

Fernuniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang
Master of Computer Science

Ein Refaktorisierungswerkzeug zur Umsetzung des Law of Demeter

Diethelm Berens
Oktober 2011

Betreuer:
Prof. Dr. Friedrich Steimann

Abstract

„Sprich nicht mit Fremden“ - so lautet die saloppe und verkürzte Darstellung einer objektorientierten Design-Empfehlung, die unter dem Namen *Law of Demeter* bekannt ist. Für die objektorientierte Programmierung besagt diese Empfehlung vereinfacht dargestellt, dass Methoden nur von unmittelbar bekannten Objekten aufgerufen werden sollen. Als Indiz für eine Verletzung des Law of Demeter gelten typischerweise verkettete Methodenaufrufe. Bei einer Verletzung des Gesetzes kann das unter dem Namen *Hide Delegate* geführte Refactoring angewandt werden. Neben der detaillierten Darstellung des *Law of Demeter* beschreibt diese Arbeit die Implementierung dieses Refactorings als Eclipse-Plug-in für die Programmiersprache Java. Ein Schwerpunkt bildet dabei die Aufstellung der notwendigen Vorbedingungen. Bei der Evaluierung des entwickelten Plug-ins wurde dieses systematisch für eine Reihe bekannter quelloffener Java-Projekte getestet. Hierbei konnten ebenfalls erste Hinweise gemessen werden, dass eine Einhaltung des *Law of Demeter* tatsächlich zu einer geringeren, durchschnittlichen Kopplung innerhalb eines objektorientierten Programms führt.

Inhaltsverzeichnis

1. Einleitung	5
1.1. Motivation	5
1.2. Aufbau der Arbeit	6
2. Problemstellung	8
2.1. Einführung	8
2.2. Refactoring Hide Delegate	9
2.3. Refaktorisierungswerkzeug für Hide Delegate	10
3. Law of Demeter	12
3.1. Einführendes Beispiel	12
3.2. Definition des Law of Demeter	16
3.2.1. Objektform	16
3.2.2. Klassenform	17
3.3. Vorteile und Berührungspunkte mit anderen OO-Prinzipien	18
3.4. Grenzen der Anwendbarkeit und Nachteile	19
4. Anforderungen an das Refaktorisierungswerkzeug	21
4.1. Allgemeine Anforderungen	21
4.2. Vorbedingungen	21
4.3. Bedingungen für das Einfügen einer Methode	22
4.3.1. Methodensignaturen, die <i>override-equivalent</i> zueinander sind	23
4.3.2. <code>final</code> -deklarierte Methode in Supertyp	24
4.3.3. Reduzierung der Sichtbarkeit von Methoden in Subtypen	24
4.3.4. Probleme durch Überschreiben	25
4.3.5. Name clash	28
4.3.6. Probleme durch Überladen	29
4.3.7. Geschachtelte Klassen	33
4.3.8. Zusammenfassung der Bedingungen	35
4.3.9. Fazit aus den Vorbedingungen	36
4.4. Bedingungen für das Löschen der selektierten Methode	37
5. Implementierung	39
5.1. Einführung Refactorings in Eclipse	39

5.1.1. Refactoring-Framework	39
5.1.2. Kernkomponenten der Java Development Tools (JDT)	42
5.2. Aufbau	45
5.3. Funktionsweise wesentlicher Klassen	46
6. Evaluation	52
6.1. Beschreibung der untersuchten Kriterien	52
6.2. Beschreibung des Refactoring Tool Testers	53
6.3. Prüfung mit dem Refactoring Tool Tester	55
6.4. Ergebnisse	56
7. Diskussion	61
7.1. Interpretation der Ergebnisse	61
7.2. Verwandte Arbeiten	62
8. Schlussbetrachtungen	65
8.1. Zusammenfassung und Fazit	65
8.2. Ausblick	65
A. Installation und Bedienungshinweise	67
B. Beschreibung der beiliegenden CD	69
Abbildungsverzeichnis	70
Listings	71
Literatur	72

1. Einleitung

1.1. Motivation

„Sprich nicht mit Fremden“¹ - so lautet die saloppe und verkürzte Darstellung einer objektorientierten Design-Empfehlung, die unter dem Namen *Law of Demeter* [LHR88] (bzw. Gesetz von Demeter) bekannt ist. Für die objektorientierte Programmierung besagt diese Empfehlung vereinfacht dargestellt, dass Methoden nur von unmittelbar bekannten Objekten aufgerufen werden sollen. Zu den unmittelbar bekannten Objekten zählen beispielsweise Objekte, die einer Methode als Parameter übergeben werden oder aber Objekte, zu denen direkte Beziehungen über die Instanzvariablen eines Objekts bestehen.

Die Berücksichtigung dieses Gesetzes soll insbesondere die Kopplung innerhalb eines objektorientierten Systems gering halten und dadurch positiven Einfluss auf die Wartbarkeit eines objektorientierten Programms nehmen. Als Indiz für eine Verletzung des Law of Demeter gelten typischerweise verkettete Methodenaufrufe, bei denen ein Methodenaufruf unmittelbar auf einem anderen folgt. Dabei werden dann Methoden von Objekten aufgerufen, die eben nicht mehr unmittelbar, sondern nur noch mittelbar, bekannt sind.

```
1 public class Client {
2     private Server server;
3     ...
4     public void m(){
5         // Verletzung des Law of Demeter
6         server.getDelegate().doSomething();
7         ...
8     }
9 }
```

Listing 1: Beispiel zum Law of Demeter

Listing 1 zeigt ein Beispiel für die Programmiersprache Java: Ein Objekt *Client* kennt über eine Instanzvariable unmittelbar ein Objekt *Server* und darf nach dem Gesetz Methoden wie hier `getDelegate()` auf diesem Objekt aufrufen. Wird auf dem Objekt, welches `getDelegate()` zurückliefert (im folgenden *Delegate* genannt), eine weitere Methode wie hier `doSomething()` aufgerufen, so liegt eine Verletzung des Law of Demeter vor, da das Objekt *Delegate* dem Objekt *Client* nicht unmittelbar bekannt ist. Das Gesetz wird

¹ Diese Phrase wurde erstmals von [Lar04] eingeführt.

dabei ebenfalls verletzt, wenn *Delegate* zunächst einer temporären Variablen zugewiesen wird und über dieser ein weiterer Methodenaufruf ausgeführt wird.

Für die Behebung einer Verletzung des Gesetzes existiert ein Refactoring mit dem Namen *Hide Delegate*. Dieses sorgt durch Einfügen einer neuen Methode und Änderung des verketteten Methodenaufrufs dafür, dass eine dem Gesetz konforme Lösung erstellt wird. In dem Beispiel würde konkret einer Klasse *Server* eine neue Methode `doSomething()` hinzugefügt, die Aufrufe an das Objekt *Delegate* weiterleitet. Zeile 6 des Listings würde dann geändert in `server.doSomething();`.

Das Ziel dieser Arbeit besteht darin, ein Refaktorisierungswerkzeug für Hide Delegate für die Programmiersprache Java zu erstellen, mit dem das Law of Demeter umgesetzt werden kann. Ein Schwerpunkt wird dabei die Aufstellung aller für das Refactoring notwendigen Vorbedingungen sein, insbesondere die Frage, welche Bedingungen zu berücksichtigen sind, um eine gegebene Methode in einer existierenden Klasse einzufügen.

1.2. Aufbau der Arbeit

Dieses Kapitel bringt eine Motivation für die vorliegende Arbeit und beschreibt deren Aufbau. In Kapitel 2 folgt die detaillierte Beschreibung des zu lösenden Problems inklusive notwendiger Begriffsdefinitionen sowie eine Darstellung des Refactorings Hide Delegate. Das Law of Demeter, auf dessen Theorie sich Hide Delegate stützt, wird eingehend in Kapitel 3 vorgestellt. Neben einer Definition des Gesetzes erfolgt auch eine Einordnung in den Zusammenhang anderer objektorientierter Designprinzipien sowie eine Gegenüberstellung der positiven und negativen Aspekte des Gesetzes.

Kapitel 4 geht auf die Anforderungen an das zu erstellende Refaktorisierungswerkzeug ein und beschreibt detailliert die Bedingungen, die für eine Anwendung des Refactorings erfüllt sein müssen. Insbesondere wird dargestellt, welche Prüfungen notwendig sind, bevor eine gegebene Methode in eine existierende Klasse hinzugefügt werden kann. Die notwendigen Prüfungen werden hierbei aus der Java-Sprachspezifikation abgeleitet. Der Realisierung des Werkzeugs, das als Plug-in für die Entwicklungsumgebung Eclipse umgesetzt wird, widmet sich Kapitel 5. Neben einer Beschreibung der verwendeten Frameworks und Tools von Eclipse werden die wesentlichen Komponenten des Plug-ins vorgestellt.

In Kapitel 6 schließt sich die Darstellung der durchgeführten Evaluierung des Plug-ins und die Aufbereitung und Beschreibung der hierbei erzielten Ergebnisse an. Diese Ergebnisse

werden im darauf folgenden Kapitel 7 interpretiert und bewertet und in den Zusammenhang verandter Arbeiten eingeordnet. Kapitel 8 schließt mit einer Zusammenfassung und einem Fazit dieser Arbeit sowie einem Ausblick.

2. Problemstellung

In diesem Kapitel wird die Problemstellung dieser Arbeit vorgestellt. Nach einer Definition und Einordnung des Begriffs *Refaktorisierung* erfolgt in Abschnitt 2.2 die Beschreibung einer Refaktorisierung, die unter dem Namen *Hide Delegate* bekannt ist. Abschließend wird dargelegt, welche Aufgaben das zu erstellende Refaktorisierungswerkzeug zu erledigen hat.

2.1. Einführung

Bevor die konkrete Aufgabenstellung und das zu lösende Problem näher vorgestellt werden, soll zunächst ein zentraler Begriff dieser Arbeit definiert und eingeordnet werden:

Refaktorisierung: “Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern.” [FBB⁺99]

Das Ziel einer Refaktorisierung liegt also in der Verbesserung des Designs, um Lesbarkeit, Wartbarkeit und Änderbarkeit der Software zu verbessern. Im Rahmen einiger agiler Vorgehensmodelle wie dem des Extreme Programming ist die Bedeutung von Refaktorisierungen (oder engl. *Refactorings*) in den letzten Jahren noch weiter gestiegen, da sie zum festen Bestandteil des Vorgehensmodells gehören. Aber auch bei „klassischen“ Vorgehensmodellen besteht der Bedarf, das Design, welches ja vor einer Entwicklung erstellt wird, während der Entwicklungsphase zu überarbeiten: bspw. dadurch bedingt, dass sich die Anforderungen während der Entwicklungsphase ändern². Refaktorisierungen können ebenfalls durch (regelmäßig) durchgeführte Code Reviews ausgelöst werden.

Als Standardwerk zum Thema Refactorings kann das gleichnamige Buch von Martin Fowler aus dem Jahre 1999 angesehen werden [FBB⁺99]. Es enthält einen Katalog von Beschreibungen für Refaktorisierungen inkl. eines jeweiligen Namens für das Refactoring. Diese Beschreibungen enthalten jeweils ein bestimmtes Design als Ausgangsbasis, welches es zu ändern gilt, sowie die vom Entwickler durchzuführenden Schritte, um das Ziel eines verbesserten Designs zu erreichen.

Viele integrierte Entwicklungsumgebungen wie IntelliJ IDEA oder Eclipse unterstützen bereits eine Menge von Refactorings, die automatisiert durchgeführt werden können. Das

² Dieser Umstand wird als *Requirements drift* bezeichnet.

Ziel dieser Abschlussarbeit besteht darin, ein Refaktorisierungswerkzeug für das Refactoring *Hide Delegate*³ zu erstellen, welches im folgenden Abschnitt beschrieben wird.

2.2. Refactoring Hide Delegate

Das Refactoring *Hide Delegate* ist ebenfalls in dem o.g. Katalog von Refaktorisierungen enthalten. Es wird dort in einer Gruppe von Refactorings geführt, die sich inhaltlich mit dem Verschieben von Eigenschaften zwischen Objekten befassen. Weitere Refactorings aus dieser Kategorie sind bspw. *Move Method*, *Move Field* und *Extract Class* [FBB⁺99].

Die Ausgangslage zur Anwendung des Refactorings liegt vor, wenn ein Client eine Methode eines Server-Objekts aufruft, die ein Delegate-Objekt zurückliefert, auf dem dann unmittelbar eine weitere Methode aufgerufen wird. Das zu verbessernde Design der beteiligten Klassen veranschaulicht Abbildung 1.

