

# **Untersuchung der Eclipse Refactoringtools hinsichtlich der Beachtung von Java Zugreifbarkeitsregeln**

Masterarbeit

Eingereicht von Andreas Kahl

Matrikelnummer: 7344198

27. Januar 2010

Lehrgebiet Programmiersysteme  
Fakultät für Mathematik und Informatik  
FernUniversität in Hagen

Betreuer:  
Prof. Dr. Friedrich Steimann  
Dipl.-Inform. Andreas Thies

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Wörtlich oder inhaltlich übernommene Literaturquellen wurden besonders gekennzeichnet.

Andreas Kahl

München, 27. Januar 2010

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>5</b>
1.1. Motivation . . . . .	5
1.2. Beitrag der Arbeit . . . . .	6
1.3. Aufbau der Arbeit . . . . .	7
<b>2. Problembeschreibung</b>	<b>8</b>
2.1. Zugreifbarkeit von Deklarationen . . . . .	8
2.2. Veränderungen durch Refactoringtools . . . . .	9
2.3. Relevanz der Zugreifbarkeit bei Refactorings . . . . .	10
2.4. Constraintbasierte Refactoringtools . . . . .	12
2.4.1. Typ-Constraints . . . . .	12
2.4.2. Zugreifbarkeits-Constraints . . . . .	13
2.4.3. Kategorisierung der Zugreifbarkeits-Constraints . . . . .	25
<b>3. Auswahl des Analyseverfahrens</b>	<b>26</b>
3.1. Ausprobieren . . . . .	26
3.2. Toolgestütztes Ausprobieren der Refactoringtools . . . . .	26
3.3. Analytisches Vorgehen . . . . .	28
3.4. Auswahl des Verfahrens zur Analyse . . . . .	31
<b>4. Detaillierte Beschreibung der Analyse</b>	<b>34</b>
4.1. Beschreibung des analytischen Verfahrens . . . . .	34
4.1.1. Abhängigkeitstabelle . . . . .	35
4.1.2. Abhängigkeit zwischen Constraintregel und Veränderlicher	36
4.1.3. Abhängigkeit zwischen Refactoringtool und Veränderlicher	36
4.1.4. Ermittlung der zu untersuchenden Fälle . . . . .	37
4.2. Veränderliche der Constraintregeln . . . . .	38
4.3. Durch Refactoringtool beeinflusste Veränderliche . . . . .	42
<b>5. Untersuchungsergebnisse</b>	<b>58</b>
5.1. Interpretation der Tabellen zur Ergebnisdarstellung . . . . .	58
5.2. Detaillierte Ergebnisse für ein Refactoringtool . . . . .	59
5.2.1. Beschreibung des PullUp-Refactoringtools . . . . .	59
5.2.2. Ergebnisse für PullUp-Methode-Refactoringtool . . . . .	61
5.3. Weitere Untersuchungsergebnisse . . . . .	72

## *Inhaltsverzeichnis*

<b>6. Diskussion der Ergebnisse</b>	<b>87</b>
6.1. Interpretation der Ergebnisse . . . . .	87
6.2. Bewertung der Ergebnisse . . . . .	90
<b>7. Schlussbetrachtungen</b>	<b>91</b>
7.1. Zusammenfassung . . . . .	91
7.2. Ausblick . . . . .	92
7.3. Fazit . . . . .	92
<b>Literaturverzeichnis</b>	<b>93</b>
<b>Abbildungsverzeichnis</b>	<b>94</b>
<b>Tabellenverzeichnis</b>	<b>95</b>
<b>Listings</b>	<b>97</b>
<b>A. Anhang</b>	<b>99</b>
A.1. Beschreibung der beiliegenden CD . . . . .	99

# 1. Einleitung

In dieser Arbeit wird eine Analyse der in Eclipse 3.4.2 eingebauten Java-Refactoringtools unter dem Aspekt der Zugreifbarkeit von Deklarationen (engl.: declarations) beschrieben. Im Folgenden wird die Motivation für die Arbeit und deren Beitrag zur Problemlösung erläutert. Im Anschluss erfolgt eine Übersicht über den Aufbau der Arbeit.

## 1.1. Motivation

Die Programmiersprache Java verwendet das Konzept der Zugriffskontrolle (engl.: accessibility), um Informationen über die Implementierung innerhalb von Paketen und Klassen vor den Aufrufern zu verbergen [JLS, § 6.6]. Die Zugriffskontrolle wird in Java dabei durch Zugriffsmodifizierer wie „public“, „protected“ oder „private“ realisiert.

Als Refactoring bezeichnet man „eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern“[Fow05]. Refactoringtools unterstützen den Entwickler als Werkzeug bei der Durchführung von Refactorings. Wenn sie dabei die Struktur der Software und damit auch den Quellcode verändern, dann müssen sie auch die Zugriffskontrolle für Deklarationen berücksichtigen, um das beobachtbare Programmverhalten nicht zu verändern. In Listing 1.1 wird die Ausgangssituation für ein Beispiel vorgestellt, in dem ein fehlerhaftes Refactoringtool wegen unzureichender Berücksichtigung von Zugreifbarkeiten ein verändertes Programmverhalten hervorruft. Beim Verschieben der Klasse B in ein anderes Paket (engl.: package) wird durch den Ausdruck `(new B()).m(“abc“)` in Zeile 3 statt wie vorher die Methode `B.m(String)` nun `B.m(Object)` aufgerufen, da die erstere Methode mit dem Parameter String für den Aufruf in Methode `A.n()` (Zeile 4) nicht mehr zugreifbar ist. Der

## 1. Einleitung

```
1 package a;
2 public class A {
3     void n() { (new B()).m("abc"); } //an m(String) gebunden, wenn zugreifbar
4 }
5
6 package a;
7 public class B extends A {
8     public void m(Object o) {...}
9     void m(String s) {...} //überladen von m(Object o)
10 }
```

Listing 1.1: Beispiel für geändertes Programmverhalten [ST09]

Grund hierfür ist, dass von einem anderen Paket aus nur noch die Methode `B.m(Object)` aufgrund der Deklaration mit „public“ zugreifbar ist. Eine korrekte Implementierung des Refactoringtools würde den Zugriffsmodifizierer der Methode `B.m(String)` nach dem Verschieben auf „public“ setzen, um das Programmverhalten nicht zu verändern. Es ist festzustellen, dass es beim Refactoring zu keinem Compilerfehler kommt, es wird aber das beobachtbare Verhalten ohne weitere Hinweise geändert. Die Veränderung des Verhaltens bleibt unbemerkt, es sei denn, es sind Tests z. B. mit JUnit [JUn] vorhanden, in denen genau dieses Verhalten geprüft und eine Abweichung gemeldet wird. Wenn diese Änderungen nicht bemerkt werden, sind durch Refactorings verursachte Fehler in der Software möglich. Diese Fehler sind im Rahmen von Refactorings aber nicht akzeptabel, insbesondere widersprechen sie ihrer Definition.

## 1.2. Beitrag der Arbeit

In der vorliegenden Arbeit werden alle in der Eclipse-Version 3.4.2 der JDT (Java Development Tools) [ECL] implementierten Java-Refactoringtools auf die korrekte Berücksichtigung der Zugreifbarkeiten untersucht. In der Eclipse JDT existieren 28 Refactoringtools, die zu berücksichtigen sind. Die Ursachen für eine mangelnde Berücksichtigung der Zugreifbarkeiten und den damit verbundenen Schwierigkeiten können in Java u. a. Konzepte wie Vererbung (engl.: inheritance), Überschreiben (engl.: overriding), Überladen (engl.: overloading) oder Überdecken (engl.: hiding) [JLS] sein. Die Untersuchungsergebnisse sollen einen Beitrag zur Bewertung des Ausmaßes der fehlenden Berücksichtigung leisten und als Basis für Fehlerkorrekturen dienen. Es kann durch eine

## 1. Einleitung

zukünftige Korrektur der gefundenen Fehler sichergestellt werden, dass die Refactoringtools entsprechend ihrer Definition arbeiten, „indem sie eine Software umstrukturieren, ohne ihr beobachtbares Verhalten zu ändern“ [Fow05].

### 1.3. Aufbau der Arbeit

In Kapitel 2 wird auf das Problem der mangelnden Berücksichtigung der Zugreifbarkeit bei den Refactoringtools eingegangen. Es wird dort anhand eines Beispiels die Problematik erläutert und auf die Relevanz des Problems eingegangen. Im Rahmen von Kapitel 3 erfolgen eine Diskussion der Lösungsansätze für die Analyse der Refactoringtools sowie eine Auswahl des Analyseverfahrens für die Untersuchung. Gegenstand von Kapitel 4 ist es, den genauen Ablauf der Analyse detailliert zu erläutern und deren allgemeine Schritte zu dokumentieren. In Kapitel 5 werden nach einer Einführung zur Darstellung der Untersuchungsergebnisse diese für die einzelnen Refactoringtools beschrieben. Im folgenden Kapitel 6 werden die Ergebnisse der Analyse diskutiert, und es findet deren Bewertung statt. Das Kapitel 7 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick, der in einem Fazit mündet, ab.

## 2. Problembeschreibung

In diesem Kapitel wird das Problem der Zugreifbarkeit in Java-Programmen im Rahmen von Refactorings beschrieben. Der Aspekt des beobachtbaren Programmverhaltens bei der Durchführung von Refactorings wird erläutert und mit Beispielen illustriert. Die Zugreifbarkeit hat bei der Durchführung von Refactorings eine hohe Relevanz und kann, wenn sie nicht berücksichtigt wird, zu Fehlern führen. Durch das Einführen von Bedingungen (engl.: constraints), die bei Programmänderungen zu beachten sind, können diese Fehler vermieden werden. Im letzten Abschnitt dieses Kapitels wird deshalb die Funktionsweise constraintbasierter Refactoringtools vorgestellt und aufgezeigt, wie dieses Konzept auf die Zugreifbarkeit übertragen werden kann.

### 2.1. Zugreifbarkeit von Deklarationen

Die Zugreifbarkeit gibt an, ob auf eine Deklaration zugegriffen werden darf. Bei den Zugriffen kann es sich um Deklarationen handeln, die einen Typ referenzieren, das kann z. B. der Typ eines Feldes oder der Typ eines formalen Parameters in einer Methode sein. Ein weiteres Beispiel bildet der Zugriff auf eine Deklaration, wie z. B. bei einem Methodenaufruf, bei dem auf die Methodendeklaration zugegriffen wird. Die Zugriffskontrolle in Java regelt die Zugreifbarkeit von Deklarationen. Das Ziel hierbei besteht darin, Informationen über Implementierungen zu verbergen. Es soll beispielsweise der Benutzer einer Klasse nur deren öffentliche Methoden, d. h. die Schnittstelle, kennen und nicht die interne Realisierung.

Die Steuerung der Zugreifbarkeit erfolgt in Java durch die Zugriffsmodifizierer (engl.: access modifier) „public“, „protected“ und „private“. Sie werden den Deklarationen vorangestellt. In Java existieren folgende Deklarationen, die mit einem Zugriffsmodifizierer ausgestattet sein können [JLS, §6.6]: Klassen,

## 2. Problembeschreibung

Interfaces, Methoden, Konstruktoren, Felder und innere Typen. Im Folgenden werden die einzelnen Zugriffsmodifizierer erläutert.

Die Intention bei „public“ besteht darin, auf dieses Element von jedem beliebigen Ort aus zugreifen zu können. Im Gegensatz dazu kann man auf „private“ Elemente nur aus der umschließenden Deklaration aus zugreifen. Es kann z. B. auf Attribute, die in einer Klasse als „private“ deklariert sind, nur in dieser Klasse zugegriffen werden.

Ist kein Zugriffsmodifizierer angegeben, hat ein Element die Standardzugreifbarkeit „package“. Bei der Standardzugreifbarkeit wird der Bereich, von dem aus ein Element zugreifbar ist, auf das Paket beschränkt. Das Element ist somit von jedem Ort des Paketes, in dem es deklariert ist, zugreifbar. So sind beispielsweise Klassen, die als „package“ deklariert sind, vom Programmcode außerhalb des eigenen Paketes aus nicht zugreifbar [JLS, §6.6.5].

Bei der Zugreifbarkeit „protected“ werden die abgeleiteten Klassen als Orte von Zugriffen speziell berücksichtigt. Die Zugreifbarkeit entspricht hierbei der von „package“ mit der Erweiterung, dass der Zugriff auf protected Methoden, Felder oder Konstruktoren eines Objektes aus anderen Paketen für Code möglich ist, der für die Implementierung dieses Objektes verantwortlich ist [JLS, §6.6.2]. Es wird damit z. B. möglich, auch nicht mit „public“ veröffentlichte Methoden zu überschreiben [Dau07].

Für die Zugreifbarkeit ergibt sich folgende Rangfolge, wobei > bedeutet, dass links davon eine höhere Zugreifbarkeit besteht.

public > protected > Standardzugreifbarkeit(package) > private

## 2.2. Veränderungen durch Refactoringtools

Refactoringtools führen Änderungen an Programmen durch. Nachfolgend wird ein Beispiel für eine Veränderung an einem Programm durch ein Rename-Refactoring gezeigt. Durch die Anwendung des Refactoringtools auf den Bezeichner (engl.: identifier) einer Klasse müssen auch alle Stellen, an denen der Klassenname verwendet wird, berücksichtigt und somit mit umbenannt werden. Die Anwendung des Refactoringtools wird durch die Kommentare im Listing 2.1 verdeutlicht. Im Beispiel ist nicht nur der Name in der Klassendeklaration

```
1 package a;  
2 class A { //in A2 umbenennen  
3 }  
4  
5 class B {  
6     A value; //betroffen von Umbenennung A->A2  
7 }  
8  
9 class C {  
10     A value; //betroffen von Umbenennung A->A2  
11 }
```

Listing 2.1: Berücksichtigung von anderen Programmteilen durch Refactoringtools

zu ändern, auch bei allen Verwendungen des Typs ist der Bezeichner anzupassen. In diesem Fall sind dies die Deklarationen der Felder value in den Klassen B und C.

### 2.3. Relevanz der Zugreifbarkeit bei Refactorings

Veränderungen an Zugriffsmodifizierern von Deklarationen können das Programmverhalten beeinflussen. Es haben aber auch Veränderungen an Deklarationen einen Einfluss auf die Zugreifbarkeit, so kann durch die Ortsveränderung einer Deklaration die Zugreifbarkeit von bestimmten Orten, wie im nachfolgenden Beispiel Listing 2.2 veranschaulicht, verändert werden.

Wie bereits erläutert, muss für den Zugriff auf eine Deklaration in einem anderen Paket dieses dort den Zugriffsmodifizierer „public“ aufweisen, falls der Zugriff nicht in einer abgeleiteten Klasse, bei der dann „protected“ ausreichend wäre, stattfindet. Sollte es eine geringere Zugreifbarkeit, wie z. B. „package“, besitzen, so würde der Compiler einen Fehler melden. Das Beispiel ist in Listing 2.2 als Quellcode abgebildet. Wenn dort die Klasse A durch ein Refactoringtool vom Paket b in das Paket a verschoben wird und dabei der Zugriffsmodifizierer der Klasse A nicht von „package“ auf „public“ gesetzt wird, kommt es zu einem Fehler beim Zugriff auf die Deklaration von A in Zeile 10.

In Listing 2.2 wäre der Fehler durch den Compiler gemeldet worden, und der Programmierer hätte ihn nach dem Refactoring beseitigen können. Es gibt aber auch Fälle, in denen das beobachtbare Programmverhalten ohne Hinweise für

## 2. Problembeschreibung

```
1 package b;
2 class A { //package für Zugriff von B unzureichend, wenn in Paket a
3           //public ist Voraussetzung für Zugreifbarkeit, wenn in Paket a
4 }
5
6 package b;
7 import a.A;
8
9 class B {
10     A value; //Fehler wenn Klasse A (package) nicht zugreifbar wäre
11 }
```

Listing 2.2: Beispiel für die Verletzung der Zugriffsrechte

den Programmierer verändert wird. Diese Fälle sind besonders kritisch, da die daraus resultierenden Fehler nicht sofort entdeckt werden. Ein Beispiel dafür ist Listing 2.3 zu entnehmen. Wenn hier die Klasse B durch ein Refactoringtool ohne Anpassungen der Zugreifbarkeit in ein anderes Paket verschoben wird, so ändert sich das beobachtbare Programmverhalten, da nun der Methodenaufruf `m("abc")` in Zeile 5 nicht mehr dynamisch an `B.m(String)` gebunden wird. In diesem Fall wird die Methode `A.m(String)` nicht länger überschrieben (engl.: *override*).

```
1 package a;
2 public class A {
3     void m(String s) {...}
4     void n() {
5         ((A) new B()).m("abc");
6     }
7 }
8
9 package a;
10 public class B extends A {
11     void m(String s) {...}
12 }
```

Listing 2.3: Fehler durch Refactoringtool ohne Hinweis [ST09]

In den beiden gezeigten Beispielen wird das Refactoringtool seiner Definition, das beobachtbare Programmverhalten nicht zu verändern, nicht gerecht. Refactoringtools übernehmen eine hohe Verantwortung, da sie automatisiert Programmcode ändern. Diese Änderungen müssen eine hohe Zuverlässigkeit aufweisen. Fehler in durch Refactoringtools durchgeführten Refactorings mindern die Anwendbarkeit und auch die Akzeptanz durch den Anwender.

## 2.4. Constraintbasierte Refactoringtools

Eine Möglichkeit der Implementierung von Refactoringtools bilden constraintbasierte Refactoringtools. Bei Constraints handelt es sich um eine Teilmenge der Vorbedingungen, die während der Ausführung des Refactoringtools geprüft werden, um die Durchführbarkeit des Refactorings zu gewährleisten. Für die Erstellung der Constraints werden Constraintregeln verwendet; sie definieren, unter welchen Bedingungen ein Constraint zu erstellen ist und welchen Inhalt es aufweist.

Der Ansatz der Constraints wurde zuerst für die korrekte Behandlung von Typen eines Programms während Refaktorisierungen verwendet [TKB]. Eine Implementierung der Typ Constraints existiert bereits innerhalb der JDT und findet bei den Refactoringtools „Extract Interface“ und „PullUp-Members“ Anwendung [TKB]. Die Typ Constraints sind auch für weitere Refactoringtools in Eclipse angewendet worden, sodass es sich dabei um ein etabliertes Konzept handelt.

Es bietet sich an, dieses Verfahren auch auf die Zugreifbarkeit zu übertragen und dort ebenfalls über Constraints die Vorbedingungen zu beschreiben. Für die Zugreifbarkeitsregeln in Java werden in [ST09] Constraintregeln vorgestellt, die für Refactoringtools verwendet werden können.

In den folgenden Unterabschnitten werden die beiden Ansätze vorgestellt und die Constraintregeln für die Zugreifbarkeit detailliert erläutert.

### 2.4.1. Typ-Constraints

Der Ansatz mit Constraints für Typen verfolgt das Ziel, die Vorbedingungen für Refactorings zu bestimmen, welche den Zweck haben, vorhandene Typen in Deklarationen durch andere zu ersetzen. Für einige Refactoringtools, wie z.B. das Extract Interface, ist in Eclipse ein Constraintsystem, das auf entsprechenden Constraintregeln [TKB] beruht, implementiert worden. Das Extract-Interface-Refactoringtool extrahiert aus einer bestehenden Klasse ein Interface, das dann von ihr implementiert wird. Das neue Interface wird nach Möglichkeit bei den Deklarationen, bei denen bisher der Typ der Klasse verwendet wurde, als Typ eingesetzt. Es wird dabei geprüft, ob Deklarationen das neu erzeugte

Interface statt des bisherigen Typs verwenden können. Dafür werden aufgrund der Regeln die Constraints aufgestellt und wird eine Lösung für das System berechnet, welche eine Aussage über die Ersetzbarkeit des Typs trifft.

### 2.4.2. Zugreifbarkeits-Constraints

In [ST09] wurde ein Satz von Constraintregeln entwickelt, der die Zugreifbarkeit beschreiben und die notwendigen Zugriffsmodifizierer berechnen kann. In diesem Abschnitt werden diese Constraintregeln beschrieben. Die im Folgenden verwendeten Abkürzungen für die Bezeichnung der Constraintregeln haben die unten stehende Bedeutung:

<b>Acc</b>	Zugriff (engl.: accessing)
<b>Inh</b>	Vererbung (engl.: inheritance)
<b>Sub</b>	Subtyp (engl.: subtyping)
<b>Dyn</b>	Dynamisches Binden (engl.: dynamic binding)
<b>Ovr</b>	Überladen (engl.: overloading)
<b>Hid</b>	Verdecken (engl.: hiding)
<b>Misc</b>	Verschiedenes (engl.: miscellaneous)

Den Abkürzungen der Constraintregeln wird zur zusätzlichen Unterscheidung eine laufende Nummer angehängt, wenn es mehrere in einem Themengebiet gibt, wie z. B. ACC-1 und ACC-2 für das Thema „Zugriff“.

In den Constraintregeln werden die unten aufgeführten Definitionen verwendet:

#### Definitionen für Mengen

D	Deklarationen [JLS, §6.1]
R	Referenzen
S	Deklarationen mit Static
F	Felder
M	Methoden
C	Klassen
I	Interfaces
L	Orte
A = { <i>public, protected, package, private</i> }	Java Zugriffsmodifizierer

Unter der Menge der Referenzen R sind alle Referenzierungen von Deklarationen zu verstehen. Dazu zählen beispielsweise: Aufrufe von Methoden, Zugriffe

## 2. Problembeschreibung

auf Felder, Verwendung von Typen in Deklarationen oder auch Konstruktoren. Bei der Menge der Orte  $L$  wird unter einem Ort die Beschreibung durch das Paket und die umschließende Klasse bzw. Interface verstanden, da nur diese für die Zugreifbarkeit relevant sind. So ist der Ort für eine Methode  $m$ , die in einer Klasse  $A$  deklariert ist, welche sich im Paket  $a$  befindet, der Ort  $a.A$ . Wenn in dieser Methode  $m$  Deklarationen referenziert werden, so ist der Ort der Referenzierung dann auch  $a.A$ , da er sich auf die umschließende Klasse mit ihrem Paket bezieht.

### Definitionen für Variablen aus den Mengen

Im Folgenden sollen für die Variablen, die sich auf die definierten Mengen beziehen, die unten stehenden Konventionen gelten.

