalb der Explorationskomponente eine so genannte passenden Komponente erzeugt, in der das Zusammenspiel einer solchen Kombination von angebotenen Komponenten verwaltet wird. Ein solches Szenario ist Abb. 3 zu entnehmen.

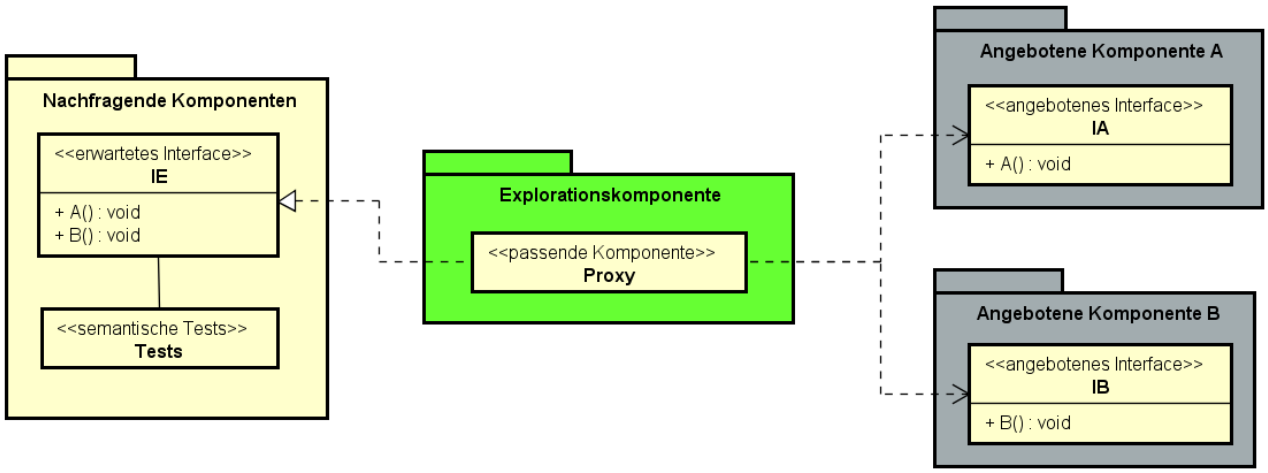

Abbildung 3: Kombination von angebotenen Komponenten

Die Suche nach einer passenden Komponente innerhalb der Explorationskomponente in zwei Schritten erfolgen. Im ersten Schritt werden die angebotenen Interfaces hinsichtlich ihrer Struktur mit dem erwarteten Interface abgeglichen. Im zweiten Schritt werden die Ergebnisse aus dem ersten Schritt hinsichtlich der semantischen Tests überprüft. Dieser mehrstufige Ansatz baut auf der Arbeit von Hummel [Hum08] auf.

4.2.1 1. Stufe - Strukturelle Übereinstimmung

Wie in [Hum08] wird in der ersten Stufe der Suche versucht die angebotenen Interfaces herauszusuchen, die strukturell mit dem erwarteten Interface übereinstimmen. Zu diesem Zweck werden Type-Matcher verwendet, durch die festgestellt werden kann, ob sich ein Typ in einen anderen Typ konvertieren lässt. Die Konvertierung erfolgt auf Objekt-Ebene über Proxies, die ihre Methodenaufrufe delegieren.

So wird bspw. bei der Konvertierung eines Objektes von TypA (ObjA) in ein Objekt von TypB (ObjB) ein Proxy-Objekt für TypB erzeugt (ProxyB), welcher die Methodenaufrufe an ObjA delegiert (vgl. auch Abb. 3).

Hummel hatte hierzu bereits auf einige Matcher von Moormann Zaremski [ZW95] zurückgegriffen, die in dieser Arbeit ebenfalls zum Einsatz kommen. Die Definitionen der Matcher beziehen sich auf die Programmiersprache Java, wodurch grundlegend von einer nominalen Typkonformität ausgegangen wird.

Notation zur Beschreibung der Matcher

Die Übereinstimmung bzw. das Matching zweier Typen A und B über einen Matcher M wird in dieser Arbeit mit $A \equiv_M B$ notiert. Weiterhin wird die Identität zweier Typen mit $A = B$ beschrieben. Eine Vererbungshierarchie in der A von B erbt wird mit $A \leq B$ beschrieben. Weiterhin ist die Adressierung von Attributen innerhalb eines Typs notwendig. Für die Adressierung der Attributs a im Typ A wird $A\#a$ geschrieben.

Die Konvertierung eines Typs A in einen Typ B wird, wie bereits erwähnt, auf technischer Ebene über Proxies umgesetzt. Von daher kann die Beschreibung des Konvertierungsverfahrens eines Matchers auf die Beschreibung der Delegation einzelner Methoden beschränkt werden. Hierfür wird folgende Notation verwendet:

Eine Methode m enthält einen Rückgabetyp rt und eine Menge von Parametertypen rt. Die Menge der Parametertypen wird zur besseren Lesbarkeit auf einen Parametertyp beschränkt. Der Aufruf einer Methode m mit dem Rückgabetyp rt und dem Parametertyp pt eines Typs A wird mit $A.m(pt) : rt$ notiert. Sofern die Konvertierung keinen Einfluss auf den Rückgabetyp oder die Parametertypen hat, wird dies verkürzt mit $A.m$ beschrieben.

In der Notationen sind Typen, konkrete Objekte bestimmter Typen und Methoden syntaktisch austauschbar. Eine logische Verknüpfung der einzelnen Elemente der Sprache im Sinne der Quantoren und Junktoren der Prädikatenlogik 1. Stufe ist ebenfalls möglich.

Die Delegation von Methodenaufrufen auf einem Objekt wird mit dem Operator beschrieben. Für eine Delegation des Aufrufs einer Methode m auf dem Objekt a, welcher an ein Objekt b und dessen Methode n delegiert wird, schreibt man $a.m \Rightarrow b.n$. Ferner ist hierbei zwischen einem Source- und einem Target-Objekt zu unterscheiden. Das Source-Objekt befindet sich links vom Operator ($\Rightarrow$). Auf diesem Objekt findet der Methodenaufruf statt. Das Target-Objekt befindet sich auf der rechten Seite des Operators. Dieses stellt das Ziel der Delegation dar. Da bei der Delegation mitunter weitere Matcher zur Anwendung kommen müssen, wird hierfür ebenfalls eine Notation benötigt. Daher soll die Konvertierung eines Objektes a über einen Matcher M wird mit $(M)a$ beschrieben.