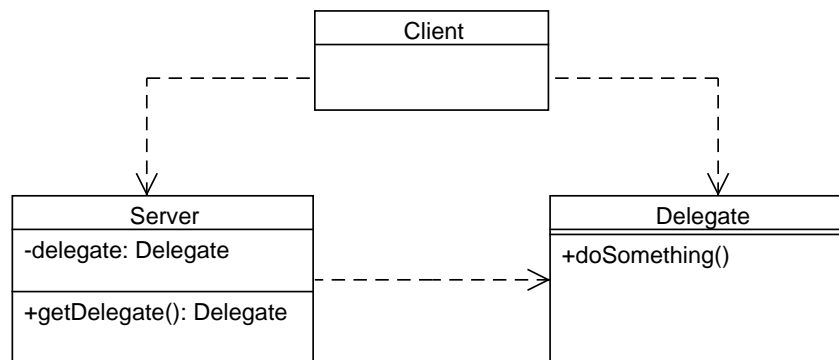


Abbildung 1: Ausgangssituation des Refactorings *Hide Delegate*

Auf diese Weise sind der Client und das Delegate-Objekt, welches von der ersten Methode zurückgeliefert wird, aneinander gekoppelt. Änderungen an der Klasse des Delegate-Objekts können sich unmittelbar auf die Client-Klassen auswirken. Ein entsprechender verketteter Methodenaufruf würde also wie folgt aussehen:

```
server.getDelegate().doSomething()
```

³ In der deutschen Übersetzung von [FBB⁺99] wird diese Refaktorisierung unter dem Namen *Delegation verbergen* geführt. Da allgemein die englischen Bezeichnungen jedoch weitaus gebräuchlicher sind, wird im Rahmen dieser Arbeit ebenfalls die englische Bezeichnung verwendet.

Zur Verbesserung des Designs empfiehlt das Refactoring *Hide Delegate* daher, dem Server-Objekt eine Methode hinzuzufügen, die die Aufrufe an das Delegate-Objekt „delegiert“⁴ und das Delegate-Objekt hinter dem Klasseninterface des Server-Objekts zu verbergen (geschieht hier durch Entfernen der Methode `getDelegate()`). Abbildung 2 zeigt die refaktorierte Form.

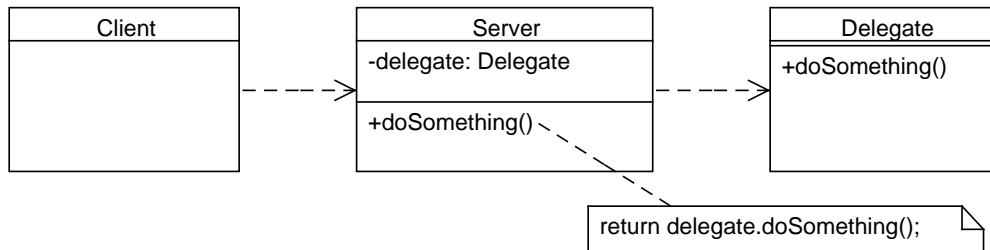


Abbildung 2: Zieldesign des Refactorings *Hide Delegate*

Das Entwurfsprinzip, auf das *Hide Delegate* zurückzuführen ist, ist jedoch schon deutlich länger unter dem Namen *Law of Demeter* bekannt. Es wird ausführlich in Kapitel 3 vorgestellt.

2.3. Refaktorisierungswerkzeug für Hide Delegate

Das Ziel dieser Arbeit besteht darin, für das in Abschnitt 2.2 vorgestellte Refactoring *Hide Delegate* ein Refaktorisierungswerkzeug zu erstellen. Der Ausgangspunkt für eine Benutzung des Werkzeugs soll eine vom Benutzer selektierte Methode sein, die ein zu verbergendes Delegate-Objekt bislang (als Rückgabeparameter der Methode) preisgibt. In dem Beispiel des vorhergehenden Abschnitts ist dies die Methode `getDelegate()` der Klasse `Server`. Das zu erstellende Werkzeug hat alle Vorbedingungen, die an die Durchführbarkeit des Refactorings geknüpft sind und in Kapitel 4 zusammengestellt werden, zu prüfen.

Es soll zunächst alle Methodenreferenzen der selektierten Methode suchen. Für jede Methodenreferenz ist zu prüfen, ob an der Aufrufstelle auf dem von der selektierten Methode zurückgelieferten Delegate-Objekt unmittelbar eine weitere Methode aufgerufen wird, d.h. ob ein verketteter Methodenaufruf der Form `server.getDelegate().doSomething()`

⁴ Genaugenommen handelt es sich hierbei nicht um eine Form der Delegation, sondern um eine Form des Forwarding (vgl. [Ste07]).

vorliegt. Ist dies der Fall, so ist eine Vermittlermethode (hier für die Klasse **Server**) zu erzeugen, die die Aufrufe an das Delegate-Objekt weiterleitet. Gleichzeitig ist die Aufrufstelle so abzuändern, dass nun die neu eingeführte Vermittlermethode aufgerufen wird (hier `server.doSomething()`).

Abschließend soll das Refaktorisierungswerkzeug prüfen, ob es möglich ist, die selektierte Methode zu löschen, da diese ja das zu verbergende Delegate-Objekt bislang zugänglich macht. Der Benutzer soll jedoch die Option besitzen, das Löschen zu unterbinden.

3. Law of Demeter

Das Gesetz von Demeter ist ein Entwurfsprinzip für objektorientierte Systeme und besagt, dass eine Methode eines Objekts nur Methoden von „unmittelbar bekannten“ Objekten aufrufen soll. Dieses Entwurfsprinzip wurde im Rahmen eines Forschungsprojekts an der Northeastern Universität in Boston unter der Leitung von Karl J. Lieberherr im Jahre 1987 aufgestellt. Das Forschungsprojekt trug den Namen „Demeter“ - daher stammt auch der Name des Gesetzes. Das Entwurfsprinzip ist programmiersprachenunabhängig und trägt mit dazu bei, eine lose Kopplung zwischen den Objekten eines Systems zu erreichen. Es ist auch unter dem Begriff *Principle of least knowledge* bekannt.

Der folgende Abschnitt gibt zunächst ein einführendes Beispiel, bei dem das Gesetz von Demeter verletzt wird und erläutert neben den negativen Aspekten des Beispiels, mit welchen Mitteln eine dem Gesetz konforme Umsetzung zu erreichen ist. Das Gesetz existiert in 2 Formen, einer Objektform und einer Klassenform - diese werden im darauf folgenden Abschnitt formal beschrieben.

Abschnitt 3.3 geht anschließend auf die Vorteile des Gesetzes von Demeter ein und zeigt auf, welche Berührungspunkte zu anderen bekannten OO-Design-Prinzipien bestehen. Allerdings beinhaltet das Gesetz nicht nur Vorteile - die Nachteile und die Grenzen der Anwendbarkeit des Gesetzes beleuchtet Abschnitt 3.4.

3.1. Einführendes Beispiel

Zunächst soll an einem Beispiel demonstriert werden, wie ein Quellcode aussieht, der das Gesetz von Demeter verletzt. An dem Beispiel wird erläutert, was es aus der Sichtweise des Gesetzes in dem vorliegenden Fall zu kritisieren gibt und welche Änderungen notwendig sind, damit das Gesetz von Demeter eingehalten wird. Das nachfolgende Beispiel basiert im wesentlichen auf dem Artikel „The Paperboy, the Wallet and the Law of Demeter“ von David Bock [Boc]. Gegeben sei ein Zeitungsjunge, der von seinem Kunden eine Bezahlung einfordert. Zur Abbildung dieses Szenarios liegen (vereinfacht) die Klassen `Customer` und `Wallet` vor:

```
1 public class Customer {  
2  
3     private Wallet myWallet;  
4 }
```

```
5 public Wallet getWallet() {
6     return myWallet;
7 }
8
9 public void setWallet(Wallet wallet){
10     this.myWallet = wallet;
11 }
12 }
13
14 public class Wallet {
15
16     private float value;
17
18     public Wallet(float value){
19         this.value = value;
20     }
21
22     public float getTotalMoney() {
23         return value;
24     }
25
26     public void setTotalMoney(float newValue) {
27         value = newValue;
28     }
29
30     public float subtractMoney(float debit) {
31         value -= debit;
32         return debit;
33     }
34 }
```

Listing 2: Klassen Customer und Wallet

Das Einfordern der Bezahlung könnte über folgenden Code-Ausschnitt innerhalb der Klasse Paperboy umgesetzt werden:

```
1 public class Paperboy {
2
3     public void collectDebt(Customer customer){
4         ...
5         float payment = 2.00f; // I want my two dollars!
6         float paidAmount = customer.getWallet().subtractMoney(payment);
7         ...
8     }
9 }
```

Listing 3: Client-Klasse Paperboy

Diese Art der Umsetzung bringt allerdings einige negative Aspekte mit sich: Auf diese Art sind die Klassen Customer, Wallet und Paperboy eng aneinander gekoppelt (vgl.

Abbildung 3). Änderungen an der Klasse `Wallet` können sich nicht nur auf die Klasse `Customer` auswirken, sondern auch auf die Klasse `Paperboy`. Das OO-Prinzip der losen Kopplung wird hier nicht befolgt. Ebenfalls wird dem Prinzip des *Information hiding* nicht vollständig Rechnung getragen, da die Klasse `Paperboy` Kenntnis über die interne Struktur der Klasse `Customer` hat - sie weiß, dass der Kunde eine Geldbörse besitzt. Doch diese Kenntnis benötigt die Klasse `Paperboy` nicht, sie muss lediglich eine Bezahlung beim Kunden einfordern. Ob der Kunde das erforderliche Geld aus seiner Geldbörse oder aber aus einem Sparschwein entnimmt, ist für den Zeitungsjungen irrelevant und sollte zum Implementationsgeheimnis der Klasse `Customer` gehören. Die obige Modellierung einer Bezahlung entspricht ebenfalls nicht einer Abbildung der „realen Welt“: Ein Zeitungsjunge fragt den Kunden nicht nach seiner Geldbörse und der Kunde würde diese auch nicht aus der Hand geben - er fordert lediglich einen Geldbetrag ein.

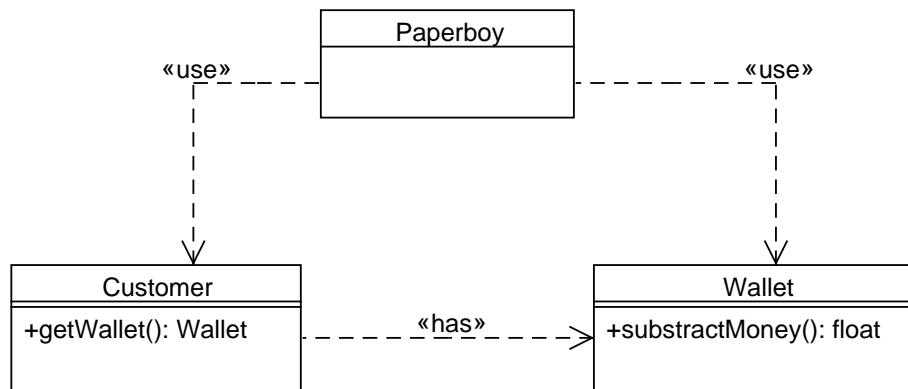


Abbildung 3: Klassendiagramm der Klassen `Paperboy`, `Customer` und `Wallet`

Eine aus OO-Sicht bessere Umsetzung, die die o.g. negativen Aspekte nicht beinhaltet, könnte wie folgt aussehen:

```

1 public class Customer {
2
3     private Wallet myWallet;
4
5     public float makePayment(float bill){
6         if (myWallet != null){
7             if (myWallet.getTotalMoney() > bill){
8                 return myWallet.subtractMoney(bill);
9             }
10        }
11    }
12 }
  
```

```

11     return 0;
12 }
13 }
14
15 public class Paperboy {
16     ...
17     Customer customer = new Customer();
18     ...
19     float payment = 2.00f; // I want my two dollars!
20     float paidAmount = customer.makePayment(payment);
21     ...
22 }

```

Listing 4: Modifizierte Klassen Customer und Paperboy

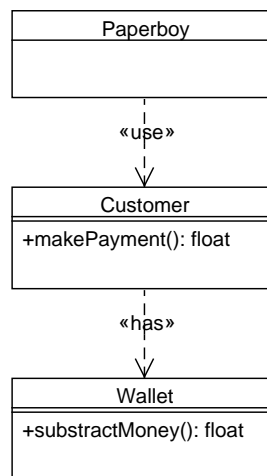


Abbildung 4: Klassendiagramm nach Einführung einer Vermittlermethode

Die wesentliche Änderung besteht dabei in der neu eingeführten Vermittlermethode⁵ `makePayment()` in der Klasse `Customer`, die - neben einer Reihe von möglichen Prüfungen - entsprechende Aufrufe an die Instanzvariable vom Typ `Wallet` weiterleitet. Die öffentliche Zugriffsmethode `getWallet()` wurde hierbei entfernt, so dass die Instanzvariable vom Typ `Wallet` zum Implementationsgeheimnis der Klasse `Customer` geworden ist. An der Aufrufstelle innerhalb der Klasse `Paperboy` wird jetzt entsprechend die neu eingeführte Vermittlermethode aufgerufen.