$d \in D$	eine Deklaration aus der Menge $D$
$d' \in D$	eine weitere Deklaration aus der Menge $D$
$r \in R$	eine Referenz aus der Menge $R$
$m \in M$	eine Methode aus der Menge $M$
$c \in C$	eine Klasse aus der Menge $C$
$i \in I$	ein Interface aus der Menge $I$
$l \in L$	ein Ort aus der Menge $L$

### Definitionen für Funktionen

$\langle \cdot \rangle : D \rightarrow A$	ermittelt Zugriffsmodifizierer einer Deklaration
$\lambda : D \cup R \rightarrow L$	ermittelt den Ort einer Deklaration oder einer Referenz $r$
$\alpha : L \times L \rightarrow A$	ermittelt Zugriffsmodifizierer, sodass der Ort $l$ von einem weiteren Ort $l'$ aus zugreifbar ist
$\beta : R \rightarrow D$	ermittelt, an welche Deklaration $d$ die Referenzierung $r$ gebunden ist
$\rho : R \rightarrow L$	ermittelt Ort $l$ des statischen Typs des Objektes, über das $r$ referenziert wird
$\iota : D \rightarrow String$	ermittelt Bezeichner einer Deklaration $d$
$subclasses : C \rightarrow C$	bestimmt alle abgeleiteten Klassen einer Klasse $c$
$superclasses : C \rightarrow C$	bestimmt alle Basisklassen einer Klasse $c$
$overrides : M \times M \rightarrow \{wahr, falsch\}$	prüft, ob eine Methodendeklaration $m$ überschrieben wird [JLS, § 8.4.8.1]
$overloads : M \times M \rightarrow \{wahr, falsch\}$	prüft, ob eine Methodendeklaration $m$ überladen wird [JLS, § 8.4.9]

## 2. Problembeschreibung

$hides : D \times D \rightarrow \{wahr, falsch\}$	prüft, ob eine Deklaration $d$ überdeckt wird [JLS, § 8.4.8.2]
$implements : C \times I \rightarrow \{wahr, falsch\}$	prüft, ob die Klasse $c$ ein Interface $i$ implementiert [JLS, § 8.1.5]
$inherits : C \times D \times C \rightarrow \{wahr, falsch\}$	prüft, ob eine Deklaration $d$ aus einer Klasse $c$ in einer anderen Klasse $c$ geerbt wird [JLS, § 8.2]
$subsignature : M \times M \rightarrow \{wahr, falsch\}$	prüft, ob zwei Methoden $m$ die gleiche Subsignatur haben [JLS, § 8.4.2]

### Weitere Definitionen

absent	in [ST09] neu eingeführter Zugriffsmodifizierer; eine Deklaration damit darf an dem gegebenen Ort nicht eingefügt bzw. vorhanden sein
--------	---

Im Folgenden werden die einzelnen Constraintregeln detailliert erläutert.

### Acc-1 Zugriff

$$\beta(r) = d \Rightarrow \langle d \rangle \geq \alpha(\lambda(r), \lambda(d))$$

Bei dem Zugriff auf eine Deklaration  $d$  über eine Referenz  $r$  muss der Zugriffsmodifizierer von  $d$  also  $\langle d \rangle$  mindestens so groß sein, wie von den Zugriffsregeln der Sprache Java gefordert. Im Listing 2.4 ist ein Beispiel dazu angegeben. Hier darf die Zugreifbarkeit der Klasse B nicht reduziert werden, da sonst der Zugriff auf die Deklaration von der Klasse A, wie in Zeile 2 abgebildet, nicht mehr möglich ist. Eine Reduzierung der Zugreifbarkeit von Klasse B würde in dem Beispiel zu einem Compilerfehler führen.

```
1 package a;  
2 public class A { /*r1*/B value; }  
3  
4 package b;  
5 public class /*d1*/ B {}
```

Listing 2.4: Beispiel für Acc-1 [ST09]

## 2. Problembeschreibung

### Acc-2 Zugriff auf protected-Methoden und -Konstruktoren

$$\beta(r) = d \wedge \alpha(\lambda(r), \lambda(d)) = \text{protected} \wedge d \notin S \wedge \rho(r) \notin \text{subclasses}(\lambda(r)) \cup \{\lambda(r)\} \Rightarrow \langle d \rangle = \text{public}$$

Bei dieser Constraintregel ist besonders die Funktion  $\rho$  von großer Bedeutung. Sie ermittelt für eine Referenzierung, wie z. B. einem Methodenaufruf, den Ort des statischen Typs, über den referenziert wird. Wenn, wie in Listing 2.5 in Zeile 9, eine Referenzierung eines Feldes stattfindet, ermittelt die Funktion den statischen Typ, der dann in diesem Beispiel der Typ A ist. Bei dem statischen Typ handelt es sich um den Typ, mit dem die Variable deklariert ist, über die andere Deklarationen referenziert werden. Im Beispiel wäre das die Variable a, die als Methodenparameter mit dem Typ A in Zeile 8 deklariert ist und dann in Zeile 9 verwendet wird.

```
1 package a;
2 public class A {
3     protected /*d1*/int i; //muss public statt protected haben für Zugriff r1
4 }
5
6 package b;
7 public class B extends a.A {
8     void m(A a) {
9         /*r1*/ a.i++; //Compilerfehler, da kein Zugriff auf A.i
10    }
11 }
```

Listing 2.5: Beispiel für Acc-2

Mit der Constraintregel Acc-2 werden Fälle berücksichtigt, bei denen auf Deklarationen mit dem Zugriffsmodifizierer „protected“ in abgeleiteten Klassen zugegriffen werden soll, die sich außerhalb des Paketes der Basisklasse mit dem protected-Member befinden. In Java darf auf geerbte protected-Deklarationen in fremden Paketen nur zugegriffen werden, wenn der Zugriff dazu dient, das Verhalten der erbenden Klassen zu implementieren. So ist der Aufruf einer paketfremden, geerbten protected-Methode auf `this` ebenso erlaubt wie auf abgeleitete Klassen, aber nicht auf Variablen, die eine Basisklasse referenzieren. In Listing 2.5 wird eine Referenz auf die Basisklasse als Parameter der Methode `B.m(A a)` in Zeile 8 übergeben. In der folgenden Zeile 9 erfolgt dann der Zugriff auf das Feld `A.i`. Für diesen Zugriff wird aber die Zugreifbarkeit „public“ für `A.i` gefordert, da der Zugriff nicht über den statischen Typ der

## 2. Problembeschreibung

Klasse B erfolgt, bei dem „protected“ ausgereicht hätte, sondern über den Typ A, bei dem „public“ gefordert wird [JLS, § 6.6.7]. Die Constraintregel Acc-2 verschärft somit die Forderungen von Acc-1.

### Inh-1 Zugriff auf geerbte Methode oder Feld

$$\beta(r) = d \wedge \rho(r) \neq \lambda(d) \Rightarrow \langle d \rangle \geq \alpha(\rho(r), \lambda(d))$$

Es soll garantiert werden, dass der statische Typ einer Variable, über die der Zugriff auf eine Methode oder ein Feld erfolgt, auch Zugriff auf diese hat, wenn er es von einer seiner Basisklassen erbt. Der statische Typ  $\rho(r)$  muss also auf die Deklaration  $\lambda(d)$  zugreifen können. Die Regel geht über die in Acc-1 geforderte Zugreifbarkeit der Deklaration von dem Ort der Referenzierung  $\lambda(r)$  hinaus. In Listing 2.6 ist aufgezeigt, dass Acc-1, weil d1 und r1 in der gleichen Klasse A sind, die Zugreifbarkeit „private“ für d1 bestimmen würde. Diese Zugreifbarkeit wird aber von Inh-1 auf mindestens „protected“ festgelegt, weil der Aufruf über die Klasse B erfolgt und diese A.i dafür erben muss.

```
1 package a;
2 public class A {
3     protected /*d1*/int i;
4     void n() { /*r1*/(new B()).i = 1; }
5 }
6
7 package b;
8 public class B extends a.A {}
```

Listing 2.6: Beispiel für Inh-1 [ST09]

### Inh-2 Zugriff auf mehrfach geerbte statische Felder

$$\begin{aligned} \{d, d'\} \subseteq F \cap S \wedge \iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in \text{superclasses}(\rho(r)) \\ \Rightarrow \langle d' \rangle < \alpha(\rho(r), \lambda(d')) \end{aligned}$$

Es geht hierbei darum, einen mehrdeutigen Zugriff auf geerbte statische Felder zu verhindern [JLS, § 8.3.3.3]. In Listing 2.7 ist ein Beispiel dazu angegeben. Es sind dort die beiden Felder d1 und d2 mit gleichen Bezeichnern in der Methode B.m(...) zugreifbar und somit nicht eindeutig auflösbar, woraus ein Compilerfehler resultiert.

## 2. Problembeschreibung

```
1 public class A {
2     public /*d1*/ static int i=1;
3 }
4
5 public interface I {
6     public /*d2*/ static int i=2;
7 }
8
9 public class B extends A implements I{
10     void m() { int j = /*r1*/i; } // i nicht eindeutig
11 }
```

Listing 2.7: Beispiel für Inh-2 [ST09]

### Sub-1 Subtyp

$$\{d, d'\} \subseteq M \wedge (\text{overrides}(d', d) \vee \text{hides}(d', d)) \Rightarrow \langle d' \rangle \geq \langle d \rangle$$

In der Java-Sprachdefinition [JLS, § 8.4.8.3] wird gefordert, das überschreibende oder überdeckende Methoden die Zugreifbarkeit der überschriebenen oder überdeckten Methode nicht verringern dürfen. Diese Forderung wird mit der Constraintregel Sub-1 berücksichtigt. Im Beispiel von Listing 2.8 ist es demnach nicht zulässig, die Zugreifbarkeit für die Methode B.m() zu verringern, weil sie dann kleiner als bei der überschriebenen Methode A.m() wäre.

```
1 package a;
2 public class A {
3     public void m() { } /* d */
4 }
5
6 package b;
7 public class B extends A{
8     public void m() { } /* d' */ //überschreibt A.m
9 }
```

Listing 2.8: Beispiel für Sub-1

### Sub-2 Subtyp, der ein Interface über eine geerbte Methode implementiert

$$\{d, d'\} \subseteq M \wedge \text{subsignature}(d', d) \wedge \{c, c'\} \subseteq C \wedge i \in I \wedge \lambda(d) = i \wedge \lambda(d') = c' \wedge \text{implements}(c, i) \wedge \text{inherits}(c, d', c') \Rightarrow \langle d' \rangle = \text{public}$$

Die JLS [JLS, § 8.4.2] fordert, dass eine Methode, die von einer Klasse geerbt wird, den Zugriffsmodifizierer „public“ aufweist, wenn sie einer Klasse ermöglicht, ein Interface zu implementieren. Diese Forderung wird durch die

## 2. Problembeschreibung

Constraintregel Sub-2 abgebildet. In Listing 2.9 ist ein Beispiel für die Regel abgebildet. Die Methode A.m() in Zeile 6 wird durch die Klasse B geerbt und ermöglicht ihr somit die Implementierung des Interfaces I, welches die Methode m() in Zeile 2 deklariert. Entsprechend der Regel, besitzt die Methode A.m() den Zugriffsmodifizierer „public“.

```
1 public interface I {
2     void m(); /* d */
3 }
4
5 public class A /* c' */ {
6     public void m() { } /* d' */
7 }
8
9 public class B /* c */ extends A implements I /* implements(c, i) */
10 {}
```

Listing 2.9: Beispiel für Sub-2

**Dyn-1** Verhindern, dass das Überschreiben von Methoden entfernt wird

$$\text{overrides}(d', d) \Rightarrow \langle d \rangle \geq \alpha(\lambda(d'), \lambda(d))$$

Wenn davon ausgegangen wird, dass eine Methode eine andere überschreibt, so soll der Zugriffsmodifizierer für die überschriebene Methode d einen Wert haben, der sie für die überschreibende Methode d' zugreifbar macht. Die Einflussgrößen dafür sind die mit  $\lambda$  bestimmten Orte der beiden der beiden Deklarationen. Wird die Regel missachtet, so kann sich das Programmverhalten ändern. Würde in dem Beispiel von Listing 2.10 der Zugriffsmodifizierer der Methode A.m() auf „package“ reduziert, so würde das Überschreiben durch B.m() entfernt. Als Konsequenz daraus würde sich für den Methodenaufruf in Zeile 7 das Verhalten ändern, da A.m() statt wie vorher B.m() aufgerufen wird. Für den Aufruf in Zeile 5 bleibt das Verhalten unverändert.

**Dyn-2** Verhindern, dass das Überschreiben von Methoden eingeführt wird

$$\begin{aligned} \lambda(d) \in \text{superclasses}(\lambda(d')) \wedge \text{subsignature}(d', d) \wedge \neg \text{overrides}(d', d) \\ \Rightarrow \langle d \rangle < \alpha(\lambda(d'), \lambda(d)) \end{aligned}$$

## 2. Problembeschreibung

```
1 package a;
2 public class A {
3     public void m() { } /* d */
4     public void n() {
5         new A().m(); //Verhalten unverändert
6         A a= new B();
7         a.m();      //Verhalten verändert !!!
8     }
9 }
10
11 package b;
12 public class B extends A{
13     public void m() { } /* d' */ //überschreibt A.m
14 }
```

Listing 2.10: Beispiel für Dyn-1

Wenn eine Methode  $d$  bisher nicht von einer Methode  $d'$  überschrieben wurde, soll das auch nach einem Refactoring nicht der Fall sein. Zu diesem Zweck ist der Zugriffsmodifizierer einer überschriebenen Methode  $d$  entsprechend so zu setzen, dass die Deklaration von einer Methode  $d'$  mit gleicher Signatur in einer abgeleiteten Klasse aus nicht zugreifbar ist. Bei Missachtung der Regel kann sich das Programmverhalten ändern. Im Beispiel von Listing 2.11 darf für die Methode  $A.m()$  die Zugreifbarkeit nicht erhöht werden, da sie sonst von der Methode  $B.m()$  aus zugreifbar wäre und überschrieben würde. Durch das Einführen des Überschreibens würde dann der Methodenaufruf in Zeile 6 nicht mehr an  $A.m()$ , sondern an  $B.m()$  gebunden. In diesem Fall würde sich das Programmverhalten durch das geänderte dynamische Binden ändern.

```
1 package a;
2 public class A {
3     package void m() { } /* d */
4     public void n() {
5         A a= new B();
6         a.m();      //Verhalten verändert !!!
7     }
8 }
9 }
10
11 package b;
12 public class B extends A{
13     public void m() { } /* d' */ //überschreibt A.m nicht, da nicht zugreifbar
14 }
```

Listing 2.11: Beispiel für Dyn-2

### Ovr Überladen

$$\{d, d'\} \subseteq M \wedge \text{overloads}(d', d) \wedge \beta(r) = d \wedge \lambda(d') \in$$

## 2. Problembeschreibung

$$\text{superclasses}(\rho(r)) \cup \{\rho(r)\} \Rightarrow \langle d' \rangle < \alpha(\lambda(r), \lambda(d'))$$

Wenn im Beispiel von Listing 2.12 die Methode `B.m(...)` in die Klasse `A` verschoben würde, so wäre sie für den Aufruf in `B` zugreifbar und würde aufgerufen. Der Grund für den Aufruf bestünde darin, dass nun die Methode `A.m(Object)` von der nach `A` verschobenen Methode `m(String)` überladen wird. Wenn hier also die Zugreifbarkeit nicht angepasst wird, ändert sich die Methodenbindung, und es könnte zu einer Änderung des beobachtbaren Verhaltens der Software kommen. Mittels dieser Regel soll verhindert werden, dass ein

```
1 public class A {
2     /*d1*/ void m(Object o) {...}
3 }
4
5 public class B extends A {
6     /*d2*/ void m(String o) {...}
7 }
8
9 public class C {
10     void n() { /*r1*/(new A()).m("abc")}
11 }
```

Listing 2.12: Beispiel für Ovr [ST09]

Methodenaufruf auf `r`, der an eine Methode `d` gebunden ist, auf eine Methode zugreifen kann, welche die Methode `d'` überlädt und dabei das Binden ändert. Der Zugriffsmodifizierer von `d'` muss also so gewählt sein, dass kein Zugriff vom Ort der Referenzierung  $\lambda(r)$  auf seine Deklaration möglich ist. Es handelt sich hierbei um eine vorausschauende Regel, welche die Situation betrachtet, die entsteht, wenn das Refactoring ausgeführt wäre. In Listing 2.12 wird somit ermittelt, welche Zugreifbarkeit die Methode `B.m(...)` haben müsste, wenn sie in die Klasse `A` verschoben wird. In diesem Fall wäre das „private“ und das Refactoringtool könnte diese Anpassung durchführen, da keine anderes Constraint dem widerspricht.

### Hid Verdecken

$$\iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in (\text{superclasses}(\rho(r)) \cup \{\rho(r)\}) \setminus \text{superclasses}(\lambda(d)) \Rightarrow \langle d' \rangle = \text{absent}$$

In Java verdeckt eine Deklaration eine andere, wenn bei einem Feld eines mit dem gleichen Namen [JLS, § 8.3] bzw. bei einer statische Methode eine mit

## 2. Problembeschreibung

gleicher Subsignatur [JLS, § 8.4.8.2], in einer Basisklasse oder Interface ihrer Klasse, vorhanden ist. Die obige Constraintregel verhindert das Einführen des Verdeckens für referenzierte Deklarationen, da sich dadurch das Programmverhalten verändern kann. Die Constraintregel geht von der Situation aus, dass eine Deklaration  $d$  von einer Referenz  $r$  referenziert wird, und es eine neue Deklaration  $d'$  gibt, deren Bezeichner identisch zu  $d$  ist. Wenn die neue Deklaration  $d'$  in einer Basisklasse des statischen Typs der Referenzierung  $\rho(r)$  oder in diese Klasse selbst eingefügt wird, ist dies nicht zulässig, da das Binden an die bisherige Deklaration  $d$  dadurch auf  $d'$  geändert wird. Das ein Einfügen einer Deklaration an einem Ort nicht durchgeführt werden darf, wird durch den Zugriffsmodifizierer `absent` modelliert. Davon ausgenommen ist der Fall, in dem die neue Deklaration  $d'$  in einer Basisklasse der Klasse, in der  $d$  definiert ist, eingefügt wird, weil sie dann  $d$  nicht verdecken kann. Im Listing 2.13 ist ein Beispiel zur Constraintregel abgebildet. Hier ist es nicht zulässig, ein Feld mit dem gleichen Bezeichner wie `B.i` in den Klassen `C` und `D` zu deklarieren, da dieses dann die Deklaration von `B.i` für die Referenzierung in Zeile 15 verdecken würde.

```
1 public class A {
2     /* int i; hier wäre d' zulässig */
3 }
4
5 public class B extends A {
6     int i; /* d */
7 }
8
9 public class C extends B {
10    /*int i; hier d' nicht*/
11 }
12
13 public class D extends C {
14    /* int i; /* hier d' nicht*/
15    void n() { /*r1*/ i++; }
16 }
```

Listing 2.13: Beispiel für Hid

**Misc** Die Constraintregeln mit der Abkürzung „Misc“ sind nicht formal mit Gleichungen definiert, sondern direkt aus der Java-Sprachdefinition abgeleitet und werden hier in textueller Form präsentiert.

**Misc-1** Zugreifbarkeit von Array-Typen [JLS, § 6.6.1]

## 2. Problembeschreibung

Ein Array Typ ist genau dann zugreifbar, wenn es auch die Typen seiner Elementes sind. Listing 2.14 zeigt ein Beispiel für den Array Typ `A[]` in Zeile 7, bei dessen Referenzierung an diesem Ort sein Elementtyp, der vom Typ `A` ist, den Zugriffsmodifizierer „public“ haben muss.

```
1 package a;
2 public class A {
3 }
4
5 package b;
6 public class B {
7     A[] a; //der Typ A muss hier zugreifbar sein
8 }
```

Listing 2.14: Beispiel für Misc-1

### Misc-2 Zugreifbarkeit von gemeinsam deklarierten Feldern [JLS, § 8.3]

Es müssen alle Felder, die sich in derselben Felddeklaration befinden, die gleiche Zugreifbarkeit besitzen. Im Beispiel von Listing 2.15 sind zwei Felder zusammen deklariert und haben deshalb auch nur einen Zugriffsmodifizierer, der in diesem Fall „public“ ist.

```
1 public class A {
2     public int i, j;
3 }
```

Listing 2.15: Beispiel für Misc-2

### Misc-3 Einstiegsmethoden in ein Programm [JLS, § 12.1.4]

Alle Einstiegspunkte in ein Programm, wie z. B. main-Methoden, JUnit-Tests und öffentliche Schnittstellen, müssen den Zugriffsmodifizierer „public“ aufweisen. Im Beispiel von Listing 2.16 ist eine Einstiegsmethode in Zeile 5 mit dem entsprechenden Zugriffsmodifizierer abgebildet.

```
1 public class A {
2     public static void main(String[] args) { //die main Methode
3     }
4 }
```

Listing 2.16: Beispiel für Misc-3

## 2. Problembeschreibung

### Misc-4 Top-Level-Klasse [JLS, § 7.6]

Nur die Top-Level-Klasse, die einen mit dem Dateinamen der Kompilationseinheit (engl.: compilation unit) identischen Bezeichner hat, darf den Zugriffsmodifizierer „public“ aufweisen. In der Datei A.java als Kompilierungseinheit, die im Listing 2.17 dargestellt ist, darf nur die Klasse A den Zugriffsmodifizierer „public“ haben.

```
1 public class A {  
2 }  
3  
4 class A2 { //für diese Klasse ist public nicht erlaubt  
5 }
```

Listing 2.17: Beispiel für Misc-4 mit Inhalt von Datei A.java

### Misc-5 Einzel importierter Typ [JLS, § 7.5]

Ein Typ, der ohne weitere Verwendung innerhalb einer folgenden Typdeklaration importiert wird, muss innerhalb der importierenden Kompilationseinheit trotzdem zugreifbar sein. Aus diesem Grund muss z. B. eine Klasse, welche in einem anderen Paket importiert wird, den Zugriffsmodifizierer „public“ besitzen. In Listing 2.18 ist ein Beispiel mit der Verletzung dieser Regel veranschaulicht. Der Zugriffsmodifizierer der Klasse a.A müsste hier „public“ sein, da er einzeln in einem anderen Paket b importiert wird.

```
1 package a;  
2 class A { //müsste mit public deklariert sein  
3 }  
4  
5 package b;  
6 import a.A; // Kompilerfehler, da A.a nicht zugreifbar  
7 public class B {  
8 }
```

Listing 2.18: Beispiel für Misc-5

### Misc-6 Importe von Paketen, die Deklarationen mit gleichen Bezeichnern haben (engl.: multiple one demand imports) [JLS, § 7.5]

Wenn Deklarationen über den Import von Paketen importiert werden, so dürfen in diesen Paketen nicht zwei Deklarationen mit dem gleichen Bezeichner

## 2. Problembeschreibung

referenziert werden. Im Beispiel von Listing 2.19 ist eine Verletzung der Regel dargestellt. In Zeile 9 und 10 existieren die Import-Deklarationen, und in Zeile 12 kann der Bezeichner A nicht eindeutig aufgelöst werden, da die Klassen a.A und b.A beide importiert sowie zugreifbar sind.

```
1 package a;  
2 public class A {  
3 }  
4  
5 package b;  
6 public class A {  
7 }  
8  
9 import a.*;  
10 import b.*;  
11 public class B {  
12     A a; //Bezeichner A nicht eindeutig  
13 }
```

Listing 2.19: Beispiel für Misc-6

### 2.4.3. Kategorisierung der Zugreifbarkeits-Constraints

Die oben erläuterten Constraintregeln lassen sich in zwei Kategorien unterscheiden, abhängig davon, wie Verletzungen sichtbar werden. Die erste Kategorie bilden die Regeln, deren Verletzung zu einem Compilerfehler führen. In diese Gruppe lassen sich Acc-1, Acc2, Inh-1, Inh-2, Sub-1, Sub-2, Misc-1, Misc-2, Misc-4, Misc-5 und Misc-6 einordnen. Durch den Compilerfehler lassen sich diese Fehler sehr leicht lokalisieren und werden entdeckt, falls ein Refactoringtool die Zugreifbarkeiten ungenügend berücksichtigt. Fehler der zweiten Kategorie sind wesentlich schwerer zu entdecken. Mitunter ändert sich das beobachtbare Programmverhalten auch nicht unmittelbar, sondern hat erst bei fortschreitenden Implementierungen Auswirkungen. Dieser Gruppe lassen sich die verbleibenden Constraintregeln Dyn-1, Dyn-2, Ovr, Hid und Misc-3 zuordnen.

