Masterarbeit

Thema der Arbeit

Name des Erstellers

Themensteller: Prof. Dr. Wolfram Schiffmann
Betreuer: Titel des Seminars
Lehrgebiet Rechnerarchitektur
Fachbereich Informatik

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kurzfassung

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).

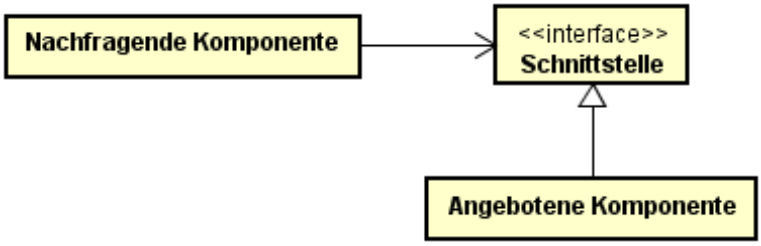


Abbildung 1: Abhängigkeiten von nachfragenden und angebotenen Komponenten

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.

1.2 Verwandte Arbeiten

Ein solcher Ansatz wurde bereits in [BNL+06] von Bajaracharya et al. verfolgt. Diese Gruppe entwickelte eine Search Engine namens Sourcerer, welche Suche von Open Source Code im Internet ermöglichte. Darauf aufbauend wurde von derselben Gruppe in [LLBO07] ein Tool namens CodeGenie entwickelt, welches einem Softwareentwickler die Code Suche über ein Eclipse-Plugin ermöglicht. In diesem Zusammenhang wurde 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, welches ebenfalls das Konzept der TDCS verfolgt. TDCS 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 Search Engines, das Internet nach bestehendem Source Code zu durchsuchen und damit bereits bestehende Implementierungen für eine nachfragende Komponente zu ermitteln.

1.3 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 dieses Interfaces beschrieben. Das Interfaces einer angebotenen Komponente 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 folgendem Szenario kommen, welches durch die 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.

1.3.1 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.

1.3.2 Nichtfunktionale Anforderungen

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

- Die Suche muss innerhalb des Transaktionstimeouts zu einem Ergebnis führen. (Im verwendeten System ist dieses auf 5 Minuten festgesetzt.)
- 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.

2 Voraussetzungen

2.1 Spezifikation der Erwartungen

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

2.1.1 Erwartete Syntax

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.

2.1.2 Erwartete Semantik

Der zweite Teil besteht aus einer Menge von Testfällen, durch die die erwartete 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 einen 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

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

  public void intubate( AccidentParticipant injured );

  public void extinguishFire( Fire fire );
}
```

Listing 2: Testklasse des erwarteten Interfaces IntubatingFireFighter

```
public class IntubatingFireFighterTest {

  private IntubatingFireFighter intubatingFireFighter;

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

  @QueryTypeTest
  public void free() {
    Fire fire = new Fire();
    intubatingFireFighter.extinguishFire( fire );
    assertFalse( fire.isActive() );
  }