ExactTypeMatcher

Übereinstimmung (ExactTypeMatcher)

$$A \equiv_{exact} B \text{ wenn } A = B$$

Konvertierung (ExactTypeMatcher)

Sei m eine Methode des Typs A (bzw. B) und a und b jeweils ein Objekt vom Typ A (bzw. B).

$$a.m \Rightarrow b.m$$

GenSpecTypeMatcher

Übereinstimmung (GenSpecTypeMatcher)

$$A \equiv_{genspec} B \text{ wenn } A \leq B \vee B \leq A$$

Konvertierung (GenSpecTypeMatcher)

Sei m eine Methode des Typs A (bzw. B) und a und b jeweils ein Objekt vom Typ A (bzw. B).

$$\text{Wenn } B \leq A \text{ dann } a.m \Rightarrow b.m$$

$$\text{Wenn } A \leq B \wedge \exists B.n : A.m \leq B.n \text{ dann } a.m \Rightarrow b.n$$

Der erste Fall stellt hierbei nichts anderes als einen Upcast des Typs B auf den Typ A dar. Der zweite Fall hingegen stellt einen Downcast des Typs B auf den Typ A dar. Hierbei ist zu beachten, dass ein Downcast eine gewisse Unsicherheit bzgl. der Ausführung birgt. So können beim Aufruf einer Methode auf einem Objekt a von Typ A nur solche Methoden delegiert werden, die von B an A vererbt wurden. Ein Aufruf einer Methode auf a, die nicht von B an A vererbt wurde, führt zu einem Laufzeitfehler.

WrappedTypeMatcher

Die bisherigen Type-Matcher sind in der Lade das Matching für zwei Typen festzustellen, ohne dafür Rücksicht auf deren innere Struktur nehmen zu müssen. Dies ist für identische oder hierarchisch organisierte Typen auch nicht notwendig. Es ist jedoch auch denkbar, dass die sich beiden Typen anderweitig assoziieren lassen. Ein Beispiel dafür wäre Boxed- bzw., noch allgemeiner gefasst, Wrapper-Typen. Abb. 4 stellt ein einfaches Szenario für einen Typ A und einen dazu passenden Wrapper-Typen W dar.

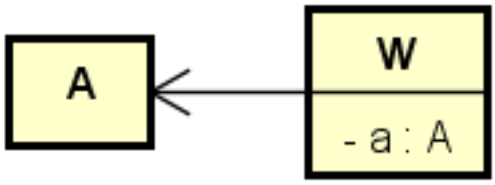

Abbildung 4: Beispiel Wrapper-Typ allgemein

Ein Anwendungsfall, in dem eine solche Konstellation der Typen auftreten könnte, ist Abb. 5 zu entnehmen. Hier wurde in dem erwarteten Interface Directory eine Methode add deklariert, die einen Parameter vom Typ String erwartet - in diesem Fall bspw. der für das betrachtete System eindeutige Name einer Person. Auf der Angebotsseite steht jedoch nur ein angebotenes Interface AdressBook zur Verfügung, welches den Typ Person erwartet. Der Typ Person enthält jedoch ein Attribut vom Typ String, welches letztendlich den für das betrachteten System eindeutigen Namen der Person darstellt. So kann es zielführend, dass die hier angebotene Komponente unter der Annahme, dass der Parameter aus dem erwarteten Interface das Attribut name aus dem Typ Person darstellt, weiter zu untersuchen. Vor der Delegation der Methode könnte hierbei durch die Explorationskomponente ein Proxy für den Typ Person (Person-Proxy) erzeugt werden, der sich genauso verhält wie der ursprüngliche Typ. Allerdings würde das Objekt, welches in der Methode des erwarteten Interfaces übergeben wurde, in das Attribut name des Person-Proxys injiziert werden.

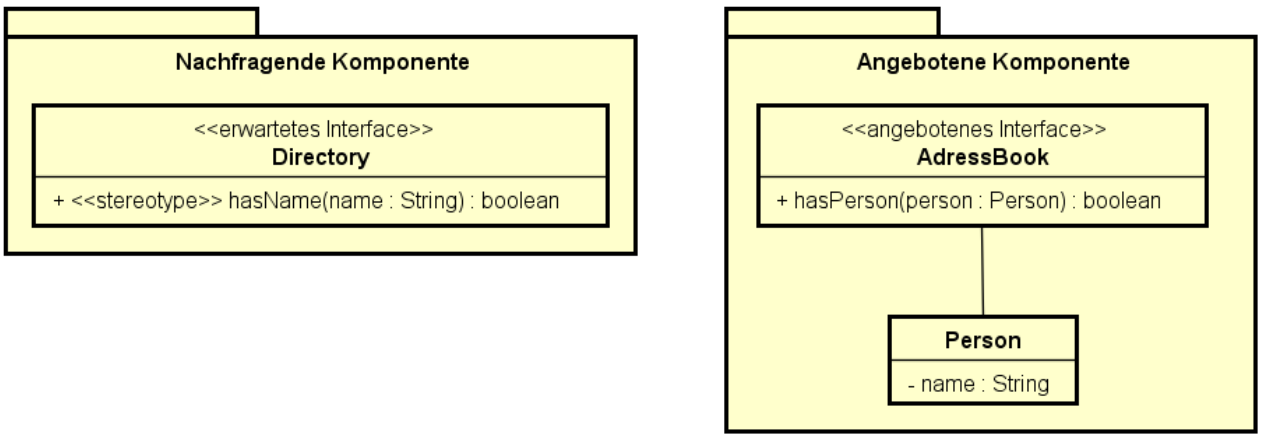

Abbildung 5: Beispiel Wrapper-Typ AdressBook

Aus diesem Grund wird ein WrappedTypeMatcher wie folgt beschrieben.

Übereinstimmung (WrappedTypeMatcher)

$$A \equiv_{wrapped} B \text{ wenn } \exists A\#attr : attr \equiv_M B \vee \exists B\#attr : attr \equiv_M A$$

Wie an dieser Beschreibung zu erkennen ist, wird im WrappedTypeMatcher wiederum die Überein-

stimmung von Typen gefordert, die ebenfalls durch die anderen Matcher festgestellt werden kann.

Konvertierung (WrappedTypeMatcher)

Sei m eine Methode des Typs A und a ein Objekt vom Typ A.

Weiterhin sei n eine Methode des Typs B und b ein Objekt vom Typ B.

$$\text{Wenn } \exists B\#attr : attr \equiv_M A \text{ dann } a.m \Rightarrow (M)B\#attr.m$$

$$\text{Wenn } \exists A\#attr : attr \equiv_M B \text{ dann } a.m \Rightarrow a.m$$

Hervorzuheben ist, dass bei einem Methodenaufruf auf dem Wrapper-Typ, keine Delegation vorgenommen wird.

StructuralTypeMatcher

Die bisher beschriebenen Type-Matcher ermöglichen, lediglich eine 1:1-Beziehung zwischen erwartetem und angebotenem Interface. Bei der Suche nach einer passenden Komponente muss jedoch in Betracht gezogen werden, dass diese aus einer Menge von angebotenen Komponenten besteht. Beispielsweise wäre es vorstellbar, dass ein erwartetes Interface zwei Methoden deklariert, die aufgrund der strukturellen Eigenschaften der Methoden und den spezifizierten semantischen Tests auf der Angebotsseite in zwei unterschiedlichen Komponenten zu suchen sind. Abb. 3 verdeutlicht dieses Szenario wobei bei der Ausführung der Methode A die Methode der angebotenen Komponente AngA und bei der Ausführung der Methode B die Methode der angebotenen Komponente AngB ausgeführt werden müsste.