## 3. Auswahl des Analyseverfahrens

In diesem Kapitel werden die möglichen Verfahren für die Analyse der Refactoringtools diskutiert. Bei der Auswahl ist zu berücksichtigen, dass alle 28 Refactoringtools in Eclipse zu analysieren sind. Ein weiteres Kriterium verkörpert die möglichst vollständige Abdeckung der Fehlermöglichkeiten für das jeweilige Refactoringtool. In den folgenden Abschnitten werden drei Verfahren vorgestellt. Zum Abschluss wird eine Auswahl getroffen und die Entscheidung begründet.

### 3.1. Ausprobieren

Unter Ausprobieren ist hier das unstrukturierte Konstruieren von Beispielen zu verstehen. Hier erfolgt in Bezug auf jedes Refactoringtool der Versuch, Szenarien zu finden, in, denen Zugreifbarkeitsregeln verletzt werden. Das Vorgehen besteht darin, dass Quellcodebeispiele erzeugt und die Refactoringtools darauf angewendet werden. Mit diesen wird versucht, immer wieder neue Beispiele zu konstruieren, bei denen die Zugreifbarkeit unzureichend berücksichtigt wird. Dabei wird unterstellt, dass keine systematische Analyse der Constraintregeln stattfindet. Die Durchführung des Verfahrens erfolgt manuell durch eine Testperson. Das Ausprobieren ist sehr zeitaufwendig und liefert eine schlechte Abdeckung der Möglichkeiten. Bei diesem Verfahren wird stets die Frage zu beantworten sein, ob kein Szenario vergessen wurde.

### 3.2. Toolgestütztes Ausprobieren der Refactoringtools

Im Rahmen des toolgestützten Ausprobierens der Refactoringtools werden alle möglichen Anwendungsstellen automatisiert ausprobiert. Hierbei wird ein

### 3. Auswahl des Analyseverfahrens

Orakel befragt, ob sich das beobachtbare Programmverhalten verändert hat oder Compilerfehler vorhanden sind. Unter einem Orakel versteht man allgemein etwas, das eine Entscheidungsfrage löst. In dem hier vorliegenden Fall dient es zur Beantwortung der Frage, ob das beobachtbare Verhalten der Software erhalten geblieben ist. Hier bieten sich zwei Orakel an, der Compiler und die Unittests mit z. B. JUnit [JUn]. Ein Refactoring wird somit als erfolgreich bewertet, wenn das Programm keine Compilerfehler aufweist bzw. die Unittests ohne Fehler ausgeführt werden können. Eine wichtige Voraussetzung aber besteht darin, dass Unittests für das Projekt, auf denen die Refactoringtools getestet werden, vorhanden sind und diese das Verhalten möglichst vollständig beschreiben. Diese Voraussetzung ist für die meisten Projekte nur sehr eingeschränkt zu erfüllen. Das Verfahren wurde für Tests in der unter [ST09] erwähnten Arbeit verwendet. Dabei wurde ein Testtool mit der Bezeichnung RTT (Refactoring Test Tool) entwickelt, um die Durchführung der Refactorings und Tests automatisiert ablaufen und auswerten zu lassen. Im Quellcode werden dabei mögliche Anwendungsstellen gesucht, auf die sich das entsprechende Refactoringtesttool anwenden lässt und dieses dann darauf angewendet. Es wird für jedes Refactoringtool ein Adapter, der die Benutzung durch das Testtool ermöglicht, implementiert. Eine wichtige Aufgabe des Adapters ist es, die Anwendungsstellen für das Refactoringtool im Quellcode zu ermitteln.

Das Verfahren lässt sich unterschiedlich gut auf die Refactoringtools anwenden. Für das Refactoringtool PullUp ist die Zahl der möglichen Variationen für eine Anwendungsstelle und der damit verbundenen Eingabeparameter relativ einfach. Es muss dabei nur versucht werden, das Refactoring so auszuführen, dass die zu verschiebende Deklaration in alle Basisklassen verschoben wird. Die Zahl der möglichen Eingabeparameter für die Zielklasse ist in diesem Fall auf alle Basisklassen der Klasse, in welcher die Deklaration besteht, beschränkt. Die Ergebnisse beziehen sich dann stets auf eine Basisklasse. Bei dem Refactoringtool Rename gestaltet sich die Anwendung des Verfahrens ungleich schwieriger, da es sehr viele Variationen für eine Anwendungsstelle gibt. So müsste für jeden Bezeichner auf, den das Refactoringtool angewendet werden soll, als Eingabeparameter jeder mögliche Bezeichner ausprobiert werden. Das Ausprobieren aller Variationen wird viel Rechenzeit in Anspruch nehmen. Dabei ist zu bedenken, dass für jede Variation ein Ergebnis produziert wird, das

dann auch auszuwerten ist. Die manuelle Begutachtung der Ergebnisse wird sehr viel Zeit in Anspruch nehmen. Eine Lösung dafür bestünde darin, nur Bezeichner zu verwenden, die im Programm bereits vorkommen und somit eine hohe Wahrscheinlichkeit für einen Einfluss besitzen. Die nächste Verfeinerung wäre dann zu kategorisieren und dabei z. B. die Bezeichner von Methoden für das Umbenennen von Methoden oder den Bezeichner von Feldern für Felder zu verwenden. Nach demselben Schema könnte dann die Ausgabe der Ergebnisse kategorisiert werden. Es ist zu erkennen, dass dabei entsprechende Analysen der möglichen Zugriffsverletzungen notwendig sind, um die Effektivität des Verfahrens zu erhöhen. Es stellt sich hierbei die Frage, ob man nicht gleich analytisch vorgeht. Genau auf diesen Aspekt geht das nächste Verfahren ein.

### 3.3. Analytisches Vorgehen

Die beiden vorangegangenen Verfahren sind gut geeignet, um einen groben Überblick über Fehler in den Refactoringtools zu bekommen. Mittels des letzteren Verfahrens wurden die Constraintregeln für die Zugreifbarkeit validiert [ST09].

Wenn die Constraintregeln schon vorliegen und validiert sind, kann sich die Analyse von Refactoringtools darauf stützen. Diese Möglichkeit wird beim analytischen Vorgehen ergriffen. Der Vorteil ist, dass man dabei analytisch vorgehen kann und sich nur an die Definitionen der Constraintregeln halten muss.

In den Constraintregeln lassen sich veränderliche Bestandteile identifizieren. Es handelt sich dabei nicht ausschließlich um die Constraintvariablen auf der rechten Seite, sondern auch um die Regelprämisse auf der linken Seite der Constraintregel. Diese Veränderlichen zeichnen sich dadurch aus, dass sie bei Änderungen am Programmcode durch Refactoringtools verändert werden und damit die Auswertung der Constraints beeinflussen. Durch diese Beeinflussung kann es notwendig sein, die Zugriffsmodifizierer des Programms anzupassen.

Als Beispiel wird  $\rho(r)$  erläutert. Es handelt sich hierbei um den Ort des statischen Typs eines Aufrufs, der z. B. in Form von `this` bei Verschieben einer Methode in eine andere Klasse geändert wird. Wenn in Listing 3.1 die Methode `A.n(...)` in die abgeleitete Klasse `B` verschoben wird, ändert sich der Ort des

### 3. Auswahl des Analyseverfahrens

```
1 public class A {  
2     /*d1*/int i;  
3     void n() { /*r1*/ i++; }  
4 }  
5  
6 public class B extends A {  
7  
8 }
```

Listing 3.1: Beispiel für Veränderliche

statischen Typs  $\rho(r)$  für die Referenzierung von A nach B durch den impliziten Typwechsel von `this`. Damit ist gezeigt, dass  $\rho(r)$  für mindestens ein Refactoring veränderlich ist und somit im Rahmen dieser Arbeit als Veränderliche betrachtet wird.

Aus den oben aufgeführten Constraintregeln lassen sich die folgenden Veränderlichen extrahieren:  $\lambda(r)$ ,  $\lambda(d)$ , Superklasse, Überschreiben, Überdecken, Überladen, Subsignatur,  $\rho(r)$ ,  $\iota(d)$ , `implements(c,i)` und  $\langle d \rangle$ . Das Extrahieren der Veränderlichen wird durch das Auflisten aller Funktionen auf der linken und rechten Seite der Constraintregeln durchgeführt. Der Bestandteil `inherits(c,d',c')` aus der Constraintregel Sub-2 wird nicht als Veränderliche in die Untersuchung mit aufgenommen, da er nur eine Rahmenbedingung für die Anwendung von Sub-2 darstellt und dabei nicht direkt von den Refactorings beeinflusst wird. Die Veränderlichen  $\lambda(d)$  und `implements(c,i)` reichen aus, um die Beeinflussung von Sub-2 durch Refactoringtools abzubilden. Der Bestandteil  $\beta(r)$  wird nicht zu den Veränderlichen gezählt, da das Binden einer Referenz an eine Deklaration nicht durch Refactorings verändert werden sollte. In den Fällen, bei denen eine Änderung berechtigt ist, hat dies dann aber keine Auswirkungen auf die Zugreifbarkeit, da diese bereits über die Veränderlichen  $\lambda(r)$  und  $\lambda(d)$  geprüft werden.

Das Verfahren beruht darauf, die Veränderlichen der Constraintregeln zu identifizieren und für das jeweilige Refactoringtool festzustellen, ob es diese Veränderlichen beeinflussen kann. Wenn eine Beeinflussbarkeit vorliegt, hat man eine Situation gefunden, in der eine bestimmte Veränderliche im Rahmen eines Refactorings für eine Constraintregel zu untersuchen ist. Für eine Situation werden dann Quellcodebeispiele erstellt und das Refactoringtool dagegen getestet. Im weiteren Verlauf der Arbeit wird der Begriff „Situation“ zur Beschreibung eines speziellen Kontexts, der hinsichtlich der Zugreifbarkeit zu untersuchen

### 3. Auswahl des Analyseverfahrens

ist, verwendet.

An dieser Stelle soll ein kurzes Beispiel das Prinzip erläutern. Aus der Constraintregel Acc-1:  $\beta(r) = d \Rightarrow \langle d \rangle \geq \alpha(\lambda(r), \lambda(d))$  lassen sich beispielsweise die Veränderlichen  $\lambda(r)$  und  $\lambda(d)$  identifizieren. Für alle Refactoringtools, die eine Ortsveränderung für eine Referenz  $\lambda(r)$  durchführen, ergibt sich somit jeweils ein Szenario bezogen auf die Constraintregel Acc-1. Für die Veränderliche  $\lambda(d)$  gilt dies analog.

Eine Situation wird durch die Parameter Veränderliche, Refactoringtool und Constraintregel definiert. Der zu analysierende Raum der Parameter ist also dreidimensional. Im Beispiel von Listing 3.2 ist der erste Parameter der Situation das Refactoringtool PullUp. Den zweiten Parameter bildet die Veränderliche. In dem Beispiel kann man die zwei Veränderlichen  $\lambda(r)$  für den Ausdruck `i++` in Zeile 9 und  $\lambda(d)$  in Zeile 8 für die zu verschiebende Methodendeklaration `B.m(String s)` identifizieren. Für die beiden Veränderlichen kann die Constraintregel Acc-1 angewendet werden, da sie die Ergebnisse der mit ihr erstellten Constraints beeinflussen. Es ergeben sich folglich zwei Situationen, die durch unterschiedliche Parameter definiert sind. Die erste Situation wird durch die Parameter PullUp,  $\lambda(r)$  und Acc-1 beschrieben, die zweite durch PullUp,  $\lambda(d)$  und Acc-1.

```
1 package a;
2 public class A {
3     protected i=0;
4 }
5
6 package b;
7 public class B extends A{
8     void m(String s) { //PullUp in Klasse A
9         i++;
10    }
11 }
```

Listing 3.2: Beispiel für Parameter von Situationen

Für eine Situation können mehrere Quellcodebeispiele in Form von Szenarien existieren. Ein Beispiel hierfür wäre, wenn im Listing 3.2 beide Klassen A und B im gleichen Paket statt in unterschiedlichen Paketen wären. Es würde sich dabei immer noch um die gleiche Situation, z. B. (PullUp,  $\lambda(r)$  und Acc-1), handeln, aber um ein weiteres Szenario zur Fehlersuche. Ferner sind Situationen möglich, für die keine Beispiele konstruierbar sind. Das kann u. a. durch

### 3. Auswahl des Analyseverfahrens

Einschränkungen bei den Refactoringtools vorkommen, die es teilweise nicht erlauben, eine neuen Klasse in einem anderen Paket zu erstellen als dem Paket, in dem die Deklaration, von der das Refactoring ausgeht, deklariert ist.

Das Verfahren ist unabhängig von einer statistisch repräsentativen Quellcodebasis, aus der man beim vorherigen Verfahren Szenarien ermittelt. Es wird hier von einer Situation ausgegangen und hierfür werden dann Szenarien entwickelt. Die Abdeckung des Problemraums ist hier vollständig und nicht abhängig von Quellcodebeispielen wie beim vorherigen Verfahren. Mithilfe dieses Verfahrens können aber auch nicht alle Fehler gefunden werden, da man immer noch davon abhängig ist, dass eine Testperson Beispiele für mögliche Fehlerszenarien findet. Diesen Nachteil birgt auch das erste Verfahren, mit dem Unterschied, dass bei dem hier vorgestellten Verfahren der Testperson ein enger strukturierter Rahmen in Form der Situationen vorgegeben wird. Wenn für eine Situation keine Szenarien mit Fehlern gefunden werden, heißt das nicht, dass deren Existenz ausgeschlossen werden kann. Eine weitere Einschränkung des Verfahrens besteht darin, dass es auf den Constraintregeln basiert und kein Beweis für deren Vollständigkeit existiert.

Das Verfahren ist unabhängig von dem konkret verwendeten Refactoringtool, das ein Refactoring implementiert. Es lassen sich Ergebnisse daraus, wie z. B. die Extraktion der Veränderlichen, auch auf andere Java-Refactoringtools, die unabhängig von Eclipse sind, anwenden.

### 3.4. Auswahl des Verfahrens zur Analyse

Das „Ausprobieren“ besitzt lediglich den Vorteil, dass keine Vorbereitung notwendig ist. Der große Nachteil besteht darin, dass durch die fehlende Systematik keine Aussage über den Grad der Abdeckung für die Untersuchung getroffen werden kann. Unter Abdeckung wird die Berücksichtigung aller Situationen, in denen die Zugreifbarkeit bei einem Refactoring zu beachten ist, verstanden. Das Verfahren hat im Vergleich zum „Analytischen Vorgehen“ den Nachteil, dass der Testperson keine Struktur zur Untersuchung der Refactoringtools zur Verfügung gestellt wird. Die zu untersuchenden Situationen sind nicht systematisch erarbeitet worden und somit nicht definiert, demzufolge können relevante Situationen vergessen werden. Durch den Einsatz von mehr

### 3. Auswahl des Analyseverfahrens

Zeit für die Untersuchung lässt sich die Erkennungsrate für Fehler steigern, es gibt dafür aber sicherlich eine Grenze, ab welcher Aufwand und Nutzen kritisch zu hinterfragen sind.

Das Verfahren „Toolgestütztes Ausprobieren der Refactorings“ birgt den Vorteil, dass es die relevanten Probleme aus dem gegebenen Quellcode hervorbringt. Wenn alle Fehler in diesem Quellcode gefunden und in den Refactoringtools beseitigt wurden, so können diese Fehler in weiteren Projekten nicht mehr vorkommen. Das heißt aber nicht, dass alle Fehler gefunden wurden, sondern es können immer nur die Fehler, die bei dem vorliegenden Quellcode vorkommen, identifiziert werden. Nachteilig ist ferner, dass der Fehler, sofern das Orakel nach der Ausführung des Refactoringtools einen solchen findet, dahin gehend zu bewerten ist, ob es sich wirklich um ein Problem der Zugreifbarkeit handelt. Der damit verbundene Aufwand steigt mit dem Umfang der Codebasis, auf welcher der Test durchgeführt wird. Es gibt weiterhin keine Garantie auf Vollständigkeit, und es ist eine Frage des Zufalls, bei der Auswahl des Quellcodes eine praxisrelevante Abdeckung zu erreichen. Es werden Lücken bleiben, da nur die in dem Quellcode der Testprojekte vorhandenen Szenarien überprüft werden. Wie auch beim ersten Verfahren lässt sich die Erkennungsrate für Fehler durch den Einsatz von mehr Ressourcen steigern. Die Ressourcen wären hier die Rechenzeit, die dem Tool zum Ausprobieren gegeben wird, und das Einrichten von Testprojekten durch Personen. Die beiden ersten Verfahren bieten den Vorteil, dass sie sich auf alle Arten von Fehlern anwenden lassen, was aber bei der hier notwendigen Untersuchung keinen Vorteil bietet, da sie sich ausschließlich auf die Zugreifbarkeit konzentriert.

Das Verfahren „Analytische Vorgehen“ birgt den Vorteil, dass alle Kombinationen aus Constraintregel und Veränderlichen für die jeweiligen Refactoringtools ermittelt und untersucht werden. Im Gegensatz dazu können die anderen beiden Verfahren nur ein unvollständiges Ergebnis liefern, da sie entweder vom Quellcode existierender Programme abhängig sind oder kein Maß für die Vollständigkeit haben. Es entsteht bei dem gewählten Verfahren weiterhin kein Aufwand für das Erstellen von Testtools bzgl. jedes Refactoringtools, da hier nur Szenarien in Form von Quellcodebeispielen konstruiert und ausprobiert werden. Durch die Allgemeingültigkeit des Verfahrens existiert darüber hinaus der Vorteil der Übertragbarkeit auf andere Refactoringtools. Das Verfahren hat die Einschränkung, dass es abhängig von der Existenz der Constraintregeln ist

### *3. Auswahl des Analyseverfahrens*

– diese sind aber für den Aspekt der Zugreifbarkeit verfügbar [ST09].

Als Analyseverfahren wird das „Analytische Vorgehen“ ausgewählt, da insbesondere schon Constraintregeln vorliegen und damit ein strukturiertes Vorgehen geboten wird.

## 4. Detaillierte Beschreibung der Analyse

Nachdem das „Analytische Vorgehen“ als Analyseverfahren ausgewählt wurde, wird in diesem Kapitel das Vorgehen detailliert beschrieben. Im Anschluss werden die allgemeinen Schritte der Analyse, die notwendig sind, um die einzelnen Refactoringtools zu untersuchen, in der Durchführung erläutert und dokumentiert.

### 4.1. Beschreibung des analytischen Verfahrens

Den Ausgangspunkt für die Analyse bilden die Constraintregeln für die Zugreifbarkeit. Diese Constraintregeln ermöglichen es, für spezifische Situationen über Constraints zu jeder Deklaration den Zugriffsmodifizierer so festlegen, dass es im Rahmen des Refactorings nicht zu einer Verhaltensänderung durch geänderte Zugreifbarkeiten kommt. Es gilt, die Situationen mit möglichen Verletzungen ihrer Constraints zu identifizieren und zu bewerten.

Bei der Vorstellung der Constraintregeln ist festgestellt worden, dass eine Situation durch drei Parameter beschrieben wird. Die Parameter sind Refactoringtool, Constraintregel und Veränderliche. Sie lassen sich nicht gut gemeinsam in den drei Dimensionen darstellen. Aus dem Grund der besseren Darstellbarkeit wird das System in ein mehrstufiges zweidimensionales transformiert, das den Vorteil aufweist, in Tabellen abbildbar zu sein. Das allgemeine Prinzip dieser Tabellen wird in Abschnitt 4.1.1 dargestellt. Die erste Stufe bildet die Abhängigkeit zwischen den Constraintregeln und den aus ihnen extrahierten Veränderlichen. Die Abbildung dieser Beziehung wird in Abschnitt 4.1.2 verdeutlicht. Die zweite Stufe ist dann, ausgehend von den Veränderlichen, ihre Beziehung zu den Refactoringtools, was in Abschnitt 4.1.3 aufgezeigt wird. Über die beiden Stufen kann dann für jedes Refactoringtool ermittelt werden,

welche Constraintregeln mit welchen Veränderlichen zu untersuchen sind. In den folgenden Abschnitten werden alle Schritte der Analyse erläutert.

### 4.1.1. Abhängigkeitstabelle

Die Abhängigkeit zwischen den Parametern wird durch sogenannte Abhängigkeitstabellen veranschaulicht. In den Tabellen wird bewertet, inwieweit die Abhängigkeit zwischen den Werten in den Spalten und Zeilen gegeben ist. Die Abhängigkeit repräsentiert dabei eine gerichtete Eigenschaft. Die Kennzeichnung der Richtung erfolgt in der Beschriftung der Zeilen und Spalten. Hierbei ist die mit „abhängig“ beschriftete Größe von der „Eingangsgröße“ abhängig. In den Tabellen zeigt ein Eintrag „x“ eine Abhängigkeit an und „-“ eine nicht vorhandene Abhängigkeit. Eine leere Zelle bedeutet, dass für diese Abhängigkeit keine Bewertung stattgefunden hat. In Tabelle 4.1 ist ein Beispiel für eine Abhängigkeitstabelle angegeben. Es handelt sich dabei um abstrakte Werte Y und X, die nur zur prinzipiellen Erläuterung dienen. Die Komponente Y1 ist von X2 abhängig, und zwischen Y2 und X1 besteht keine Abhängigkeit. Für die restlichen Kombinationen hat keine Bewertung stattgefunden. Die Tabellen in den folgenden Abschnitten werden sich auf die Anwendung im konkreten Bereich der Refactorings beziehen.

	<b>Spaltenbeschriftung</b> (Eingangsgröße)		
<b>Zeilenbeschriftung</b> (abhängig)	<b>X1</b>	<b>X2</b>	<b>X3</b>
<b>Y1</b>		x	
<b>Y2</b>	-		

Tabelle 4.1.: Beispiel für Abhängigkeitstabelle

Bei den Abhängigkeitstabellen im weiteren Verlauf werden die Abkürzungen C1..C3 für Constraintregeln, V1..V3 für Veränderliche und R1..R3 für Refactoringtools verwendet, um die Beispiele zu illustrieren.

### 4.1.2. Abhängigkeit zwischen Constraintregel und Veränderlicher

Die Tabelle 4.2 präsentiert prinzipiell die Abhängigkeiten zwischen Constraintregeln und Veränderlichen an einem abstrakten Beispiel. Bei den Veränderlichen handelt es sich um die Komponenten der Constraintregeln, die durch ihre Veränderung das Ergebnis der Constraints beeinflussen. Die Zugreifbarkeit, die durch die Constraintregeln modelliert wird, ist somit abhängig von den Werten der Veränderlichen.

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)		
	V1	V2	V3
C1		x	
C2		x	x
C3	-		

Tabelle 4.2.: Beispiel für CV-Tabelle

Mit dieser Darstellung der Abhängigkeit für jede Veränderlich kann ermittelt werden, was die Auswertung von Constraints, die aufgrund einer bestimmten Constraintregel erstellt wurden, beeinflussen kann. In dem aufgeführten Beispiel von Tabelle 4.2 kann aus der mit V2 beschrifteten Spalte ermittelt werden, dass durch eine Veränderung von V2 die Constraintregeln C1 und C2 beeinflusst werden und somit zu untersuchen sind. Diese Zuordnung wird im Rahmen des folgenden Analyseschritts benötigt. Als Bezeichnung für diese Tabelle wird im weiteren Verlauf die Abkürzung „CV-Tabelle“ verwendet.