  @QueryTypeTest
  public void intubate() {
    Collection<Suffer> suffer = Arrays.asList( Suffer.BREATH_PROBLEMS );
    AccidentParticipant patient = new AccidentParticipant( suffer );
    intubatingFireFighter.intubate( patient );
    assertTrue( patient.isStabilized() );
  }
}
```

2.2 Ermittlung angebotener Komponenten

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.

3 Explorationsalgorithmus

Mit diesen Voraussetzungen kann eine Komponente entwickelt werden, welche die Erwartungen der nachfragenden Komponente mit den bestehenden Funktionalitäten der angebotenen Komponenten zusammenbringt. In Abbildung 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.

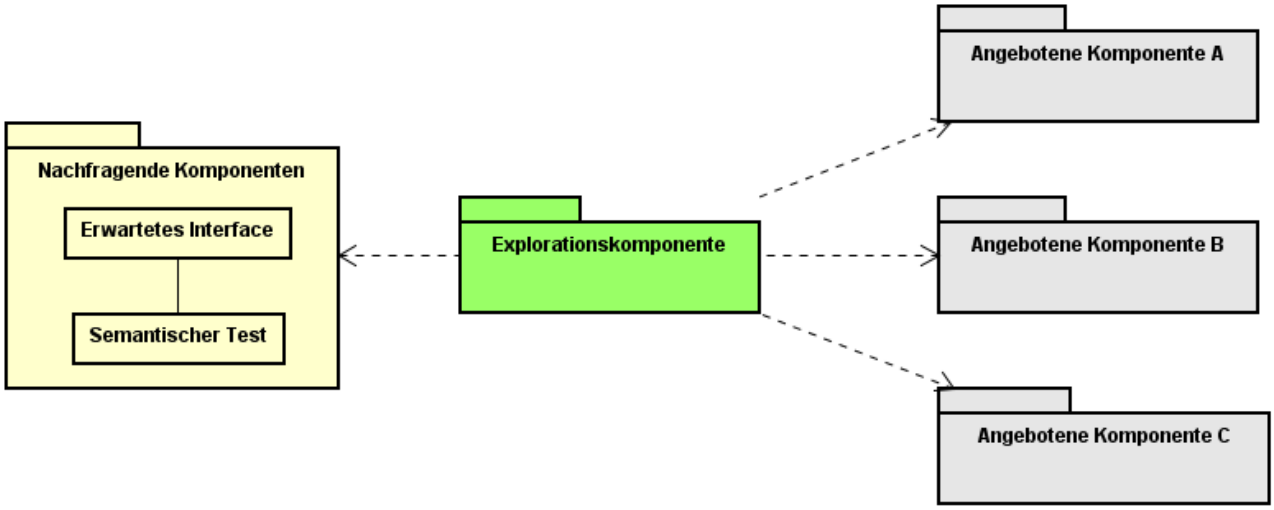


Abbildung 2: Allgemeiner Aufbau des System mit der Explorationskomponente

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 java.util.Function, über die die konkreten Implementierungen der angebotenen Interfaces ermittelt werden können. Die Suche wird mit dem Aufruf der Methode getDesiredComponent gestartet, welcher das erwartete Interface als Parameter übergeben werden muss. Somit kann ein Objekt der Klasse DesiredComponentFinder für mehrere Suchen mit unterschiedlichen erwarteten Interfaces verwendet werden.

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

mitunter nur durch eine Kombination von angebotenen Komponenten erfüllt werden können. Aus diesem Grund wird innerhalb der Explorationskomponente eine so genannte benötigte Komponente erzeugt, in der das Zusammenspiel einer solchen Kombination von angebotenen Komponenten verwaltet wird. Ein solches Szenario ist Abbildung 3 zu entnehmen.

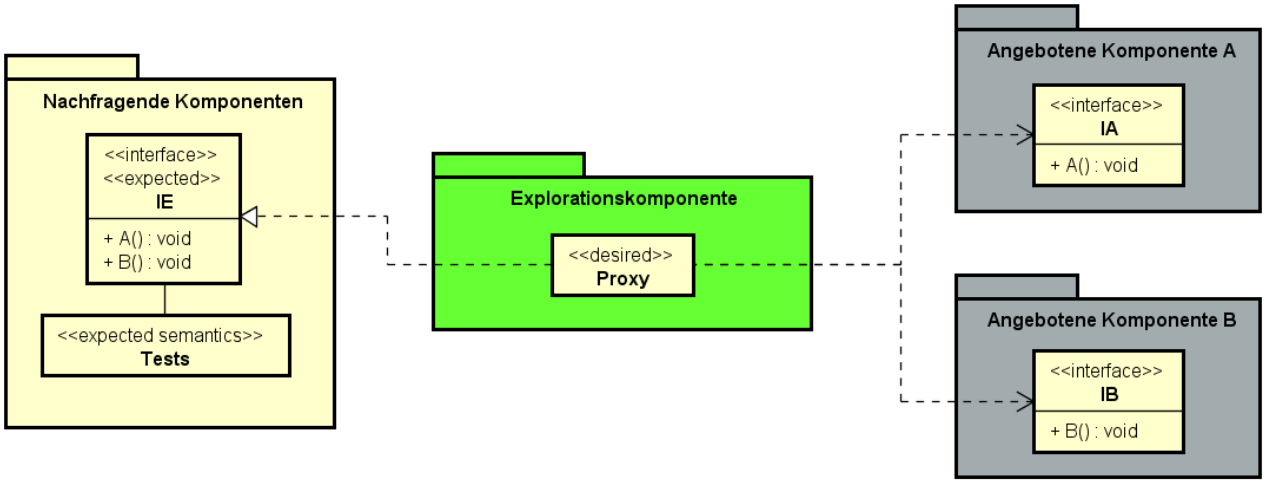


Abbildung 3: Kombination von angebotenen Komponenten

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

3.1 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), welches die Methodenaufrufe an ObjA delegiert (vgl. Abbildung 3).

Hummel hatte hierzu bereits auf einige Matcher von Zaremski und Wing [ZW95] zurückgegriffen, die in dieser Arbeit ebenfalls zum Einsatz kommen. Die Definitionen der Matcher beziehen sich vorrangig auf die Programmiersprache Java, weshalb grundlegend von einer nominalen Typkonformität auszugehen ist.

3.1.1 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 $A < B$ 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 pt. 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 über die Quantoren und Junktoren der Prädikatenlogik 1. Stufe ist ebenfalls möglich.

Die Delegation von Methodenaufrufen auf einem Objekt wird mit dem Operator $\Rightarrow$ 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 ($\Rightarrow$). 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.

Die folgenden Definitionen der Matcher bestehen jeweils aus zwei Teilen. Der erste Teil (Übereinstimmung) definiert, unter welchen Bedingungen über den entsprechenden Matcher zwei Typen als übereinstimmend gelten. Der zweite Teil (Konvertierung) beschreibt, wie die Methoden-Aufrufe auf einem Objekt eines der beiden übereinstimmenden Typen an das Objekt des anderen Typen delegiert werden.

Jeder Matcher wird zusätzlich im Vorfeld durch ein Szenario motiviert, in dem der jeweilige Matcher zur Anwendung kommen kann. In den dazugehörigen Diagrammen ist das Sourc-Objekt jeweils mit *source* und das Target-Objekt jeweils mit *target* bezeichnet. Um die Verwendung der Implementierungen der einzelnen Matcher in Verbindung mit diesem Szenario nachvollziehen zu können, wird jeweils auf einen Abschnitt aus Anhang A verweisen. Dort sind Code-Beispiele für die Verwendung der Matcher in Bezug auf das jeweilige Szenario mit entsprechenden Nachbedingungen hinterlegt.

3.1.2 ExactTypeMatcher

Szenario

Dieser Matcher stellt das Matching zweier identischer Typen fest. In dem Szenario wird von zwei Objekten vom Typ SuperClass ausgegangen. Die Klasse SuperClass ist in Abbildung 4 dargestellt. Der Aufruf einer Methode auf dem Source-Objekt führt zu einer Delgation der Methode an das Target-Objekt (siehe Abbildung 5.

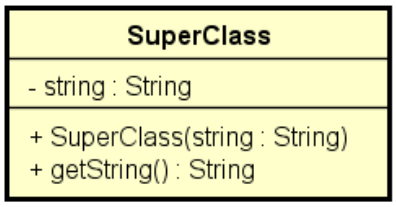


Abbildung 4: SuperClass

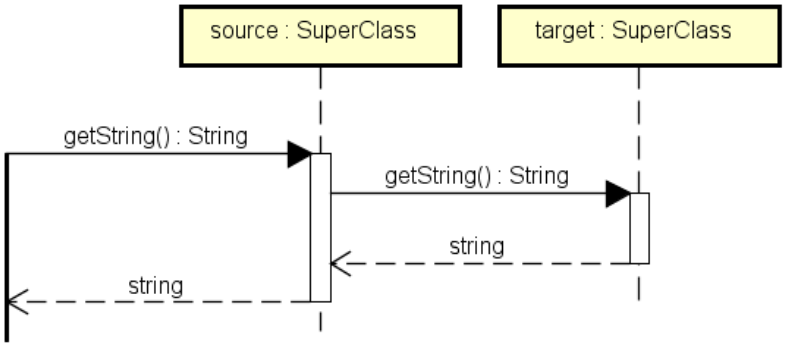


Abbildung 5: Szenario ExactTypeMatcher

Definition

Übereinstimmung (ExactTypeMatcher)

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

Konvertierung (ExactTypeMatcher)
Sei m eine Methode des Typen A und B.

$$A.m \Rightarrow B.m$$

Ein Beispiel für die Verwendung des Matchers ist in Anhang A.1 zu finden.

3.1.3 GenTypeMatcher

Szenario

Dieser Matcher stellt das Matching zwischen zwei Typen her, die in einer Vererbungsbeziehung stehen. Speziell erlaubt dieser Matcher das Matching eines Supertyps als Source-Typen mit einem Subtypen als Target-Typen. In dem Szenario wird neben dem Typ SuperClass aus Abbildung 4 von einem weiteren Typen SubClass ausgegangen. Dabei stehen diese beiden Typen in einer Vererbungsbeziehnung, die in Abbildung 11 dargestellt wird. Der Aufruf einer Metho-

de auf dem Source-Objekt führt zu einer Delgation der Methode an das Target-Objekt (siehe Abbildung 12.

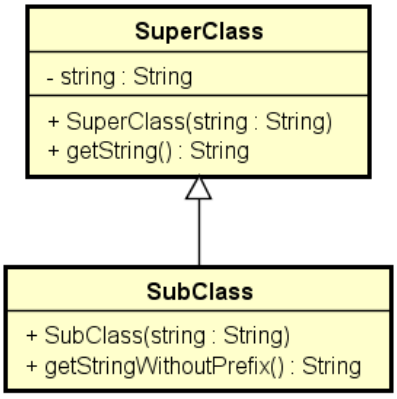


Abbildung 6: Beziehung zwischen SuperClass und SubClass

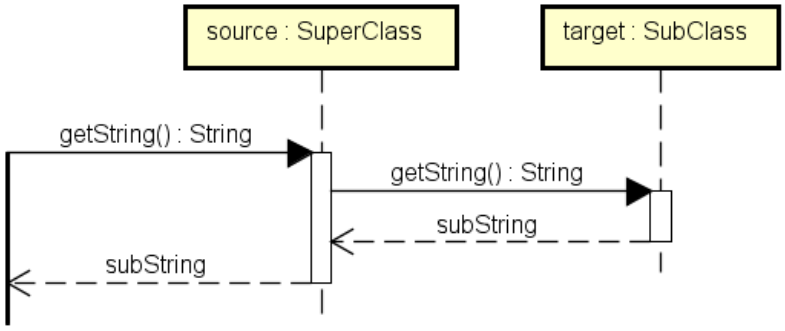


Abbildung 7: Szenario GenTypeMatcher

Definition

Übereinstimmung (GenTypeMatcher)

$$A \equiv_{gen} B \text{ wenn } B < A$$

Konvertierung (GenTypeMatcher)
Sei m eine Methode des Typs A, die aufgrund der Vererbung auch von Typ B bereitgestellt wird.

$$A.m \Rightarrow B.m$$

Ein Beispiel für die Verwendung des Matchers ist in Anhang A.2 zu finden.

3.1.4 SpecTypeMatcher

Szenario

Analog zum GenTypeMatcher stellt der SpecTypeMatcher ebenfalls das Matching zwischen Typen fest, die in einer Vererbungsbeziehung stehen. Allerdings ist der Source-Typ in diesem Matcher der Subtyp und der Target-Type der Supertyp. In dem Szenario wieder wiederum von den Klassen SuperClass und SubClass aus Abbildung 11 ausgegangen. Der Methodenaufruf erfolgt hier aber auf dem Subtypen und wird an den Supertypen delegiert (siehe Abbildung 8).