Diese Umsetzung bietet eine bessere Wartbarkeit, da bspw. die Klasse `Wallet` geändert werden kann und sich diese Änderung nur noch auf die Klasse `Customer` auswirkt, nicht

⁵ In der Literatur werden hierfür häufig auch die Begriffe *Delegatemethode* oder *Wrappermethode* verwendet.

mehr aber auf die Klasse `Paperboy` bzw. alle Klassen, die die Methode `getWallet()` der Klasse `Customer` aufgerufen haben. Die neue Methode `makePayment()` bietet auch mehr Flexibilität: ändert sich die Art, wie der Kunde das erforderliche Geld besorgt, kann dies innerhalb der Klasse `Customer` und der Methode `makePayment()` modelliert werden, ohne das davon die Klasse `Paperboy` betroffen wäre. Ebenfalls kann (wie im modifizierten Code geschehen) sichergestellt werden, dass es zu keiner *NullPointerException* kommt. Diese Überprüfung könnte natürlich auch schon bereits in Listing 3 eingefügt werden, allerdings würde der Code dann dort etwas unübersichtlicher und sich mit weiteren Internas der Klasse `Customer` beschäftigen.

3.2. Definition des Law of Demeter

Das Gesetz von Demeter gibt es in 2 verschiedenen Grundformen: der Objektform und der Klassenform. Die nachfolgenden Definitionen gehen auf [LHR88] zurück (eine kürzere Version der Arbeit erschien in [LH89a]).

3.2.1. Objektform

Eine Methode `M` des Objekts `O` darf an ein Objekt `P` nur dann eine Nachricht schicken, wenn einer der folgenden Fälle gilt:

- `P` und `O` sind identische Objekte
- `P` wurde als Parameter der Methode `M` übergeben
- `P` wurde innerhalb der Methode `M` erzeugt
- `P` ist ein Attribut von `O`
- `P` ist ein globales Objekt

Ob ein gegebener Programmcode die Objektform des Law of Demeter einhält, kann statisch nicht exakt geprüft werden. Als eine Annäherung an die Objektform kann die Klassenform angesehen werden, die zur Kompilierzeit statisch geprüft werden kann.

3.2.2. Klassenform

Bei der Klassenform werden wiederum 2 Varianten unterschieden: eine *strikte Form* und eine *minimierte Form* [LHR88]. Für die *strikte Form* gilt folgende Definition:

Eine Methode M der Klasse A darf an eine Klasse B nur dann Nachrichten schicken, wenn einer der folgenden Fälle gilt:

- A ist Subtyp von B
- B ist der deklarierte Typ eines Parameters der Methode M
- Instanzen von B werden innerhalb der Methode M erzeugt
- B ist der deklarierte Typ einer Instanzvariablen von A
- B ist der deklarierte Typ eines globalen Objekts

Die *minimierte Form* „erlaubt“ dagegen den Zugriff auf Methoden einiger weiterer Klassen. Hierfür nennen die Autoren des Law of Demeter folgende Ausnahmen:

- Gilt eine Klasse bzw. deren Klasseninterface als stabil, so darf auf die Methoden dieser Klasse von überall her zugegriffen werden.
- Ist es aus Gründen des Laufzeitverhaltens erforderlich, dass auf Methoden von Klassen direkt zugegriffen werden muss, obwohl diese Klasse nicht unter die obige Definition fällt, so kann auch hier eine Ausnahme gemacht werden.
- Methoden zur Objekterzeugung

In dieser minimierten Form wird deutlich, dass das Gesetz nicht wirklich als starr zu befolgendes Gesetz verstanden werden soll, sondern vielmehr als Richtlinie. Ein Entwickler sollte sich bei Verletzungen der strikten Form des Law of Demeter darüber im Klaren sein, diese dokumentieren und die Anzahl der „Verletzungen“ minimieren.

Zurück zum einführenden Beispiel aus Abschnitt 3.1. Im Listing 3 wird das Gesetz Demeters in Zeile 6 verletzt: der Aufruf von `subtractMoney()` verstößt gegen das Gesetz, da `Wallet` weder ein Feld der Klasse `Paperboy` ist, noch als Parameter der Methode `collectDebt()` übergeben bzw. in der Methode instanziiert wurde. Weiterhin ist die Klasse `Paperboy` kein Subtyp der Klasse `Wallet`.

3.3. Vorteile und Berührungspunkte mit anderen OO-Prinzipien

Das Gesetz von Demeter adressiert eine Reihe allgemein anerkannter Design-Prinzipien objektorientierter Systeme, die im folgenden kurz dargestellt werden [LH89b].

Das Prinzip der geringen Kopplung besagt, dass zwischen den Klassen eines OO-Systems eine möglichst geringe Kopplung bestehen soll [Rie96]. Je weniger Beziehungen eine Klasse zu anderen Klassen eingeht, desto geringer ist der Wartungsaufwand, wenn sich die Schnittstelle dieser Klasse zukünftig ändert. Die Einhaltung des Gesetzes von Demeter führt zu einer geringeren Kopplung. Im Beispiel war die Klasse `Paperboy` sowohl an die Klasse `Customer` als auch an die Klasse `Wallet` gekoppelt. Nach Einführung der Vermittlermethode in der Klasse `Customer` ist die Klasse `Paperboy` nur noch an `Customer` gekoppelt, nicht mehr an `Wallet`.

Das Geheimnisprinzip (Information hiding) besagt, dass eine Klasse ihre interne Struktur vor ihren Benutzerinnen verbergen sollte. Dadurch kann die interne Struktur zukünftig geändert werden, ohne dass davon die Klassen der Benutzerinnen betroffen sind. Bezogen auf das Beispiel bedeutet dies, dass die Klasse `Customer` die Tatsache, dass sie ein `Wallet` besitzt, verbergen sollte. Diese Kenntnis ist für die Benutzerinnen der Klasse nicht relevant und sollte daher zum Implementationsgeheimnis der Klasse `Customer` gehören.

Über diese allgemein bekannten Prinzipien hinaus tangiert das Gesetz von Demeter noch weitere Prinzipien. Das von Lieberherr et al. bezeichnete *Information restriction* geht auf eine Arbeit von Parnas et al. [PCW85] zurück, die aufzeigt, dass es nicht immer möglich ist, bestimmte Informationen in einem Modul zu kapseln und daher empfiehlt, die Benutzung solcher Module zu minimieren bzw. einzuschränken. Das Gesetz von Demeter schränkt ebenfalls, wie in der Definition zu sehen, die Nutzung von Methoden ein.

Das Lokalitätsprinzip von Programmen besagt, dass diejenigen Dinge, die zusammen gehören, auch im Programmtext beieinander stehen sollen (vgl. [Ste08]). Das Prinzip dient dem besseren Programmverständnis - schaut man sich eine Programmstelle an, so kann man aus dem unmittelbaren Kontext dieser Programmstelle ein Verständnis dafür entwickeln, um was es dort geht, woher die Variablen ihre Werte beziehen usw. Zu den Eigenheiten der objektorientierten Programmierung gehört es, dass durch die vielen Methodenaufrufe das Lokalitätsprinzip verletzt wird und u.a. zum Problem einer schlechten Tracebarkeit von objektorientierten Programmen führt ([Ste08]). Das Gesetz von Demeter trägt ein wenig dazu bei, dass man ein besseres Verständnis beim Studieren eines Program-

mabschnitts (beispielsweise einer Methode) entwickeln kann, da es die Zahl derjenigen Klassen, die zum Programmverständnis einer Methode notwendig sind, einschränkt.

Das Vorgehen, dass ein Objekt als Attribut ein anderes Objekt aufnimmt, dieses vor dem Zugriff von außen kapselt verbunden mit der Absicht, gewisse Aufgaben an dieses weiterzuleiten, findet sich auch in einer Reihe von Design Patterns wieder (vgl. [Boc]). So zum Beispiel beim *Objektadapter* oder beim *Proxy* [GHJV95].

3.4. Grenzen der Anwendbarkeit und Nachteile

Wie bereits im Zusammenhang mit der minimierten Form des Gesetzes angedeutet, ist das Gesetz nicht als unumstößliches Gesetz zu verstehen, sondern vielmehr als Richtlinie und Empfehlung. Es gibt eine Reihe von Ausnahmen, in denen es nicht sinnvoll bzw. möglich ist, das Gesetz zu befolgen. So stößt das Gesetz beispielsweise bei der Verwendung von Collections an seine Grenzen: besteht zwischen 2 Objekten A und B eine 1:n-Assoziation, so wird diese Beziehung bei einer objektorientierten Sprache wie Java durch ein „Zwischenobjekt“ umgesetzt (das Zwischenobjekt kann in Java bspw. vom Typ `java.util.Vector` sein). Dieses Zwischenobjekt nimmt eine Menge von Objekten vom Typ B auf. Das heißt, die eigentlich vorhandene 1:n-Beziehung zwischen A und B bildet sich ab in eine 1:1-Beziehung zwischen Objekt A und dem Zwischenobjekt sowie einer 1:n-Beziehung zwischen dem Zwischenobjekt und dem Objekt B. Übertragen auf das Gesetz von Demeter hieße dies: enthält eine Klasse A eine Instanzvariable vom Typ einer Collection-Klasse (bspw. `java.util.Vector`), so dürfte man innerhalb der Methoden von Klasse A zwar auf die Methoden der Collection-Klasse zugreifen, nicht jedoch auf die Methoden der Elemente, die diese Collection-Klasse enthält.

Hieran wird deutlich, dass das Gesetz nicht immer und überall gelten kann und es Ausnahmen gibt. Eine ähnlich gelagerte Ausnahme liegt dann vor, wenn eine Klasse als Container konzipiert ist, wie beispielsweise bei der Klasse `java.awt.Container`.

Das Gesetz von Demeter ist nicht unumstritten. So kann eine Befolgung des Gesetzes dazu führen, dass eine Klasse sehr sehr viele reine Vermittlermethoden enthält und damit das Klasseninterface sehr mächtig und unübersichtlich wird. Das Prinzip einer starken Kohäsion der Klasse wird dadurch abgeschwächt. Besteht der Hauptzweck der Klasse

lediglich im Weiterleiten von Methodenaufrufen, so liegt ebenfalls ein *Code smell* vor, zu dem das Refactoring *Remove Middle Man* [FBB⁺99] existiert⁶.

Ein weiterer Einwand gegen das Gesetz kommt von [Kno01]: Wenn eine Methode, die eine Delegate-Klasse zurückliefert (auf die der Aufrufer dann weitere Methoden aufrufen kann), nicht eine konkrete Klasse zurückliefern würde, sondern lediglich ein Interface, dann besteht zumindest keine Abhängigkeit mehr zwischen der Client-Klasse und einer konkreten (Delegate-)Klasse, sondern nur noch zu einer Abstraktion derselben. Dieses Prinzip ist als *Dependency inversion principle* [Mar03] bekannt⁷.

Ein weiterer Nachteil ist das etwas schlechtere Laufzeitverhalten durch die Einführung einer Vermittlermethode, dem die Autoren des Gesetzes in ihrer minimierten Form bereits Rechnung getragen haben.

Schaut man sich die Vor- und Nachteile zusammenfassend an, so bleibt es also ein Abwägungsprozeß und hängt vom Kontext des Einzelfalls ab, ob die Vor- oder die Nachteile bei Anwendung des Gesetzes überwiegen. Das Gesetz bringt einige bekannte und bewährte OO-Designprinzipien (insbesondere das Geheimnisprinzip und das Prinzip der geringen Kopplung) auf eine kurze und knappe Formel: „Sprich nicht mit Fremden“⁸.

⁶ Daher ist es aus Sicht von Martin Fowler angebrachter, statt vom *Gesetz von Demeter* eher von der *Empfehlung von Demeter* zu sprechen: “I’ve always felt I’d be more comfortable with the Law of Demeter if it were called the Suggestion of Demeter.” [Fow07]

⁷ “Depend upon abstractions. Do not depend upon concretions.” [Mar03].

⁸ Diese Formel wurde erstmals von [Lar04] eingeführt.

4. Anforderungen an das Refaktorisierungswerkzeug

Dieses Kapitel beschreibt die an das Refaktorisierungswerkzeug zu stellenden Anforderungen. Hierbei werden im ersten Abschnitt allgemeine Anforderungen aus der Sicht des Benutzers aufgeführt, die größtenteils als Standardanforderungen bei Refactorings angesehen werden können und daher hier nur kurz skizziert werden. Abschnitt 4.2 geht dann näher auf diejenigen Vorbedingungen des Refactorings *Hide Delegate* ein, die bereits ohne größere Analyse eines abstrakten Syntaxbaumes geprüft werden können. Im darauf folgenden Abschnitt 4.3 werden detailliert diejenigen Vorbedingungen vorgestellt, die für das Einfügen einer Methode in eine Klasse erfüllt sein müssen. Abschließend stellt Abschnitt 4.4 die Bedingungen vor, die geprüft werden müssen, um festzustellen, ob die selektierte Methode gelöscht werden kann.