### 4.1.3. Abhängigkeit zwischen Refactoringtool und Veränderlicher

Nachdem im vorangegangenen Abschnitt der Einfluss von Veränderlichen auf die Auswertung der Constraints, die mithilfe von Constraintregeln erstellt werden, abgebildet wurde, ist noch die Frage offen, wie die Veränderlichen beeinflusst werden. Hier wird die Abhängigkeit „Veränderliche von Refactoringtool“ eingeführt. In Tabelle 4.3 ist die Abhängigkeit prinzipiell visualisiert.

#### 4. Detaillierte Beschreibung der Analyse

Man kann in dem gewählten Beispiel erkennen, dass durch das Refactoringtool R2 die Veränderlichen V2 und V3 beeinflusst werden. Als Bezeichnung für die-

Refactoringtool (Eingangsgröße)	Veränderliche (abhängig)		
	V1	V2	V3
R1		x	
R2		x	x
R3	-		

Tabelle 4.3.: RV-Tabelle

se Tabelle wird im weiteren Verlauf die Abkürzung „RV-Tabelle“ verwendet. Es ist für jedes Refactoringtool eine Zeile in die Tabelle RV-Tabelle aufzunehmen und zu bewerten, welche Veränderlichen der Constraintregeln beeinflusst werden.

#### 4.1.4. Ermittlung der zu untersuchenden Fälle

Um herauszufinden, in welchen Fällen ein Refactoringtool mit bestimmten Veränderlichen die Zugreifbarkeiten berücksichtigen muss, werden die RV-Tabelle und die CV-Tabelle angewendet. Aus der RV-Tabelle 4.3 sind die Veränderlichen zu eruieren, welche vom Refactoringtool beeinflusst werden. Im Fall von R2 sind das V2 und V3. Mit den ermittelten Veränderlichen kann nun aus der CV-Tabelle 4.2 der Zusammenhang zu den Constraintregeln hergestellt werden. Dabei wird nicht nur die Constraintregel ermittelt, sondern die Kombination aus Veränderlicher und Constraintregel. Es werden die Spalten für die Veränderlichen mit Angabe der Zeilenbeschriftung aus der CV-Tabelle in eine neue Tabelle kopiert. Im Falle von R2 ergibt sich daraus die resultierende CV-Tabelle 4.4 speziell für das ausgewählte Refactoringtool.

Es können in der neuen Tabelle Zeilen, d. h. Constraintregeln, existieren, die keine Abhängigkeit von den für das Refactoringtool ermittelten Veränderlichen haben. Sie sind dadurch zu erkennen, dass kein „x“ in der Zeile steht. Diese Zeilen können gelöscht werden, und es entsteht daraus die kompaktere CV-Tabelle 4.5.

#### 4. Detaillierte Beschreibung der Analyse

	<b>Veränderliche</b> (Eingangsgröße)	
<b>Constraintregel</b> (abhängig)	<b>V2</b>	<b>V3</b>
<b>C1</b>	x	
<b>C2</b>	x	x
<b>C3</b>		

Tabelle 4.4.: CV-Tabelle für das Refactoringtool R2

	<b>Veränderliche</b> (Eingangsgröße)	
<b>Constraintregel</b> (abhängig)	<b>V2</b>	<b>V3</b>
<b>C1</b>	x	
<b>C2</b>	x	x

Tabelle 4.5.: CV-Tabelle für das Refactoringtool R2 (kompakt)

Für die weitere Analyse eines Refactoringtools wird dessen spezielle CV-Tabelle 4.5 verwendet. Sie enthält für das jeweilige Refactoringtool eine Auswahl potenzieller Problemstellen. Diese Situationen gilt es, auf die Berücksichtigung der Zugreifbarkeit von Deklarationen zu analysieren. Dazu kann aus der Tabelle die Information entnommen werden, welche Veränderungen das Refactoringtool auszulösen hat und welche Constraints daraufhin zu untersuchen sind. Die Veränderung wird dabei durch die Veränderliche repräsentiert.

## 4.2. Veränderliche der Constraintregeln

Der erste Schritt der Analyse besteht darin, den Einfluss der jeweiligen Veränderlichen auf die mit den entsprechenden Constraintregeln erstellten Constraints zu ermitteln. Die Grundlage hierfür bildet die in Abschnitt 4.1.2 vorgestellte CV-Tabelle. Dieser Analyseschritt ist nur einmalig durchzuführen und nur bei Änderungen der Constraintregeln zu wiederholen. Das Ergebnis ist in der konkreten CV-Tabelle 4.6 dokumentiert. Als Eingang für diese Analyse dienen die in Abschnitt 2.4.2 vorgestellten Constraintregeln. Anhand der Constraintregel Acc-1 wird das Vorgehen exemplarisch erläutert. Die Regel Acc-1

#### 4. Detaillierte Beschreibung der Analyse

$\beta(r) = d \Rightarrow \langle d \rangle \geq \alpha(\lambda(r), \lambda(d))$  wird dabei zur Festlegung der Zugreifbarkeit einer Deklaration  $\langle d \rangle$  in ihre Einzelteile zerlegt. Es lassen sich im vorliegenden Fall die Veränderlichen  $\lambda(r)$  und  $\lambda(d)$  ermitteln. Diese beiden Veränderlichen werden jetzt als Einflussfaktoren für Constraints der Constraintregel Acc-1 in der ersten Zeile der Tabelle 4.6 mit einem „x“ markiert. Für die weiteren Constraintregeln wird die Tabelle analog ausgefüllt.

Constraint- regel (abhängig)	Veränderliche (Eingangsgröße)										
	$\lambda(r)$	$\lambda(d)$	Superklasse	Überschreiben	Überdecken	Überladen	Subsignatur	$\rho(r)$	$\iota(d)$	implements(c,i)	$\langle d \rangle$
Acc-1	x	x									x
Acc-2	x	x						x			x
Inh-1		x						x			x
Inh-2	x	x	x						x		x
Sub-1				x	x						x
Sub-2		x								x	x
Dyn-1		x		x							x
Dyn-2		x	x	x			x				x
Ovr	x		x			x		x			x
Hid		x	x					x	x		

Tabelle 4.6.: Ergebnis in CV-Tabelle

Es handelt sich bei den Eintragungen um die Möglichkeit der Beeinflussung und nicht um den Beweis dafür. Eine Abhängigkeit ist eine notwendige, aber keine hinreichende Bedingung für eine Beeinflussung. Für die Zellen der Tabelle, die keine Eintragung aufweisen, beeinflusst die Auswertung einer Veränderlichen mit Sicherheit nicht die nach dieser Regel erstellten Constraints, da sie in ihrer Definition nicht vorkommen.

Durch dieses Verfahren werden die zu untersuchenden Constraintregeln und Situationen erheblich eingeschränkt. Man kann erkennen, dass die in der Tabelle 10 x 11 (Constraintregel x Veränderliche) = 110 vorhandenen Möglichkeiten auf 38 eingeschränkt werden.

#### 4. Detaillierte Beschreibung der Analyse

In Tabelle 4.6 sind die Constraintregeln Misc-1 bis Misc-6, die in Abschnitt 2.4.2 vorgestellt wurden, nicht mit aufgeführt. Sie sind aus den im Folgenden erläuterten Gründen nicht Bestandteil der Untersuchung.

Die Regel Misc-1 ist nicht gesondert zu untersuchen, da es sich als ausreichend erweist, die Zugreifbarkeit des Typen des Array implizit bei der Untersuchung von Acc-1 zu berücksichtigen. Mit der Definition, dass eine Deklaration eines Array-Typen als eine Referenzierung seines Elementtyps gesehen werden kann, ist Acc-1 für die Analyse ausreichend.

Bei Misc-2 existiert kein direkter Einfluss durch Refactoringtools, da es sich hierbei nicht um die Festlegung des richtigen Zugriffsmodifizierers handelt, sondern um die Einhaltung der Syntax von Java, schließlich können zwei gemeinsam deklarierte Felder mit einem sehr einfachen Refactoring auch mit separaten Zugreifbarkeitsmodifizierern ausgestattet werden. Eine Betrachtung im Rahmen dieser Arbeit wird demzufolge hier nicht weiter verfolgt.

Die Zugreifbarkeit der Einstiegsmethoden, welche durch Misc-3 geregelt ist, wird bei der Analyse der Refactoringtools vernachlässigt. Dies ist dem Umstand geschuldet, dass bei den Refactorings die Zugreifbarkeit von Deklarationen eher erhöht als reduziert wird und sich dabei kein Einfluss auf die Einstiegsmethoden ergibt, da sie per Definition den Zugriffsmodifizierer „public“ haben. Die einzigen Ausnahmefälle sind die Refactoringtools, wie z. B. `ChangeMethodSignature`, deren Aufgabe es ist, auf Wunsch des Programmierers, den Zugriffsmodifizierer einer Methode auf einen anderen Wert zu setzen. Bei der Ausführung eines Refactoringtools bei dem der Zugriffsmodifizierer einer Methode mit der Signatur einer Einstiegsmethode geändert wird, ist der Programmierer sich in der Regeln der Konsequenzen bewusst. Eine wesentlich kritische Betrachtung bedürfen Fälle bei denen eine Methode durch ein Refactoring erst zu einer Einstiegsmethode wird. Diese Fälle werden aber durch diese Arbeit nicht weiter verfolgt, da es sich hierbei um ein bewusstes Einführen einer Einstiegsmethode durch den Programmierer handelt sollte.

Für Misc-4 kann es durch die Erhöhung der Zugreifbarkeit auf „public“ zu Verletzungen durch Refactoringtools kommen. Da es sich dabei aber um einen Seiteneffekt bei der Anpassung der Zugriffsmodifizierer durch Refactoringtools aufgrund anderer Constraintregeln handelt, wird dies im Rahmen der hier vorliegenden Untersuchung nicht mit berücksichtigt.

#### 4. Detaillierte Beschreibung der Analyse

Die Misc-5 Constraintregel kann mit der Untersuchung von Acc-1 zusammengefasst werden. Die Constraintregel Misc-5 ist immer dann erfüllt, wenn es auch Acc-1 ist. Die Begründung hierfür ist, dass die Deklarationen, in denen die Referenzierung von Deklarationen aus fremden Kompilierungseinheiten stattfinden, die gleichen Möglichkeiten der Zugreifbarkeit benötigen, wie auch die Imports ihrer Kompilierungseinheit selbst. Unter der Annahme, dass die Referenzierungen und die Imports die gleiche Zugreifbarkeit für ein benötigtes Element haben müssen, ist die Untersuchung von Acc-1 ausreichend.

Um keine Verletzungen der Constraintregel Misc-6 durch Refactoringtools zu erhalten, ist generell vor der Anwendung von Refactoringtools zu empfehlen die Imports neu zu organisieren. In den Eclipse JDT gibt es dazu die Funktion „Organize Imports“, durch die alle Importe von Paketen durch explizite Importe der Deklarationen ersetzt werden. Durch diese Maßnahme kann verhindert werden, dass Refactoringtools nicht ausgeführt werden können, weil die Importe nicht eindeutig sind. Mit dieser generellen Lösung lässt sich das Problem vermeiden und es entfällt die Notwendigkeit Misc-6 im Rahmen dieser Arbeit analytisch für die einzelnen Refactoringtools zu untersuchen.

### 4.3. Durch Refactoringtool beeinflusste Veränderliche

In RV-Tabelle 4.7 wird der Einfluss der Eclipse-JDT-Refactoringtools auf die Veränderlichen abgebildet. Es werden hier die Ergebnisse des in 4.1.3 vorgestellten Analyseschritts mithilfe einer RV-Tabelle dokumentiert. Die Veränderlichen müssen sich dazu nicht direkt auf das für das Refactoringtool ausgewählte Element beziehen, sondern können auch Seiteneffekte auf andere Elemente mit einbeziehen. Bei dem PullUp-Refactoringtool für eine Methode wird beispielsweise nicht nur die Methode selbst verschoben, sondern auch gleichzeitig der Ort von den Referenzierungen, die in dieser Methode stattfinden, verändert.

Die Spalte  $\langle d \rangle$  in Tabelle 4.7 enthält nur direkte Veränderungen der Veränderlichen durch die Refactoringtools. Würde man auch die indirekten Änderungen mit betrachten, so würden alle Refactoringtools, die durch Berücksichtigung der Zugreifbarkeitsregeln nun Zugriffsmodifizierer verändern, einen Einfluss auf  $\langle d \rangle$  besitzen. Diese Betrachtung würde gleichwohl dazu führen, dass alle Konsequenzen bei Änderungen der Zugriffsmodifizierer stufenweise zu berücksichtigen sind. Dieser Umfang würde den Rahmen der hier vorliegenden Untersuchung übersteigen und wird demzufolge nicht mit abgebildet.

Refactoringtool (Eingangsgröße)	Veränderliche der Constraintregeln (abhängig)										
	$\lambda(r)$	$\lambda(d)$	Superklasse	Überschreiben	Überdecken	Überladen	Subsignatur	$\rho(r)$	$\iota(d)$	implements(c,i)	$\langle d \rangle$
Generalize Declared Type	-	x	-	-	-	-	x	x	x	-	-
PullUp (hier Methode)	x	x	-	x	x	x	-	x	-	-	-
PullUp (hier Feld)	x	x	-	-	-	-	-	x	-	-	-
PushDown (hier Methode)	x	x	-	x	-	x	-	x	-	-	-

4. Detaillierte Beschreibung der Analyse

	Veränderliche der Constraintregeln (abhängig)										
	$\lambda(r)$	$\lambda(d)$	Superklasse	Überschreiben	Überdecken	Überladen	Subsignatur	$\rho(r)$	$\iota(d)$	implements(c,i)	$\langle d \rangle$
<b>Refactoringtool</b> (Eingangsgröße)											
PushDown (hier Feld)	x	x	-	-	-	-	-	-	-	-	-
RenameJava Element (hier Instanzmethode)	-	-	-	x	-	x	-	-	x	-	-
Move-Methode	x	x	-	x	x	x	-	x	-	-	-
Encapsulate Field	-	x	-	x	x	x	-	-	-	-	-
Move Class	x	x		x	x	x	-	-	-	-	-
ChangeMethode Signature	x	-	-	x	x	x	x	x	x	-	x
Convert Anonymous	-	-	-	-	-	-	-	-	-	-	-
ConvertLocal Variable	-	x	-	-	-	-	-	-	-	-	-
ConvertMember Type	x	x	-	x	x	x	-	-	-	-	-
ExtractClass	x	x	-	-	x	-	-	x	-	-	-
ExtractConstant	-	x	-	-	-	-	-	-	-	-	-
ExtractInterface	-	x	-	-	-	-	-	x	-	x	-
ExtractLocal Variable	-	-	-	-	-	-	-	-	-	-	-
ExtractMethod	-	x	-	x	x	x	-	-	-	-	-
ExtractSuperclass	x	x	x	x	x	x	-	x	-	-	-
InferGeneric TypeArguments	x	-	-	-	-	-	-	-	-	-	-
InlineConstant	x	-	-	-	-	-	-	-	-	-	-
InlineLocal Variable	-	-	-	-	-	-	-	-	-	-	-
InlineMethode	x	-	-	-	-	-	-	-	-	-	-
IntroduceFactory	x	x	-	x	x	x	-	x	-	-	-
Introduce Indirection	x	x	-	-	x	x	-	x	-	-	-

#### 4. Detaillierte Beschreibung der Analyse

Refactoringtool (Eingangsgröße)	Veränderliche der Constraintregeln (abhängig)										
	$\lambda(r)$	$\lambda(d)$	Superklasse	Überschreiben	Überdecken	Überladen	Subsignatur	$\rho(r)$	$\iota(d)$	implements(c,i)	$\langle d \rangle$
Introduce Parameter	x	-	-	x	x	x	x	-	x	-	-
Introduce ParameterObject	-	-	-	-	-	-	-	-	-	-	-
MoveStatic Member (Methode)	x	x	-	-	x	x	-	x	-	-	-
MoveStatic Member (Feld)	x	x	-	-	-	-	-	x	-	-	-
Use Supertype where possible	x	x	-	x	x	x	x	x	-	-	-
RenameJava Element (Feld)	-	-	-	-	-	-	-	-	x	-	-
RenameJava Element (stat. Methode)	-	-	-	-	x	x	-	-	x	-	-
RenameLocal Variable	-	-	-	-	-	-	-	-	-	-	-

Tabelle 4.7.: Konkrete RV-Tabelle für Untersuchung

Im Folgenden werden die Eintragungen zeilenweise für das jeweilige Refactoringtool erläutert.

**Generalize Declared Type** Bei diesem Refactoringtool wird lediglich der Typ einer Variablendeklaration geändert. Deshalb können sich der Ort einer referenzierten Typdeklaration  $\lambda(d)$  und der statische Typ eines Aufrufs  $\rho(r)$  ändern, da nun ein anderer Typ als vorher benutzt wird und dieser demnach einen anderen Ort aufweist. Weiterhin wird bei der Anwendung auf den Typ eines formalen Parameters die Signatur der entsprechenden Methode geändert und damit auch die Veränderliche Subsignatur beeinflusst. Es kann dadurch aber

#### 4. Detaillierte Beschreibung der Analyse

auch eine Methode identisch zu einer anderen und damit die Veränderliche  $\iota(d)$  beeinflusst werden. Es sind somit diese vier Veränderlichen in der Zeile für das Refactoringtool von Tabelle 4.7 ausgewählt.

**PullUp Methode und PushDown Methode** Für die beiden Refactoringtools sind in ihren jeweiligen Zeilen von Tabelle 4.7 die gleichen Eintragungen vorhanden, da die beiden Refactoringtools ein ähnliches Verhalten offenbaren. Beide verschieben eine Methode innerhalb einer vorhandenen Klassenhierarchie. Durch dieses Verschieben wird die Veränderliche  $\lambda(d)$  entsprechend beeinflusst. Die Änderung  $\lambda(r)$  ergibt sich durch das Verschieben der in der Methode verwendeten Referenzen. Der Parameter  $\rho(r)$  ändert sich durch die Änderung des mitunter nur implizit verwendeten Typs `this`-Referenz beim Verschieben von Methoden- und Feldzugriffen. Die für Methoden nahe liegenden Veränderlichen Überschreiben, Überdecken und Überladen werden beeinflusst. Bei dem Eclipse-JDT-Refactoringtool-PushDown kann es nicht zum Überdecken kommen, da es eine Anwendung auf statische Methoden ausschließt. Die beiden Refactoringtools verändern keine Methodensignaturen, folglich existiert kein Einfluss auf die Veränderliche Subsignatur.

**PullUp Feld und PushDown Feld** Bei den Refactoringtools werden Felder einer Klasse innerhalb der Klassenhierarchie nach oben bzw. nach unten geschoben. Durch das Verschieben der Felddeklaration wird die Veränderliche  $\lambda(d)$  als Repräsentation einer Deklaration verändert. Bei der Deklaration des Feldes wird aber auch der Typ, mit dem es deklariert wird, referenziert und somit  $\lambda(r)$  als Referenz darauf ebenfalls verändert. Die Veränderlichen Überschreiben, Überdecken, Überladen und Subsignatur werden nicht verändert, da diese hier nur für Methoden relevant sind. Der statische Typ von Aufrufen  $\rho(r)$  ändert sich, wenn auf ein Feld von außerhalb seiner Klasse zugegriffen und damit dieser bei der Ortsveränderung der Deklaration im Falle von PullUp aktualisiert wird. Bei PushDown findet keine Aktualisierung statt.

**Rename Java Element (für Instanzmethode)** Es können bei diesem Refactoringtool keine Orte  $\lambda$ , Klassenbeziehungen, Signaturen oder statische Typen verändert werden. Die Änderung des Bezeichners kann aber das Überschreiben oder Überladen von Methoden entfernen oder einführen. Die Voraussetzung für

das Einführen ist hierbei die passende Subsignatur [JLS, § 8.4.2]. Methodensignaturen werden bezüglich ihrer Typen nicht verändert, und auch der statische Typ von Aufrufen erfährt keine Beeinflussung, sodass die Zellen für die Veränderlichen Subsignatur und  $\rho(r)$  in der Tabelle 4.7 für das Refactoringtool als ohne Einfluss gekennzeichnet werden.

**Move-Methode** Der Ort der Methodendeklaration  $\lambda(d)$  und damit auch die Orte der darin enthaltenen Referenzen  $\lambda(r)$  werden im Rahmen des Move-Methode-Refactoringtools in eine andere Klasse verschoben. Da die Methodendeklaration nun in einer anderen Klasse ist, muss sie für alle Orte von Aufrufen  $\lambda(r)$  weiterhin zugreifbar bleiben. Der statische Typ  $\rho(r)$ , mit dem die Methode aufgerufen wird, ändert sich durch den Klassenwechsel für jeden Methodenaufruf. Davon sind insbesondere auch die bisherigen Aufrufe der Methode auf `this` betroffen. Da die Methode in eine andere Klasse verschoben wird, sind Überschreiben, Überdecken und Überladen möglich. Es sind in Tabelle 4.7 dementsprechend Eintragungen für  $\lambda(d)$ ,  $\lambda(r)$ ,  $\rho(r)$  sowie Überschreiben, Überdecken und Überladen vorhanden. Beim Verschieben der Methode wird, wenn der Typ eines Parameters mit dem Typ der zu verschiebenden oder verschobenen Klasse übereinstimmt, die Signatur der Methode geändert. Trotz dieser Änderung der Subsignatur ist die Veränderliche Subsignatur nicht zu untersuchen, da diese Änderung nur relevant ist, wenn sich der Ort der Methode nicht ändert. Durch die Ortsänderung der Methode beim Verschieben in eine andere Klasse sind die Untersuchungen für  $\lambda(d)$  relevant, und die Veränderliche Subsignatur kann vernachlässigt werden.

**Encapsulate Field** Beim Refactoringtool wird der Zugriffsmodifizierer des zu kapselnden Feldes auf „private“ gesetzt, was aber direkt keine Auswirkungen auf die Zugreifbarkeit hat, da seine Referenzierungen durch Getter bzw. Setter ersetzt werden. Es ist eine wichtige Aufgabe des Refactoringtools, die Zugreifbarkeit der Getter und Setter korrekt zu setzen. Dabei reicht es nicht aus, den Zugriffsmodifizierer auf den gleichen Wert, welchen das vorherige Feld aufwies, zu setzen. Der Grund hierfür ist, dass wegen der neuen Deklarationen für Getter und Setter auch deren Auswirkungen an ihrem Ort  $\lambda(d)$  zu untersuchen sind. Es findet hier keine Ortsveränderung von Referenzen  $\lambda(r)$  statt. Weiterhin sind die für die Methoden relevanten Veränderlichen Überschreiben,

Überdecken und Überladen zu berücksichtigen. Da keine Methodensignaturen oder Bezeichner geändert werden, können die Veränderlichen Subsignatur und  $\iota(d)$  vernachlässigt werden. Der Zugriff auf das Feld erfolgt nur in der Klasse, in welcher es deklariert ist. In den anderen Klassen werden anstatt des Feldes nun die Getter und Setter aufgerufen. Die Veränderliche  $\rho(r)$  muss demzufolge nicht berücksichtigt werden, da sich der statische Typ, mit dem die Aufrufe erfolgen, nicht geändert hat.