Die Möglichkeit eines solchen Szenarios bedingt, dass die erwarteten und angebotenen Interfaces je Methode bzgl. der strukturellen Übereinstimmung untersucht werden. Das erfordert einen weiteren Matcher, der die o.g. Matcher je erwartete Methode - sprich je Methode des erwarteten Interfaces - und angebotenen Methode - sprich je Methode der angebotenen Interfaces - untersucht. Zu diesem Zweck wird der StructuralTypeMatcher ergänzt.

Übereinstimmung (StructuralTypeMatcher)

$$A \equiv_{struct} B \text{ wenn}$$

$$\exists(A.m(MP) : MR) : \exists(B.n(NP) : NR) : MP \equiv_P NP \wedge NR \equiv_R MR$$

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

Konvertierung (StructuralTypeMatcher)

Sei m eine Methode des Typs A und a ein Objekt vom Typ A.

Der Rückgabetyp von m sei MR und mr ein Objekt dessen.

Zudem sei MP der Parametertyp von m und mp ein Objekt von MP.

Weiterhin sei n eine Methode des Typs B und b ein Objekt vom Typ B.
Der Rückgabetyp von n sei NR und nr ein Objekt dessen.
Zudem sei NP der Parametertyp von n und np ein Objekt von NP.

$$a.m(mp) : mr \Rightarrow b.n((P)np) : (R)nr$$

Typ-Konvertierungsvariante

Die Konvertierung der einzelnen Type-Matcher liefert eine Menge von so genannten Typ-Konvertierungsvariante
Eine Typ-Konvertierungsvariante beschreibt eine Möglichkeit, wie ein Typ in einen anderen konvertiert werden kann. Zu diesem Zweck enthält eine Typ-Konvertierungsvariante zwei Arten von Information:

1. Objekterzeugungsrelevante Informationen
2. Methodendelegationsrelevante Informationen

Typ-Konvertierungsvarianten werden von einem konkreten Typ-Matcher für jede mögliche Form der Übereinstimmung erzeugt. Im speziellen Fall des ExactTypeMatchers und des SpecGenTypeMatcher kann, wenn überhaupt, nur eine Typ-Konvertierungsvariante erzeugt werden. Da die anderen Typ-Matcher mit internen Type-Matchern arbeiten, sind von diesen mehrere Typ-Konvertierungsvarianten zu erwarten.

Die objekterzeugungsrelevanten Informationen sorgen dafür, dass das Proxy-Objekt zum Source-Typ korrekt erzeugt werden kann.

Die methodendelegationsrelevanten Informationen sorgen dafür, dass das Rückgabe-Objekt und die Parameter-Objekte beim Methodenaufruf korrekt konvertiert werden und dass der Aufruf an die richtige Methode des Target-Typs delegiert wird. Daher wird eine methodendelegationsrelevante Information auch als Methoden-Konvertierungsvariante bezeichnet.

In Abb. 6 ist dieser Zusammenhang für ein angebotenes Interface AIv und einem erwarteten Interface EI, welche jeweils zwei Methoden enthalten (AM * bzw. EM *), skizziert. Hier wird angenommen, dass jeder der angebotenen Methoden strukturell zu jeder der erwarteten Methoden passen würde. Dementsprechend enthält die Typ-Konvertierungsvariante (TKV) insgesamt 4 Methoden-Konvertierungsvarianten, wovon jede eine Konvertierung entlang der eingezeichneten Pfeile ermöglicht.

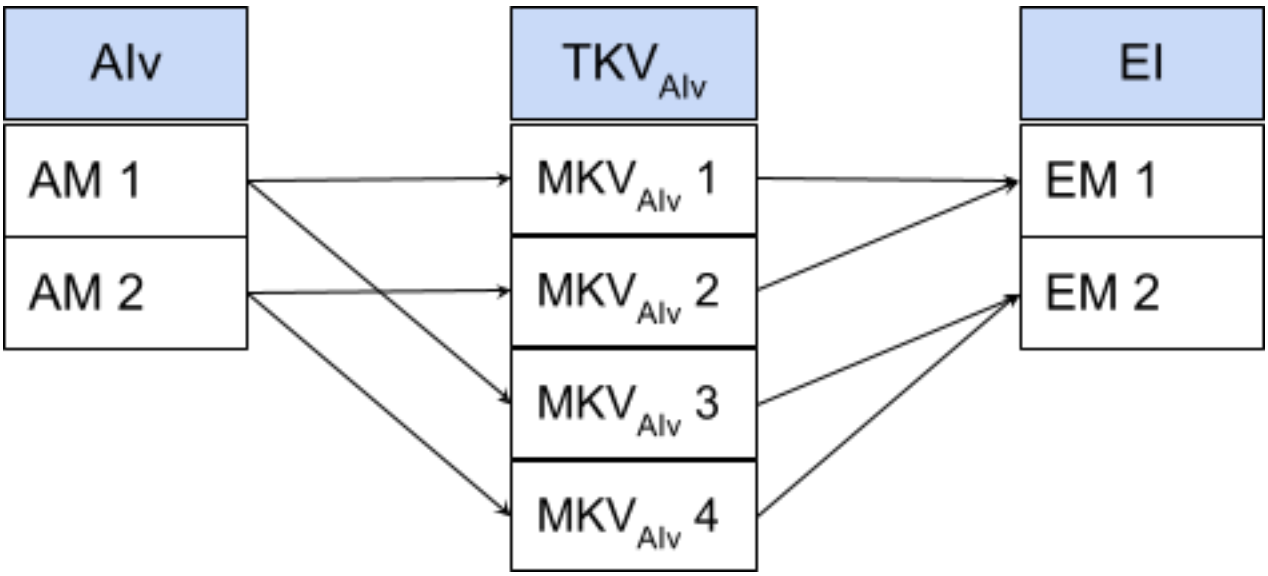

Abbildung 6: Typ- und Methoden-Konvertierungsvarianten von AIv

Dabei gilt jedoch, aufgrund der Überlegungen zur Kombination von angebotenen Komponenten (siehe auch Abb. 3), dass eine Typ-Konvertierungsvariante nicht zu jeder der erwarteten Methoden solche methodendelegationsrelevanten Informationen enthält. Abb. 7 zeigt einen solchen Fall mit dem angebotenen Interface AIu.