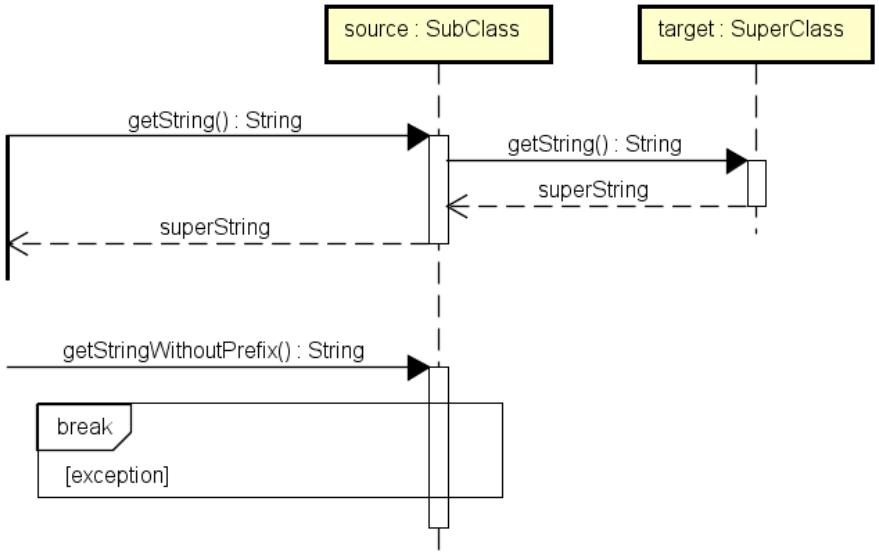


Abbildung 8: Szenario SpecTypeMatcher

Dabei sind zwei Methodenaufrufe auf dem Subtyp beschrieben. Während der Aufruf der Methode getString erfolgreich delegiert werden kann, führt der Aufruf der Methode getStringWithoutPrefix zu einem Laufzeitfehler, da der Matcher keine passende Methode in dem Target-Typ ermitteln kann. Dieses Problem tritt bei allen Methoden auf, die nicht vom Supertyp an den Subtyp vererbt oder mitunter dort überschrieben wurden.[1] Aus diesem Grund muss diese Bedingung in der Definition der Konvertierung dieses Matchers mit aufgenommen werden.

[1] Anders gesagt, ermöglicht dieser Match einen Downcast, bei dem ein Objekt eines allgemeinen Typen auf einen spezielleren Typen gecastet wird. Das Problem bzgl. des fehlschlagenden Methodenaufrufs in der beschriebene Form ist bei einem Downcast allgegenwärtig.

Definition

Übereinstimmung (SpecTypeMatcher)

$$A \equiv_{genspec} B \text{ wenn } A < B$$

Konvertierung (SpecTypeMatcher)
Sei m eine Methode des Typs A, die von B an A vererbt wurde.

$$A.m \Rightarrow B.m$$

Ein Beispiel für die Verwendung des Matchers ist in Anhang A.3 zu finden.

3.1.5 WrappedTypeMatcher

Szenario

Die bisherigen Type-Matcher sind in der Lage 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 sich beiden Typen auf anderem Wegen assoziieren lassen. Ein Beispiel dafür wäre Boxed- bzw. - noch allgemeiner gefasst - Wrapper-Typen. Abbildung 9 zwei Klassen dar, die in einer solchen Beziehung zueinander stehen. Bezüglich des Matchings sind auch hier wiederum zwei Fälle zu unterscheiden. Der erste Fall, in demdas Matching des Source-Typen SubClass mit dem Typen eines Attributs wrapped des Traget-Typen SubWrapper festgestellt werden kann, ist in Abbildung 10 dargestellt.

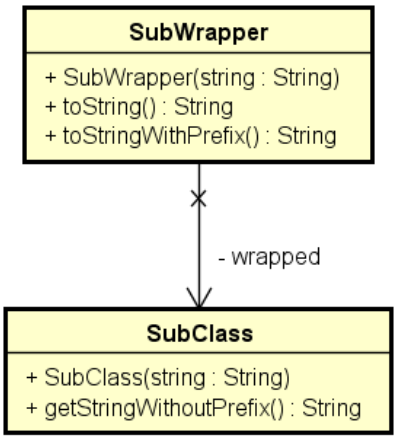


Abbildung 9: Beziehung zwischen SubClass und SubWrapper

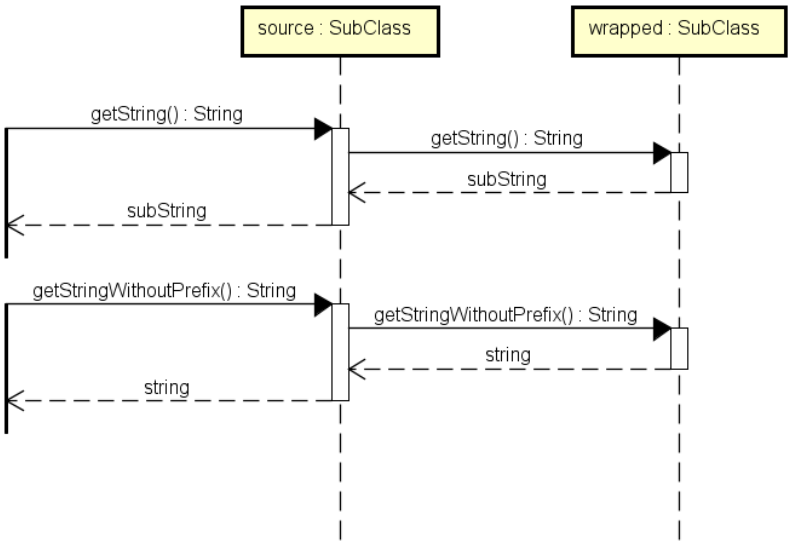


Abbildung 10: Szenario WrappedTypeMatcher

Der WrappedTypeMatcher stellt das Matching für ein solches Szenario fest. Das Matching der beiden Typen beruht letztendlich auf einem Matching zwischen dem Source-Type und dem Typen eines Attributs des Target-Typs. Der Matcher, über den dieses Matching innerhalb des WrappedTypeMatchers festgestellt wird, wird als interner Matcher bezeichnet. In dem Szenario aus Abbildung 10 wird als interner Matcher der bereits beschriebene ExactTypeMatcher verwendet, weil der Source-Type und der Typ des Attributs wrapped identisch sind.

Definition

Übereinstimmung (WrappedTypeMatcher)

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

Der zuvor genannte interne Matcher wird in der Definition mit M beschrieben, was stellvertretend für eine Menge von Matchern steht. Als interne Matcher kommen hierbei der ExactTypeMatcher, der GenTypeMatcher und der SpecTypeMatcher in Frage.

Konvertierung (WrappedTypeMatcher)

Sei m eine Methode des Typs A. Sei weiterhin B ein Typ, der ein Attribut vom Typ $attr$ enthält, für den gilt $A \equiv_M attr$.

$$A.m \Rightarrow (M)attr.m$$

Ein Beispiel für die Verwendung des Matchers in Bezug auf das o.g. Szenario ist in Anhang A.4 zu finden. Außerdem sind dort auch weitere Szenarien aufgeführt, in denen der GenTypeMatcher oder der SpecTypeMatcher als interner Matcher zur Anwendung kommen.

3.1.6 WrapperTypeMatcher

Szenario

Die bisherigen Type-Matcher sind in der Lage 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 sich beiden Typen auf anderem Wegen assoziieren lassen. Ein Beispiel dafür wäre Boxed- bzw. - noch allgemeiner gefasst - Wrapper-Typen. Abbildung 13 zwei Klassen dar, die in einer solchen Beziehnung zueinander stehen. Bezüglich des Matchings sind auch hier wieder zwei Fälle zu unterscheiden. Der erste Fall, in dem der Source-Typ ...., ist in Abbildung **??** dargestellt. Der WrappedTypeMatcher stellt das Matching für ein solches Szenario fest.

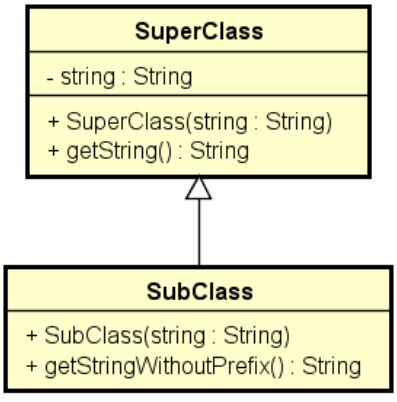


Abbildung 11: Beziehung zwischen SuperClass und SubClass

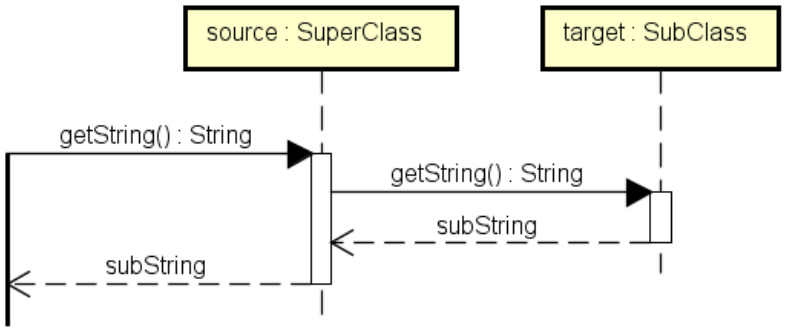


Abbildung 12: Szenario GenTypeMatcher

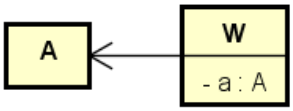


Abbildung 13: Beispiel Wrapper-Typ allgemein

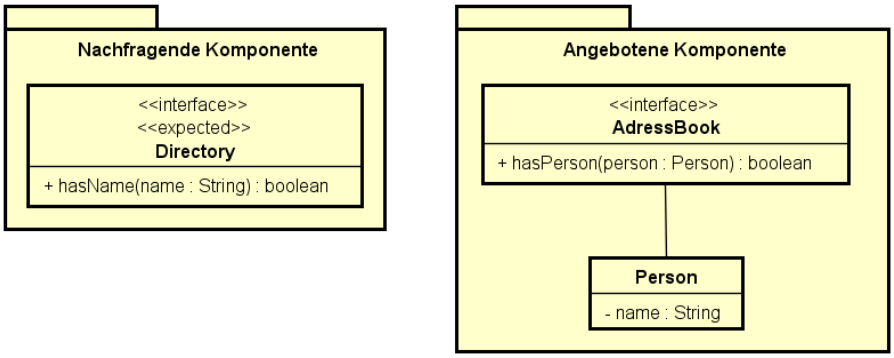


Abbildung 14: Beispiel Wrapper-Typ AdressBook

Definition

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 Übereinstimmung von Typen gefordert.

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.

3.1.7 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. Abbildung 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.

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

3.1.8 Typ- und Methoden-Konvertierungsvarianten

Die Konvertierung der einzelnen Type-Matcher liefert eine Menge von so genannten Typ-Konvertierungsvarianten. 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 intern wiederum eine Übereinstimmung von Typen fordern, 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 werden verwendet um so genannten Methoden-Konvertierungsvarianten zu erzeugen. Diese 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.

In Abbildung 15 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 jede der angebotenen Methoden strukturell mit jeder der erwarteten Methoden übereinstimmen würde. Dementsprechend enthält die Typ-Konvertierungsvariante (TKV) methodendelegationsrelevante Informationen, aus denen insgesamt 4 Methoden-Konvertierungsvarianten erzeugt werden, wovon jede eine Konvertierung entlang der eingezeichneten Pfeile ermöglicht.

Dabei gilt jedoch, aufgrund der Überlegungen zur Kombination von angebotenen Komponenten (siehe auch Abbildung 3), dass eine Typ-Konvertierungsvariante nicht zu jeder der erwarteten