**Move Class** Das Ziel dieses Refactoringtools besteht darin, eine Klasse in ein anderes Paket zu verschieben. Hierbei wird der Ort von Deklarationen  $\lambda(d)$  und Referenzen  $\lambda(r)$  mitsamt ihrer umschließenden Klasse in ein anderes Paket verschoben. Es sind hierfür die Veränderlichen  $\lambda(d)$  und  $\lambda(r)$  in der Tabelle 4.7 eingetragen, sowie auch Überschreiben, Überdecken und Überladen. Die letzten drei Veränderlichen ergeben sich durch eine geänderte Zugreifbarkeit der Methoden in der Klassenhierarchie. Die Klassenhierarchie selbst wird aber nicht verändert. Die geänderte Zugreifbarkeit ergibt sich daraus, dass nun die Deklarationen der Klasse für bzw. von anderen Klassen in dem Paket in das Verschoben wird zugreifbar werden können und in dem Paket aus dem verschoben wurde nicht mehr zugreifbar sind. Es werden weiterhin auch keine Methodensignaturen, d. h. die Veränderliche Subsignatur, oder statische Typen von Aufrufen  $\rho(r)$  verändert.

**Change Methode Signature** Das Refactoringtool bietet die Möglichkeit, die Signatur einer Methode zu verändern. Es werden bei dem Refactoringtool keine Deklarationen verschoben, weshalb die Veränderliche  $\lambda(d)$  nicht beeinflusst wird. Die in der Signatur neu eingeführten Deklarationen werden vorerst nicht referenziert. Finden Änderungen an Parameter- oder dem Rückgabetypen statt, ergeben sich mitunter neue Referenzen auf Typdeklarationen. Daraus ergibt sich, dass dieser Typ jetzt an einem neuen Ort  $\lambda(r)$  referenziert wird. Bei den Parameter- oder den Rückgabetypen können auch bisher verwendete Typen nicht mehr referenziert werden, was aber keinen Einfluss auf die Zugreifbarkeiten hat. Da die Methodensignatur verändert wird, sind davon die Veränderlichen Überschreiben, Überdecken, Überladen und Subsignatur beeinflusst. Wird der Typ für einen bestehenden formalen Parameter geändert, ist davon der statische Typ der Methodenaufrufe für diesen Parameter und somit

$\rho(r)$  betroffen. Die mögliche Änderung des Methodenbezeichners wird durch die Veränderliche  $\iota(d)$  in Tabelle 4.7 als Einfluss eingetragen. Der Zugriffsmodifizierer der Methode kann durch das Refactoringtool direkt geändert werden und stellt folglich eine Beeinflussung der Veränderlichen  $\langle d \rangle$  dar.

**Convert Anonymous** Es wird eine anonyme Klasse in eine innere Klasse umgewandelt. Da vorher keine Zugriffe auf diese Klasse über ihren eigenen Typ möglich waren, gibt es demzufolge keine Aufrufe, die sie als statischen Typ verwenden. Mithin wird von dieser Klasse nichts direkt von externen Aufrufern aufgerufen, da sie vorher durch den fehlenden Bezeichner nichts anbieten konnte. Es besteht also kein Einfluss auf  $\rho(r)$ . Es können auch bisher keine Klassen von ihr abgeleitet sein. Alles, was durch die Methoden der Klasse vorher überdeckt, überschrieben und überladen wurde, ändert sich nicht, da immer noch die gleichen Zugriffsmöglichkeiten vorhanden sind. Die Klasse ist vor und nach dem Refactoring innerhalb der umschließenden Klasse definiert, und es findet somit keine Änderung der Orte und der damit verbundenen Beeinflussung der Veränderlichen für Referenzen  $\lambda(r)$  und Deklarationen  $\lambda(d)$  statt. An Signaturen, d. h. der Veränderlichen Subsignatur, gibt es keine Veränderungen, und auch Bezeichner  $\iota(d)$  ändern sich nicht. Zusammenfassend kann festgestellt werden, dass keine Beeinflussungen bei den Veränderlichen auftreten. Es gibt insgesamt keine Veränderliche, die beeinflusst wird, und so ist dies auch in der entsprechenden Zeile von Tabelle 4.7 dokumentiert. Es müssen keine weiteren Untersuchungen hierfür durchgeführt werden.

**Convert Local Variable** Eine lokale Variable wird in ein Feld einer Klasse umgewandelt. Es kann dabei für das neue Feld der Zugriffsmodifizierer festgelegt werden. Weiterhin kann das Feld auch mit `static` bzw. `final` deklariert werden. Daraus resultiert die große Auswirkung, dass ein neues Feld in der Klasse deklariert wird und nun außerhalb der Methode, in der es ursprünglich als lokale Variable deklariert war, zugreifbar ist. Der Ort der Referenzierung  $\lambda(r)$  für den Typ der Variablen bleibt weiterhin die Klasse, und es gibt bei dieser Veränderlichen keine Beeinflussung. Es ändert sich  $\lambda(d)$ , da nun eine „neue“ Deklaration in der Klasse vorhanden ist, was vorher nicht der Fall war. Da es sich um ein Feld handelt, können die Veränderlichen Superklasse, Überschreiben, Überdecken, Überladen und Subsignatur vernachlässigt werden. Es ist in

der entsprechenden Zeile in Tabelle 4.7 nur die Zelle für  $\lambda(d)$  als Beeinflussung gekennzeichnet.

**Convert Member Type** In diesem Fall wird ein eingebetteter Typ, der innerhalb einer Klasse definiert ist, aus dieser in eine eigene Datei verschoben, sodass er jetzt ein Typ auf der obersten Ebene ist und nicht mehr ein Member-typ. Es wird der Ort für eine Klassendefinition  $\lambda(d)$  geändert. Der Ort wird von der umschließenden Klasse in das Paket derselben geändert. Die konvertierte Klasse wird folglich eine TopLevel-Klasse. Die in der Klasse enthaltenen Referenzen bekommen dadurch einen anderen Ort  $\lambda(r)$ . Es besteht kein Einfluss auf die Klassenhierarchien, d. h. die Veränderliche Superklasse, da die Beziehung zu Basis- und Subklassen nicht verändert wird. Die Veränderlichen Überschreiben, Überdecken, Überladen sind indirekt durch die neuen Orte betroffen. Der indirekte Einfluss entsteht dadurch, dass nun die umschließende Klasse bzw. auch die eingebettete, Privilegien beim Zugriff auf die jeweils andere verlieren, da die eingebettete Klasse nicht mehr in der umschließenden Klasse deklariert ist. Die Privilegien sind der gegenseitige Zugriff auf die mit „private“ deklarierten Member der anderen Klasse. Da keine Subsignaturen oder statische Typen von Aufrufen  $\rho(r)$  geändert werden, liegt für diese beiden Veränderlichen keine Beeinflussung vor.

**Extract Class** In diesem Fall werden ein oder mehrere Felder einer Klasse in eine neu zu erstellende Klasse verschoben. Es ändert sich somit der Ort der Felddeklarationen  $\lambda(d)$  und dabei auch der Ort für die Referenzierung  $\lambda(r)$  der Typen, mit denen sie deklariert sind. Die Felder werden in eine andere Klasse verschoben, deshalb ändert sich für alle Aufrufe, die bisher für diese Felder stattgefunden haben, der statische Typ  $\rho(r)$ . Es wird weiterhin ein neues Feld mit dem Typ der neuen Klasse eingefügt. Dem neuen Feld obliegt die Aufgabe, die bisherigen Referenzierungen der extrahierten Felder durch Delegation an diese Felder in der neuen Klasse weiterzugeben. Für diese Situation ist  $\lambda(d)$  zu untersuchen. Für  $\lambda(r)$  muss bezüglich des neuen Feldes keine Untersuchung stattfinden, da sich beim Eclipse-JDT-Refactoringtool die neue Klasse im gleichen Paket befindet und so gestaltet sein muss, dass auf sie vom ursprünglichen Ort in jedem Fall zugegriffen werden kann. Im Falle des Eclipse-JDT-Refactoringtools wird dies stets garantiert, da die Zugreifbarkeit für den neuen Typ

und dessen Felder oder ggf. Getter und Setter immer „public“ ist. Von den Orten, an denen die Felder bisher referenziert wurden, muss der Zugriff auf das neue Feld und auf die neue Klasse als Typ möglich sein. Das gilt insbesondere auch für die Subklassen, die in anderen Paketen sein können und über „protected“ vorher Zugriff auf die Felder hatten. Das Refactoringtool ist nicht für statische Felder anwendbar, weshalb diese keine Berücksichtigung finden müssen. Da es sich weder um ein statisches Feld noch um eine Methode handelt, sind Superklasse, Überschreiben, Überdecken, Überladen und Subsignatur als Veränderliche nicht zu berücksichtigen. Es werden nur die Beeinflussungen für  $\lambda(r)$ ,  $\lambda(d)$  und  $\rho(r)$  in Tabelle 4.7 mit aufgenommen.

**Extract Constant** Es entsteht hierbei ein neues statisches Feld, das als Konstante mit `final` deklariert ist. Die neue Deklaration entsteht an einem neuen Ort  $\lambda(d)$ . Es ist folglich nur diese Veränderliche  $\lambda(d)$  zu betrachten und in Tabelle 4.7 einzutragen. Alle anderen Veränderlichen sind nicht zu berücksichtigen, da sie nicht beeinflusst werden. Der Zugriffsmodifizierer ist bei diesem Refactoringtool frei wählbar; es können bisher keine Referenzen auf dieses neue Feld existieren, da es vorher nicht existiert hat. Es ist möglich, alle identischen konstanten Ausdrücke in einer Top-Level-Klasse durch eine Referenz auf das neue Feld zu ersetzen. Diese Funktion ist aber auf diese Klasse beschränkt und nicht für ganze Pakete oder Projekte anwendbar.

**Extract Interface** Es wird von genau einer Klasse ein Interface extrahiert. In dem Interface ist dann eine Teilmenge des Klasseninterfaces dieser Klasse enthalten. Im Refactoringtool der Eclipse JDT erhält das Interface immer den Zugriffsmodifizierer „public“. Das ist aber nicht in allen Fällen notwendig, da bei ausschließlicher Benutzung im eigenen Paket die Standardzugreifbarkeit ausreichen würde. Nachdem bei Referenzen, die bisher den Typ der Klasse hatten, nun der Typ des Interface eingesetzt wird, sind davon die Veränderlichen  $\rho(r)$  und  $\lambda(d)$  beeinflusst. Aufgrund der erstellten und verschobenen Deklarationen des Interfaces ist  $\lambda(d)$  zu untersuchen. Es werden weiterhin auch die Orte für Typreferenzierungen  $\lambda(r)$  in das Interface mit aufgenommen. Es kann hier aber zu keiner Beeinflussung der Veränderlichen kommen, da das Interface bei dem Refactoringtool der Eclipse JDT immer im selben Paket ist wie die Klasse, von der das Interface referenziert wird. Es hat sich somit der Ort für

Zugriffe auf die Typen nicht maßgeblich verändert. Außerdem sind nur Methoden mit dem Zugriffsmodifizierer „public“ für das Extrahieren des Interfaces wählbar. Die Veränderliche  $\text{implements}(c,i)$  wird beeinflusst, da das neue Interface durch die Klasse, aus der es extrahiert ist, implementiert wird.

**Extract Local Variable** Aus einem Ausdruck in einer Methode wird eine lokale Variable erstellt. Es bleibt alles am selben Ort, und Typen kommen nicht hinzu oder werden verändert. Es sind deshalb in der entsprechenden Zeile von Tabelle 4.7 keine Beeinflussungen von Veränderlichen eingetragen.

**Extract Method** Es entsteht hierbei aus einem Codesegment einer Methode eine neue Methode in der gleichen Klasse. Die in der Methode enthaltenen Referenzen ändern ihren Ort, der durch die Klasse definiert ist, nicht. Die Veränderliche  $\lambda(r)$  wird insofern nicht beeinflusst. Durch die neue Deklaration einer Methode werden, wie schon bei anderen Refactoringtools auch, die Veränderlichen  $\lambda(d)$ , Überschreiben, Überdecken und Überladen beeinflusst. Der statische Typ von Aufrufen  $\rho(r)$  ändert sich nicht, da `this` gleich bleibt. Die Veränderliche Subsignatur stellt keinen Einfluss dar, da die Methode vorher nicht existiert hat, um jetzt eine geänderte Subsignatur zu haben.

**Extract Superclass** Es werden Felder und Methoden einer Klasse in eine neue Basisklasse verschoben. Weiterhin kann die neue Klasse auch zu einer Basisklasse für weitere Klassen gemacht werden. Durch diese umfangreichen Änderungen ist die Mehrheit der Veränderlichen beeinflusst, die einzigen Ausnahmen bilden  $\iota(d)$ , Subsignatur,  $\text{implements}(c,i)$  und  $\langle d \rangle$ . Der Grund für die Ausnahmen ist, dass keine Bezeichner für Deklarationen oder Methodensignaturen geändert werden.

**Infer Generic Type Arguments** Es werden durch das Refactoringtool die unparametrisierten Verwendungen von parametrisierbaren Typen durch Typparameter ersetzt. Ein Beispiel hierfür wäre eine Deklaration mit dem Typ „List“, in der Strings gespeichert werden. In diesem Fall würde die Deklaration durch das Refactoring den Typ `List<String>` erhalten. Die Zugreifbarkeit ist hierbei dahin gehend zu berücksichtigen, dass nun auf die Deklaration

„String“ an neuen Orten  $\lambda(r)$  zugegriffen wird. Die Orte von Deklarationen  $\lambda(d)$  werden nicht beeinflusst. Die Veränderlichen Überschreiben, Überladen und Subsignatur werden nicht beeinflusst, da unparametrisierte und parametrisierte Varianten der Deklaration äquivalent hierfür sind [JLS, § 8.4.2, § 4.6] und sich somit nichts ändert. Es werden keine Subsignaturen, statische Typen von Aufrufen oder Bezeichner geändert und folglich auch nicht die Veränderlichen Subsignatur,  $\rho(r)$  und  $\iota(d)$ .

**Inline Constant** Es handelt sich hier um die inverse Operation von Extract Constant. Es existieren zwei Varianten für das Refactoringtool. Bei der ersten Variante bleibt die Konstante erhalten. Es gibt hierbei keinen Einfluss auf Veränderliche. Bei der zweiten Variante werden alle Referenzen auf die Konstante ersetzt, und die Konstante wird dann entfernt. Das Einzige, was sich hierbei ändert, ist  $\lambda(r)$  als Ort der gelöschten Referenz. Es werden vom Refactoringtool alle Referenzen auf die Konstante entfernt. Aus diesem Grund kann es zu keinem geänderten beobachtbaren Programmverhalten durch Änderungen bei  $\lambda(d)$  kommen. Wenn die Konstante eine andere verdeckt hat, so gibt es jetzt keine Referenzen, welche auf die ursprünglich Verdeckte zugreifen können, da sie ersetzt wurde. Wenn die Konstante selbst verdeckt wurde, dann hat dies keinen Einfluss mehr, da die Referenzierungen darauf ersetzt sind. Die Zugreifbarkeit besitzt insofern beim Refactoringtool keinen Einfluss. Es ist die Aufgabe des Refactoringtools, die richtigen Referenzen zu finden, damit es zu keinen Änderungen beim beobachtbaren Programmverhalten kommt. Durch das Ersetzen der Konstante wird der Ausdruck auf der rechten Seite der Wertzuweisung bei ihrer Deklaration an die Stellen kopiert, an denen bisher die Konstante referenziert wurde. Mit diesem Ausdruck kommen ihre Referenzierungen an einen neuen Ort  $\lambda(r)$ . Unter diesem Aspekt ist die Veränderliche  $\lambda(r)$  zu untersuchen.

**Inline Local Variable** Bei diesem Refactoringtool wird eine lokale Variable entfernt und an den Stellen, an denen sie referenziert wurde, der Ausdruck, mit dem sie initialisiert wurde, eingesetzt. Alle Aktionen dieses Refactoringtools beschränken sich auf die Implementierung der Methode, in welcher die lokale Variable entfernt wird. Es werden die Veränderlichen nicht beeinflusst, und die Zugreifbarkeit ist hier nicht zu berücksichtigen.

**Inline Methode** Für das Refactoringtool wird eine Methode ausgewählt und ihre Implementierung an allen Stellen, an denen sie aufgerufen wird, anstatt des Aufrufs eingesetzt. Es werden hierbei alle Aufrufe ersetzt, einschließlich derer, die sich in anderen Klassen oder Paketen befinden. Durch das Einsetzen der Implementierung an den Orten der bisherigen Aufrufe verändert sich der Ort der darin enthaltenen Referenzierungen  $\lambda(r)$ . Die Veränderlichen  $\lambda(d)$ , Superklasse, Überschreiben, Überladen, Subsignatur,  $\rho(r)$  und  $\iota(d)$  werden nicht beeinflusst. Es kommt bei diesen Veränderlichen auch nicht zu einem Einfluss, wenn die ausgewählte Methode komplett gelöscht wird wie als Option im Eclipse-JDT-Refactoringtool möglich. Die Begründung dafür ist, dass die Methode durch das Refactoringtool gelöscht wird und sich damit das Binden von Aufrufen auf jeden Fall ändert, was die Zugreifbarkeit zunächst nicht berührt.

**Introduce Factory** Es wird hier eine neue statische Methode in einer beliebigen Klasse erzeugt und ggf. die Zugreifbarkeit für einen Konstruktor reduziert. Es können somit die Veränderlichen Überschreiben, Überdecken, Überladen beeinflusst werden. Die neue Methode muss von allen Orten, wo dies auch beim Konstruktor möglich war, zugreifbar sein; aus diesem Grund ist die Veränderliche  $\lambda(d)$  zu berücksichtigen. Für die Veränderliche  $\lambda(r)$  gibt es den Zugriff von der neuen Methode auf den Konstruktor. Es ist dabei aber zu beachten, dass er von der neuen Methode aus  $\lambda(r)$  noch zugreifbar sein muss. Veränderungen von Signaturen werden nicht durchgeführt, weshalb die Veränderliche Subsignatur nicht beeinflusst wird. Es ist bei der Untersuchung zu beachten, dass bei dem Refactoringtool die Option existiert, den Konstruktor auf die Zugreifbarkeit „private“ zu setzen. Dieses mögliche Reduzierung ist insbesondere für die Fälle zu beachten, bei denen die Factory-Methode in einer anderen Klasse als der Konstruktor deklariert wird und demzufolge der Zugriff auf den Konstruktor nicht möglich ist.

**Introduce Indirection** Es wird eine neue statische Methode erzeugt, die stellvertretend für die ausgewählte Methode aufgerufen wird. Die Aufrufe, die sie erhält, delegiert sie an die ausgewählte Methode weiter. Der Ort für die neue Methode ist frei wählbar. Die neue statische Methode kann in ihrer Klasse die Veränderlichen Überdecken und Überladen beeinflussen. Die neue Methode

#### 4. Detaillierte Beschreibung der Analyse

entsteht an einem Ort und beeinflusst somit die Veränderliche  $\lambda(d)$  insbesondere für die Aufrufe, welche an die bisherige Methode gesendet wurden. Durch das Delegieren der Aufrufe, referenziert die neue Methode die bisherige Methode, und zwar an einem neuen Ort  $\lambda(r)$ . Mit dem Aktualisieren der Aufrufe ändert sich der statische Typ des Aufrufs und folglich die Veränderliche  $\rho(r)$ , weil sich die neue Methode in einer anderen Klasse befinden kann als die bisherige.

**Introduce Parameter** Es wird ein Ausdruck in einer Methode durch einen neuen formalen Parameter ersetzt und für seine Deklaration ein bestehender Typ verwendet. Die Signatur der betroffenen Methode wird dabei geändert und mithin auch die Veränderliche Subsignatur. Die Veränderlichen Überschreiben, Überladen und Überdecken sind durch die veränderte Methodensignatur mit zu untersuchen. Alle Aufrufer der Methode müssen auf die Typdeklaration  $\lambda(d)$  des neuen Parameters zugreifen können. Die Veränderliche  $\lambda(d)$  wird dadurch nicht beeinflusst, da sich die Orte von Deklarationen, mit Ausnahme des neuen Parameters, nicht ändern. Es wird der Ausdruck, der durch den neuen Parameter ersetzt wird, an alle Stellen, an denen die Methode bisher aufgerufen wurde, kopiert. Es werden damit alle bisherigen Referenzierungen nun an dieser Stelle des bisherigen Aufrufs durchgeführt und nicht mehr an der ursprünglichen. Deshalb entsteht ein Einfluss auf die Veränderliche  $\lambda(r)$ . Der statische Typ für Aufrufe  $\rho(r)$  ändert sich nicht, da keine Aufrufe verändert werden. Innerhalb der Methode gibt es keine relevanten Veränderungen, da dort keine zusätzlichen Zugriffe hinzukommen. Es wird im Falle, dass die Methode eine andere überschrieben hat bzw. von einer anderen überschrieben wurde, auch deren Signatur mit verändert. Dort existiert dann eine neue Referenz auf den Typ des neuen formalen Parameters. Es ist in diesem Fall  $\lambda(r)$  als Veränderliche zu berücksichtigen.

**Introduce Parameter Object** Das Refactoringtool bringt zwei Veränderungen mit sich. Zum Ersten wird eine neue Klasse erzeugt und diese mit den Feldern für die ursprünglichen Parameter versehen. Zum Zweiten werden die Signaturen der Methode verändert, für die das Refactoringtool angewendet wird. Es gibt keine Beeinflussung bezüglich der Methoden, da der neue Parameter einen Typ aufweist, der bisher noch nicht existiert. Durch das Refacto-

ringtool werden keine Orte geändert und somit auch nicht die Veränderlichen  $\lambda(r)$  und  $\lambda(d)$  beeinflusst. Da in Eclipse die neue Klasse und ihre Elemente mit dem Zugriffsmodifizierer „public“ deklariert werden, findet sich keine Beeinflussung für die Veränderlichen. Durch die maximale Zugreifbarkeit sind keine Zugreifbarkeitsregeln vorhanden, die für Referenzen auf die neue Klasse zu beachten sind. Die Veränderliche  $\iota(d)$  wird nicht beeinflusst, da durch die Verwendung eines neuen Typs bisher keine Methode mit dieser Signatur aufgerufen worden konnte.

**Move Static Members (Methode)** Hier wird eine statische Methode von einer Klasse in eine andere verschoben und die Stellen, an denen sie aufgerufen wird, bezüglich dieser Änderung aktualisiert. Durch den veränderten Ort der Methode und ihrer enthaltenen Referenzen werden demzufolge die Veränderlichen  $\lambda(d)$  und  $\lambda(r)$  beeinflusst. Da die Aufrufe aktualisiert werden und den Typ der Klasse erhalten, in welche die Methode verschoben wird, entsteht auch eine Beeinflussung von  $\rho(r)$ . Durch die neue Methode in der Klasse, die das Ziel für das Verschieben ist, sind auch die Veränderlichen Überdecken und Überladen zu berücksichtigen. Da es sich um eine statische Methode handelt, ist Überschreiben nicht betroffen. Es werden keine Klassenhierarchien oder Methodensignaturen verändert und folglich auch nicht die Veränderlichen Superklasse und Subsignatur.

**Move Static Members (Feld)** Es wird ein statisches Feld einer Klasse in eine andere verschoben, und dabei werden ihre Referenzierungen entsprechend angepasst. Die wesentliche Veränderung besteht darin, dass sich der Ort der Referenz und dadurch das verschobene Feld ändert. Diese Änderung wird durch  $\lambda(r)$  abgebildet. Die Veränderlichen Überschreiben, Überdecken, Überladen und Subsignatur werden bei dem Refactoring nicht beeinflusst, da keine Änderungen an Methoden stattfinden bzw. Überdecken aktuell bei den Constraintregeln nur bei Methoden einen Einfluss hat. Bei dem Verschieben der Deklaration des Feldes entsteht ein Einfluss auf  $\lambda(d)$ . Die Aufrufe werden aktualisiert und erhalten den Typ der Klasse, in welche das Feld verschoben wird, weswegen es zu einer Beeinflussung von  $\rho(r)$  kommt.