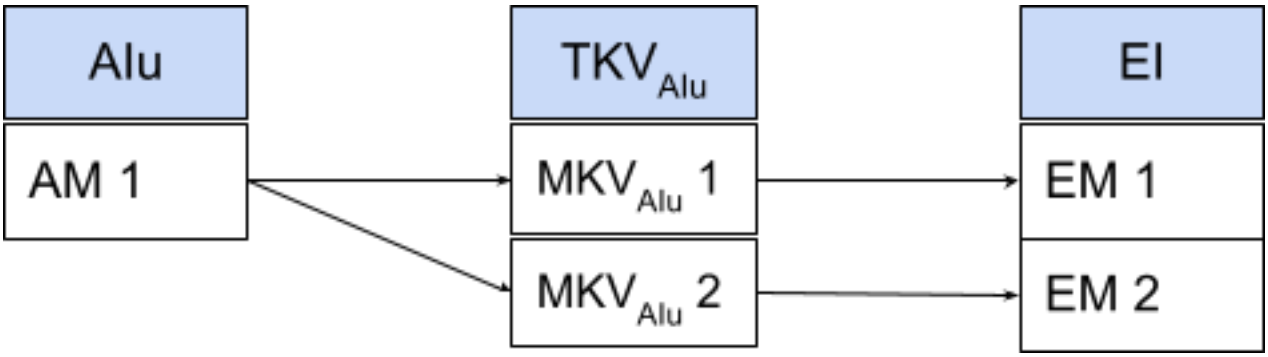

Abbildung 7: Typ- und Methoden-Konvertierungsvarianten von AIu

4.2.2 2. Stufe - Semantische Evaluation

Sofern alle Typ-Konvertierungsvarianten des erwarteten Interfaces bzgl. einer Menge von angebotenen Interfaces in der 1. Stufe ermittelt wurden, können die konkreten Implementierungen der angebotenen Interfaces hinsichtlich der spezifizierten erwarteten Semantik überprüft werden.

Diese Prüfung wird über die vorab spezifizierten Testfälle des erwarteten Interfaces vorgenommen. In einem vorherigen Abschnitt wurde schon kurz beschrieben, wie eine solche Testklasse aufgebaut ist. In diesem Abschnitt wird beschrieben, wie die zu testenden Komponenten ermittelt werden und

wie die Tests durchgeführt werden. Die semantische Evaluation erfolgt in 6 Schritten:

1. Ermittlung der Testklassen zum erwarteten Interface
2. Ermitteln aller Kombinationen von Konvertierungsvarianten
3. Erzeugen einer benötigten Komponente
4. Injizieren der benötigten Komponente
5. Durchführung der Tests
6. Auswertung des Testergebnisses

Ermittlung der Testklassen zum erwarteten Interface

Die Ermittlung der Testklassen erfolgt über die Annotation @QueryTypeTestReference, welche im erwarteten Interface spezifiziert wird. Von diesen Testklassen wird ein Testobjekt über den Default-Konstruktor erzeugt.

Kombination von Typ-Konvertierungsvarianten

In diesem Schritt werden die ermittelten Typ-Konvertierungsvarianten miteinander kombiniert, was einer Kombination der angebotenen Interfaces gleicht. Die Anzahl k der kombinierten Typ-Konvertierungsvarian kann jedoch variieren. Wenn $|EM|$ die Anzahl der Methoden im erwarteten Interface ist, gilt für k:

$$1 \leq k \leq |EM|$$

Da k variabel ist, wird dieser Schritt zusammen mit allen folgenden Schritten mitunter mehrfach durchlaufen. Die Nummer des jeweiligen Iterationsschrittes wird mit k gleichgesetzt. Abb. 8 zeigt die Kombinationen von Typ-Konvertierungsvarianten, die sich bezogen auf die Beispiele aus Abb. 6 und Abb. 7 im ersten Durchlauf ergeben. Im zweiten Durchlauf würde sich nur eine Kombination von Typ-Konvertierungsvarianten ergeben, da die beiden Typ-Konvertierungsvarianten von AIv und AIu miteinander kombiniert werden (siehe Abb. 9).

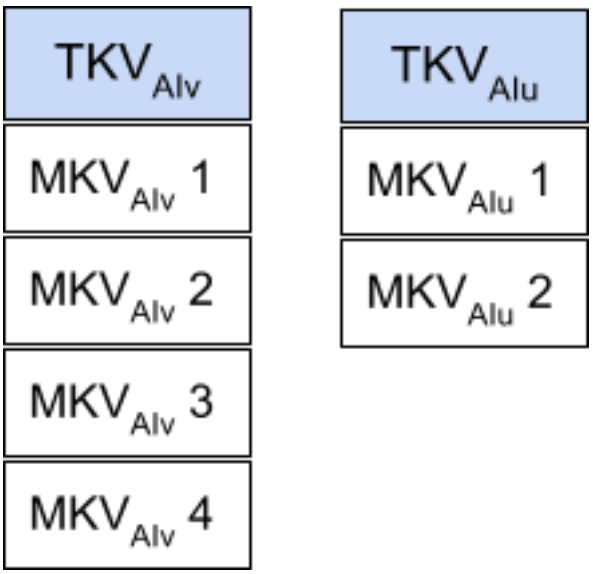

Abbildung 8: Kombinationen von Typ-Konvertierungsvarianten von AIu und AIv im ersten Durchlauf

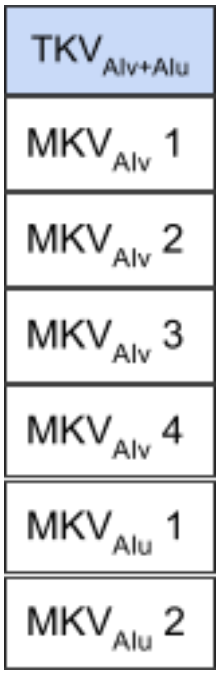

Abbildung 9: Kombinationen von Typ-Konvertierungsvarianten von AIu und AIv im zweiten Durchlauf

So berechnet sich die Anzahl an ermittelten Kombinationen von Typ-Konvertierungsvarianten ($|KombTKV|$) für jeden Durchlauf k in Abhängigkeit von der Anzahl der in der 1. Stufe ermittelten Typ-Konvertierungsvarianten ($|TKV|$) wie folgt:

$$|KombTKV| = \frac{|TKV|!}{((|TKV|! - k!) * k!)}$$

Erzeugen einer benötigten Komponente

Eine benötigte Komponente besteht aus einer Kombination von Methoden-Konvertierungsvarianten, wobei für jede erwartete Methode genau eine Methoden-Konvertierungsvariante innerhalb der benötigten Komponente existiert.

Für die Ermittlung der Kombinationen von Methoden-Konvertierungsvarianten wird eine Kombination von Typ-Konvertierungsvarianten aus der Ergebnismenge des aktuellen Durchlaufs des zweiten Schrittes selektiert. Die darin enthaltenen Methoden-Konvertierungsvarianten werden hinsichtlich der Methoden aus dem erwarteten Interface miteinander kombiniert.

Für die erste Kombinationen von Typ-Konvertierungsvarianten, die Abb. 8 zu entnehmen sind, (TKV_{AIu}) können folgende Kombinationen von Methoden-Konvertierungsvarianten erzeugt werden (siehe Abb. 10).

Abbildung 10: Kombinationen von Methoden-Konvertierungsvarianten AIu

Analog dazu werden für die zweite Kombinationen von Typ-Konvertierungsvarianten, die Abb. 8 zu entnehmen sind, (TKV_{AIu}) folgende Kombinationen von Methoden-Konvertierungsvarianten erzeugt (siehe Abb. 11).