4.1. Allgemeine Anforderungen

In diesem Abschnitt werden allgemeine Anforderungen an das Refaktorisierungswerkzeug aus Sicht des Benutzers kurz vorgestellt. Das Refaktorisierungswerkzeug soll nach Selektion einer Methode aus dem Kontextmenü heraus aufgerufen werden können. Der Benutzer soll dabei die Gelegenheit haben, sich die vom Refactoring durchzuführenden Änderungen in einer Vorschau anzeigen zu lassen, in der Original-Code und veränderter Code gegenübergestellt werden. Bei Ansicht der Vorschau soll es möglich sein, den Refactoring-Vorgang abubrechen. Wird ein Refactoring vollständig durchgeführt, so soll die Möglichkeit bestehen, die durchgeführten Code-Änderungen über eine (allgemeine) Undo-Funktion rückgängig zu machen. Dem Benutzer soll ebenfalls über einen entsprechenden Dialog die Möglichkeit gegeben werden zu entscheiden, ob die von ihm selektierte Methode gelöscht werden soll, sofern die Methode alle hierfür erforderlichen Bedingungen erfüllt (vgl. Abschnitt 4.4). Ferner kann er angeben, ob im Rahmen des Refactorings Interfaces erweitert werden sollen, falls diese involviert sind (vgl. hierzu auch Abschnitt 5.3).

4.2. Vorbedingungen

Das Refaktorisierungswerkzeug hat zu Beginn eine Reihe von Fällen zu prüfen, die eine grundsätzliche Verwendung ausschließen und die bereits ohne tiefere Analyse eines

abstrakten Syntaxbaums durchführbar sind. Dazu zählt die naheliegende Bedingung, dass die zu refaktorisierende Klasse als zu ändernde Quellcodedatei vorliegt und analysierbar sein muss. Dies ist nicht der Fall, falls die zu untersuchende Methode per Reflection aufgerufen wird. Ebenfalls wird ein Refactoring einer Klasse ausgeschlossen, die Compile-Fehler enthält. Da alle Aufrufstellen der selektierten Methode ermittelt werden, wird verlangt, dass das zugrunde liegende Projekt ebenfalls keine Compile-Fehler enthält⁹.

Darüber hinaus muss die selektierte Methode folgende Vorbedingungen erfüllen:

1. Die Sichtbarkeit der Methode darf nicht `private` sein. Nach der Definition der Klassenform des Gesetzes von Demeter (vgl. Abschnitt 3.2.2) kann die Verwendung einer Methode mit der Sichtbarkeit `private` nicht zur Verletzung des Gesetzes führen. Daher wird dies von dem Refactoring *Hide Delegate* ausgeschlossen.
2. Der Rückgabeparameter der Methode darf kein primitiver Typ oder mit dem Schlüsselwort `void` gekennzeichnet sein.
3. Bei der selektierten Methode darf es sich nicht um einen Konstruktor handeln.
4. Der Rückgabetyt der Methode darf nicht identisch sein mit dem Typ, der die Methode deklariert.

4.3. Bedingungen für das Einfügen einer Methode

Wie im Abschnitt 2.2 dargestellt, werden bei einer Anwendung des Refactorings *Hide Delegate* der Server-Klasse Vermittlermethoden hinzugefügt, die die Aufrufe an das Delegate-Objekt weiterleiten. Das zu erstellende Refaktorisierungswerkzeug hat zunächst alle Aufrufstellen der selektierten Methode zu ermitteln. Für jede Aufrufstelle gilt es festzustellen, ob unmittelbar auf dem Delegate-Objekt eine weitere Methode *m* aufgerufen wird (d.h. an der Aufrufstelle liegt ein verketteter Methodenaufruf vor). Für diese Methode *m* muss dann jeweils überprüft werden, ob sie in die Server-Klasse hinzugefügt werden kann, ohne dass es dabei zu syntaktischen oder semantischen Fehlern kommt.

Dieser Abschnitt geht der Frage nach, welche Bedingungen für die Programmiersprache Java (Version 5) erfüllt sein müssen, damit eine gegebene Methode *m1* in eine gegebene Klasse *C* eingefügt werden darf, ohne dass es dabei zu syntaktischen oder semantischen

⁹ Sollte eine oder mehrere Dateien, für die im Rahmen des Refactorings Code-Änderungen ermittelt werden, schreibgeschützt vorliegen, so wird dem Benutzer abschließend ein Dialog mit den betreffenden Dateien angezeigt, über den er veranlassen kann, dass der Schreibschutz aufgehoben wird.

Fehlern kommt. Hierfür wurde die Java-Sprachspezifikation von Gosling et al. [GJSB05] herangezogen.

Für das Verständnis der aufgestellten Bedingungen sind eine Reihe von Begriffen im Zusammenhang mit Methoden aus der Java-Sprachspezifikation wesentlich, die hier zunächst kurz vorgestellt werden sollen. Es handelt sich hierbei um die Begriffe *Signatur*, *Subsignatur* und *override-equivalent*.

Die *Signatur* einer Methode besteht aus dem Methodennamen, der Anzahl und Typen der formalen Parameter sowie der Anzahl und Schranken der Typparameter der Methode. Die konkreten Bezeichner von Typparametern spielen für die Unterscheidung zweier Signaturen ebenso keine Rolle wie die Reihenfolge von Schranken der Typparameter.

Definition: Subsignatur: (vgl. [GJSB05] §8.4.2)

Die Signatur einer Methode $m1$ ist eine Subsignatur einer Methode $m2$, wenn gilt:

1. Methode $m2$ hat die gleiche Signatur wie $m1$ oder
2. die Signatur der Methode $m1$ ist gleich der *Erasure*¹⁰ der Signatur von $m2$

Definition: override-equivalent: (vgl. [GJSB05] §8.4.2)

Zwei Methodensignaturen $m1$ und $m2$ werden als *override-equivalent* bezeichnet, wenn $m1$ eine Subsignatur von $m2$ ist oder umgekehrt $m2$ eine Subsignatur von $m1$.

Diese Definition spielt eine Rolle bei der Frage, ob eine Methode eine andere überschreibt oder überlädt. Es folgt eine Beschreibung der ermittelten Bedingungen. Die zugehörigen Paragraphen der Java-Sprachspezifikation werden jeweils in Klammern mit genannt.

4.3.1. Methodensignaturen, die override-equivalent zueinander sind

Die erste Bedingung ist naheliegend: es darf in der Klasse C keine Methode $m2$ geben, deren Methodensignatur *override-equivalent* ist zur Methodensignatur von $m1$ ist. Wird diese Bedingung verletzt, liegt ein syntaktischer Fehler vor (vgl. [GJSB05] § 8.4.2).

¹⁰ Zum Begriff der *Erasure* vgl. [GJSB05] §4.6

4.3.2. final-deklarierte Methode in Supertyp

Eine mit dem Schlüsselwort `final` deklarierte Methode kann nicht überschrieben werden (vgl. [GJSB05] § 8.4.3.3). Daher darf es in den Supertypen der Klasse C keine Methode $m2$ geben, deren Signatur *override-equivalent* zur Methodensignatur von $m1$ ist und als `final` deklariert wurde.

```
1 public class A {
2     final public void m() {}
3 }
4
5 public class B extends A {
6     public void m() {} // compile error: cannot override the final method A.m
7 }
```

Listing 5: Überschreiben einer als `final` deklarierten Methode nicht möglich

Die Methode $B.m()$ kann nicht eingefügt werden und führt zu einer Fehlermeldung des Compilers (“Cannot override the final method A.m”).

4.3.3. Reduzierung der Sichtbarkeit von Methoden in Subtypen

Damit eine Methode $m2$ in einer Klasse C eine Methode $m1$ in einer Superklasse von C überschreiben kann, sind neben der Bedingung, dass die Signatur von $m2$ *override-equivalent* zu der Signatur von $m1$ sein muss, folgende Bedingungen an die Sichtbarkeit der Methode $m2$ geknüpft ([GJSB05] § 8.4.8.3):

1. Ist die Methode $m1$ `public`, so muss auch die Methode $m2$ `public` sein.
2. Ist die Methode $m1$ `protected`, so muss die Methode $m2$ entweder `protected` oder `public` sein.
3. Hat die Methode $m1$ `default access`, so muss die Methode $m2$ in demselben Paket wie $m1$ liegen und eine höhere Sichtbarkeit als `private` aufweisen.

Die überschreibende Methode darf also keine geringere Sichtbarkeit aufweisen als die überschriebene. Bei dem umzusetzenden Refactoring *Hide Delegate* sollen die einzuführenden Vermittlermethoden mit der Sichtbarkeit `public` erstellt werden. Für die Frage, ob eine

Methode $m1$ mit der Sichtbarkeit `public` in eine Klasse C eingefügt werden darf, muss also überprüft werden, ob in den Subklassen von C eine Methode $m2$ existiert, deren Signatur *override-equivalent* zur Signatur von $m1$ und deren Sichtbarkeit geringer ist als `public`.

Ein Einfügen einer Methode `public void m()` in die Klasse C (Listing 6) ist also nicht möglich und führt zu einem Compile-Fehler innerhalb der Klasse D (“Cannot reduce the visibility of the inherited method from C”):

```
1 public class C {
2
3 }
4
5 public class D extends C{
6     protected void m() {}
7 }
```

Listing 6: Einfügen einer Methode würde zur Reduzierung der Sichtbarkeit einer geerbten Methode führen.

4.3.4. Probleme durch Überschreiben

Die bisherigen Bedingungen führen bei einer Nichteinhaltung jeweils zu syntaktischen Fehlern. Im Kontext des Überschreibens von Methoden kann es durch Einfügen einer Methode $m1$ in eine Klasse C auch zu semantischen Änderungen des Programms kommen, die natürlich ebenfalls zu verhindern sind.

Erbt die Klasse C von ihrer Superklasse B eine Methode $m0$ und ist die einzufügende Methode $m1$ eine Subsignatur von $m0$, so würde die Klasse C nach dem Hinzufügen der Methode $m1$ nicht mehr die Methode $m0$ erben, sondern Methode $m1$ würde $m0$ überschreiben¹¹. Dies kann dann zu einem geänderten Programmverhalten führen, wenn es nämlich Aufrufstellen im Programm gibt, an denen nun nicht mehr die geerbte Methode $m0$, sondern die neu eingefügte Methode $m1$ aufgerufen wird. Das folgende Beispiel illustriert diesen Sachverhalt.

¹¹ Auf die für das Überschreiben einer Methode ebenfalls notwendigen Bedingungen bzgl. des Rückgabeparameters sowie der `throws`-Klausel wird weiter unten eingegangen

```
1 public class B {
2     public void m(){
3         System.out.println("B.m()");
4     }
5 }
6
7 public class C extends B{
8
9     // führt zu einem geänderten Programmverhalten
10    // public void m(){
11    //     System.out.println("C.m()");
12    // }
13
14    public static void main(String [] args) {
15        B b = new C();
16        b.m();        // Aufrufstelle
17    }
18 }
```

Listing 7: Eingefügte Methode *m* überschreibt eine andere und kann dadurch zu einer semantischen Änderung führen

Vor dem Einfügen der Methode *C.m()* gibt das Programm „B.m()“ aus, nach dem Einfügen würde „C.m()“ ausgegeben - die Semantik des Programms hätte sich geändert.

Aufgrund des dynamischen Bindens von Methodenaufrufen ist es nun aber nahezu unmöglich, statisch zu überprüfen, ob eine entsprechende Aufrufstelle im Programm existiert, die statt einer vor der Änderung geerbten Methode nun nach einer Änderung die eingefügte Methode aufrufen wird. Daher beschränkt sich die Bedingung darauf, dass eine solche Aufrufstelle existieren könnte. Es muss also geprüft werden, ob es in einer Superklasse von *C* eine Methode *m0* gibt, für die gilt, dass *m1* eine Subsignatur von *m0* ist und bei der *m0* in der Klasse *C* zugreifbar ist. Existiert solch eine Methode *m0*, so ist ein Einfügen der Methode *m1* nicht möglich.

An das Überschreiben von Methoden sind neben der Bedingung einer Subsignatur noch weitere Bedingungen geknüpft. So darf bspw. der Rückgabeparameter nur kovariant redefiniert werden (vgl. [GJSB05] § 8.4.5). Für die zuvor angestellten Überlegungen reicht es jedoch aus, lediglich die Subsignatureigenschaft zu überprüfen, da verhindert werden soll, dass *m1* eine (bisher geerbte) Methode *m0* überschreibt. Wäre Methode *m1* eine Subsignatur von *m0* und der Typ des Rückgabeparameters von *m1* kann den Typ des Rückgabeparameters von *m0* nicht ersetzen (nach den Regeln von § 8.4.5 der Sprachspezifikation), so überschreibt *m1* zwar nicht *m0*, es würde jedoch aufgrund der Regelverletzung

zu einem Syntaxfehler kommen.

Durch die zuvor gestellte Bedingung, die auf potentiell entstehende semantische Fehler abzielt, wird also gleichzeitig auch der genannte Syntaxfehler vermieden. Das Gleiche gilt für die Bedingungen, die bzgl. der `throws`-Klausel an überschreibende Methoden gestellt werden (vgl. [GJSB05] § 8.4.6).

Der Rückgabetyt und die `throws`-Klausel spielen allerdings eine wichtige Rolle bei der Betrachtung der Frage, ob durch Einfügen einer Methode $m1$ eine existierende Methode $m2$ in einer Subklasse von C zur überschreibenden Methode von $m1$ werden kann und ob es hierdurch zu semantischen oder syntaktischen Fehlern kommen kann. Überschreibt $m2$ bereits eine andere Methode, so sind diese Fälle bereits durch die Betrachtungen zu Beginn dieses Abschnitts sowie des Abschnitts 4.3.1 abgedeckt. Es muss also noch der Fall betrachtet werden, dass $m2$ vor dem Einfügen von $m1$ keine andere Methode überschreibt und nach dem Einfügen dann zur überschreibenden Methode von $m1$ wird.