**Use Supertype Where Possible** Beim Refactoring erfolgt der Versuch, an allen Stellen, an denen ein Typ referenziert wird, diesen durch einen seiner Supertypen (Basisklasse oder Interface) zu ersetzen. Die Veränderliche  $\lambda(r)$  wertet zu einem Ort aus, an dem nun der Supertype referenziert wird, der sich an dem Ort  $\lambda(d)$ , welcher sich damit auch geändert hat, befindet. Die Veränderliche Superklasse wird nicht beeinflusst, da keine Änderungen an der Klassenhierarchie vorgenommen wird. Von der Änderung auf den Supertypen sind auch die Methodenparameter betroffen, weswegen die Veränderlichen Überschreiben, Überdecken, Überladen und Subsignatur beeinflusst werden. Wenn sich der Typ von Deklarationen ändert, hat dies Auswirkungen auf den statischen Typ von Aufrufen  $\rho(r)$ . Es werden keine Bezeichner von Deklarationen verändert und folglich auch nicht  $\iota(d)$ .

**Rename Java Element (Feld)** Die Bezeichnung eines Feldes wird umbenannt. Durch die Umbenennung ist die Veränderliche  $\iota(d)$  zu untersuchen. Weitere Veränderliche werden nicht beeinflusst, da weder Orte noch Aufrufe verändert werden. Die Veränderlichen für Methoden, d. h. Überschreiben, Überdecken, Überladen und Subsignatur werden hier nicht beeinflusst, da nur Änderungen an einem Feld erfolgen und Überdecken aktuell bei den Constraintregeln nur bei Methoden einen Einfluss hat.

**Rename Java Element (statische Methode)** Eine statische Methode bekommt hier einen anderen Bezeichner zugewiesen. Da es sich um eine statische Methode handelt, können durch die Bezeichneränderung das Überdecken oder Überladen eingeführt werden. Die beiden dafür zuständigen Veränderlichen werden somit beeinflusst. Die Veränderliche Überschreiben wird nicht beeinflusst, da sie für statische Methoden keine Anwendung findet. Die Methodensignaturen und damit die Veränderliche Subsignatur werden nicht verändert. Auf die anderen Veränderlichen gibt es keinen Einfluss, da weder Typen noch Orte verändert werden.

**Rename Local Variable** Der Bezeichner einer lokalen Variablen, die in einer Methode definiert ist, wird bei diesem Refactoringtool verändert. Es werden keine Veränderlichen beeinflusst, da die Änderung ausschließlich lokalen

#### 4. *Detaillierte Beschreibung der Analyse*

Einfluss innerhalb des Methodenrumpfes, in dem die Variable deklariert ist, besitzt. Die Zugreifbarkeiten sind hier nicht zu berücksichtigen.

# 5. Untersuchungsergebnisse

In diesem Kapitel werden die Ergebnisse der Untersuchung der jeweiligen Eclipse-JDT-Refactoringtools dokumentiert. Zu Beginn findet sich ein einleitender Abschnitt, der auf die Darstellung der Ergebnisse in den Tabellen eingeht. Um die Ergebnisse besser verstehen zu können, werden sie für ein ausgewähltes Refactoringtool detailliert erläutert. Die Ergebnisdarstellung beinhaltet dabei eine kurze Einführung in dieses Refactoringtool. Im Anschluss werden dann die Ergebnisse für alle weiteren Refactoringtools in kompakter Form durch Tabellen präsentiert. Kapitel 6 befasst sich schließlich mit der Gesamtheit der Ergebnisse.

## 5.1. Interpretation der Tabellen zur Ergebnisdarstellung

Es handelt sich bei den Tabellen mit den Ergebnissen um die Weiterführung der spezifischen CV-Tabellen für das jeweilige Refactoringtool. In den Zellen, in denen bisher eine Abhängigkeit dokumentiert wurde, wird jetzt das Ergebnis der Untersuchung genau dieser Abhängigkeit dargestellt. Es ergibt sich somit für die in diesem Kapitel verwendeten Ergebnistabellen die folgende Legende:

- leer** - Es ist keine Untersuchung notwendig, da diese Situation für das Refactoringtool aufgrund der vorherigen Analyseschritte nicht relevant ist.
- kE** - Eine untersuchte Situation ohne Auswirkungen auf die Zugreifbarkeit.
- EoF** - Die Untersuchung hat eine Auswirkung auf die Zugreifbarkeit ohne Fehler festgestellt.
- EmF** - Die Untersuchung hat eine Auswirkung auf die Zugreifbarkeit und dabei einen Fehler bei der Umsetzung in Eclipse festgestellt.

## 5.2. Detaillierte Ergebnisse für ein Refactoringtool

In diesem Abschnitt wird stellvertretend für alle untersuchten Refactoringtools eines detailliert mit seinen Untersuchungsergebnissen beschrieben. Für diese exemplarische Beschreibung wird das Refactoringtool „PullUp“ ausgewählt. Der Grund für die Auswahl ist, dass es sich hierbei um ein verbreitetes und leicht verständliches Refactoring handelt. Für die Präsentation der Ergebnisse wird die Anwendung des Refactoringtools bezogen auf Methoden betrachtet. Die Anwendung auf Felder wird in einem späteren Abschnitt in der Übersicht veranschaulicht.

Als Einführung wird im Folgenden das Refactoringtool als solches unabhängig von den Ergebnissen beleuchtet. Die anschließende Beschreibung der Ergebnisse orientiert sich in der Reihenfolge an den Veränderlichen, die das Refactoringtool am Quellcode vornimmt, und den damit verbundenen Einflüssen auf die Zugreifbarkeiten.

### 5.2.1. Beschreibung des PullUp-Refactoringtools

Das Refactoringtool hat die Aufgabe, eine Methode innerhalb einer Klassenhierarchie nach oben zu ziehen (engl.: Pull up). Wenn drei Klassen, z. B. „Angestellter“, „Verkäufer“, „Entwickler“ existieren, und wenn eine gleiche Methode in den abgeleiteten Klassen Verkäufer und Entwickler implementiert ist, so kann sie über das Refactoring in die Basisklasse „Angestellter“ gezogen werden. Ein Beispiel dazu ist im Klassendiagramm von Abbildung 5.1 dargestellt.

Die Motivation des Refactorings besteht darin, redundantes Verhalten zu eliminieren. Der redundante Code birgt das Risiko, dass er als Ausgangspunkt für künftige Fehler fungiert und bei Änderungen vergessen werden kann, alle Stellen zu korrigieren [Fow05].

Das Vorgehen beim Refactoring lässt sich vereinfacht wie folgt beschreiben: Es wird eine neue Methode in der Basisklasse erstellt und der Rumpf aus der zu verschiebenden Methode in ihren kopiert. Anschließend werden die zu ersetzenden Methoden in den abgeleiteten Klassen gelöscht.

## 5. Untersuchungsergebnisse

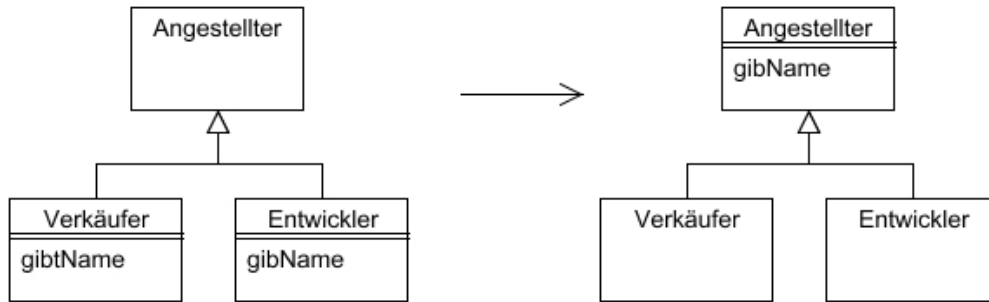


Abbildung 5.1.: Beispiel für PullUp-Methode [Fow05]

In Listing 5.1, das dem Beispiel aus Abbildung 5.1 entstammt, stellt sich dies wie folgt dar: Es gibt in den beiden Klassen die Methode `gibtName()`, welche in die Basisklasse verschoben werden soll. Dabei sind aber auch die in der Methode referenzierten Deklarationen zu beachten und ggf. anzupassen.

```
1 String gibtName(){
2     String titel=gibTitel();
3     return name + titel;
4 }
```

Listing 5.1: Methode in Refactoringbeispiel

In Zeile 2 des Listings wird eine Methode aufgerufen, dieser Aufruf muss auch nach dem Verschieben in die Basisklasse noch funktionieren. Diesbezüglich existieren zwei Möglichkeiten: Einerseits kann die Methode `gibTitel()` bereits in der Basisklasse existieren, andererseits kann `gibTitel()` als abstrakte Methode in der Basisklasse deklariert und dann in den abgeleiteten Klassen implementiert und überschrieben werden.

Das Refactoringtool „PullUp“ in Eclipse bietet die folgenden Funktionen zur Durchführung des Refactorings an: Zum Start benötigt es die Auswahl einer Methode in einer abgeleiteten Klasse und die Information, in welche Basisklasse die Methode zu verschieben ist. Nach dieser Auswahl kann für alle Methoden mit der gleichen Signatur in den abgeleiteten Klassen entschieden werden, ob sie gelöscht werden. Vor der eigentlichen Ausführung werden dann die Ergebnisse des Refactorings als Vorschau präsentiert. Danach gibt es ggf. ein Fenster mit Fehlermeldungen oder Warnungen verbunden mit der Frage, ob das Refactoring wirklich ausgeführt werden soll. Nach Bestätigung der Ausführung

werden die Quellcodedateien entsprechend geändert, und das Refactoring ist ausgeführt.

### 5.2.2. Ergebnisse für PullUp-Methode-Refactoringtool

Basierend auf der RV-Tabelle 4.7 für die Eclipse-JDT-Refactoringtools aus Abschnitt 4.3 ergibt sich speziell für dieses Refactoringtool die nachstehende CV-Tabelle 5.1. Es sind dazu aus der allgemeingültigen CV-Tabelle 4.6 die Spalten entfernt worden, welche keine Veränderlichen des Refactoringtools enthalten. In diesem Fall sind das Superklasse, Subsignatur,  $\iota(d)$ ,  $\text{implements}(c,i)$  und  $\langle d \rangle$ . Für das Refactoringtool sind nach der RV-Tabelle die Veränderlichen  $\lambda(r)$ ,  $\lambda(d)$ , Überschreiben, Überdecken, Überladen und  $\rho(r)$  zu berücksichtigen. Diese Veränderlichen stellen die Spalten in der hier abgebildeten CV-Tabelle dar.

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)					
	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden	$\rho(r)$
Acc-1	X	X				
Acc-2	X	X				X
Inh-1		X				X
Inh-2	X	X				
Sub-1			X	X		
Sub-2		X				
Dyn-1		X	X			
Dyn-2		X	X			
Ovr	X				X	X
Hid		X				X

Tabelle 5.1.: CV-Tabelle für PullUp

Im Folgenden wird die Berücksichtigung der Zugreifbarkeit anhand der Veränderlichen fortlaufend erläutert.

Den Anfang bildet die Veränderliche  $\lambda(r)$ . Es geht hierbei darum, dass sich der Ort von Referenzen ändert und dadurch Zugreifbarkeiten beeinflusst werden können. Diese sind durch die Constraintregeln modelliert und können in der zweiten Spalte für  $\lambda(r)$  mit Acc-1, Acc-2, Inh-2 und Ovr abgelesen werden. Im

## 5. Untersuchungsergebnisse

Methodenrumpf der zu verschiebenden Methode befinden sich Referenzen auf Typen, Methoden- und Felddeklarationen, welche zugreifbar sein müssen. Ein Beispiel hierzu bietet Listing 5.2. Wird die Methode A.m(...) in eine Basis-klasse verschoben, so müssen alle Referenzen an diesem neuen Ort  $\lambda(r)$  in den Constraintregeln berücksichtigt werden. Die einfachste Regel für die Zugreifbarkeit ist Acc-1, die aussagt, dass alle verwendeten Deklarationen von diesem neuen Ort der Referenzen  $\lambda(r)$  noch zugreifbar sein müssen.

```
1 class A extends C{
2   void m(B b) { //Parameter ist Referenz auf d3
3     b.i++; //Referenz auf d1
4     b.n(); //Referenz auf d2
5   }
6 }
7
8 class B { //d3
9   int i; //d1
10  void n() {} //d2
11 }
12
13 class C {}
```

Listing 5.2: Beispiel für Referenzen

Im folgenden Szenario von Listing 5.3 ist ein Beispiel für die Constraintregel Acc-1 abgebildet. Es ist dort zu berücksichtigen, dass die Klassendeklaration d1 und die darin enthaltene Methodendeklaration d2 für die Aufrufer wie r1 zugreifbar bleiben müssen. Durch das Verschieben der Methode B.m() in die Basisklasse A befinden sich nun die enthaltenen Referenzen in einem anderen Paket als die dazugehörigen Deklarationen, C und C.m(). Da diese Deklarationen nur die Standardzugreifbarkeit aufweisen, wären sie ohne Änderung der Zugreifbarkeit auf „public“ nicht mehr zugreifbar.

```
1 package a;
2 public class A {
3 }
4
5 package b;
6 class B extends A {
7   void m() {(new C()).n()} /* r1 */;
8 }
9
10 class C { /* d1 */
11   void n() {...} /* d2 */
12 }
```

Listing 5.3: Szenario  $\lambda(r)$  Acc-1

Das Refactoringtool in Eclipse zeigt zum Abschluss des Refactorings eine War-

## 5. Untersuchungsergebnisse

nung an, dass die Zugriffsmodifizierer von C und C.m() nicht ausreichend sind und diese erhöht werden. In Abbildung 5.2 ist die Warnung abgebildet. Nach der Ausführung sind diese Änderungen aber nicht durchgeführt worden, weshalb das Programm Compilerfehler aufweist. Es ist festzustellen, dass das Problem der Zugreifbarkeit zwar erkannt, aber nicht hinreichend gelöst wurde. Für

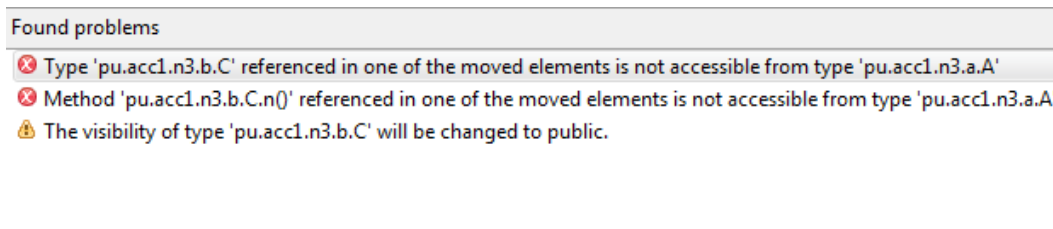


Abbildung 5.2.: Fehlermeldung PullUp-Refactoringtool

die Constraintregel Acc-2 lassen sich keine Szenarien im Kontext der Veränderlichen  $\lambda(r)$  konstruieren. Die Ursache dafür ist, dass für eine Verletzung von Acc-2 eine Referenzierung von einer Methode oder einem Feld eines Methodenparameters vom Typ der Basisklasse in einer Subklasse stattfinden muss, bei der die Zugreifbarkeit nicht ausreicht. Diese Situation lässt sich nicht durch das Verschieben eines Referenzortes  $\lambda(r)$  im Rahmen eines PullUp Refactoring erreichen.

Bei der Constraintregel Inh-2 gibt es ein Szenario, das im Listing 5.4 abgebildet ist. Es wird dort von der Referenz r1 die Deklaration d2 und somit die Konstante I.i referenziert. Durch das Verschieben der Methode C.m() in die Klasse B wird nun auch für die Referenz r1 dort die Deklaration A.i zugreifbar. Es kann nun nicht mehr eindeutig bestimmt werden, welche Deklaration von i zu verwenden ist, woraus ein Compilerfehler resultiert. Die Implementierung in Eclipse bemerkt dieses Problem nicht und führt das Refactoring aus. Aufgrund dessen meldet der Compiler danach, dass in der Methode B.m() die Referenzierung von i nicht eindeutig ist. Mit einer Reduzierung der Zugreifbarkeit von Deklaration A.i auf „private“ hätte der Fehler vermieden werden können.

Die letzte zu untersuchende Constraintregel für  $\lambda(r)$  im Rahmen dieses Refactoringtools bildet Ovr. Es lassen sich hier zwei Szenarien konstruieren, in denen die Zugreifbarkeit zu berücksichtigen ist. Im ersten Szenario aus Listing 5.5 wird die Referenz r1 mithilfe der Methode B.n in die Klasse A verschoben.

## 5. Untersuchungsergebnisse

```
1 package a;
2 public class A {
3     static int i=1; // d1
4 }
5
6 public Interface I {
7     static final int i=2; // d2
8 }
9
10 public class B extends A implements I{
11 }
12
13
14 package b;
15 class C extends B {
16     void m() { int j=i; /* r1 */;}
17 }
```

Listing 5.4: Szenario  $\lambda(r)$  Inh-2

Dieses Verschieben kann aber nicht stattfinden, ohne das beobachtbare Programmverhalten zu verändern. Die Ursache hierfür liegt in der Zugreifbarkeit der Methodendeklaration `A.m(String)`. Diese ist bisher aus `B.n()` nicht zugreifbar, wird es aber durch das Refactoring. Es kann mit dieser Erkenntnis entweder das Refactoring abgebrochen werden, oder die Methode `A.m(String)` kann einen Zugriffsmodifizierer kleiner als „private“ bekommen. Da eine kleinere Zugreifbarkeit als „private“ nicht möglich ist, könnte die Methode `A.m(String)` ggf. gelöscht werden.

```
1 package a;
2 public class A {
3     private void m(String s) {System.out.println("S");} // d1
4     void m(Object o) {System.out.println("O");} // d2
5 }
6
7 public class B extends A{
8     public void n(A a) {
9         a.m(""); // r1
10    }
11 }
```

Listing 5.5: Szenario  $\lambda(r)$  Ovr – gleiches Paket

Im zweiten Szenario, wie in Listing 5.6 abgebildet, wird die Methode im Gegensatz zum vorherigen Beispiel in Bezug auf die referenzierten Methoden sowohl in ein anderes Paket als auch in eine andere Klasse verschoben. Daraus ergibt sich die Möglichkeit, das Verhalten durch Zugriffsmodifizierer zu korrigieren. Es wird die Methode `C.n(...)` in die Basisklasse `B` verschoben. Durch den Ortswechsel der Referenz `r1` von `b.C` nach `a.B` ist nun auch die Metho-

## 5. Untersuchungsergebnisse

de `A.m(String)` zugreifbar, und es würde zum Überladen kommen. Der Aufruf `a.m("abc")` in Zeile 13 wäre durch das eingeführte Überladen dann nicht mehr an `A.m(Object)`, sondern `A.m(String)` gebunden, und das beobachtbare Programmverhalten würde sich ändern. Die Lösung bestünde darin, den von der Constraintregel `Ovr` berechneten Zugriffsmodifizierer für `A.m(String)`, in diesem Falle „private“, zu setzen. Das Refactoringtool erkennt das Problem aber nicht, und es findet auch keine Korrektur des Fehlers durch Setzen der Zugreifbarkeit „private“ für `A.m(String)` statt. Das Kritische an dem Fehler besteht darin, dass es keine Hinweis darauf gibt. Es gibt weder während des Refactorings noch vom Compiler eine Meldung des Fehlers.

```
1 package a;
2 public class A {
3     void m(String s) {System.out.println("S");} // d1
4     public void m(Object o) {System.out.println("O");} // d2
5 }
6
7 public class B extends A{
8 }
9
10 package b;
11 public class C extends B{
12     public void n(A a) {
13         a.m("abc"); // r1
14     }
15 }
```

Listing 5.6: Szenario  $\lambda(r)$  Ovr – unterschiedliche Pakete

Zusammenfassend kann festgestellt werden, dass für die Veränderliche  $\lambda(r)$  drei von vier Constraintregeln nicht beachtet wurden. Von diesen drei Regeln wären aber zwei durch den Compiler gemeldet worden. Die letzte Missachtung der Regeln ist allerdings ohne Hinweise als Fehler in das Programm mit eingegangen.

Der Untersuchung der Veränderlichen  $\lambda(r)$  schließt sich nachfolgend die Untersuchung von  $\lambda(d)$  an. Es sind dafür die Constraintregeln `Acc-1`, `Acc-2`, `Inh-1`, `Inh-2`, `Dyn-1`, `Dyn-2` und `Hid`, wie in CV Tabelle 5.1 angegeben, zu bewerten. Die Veränderung von  $\lambda(d)$  bezieht sich darauf, dass die Deklaration der Methode in ihrem Ort verändert wird. Für `Acc-1` muss die Methode weiterhin von ihren Aufrufern aus zugreifbar sein. Dieser Fall wird im Szenario von Listing 5.7 veranschaulicht. Dort wird die Methode `B.m(...)` in die Klasse `A` geschoben. In der Methode `C.n()` wird die Methode `B.m(...)` aus Zeile 11 aufgerufen und somit referenziert. Um dies zu ermöglichen, muss sie von dort aus zugreifbar

## 5. Untersuchungsergebnisse

sein, und es ist dafür an ihrem neuen Ort, der Klasse A, die Zugreifbarkeit „public“ notwendig. Diese notwendige Änderung wird von Eclipse nicht erkannt, was einen Compilerfehler nach dem Refactoring bedingt.

```
1 package a;
2 public class A {
3 }
4
5 package b;
6 class B extends A {
7     void m(String s) {...} /* d1 */
8 }
9
10 class C {
11     void n() {(new B()).m("a")} /* r1 */;
12 }
```

Listing 5.7: Szenario  $\lambda(d)$  mit Acc-1

Für die Constraintregel Acc-2 wird die Zugreifbarkeit auch nicht ausreichend berücksichtigt, wie anhand des Beispiels in Listing 5.8 aufgezeigt wird.

```
1 package a;
2 public class A {
3 }
4
5 package b;
6 public class B extends A{
7     void n() {}; /* d1*/
8 }
9 public class C extends B {
10     void m() {
11         B b=new B();
12         b.n(); }
13 }
```

Listing 5.8: Szenario  $\lambda(d)$  mit Acc-2

Es wird hier die Methode B.n() in die Basisklasse A verschoben, und daraus würde sich für den Zugriff über eine Referenz vom Typ B der Zugriffsmodifizierer „public“ für diese Methode ergeben. Diese Erweiterung der Zugreifbarkeit wird durch das Refactoringtool nicht durchgeführt und dadurch ein Compilerfehler nach dem Refactoring angezeigt.

Für die Constraints der Constraintregel Inh-1 gibt es in diesem Kontext keine Beeinflussung, da die Zugreifbarkeit der Deklaration des geerbten Elements durch das Refactoringtool nicht verringert wird und es schon vorher zugreifbar sein musste.

## 5. Untersuchungsergebnisse

Auch die Constraints der Constraintregel Inh-2 werden nicht beeinflusst, da diese nur für statische Felder zur Anwendung kommt.