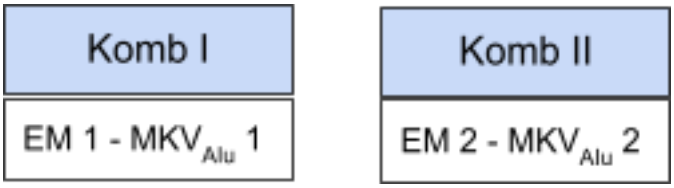

Abbildung 11: Kombinationen von Methoden-Konvertierungsvarianten AIu

Ausgehend von der Kombination von Typ-Konvertierungsvarianten aus Abb. 9 ($TKV_{AIu+AIv}$), sind in Abb. 12 die daraus resultieren Methoden-Konvertierungsvarianten dargestellt.

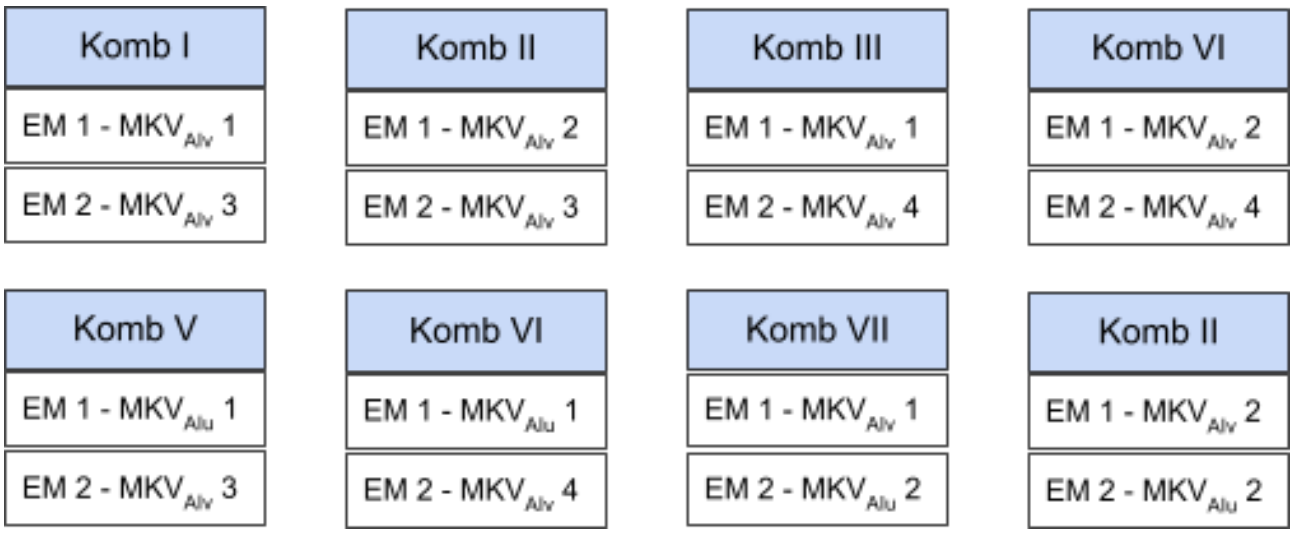

Abbildung 12: Kombinationen von Methoden-Konvertierungsvarianten AIu+AIv

Zu beachten ist, dass die ersten vier Kombinationen bereits im vorherigen Durchlauf erzeugt wurden (siehe Abb. 10) und dementsprechend auch getestet wurden.

Im Allgemeinen lässt sich sagen, dass die Anzahl der Kombinationen von Methoden-Konvertierungsvarianten von der Anzahl der Methoden im erwarteten Interface ($|EM|$) und der Anzahl an Methoden-Konvertierungsvaria ($|MKV|$) innerhalb der selektierten Kombination von Typ-Konvertierungsvarianten abhängig ist. Da aus einer Kombination von Methoden-Konvertierungsvarianten jeweils eine benötigte Komponente erzeugt werden kann, gilt für die Anzahl der benötigten Komponenten ($|Komb_{ben}|$) dasselbe.

$$|Komb_{ben}| = |Komb_{MKV}| = \frac{|MKV|!}{(|MVK|! - |EM|!) * |EM|!}$$

Injizieren der benötigten Komponente

Der Setter für die Setter-Injection wird in der Testklasse über die Annotation @QueryTypeInstance-Setter ermittelt. Danach wird diese Methode auf dem Testobjekt mit der benötigten Komponente als Parameter aufgerufen.

Durchführen der Tests

Die Testfälle aus der Testklasse werden über die Annotation @QueryTypeTest ermittelt und sequentiell ausgeführt. Als Ergebnis der Testausführung für eine benötigte Komponente wird ein Objekt des Typs TestResult zurückgegeben. Tritt bei der Testausführung eine Exception auf, wird diese im TestResult-Objekt hinterlegt. Im Anschluss wird das TestResult-Objekt direkt zurückgegeben, um die Ausführung der übrigen Tests zu verhindern. Wenn ein Test mit positivem Ergebnis durchgeführt wird, wird das Attribut passedTests im TestResult-Objekt inkrementiert. Sollten alle Tests erfolgreich durchgeführt worden sein, wird das TestResult Objekt zurückgegeben.

Ab dem zweiten Durchlauf werden werden die benötigten Komponenten in Schritt 3 aus den Typ-Konvertierungsvarianten mehrere angebotener Interfaces erzeugt. Das führt dazu, dass die Methodenaufrufe auf dem erwarteten Interface an unterschiedliche angebotene Komponenten delegiert werden. Hierbei kann der Fall eintreten, dass mehrere dieser Methoden von der Semantik her auf den gleichen Daten operieren müssen, die Delegation der Aufrufe aber an unterschiedliche Komponenten vorgenommen wird, die auch auf unterschiedlichen Daten arbeiten.

Ein Beispiel hierfür wäre eine Art Stack, der durch das erwartete Interface Stack beschrieben. Dieses enthält eine push und eine pop Methoden mit der Elemente im Stack hinzugefügt bzw. entfernt werden können (siehe Abb. 13). Hierbei ist anzunehmen, dass die beiden Methoden auf denselben Daten arbeiten, sodass nach dem Hinzufügen eines Elements a (push(a)) und dem darauf folgenden Aufruf der Methode pop() als Rückgabewert wieder das zuvor hinzugefügte Element a geliefert wird (siehe Abb. 14). Wenn die beiden Methoden-Aufrufe jedoch an zwei unterschiedliche Objekte StackA und StackB delegiert werden, die auf unterschiedlichen Daten operieren, dann würde dieses Verhalten nicht nachgewiesen werden können (siehe Abb. 15).