Angenommen, die Methode $m2$ liegt in der direkten Subklasse D von C . Dann kann es nur solche Aufrufstellen von $m2$ im Programmtext geben, an denen der deklarierte Typ des Empfängers von $m2$ in der Typhierarchie maximal vom Typ D ist. Somit können keine Aufrufstellen existieren, die nach Einfügen der Methode $m1$ nicht mehr $m2$, sondern (zur Laufzeit) $m1$ aufrufen. Daher sind hierbei semantische Fehler nicht zu erwarten.

```
1 public class C {
2     // Einfügen dieser Methode führt zu Syntaxfehler in Klasse D
3     // public Integer m(){
4     //     ...
5     // }
6 }
7
8 public class D extends C {
9     public Object m(){
10        ...
11    }
12 }
```

Listing 8: Das Einfügen einer Methode verursacht Syntaxfehler in Subklasse

Anders verhält es sich jedoch bei der Frage nach syntaktischen Fehlern. Ist $m2$ eine Subsignatur von $m1$, so können in der Klasse von $m2$ syntaktische Fehler hervorgerufen werden, nämlich dann, wenn entweder der Rückgabetyt von $m2$ den Rückgabetyt von $m1$ nicht ersetzen kann oder aber die `throws`-Klausel von $m2$ mehr *checked Exceptions*

wirft als *m1*.

Das Listing 8 zeigt solch einen Fall: Würde man in der Klasse *C* eine Methode `public Integer m()` einfügen, so käme es in Klasse *D* zu einem Syntaxfehler (“The return type is incompatible with C.m()”), da die Methode *D.m()* eine Subsignatur von *C.m()* ist, aber der Rückgabotyp `Object` nicht den Rückgabotyp `Integer` ersetzen kann.

4.3.5. Name clash

Ist eine Methode *m2* in einem Suptypen namensgleich wie eine Methode *m1* in einem Supertypen und *m2* ist keine Subsignatur von *m1*, so kommt es zu einem Syntaxfehler, wenn folgende Voraussetzungen vorliegen (vgl. [GJSB05] § 8.4.8.3):

1. *m1* ist namensgleich zu *m2*
2. *m1* ist in dem deklarierenden Typ von *m2* zugreifbar
3. *m2* ist keine Subsignatur von *m1*
4. *m1* hat die gleiche *Erasure* wie *m2*

Sind alle genannten Voraussetzungen erfüllt, so liegt bezüglich *m2* weder ein Überschreiben noch ein Überladen von *m1* vor, sondern ein Syntaxfehler. In solch einem Fall liegt ein so genannter *name clash* vor.

Bei der Frage, ob eine Methode *m1* in eine Klasse *C* eingefügt werden darf, ist also ebenfalls das Auftreten eines *name clash* zu verhindern. Aus den o.g. Voraussetzungen lassen sich hierbei folgende Prüfungen ableiten.

In der Klasse *C* oder ihren Superklassen darf es keine namensgleiche Methode *m0* geben, die in *C* zugreifbar ist, bei der *m1* keine Subsignatur von *m0* ist und bei der die *Erasure* von *m1* gleich der *Erasure* von *m0* ist.

Listing 9 zeigt einen solchen Fall: Würde man die Methode `public Object m(Object o)` in die Klasse *C* einfügen, so käme es zu einem *name clash*. Die Methode *C.m(Object)* hat den gleichen Namen wie *B<String>.m(String)*, ist aber keine Subsignatur zu dieser. Jedoch haben beide Methoden die gleiche *Erasure* (nämlich *m(Object)*) und die Methode *B<String>.m(String)* ist in der Klasse *C* zugreifbar.

```
1 public class B<T> {
2     public T m(T a) { ... }
3 }
4
5 public class C extends B<String> {
6     // Name clash: The method m(Object) of type C has the same erasure
7     // as m(T) of type A<T> but does not override it
8     public Object m(Object o) { ... }
9 }
```

Listing 9: Ein *name clash* wird verursacht.

Umgekehrt muss ebenfalls verhindert werden, dass es durch Einfügen einer `public`-Methode *m1* in eine Klasse *C* zu einem *name clash* in einer Subklasse von *C* kommt. Es darf also in den Subklassen von *C* keine Methode *m2* mit dem gleichen Namen wie *m1* geben, die keine Subsignatur zu *m1* ist, jedoch die gleiche *Erasure* wie *m1* besitzt. Listing 10 zeigt ein Beispiel hierfür: Das Einfügen der Methode `public void m(Collection c)` in die Klasse *C* würde zu einem *name clash* in der Klasse *D* führen.

```
1 public class C {
2     public void m(Collection c) { ... }
3 }
4
5 public class D extends C {
6     // Name clash: The method m(Collection<String>) of type D has the same erasure
7     // as m(Collection) of type C but does not override it
8     public void m(Collection<String> c) { ... }
9 }
```

Listing 10: Ein weiterer Fall, in dem ein *name clash* verursacht wird.

4.3.6. Probleme durch Überladen

Besitzen in einer Klasse 2 Methoden den gleichen Namen und ihre Signaturen sind nicht *override-equivalent*, so spricht man davon, dass die Methoden *überladen* sind (vgl. [GJSB05] §8.4.9). Dies ist unabhängig davon, ob die Methoden geerbt wurden oder in der Klasse deklariert wurden (oder eine geerbt und eine deklariert wurde). Im Gegensatz zum Überschreiben gibt es beim Überladen keine weiteren syntaktischen Bedingungen bzgl. des Rückgabetyps der Methode oder der `throws`-Klausel.

Es können jedoch semantische Fehler im Zusammenhang mit dem Überladen entstehen.

Um bei einem Methodenaufruf bei überladenen Methoden eine Methode auszuwählen, wendet der Compiler den sogenannten Most-Specific-Algorithmus (vgl. [GJSB05] §15.12.2.5) an. Überlädt eine einzufügende Methode $m1$ eine andere $m0$ und ist sie „spezieller“ als diese, so kann dies bei einer entsprechend vorhandenen Methodenaufrufstelle dazu führen, dass nach dem Most-Specific-Algorithmus nun nicht mehr Methode $m0$ aufgerufen wird, sondern $m1$. Die Semantik des Programms kann sich dadurch ändern.

```
1 public class C{
2     public void m(Object o){
3         System.out.println("C.m(Object)");
4     }
5
6     public static void main(String[] args) {
7         C c = new C();
8         c.m(new String());    // Aufrufstelle
9     }
10
11     // folgende Methode würde zu einer Semantik-Änderung führen
12     // public void m(String s){
13     //     System.out.println("C.m(String)");
14     // }
15 }
```

Listing 11: Semantischer Fehler durch Überladen

Listing 11 zeigt ein Beispiel hierfür. Würde man die Methode `public void m(String s)` einfügen, so würde an der Aufrufstelle nicht mehr Methode `m(Object o)` aufgerufen, sondern die neu eingefügte Methode, da der deklarierte Typ des formalen Parameters (`String`) besser zum deklarierten (statischen) Typ des aktuellen Parameters an der Aufrufstelle (`String`) passt.

Informell gilt eine Methode $m1$ als *spezieller* als eine Methode $m0$, wenn alle Methodenaufrufe an $m0$ auch an $m1$ gerichtet werden können, ohne dass der Compiler hierbei Typfehler melden würde ([GJSB05] §15.12.2.5). Für die Parametertypen von $m1$ bedeutet dies, dass sie jeweils diejenigen von $m0$ *syntaktisch substituieren* (vgl. [Ste08]) können: der n .te Parametertyp von $m1$ muss ein Subtyp des n .ten Parametertypen von $m0$ sein. Damit eine einzufügende Methode $m1$ in einer Klasse C eine existierende Methode $m0$ in dieser Klasse überlädt und potentiell eine semantische Änderung wie zuvor beschrieben verursachen könnte, müssen also folgende Bedingungen erfüllt sein:

1. $m1$ hat den gleichen Namen wie $m0$

2. Die Signaturen von $m0$ und $m1$ sind nicht *override-equivalent*.
3. Die beiden Methoden haben die gleiche Anzahl an Parametern.
4. Der i .te Parametertyp ($1 \leq i \leq \text{Anzahl Parameter}$) von $m1$ muss den i .ten Parametertyp von $m0$ syntaktisch substituieren können.

Die Bedingungen gelten analog für den Fall, bei dem $m0$ von einer Superklasse von C geerbt wird. Bei der Prüfung der Vorbedingungen für das Refactoring *Hide Delegate* wird darauf verzichtet, zu untersuchen, ob tatsächlich eine Aufrufstelle im Programm existiert, die zu der beschriebenen Semantikänderung führt. Liegt eine Methode $m0$ vor, zu der die einzufügende Methode $m1$ spezieller ist, so wird das Einfügen von $m1$ nicht durchgeführt.

Bezüglich der syntaktischen Substituierbarkeit von Parametertypen, die in der 4. Bedingung erwähnt wird, können 2 Fälle unterschieden werden: Zum einen kann ein Parametertyp $T1$ einen anderen $T2$ ersetzen, wenn $T2$ ein Subtyp von $T1$ ist. Die Subtypbeziehung in Java ist hierbei auch für die primitiven Typen definiert, vgl. hierzu [GJSB05] § 4.10.1. Zum anderen kann im Rahmen des so genannten *Auto boxing / Auto unboxing* ein Wrappertyp einen primitiven Typen ersetzen und umgekehrt. Listing 12 zeigt hierfür ein Beispiel: an der Aufrufstelle liegt der primitive Typ `int` vor, der im Rahmen des *boxing* durch den Wrappertyp `Integer` ersetzt wird und zum Aufruf der Methode $B.m(Integer)$ führt. Ein Einfügen der Methode `public void m(int i)` würde nun dazu führen, dass an der Aufrufstelle die neue Methode aufgerufen würde und somit die Semantik des Programms geändert wäre.

```

1 public class B{
2     public void m(Integer i){
3         System.out.println("B.m(Integer)");
4     }
5 }
6
7 public class C extends B{
8     public static void main(String[] args) {
9         C c = new C();
10        c.m(1);    // Aufrufstelle
11    }
12
13    // folgende Methode würde zu einer Semantik-Änderung führen
14    // public void m(int i){
15    //     System.out.println("C.m(int)");
16    // }

```

17 | }

Listing 12: Beispiel für Auto boxing

Bezüglich des Überladens wurden bislang existierende Methoden untersucht, die entweder in der Klasse C deklariert werden oder aber in C als geerbte Methoden zur Verfügung stehen. Es fehlt noch die Betrachtung von Methoden in den Subklassen von C . Die in einer Klasse C einzufügende Methode $m1$ darf ebenfalls nicht eine andere Methode $m2$ in einer Subklasse D von C derart überladen, dass sie „spezieller“ ist als $m2$ (die Subklasse D erbt aufgrund der Vererbungsbeziehung die Methode $m1$).

```

1 public class C {
2     // Einfügen dieser Methode würde zu einer Semantikänderung führen
3     // public void m(String s){
4     //     System.out.println("C.m(String)");
5     // }
6 }
7
8 public class D extends C{
9     public void m(Object o){
10        System.out.println("D.m(Object)");
11    }
12
13    public static void main(String[] args) {
14        D d = new D();
15        d.m("abc");    // Aufrufstelle
16    }
17 }

```

Listing 13: Semantischer Fehler durch Überladen in einem Subtyp.

Diesen Fall veranschaulicht Listing 13: die Subklasse D enthält eine Methode m mit dem deklarierten Typ des formalen Parameters `Object`. Eine Methode `m(String)` ist spezieller als diese, da aufgrund der Subtypbeziehung zwischen `String` und `Object` die Methode `m(Object)` alle existierenden Methodenaufrufe von `m(String)` syntaktisch substituieren kann, ohne dass es hierbei zu Typfehlern des Compilers kommt. Ein Einfügen der Methode `m(String)` in die Klasse C führt zu einer Semantikänderung, da an der Aufrufstelle nicht mehr die Methode $D.m()$, sondern $C.m()$ aufgerufen wird.

Im Zusammenhang mit dem Überladen von Methoden kann auch ein syntaktischer Fehler durch Einfügen einer Methode hervorgerufen werden: überlädt die einzufügende Methode eine andere und existiert ein Methodenaufruf im Programm, bei dem der Compiler

nach Anwenden des *Most-Specific-Algorithmus* nicht eindeutig die „am besten passende“ Methode ermitteln kann, so wird ein Syntaxfehler gemeldet (*method is ambiguous*). Das nachfolgende Listing 14 zeigt hierzu ein Beispiel.

```
1 public class C {
2     public void m(Object o, String s) { }
3
4     public void m(String s, Object o) { }
5
6     public static void main(String[] args) {
7         C c = new C();
8         c.m("abc", "def");           // method is ambiguous
9     }
10 }
```

Listing 14: Syntaktische Fehler (method is ambiguous) im Zusammenhang mit dem Überladen

Im Rahmen des zu erstellenden Plugins wird jedoch auf eine Überprüfung und Verhinderung dieses Syntaxfehlers verzichtet, da es bedeuten würde, den *Most-specific-Algorithmus* auf die einzufügende Methode, alle vorhandenen gleichnamigen Methoden, die überladen würden und alle vorhandenen Aufrufstellen im Programm anzuwenden, um feststellen zu können, dass es zu diesem Syntaxfehler kommt.