Bei der Constraintregel Sub-2 ist die Zugreifbarkeit zu berücksichtigen, was vom Refactoringtool auch getan wird. Ein Beispiel dazu ist Listing 5.9 zu entnehmen. Wenn dort die Methode B.m() in die Basisklasse A verschoben wird, so muss sie immer noch den Zugriffsmodifizierer „public“ aufweisen, um in Klasse B geerbt die Implementierung der Interface I zu ermöglichen.

```
1 public class A {
2 }
3
4 public class B extends A implements I{
5     public void m() {} //wird in Basisklasse A verschoben
6 }
7
8 public interface I {
9     void m();
10 }
```

Listing 5.9: Szenario  $\lambda(d)$  mit Sub-2

Die Constraints der Constraintregel Dyn-1 werden durch die Veränderliche  $\lambda(d)$  beeinflusst. Es lässt sich hier das in Listing 5.10 gezeigte Szenario konstruieren.

```
1 package a;
2 class A {
3     void m(String s) {...} /* d */
4 }
5
6 package b;
7 class B extends A {
8 }
9
10 package a;
11 class C extends B {
12     void m(String s) {...} /* d' */
13 }
```

Listing 5.10: Szenario  $\lambda(d)$  mit Dyn-1

Durch das Verschieben der Methode C.m(...) in die Basisklasse B kann diese Methode nicht mehr auf die von ihr überschriebene Methode A.m(...) zugreifen. Die Zugreifbarkeit geht deshalb verloren, weil sich B.m(...) nun in einem anderen Paket befindet als A.m(...) und der Zugriffsmodifizierer gegenüber der ursprünglichen Situation, als C.m(...) im gleichen Paket wie A.m(..) war, nicht

## 5. Untersuchungsergebnisse

ausreichend ist. Aus diesem Grund wird die Methode `A.m(...)` nach dem Refactoring nicht mehr überschrieben. Das Programmverhalten kann sich durch das entfernte Überschreiben von `A.m(...)` verändern. Der Fehler lässt sich durch die Erhöhung der Zugreifbarkeit von `A.m(...)` auf „protected“ beheben. Das Refactoringtool gibt zum Abschluss einen Warnhinweis mit der Information, dass das Überschreiben von `A.m(...)` verloren gegangen ist, aus. Die notwendige Korrektur in Form von Setzen des Zugriffsmodifizierers „public“ für die beiden Methoden wird nicht durchgeführt. Zur Sicherheit, dass das Programmverhalten nicht verändert wird, sollte das Refactoringtool anstatt der Ausgabe des Warnhinweises die Möglichkeit der Anpassung der Zugriffsmodifizierer nutzen. Hier und auch bei weiteren Untersuchungen wird das Unterlassen von Korrekturen an den Zugriffsmodifizierern, das eine mögliche Änderung des Programmverhaltens verhindern kann, als Fehler gewertet.

Für die Constraintregel Dyn-2 zeigt sich im Beispiel von Listing 5.11 ein ähnliches Fehlverhalten. Es wird die Methodendeklaration `d1` in die Klasse `B` verschoben. Der Zugriffsmodifizierer der Methode `A.m(...)` müsste auf „private“ geändert werden, um dann das Überschreiben durch `B.m(...)` zu verhindern, da dies vorher auch nicht der Fall war und sich sonst das Programmverhalten ändert. Durch das Refactoringtool erfolgt weder eine Korrektur noch ein Hinweis auf das geänderte Verhalten.

```
1 package a;
2 class A {
3     void m(String s) {...} /* d2 */
4 }
5
6 package a;
7 class B extends A {
8 }
9
10 package b;
11 class C extends B{
12     void m(String s) {...} /* d1 */
13 }
```

Listing 5.11: Szenario  $\lambda(d)$  mit Dyn-2

Die letzte Constraintregel zu  $\lambda(d)$  ist Hid. Im Beispiel von Listing 5.12 wird die Methode `D.m()` in die Klasse `B` verschoben. Dort verdeckt sie aus Sicht der Referenz `r1` die Methode `A.m()`, die dort bisher referenziert wurde. Aus diesem Grund dürfte die neue Methode `B.m()` für `r1` in Zeile 11 nicht zugreifbar sein. Das Refactoringtool müsste demzufolge nach der Prüfung seiner Vorbedingun-

## 5. Untersuchungsergebnisse

gen erkennen, dass durch die Ausführung das Programmverhalten geändert würde, und die Durchführung ablehnen.

```
1 package a;
2 class A {
3     static void m() {} /* d1 */
4 }
5
6 class B extends A { //Basisklasse für C und D
7 }
8
9 class C extends B{
10     void n() {
11         m(); /* r1 */
12     }
13 }
14
15 public class D extends B {
16     static void m() {} /* d2 */
17 }
```

Listing 5.12: Szenario  $\lambda(d)$  mit Hid

Zum Abschluss für die Erörterung von  $\lambda(d)$  ist festzustellen, dass in sechs von acht Situationen die Zugreifbarkeit zu berücksichtigen gewesen wäre, was aber in fünf Situationen nur unzureichend stattgefunden hat.

Im Weiteren werden die Untersuchungsergebnisse für die Veränderliche Überschreiben erörtert. Wird durch das Refactoringtool das Überschreiben von Methoden verändert, und ist dies auch gewollt, so muss dafür die Zugreifbarkeit Berücksichtigung finden. Den Anstoß für die Betrachtung gibt hierbei die Veränderliche Überschreiben. Die Constraintregeln Dyn-1 und Dyn-2 aus CV-Tabelle 5.1 sind hier aber nicht zu berücksichtigen, da davon ausgegangen wird, dass die Änderungen bezüglich des Überschreibens gewollt sind. Die Constraintregel Sub-1 ist hier aber zu analysieren, da sie für Situationen, bei denen das Überschreiben eingeführt wird, relevant ist. In Listing 5.13 wird ein Szenario dargestellt, in dem das Verschieben der Methode C.m() in die Klasse B, welche sich in einem anderen Paket befindet, das Überschreiben der Methode A.m() einführt. Es wird dabei die Constraintregel Sub-1, die das Einschränken der Zugreifbarkeit in der überschreibenden Methode verhindern soll, verletzt. Das Refactoringtool berücksichtigt die Regel nicht, was einen Compilerfehler bedingt.

Ähnlich wie die vorherige Veränderliche Überschreiben kann für statische Methoden die Veränderliche Überdecken beeinflusst werden. In diesem Fall ist

```

1 package a;
2 class A {
3     public void m() {} /* d1 */
4 }
5
6 class B extends A {
7 }
8
9 package b;
10 class C extends B{
11     void m() {} /* d2 */
12 }

```

Listing 5.13: Szenario Überschreiben mit Sub-1

auch die Constraintregel Sub-1 zu beachten. Wie im vorherigen Beispiel wird die Zugreifbarkeit vom Refactoringtool nicht berücksichtigt, woraus ein Compilerfehler resultiert. Der Quellcode für das Szenario unterscheidet sich zu dem im Listing 5.13 lediglich dadurch, dass die Methoden `A.m()` und `B.m()` statisch sind.

Wenn das Überladen von Methoden durch das Refactoring eingeführt wird, so wird davon die Veränderliche Überladen beeinflusst.

```

1 class A {
2     void m(Object o) {...}
3 }
4
5 class B extends A {
6     void m(String s) {...} /* d' */
7 }
8
9 class C {
10     void n() {(new A()).m("a");}
11 }

```

Listing 5.14: Szenario Überladen mit Ovr-1

In Szenario 5.14 wird die Methode `m(...)` von der Klasse B in deren Basisklasse A verschoben. Bei diesem Refactoring wird nun der Aufruf von Methode `m(...)` in der Klasse C anders gebunden, d. h., es wird nun nicht mehr die Methode in der Klasse A mit dem formalen Parameter vom Typ `Object` aufgerufen, sondern die verschobene Methode mit dem formalen Parameter vom Typ `String`. Für das Refactoring wertet die Constraintregel den Zugriffsmodifizierer von `d'` auf kleiner „package“ aus, d. h., er muss dann „private“ sein. Das Refactoringtool führt diese Korrektur nicht durch, und es ändert sich das Programmverhalten. Dabei werden weder Compilerfehler noch andere Fehler-

## 5. Untersuchungsergebnisse

meldungen ausgegeben.

Als letzte Veränderliche wird  $\rho(r)$  untersucht. Es kann beim Refactoring zu einer Beeinflussung durch den `this`-Pointer kommen, der mit dem Verschieben implizit auf die Basisklasse geändert wird. Beim Refactoringtool gibt es aber auch die Option, den Typ aller Referenzen, welche die abgeleiteten Klassen aufrufen, nun auf die Basisklasse ändern zu lassen, falls das möglich ist. Es sind hier nach CV-Tabelle 5.1 die Constraintregeln Acc-2, Inh-1, Ovr und Hid zu analysieren.

Für die Constraintregel Acc-2 gibt es das in Listing 5.15 dargestellte Szenario. Es wird dabei der Typ des formalen Parameters `b` von Methode `B.n(B)` von `B` nach `A` geändert, wenn die Methode `B.n()` in die Basisklasse `A` verschoben wird. Es würde nach Acc-1 die Zugreifbarkeit „protected“ ausreichen, aber durch Acc-2 wird für die neue Methode `A.m()` „public“ benötigt.

```
1 package a;
2 class A {
3 }
4
5 package b;
6 class B extends A {
7     void n(B b) {
8         b.m(); /* r1 */
9     }
10
11     void m() {}; /* d1 */
12 }
```

Listing 5.15: Szenario  $\rho(r)$  mit Acc-2

Nach der Anwendung des Refactoringtools entsteht der in Listing 5.16 veranschaulichte Quellcode. Der Compiler gibt dann in Zeile 9 des Listings einen Fehler aus, mit dem Hinweis, dass die Methode `A.m()` nicht zugreifbar ist. Das Refactoringtool hat durch das Erhöhen der Zugreifbarkeit der verschobenen Methode `A.m()` zwar die Constraintregel Acc-1 berücksichtigt, nicht aber Acc-2. Das Tool arbeitet somit bezogen auf die Regel Acc-2 fehlerhaft.

Für die Constraints der Constraintregel Inh-1 ist kein Einfluss möglich, da durch die Änderung des statischen Typs des Aufrufs hin zur Basisklasse keine Einschränkung der Zugreifbarkeit für geerbte Deklarationen erfolgt. Die Zugriffsmodifizierer der Deklarationen müssen vor dem Refactoring einen Wert aufgewiesen haben, der auch danach noch ausreichend ist. Auch die Constraints der Constraintregel Ovr sind im Rahmen der Veränderlichen  $\rho(r)$  für dieses

```

1 package a;
2 class A {
3     protected void m() {}; /* d1 */
4 }
5
6 package b;
7 class B extends A {
8     void n(A b) {
9         b.m(); /* r1 */ //dann Compilerfehler
10    }
11 }

```

Listing 5.16: Szenario  $\rho(r)$  mit Acc-2 nach Refactoring

Refactoringtool ohne Einfluss. Der Grund dafür ist, dass sich die in der Constraintregel angegebene Bedingung auf den Typ  $\rho(r)$  und die Menge seiner Superklassen bezieht. Durch die Veränderung  $\rho(r)$  durch das Refactoringtool hin zu einer Superklasse gibt es keine Veränderung an dieser Bedingung.

Die Constraintregel Hid-1 ist nicht zu beachten, da sich die statischen Typen von Aufrufen nur auf eine Basisklasse hin ändern können. Dabei kann keine Methode oder kein Feld verdeckt werden, das nicht schon vorher verdeckt war.

Abschließend kann festgestellt werden, dass 21 Situationen als Kombination aus Constraintregel und Veränderlicher zu beurteilen waren. In zwölf dieser Situationen hat sich herauskristallisiert, dass das Verhalten des Refactoringtools unter dem Aspekt der Zugreifbarkeit unzureichend ist. Die Ergebnisse sind in Tabelle 5.2 als Übersicht visualisiert.

### 5.3. Weitere Untersuchungsergebnisse

In diesem Abschnitt werden die Ergebnisse für alle weiteren Eclipse-JDT-Refactoringtools vorgestellt. Es hat für jedes der Refactoringtools eine ausführliche Untersuchung, im gleichen Umfang wie in Abschnitt 5.2.2 für PullUp, stattgefunden. Um eine bessere Übersicht zu gewährleisten, sind in den Tabellen 5.3 bis 5.29 die Ergebnisse in kompakter Form veranschaulicht. Die ausführlichen Ergebnisse mit Codebeispielen sind im Anhang verfügbar.

## 5. Untersuchungsergebnisse

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)					
	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden	$\rho(r)$
Acc-1	EmF	EmF				
Acc-2	kE	EmF				EmF
Inh-1		kE				kE
Inh-2	EmF	kE				
Sub-1			EmF	EmF		
Sub-2		EoF				
Dyn-1		EmF	kE			
Dyn-2		EmF	kE			
Ovr	EmF				EmF	kE
Hid		EmF				kE

Tabelle 5.2.: Untersuchungsergebnisse für PullUp (Methode)

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)			
	$\lambda(d)$	Sub- signatur	$\rho(r)$	$\iota(d)$
Acc-1	EmF			
Acc-2	EmF		EmF	
Inh-1	kE		kE	
Inh-2	kE			kE
Sub-2	kE			
Dyn-1	kE			
Dyn-2	kE	EmF		
Ovr			kE	
Hid	kE		kE	EmF

Tabelle 5.3.: Untersuchungsergebnisse für Generalize Declared Type

## 5. Untersuchungsergebnisse

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)				
	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- laden	$\rho(r)$
Acc-1	EmF	kE			
Acc-2	EmF	kE			kE
Inh-1		kE			EmF
Inh-2	EmF	kE			
Sub-1			EmF		
Sub-2		EmF			
Dyn-1		EmF	EoF		
Dyn-2		EmF	EoF		
Ovr	EmF			EmF	EmF
Hid		kE			EmF

Tabelle 5.4.: Untersuchungsergebnisse für PushDown (Methode)

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)		
	$\lambda(r)$	$\lambda(d)$	$\rho(r)$
Acc-1	EmF	EmF	
Acc-2	kE	EmF	EmF
Inh-1		kE	kE
Inh-2	EmF	EmF	
Sub-2		kE	
Dyn-1		kE	
Dyn-2		kE	
Ovr	EmF		kE
Hid		EmF	kE

Tabelle 5.5.: Untersuchungsergebnisse für PullUp (Feld)

5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)	
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$
Acc-1	EmF	kE
Acc-2	kE	kE
Inh-1		kE
Inh-2	kE	kE
Sub-2		kE
Dyn-1		kE
Dyn-2		kE
Ovr	EmF	
Hid		kE

Tabelle 5.6.: Untersuchungsergebnisse für PushDown (Feld)

	<b>Veränderliche</b> (Eingangsgröße)					
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden	$\rho(r)$
Acc-1	EmF	EoF				
Acc-2	EmF	EoF				EoF
Inh-1		kE				kE
Inh-2	EmF	kE				
Sub-1			kE	kE		
Sub-2		kE				
Dyn-1		kE	kE			
Dyn-2		EmF	kE			
Ovr	EmF				kE	EmF
Hid		kE				kE

Tabelle 5.7.: Untersuchungsergebnisse für Move-Methode

## 5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)			
<b>Constraintregel</b> (abhängig)	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden
Acc-1	EoF			
Acc-2	EoF			
Inh-1	EoF			
Inh-2	kE			
Sub-1		EmF	EmF	
Sub-2	kE			
Dyn-1	kE	kE		
Dyn-2	EmF	EoF		
Ovr				EmF
Hid	EoF			

Tabelle 5.8.: Untersuchungsergebnisse für EncapsulateField

	<b>Veränderliche</b> (Eingangsgröße)				
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden
Acc-1	EmF	EmF			
Acc-2	EmF	EmF			
Inh-1		EmF			
Inh-2	EmF	EmF			
Sub-1			kE	kE	
Sub-2		kE			
Dyn-1		EmF	kE		
Dyn-2		EmF	kE		
Ovr	EmF				kE
Hid		kE			

Tabelle 5.9.: Untersuchungsergebnisse für Move Class

## 5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)							
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	Über- schreiben	Über- decken	Über- laden	Sub- signatur	$\rho(r)$	$\iota(d)$	$\langle d \rangle$
Acc-1	EoF							EmF
Acc-2	kE					EoF		EmF
Inh-1						EoF		EoF
Inh-2	kE						kE	kE
Sub-1		EoF	EoF					EoF
Sub-2								EoF
Dyn-1		kE						EmF
Dyn-2		EoF			EoF			EmF
Ovr	kE			EoF		EoF		EmF
Hid						EoF	EoF	

Tabelle 5.10.: Untersuchungsergebnisse für ChangeMethodeSignature

	<b>Veränderliche</b> (Eingangsgröße)
<b>Constraintregel</b> (abhängig)	$\lambda(d)$
Acc-1	kE
Acc-2	kE
Inh-1	kE
Inh-2	EmF
Sub-2	kE
Dyn-1	kE
Dyn-2	kE
Hid	EmF

Tabelle 5.11.: Untersuchungsergebnisse für ConvertLocalVariable

## 5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)				
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden
Acc-1	EoF	EoF			
Acc-2	kE	kE			
Inh-1		kE			
Inh-2	kE	EmF			
Sub-1			EmF	EmF	
Sub-2		kE			
Dyn-1		kE	kE		
Dyn-2		EmF	EmF		
Ovr	kE				EmF
Hid		EmF			

Tabelle 5.12.: Untersuchungsergebnisse für ConvertMemberType

	<b>Veränderliche</b> (Eingangsgröße)			
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$	Über- decken	$\rho(r)$
Acc-1	EmF	EoF		
Acc-2	kE	EoF		kE
Inh-1		EoF		kE
Inh-2	kE	kE		
Sub-1			kE	
Sub-2		kE		
Dyn-1		kE		
Dyn-2		kE		
Ovr	kE			kE
Hid		EmF		kE

Tabelle 5.13.: Untersuchungsergebnisse für ExtractClass

## 5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)
<b>Constraintregel</b> (abhängig)	$\lambda(d)$
Acc-1	kE
Acc-2	kE
Inh-1	kE
Inh-2	EmF
Sub-2	kE
Dyn-1	kE
Dyn-2	kE
Hid	EoF

Tabelle 5.14.: Untersuchungsergebnisse für ExtractConstant

	<b>Veränderliche</b> (Eingangsgröße)		
<b>Constraintregel</b> (abhängig)	$\lambda(d)$	$\rho(r)$	implements(c,i)
Acc-1	kE		
Acc-2	kE	kE	
Inh-1	kE	kE	
Inh-2	EmF		
Sub-2	kE		kE
Dyn-1	kE		
Dyn-2	kE		
Ovr		kE	
Hid	kE	kE	

Tabelle 5.15.: Untersuchungsergebnisse für ExtractInterface

## 5. Untersuchungsergebnisse

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)			
	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden
Acc-1	kE			
Acc-2	kE			
Inh-1	kE			
Inh-2	kE			
Sub-1		EoF	EoF	
Sub-2	kE			
Dyn-1	EoF	EoF		
Dyn-2	EoF	EoF		
Ovr				EoF
Hid	EoF			

Tabelle 5.16.: Untersuchungsergebnisse für ExtractMethod

Constraint- regel (abhängig)	Veränderliche (Eingangsgröße)						
	$\lambda(r)$	$\lambda(d)$	Super- klasse	Über- schreiben	Über- decken	Über- laden	$\rho(r)$
Acc-1	EmF	EoF					
Acc-2	kE	kE					kE
Inh-1		EmF					kE
Inh-2	kE	EmF	kE				
Sub-1				EmF	EmF		
Sub-2		EmF					
Dyn-1		EoF		kE			
Dyn-2		kE	kE	EmF			
Ovr	kE		EmF			EmF	kE
Hid		EmF					kE

Tabelle 5.17.: Untersuchungsergebnisse für ExtractSuperclass

## 5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)
<b>Constraintregel</b> (abhängig)	$\lambda(r)$
Acc-1	kE
Acc-2	kE
Inh-2	kE
Ovr	kE

Tabelle 5.18.: Untersuchungsergebnisse für InferGenericTypeArguments

	<b>Veränderliche</b> (Eingangsgröße)
<b>Constraintregel</b> (abhängig)	$\lambda(r)$
Acc-1	EmF
Acc-2	kE
Inh-2	kE
Ovr	kE

Tabelle 5.19.: Untersuchungsergebnisse für InlineConstant

5. Untersuchungsergebnisse

	<b>Veränderliche</b> (Eingangsgröße)
<b>Constraintregel</b> (abhängig)	$\lambda(r)$
Acc-1	EmF
Acc-2	EmF
Inh-2	EmF
Ovr	EmF

Tabelle 5.20.: Untersuchungsergebnisse für InlineMethode

	<b>Veränderliche</b> (Eingangsgröße)					
<b>Constraintregel</b> (abhängig)	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden	$\rho(r)$
Acc-1	EmF	EmF				
Acc-2	kE	kE				kE
Inh-1		kE				kE
Inh-2	kE	kE				
Sub-1			kE	EoF		
Sub-2		kE				
Dyn-1		kE	kE			
Dyn-2		kE	kE			
Ovr	kE				EmF	EmF
Hid		EmF				kE

Tabelle 5.21.: Untersuchungsergebnisse für IntroduceFactory

## 5. Untersuchungsergebnisse

Constraint- regel (abhängig)	Veränderliche (Eingangsgröße)					
	$\lambda(r)$	$\lambda(d)$	Über- decken	Über- laden	Sub- signatur	$\rho(r)$
Acc-1	EmF	kE				
Acc-2	EoF	kE				kE
Inh-1		kE				kE
Inh-2	kE	kE				
Sub-1			kE			
Sub-2		kE				
Dyn-1		kE				
Dyn-2		kE			kE	
Ovr	kE			EmF		EmF
Hid		EmF				kE

Tabelle 5.22.: Untersuchungsergebnisse für IntroduceIndirection

Constraint- regel (abhängig)	Veränderliche (Eingangsgröße)					
	$\lambda(r)$	Über- schreiben	Über- decken	Über- laden	Sub- signatur	$\iota(d)$
Acc-1	EmF					
Acc-2	EmF					
Inh-2	EmF					kE
Sub-1		EmF	EmF			
Dyn-1		kE				
Dyn-2		EmF			EoF	
Ovr	EmF			EmF		
Hid						EmF

Tabelle 5.23.: Untersuchungsergebnisse für IntroduceParameter

## 5. Untersuchungsergebnisse

<b>Constraintregel</b> (abhängig)	<b>Veränderliche</b> (Eingangsgröße)				
	$\lambda(r)$	$\lambda(d)$	Über- decken	Über- laden	$\rho(r)$
Acc-1	EoF	EoF			
Acc-2	EmF	kE			kE
Inh-1		kE			kE
Inh-2	kE	kE			
Sub-1			EmF		
Sub-2		kE			
Dyn-1		kE			
Dyn-2		kE			
Ovr	EmF			EmF	EmF
Hid		EmF			EmF

Tabelle 5.24.: Untersuchungsergebnisse für MoveStaticMembers (Methode)

<b>Constraintregel</b> (abhängig)	<b>Veränderliche</b> (Eingangsgröße)		
	$\lambda(r)$	$\lambda(d)$	$\rho(r)$
Acc-1	EoF	EoF	
Acc-2	kE	kE	kE
Inh-1		kE	kE
Inh-2	kE	EmF	
Sub-2		kE	
Dyn-1		kE	
Dyn-2		kE	
Ovr	kE		kE
Hid		EmF	kE

Tabelle 5.25.: Untersuchungsergebnisse für MoveStaticMembers (Feld)

5. Untersuchungsergebnisse

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)						
	$\lambda(r)$	$\lambda(d)$	Über- schreiben	Über- decken	Über- laden	Sub- signatur	$\rho(r)$
Acc-1	EmF	kE					
Acc-2	kE	kE					EmF
Inh-1		kE					kE
Inh-2	kE	kE					
Sub-1			EmF	EmF			
Sub-2		kE					
Dyn-1		kE	kE				
Dyn-2		kE	kE			EmF	
Ovr	kE				EmF		kE
Hid		kE					kE

Tabelle 5.26.: Untersuchungsergebnisse für UseSupertypeWherePossible

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)
Inh-2	EmF
Hid	EoF

Tabelle 5.27.: Untersuchungsergebnisse für RenameJavaElement (Feld)

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)		
	Über- decken	Über- laden	$\iota(d)$
Inh-2			EoF
Sub-1	EmF		
Ovr		EoF	
Hid			EoF

Tabelle 5.28.: Untersuchungsergebnisse für RenameJavaElement (statische Methode)

## 5. Untersuchungsergebnisse

Constraintregel (abhängig)	Veränderliche (Eingangsgröße)		
	Über- schreiben	Über- laden	$\iota(d)$
Inh-2			kE
Sub-1	EmF		
Dyn-1	kE		
Dyn-2	EmF		
Ovr		EmF	
Hid			kE

Tabelle 5.29.: Untersuchungsergebnisse für RenameJavaElement (Methode)

## 6. Diskussion der Ergebnisse

In diesem Kapitel werden die Ergebnisse der Analyse diskutiert. Bei der Interpretation der Ergebnisse wird die Frage beantwortet, in welchem Umfang die Refactoringtools die Zugreifbarkeit berücksichtigt haben. Bei der Bewertung wird das Ausmaß der Fehler festgestellt.