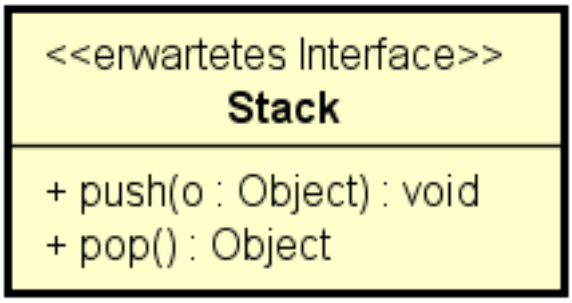

Abbildung 13: Erwartetes Interface Stack

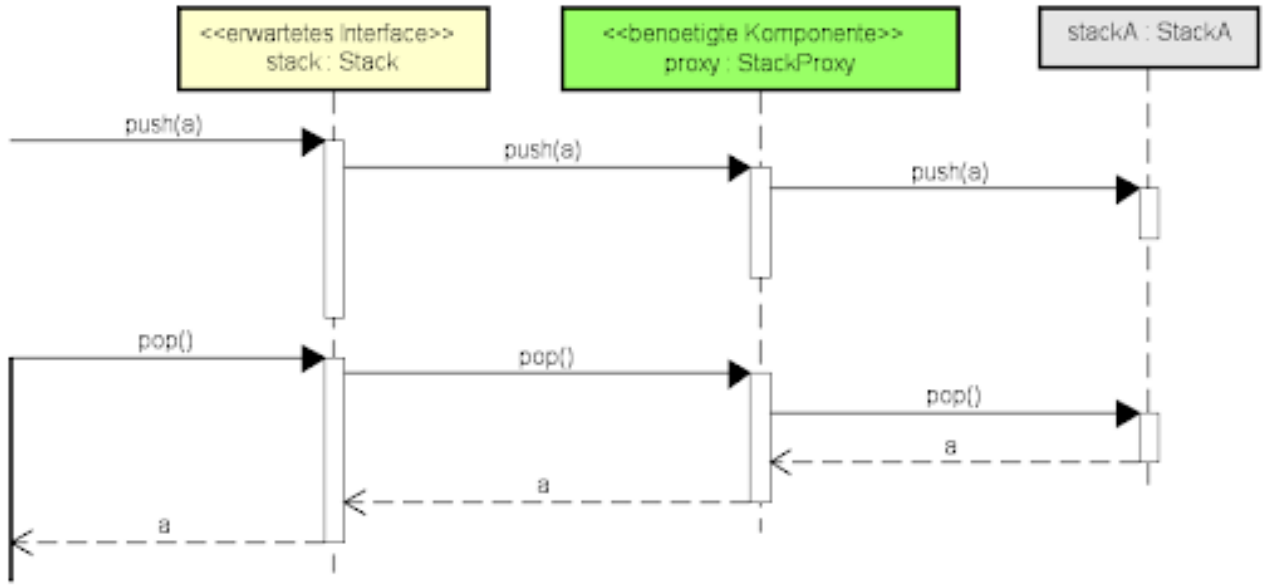

Abbildung 14: Delegation der Stack-Methoden an genau eine angebotene Komponente

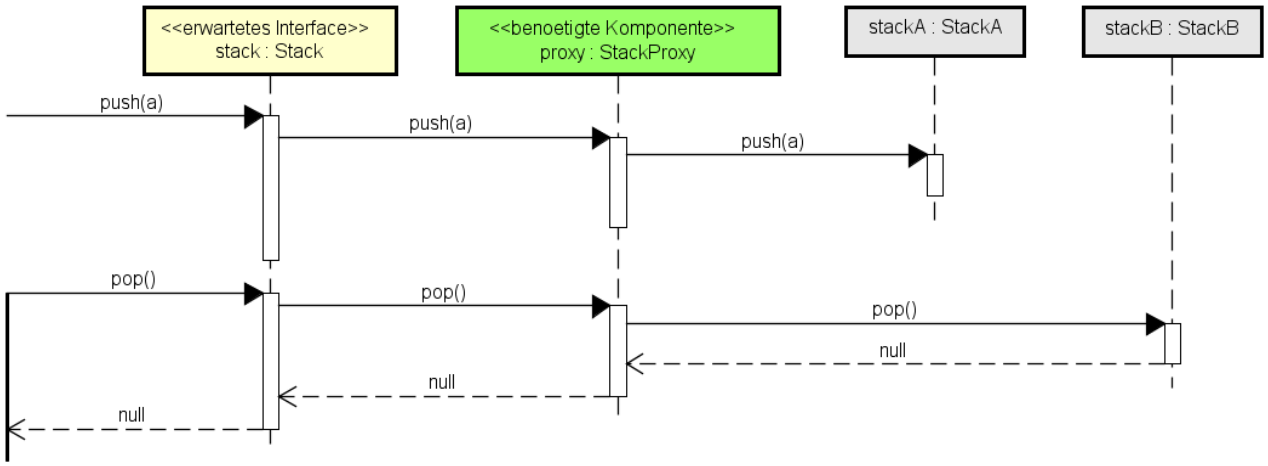

Abbildung 15: Delegation der Stack-Methoden an unterschiedliche angebotene Komponenten

In einem solchen Fall sollte der Zusammenhang dieser erwarteten Methoden in den Tests spezifiziert werden, sodass diese besonderen semantischen Anforderungen in diesem Schritt evaluiert werden können. Listing 3 zeigt ein Beispiel bezogen auf das Szenario aus Abb. 14.

Listing 3: Testklasse für ein erwartetes Interfaces Stack

```
1 public class StackTest {
2
3   private Stack stack;
4
5   @QueryTypeInstanceSetter
6   public void setProvider( Stack stack ) {
7     this.stack = stack;
8   }
```

```
9
10  @QueryTypeTest
11  public void pushPop() {
12    Object a = new Object();
13    stack.push( a );
14    Object evalObj = stack.pop();
15    assertTrue( a == evalObj );
16  }
17
18 }
```

Auswertung des Testergebnisses

Sofern alle Tests erfolgreich durchgelaufen sind, wird die aktuell selektierte benötigte Komponente als passend bewertet und als Ergebnis des Explorationsalgorithmus zurückgegeben.

Sollte einer der Tests nicht erfolgreich sein, wird die semantische Evaluation ab Schritt 3 (siehe 4.2.2) wiederholt. Sofern keine benötigten Komponenten mehr erzeugt werden können, ist die Suche nach einer passenden Komponente gescheitert.

Da die Suche zur Laufzeit ausgeführt wird, reicht es, wenn eine passende Komponente gefunden wird. Selbst wenn es mehrere passende Komponenten geben sollte, gäbe es in dem beschriebenen Verfahren keine Möglichkeit festzustellen, welche die semantischen Anforderungen besser erfüllt. Zwar wären könnte man die passenden Komponenten hinsichtlich der benötigten Systemressourcen untersuchen. Aufgrund der Vielzahl von möglichen Kombinationen (siehe 4.2.2 und 4.2.2) rechtfertigt der dafür notwendige Aufwand den daraus resultierenden Performancegewinn vermutlich nicht.