4.3.7. Geschachtelte Klassen

Im Zusammenhang mit geschachtelten Klassen kann es ebenfalls zu syntaktischen und semantischen Fehlern kommen. Dies kann dann auftreten, wenn es sich bei der Klasse *C*, in die eine Methode *m1* eingefügt werden soll, um eine innere Klasse handelt. Die Methode *m1* könnte dann eine Methode *m0* verdecken¹² und so zu syntaktischen und semantischen Fehlern führen. Listing 15 zeigt zunächst ein Beispiel für geschachtelte Klassen: die Klasse *B* wird als *Top-level-Klasse* bezeichnet, die die innere Klasse *C* deklariert.

```
1 public class B {
2
3     public void m(){
4         System.out.println("B.m()");
5     }
6 }
```

¹² Die Sprachspezifikation spricht hierbei von *shadowing declarations* vgl. [GJSB05] §6.3.1.

```
7 public class C {
8     public void doSomething(){
9         m();
10    }
11 }
12
13 public static void main(String [] args){
14     B b = new B();
15     B.C c = b.new C();
16     c.doSomething();
17 }
18 }
```

Listing 15: Beispiel für geschachtelte Klassen

In der inneren Klasse *C* wird die Methode *m* der äußeren Klasse *B* aufgerufen. Das Programm gibt `B.m()` aus. Würde man nun, wie in Listing 16 geschehen, in der inneren Klasse *C* eine gleichnamige Methode mit einer *override-equivalent* Signatur einfügen, so kann sich dadurch die Semantik des Programms ändern: nun gibt das Programm `C.m()` aus.

```
1 public class B {
2
3     public void m(){
4         System.out.println("B.m()");
5     }
6
7     public class C {
8         public void doSomething(){
9             m();
10        }
11
12        public void m(){
13            System.out.println("C.m()");
14        }
15    }
16
17    public static void main(String [] args){
18        B b = new B();
19        B.C c = b.new C();
20        c.doSomething();
21    }
22 }
```

Listing 16: Semantikänderung bei geschachtelten Klassen

Auf ähnliche Weise kann ein Syntaxfehler hervorgerufen werden, wenn die Signatur von $m1$ nicht *override-equivalent* zu $m0$ ist (aber den gleichen Namen aufweist). Würde man beispielsweise statt `public void m()` in Zeile 12 `public void m(int i)` schreiben, so würde der Compiler für Zeile 9 einen Syntaxfehler melden

4.3.8. Zusammenfassung der Bedingungen

Bei einer einzufügenden Methode $m1$, die als `public` deklariert wird, in eine Klasse C müssen folgende Bedingungen erfüllt sein, damit es zu keinen syntaktischen oder semantischen Fehlern kommt:

1. Innerhalb der Klasse C darf keine andere Methode $m2$ existieren, die *override-equivalent* zu Methode $m1$ ist (vgl. Abschnitt 4.3.1).
2. In den Superklassen der Klasse C darf keine als `final` deklarierte Methode $m0$ existieren, bei der $m1$ eine Subsignatur von $m0$ ist (vgl. Abschnitt 4.3.2).
3. In den Subklassen von C darf keine Methode $m2$ existieren, die eine Subsignatur zu $m1$ ist und deren Sichtbarkeit nicht `public` ist (vgl. Abschnitt 4.3.3).
4. In den Superklassen von C darf keine Methode $m0$ existieren, bei der $m1$ eine Subsignatur von $m0$ und $m0$ in C zugreifbar ist (vgl. Abschnitt 4.3.4).
5. In den Subklassen von C darf keine Methode $m2$ existieren, die eine Subsignatur von $m1$ ist und die nicht die weiteren Bedingungen für das Überschreiben von Methoden erfüllt (der Rückgabotyp muss den von $m1$ ersetzen können sowie die `throws`-Klausel darf nicht mehr *Checked exceptions* werfen als $m1$) (vgl. Abschnitt 4.3.4).
6. In der Klasse C oder ihren Superklassen darf es keine namensgleiche Methode $m0$ geben, die in C zugreifbar ist, bei der $m1$ keine Subsignatur von $m0$ ist und bei der die *Erasure* von $m1$ gleich der *Erasure* von $m0$ ist (vgl. Abschnitt 4.3.5).
7. In den Subklassen von C darf es keine Methode $m2$ mit dem gleichen Namen wie $m1$ geben, die keine Subsignatur zu $m1$ ist, jedoch die gleiche *Erasure* wie $m1$ besitzt (vgl. Abschnitt 4.3.5).

8. Es darf keine namensgleiche Methode $m2$ mit derselben Anzahl an Parametern wie $m1$ geben, für die gilt:
 - a) $m2$ wird in Klasse C oder einer Subklasse von C deklariert oder sie liegt in Klasse C als geerbte Methode vor
 - b) Der i .te Parametertyp von $m1$ kann den i .ten Parametertyp von $m2$ syntaktisch substituieren (vgl. Abschnitt 4.3.6).
9. Handelt es sich bei der Klasse C um eine innere Klasse, so darf es in den C umschließenden äußeren Klassen keine namensgleiche Methode $m0$ geben. Wäre die Signatur der namensgleichen Methode $m0$ *override-equivalent* zur Methode $m1$, so kann dadurch ein semantischer Fehler entstehen; andernfalls kommt es zu einem syntaktischen Fehler (vgl. Abschnitt 4.3.7).

4.3.9. Fazit aus den Vorbedingungen

Für die im Abschnitt 4.3 aufgestellten Bedingungen kann nicht bewiesen werden, dass diese Liste vollständig ist. Die Aufstellung macht jedoch deutlich, wie umfangreich und komplex bereits diese Liste an Vorbedingungen für das Einfügen einer Methode in einer Klasse ist. Alle zuvor genannten Bedingungen haben gemeinsam, dass der Methodennamen der neu einzufügenden Methode immer eine ausschlaggebende Bedeutung hat. Für die Umsetzung innerhalb des zu erstellenden Refaktorisierungswerkzeugs soll daher ein neuer Methodennamen erzeugt werden, für den gefordert wird, dass er in der Typhierarchie der betroffenen Klasse neu ist. Durch diesen Schritt kann die Umsetzung aller in diesem Abschnitt ermittelten Vorbedingungen ausbleiben. Dies reduziert den Aufwand für die Umsetzung, das Laufzeitverhalten und insbesondere auch die Fehleranfälligkeit des Refaktorisierungswerkzeugs in bedeutendem Maße. Der Name der neuen Methode wird sich aus dem Namen der Methode der Delegate-Klasse sowie einem eindeutigen Postfix zusammensetzen. Der Benutzer hat nach Durchführung des Refactorings die Möglichkeit, falls ihm der bzw. die neuen Methodennamen nicht gefallen, diese über ein *Rename* zu ändern.

4.4. Bedingungen für das Löschen der selektierten Methode

Das Refaktorisierungswerkzeug prüft ebenfalls, ob es möglich ist, die vom Benutzer selektierte Methode zu löschen, da diese ja das zu versteckende Delegate-Objekt bislang preisgibt. Um die selektierte Methode löschen zu können, ohne dabei syntaktische oder semantische Fehler zu erzeugen, müssen eine Reihe von Bedingungen erfüllt sein, die im Folgenden erläutert werden.

Wenn die selektierte Methode eine abstrakte Methode oder ein Interface implementiert, so darf die Methode nicht gelöscht werden, da andernfalls ein Syntaxfehler hervorgerufen wird. Damit die zu erzeugenden Vermittlermethoden Aufrufe direkt an ein Delegate-Objekt weiterleiten können, muss die Bedingung erfüllt sein, dass der Methodenrumpf einzig und allein aus einem Return-Statement der Art

```
return x oder return this.x
```

besteht, wobei `x` jeweils für eine Instanzvariable stehen soll. Ist dies nicht gegeben, so müssen die Vermittlermethoden Aufrufe an die selektierte Methode weiterleiten. Enthält die selektierte Methode beispielsweise vor dem Return-Statement einen Ausdruck der Form `System.out.println('abc')`, so würde bei Weiterleitung von Aufrufen direkt an das Delegate-Objekt die Semantik des Programms geändert. Da also die selektierte Methode weiterhin (mindestens) von den Vermittlermethoden aufgerufen werden muss, kann sie in diesen Fällen nicht entfernt werden.

Es darf ganz allgemein kein Methodenaufruf nach Durchführung des Refactorings mehr zur selektierten Methode vorliegen. Dabei gibt es verschiedene Arten von Methodenaufrufen, die im Rahmen des Refactorings untersucht werden, bei denen aber keine Änderung des Methodenaufrufs erfolgt. Sei *method1* die selektierte Methode. So wird bei den folgenden Aufrufen von *method1* die Aufrufstelle nicht verändert:

1. Das Objekt, welches *method1* zurückgibt, wird als Parameter innerhalb eines Methodenaufrufs verwendet und es erfolgt kein weiterer Methodenaufruf auf *method1* (Beispiel: `m(param1, foo.method1(), param3)`).
2. Die selektierte Methode wird in einem Subtypen mittels `super` aufgerufen.
3. Handelt es sich bei der Methode, die auf dem Delegate-Objekt aufgerufen wird, um eine statische Methode, so wird hierfür keine statische Vermittlermethode eingefügt, da statische Methoden auf statische Weise referenziert werden sollten. Beispiel:

`foo.method1().doSomething()`, wobei `doSomething()` hierbei für eine statische Methode stehen soll.

4. Für eine Verletzung des Law of Demeter muss nicht zwingend ein verketteter Methodenaufruf vorliegen. Erfolgt beispielsweise zunächst eine Zuordnung des Delegate-Objekts zu einer temporären Variablen und dann später ein Methodenaufruf über diese temporäre Variable, so kann ebenfalls eine Verletzung des Gesetzes von Demeter vorliegen. Abbildung 17 illustriert diesen Fall. Eine weitere automatische Untersuchung der Verwendung der temporären Variablen ist jedoch sehr umfangreich und liegt außerhalb des Rahmens des Refactorings, so dass entsprechende Programmstellen dazu führen, dass die selektierte Methode nicht gelöscht werden darf.

```
1 public class A {
2     public void m(Foo foo) {
3         Delegate temp = foo.method1();
4         ...
5         temp.doSomething();
6     }
7 }
```

Listing 17: Verletzung des Gesetzes von Demeter auch ohne verketteten Methodenaufruf

Die Methode `method1` wird ebenfalls dann nicht gelöscht, wenn sie nicht parameterlos ist: Da man grundsätzlich unterstellen sollte, dass eine Methode mit Parametern auch etwas mit diesen Parametern innerhalb des Methodenrumpfs macht, werden die Parameter von `method1` mit in die Parameterliste der Vermittlermethode aufgenommen und die Vermittlermethode ruft dann `method1` auf.

Beispiel: eine Aufrufstelle

```
foo.method1(param1).doSomething()
```

wird geändert in

```
foo.doSomething(param1).
```

Innerhalb der Vermittlermethode `doSomething` erfolgt dann ein Aufruf der Form

```
method1(param1).doSomething().
```

5. Implementierung

Das folgende Kapitel beschreibt die Umsetzung des Refaktorisierungswerkzeugs, welches als Plug-in für die Entwicklungsumgebung Eclipse (Version 3.5.1) realisiert wurde. Eclipse wurde gewählt, da es als Java IDE weit verbreitet ist, kostenlos verfügbar ist und bereits ein Refactoring-Framework enthält, welches das Hinzufügen eigener Refactorings unterstützt und erleichtert.

Dieses Refactoring-Framework wird im ersten Abschnitt näher vorgestellt. Es beschreibt zudem einige Teile der Java Development Tools (JDT), die wesentlich für die Realisierung des Refaktorisierungswerkzeugs sind. Der darauf folgende Abschnitt erläutert den Aufbau des Refactoring-Plug-ins und Abschnitt 5.3 beschreibt abschließend einige wesentliche Klassen.

5.1. Einführung Refactorings in Eclipse

Im folgenden Abschnitt werden das Eclipse-Refactoring-Framework sowie einige Teile der Java Development Tools näher vorgestellt. Für eine allgemeine Einführung in die Entwicklung von Eclipse-Plug-ins sei auf [CR08] und [DFK⁺04] verwiesen.