Methoden solche methodendelegationsrelevanten Informationen enthält. Abbildung 16 zeigt einen solchen Fall mit dem angebotenen Interface AIu.

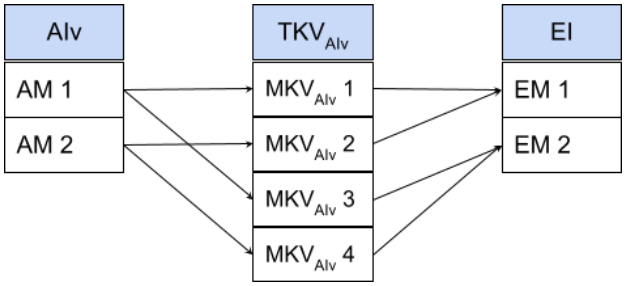


Abbildung 15: Typ- und Methoden-Konvertierungsvarianten von AIv

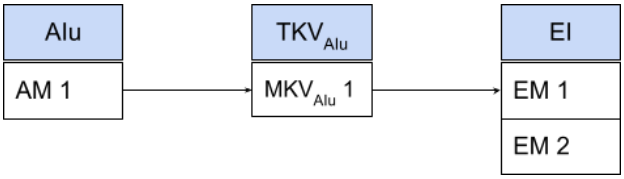
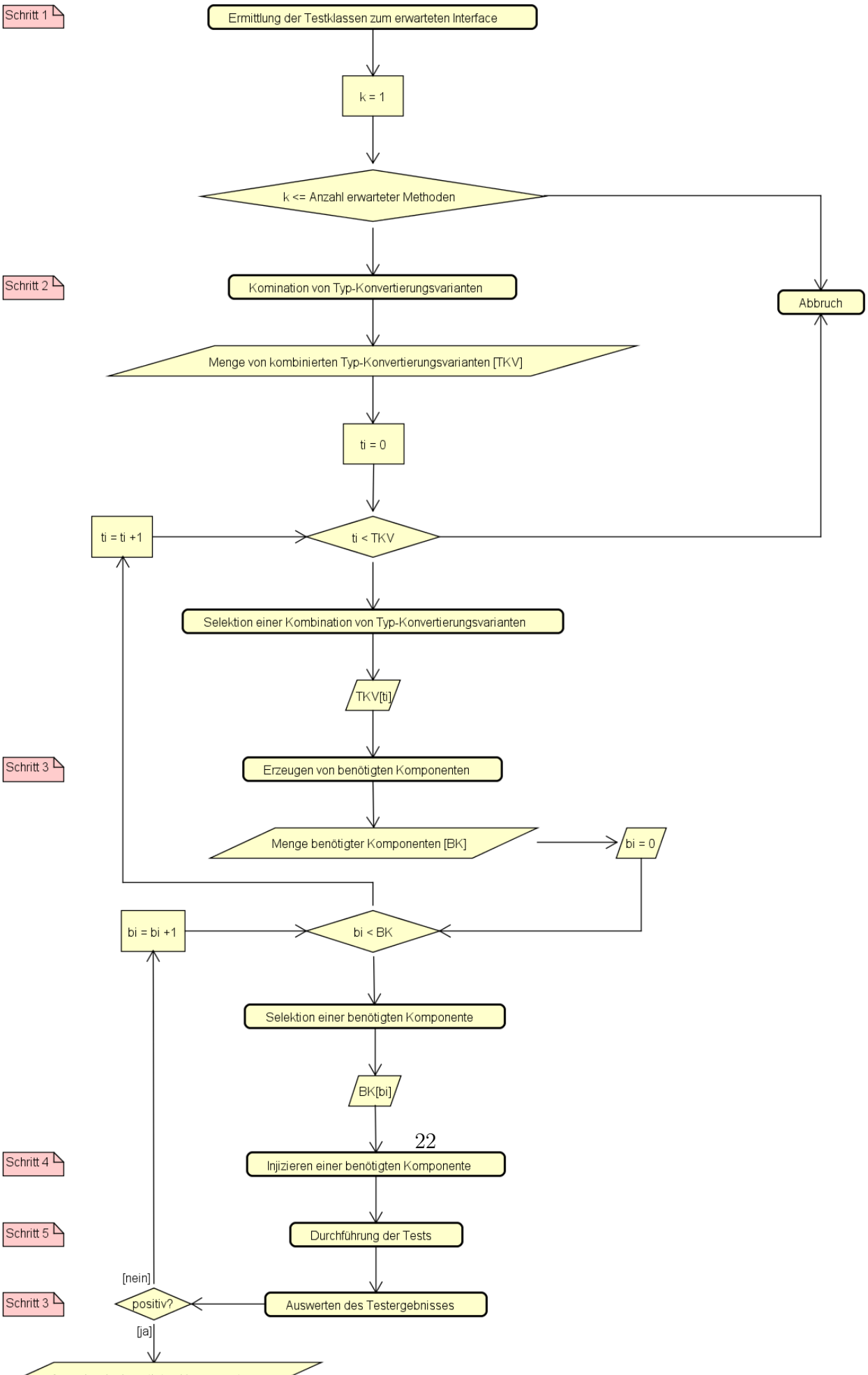


Abbildung 16: Typ- und Methoden-Konvertierungsvarianten von AIu

3.2 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 benötigten Komponenten erzeugt und getestet 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 benötigten Komponenten ermittelt werden und wie die Tests durchgeführt werden. Dies erfolgt in 6 Schritten, die im Folgenden erläutert werden. Eine schmatische Darstellung der Semantischen Evaluation ist Abbildung 17 zu entnehmen.


Schritt 1
Ermittlung der Testklassen zum erwarteten Interface
k = 1
k <= Anzahl erwarteter Methoden
Schritt 2
Komination von Typ-Konvertierungsvarianten
Abbruch
Menge von kombinierten Typ-Konvertierungsvarianten [TKV]
ti = 0
ti = ti +1
ti < TKV
Selektion einer Kombination von Typ-Konvertierungsvarianten
TKV[ti]
Schritt 3
Erzeugen von benötigten Komponenten
Menge benötigter Komponenten [BK]
bi = 0
bi = bi +1
bi < BK
Selektion einer benötigten Komponente
BK[bi]
Schritt 4
Injizieren einer benötigten Komponente
Schritt 5
Durchführung der Tests
[nein]
Schritt 3
positiv?
Auswerten des Testergebnisses
[ja]

3.2.1 Schritt 1: 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.

3.2.2 Schritt 2: 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 zu kombinierenden Typ-Konvertierungsvarianten 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. Somit wird die Anzahl der zu kombinierenden Typ-Konvertierungsvarianten mit jedem Durchlauf erhöht. Abbildung 18 zeigt die Kombinationen von Typ-Konvertierungsvarianten, die sich - bezogen auf die Beispiele aus Abbildung 15 und Abbildung 16 - 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 Abbildung 19).

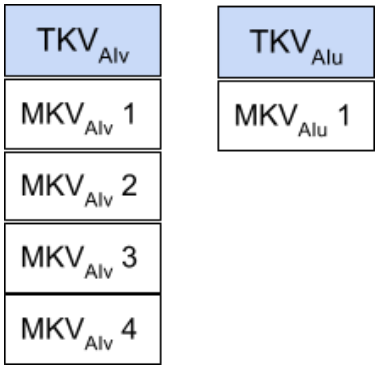


Abbildung 18: Kombinationen von Typ-Konvertierungsvarianten von AIu und AIv im ersten Durchlauf

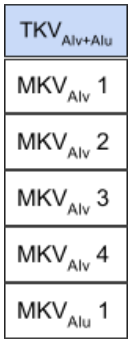


Abbildung 19: Kombinationen von Typ-Konvertierungsvarianten von AIu und AIv im zweiten Durchlauf

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:

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

3.2.3 Schritt 3: Erzeugen von benötigten Komponenten

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 zweiten Schrittes im aktuellen Durchlauf selektiert. Die daraus erzeugten Methoden-Konvertierungsvarianten werden hinsichtlich der Methoden aus dem erwarteten Interface miteinander kombiniert.

Für die erste Kombination von Typ-Konvertierungsvarianten, die Abbildung 18 zu entnehmen ist (TKV_{AIv}), können folgende Kombinationen von Methoden-Konvertierungsvarianten erzeugt werden (siehe Abbildung 20).

Abbildung 20: Kombinationen von Methoden-Konvertierungsvarianten AIv

Analog dazu wird für die zweite Kombination von Typ-Konvertierungsvarianten, die Abbildung 18 zu entnehmen ist (TKV_{AIu}), folgende Kombination von Methoden-Konvertierungsvarianten erzeugt (siehe Abbildung 21).

Ausgehend von der Kombination von Typ-Konvertierungsvarianten aus Abbildung 19 ($TKV_{AIu+AIv}$), sind in Abbildung 22 die daraus resultieren Methoden-Konvertierungsvarianten dargestellt.Zu beachten ist, dass die ersten vier Kombinationen bereits im vorherigen Durchlauf erzeugt wurden (siehe Abbildung 20) und dementsprechend auch getestet wurden.

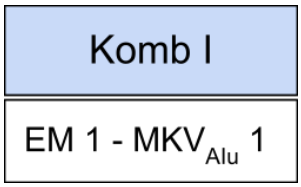


Abbildung 21: Kombinationen von Methoden-Konvertierungsvarianten AIu

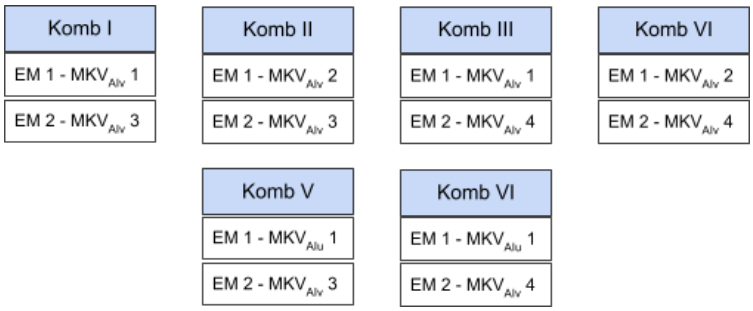


Abbildung 22: Kombinationen von Methoden-Konvertierungsvarianten AIu+AIv

Im Allgemeinen lässt sich sagen, dass die Anzahl der Kombinationen von Methoden-Konvertierungsvariante von der Anzahl der Methoden im erwarteten Interface ($|EM|$) und der Anzahl von Methoden-Konvertierungsvarianten ($|MKV|$), die aus der selektierten Kombination von Typ-Konvertierungsvarianten erzeugt werden können. Da aus einer Kombination von Methoden-Konvertierungsvarianten jeweils eine benötigte Komponente erzeugt werden kann, gilt für die Anzahl der benötigten Kom-

ponenten ($|Komb_{ben}|$) dasselbe. Im schlimmsten Fall berechnet sich die Anzahl der benötigten Komponenten wie folgt:

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

3.2.4 Schritt 4: Injizieren der benötigten Komponente

Der Setter für die Setter-Injection wird in der Testklasse über die Annotation @QueryTypeInstanceSetter ermittelt. Danach wird diese Methode auf dem Testobjekt (siehe 3.2.1) mit der benötigten Komponente als Parameter aufgerufen.

3.2.5 Schritt 5: 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.

Umgang mit kombinierten angebotenen Komponenten
Ab dem zweiten Durchlauf werden werden die benötigten Komponenten in Schritt 3 aus Kombinationen von 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 Aufrufe dieser jedoch an unterschiedliche Komponenten delegiert werden, welche auch auf unterschiedlichen Daten operieren.

Ein Beispiel hierfür wäre ein Stack, der durch das erwartete Interface Stack beschrieben. Dieses enthält eine push und eine pop Methoden mit der ein Element im Stack hinzugefügt bzw.

entfernt werden kann (siehe Abbildung 23). 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 Abbildung 24). 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 Abbildung 25).

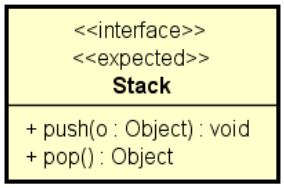


Abbildung 23: Erwartetes Interface Stack

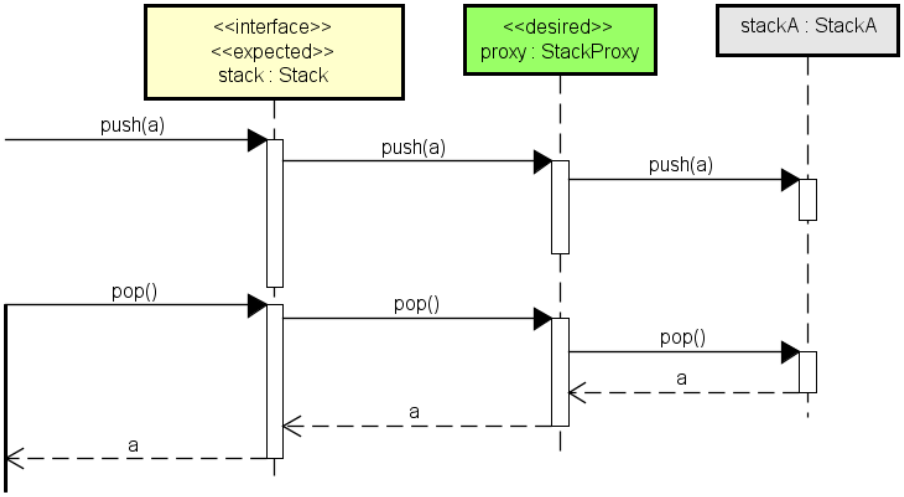


Abbildung 24: Delegation der Stack-Methoden an genau eine angebotene Komponente

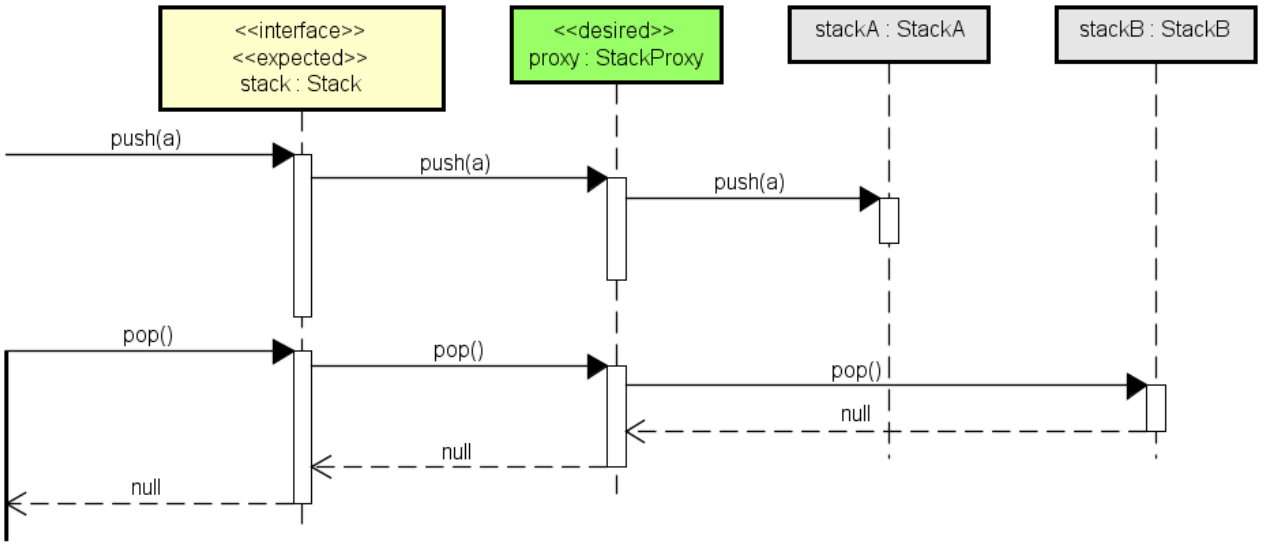


Abbildung 25: Delegation der Stack-Methoden an unterschiedliche angebotene Komponenten

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 Abbildung 24.

Listing 3: Testklasse für ein erwartetes Interfaces Stack