4.3 Heuristiken

4.3.1 Type-Matcher Rating basierte Heuristiken

Wie die Überschrift bereits andeutet, werden die Type-Matcher mit einem Rating versehen. Das Rating wird durch einen numerischen Wert dargestellt. Auf dieser Basis werden zwei Kategorien von Type-Matcher Ratings unterschieden:

1. Qualitatives Type-Matcher Rating

2. Quantitatives Type-Matcher Rating

Qualitatives Type-Matcher Rating

Das qualitative Type-Matcher Rating beschreibt den Grad der strukturellen Übereinstimmung des Source- und des Target-Typen. Dafür wird jeder Type-Matcher mit einem Basiswert versehen. Die konkreten Basiswerte sind im Abschnitt Evaluation beschrieben. Dieser Basiswert wird beim Erzeugen der Typ-Konvertierungsvarianten an die methodendelegationsrelevanten Informationen gehängt, sodass zu jeder Methode, zu der eine Methoden-Konvertierungsvariante existiert, auch ein Wert bzgl. des qualitatives Type-Matcher Rating zur Verfügung steht.

Der konkrete Wert für das qualitative Type-Matcher-Rating ermittelt sich grundlegend, wie bereits erwähnt, anhand eines Basiswertes. Sofern ein Type-Matcher jedoch einen internen Type-Matcher verwendet, um die konkreten methodendelegationsrelevanten Informationen zu erzeugen, ergibt sich der Wert des Type-Matcher-Ratings aus einer Akkumulation der Basiswerte des Type-Matchers und des internen Type-Matchers.

Damit ist das qualitative Type-Matcher Rating von folgenden Faktoren abhängig:

1. Die Wahl des Basiswertes der einzelnen Type-Matcher

2. Das Akkumulationsverfahren für das Type-Matcher Rating einer Typ-Konvertierungsvariante

3. Das Akkumulationsverfahren für das Type-Matcher Rating einer Methoden-Konvertierungsvariante

Alle drei Punkt werden bei der Evaluierung dieser Heuristik betrachtet.

Quantitatives Type-Matcher Rating

Das quantitative Type-Matcher Rating beschreibt, zu wie vielen der erwarteten Methoden in der erzeugten Typ-Konvertierungsvariante methodendelegationsrelevante Informationen vorliegen. Hierzu wird der methodendelegationsrelevante Informationen innerhalb der Typ-konvertierungsvariante durch die Anzahl der erwarteten Methoden geteilt. So stellt das quantitative Type-Matcher Rating also einen Prozentsatz dar.

Darauf aufbauend werden im Folgenden zwei Heuristiken vorgestellt, durch die es ermöglicht wird die ermittelten Typ-Konvertierungsvarianten in eine Reihenfolge zu bringen. So kann die Selektion der Kombination von Methoden-Konvertierungsvarianten im 3. Schritt der 2. Stufe des Explorationsalgorithmus anhand dieser Sortierung erfolgen.

4.3.2 Heuristik - TMR_Quant: Beachtung des quantitativen Type-Matcher Ratings

Es wird davon ausgegangen, dass eine angebotene Komponente, deren Schnittstelle strukturell mit der erwarteten Schnittstelle übereinstimmt und dabei zu jeder erwarteten Methode eine passende Methode anbietet, eher die semantischen Erwartungen erfüllt, als eine Kombination aus mehreren angebotenen Komponenten.

Daher sollten bei der Selektion der Kombinationen von Methoden-Konvertierungsvarianten in Schritt 3 der 2. Stufe des Explorationsalgorithmus (siehe 4.2.2) zuerst diejenigen Kombinationen gewählt werden, deren Elemente (Methoden-Konvertierungsvarianten) aus ein und derselben Typ-Konvertierungsvariant stammen, sprich deren quantitatives Type-Matcher Rating möglichst hoch ist.

4.3.3 Heuristik - TMR_Qual: Beachtung des qualitativen Type-Matcher Ratings

Es wird davon ausgegangen, dass eine von einer angebotene Komponente angebotene Methode, die strukturell mit einer erwarteten Methode zu einem höheren Grad übereinstimmt, auch eher die semantischen Erwartungen an diese Methode erfüllt.

Daher sollten bei der Selektion der Kombinationen von Methoden-Konvertierungsvarianten in Schritt 3 der 2. Stufe des Explorationsalgorithmus (siehe 4.2.2) zuerst diejenigen Kombinationen gewählt werden, deren Elemente (Methoden-Konvertierungsvarianten) aus den methodendelegationsrelevanten Informationen mit den niedrigsten qualitativen Type-Matcher Rating der Typ-Konvertierungsvarianten erzeugt wurden.

4.3.4 Testergebnis basierte Heuristiken

Diese Heuristiken werden auf der Basis des TestResult-Objektes, welches bei der semantischen Evaluation (2. Stufe, siehe 4.2.2) bei der Durchführung der Tests (Schritt 5, siehe 4.2.2) erzeugt wird. Die dafür notwendigen Informationen werden im TestResult-Objekt dementsprechend bei der Testausführung vermerkt. Ausgehend von dieser Basis führen diese Heuristiken im Allgemeinen dazu, dass bestimmte Methoden-Konvertierungsvarianten bei der weiteren Suche nach Kombinationen solcher (Schritt 2, , siehe 4.2.2) nicht mehr oder bevorzugt verwendet werden.

Heuristik - PREV_PASSED: Beachtung der teilweise bestandenen Tests

Es wird davon ausgegangen, dass es innerhalb der Testklassen Testmethoden gibt, die einzelne erwartete Methoden testen. Eine benötigte Komponente, die einen Teil dieser Tests besteht, verwendet für bestimmte Methoden scheinbar eher passende Methoden-Konvertierungsvarianten, als solche benötigten Komponenten, die keine dieser Tests bestehen.

Sofern sichergestellt ist, dass die benötigte Komponente aus einer einzigen Typ-Konvertierungsvariante erzeugt wurde und einen Teil der Tests besteht, sollte sie bei der Erzeugung benötigter Komponenten aus mehreren Typ-Konvertierungsvarianten bevorzugt verwendet werden.

Heuristik - BL_PM: Beachtung aufgerufener Pivot-Methode

Es wird davon ausgegangen, dass es beim Aufruf von Methoden und deren Delegation an angebotene Komponenten zu Fehlern/Exceptions kommen kann. Eine Methoden-Konvertierungsvariante, die zu solchen Fehlern führt, ist offensichtlich unbrauchbar. Daher ist es sinnvoll, diese Methoden-Konvertierungsvariante bei der weiteren Suche zu ignorieren. Zu diesem Zweck wird beim Auftreten einer Exception bei der Delegation einer Methode eine spezielle Excpetion (SigMaGlueException) geworfen, die bei der Testdurchführung entsprechend ausgewertet werden kann.