5.1.1. Refactoring-Framework

In Eclipse sind bereits eine Vielzahl automatisierter Refactorings (innerhalb der JDT) für die Programmiersprache Java enthalten. Mit der Version 3.1 von Eclipse wurde derjenige Teil, der sich bei einer Implementierung eines Refactorings immer wiederholte, in eine sprachunabhängige Schicht, die als Language Toolkit (LTK) bezeichnet wird, ausgelagert ([Fre05]). Somit ist es möglich, Refactorings nicht nur für Java, sondern auch für andere Sprachen und Anwendungsfälle zu schreiben und in Eclipse zu integrieren. Das Eclipse-Refactoring-Framework als Teil des LTK besteht aus den beiden Plug-ins `org.eclipse.ltk.core.refactoring` und `org.eclipse.ltk.ui.refactoring`.

Die wesentlichen, an einem Refactoring-Prozess beteiligten Elemente des Refactoring-Frameworks stellt Abbildung 5 dar. Ein *Refactoring Implementer* benötigt Kenntnis über diejenigen Elemente (*Selected Element*), auf die das Refactoring angewendet werden

soll. Ebenfalls benötigt er Zugriff auf die Ressourcen des Workspaces (*Workspace Resources*). Über entsprechende Dialoge hat der Benutzer die Möglichkeit, Eingaben für das durchzuführende Refactoring zu tätigen (bspw. einen neuen Namen bei einem Rename-Refactoring einzugeben). Dem Benutzer wird zudem angeboten, dass er sich die Änderungen vorher in einer Vorschau anzeigen lassen kann. Der *Refactoring Implementer* ist insbesondere für die Überprüfung der Vorbedingungen eines Refactorings sowie für die Erzeugung von Änderungen (*Change*) zuständig, die auf den Ressourcen des Eclipse-Workspace angewendet werden können.

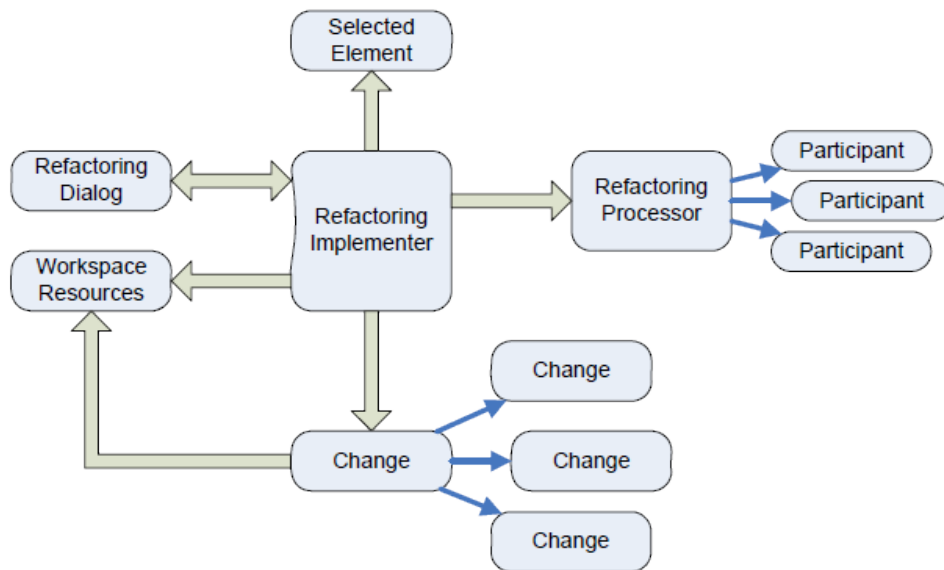


Abbildung 5: Abstrakte Sicht auf die wesentlichen Elemente des Refactoring-Frameworks (Quelle: [Pet07])

Das Refactoring-Framework von Eclipse bietet zudem die Möglichkeit an, dass andere Plug-ins an einem Refactoring (über den Extension-Mechanismus) teilhaben können. Dieser Aspekt des Frameworks wird als *Refactoring Participants* bezeichnet. In der Abbildung 5 zählen die Elemente *Refactoring Processor* und *Participant* hierzu. Refactoring Participants kommen beispielsweise beim Rename-Refactoring zum Einsatz. Soll ein Klassenna-
 me umbenannt werden, so muss bei der Entwicklung von Plug-ins beachtet werden, dass dies sich auch auf die Manifest-Dateien (Manifest.mf und plugin.xml) auswirken kann, da diese oft auch Namen von Java-Klassen enthalten. Seit Eclipse 3.2 wird eine notwendige Änderung der Manifest-Dateien von der Plug-in Development Environment (PDE) von Eclipse überwacht und durchgeführt. Dies geschieht über den Ansatz der Refactoring Participants, indem das PDE an den Java-Rename-Refactorings partizipiert ([Fre05]). Wird

in einem Java-Quellcode eine Umbenennung eines Klassennamens angestoßen, so wird eine ggf. vorhandene Deklaration innerhalb der Manifest-Dateien automatisch mit geändert.

Darüber hinaus bietet das Refactoring-Framework eine Infrastruktur an, über die man eigene Refactorings an der Historisierung und an den Scripting-Möglichkeiten von Eclipse teilhaben lassen kann (im „Refactor“-Menü sind dies die Menüpunkte „Create Script...“, „Apply Script...“ und „History...“).

Den Kern eines Refactorings stellt der *Refactoring Implementer* dar. Dieser muss die abstrakte Klasse `org.eclipse.ltk.core.refactoring.Refactoring` implementieren. Den Ablauf dieser Klasse und eines Refactoring-Prozesses des Refactoring-Frameworks stellt Abbildung 6 dar. Zu Beginn erfolgt die Initialisierung eines Refactorings. Hierbei werden insbesondere auch die selektierten Elemente, auf die das Refactoring anzuwenden ist, der Klasse übergeben. Anschließend führt diese Klasse eine Reihe von initialen Prüfungen durch (Methode `checkInitialConditions`). Diese Prüfungen beziehen sich bspw. typischerweise darauf, festzustellen, dass die zugehörigen Ressourcen als editierbarer Quellcode vorliegen und keine Compile-Fehler aufweisen. Des Weiteren werden erste Prüfungen durchgeführt, ob das Refactoring auf den selektierten Elementen anwendbar ist.

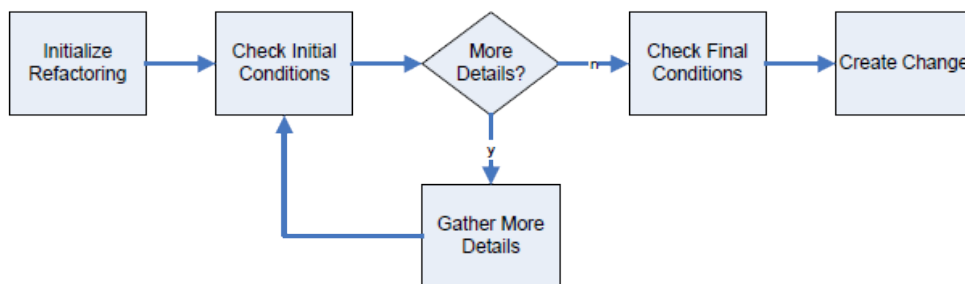


Abbildung 6: Lebenszyklus eines Refactorings (Quelle: [Pet07])

Oftmals werden für ein Refactoring weitere Eingaben des Benutzers benötigt, die über entsprechende Dialoge einzugeben sind. Hat der Benutzer alle erforderlichen Eingaben getätigt, so werden zunächst noch einmal die initialen Bedingungen geprüft. Werden hierbei keine Verletzungen der Bedingungen festgestellt, wird anschließend die Methode `checkFinalConditions` aufgerufen. Diese Methode hat die Aufgabe, alle Vorbedingungen des Refactorings zu überprüfen. Hierzu werden oftmals (laufzeitintensive) Untersuchungen des Abstrakten Syntaxbaumes (AST) notwendig sein. Neben den Prüfungen übernimmt diese Methode häufig auch die Aufgabe, bereits die notwendigen Daten für die spätere Durchführung von Änderungen zu sammeln. Das Durchlaufen von `checkInitialCon-`

`conditions` und `checkFinalConditions` wird dabei als Prüfung der Vorbedingungen eines Refactorings bezeichnet ([Pet07]). Schlägt eine oder mehrere der Bedingungen fehl, so wird dem Benutzer eine entsprechende Meldung angezeigt.

Sind alle Vorbedingungen erfüllt, so obliegt es der Methode `createChange`, alle notwendigen Änderungen an Elementen des Workspace zu beschreiben und zurückzugeben. Hierbei greift die Methode wie oben angedeutet häufig auf die innerhalb der Methode `checkFinalConditions` gesammelten Informationen zurück. Die Methode gibt ein Objekt vom Typ `org.eclipse.ltk.core.refactoring.Change` zurück. Dieses Change-Objekt wird vom Refactoring-Dialog des Frameworks dazu verwendet, dem Benutzer eine Vorschau über die Änderungen anzuzeigen. Zu diesem Zeitpunkt sind also die von einer Änderung betroffenen Ressourcen (bspw. Java-Dateien) noch unverändert. Bestätigt der Benutzer die Vorschau bzw. hat er auf die Anzeige einer Vorschau vorher schon verzichtet, werden die Angaben aus dem Change-Objekt vom Framework dazu verwendet, die Änderungen auf dem Workspace tatsächlich durchzuführen ([Wid07]).

5.1.2. Kernkomponenten der Java Development Tools (JDT)

Für die programmgesteuerte Manipulation von Java-Quellcode bieten die Java Development Tools (JDT) von Eclipse einige hilfreiche Komponenten an, von denen einige für das erstellte Refaktorisierungswerkzeug wesentliche Teile im folgenden kurz vorgestellt werden sollen. Es handelt sich dabei um die Java Search Engine, das Java-Modell sowie den Abstrakten Syntaxbaum (engl. abstract syntax tree, AST).

Mit Hilfe der Java Search Engine ist es möglich, nach Java-Elementen wie Methodendeklarationen, Typdeklarationen, Paketdeklarationen usw. zu suchen. Der Bereich, in dem gesucht werden soll, lässt sich dabei vorgeben. Möglich sind bspw. Suchen innerhalb des gesamten Workspace, eines Projekts, eines Pakets oder aber auch innerhalb der Typhierarchie eines vorzugebenden Typs. Neben dem Suchbereich ist der Search Engine ein Suchmuster zu übergeben. Für Refactorings von besonderem Interesse ist dabei die Möglichkeit, dass man mit Hilfe der Search Engine auch nach Referenzen bspw. von Methoden suchen kann. Einstiegspunkt für eine Java Suche ist die Klasse `org.eclipse.jdt.core.search.SearchEngine`.

Das Java-Modell bildet Elemente ab, die für das Erstellen, Verändern, Navigieren und Analysieren von Java-Programmen verwendet werden können. Das Modell ist in einer

Baumstruktur aufgebaut. An der Spitze steht das Element `IJavaModel`, welches den Workspace von Eclipse abbildet. Eine Auswahl der Elemente des Java-Modells zeigt Abbildung 7. Eine `.java`-Datei wird dabei durch den Typ `ICompilationUnit` abgebildet. Abbildung 8 zeigt beispielhaft einige derjenigen Elemente, aus denen sich eine `ICompilationUnit` zusammensetzen kann.

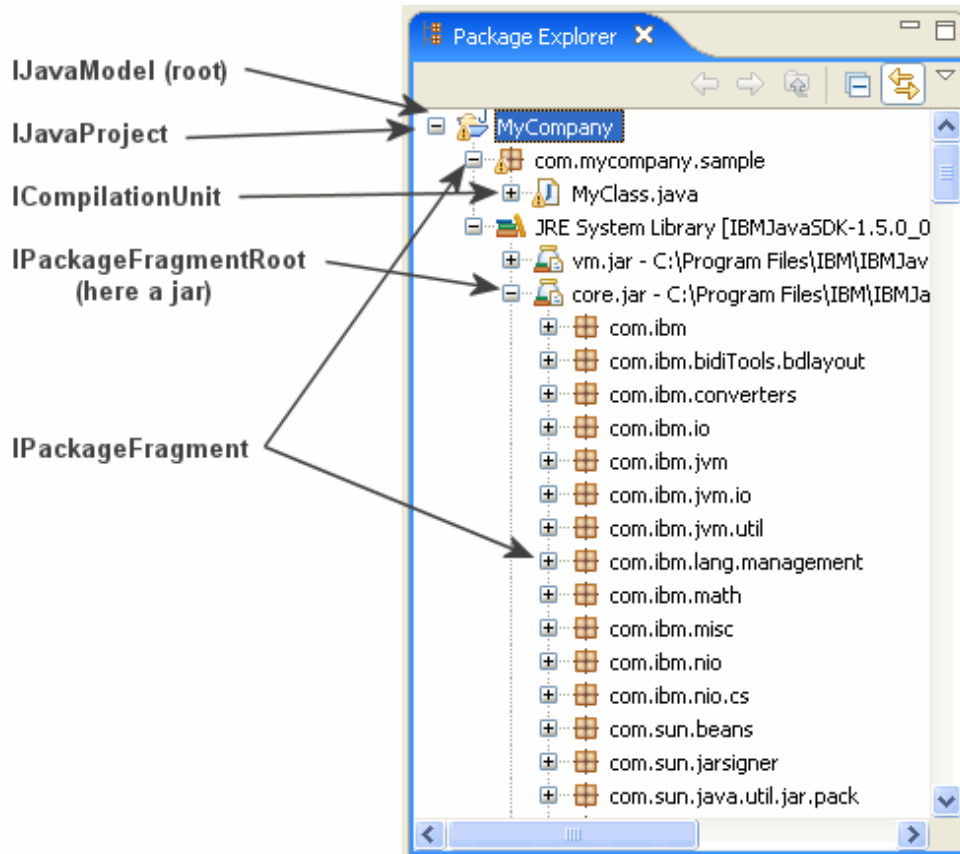


Abbildung 7: Elemente des Java-Modells (Quelle: [Ecl09])

Alle Elemente des Java-Modells implementieren das Interface `IJavaElement`, welches Funktionalitäten zum Navigieren innerhalb der Baumstruktur sowie zur Analyse des Elements bereithält. Für eine detaillierte Darstellung des Java-Modells sei auf [Aes08] und [Ecl09] verwiesen.