```
public class StackTest {

  private Stack stack;

  @QueryTypeInstanceSetter
  public void setProvider( Stack stack ) {
    this.stack = stack;
  }

  @QueryTypeTest
  public void pushPop() {
    Object a = new Object();
    stack.push( a );
    Object evalObj = stack.pop();
    assertTrue( a == evalObj );
  }

}
```

3.2.6 Schritt 6: 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 3.2.3) wiederholt. Sofern keine benötigten Komponenten mehr erzeugt werden können, ist die Suche nach einer passenden benötigten Komponente gescheitert.

Da die Suche zur Laufzeit ausgeführt wird, reicht es, wenn eine passende benötige Komponente gefunden wird. Selbst wenn es mehrere von diesen 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, jedoch rechtfertigt der dafür notwendige Aufwand, aufgrund der Vielzahl von möglichen Kombinationen (siehe 3.2.2 und 3.2.3), den daraus resultierenden Performancegewinn vermutlich nicht.

4 Heuristiken

4.1 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:

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 Ratings 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 wiederum eine Übereinstimmung verwendeter Typen fordert, um die konkreten methodendelegationsrelevanten Informationen zu erzeugen (WrappedTypeMatcher und StructuralTypeMatcher), ergibt sich der Wert des Type-Matcher-Ratings aus einer Akkumulation der Basiswerte aller verwendeten Type-Matcher.

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 untersucht.

Quantitatives Type-Matcher Rating
Das quantitative Type-Matcher Rating beschreibt, zu wie vielen der erwarteten Methoden in

der erzeugten Typ-Konvertierungsvariante methodendelegationsrelevante Informationen vorliegen. So stellt das quantitative Type-Matcher Rating also einen Prozentsatz dar.

Im Folgenden werden zwei Heuristiken vorgestellt, die auf den oben genannten Type-Matcher Ratings basieren.

4.1.1 TMR_Quant: Beachtung des quantitativen Type-Matcher Ratings

Szenario

Abbildung 26 zeigt ein Szenario, in dem ein erwartetes Interface IExpect mit zwei Methoden deklariert wurde. Auf der rechten Seite sind die beiden angebotenen Interfaces abgebildet (IOfferedA und IOfferedB), die laut den oben genannten Type-Matchern zu dem erwarteten Interface passen. Weiterhin soll in diesem Szenario davon ausgegangen werden, dass eine passende benötigte Komponente (Proxy) nur aufbauend auf den Methoden-Konvertierungsvarianten erzeugt werden kann, die auf der Basis des angebotenen Interfaces IOfferedB erzeugt werden können.

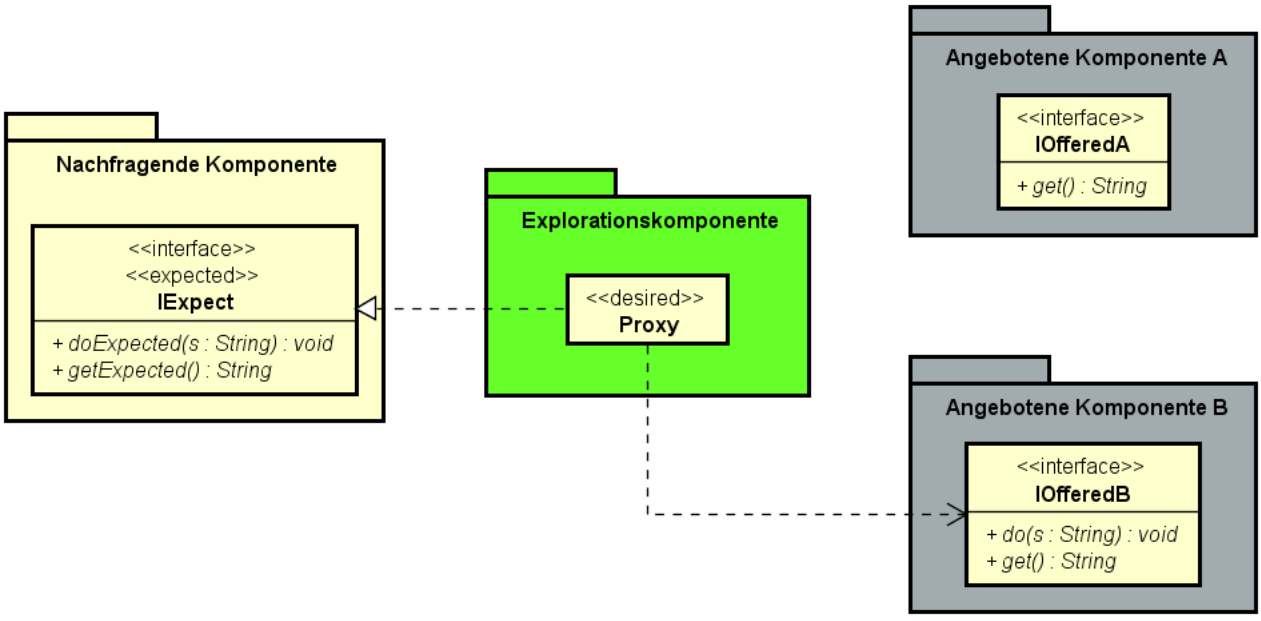


Abbildung 26: Szenario für TMR_Quant

Exploration ohne Heuristik

Der Explorationsalgorithmus würde die passende benötigte Komponente im ersten Durchlauf finden. Allerdings würde auch die Typ-Konvertierungsvariante für IOfferedA im schlimmsten Fall innerhalb des ersten Durchlaufs analysiert werden. Die Kombinationen von Typ-Konvertierungsvarianten der beiden angebotenen Interfaces, die im ersten Durchlauf des Explorationsalgorithmus erzeugt werden, sind Abbildung 27 zu entnehmen. Darauf aufbauend, würde aus diesen Kombinationen von Typ-Konvertierungsvarianten die Kombinationen von Methoden-Konvertiertungsvarianten erzeugt werden, welche Abbildung 28 zu entnehmen sind.

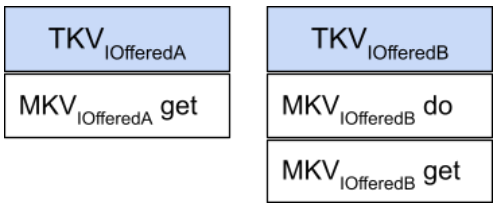


Abbildung 27: Typ-Konvertierungsvarianten dem Szenario zu TMR_Quant

Abbildung 28: Methoden-Konvertierungsvarianten dem Szenario zu TMR_Quant

Ableitung der Heuristik

Hierbei fällt auf, dass das Erzeugen der Methoden-Konvertierungsvarianten für IOfferedA unnötig war, weil nur für eine der erwarteten Methoden eine Methoden-Konvertierungsvariante erzeugt werden konnte. Dies kann durch die Beachtung des quantitativen Type-Matcher Ratings im ersten Durchlauf des Explorationsalgorithmus verhindert werden.

Hierzu werden im Schritt 2 der 2. Stufe des Explorationsalgorithmus (siehe 3.2.2) nur die Typ-Konvertierungsvarianten als Ergebnis ermittelt, die ein quantitatives Type-Matcher Rating von 100% aufweisen. Damit wird bezogen auf das Szeanrio nur die Typ-Konvertierungsvariante des angebotenen Interfaces IOfferedB weiter analysiert.

4.1.2 TMR_Qual: Beachtung des qualitativen Type-Matcher Ratings

Szenario

Abbildung 29 zeigt ein Szenario, in dem ein erwartetes Interface IExpect mit einer Methode de-

klariert wurde. Die Deklaration erfolgte unter der Verwendung bestehender allgemeiner Typen innerhalb des gesamten Systems, die in dem unteren Bereich der Abbildung dargestellt sind. Dabei gibt es zwei Typen, die in einer Vererbungsbeziehung stehen (Common und Specific), und einen Wrapper-Typen, der ein Attribut vom Typ Common enthält. Zusätzlich sind auf der rechten Seite die angebotenen Interfaces abgebildet (IOfferedA und IOfferedB), die laut den oben genannten Type-Matchern zu dem erwarteten Interface passen. Auch diese verwenden die allgemeinen Typen. Für dieses Szenario ist davon auszugehen, dass eine passende benötigte Komponente (Proxy) nur aufbauend auf den Methoden-Konvertierungsvarianten erzeugt werden kann, die auf der Basis des angebotenen Interfaces IOfferedB erzeugt werden können.

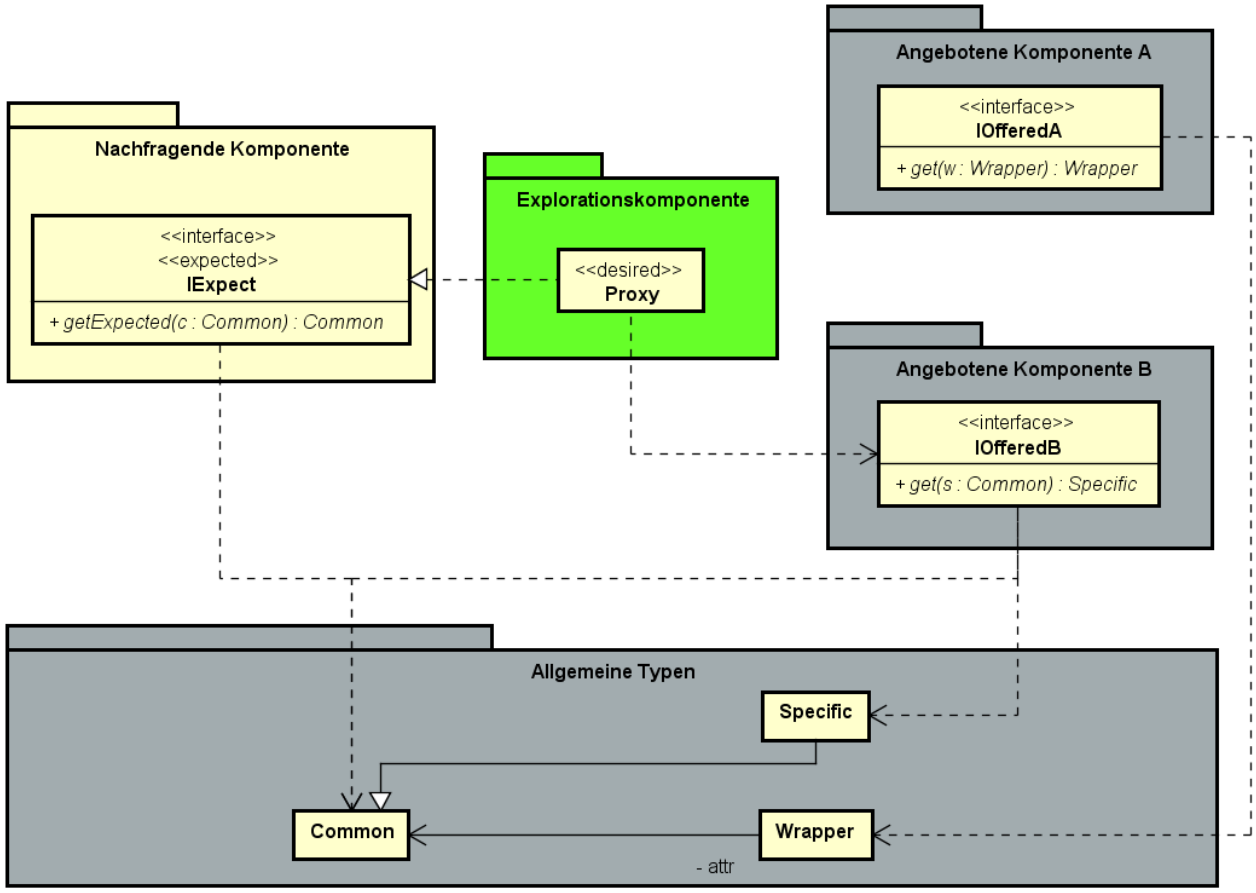


Abbildung 29: Szenario für TMR_Qual

Exploration ohne Heuristik

Der Explorationsalgorithmus würde die passende benötigte Komponente im ersten Durchlauf finden. Hierbei werden zuerst die Typ-Konvertierungsvarianten der beiden angebotenen Interfaces erzeugt, welche Abbildung 30 zu entnehmen sind. Abbildung 31 zeigt die im darauffolgenden Schritt des Explorationsalgorithmus (siehe 3.2.3) erzeugten Kombinationen von Methoden-Konvertierungsvarianten. Zu erkennen ist, dass auch die Kombinationen von Methoden-Konvertierungsvara von IOfferedA erzeugt wurden und demnach im schlimmsten Fall auch weiter analysiert werden.

Abbildung 30: Typ-Konvertierungsvarianten dem Szenario zu TMR_Qual

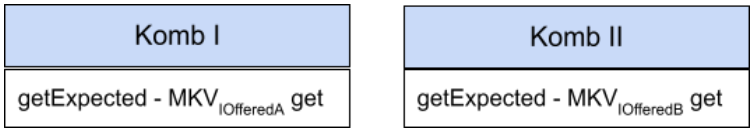


Abbildung 31: Methoden-Konvertierungsvarianten dem Szenario zu TMR_Qual

Ableitung der Heuristik

Bei der Betrachtung der Methode, die von der nachfragenden Komponente erwartet und von den angebotenen Komponenten B bereitgestellt wird, fällt auf, dass die Typen des Parameter und der Rückgabewerte der jeweiligen Methoden in einer Vererbungsbeziehung stehen. Dadurch ist eine Delegation der Form IExpect.getExpected ⇒ IOfferedB.get aufgrund des liskovschen Substitutionsprinzip ohne weitere Konvertierung der Parameter- und Rückgabe-Typen möglich. Eine Delegation der Methoden aus IExpect an IOfferedA ist hingegen weitaus komplizierter (siehe 3.1.5). In Bezug auf das oben beschrieben Type-Matcher Rating bedeutet das, dass das qualitative Type-Matcher Rating der Typ-Konvertierungsvariante von IOfferedB geringer ist, als das der Typ-Konvertierungsvariante von IOfferedA.

Für die Heuristik TMR_Qual kann demnach abgeleitet werden, dass die Kombinationen von Methoden-Konvertierungsvarianten in Schritt 3 der 2. Stufe des Explorationsalgorithmus (siehe 3.2.3) zuerst aus denjenigen Kombinationen von Typ-Konvertierungsvarianten mit den niedrigsten qualitativen Type-Matcher Rating erzeugt und analysiert werden sollten.

4.2 Testergebnis basierte Heuristiken

Diese Heuristiken werden auf der Basis des TestResult-Objektes, welches bei der semantischen Evaluation (2. Stufe, siehe 3.2) bei der Durchführung der Tests (Schritt 5, siehe 3.2.5) 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 3.2.2) nicht mehr oder bevorzugt verwendet werden.

4.2.1 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.

4.2.2 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 (SigMaGlue-Exception) 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 markPivotMethodCallExecuted() 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.

4.2.3 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 auf-

gerufen 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 Koordination im Explorationsalgorithmus

Der Einsatz der Heuristiken muss bei der Suche koordiniert werden. Der Explorationsalgorithmus ist so aufgebaut, dass im 2. Schritt der 2. Stufe (siehe 3.2.2) die Heuristiken zum Einsatz kommen.

Begonnen wird mit der Heuristik TMR_Quant, sodass zuerst alle möglichen Kombinationen von Methoden-Konvertierungsvarianten ermittelt werden, die aus einer einzelnen Typ-Konvertierungsvariante stammen. Sind die möglichen Kombinationen von Methoden-Konvertierungsvarianten ausgeschöpft, wird der Prozess mit der Ermittlung aller möglichen Kombinationen von Methoden-Konvertierungsvarianten, die aus 2 Typ-Konvertierungsvarianten stammen, wiederholt. Die Anzahl der zu kombinierenden Typ-Konvertierungsvarianten wird somit jedes mal erhört, wenn die bereits ermittelten Kombinationen von Methoden-Konvertierungsvarianten ausgeschöpft 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 für die Anwendung dieser Heuristiken bereist Testergebnisse vorliegen müssen. PREV_PASSED sorgt nochmals für eine Umsortierung der Typ-Konvertierungsvarianten, die bei der Ermittlung der Kombinationen von Methoden-Konvertierungsvariante verwendet werden, sodass die bevorzugten Typ-Konvertierungsvarianten zuerst verwendet werden.

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

5 Evaluierung

Die Evaluierung erfolgt innerhalb von Systemen, in denen mindestens 889 angebotene Interfaces existieren. Es wird zwischen einem Test-System und einem Heiß-System unterschieden.

Das Test-System wurde vorrangig für die Evaluation der Type-Matcher Rating basierten Heuristiken verwendet, da für diese Heuristiken keine Implementierungen der angebotenen Interfaces vorliegen müssen.

Das Heiß-System wurde vorrangig für die Evaluation der testergebnis basierten Heuristiken verwendet, da hier zu jedem der 889 angebotenen Interfaces eine Implementierung existiert. Die angebotenen Komponenten wurden im Heiß-System als Java Enterprise Beans umgesetzt.

Darstellung der Evaluationsergebnisse