Da in einer Testmethode jedoch mehrere erwartet Methoden aufgerufen werden können, besteht die Möglichkeit, dass das Ergebnis der zuerst aufgerufenen erwarteten Methoden aufgrund einer passenden Methoden-Konvertierungsvariante nicht direkt bei deren Aufruf zu einer Exception führt, sondern erst bei der Verwendung des Ergebnisses als Parameter des folgenden Aufrufs einer erwarteten Methode. In so einem Fall ist es nicht möglich zu erkennen, für welche der beiden Methoden tatsächlich eine unpassende Methoden-Konvertierungsvariante verwendet wird. Beim Auftreten einer Exception während des Aufrufs der ersten erwarteten Methode, ist jedoch davon auszugehen, dass für diesen Aufruf eine unpassende Methoden-Konvertierungsvariante verwendet wurde, weshalb diese bei der weiteren Suche ignoriert werden sollte.

Eine Pivot-Methode beschreibt dabei die zuerst aufgerufene erwartete Methode innerhalb einer Testmethode. Um eine Möglichkeit zu schaffen, den Aufruf dieser Pivot-Methode nach außen mitzuteilen, müssen die Testklassen erweitert werden. Hierzu steht das Interface PivotMethodTestInfo bereit.

Dieses Interface deklariert drei Methoden, die in der Testklasse spezifiziert werden müssen. Grundlegend ist der Mechanismus so angedacht, dass innerhalb der Testklasse ein Flag spezifiziert wird, welches durch die Methode reset() auf den Ausgangswert zurückgesetzt wird und durch die Methode markPivotMethodCallExecuted() auf einen anderen Wert umgesetzt wird. Der Aufruf der Methode pivotMethodCallExecuted() sollte true liefern, wenn dieses Flag nicht dem Ausgangswert übereinstimmt.

Innerhalb der Testmethode sollte dann vor zu Beginn immer die Methode reset() aufgerufen werden, da andernfalls die Testergebnisse verfälscht werden können. Zudem muss die Methode markPivot-

MethodCallExecuted() direkt nach dem Aufruf der Pivot-Method aufgerufen werden.

So kann bei der Testdurchführung festgestellt werden, ob die mögliche Exception vor oder nach dem Aufruf der Pivot-Methode erfolgte. Welche Methode beim Aufruf zu einer Exception geführt hat, wird innerhalb der SigMaGlueException überliefert.

Heuristik - BL_SM: Beachtung fehlgeschlagener Single-Method Tests

Es wird davon ausgegangen, dass es innerhalb der Testklassen Testmethoden gibt, die auf eine ganz bestimmte erwartete Methode zugeschnitten sind (Single-Method Test). Das setzt unter anderem voraus, dass in dieser Testmethode von den erwarteten Methoden nur diese eine aufgerufen wird.

Weiterhin muss an der Testmethode eine Information zur Verfügung stehen, die eine Auskunft darüber gibt, welche Methode dort getestet wird. Zu diesem Zweck kann an der @QueryTypeTest-Annotation ein Parameter mit der Bezeichnung testedSingleMethod spezifiziert werden. Dort soll dementsprechend der Name der getesteten Methode angegeben werden. So kann bei der Testausführung evaluiert werden, ob eine bestimmte Methode von der getesteten benötigten Komponente semantisch nicht passt.

Da die konkrete Methode, deren Test fehlschlägt, bekannt ist, kann auch die verwendete Methoden-Konvertierungsvariante ermittelt werden. Diese Methoden-Konvertierungsvariante sollte bei der weitere Suche nicht mehr beachtet werden, da mit diesem Test sichergestellt wurde, dass sie nicht Teil einer passenden benötigten Komponente sein kann.

4.3.5 Koordination

Der Einsatz der Heuristiken muss bei der Suche koordiniert werden. Der Explorationsalgorithmus ist so aufgebaut, dass im 2. Schritt der 2. Stufe (siehe 4.2.2) die Heuristiken zum Einsatz kommen.

Begonnen wird mit der Heuristik TMR_Quant, sodass zuerst alle mÃ¶glichen Kombinationen von Methoden-Konvertierungsvarianten ermittelt werden, die aus einer einzelnen Typ-Konvertierungsvariante stammen. Sind die mÃ¶glichen Kombinationen von Methoden-Konvertierungsvarianten ausgeschÃ¶pft, wird der Prozess mit der Ermittlung aller mÃ¶glichen Kombinationen von Methoden-Konvertierungsvarianten, die aus 2 Typ-Konvertierungsvarianten stammen, wiederholt. Die Anzahl der zu kombinierenden Typ-Konvertierungsvarianten wird somit jedes mal erhÃ¶rt, wenn die bereits ermittelten Kombinationen von Methoden-Konvertierungsvarianten ausgeschÃ¶pft sind.

Innerhalb eines der eben beschriebenen Iterationsschritte werden die anderen Heuristiken eingesetzt.

Die Heuristik TMR_Qual sortiert die Typ-Konvertierungsvarianten, die bei der Ermittlung der Kombinationen von Methoden-Konvertierungsvarianten verwendet werden. Diese werden dann ihrer Reihenfolge entsprechend verwendet um Methoden-Konvertierungsvarianten zu ermittelt. Diesen Methoden-Konvertierungsvarianten wurde ebenfalls ein Type-Matcher Rating mitgegeben, nach welchem jene nun sortiert werden und dann entsprechend dieser Reihenfolge sequentiell getestet werden.

Die Heuristik PREV_PASSED wird, wie alle weiteren Heuristiken, erst nach der ersten Iterationsstufe eingesetzt. Das liegt daran, dass fÃ¼r die Anwendung dieser Heuristiken bereist Testergebnisse vorliegen mÃ¼ssen. PREV_PASSED sorgt nochmals fÃ¼r eine Umsortierung der Typ-Konvertierungsvarianten, die bei der Ermittlung der Kombinationen von Methoden-Konvertierungsvarianten verwendet werden, sodass die bevorzugten Typ-Konvertierungsvarianten zuerst verwendet werden.

Die Heuristiken BL_PM und BL_SM sorgen dafÃ¼r, dass bei der Kombination von Methoden-Konvertierungsvar
diejenigen Ã¼bersprungen werden, die laut der jeweiligen Heuristik nicht mehr in Betracht gezogen werden sollen.

Literaturverzeichnis

[BNL+06] BAJRACHARYA, SUSHIL, TRUNG NGO, ERIK LINSTEAD, YIMENG DOU, PAUL RIGOR, PIERRE BALDI und CRISTINA LOPES: *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, Seiten 681–682, New York, NY, USA, 2006. Association for Computing Machinery.

[Hum08] HUMMEL, OLIVER: *Semantic Component Retrieval in Software Engineering*. Doktorarbeit, April 2008.

[LLBO07] LAZZARINI LEMOS, OTAVIO AUGUSTO, SUSHIL KRISHNA BAJRACHARYA und JOEL OSSHER: *CodeGenie: A Tool for Test-Driven Source Code Search*. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, Seite 917?918, New York, NY, USA, 2007. Association for Computing Machinery.

[ZW95] ZAREMSKI, AMY MOORMANN und JEANNETTE M. WING: *Signature Matching: A Tool for Using Software Libraries*. ACM Trans. Softw. Eng. Methodol., 4(2):146?170, April 1995.