### 6.1. Interpretation der Ergebnisse

Bei der Untersuchung der Refactoringtools wurden die durch Constraint und Veränderliche definierten Situationen bewertet. Die Situationen wurden dabei in die folgenden Kategorien eingeteilt. Die erste Kategorie ist „kein EinflusskE“, bei der das Refactoringtool die Situation nicht beachten muss, weil es die Constraintregel mit der Änderungen, die es vornimmt, nicht verletzen kann. Ein Beispiel hierfür wäre, dass zwar ein Refactoringtool Orte von Deklarationen  $\lambda(d)$  verändert, aber die Constraintregel Inh-2 trotzdem nicht anwendbar ist, weil es dort die Einschränkung gibt, dass sie nur für statische Felder anzuwenden ist. In die zweite Kategorie „Einfluss ohne Fehler-EoF“ sind Situationen einzuordnen, bei denen das Refactoringtool die Zugreifbarkeit zu berücksichtigen hat und dies umfangreichen, händischen Tests nach scheinbar auch fehlerfrei umsetzt. Die letzte Kategorie „Einfluss mit Fehler-EmF“ nimmt die Situationen auf, bei denen sich Szenarien konstruieren lassen, in denen das Refactoringtool die Zugreifbarkeit nicht ausreichend berücksichtigt. In Abbildung 6.1 ist die Verteilung der Situationen auf die Kategorien zu sehen.

Die gefundenen Fehler lassen sich weiter nach den Constraintregeln differenzieren, welche sie verletzen. Es werden in Abbildung 6.2 die Ergebnisse pro Constraintregel dargestellt. Die Säule „Einfluss Gesamt“ gibt die Summe aller



Abbildung 6.1.: Verteilung der Situationen nach Einfluss und Fehlern

Situationen für eine Constraintregel an, in denen die Zugreifbarkeit zu berücksichtigen ist. In der Säule „Einfluss mit Fehler“ sind alle Situationen, in denen die Constraintregeln nicht beachtet werden und Fehler entstehen, abgebildet.

Einen weiteren Gesichtspunkt bei der Interpretation der Ergebnisse bildet die Verteilung der Fehler zwischen Constraintregeln, deren Verletzung durch den Compiler gemeldet wird, und denen, wo das Programmverhalten ohne Meldung geändert wird. In Abbildung 6.3 wird nach diesen beiden Gesichtspunkten unterschieden. Bei dem geänderten Programmverhalten ohne Compilerfehler lässt sich dann noch weiter unterscheiden in solche, die lautlos passieren, und in solche, bei denen ein Hinweis gegeben wird, der aber unzureichend auf die Probleme eingeht.

Es ist festzustellen, dass in mehr als der Hälfte der Situationen die Änderung des beobachtbaren Programmverhaltens ohne einen gemeldeten Fehler durch den Compiler passiert wäre. Die Bewertung ist allerdings dahin gehend zu relativieren, dass bei einigen Refactoringtools darauf hingewiesen und auch das Angebot unterbreitet wird, das Verhalten zu korrigieren. Dabei kommt es sehr oft vor, dass die angekündigten Änderungen nicht durchgeführt wurden, was vom Autor genauso negativ bewertet wird, als wäre der Fehler nicht gemeldet worden.

## 6. Diskussion der Ergebnisse

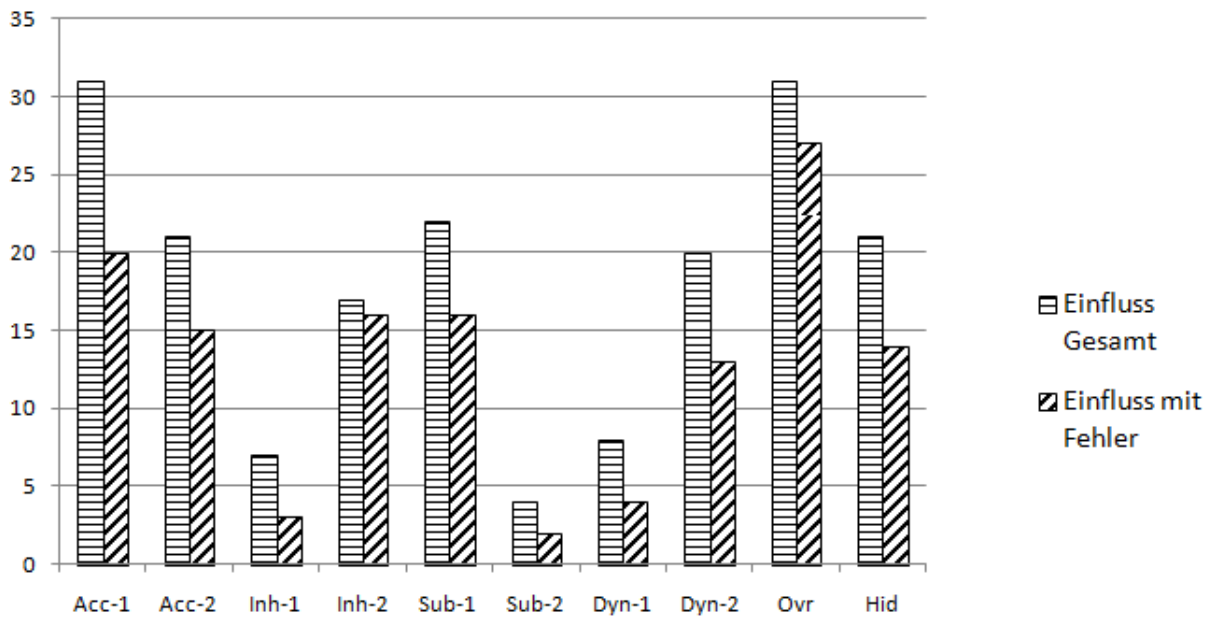


Abbildung 6.2.: Ergebnis nach Constraintregeln



Abbildung 6.3.: Differenzierung der Fehler – mit oder ohne Compilermeldung

## 6.2. Bewertung der Ergebnisse

Es ist festzustellen, dass alle Refactoringtools durch mangelnde Berücksichtigung der Zugreifbarkeit Fehler im Programm erzeugen. Die Ergebnisse zeigen auf, dass bei dem analytischen Verfahren die Hälfte der ermittelten Situationen keine Berücksichtigung durch das jeweilige Refactoringtool benötigt. Trotz dieser Tatsache hat sich das Verfahren bewährt, um überhaupt den Raum für die Situationen einzuschränken und diese zu definieren. Bei der Untersuchung haben sich diese Situationen auch in kurzer Zeit herauskristallisieren lassen. In den anderen Situationen, bei denen die Berücksichtigung notwendig ist, gab es eine erhebliche Anzahl von Fehlern, die ermittelt wurden. Besonders hervorzuheben sind hier die Fehler, die nicht vom Compiler gemeldet werden, sondern teils still das beobachtbare Programmverhalten ändern. Diese Fehler sind besonders kritisch, da sie nicht sofort vom Programmierer erkannt werden. Sie treten erst zur Laufzeit auf, und im günstigsten Fall werden sie durch Unittests gefunden. Besonders negativ ist, dass jeder zweite Fehler in diese Fehlerklasse einzuordnen ist. Diese fehlerhaften Refactoringtools können vorher nicht vorhandene Fehler in ein Programm einbauen. Dieses Problem kann sich negativ auf die Akzeptanz der Refactoringtools auswirken.

Es konnten bei der Untersuchung insgesamt 130 Fehler festgestellt werden. Diese Zahl ist sehr hoch und zeigt die mangelnde Berücksichtigung der Zugreifbarkeiten bei der Implementierung der Refactoringtools. Die Verteilung der Fehler ist hierbei unabhängig von den Constraintregeln gleichmäßig auf diese verteilt. Fast jeder zweite Fehler wurde dabei nicht durch den Compiler angezeigt und hat somit das Programmverhalten ohne Hinweis verändert.

# 7. Schlussbetrachtungen

## 7.1. Zusammenfassung

Die vorliegende Arbeit hegte den Anspruch, die Refactoringtools der Eclipse JDT 3.4.2 hinsichtlich der Berücksichtigung von Zugreifbarkeiten zu untersuchen. Durch die Auswahl und Entwicklung des „Analytischen Verfahrens“ konnte eine effektive Untersuchung aller Refactoringtools durchgeführt werden. Die Analyse hat sich dabei auf die Constraintregeln für die Zugreifbarkeiten gestützt und ihre Anwendbarkeit auf die verschiedenen Refactoringtools unter Beweis gestellt. Das gewählte Verfahren hat sich als gute Wahl erwiesen. Das angewendete Verfahren basiert nur auf den Constraintregeln und kann folglich leicht auf weitere Refactoringtools angewendet werden. Die Aussagen der Untersuchung sind insbesondere nicht abhängig von vorgegebenen Quellcode. Die Systematik des Vorgehens garantiert, dass keine Situation<sup>1</sup> vergessen wird und alle bewertet werden. Bei der Analyse sind entsprechende Szenarien mit den Refactoringtools getestet worden. Diese Szenarien, welche als Quellcodes vorliegen, können in anderen Refactoringtools ohne weitere Anpassungen zur Prüfung herangezogen werden. Es ist somit möglich, diese, aufbauend auf den Ergebnissen dieser Arbeit, ebenfalls zu analysieren.

Im Rahmen der Analyse wurden sehr viele Situationen aufgedeckt, in denen die Zugreifbarkeiten durch die Refactoringtools der Eclipse JDT nur unzureichend beachtet wurden. Insbesondere der hohe Anteil der Situationen, in denen sich das beobachtbare Programmverhalten ohne für den Entwickler erkennbar verändert hat, ist sehr bedenklich. Die aufgedeckten Fehler unterstreichen den Stellenwert der Untersuchung und demonstrieren ihre Notwendigkeit.

Die Korrekturen der gefundenen Fehler sind von Bedeutung, um verlässliche Refactoringtools in Eclipse JDT zu besitzen. Die Refactoringtools übernehmen

---

<sup>1</sup>vgl. Definition auf Seite 29

bei den automatisierten Refactorings eine hohe Verantwortung und müssen deshalb fehlerfrei arbeiten.

### 7.2. Ausblick

Die gefunden Fehler können in die Eclipse Community [ECL] eingebracht werden. Dazu bietet es sich an, Fehlereinträge zu erstellen und Unittests zu schreiben, die das fehlerhafte beobachtbare Verhalten der Software in Form von negativen Testresultaten dokumentieren. Durch die Tests kann die Korrektur der Fehler testgetrieben entwickelt und geprüft werden. Das analytische Verfahren ist leicht auf andere Refactoringtools für Java anwendbar. Dort müssen die beschriebenen Situationen nur ausprobiert werden, um eine Bewertung durchzuführen.

Das analytische Verfahren lässt sich aber auch auf Refactoringtools in anderen Programmiersprachen, für die Constraintregeln existieren, übertragen. Für die Sprachen C# und Eiffel sind die Regeln bereits vorhanden [ST09]; die Analyse nach dem beschriebenen Verfahren muss hier lediglich durchgeführt werden.

Bei der Ermittlung der durch Refactoringtools beeinflussten Veränderlichen in Abschnitt 4.3 wurden ausschließlich die direkten Einflüsse von Refactoringtools berücksichtigt. Die indirekten Änderungen, die durch Anpassungen der Zugriffsmodifizierer bei der Berücksichtigung der Constraintregeln entstehen, wurden aufgrund ihres Umfangs nicht untersucht. Es bleibt dadurch die Herausforderung für zukünftige Arbeiten, einen Weg zu finden diese Änderungen zu untersuchen.

### 7.3. Fazit

Die vorliegende Abhandlung hat die Relevanz der Berücksichtigung von Zugreifbarkeiten bei den Eclipse-JDT-Refactoringtools aufgezeigt. Es ist durch die Vielzahl der gefundenen Fehler die Basis für deren Korrektur gelegt worden. Die Refactoringtools können dadurch eine höhere Zuverlässigkeit erreichen, und es können Fehler bei der Durchführung von Refactorings vermieden werden.

# Literaturverzeichnis

- [Dau07] DAUM, Berthold: *Java 6*. Addison-Wesley, München, 2007
- [ECL] *Eclipse Java Development Tools (JDT), Project Website*. <http://www.eclipse.org/jdt/>. – Online-Ressource, Abruf: 10.12.2009
- [Fow05] FOWLER, Martin: *Refactoring- Studentenausgabe. Oder wie Sie das Design vorhandener Software verbessern (Programmer's Choice)*. Addison-Wesley, München, 2005
- [JLS] *The Java Language Specification*. <http://java.sun.com/docs/books/jls/>. – Online-Ressource, Abruf: 10.12.2009
- [JUn] *JUnit, Project Website*. <http://www.junit.org/>. – Online-Ressource, Abruf: 10.12.2009
- [ST09] STEIMANN, Friedrich ; THIES, Andreas: *From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility*. Hagen, Germany : FernUniversität in Hagen, 2009
- [TKB] TIP, Frank ; KIEZUN, Adam ; BÄUMER, Dirk: *Refactoring for Generalization using Type Constraints*. in: Proc. von OOPSLA (2003) 13-26

# Abbildungsverzeichnis

5.1. Beispiel für PullUp-Methode [Fow05] . . . . .	60
5.2. Fehlermeldung PullUp-Refactoringtool . . . . .	63
6.1. Verteilung der Situationen nach Einfluss und Fehlern . . . . .	88
6.2. Ergebnis nach Constraintregeln . . . . .	89
6.3. Differenzierung der Fehler – mit oder ohne Compilermeldung . .	89

# Tabellenverzeichnis

4.1. Beispiel für Abhängigkeitstabelle . . . . .	35
4.2. Beispiel für CV-Tabelle . . . . .	36
4.3. RV-Tabelle . . . . .	37
4.4. CV-Tabelle für das Refactoringtool R2 . . . . .	38
4.5. CV-Tabelle für das Refactoringtool R2 (kompakt) . . . . .	38
4.6. Ergebnis in CV-Tabelle . . . . .	39
4.7. Konkrete RV-Tabelle für Untersuchung . . . . .	44
5.1. CV-Tabelle für PullUp . . . . .	61
5.2. Untersuchungsergebnisse für PullUp (Methode) . . . . .	73
5.3. Untersuchungsergebnisse für Generalize Declared Type . . . . .	73
5.4. Untersuchungsergebnisse für PushDown (Methode) . . . . .	74
5.5. Untersuchungsergebnisse für PullUp (Feld) . . . . .	74
5.6. Untersuchungsergebnisse für PushDown (Feld) . . . . .	75
5.7. Untersuchungsergebnisse für Move-Methode . . . . .	75
5.8. Untersuchungsergebnisse für EncapsulateField . . . . .	76
5.9. Untersuchungsergebnisse für Move Class . . . . .	76
5.10. Untersuchungsergebnisse für ChangeMethodSignature . . . . .	77
5.11. Untersuchungsergebnisse für ConvertLocalVariable . . . . .	77
5.12. Untersuchungsergebnisse für ConvertMemberType . . . . .	78
5.13. Untersuchungsergebnisse für ExtractClass . . . . .	78
5.14. Untersuchungsergebnisse für ExtractConstant . . . . .	79
5.15. Untersuchungsergebnisse für ExtractInterface . . . . .	79
5.16. Untersuchungsergebnisse für ExtractMethod . . . . .	80
5.17. Untersuchungsergebnisse für ExtractSuperclass . . . . .	80
5.18. Untersuchungsergebnisse für InferGenericTypeArguments . . . . .	81
5.19. Untersuchungsergebnisse für InlineConstant . . . . .	81
5.20. Untersuchungsergebnisse für InlineMethode . . . . .	82
5.21. Untersuchungsergebnisse für IntroduceFactory . . . . .	82

## Tabellenverzeichnis

5.22. Untersuchungsergebnisse für IntroduceIndirection . . . . .	83
5.23. Untersuchungsergebnisse für IntroduceParameter . . . . .	83
5.24. Untersuchungsergebnisse für MoveStaticMembers (Methode) . .	84
5.25. Untersuchungsergebnisse für MoveStaticMembers (Feld) . . . .	84
5.26. Untersuchungsergebnisse für UseSupertypeWherePossible . . . .	85
5.27. Untersuchungsergebnisse für RenameJavaElement (Feld) . . . .	85
5.28. Untersuchungsergebnisse für RenameJavaElement (statische Me- thode) . . . . .	85
5.29. Untersuchungsergebnisse für RenameJavaElement (Methode) . .	86

# Listings

1.1. Beispiel für geändertes Programmverhalten [ST09] . . . . .	6
2.1. Berücksichtigung von anderen Programmteilen durch Refactoringtools . . . . .	10
2.2. Beispiel für die Verletzung der Zugriffsrechte . . . . .	11
2.3. Fehler durch Refactoringtool ohne Hinweis [ST09] . . . . .	11
2.4. Beispiel für Acc-1 [ST09] . . . . .	15
2.5. Beispiel für Acc-2 . . . . .	16
2.6. Beispiel für Inh-1 [ST09] . . . . .	17
2.7. Beispiel für Inh-2 [ST09] . . . . .	18
2.8. Beispiel für Sub-1 . . . . .	18
2.9. Beispiel für Sub-2 . . . . .	19
2.10. Beispiel für Dyn-1 . . . . .	20
2.11. Beispiel für Dyn-2 . . . . .	20
2.12. Beispiel für Ovr [ST09] . . . . .	21
2.13. Beispiel für Hid . . . . .	22
2.14. Beispiel für Misc-1 . . . . .	23
2.15. Beispiel für Misc-2 . . . . .	23
2.16. Beispiel für Misc-3 . . . . .	23
2.17. Beispiel für Misc-4 mit Inhalt von Datei A.java . . . . .	24
2.18. Beispiel für Misc-5 . . . . .	24
2.19. Beispiel für Misc-6 . . . . .	25
3.1. Beispiel für Veränderliche . . . . .	29
3.2. Beispiel für Parameter von Situationen . . . . .	30
5.1. Methode in Refactoringbeispiel . . . . .	60
5.2. Beispiel für Referenzen . . . . .	62
5.3. Szenario $\lambda(r)$ Acc-1 . . . . .	62
5.4. Szenario $\lambda(r)$ Inh-2 . . . . .	64

*Listings*

5.5. Szenario $\lambda(r)$ Ovr – gleiches Paket . . . . .	64
5.6. Szenario $\lambda(r)$ Ovr – unterschiedliche Pakete . . . . .	65
5.7. Szenario $\lambda(d)$ mit Acc-1 . . . . .	66
5.8. Szenario $\lambda(d)$ mit Acc-2 . . . . .	66
5.9. Szenario $\lambda(d)$ mit Sub-2 . . . . .	67
5.10. Szenario $\lambda(d)$ mit Dyn-1 . . . . .	67
5.11. Szenario $\lambda(d)$ mit Dyn-2 . . . . .	68
5.12. Szenario $\lambda(d)$ mit Hid . . . . .	69
5.13. Szenario Überschreiben mit Sub-1 . . . . .	70
5.14. Szenario Überladen mit Ovr-1 . . . . .	70
5.15. Szenario $\rho(r)$ mit Acc-2 . . . . .	71
5.16. Szenario $\rho(r)$ mit Acc-2 nach Refactoring . . . . .	72

# A. Anhang

## A.1. Beschreibung der beiliegenden CD

Auf der beigelegten CD befinden sich, basierend auf den Situationen, in denen die Zugreifbarkeit von Deklarationen zu berücksichtigen ist, die entsprechenden Szenarien. Die Szenarien sind in Form von Java-Paketen als Quellcode dokumentiert. Die Gesamtheit der untersuchten Szenarien befindet sich in einem Eclipse Workspace, in dem für jedes Refactoringtool ein Projekt mit den entsprechenden Paketen zu den Szenarien existiert. Die Paketnamen spezifizieren eindeutig die Szenarien. Es ist dadurch insbesondere möglich, ein Paket einer Situation in der CV-Tabelle eines Refactoringtools zuzuordnen. Es ergibt sich auf der CD somit die folgende beispielhafte Verzeichnisstruktur.

```
Untersuchung/  
  index.html  
  .metadata/  
  ChangeMethodSignature/  
    src/  
      cms/  
        acc1/  
          lr/  
            n1/  
              A.java  
              package.html  
            n2/  
            ...  
          ...  
        acc2/  
        ...
```

## A. Anhang

```
doc/  
  index.html  
  ...  
<Verzeichnis pro weiterem Refactoringtool>/
```

Das in der Verzeichnisstruktur abgebildete Szenario im Ordner `cms/acc1/lr/n1/` befindet sich im Java-Paket `cms.acc1.lr.n1`. Der Paketname ist wie folgt zu interpretieren:

<pre>&lt;Refactoringtool Abk.&gt; .&lt;Constraintregel Id&gt; .&lt;Abkürzung für Veränderliche&gt; .n&lt;Szenario lfd. Nummer&gt;</pre>
---

Es befinden sich in dem Paket die entsprechenden Klassen für das Szenario und die Datei `package.html`. Diese Datei stellt eine Paketbeschreibung beim Erstellen von JavaDoc-Dokumentationen zur Verfügung. Die entsprechende Dokumentation ist für die Projekte der Refactoringtools im Ordner `<Refactoringtoolname>/doc` abgelegt und dort über die Datei `index.html` aufrufbar. In der Dokumentation für jedes Paket ist eine Aussage, ob es sich um einen Fehler handelt, vorhanden. Der zentrale Einstiegspunkt für die Ergebnisse ist die Datei `index.html` im Verzeichnis „Untersuchung“. Von dieser Datei aus kann die Dokumentation der einzelnen Refactoringtools über Verweise aufgerufen werden.