Die Evaluationsergebnisse werden in der Form von Vier-Felder-Tafeln dargestellt (Beispiel siehe Tabelle 1). Für jedes erwartete Interface wird eine Vier-Felder-Tafel für jeden Durchlauf des Explorationsalgorithmus aufgezeigt. Aus der jeweiligen Tafel geht hervor, wie viele Kombinationen von Methoden-Konvertierungsvarianten aus den Kombinationen der ermittelten Typ-Konvertierungsvarianten innerhalb des Durchlaufs erzeugt werden könnten. Die Nummer des Durchlaufs wird in der oberen rechten Ecke der Tafel abgebildet. In der Spalte positiv ist die Anzahl der Kombinationen von Methoden-Konvertierungsvarianten verzeichnet, die innerhalb des Durchlaufs tatsächlich erzeugt wurden. Die Zahl in der Spalte "negativ" drückt hingegen aus, wie viele der Kombinationen aufgrund bestimmter Kriterien (bzw. Heuristiken) gar nicht erst erzeugt wurden. Die Zeile "falsch" beschreibt die Anzahl der relevanten Kombinationen, aus denen benötigte Komponenten erzeugt werden, welche die semantischen Tests nicht bestehen. Dementsprechend stellt die Zeile "richtig" die Anzahl der Kombinationen dar, aus denen sich benötigte Komponenten erzeugen lassen, welche die semantischen Test bestehen. Der Fall, in dem eine Kombination nicht erzeugt wurde, aber dennoch für die Erstellung einer benötigten Komponente genutzt wurde und die semantischen Tests besteht, (negativ und richtig) kann nicht auftreten.

Für die Anzahl der zu kombinierenden Methoden-Konvertierungsvarianten MK wird der höchste

mögliche Wert angenommen. Dieser ist von der Anzahl der angebotenen Methoden am sowie der Anzahl der erwarteten Methoden em abhängig und wird wie folgt berechnet:

$$MK = \frac{am!}{(am - em)! * em!}$$

Die Anzahl der angebotenen Methoden am ist wiederum abhängig von den angebotenen Interfaces deren Typ-Konvertierungsvarianten im jeweiligen Durchlauf miteinander kombiniert wurden. Die Anzahl der Kombinationen von Typ-Konvertierungsvarianten innerhalb des Durchlaufs sei mit TK beschrieben. Der Wert für TK berechnet sich in Abhängigkeit von der Nummer des Durchlaufs d und der Anzahl der strukturell passenden angebotenen Interfaces n (siehe auch Abschnitt Explorationskomponente, 2. Stufe, 2. Kombination von Typ-Konvertierungsvarianten).

$$TK = \frac{n!}{(n - d)! * d!}$$

Da die Anzahl der angebotenen Methoden von System zu System schwanken kann, sei die Funktion $am(TK)$ eine näherungsweise Darstellung von am, in Abhängigkeit von der Anzahl der kombinierten Typ-Konvertierungsvarianten TK.

Da durch die Heuristiken letztendlich Methoden-Konvertierungsvarianten aus der Suche herausfallen, wird die Anzahl der entsprechenden Methoden-Konvertierungsvarianten in dem jeweiligen Feld der Vier-Felder-Tafeln als Funktion $mk(TK)$ dargestellt, die wie folgt definiert wird:

$$mk(TK) = \frac{am(TK)!}{(am(TK) - em)! * em!}$$

Tabelle 1 zeigt ein Beispiel für eine solche Vier-Felder-Tafel, in der die Ergebnisse des 1. Durchlauf des Explorationsalgorithmus dargestellt sind. Dabei wurden Methoden-Konvertierungsvarianten aus 10 Kombinationen von Typ-Konvertierungsvarianten erzeugt. Den Methoden-Konvertierungsvarianten, die nicht beachtet wurden, lagen insgesamt 20 Typ-Konvertierungsvarianten zugrunde. Weiterhin zeigt das Beispiel, dass es eine Kombination von Methoden-Konvertierungsvarianten gibt, aus der eine passende benötigte Komponente erzeugt werden konnte.

1	**positiv**	**negativ**
falsch	$mk(10)$	$mk(20)$
richtig	1	0

Tabelle 1: Beispiel: Vier-Felder-Tafel

5.1 Test-System

Wie bereits erwähnt werden im Test-System die 889 angebotenen Interfaces verwendet, die auch im Heiß-System verwendet werden. Darüber hinaus wurden noch 6 weitere angebotene Interfaces dem Test-System hinzugefügt, um bestimmte Konstellationen geziel zu evaluieren. Die 6 erwarteten Interfaces wurden wie folgt deklariert (siehe Abbildungen 32-37).

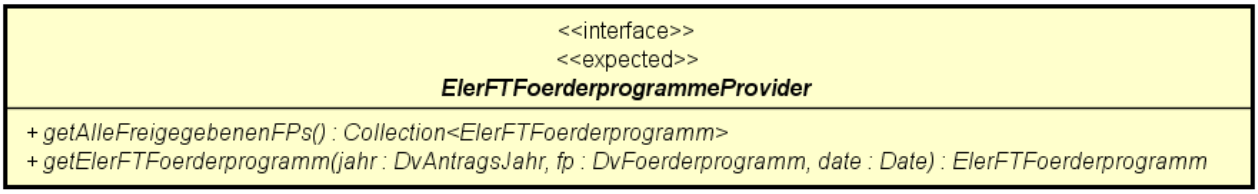


Abbildung 32: Erwartetes Interface: ElerFTFoerderprogrammeProvider

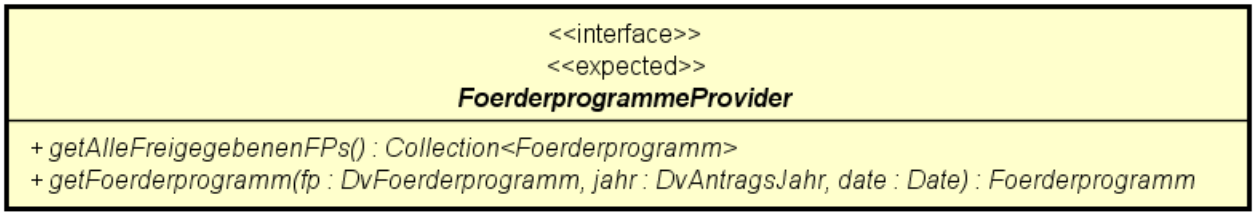


Abbildung 33: Erwartetes Interface: FoerderprogrammeProvider

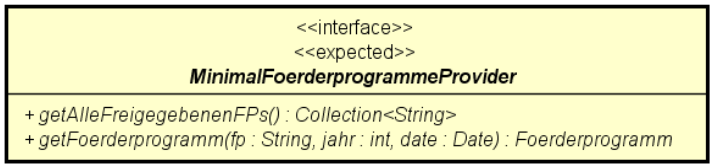


Abbildung 34: Erwartetes Interface: MinimalFoerderprogrammeProvider

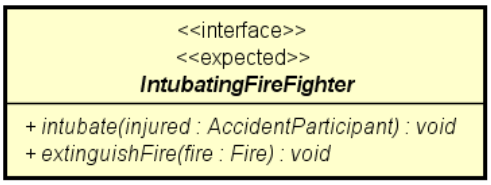


Abbildung 35: Erwartetes Interface: IntubatingFireFighter

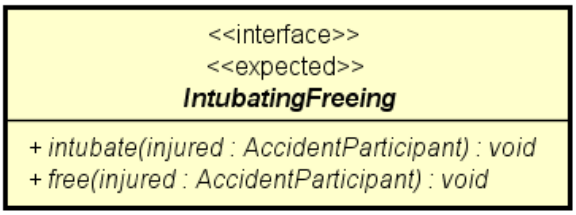


Abbildung 36: Erwartetes Interface: IntubatingFreeing

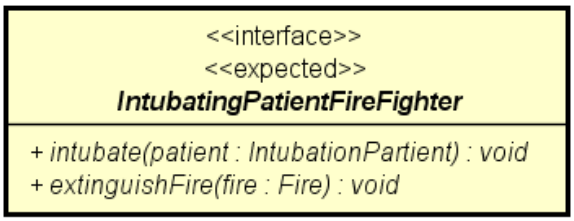


Abbildung 37: Erwartetes Interface: IntubatingPatientFireFighter

Im weiteren Verlauf werden die oben beschriebenen Interfaces durch die Kürzel in Tabelle 2 identifiziert.

erwartetes Inferface	**Kürzel**
ElerFTFoerderprogrammeProvider	TEI1
FoerderprogrammeProvider	TEI2
MinimalFoerderprogrammeProvider	TEI3
IntubatingFireFighter	TEI4
IntubatingFreeing	TEI5
IntubatingPatientFireFighter	TEI6

Tabelle 2: Kürzel der erwarteten Interfaces

5.1.1 Type-Matcher Rating basierte Heuristiken

Ausgangspunkt

Für ein erwarteten Interfaces konnten mehrere angebotene Interfaces gefunden werden, die eine strukturelle Übereinstimmung aufwiesen. Tabelle 3 zeigt die Anzahl der strukturell übereinstimmenden angebotenen Interfaces je erwartetes Interface.

erwartetes Interface	**Anzahl strukturell übereinstimmender angebotener Interfaces**
TEI1	169
TEI2	179
TEI3	187
TEI4	62
TEI5	60
TEI6	33

Tabelle 3: Anzahl strukturell übereinstimmender angebotener Interfaces je erwartetes Interfaces

Die Tabellen 4-12 zeigen die Vier-Felder-Tafeln, in denen die Ergebnisse der benötigten Durchläufe des Explorationsalgorithmus für jedes der erwarteten Interfaces aus Tabelle 3. Dabei wurden keine Heuristiken verwendet. Somit stellt dies den Ausgangspunkt für die weitere Evaluation dar.

1	positiv	negativ
falsch	$mk(169)$	0
richtig	1	0

Tabelle 4: Ausgangspunkt Test-System TMR für TEI1

1	positiv	negativ
falsch	$mk(179)$	0
richtig	1	0

Tabelle 5: Ausgangspunkt Test-System TMR für TEI2

1	positiv	negativ
falsch	$mk(187)$	0
richtig	1	0

Tabelle 6: Ausgangspunkt Test-System TMR für TEI3

1	positiv	negativ
falsch	$mk(62)$	0
richtig	0	0

Tabelle 7: Ausgangspunkt Test-System TMR für TEI4 1. Durchlauf

1	positiv	negativ
falsch	$mk(60)$	0
richtig	0	0

Tabelle 8: Ausgangspunkt Test-System TMR für TEI5 1. Durchlauf

1	positiv	negativ
falsch	$mk(33)$	0
richtig	0	0

Tabelle 9: Ausgangspunkt Test-System TMR für TEI6 1. Durchlauf

2	positiv	negativ
falsch	$mk(1891)$	0
richtig	1	0

Tabelle 10: Ausgangspunkt Test-System TMR für TEI4 2. Durchlauf

2	positiv	negativ
falsch	$mk(1770)$	0
richtig	1	0

Tabelle 11: Ausgangspunkt Test-System TMR für TEI5 2. Durchlauf

2	positiv	negativ
falsch	$mk(528)$	0
richtig	1	0

Tabelle 12: Ausgangspunkt Test-System TMR für TEI6 2. Durchlauf

Für die Interfaces TEI4 - TEI6 werden zwei Durchläufe benötigt, da die semantischen Test

nur von einer benötigten Komponente bestanden werden, die aus einer Kombination zweier Typ-Konvertierungsvarianten erzeugt wurde.

Ergebnisse TMR_Quant

Durch die Verwendung der Heuristik TMR_Quant kann für die ersten 3 erwarteten Interfaces (TEI1 - TEI3) eine Verbesserung erzielt werden. Der Grund dafür ist, dass die benötigte Komponente, die letztendlich alle semantischen Tests besteht auf der Basis genau einer Typ-Konvertierungsvariante erzeugt wurde. Damit benötigt der Explorationsalgorithmus lediglich einen Durchlauf. TMR_Quant sorgt dennoch dafür, dass die erzeugten Kombinationen von Typ-Konvertierungsvarianten im 2. Schritt reduziert werden, da solche, die ein quantitatives Type-Matcher Rating von ¡ 100% aufweisen nicht in die Ergebnismenge des 2. Schrittes einfließen. Die unten aufgeführten Tafeln zeigen die Auswirkung auf die ersten drei erwarteten Interfaces.

1	**positiv**	**negativ**
falsch	$mk(29)$	$mk(140)$
richtig	1	0

Tabelle 13: TMR_Quant Test-System TMR für TEI1

1	**positiv**	**negativ**
falsch	$mk(22)$	$mk(157)$
richtig	1	0

Tabelle 14: TMR_Quant Test-System TMR für TEI2

1	**positiv**	**negativ**
falsch	$mk(24)$	$mk(163)$
richtig	1	0

Tabelle 15: TMR_Quant Test-System TMR für TEI3

Für die anderen erwarteten Interfaces (TEI4 - TEI6) kann durch diese Heuristik höchstens für den ersten Durchlauf eine eine Verbesserung erzielen. Die unteren Tafeln zeigen, dass sich diese Verbesserung nur auf die Ergebnisse bzgl. der erwarteten Interfaces TEI4 und TEI6 auswirkt.

1	**positiv**	**negativ**
falsch	$mk(30)$	$mk(32)$
richtig	0	0

Tabelle 16: TMR_Quant Test-System TMR für TEI4

1	**positiv**	**negativ**
falsch	$mk(30)$	0
richtig	0	0

Tabelle 17: TMR_Quant Test-System TMR für TEI5

1	**positiv**	**negativ**
falsch	$mk(31)$	$mk(2)$
richtig	0	0

Tabelle 18: TMR_Quant Test-System TMR für TEI6

Ergebnisse TMR_Qual

Für die Heuristik TMR_Qual gibt es drei Aspekte, deren Konfiguration zu unterschiedlichen Ergebnissen führen kann (siehe auch 4.1.2):

Auswahl der Basiswerte

Die Basiswerte wurden bei den Untersuchungen konstant gelassen und sind der Tabelle 19 zu entnehmen.

Type-Matcher	Basiswert
ExactTypeMatcher	100
ExactTypeMatcher	200
WrappedTypeMatcher	300
StructuralTypeMatcher	400

Tabelle 19: Type-Matcher mit Basiswerten

Auswahl der Akkumulationsverfahren

Das Akkumulationsverfahren für das qualitative Type-Matcher Rating einer Typ-Konvertierungsvariante TMR_{TK} ist von dem Type-Matcher Rating der verwendeten Type-Matcher abhängig. Das Akkumulationsverfahren für das qualitative Type-Matcher Rating einer Methoden-Konvertierungsvariante TMR_{MK} ist von dem qualitativen Type-Matcher Rating der verwendeten Type-Matcher für den Rückgabe- und den Parametertypen der Methode abhängig abhängig. Somit kann das qualitative Type-Matcher Rating als Funktion von einer Typ- bzw. Methoden-Konvertierungsvariante $tmr_{Qual}(v)$ beschrieben werden. Das Type-Matcher Rating der verwendeten Type-Matcher wird als Funktion $tmr_{Base}(m)$ beschrieben. Dabei stellt m den jeweiligen Type-Matcher dar. Die Funktion $tmr_{Base}(m)$ ist durch die Tabelle 19 definiert.

Für einen Menge von Type-Matcher $m_1, m_2, ..., m_i$, die zur Erzeugung einer Typ-Konvertierungsvariante bzw. Methoden-Konvertierungsvariante v verwendet wurden, werden folgende Akkumulationsverfahren für das Type-Matcher Rating der Typ-Konvertierungsvariante bzw. Methoden-Konvertierungsvariante im weiteren Verlauf evaluiert:

1. Wahl des Durchschnitts

$$tmr_{Qual}(v) = \frac{\sum_{n=1}^{i} tmr_{Base}(m_n)}{i}$$

2. Wahl des Maximums

$$tmr_{Qual}(v) = max(tmr_{Base}(m_1), ..., tmr_{Base}(m_i))$$

3. Wahl des Minimums

$$tmr_{Qual}(v) = min(tmr_{Base}(m_1), ..., tmr_{Base}(m_i))$$

4. Wahl des Durchschnitts aus Minimum und Maximum

$$tmr_{Qual}(v) = \frac{min(tmr_{Base}(m_1), ..., tmr_{Base}(m_i)) + max(tmr_{Base}(m_1), ..., tmr_{Base}(m_i))}{2}$$

Die folgenden Abschnitte stellen eine Auswahl der Ergebnisse hinsichtlich der Kombinationen der oben genannten Akkumulationsverfahren dar. Die Ergebnisse von Kombinationen, die nicht dargestellt wurden, sind mit den Ergebnissen einer der dargestellten Kombinationen gleichzusetzen. An entsprechender Stelle wird darauf verwiesen.

An den Überschriften der folgenden Abschnitte ist abzulesen, welche Akkumulationsverfahren miteinander kombiniert wurden. Dabei haben die Überschriften die Form "Typ: T Methoden: M". "T" steht für die Nummer des Akkumulationsverfahrens, welches für die Typ-Konvertierungsvarianten verwendet wurde. "M" steht für die Nummer des Akkumulationsverfahrens, welches für die Methoden-Konvertierungsvarianten zum Einsatz kam. Typ: 1 Methoden: 2

1	**positiv**	**negativ**
falsch	$mk(48)$	$mk(121)$
richtig	1	0

Tabelle 20: TMR_Qual Test-System TMR für TEI1 mit 1-2

1	**positiv**	**negativ**
falsch	$mk(47)$	$mk(132)$
richtig	1	0

Tabelle 21: TMR_Qual Test-System TMR für TEI2 mit 1-2

1	**positiv**	**negativ**
falsch	$mk(46)$	$mk(141)$
richtig	1	0

Tabelle 22: TMR_Qual Test-System TMR für TEI3 mit 1-2

1	**positiv**	**negativ**
falsch	$mk(62)$	0
richtig	0	0

Tabelle 23: TMR_Qual Test-System TMR für TEI4 mit 1-2 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(60)$	0
richtig	0	0

Tabelle 24: TMR_Qual Test-System TMR für TEI5 mit 1-2 1. Durchlauf

1	positiv	negativ
falsch	$mk(33)$	0
richtig	1	0

Tabelle 25: TMR_Qual Test-System TMR für TEI6 mit 1-2 1. Durchlauf

2	positiv	negativ
falsch	$mk(1)$	$mk(1890)$
richtig	1	0

Tabelle 26: TMR_Qual Test-System TMR für TEI4 mit 1-2 2. Durchlauf

2	positiv	negativ
falsch	$mk(1)$	$mk(1769)$
richtig	1	0

Tabelle 27: TMR_Qual Test-System TMR für TEI5 mit 1-2 2. Durchlauf

2	positiv	negativ
falsch	$mk(1)$	$mk(527)$
richtig	1	0

Tabelle 28: TMR_Qual Test-System TMR für TEI6 mit 1-2 2. Durchlauf

Typ: 3 Methoden: 2

1	positiv	negativ
falsch	$mk(49)$	$mk(120)$
richtig	1	0

Tabelle 29: TMR_Qual Test-System TMR für TEI1 mit 3-2

1	positiv	negativ
falsch	$mk(49)$	$mk(130)$
richtig	1	0

Tabelle 30: TMR_Qual Test-System TMR für TEI2 mit 3-2

1	positiv	negativ
falsch	$mk(48)$	$mk(139)$
richtig	1	0

Tabelle 31: TMR_Qual Test-System TMR für TEI3 mit 3-2

1	**positiv**	**negativ**
falsch	$mk(62)$	0
richtig	0	0

Tabelle 32: TMR_Qual Test-System TMR für TEI4 mit 3-2 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(60)$	0
richtig	0	0

Tabelle 33: TMR_Qual Test-System TMR für TEI5 mit 3-2 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(33)$	0
richtig	1	0

Tabelle 34: TMR_Qual Test-System TMR für TEI6 mit 3-2 1. Durchlauf

2	**positiv**	**negativ**
falsch	$mk(1)$	$mk(1890)$
richtig	1	0

Tabelle 35: TMR_Qual Test-System TMR für TEI4 mit 3-2 2. Durchlauf

2	**positiv**	**negativ**
falsch	$mk(1)$	$mk(1769)$
richtig	1	0

Tabelle 36: TMR_Qual Test-System TMR für TEI5 mit 3-2 2. Durchlauf

2	**positiv**	**negativ**
falsch	$mk(1)$	$mk(527)$
richtig	1	0

Tabelle 37: TMR_Qual Test-System TMR für TEI6 mit 3-2 2. Durchlauf

Typ: 4 Methoden: 3

1	positiv	negativ
falsch	$mk(52)$	$mk(117)$
richtig	1	0

Tabelle 38: TMR_Qual Test-System TMR für TEI1 mit 4-3

1	positiv	negativ
falsch	$mk(62)$	$mk(117)$
richtig	1	0

Tabelle 39: TMR_Qual Test-System TMR für TEI2 mit 4-3

1	positiv	negativ
falsch	$mk(62)$	$mk(125)$
richtig	1	0

Tabelle 40: TMR_Qual Test-System TMR für TEI3 mit 4-3

1	positiv	negativ
falsch	$mk(62)$	0
richtig	0	0

Tabelle 41: TMR_Qual Test-System TMR für TEI4 mit 4-3 1. Durchlauf

1	positiv	negativ
falsch	$mk(60)$	0
richtig	0	0

Tabelle 42: TMR_Qual Test-System TMR für TEI5 mit 4-3 1. Durchlauf

1	positiv	negativ
falsch	$mk(33)$	0
richtig	1	0

Tabelle 43: TMR_Qual Test-System TMR für TEI6 mit 4-3 1. Durchlauf

2	positiv	negativ
falsch	$mk(1891)$	0
richtig	1	0

Tabelle 44: TMR_Qual Test-System TMR für TEI4 mit 4-3 2. Durchlauf

2	positiv	negativ
falsch	$mk(1770)$	0
richtig	1	0

Tabelle 45: TMR_Qual Test-System TMR für TEI5 mit 4-3 2. Durchlauf

2	positiv	negativ
falsch	$mk(528)$	0
richtig	1	0

Tabelle 46: TMR_Qual Test-System TMR für TEI6 mit 4-3 2. Durchlauf

Die Tabelle 47 zeigt durch die Markierung mit einem “x”, welche Kombinationen der Akkumulationsverfahren hinsichtlich der Testergebnisse mit denen gleichzusetzen sind, die oben ausführlich aufgeführt wurden. Die Kombinationen werden in der Tabelle ähnlich wie in den vorherigen Überschriften beschrieben. Die Notation “1-4” beschreibt die Kombination des 1. Akkumulationsverfahrens für die Typ-Konvertierungsvarianten und den 4. Akkumulationsverfahrens für die Methoden-Konvertierungsvarianten.

Kombination	**1-2**	**3-2**	**4-3**
1-1	x		
1-3			x
1-4	x		
2-1	x		
2-2	x		
2-3			x
2-4	x		
3-1		x	
3-3			x
3-4		x	
4-1	x		
4-2	x		
4-4	x		

Tabelle 47: Kombinationen von Akkumulationsverfahren mit gleichen Ergebnissen

Aus diesen Ergebnissen lässt sich folgendes ableiten:

1. Das Akkumulationsverfahren Nummer 3. (Minimum) führt sowohl für die Typ- und Methoden-Konvertierungsvarianten zu schlechteren Ergebnissen als die anderen drei Akkumulationsverfahren. Es sollte daher für die Heuristik TMR_Quant nicht verwendet werden.

2. Die Ergebnisse von 1-2 und 3-2 unterscheiden sich nur geringfügig, obwohl bei 3-2 das Akkumulationsverfahren Nummer 3. zum Einsatz kam. Dies konnte auch bei anderen Kombinationen festgestellt werden, bei denen das 3. Akkumulationsverfahren für die Akkumulation des Type-Matcher Ratings der Typ-Konvertierungsvariante verwendet wurde. Das lässt vermuten, dass die Beachtung des Type-Matcher Ratings einer ganzen

Typ-Konvertierungsvariante weitgehend unerheblich für die Heuristik TMR_Quant ist. Dies ist jedoch darauf zurückzuführen, dass das Type-Matcher Rating je Methoden-Konvertierungsvariante die Parameter für die Ermittlung des Type-Matcher Ratings einer Typ-Konvertierungsvariante darstellen.

3. An den Ergebnissen zu den erwarteten Interfaces TEI4-TEI6 ist zu erkennen, dass die Heuristik TMR_Quant keinen Einfluss auf den 1. Durchlauf hat. Daraus kann geschlussfolgert werden, dass die Heuristik nur in dem Durchlauf einen Gewinn bringt, in dem auch eine passende benötigte Komponente gefunden werden kann.

Aufgrund der Ergebnisse stehen für die weitere Verwendung der Heuristik TMR_Qual mehrere Kombinationen von Akkumulationsverfahren zur Auswahl. Die Entscheidung fällt aufgrund der etwas geringeren Komplexität auf die Kombination 1-2.

TMR_Quant und TMR_Qual in Kombination

Bei der Kombination der beiden Heuristiken TMR_Quant und TMR_Qual ist vor allem für die erwarteten Interfaces TEI4-TEI6 zu erwarten, dass ein gegenseitiger positiver Einfluss der Heuristiken zu erkennen ist. Der Grund dafür ist, dass die Heuristik TMR_Qual keinen Einfluss auf den ersten Durchlauf des Explorationsalgorithmus für diese erwarteten Interfaces hat, die Heuristik TMR_Quant hingegen schon. Die Tabellen 48-56 zeigen wiederum die bekannten Vier-Felder-Tafeln für den jeweiligen Durchlauf und dem jeweiligen erwarteten Interface.

1	**positiv**	**negativ**
falsch	$mk(2)$	$mk(167)$
richtig	1	0

Tabelle 48: TMR_Quant + TMR_Qual Test-System TMR für TEI1

1	**positiv**	**negativ**
falsch	$mk(2)$	$mk(177)$
richtig	1	0

Tabelle 49: TMR_Quant + TMR_Qual Test-System TMR für TEI2

1	**positiv**	**negativ**
falsch	$mk(1)$	$mk(186)$
richtig	1	0

Tabelle 50: TMR_Quant + TMR_Qual Test-System TMR für TEI3

1	**positiv**	**negativ**
falsch	$mk(30)$	$mk(32)$
richtig	0	0

Tabelle 51: TMR_Quant + TMR_Qual Test-System TMR für TEI4 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(60)$	0
richtig	0	0

Tabelle 52: TMR_Quant + TMR_Qual Test-System TMR für TEI5 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(31)$	$mk(2)$
richtig	0	0

Tabelle 53: TMR_Quant + TMR_Qual Test-System TMR für TEI6 1. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(1)$	$mk(1890)$
richtig	1	0

Tabelle 54: TMR_Quant + TMR_Qual Test-System TMR für TEI4 2. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(1)$	$mk(1769)$
richtig	1	0

Tabelle 55: TMR_Quant + TMR_Qual Test-System TMR für TEI5 2. Durchlauf

1	**positiv**	**negativ**
falsch	$mk(1)$	$mk(527)$
richtig	1	0

Tabelle 56: TMR_Quant + TMR_Qual Test-System TMR für TEI6 2. Durchlauf

Wie an diesen Ergebnissen zu erkennen ist, wird der Explorationsalgorithmus für eine Suche nach einer passenden benötigten Komponente für TEI1 und TEI2 lediglich für zwei angebotene Interfaces bzw. Typ-Konvertierungsvarianten durchlaufen. In Bezug auf TEI3 ist es sogar nur noch eine Typ-Konvertierungsvariante.

Bei der Betrachtung der Ergebnisse für die erwarteten Interfaces TEI4-TEI6 zeigt sich gut, wie sich die beiden Heuristiken gegenseitig ergänzen. So wirkt die Heuritik TMR_Quant grundsätzlich nur auf den ersten Durchlauf des Explorationsalgorithmus aus. Die Heuristik TMR_Qual hingegen erweist ihre Stärke erst in dem Durchlauf, in dem auch eine passende benötigte Komponente gefunden wird.

Im Allgemeinen kann festgehalten werden, dass die passenden benötigten Komponenten trotz der Kombination der beiden Heuristiken gefunden werden konnten. Die Reduktion der notwendigen Durchläufe des Explorationsalgorithmus ist jedoch hauptsächlich auf die Heuristik TMR_Qual zurückzuführen.

5.2 Heiß-System

A Beispiel-Implementierungen für die Matcher

Um die Beispiel-Implementierungen der Matcher nachvollziehen zu können, ist es notwendig die Implementierung der darin verwendeten Klassen aufzuzeigen. Daher sind diese in Listings 4-6 aufgeführt. Dabei handelt es sich zum einen um die Implementierungen der Klassen, die in den Szenarien der Abschnitte 3.1.2 - **??** beschrieben wurden. Zum anderen handelt es sich um Implementierung weiterer Klassen, die in Szenarien verwendet werden, welche in den folgenden Abschnitten aufgeführt werden. Um einen Überblick zu gewährleisten, zeigt Abbildung 38 alle Typen auf, die in den Szenarien verwendet werden.

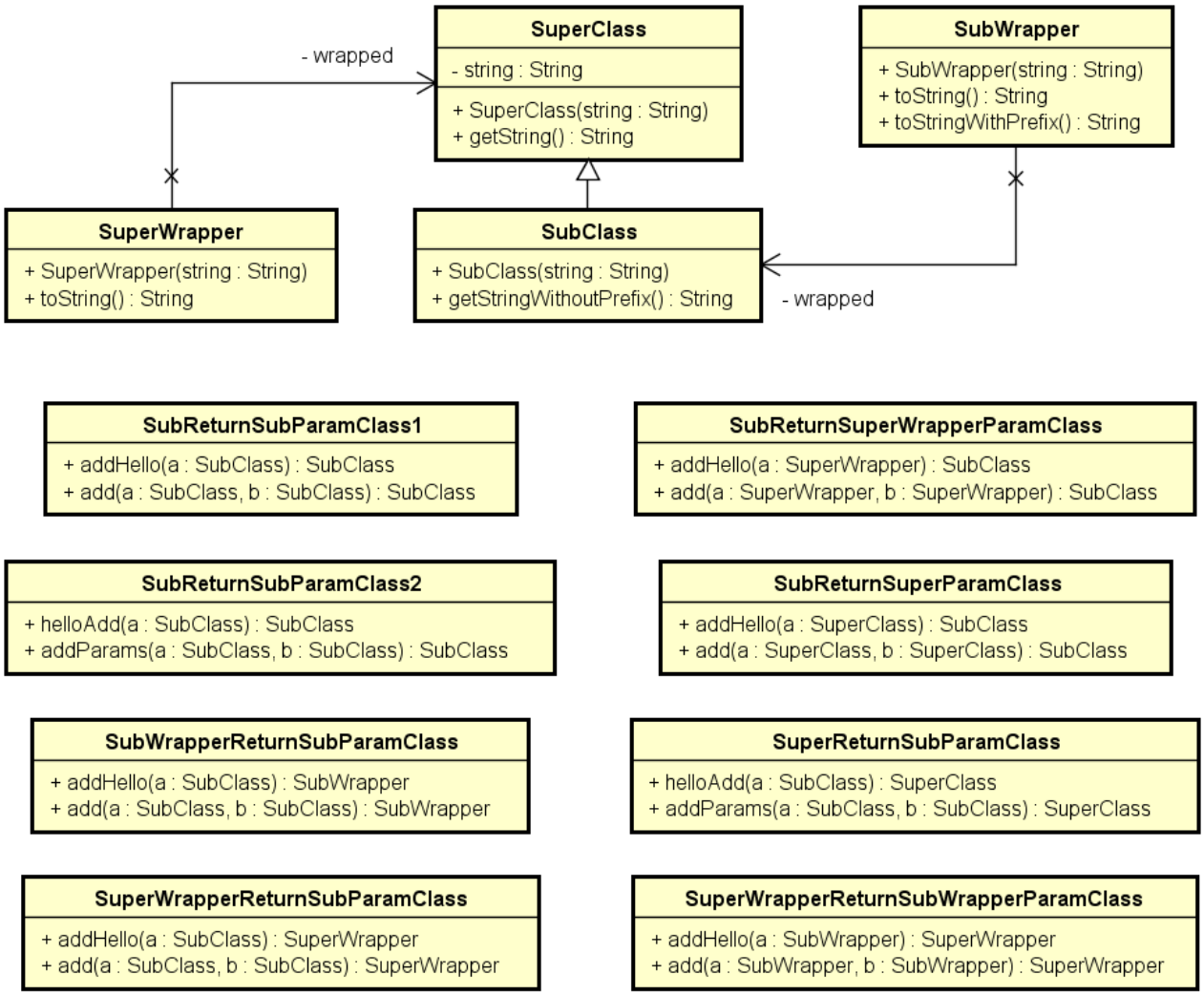


Abbildung 38: Alle Typen/Klassen, die in Matcher-Szenarien verwendet werden

Listing 4: Implemetierung: SuperClass

```
public class SuperClass {

  private String string;

  public SuperClass( String string ) {
    this.string = string;
  }
```

```
  public String getString() {
    return string;
  }
}
```

Listing 5: Implemetierung: SubClass

```
public class SubClass extends SuperClass {

  public SubClass( String string ) {
    super( "Sub" + string );
  }

  public String getStringWithoutPrefix() {
    return getString().substring( 3 );
  }
}
```

Listing 6: Implemetierung: SuperWrapperReturnSubWrapperParamClass

```
public class SuperWrapperReturnSubWrapperParamClass {

  public SuperWrapper addHello( SubWrapper a ) {
    return new SuperWrapper( a.toString() + "hello" );
  }

  public SuperWrapper add( SubWrapper a, SubWrapper b ) {
    return new SuperWrapper( a.toString() + b.toString() );
  }
}
```

A.1 Beispiel für den ExactTypeMatcher

In Listing 7 ist die Implementierung eines JUnit-Tests aufgeführt, in dem das Matching über den ExactTypeMatcher für unterschiedliche Source- und Target-Typen nachgewiesen werden soll. Die Test-Methode match enthält dabei die Aufrufe, bei denen das Matching festgestellt werden kann. Dementsprechend enthält die Test-Methode noMatch die Aufrufe, bei denen das Matching fehlschlägt.

Listing 8 enthält die Implementierung für einen JUnit-Test, in dem die Konvertierung, die

durch den ExactTypeMatcher beschrieben wird, nachgewiesen wird. Hierbei wird von dem Szenario aus 3.1.2 ausgegangen.

Listing 7: ExactTypeMatcher Matching Test

```
public class ExactTypeMatcher_MatcherTest {

  @Test
  public void match() {
    ExactTypeMatcher matcher = new ExactTypeMatcher();
    assertTrue( matcher.matchesType( String.class, String.class ) );
    assertTrue( matcher.matchesType( int.class, int.class ) );
    assertTrue( matcher.matchesType( Object.class, Object.class ) );
    assertTrue( matcher.matchesType( SuperClass.class, SuperClass.class ) );
  }

  @Test
  public void noMatch() {
    ExactTypeMatcher matcher = new ExactTypeMatcher();
    assertFalse( matcher.matchesType( String.class, int.class ) );
    assertFalse( matcher.matchesType( int.class, Object.class ) );
    assertFalse( matcher.matchesType( Object.class, String.class ) );
    assertFalse( matcher.matchesType( SuperClass.class, SubClass.class ) );
  }
}
```

Listing 8: ExactTypeMatcher Konvertierung Test

```
public class ExactTypeMatcher_ConversionTest {

  @Test
  public void convertString() {
    SuperClass target = new SuperClass( "A" );
    Collection<ModuleMatchingInfo> matchingInfos = new
        ExactTypeMatcher().calculateTypeMatchingInfos( SuperClass.class,
         SuperClass.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SuperClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
        .createProxyFactory( SuperClass.class );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SuperClass source = proxyFactory.createProxy( target, methodMatchingInfos );
```

```
    assertTrue( source.getString().equals( "A" ) );
  }
}
```

A.2 Beispiel für den GenTypeMatcher

Der GenTypeMatcher und der SpecTypeMatcher wurden gemeinsam implementiert. Daher wird in den folgenden Beispielen jeweils ein Matcher aus der Klasse GenSpecTypeMatcher erzeugt. Die weitere Verwendung den Matchers bezieht sich in diesen Beispielen aber auf die Definition des GenTypeMatchers aus 3.1.3.

In Listing 9 ist die Implementierung eines JUnit-Tests aufgeführt, in dem das Matching über den GenTypeMatcher für unterschiedliche Source- und Target-Typen nachgewiesen werden soll. Die Test-Methode match enthält dabei die Aufrufe, bei denen das Matching festgestellt werden kann. Dementsprechend enthält die Test-Methode noMatch die Aufrufe, bei denen das Matching fehlschlägt.

Listing 14 enthält die Implementierung für einen JUnit-Test, in dem die Konvertierung, die durch den GenTypeMatcher beschrieben wird, nachgewiesen wird. Hierbei wird von dem Szenario aus 3.1.3 ausgegangen.

Listing 9: GenTypeMatcher Matching Test

```
public class GenSpecTypeMatcher_Gen_MatcherTest {

  @Test
  public void match() {
    GenSpecTypeMatcher matcher = new GenSpecTypeMatcher();
    assertTrue( matcher.matchesType( Object.class, String.class ) );
    assertTrue( matcher.matchesType( SuperClass.class, SubClass.class ) );
    assertTrue( matcher.matchesType( Number.class, Integer.class ) );
  }

  @Test
  public void noMatch() {
    GenSpecTypeMatcher matcher = new GenSpecTypeMatcher();
    assertFalse( matcher.matchesType( int.class, String.class ) );
  }
}
```

Listing 10: GenTypeMatcher Konvertierung Test

```
public class GenSpecTypeMatcher_Gen_ConversionTest {

  @Test
  public void convertSpec2Gen() {
   SubClass target = new SubClass( "A" );
    Collection<ModuleMatchingInfo> matchingInfos = new
        GenSpecTypeMatcher().calculateTypeMatchingInfos(
         SuperClass.class, SubClass.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SuperClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
        .createProxyFactory( SuperClass.class );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SuperClass source = proxyFactory.createProxy( target, methodMatchingInfos );

    assertTrue( source.getString().equals( "SubA" ) );
  }
}
```

A.3 Beispiel für den SpecTypeMatcher

Wie in 3.1.3 bereits erwähnt wurde der GenTypeMatcher gemeinsam mit dem SpecTypeMatcher implementiert. Daher wird in den folgenden Beispielen jeweils ein Matcher aus der Klasse GenSpecTypeMatcher erzeugt. Die weitere Verwendung den Matchers bezieht sich in diesen Beispielen aber auf die Definition des SpecTypeMatcher aus 3.1.4.

In Listing 11 ist die Implementierung eines JUnit-Tests aufgeführt, in dem das Matching über den GenTypeMatcher für unterschiedliche Source- und Target-Typen nachgewiesen werden soll. Die Test-Methode match enthält dabei die Aufrufe, bei denen das Matching festgestellt werden kann. Dementsprechend enthält die Test-Methode noMatch die Aufrufe, bei denen das Matching fehlschlägt.

Listing 12 enthält die Implementierung für einen JUnit-Test, in dem die Konvertierung, die durch den SpecTypeMatcher beschrieben wird, nachgewiesen wird. Hierbei wird von dem Sze-

nario aus 3.1.4 ausgegangen. Die Test-Methode convertGen2Spec_positivCall enthält den Aufruf der ersten Methoden aus dem Szenario (getString). Die Test-Methoden convertGen2Spec_negativeCall beinhaltet den fehlschlagenden Aufruf der Methoden getStringWithoutPrefix.

Listing 11: SpecTypeMatcher Matching Test

```
public class GenSpecTypeMatcher_Spec_MatcherTest {

  @Test
  public void match() {
    GenSpecTypeMatcher matcher = new GenSpecTypeMatcher();
    assertTrue( matcher.matchesType( String.class, Object.class ) );
    assertTrue( matcher.matchesType( SubClass.class, SuperClass.class ) );
    assertTrue( matcher.matchesType( Integer.class, Number.class ) );
  }

  @Test
  public void noMatch() {
    GenSpecTypeMatcher matcher = new GenSpecTypeMatcher();
    assertFalse( matcher.matchesType( int.class, String.class ) );
  }
}
```

Listing 12: SpecTypeMatcher Konvertierung Test

```
public class GenSpecTypeMatcher_Spec_ConversionTest {

  @Test
  public void convertGen2Spec_positivCall() {
    SuperClass offeredComponent = new SuperClass( "A" );
    Collection<ModuleMatchingInfo> matchingInfos = new
        GenSpecTypeMatcher().calculateTypeMatchingInfos(
        SubClass.class, SuperClass.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SubClass> proxyFactory =
        moduleMatchingInfo.getConverterCreator().createProxyFactory( SubClass.class
        );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SubClass proxy = proxyFactory.createProxy( offeredComponent, methodMatchingInfos
        );
```

```
    assertTrue( proxy.getString().equals( "A" ) );
  }

  @Test( expected = SigMaGlueException.class )
  public void convertGen2Spec_negativeCall() {
    SuperClass offeredComponent = new SuperClass( "A" );
    Collection<ModuleMatchingInfo> matchingInfos = new
        GenSpecTypeMatcher().calculateTypeMatchingInfos(
        SubClass.class, SuperClass.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SubClass> proxyFactory =
        moduleMatchingInfo.getConverterCreator().createProxyFactory( SubClass.class
        );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SubClass proxy = proxyFactory.createProxy( offeredComponent, methodMatchingInfos
        );

    assertTrue( proxy.getString().equals( "A" ) );

    proxy.getStringWithoutPrefix();
  }
}
```

A.4 Beispiel für den WrappedTypeMatcher

Der WrappedTypeMatcher und der WrapperTypeMatcher wurden gemeinsam implementiert. Daher wird in den folgenden Beispielen jeweils ein Matcher aus der Klasse WrappedTypeMatcher erzeugt. Die weitere Verwendung den Matchers bezieht sich in diesen Beispielen aber auf die Definition des WrappedTypeMatcher aus 3.1.5.

In Listing 13 ist die Implementierung eines JUnit-Tests aufgeführt, in dem das Matching über den WrappedTypeMatcher für unterschiedliche Source- und Target-Typen nachgewiesen werden soll. Die Test-Methoden mit dem Präfix match enthalten dabei die Aufrufe, bei denen das Matching festgestellt werden kann. Dementsprechend enthält die Test-Methode noMatch die Aufrufe, bei denen das Matching fehlschlägt.

Listing ?? enthält die Implementierung für einen JUnit-Test, in dem die Konvertierung, die durch den WrappedTypeMatcher beschrieben wird, nachgewiesen wird. In der Test-Methode convertSubWrapper2SubClass wird von dem Szenario aus 3.1.5 ausgegangen. Die anderen Test-Methoden stellen weitere Szenarien dar, die in den folgenden Abschnitten beschrieben werden.

Listing 13: WrappedTypeMatcher Matching Test

```
public class WrappedTypeMatcher_Wrapped_MatcherTest {

  private WrappedTypeMatcher matcher = new WrappedTypeMatcher(
      MatcherCombiner.combine( new ExactTypeMatcher(), new GenSpecTypeMatcher() ) );

  @Test
  public void match() {
    assertTrue( matcher.matchesType( boolean.class, Boolean.class ) );
    assertTrue( matcher.matchesType( int.class, Integer.class ) );
  }

  @Test
  public void match_wrapped_exact() {
    assertTrue( matcher.matchesType( SubClass.class, SubWrapper.class ) );
  }

  @Test
  public void match_wrapped_spec() {
    assertTrue( matcher.matchesType( SubClass.class, SuperWrapper.class ) );
  }

  @Test
  public void match_wrapped_gen() {
    assertTrue( matcher.matchesType( SuperClass.class, SubWrapper.class ) );
  }

  @Test
  public void noMatch() {
    assertFalse( matcher.matchesType( String.class, String.class ) );
  }
}
```

Listing 14: WrappedTypeMatcher Konvertierung Test

```
public class WrappedTypeMatcher_Wrapped_ConversionTest {

  private WrappedTypeMatcher matcher = new WrappedTypeMatcher(
```

```
    MatcherCombiner.combine( new ExactTypeMatcher(), new GenSpecTypeMatcher() ) );

@Test
public void convertSubWrapper2SubClass() {
  SubWrapper offeredComponent = new SubWrapper( "A" );
  Collection<ModuleMatchingInfo> matchingInfos =
      matcher.calculateTypeMatchingInfos(
       SubClass.class, SubWrapper.class );

  ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

  ProxyFactory<SubClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
      .createProxyFactory( SubClass.class );
  Collection<MethodMatchingInfo> methodMatchingInfos =
      moduleMatchingInfo.getMethodMatchingInfos();

  SubClass proxy = proxyFactory.createProxy( offeredComponent, methodMatchingInfos
      );

  assertTrue( proxy.getString().equals( "SubA" ) );
  assertTrue( proxy.getStringWithoutPrefix().equals( "A" ) );
}

@Test
public void convertSuperWrapper2SubClass_positiveCall() {
  SuperWrapper offeredComponent = new SuperWrapper( "A" );
  Collection<ModuleMatchingInfo> matchingInfos =
      matcher.calculateTypeMatchingInfos(
       SubClass.class, SuperWrapper.class );

  ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

  ProxyFactory<SubClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
      .createProxyFactory( SubClass.class );
  Collection<MethodMatchingInfo> methodMatchingInfos =
      moduleMatchingInfo.getMethodMatchingInfos();

  SubClass proxy = proxyFactory.createProxy( offeredComponent, methodMatchingInfos
      );

  assertTrue( proxy.getString().equals( "A" ) );
}

@Test( expected = SigMaGlueException.class )
public void convertSuperWrapper2SubClass_negativeCall() {
```

```
    SuperWrapper offeredComponent = new SuperWrapper( "A" );
    Collection<ModuleMatchingInfo> matchingInfos =
        matcher.calculateTypeMatchingInfos(
        SubClass.class, SuperWrapper.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SubClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
        .createProxyFactory( SubClass.class );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SubClass proxy = proxyFactory.createProxy( offeredComponent, methodMatchingInfos
        );
    proxy.getStringWithoutPrefix();
  }

  @Test
  public void convertSubWrapper2SuperClass() {
    SubWrapper offeredComponent = new SubWrapper( "A" );
    Collection<ModuleMatchingInfo> matchingInfos =
        matcher.calculateTypeMatchingInfos(
        SuperClass.class, SubWrapper.class );

    ModuleMatchingInfo moduleMatchingInfo = matchingInfos.iterator().next();

    ProxyFactory<SuperClass> proxyFactory = moduleMatchingInfo.getConverterCreator()
        .createProxyFactory( SuperClass.class );
    Collection<MethodMatchingInfo> methodMatchingInfos =
        moduleMatchingInfo.getMethodMatchingInfos();

    SuperClass proxy = proxyFactory.createProxy( offeredComponent,
        methodMatchingInfos );

    assertTrue( proxy.getString().equals( "SubA" ) );
  }
}
```

Literatur

[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. Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 681–682, New York, NY, USA, 2006. Association for Computing Machinery.

[Hum08] Hummel, Oliver: *Semantic Component Retrieval in Software Engineering.* , April 2008.

[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.

[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.