Das Java-Modell enthält jedoch lediglich eine schlanke Sicht auf ein Java-Programm. Einfache Abfragen und Änderungen können mit Hilfe des Java-Modells durchgeführt werden. Entworfen wurde es jedoch insbesondere vor dem Hintergrund, die Struktur eines Java-Projekts oder einer Java-Klasse schnell in einer View von Eclipse anzeigen zu können.

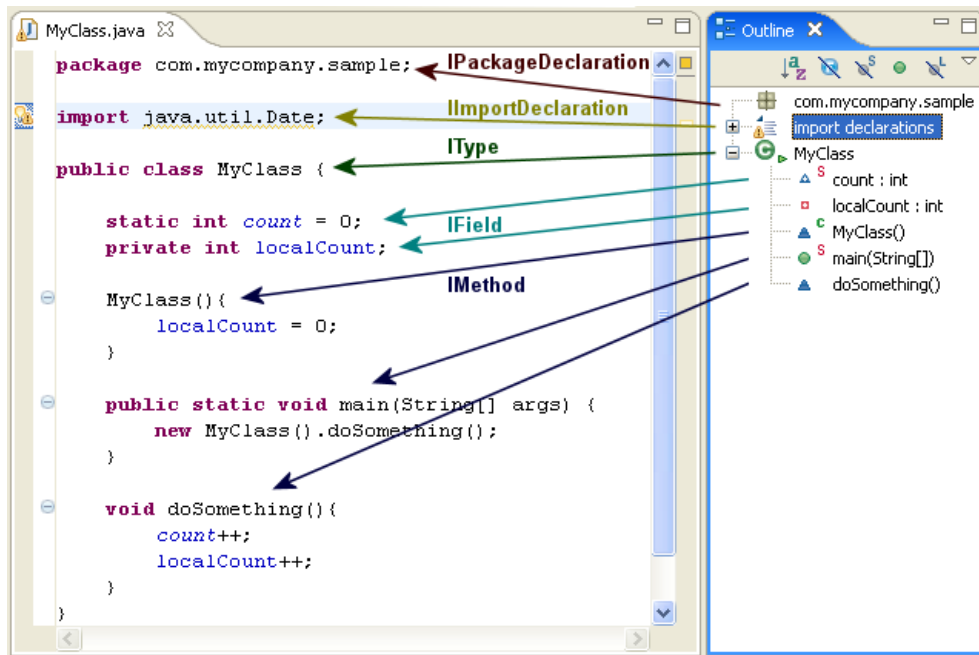


Abbildung 8: Kindelemente von ICmpilationUnit (Quelle: [Ecl09])

Für komplexere Analysen und Änderungen eines Programmcodes wird ein abstrakter Syntaxbaum (Abstract Syntax Tree, AST) benötigt. Der AST bildet sämtliche Sprachkonstrukte von Java in einer Baumstruktur ab. Er ist vergleichbar mit dem Document Object Model (DOM) einer xml-Datei [TK06]. Jeder Knoten dieser Baumstruktur ist dabei ein Subtyp von `org.eclipse.jdt.core.dom.ASTNode` und für jedes Sprachkonstrukt existiert ein eigener Knotentyp, bspw. für Variablendeklarationen, Importdeklarationen, Ausdrücken usw. Abbildung 9 zeigt eine Visualisierung eines AST für ein „Hello World“-Java-Programm, die mittels des AST-Viewer-Plug-ins¹³ erzeugt wurde. An der Abbildung wird deutlich, dass selbst ein kleines, einfaches Programm bereits in einen komplexen Syntaxbaum mündet.

Erzeugt werden kann ein AST mit Hilfe der Klasse `ASTParser`, der als Eingabeparameter eine Kompilationseinheit (entspricht einer Java-Datei) oder aber auch nur ein Code-Abschnitt übergeben werden kann. Zum Traversieren eines AST kann ein `ASTVisitor` verwendet werden, der auf dem Visitor-Pattern (siehe [GHJV95]) basiert. Die API der JDT bieten Factory-Methoden für jeden Knotentyp an, um einem AST neue Knoten hinzuzufügen, bspw. für eine neue Methodendeklaration, wie sie im Rahmen des Refaktorisierungswerkzeugs für die Erzeugung der neuen Vermittlermethoden benötigt wird.

¹³ Das Plug-in ist über diese Seite verfügbar: <http://www.eclipse.org/jdt/ui/astview/index.php>

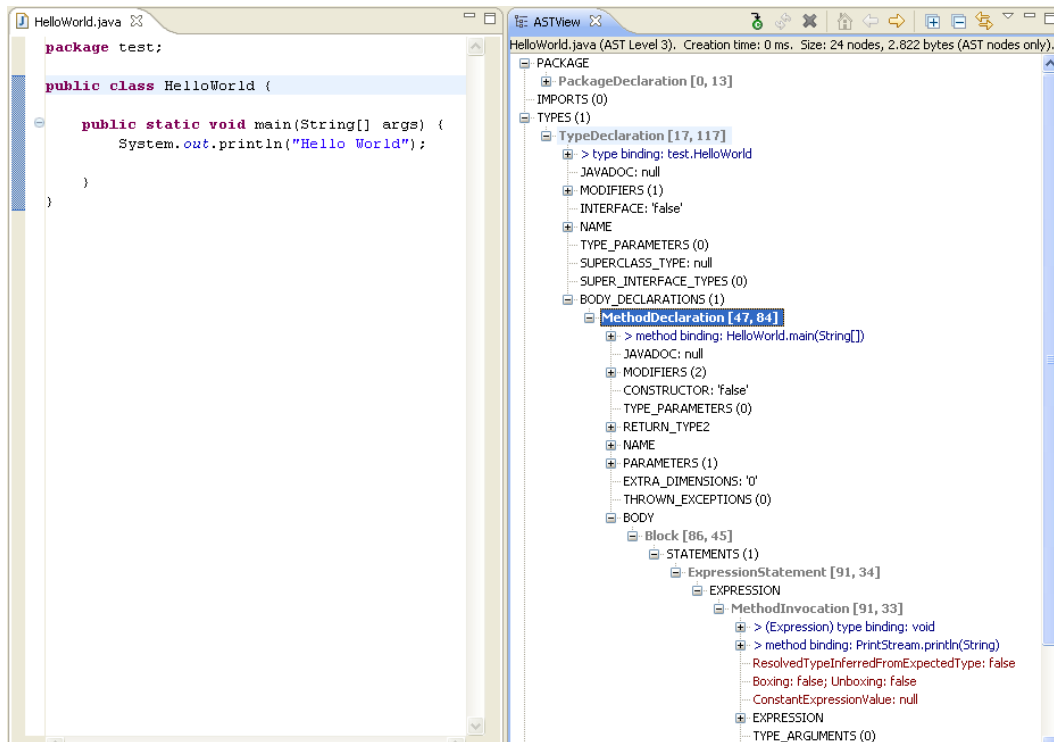


Abbildung 9: Visualisierte Darstellung eines AST anhand eines Hello-World-Programms

Für die Modifizierung eines AST gibt es 2 Möglichkeiten: der AST kann entweder direkt geändert werden oder die Änderungen werden in einem separaten Protokoll aufgezeichnet (dies geschieht mit Hilfe einer Instanz der Klasse `ASTRewrite`). Wird ein AST geändert, so wirken sich die Änderungen nicht unmittelbar auf die zugehörige Java-Datei aus. Um die durchgeführten Änderungen tatsächlich auf der Java-Datei auszuführen, ist zuvor ein Objekt vom Typ `TextEdit` zu erstellen, welches eine textbasierte Darstellung der Änderungen enthält und bspw. über die Klasse `ASTRewrite` zu bekommen ist. Für eine detailliertere Beschreibung des Abstract Syntax Tree sei auf [TK06] verwiesen.

5.2. Aufbau

In diesem Abschnitt wird die Paketstruktur des umgesetzten Refactoring-Plug-ins kurz vorgestellt, bevor im nächsten Abschnitt einige wesentliche Klassen beschrieben werden. Das Plug-in setzt sich aus den folgenden 5 Paketen zusammen:

1. `org.intoj.hidedelegate`: Dieses Paket beinhaltet die Activator-Klasse des Plug-

ins, welche den Lebenszyklus des Plug-ins steuert. Sie wird automatisch vom *Eclipse Plug-in Project Wizard* generiert. Daher wird im Folgenden nicht näher auf diese Klasse eingegangen.

2. `org.intoj.hidedelegate.actions`: Das Paket enthält eine Action-Klasse, welches das Interface `org.eclipse.ui.IObjectActionDelegate` implementiert. Der Einstiegspunkt für den Benutzer des Refactorings ist eine von ihm selektierte Methode und die Auswahl des Menüeintrags „Hide Delegate“ aus dem Kontextmenü heraus. Diese Auswahl löst eine *Action* aus, welche von der Klasse `HideDelegateAction` behandelt wird. Diese instanziiert die Refactoring-Klasse `HideDelegateRefactoring`, initialisiert das Refactoring mit der selektierten Methode und ruft eine vom LTK bereitgestellte Methode zum Öffnen des Refactoring-Dialogs auf.
3. `org.intoj.hidedelegate.refactoring`: In diesem Paket ist die Kernfunktionalität des Refaktorisierungswerkzeugs enthalten. Auf die Funktionsweise wesentlicher Klassen dieses Pakets wird im nächsten Abschnitt detaillierter eingegangen. Es beinhaltet u.a. auch die Umsetzung des im Abschnitt 5.1.1 vorgestellten *Refactoring Implementers*.
4. `org.intoj.hidedelegate.ui`: Enthält die Klassen für die graphische Benutzeroberfläche. Zum einen ist die Klasse `HideDelegateWizard` enthalten, welche die abstrakte Klasse `RefactoringWizard` des LTK implementiert. Zum anderen existiert die Klasse `HideDelegateWizardInputPage`, welche für die Darstellung der ersten Eingabeseite des Refactoring-Dialogs verantwortlich ist.
5. `org.intoj.hidedelegate.util`: Beinhaltet eine Util-Klasse (`HideDelegateUtil`), die einige Hilfsmethoden bspw. für das Erzeugen eines AST oder aber auch für die Erzeugung eines eindeutigen Methodennamens innerhalb einer Typhierarchie zur Verfügung stellt.

5.3. Funktionsweise wesentlicher Klassen

In diesem Abschnitt sollen einige wesentliche Klassen des Paketes `org.intoj.hidedelegate.refactoring` näher vorgestellt und auf Besonderheiten hingewiesen werden. Die Reihenfolge der Beschreibung orientiert sich dabei am Lebenszyklus eines Refactorings, wie er in Abschnitt 5.1.1 und Abbildung 6 bereits dargestellt wurde.

Eine der zentralen Klassen ist `HideDelegateRefactoring`, welche die abstrakte Klasse `org.eclipse.ltk.core.refactoring.Refactoring` des LTK implementiert. Sie besitzt eine Methode `setMethod(IMethod)`, über die das Refactoring mit der vom Benutzer selektierten Methode initialisiert werden kann. Nach erfolgter Initialisierung wird vom Refactoring-Framework aus die Methode `checkInitialConditions` aufgerufen, welche die initialen Vorbedingungen des Refactorings überprüft. Dies sind die in Abschnitt 4.2 beschriebenen Bedingungen (bspw. dass ein veränderbarer Quellcode vorliegt und keine Compile-Fehler existieren), für die die JDT entsprechende Prüfmethode zur Verfügung stellen.

Sind alle initialen Vorbedingungen erfüllt, so wird dem Benutzer die erste Seite des Refactoring-Dialogs angezeigt, für deren Aussehen die Klasse `HideDelegateWizardInputPage` verantwortlich ist. Der Benutzer hat hier die Möglichkeit, 2 Optionen des Refaktorisierungswerkzeugs zu verändern: zum einen kann er beeinflussen, ob die selektierte Methode am Ende des Refactorings automatisch gelöscht werden soll, falls hierfür alle erforderlichen Bedingungen (vgl. Abschnitt 4.4) erfüllt sind. Zum anderen kann er angeben, ob im Rahmen des Refactorings Interfaces grundsätzlich erweitert werden sollen, falls diese involviert sind. Für diese beiden Optionen stellt die Klasse `HideDelegateRefactoring` entsprechende `get-` und `set-`Methoden zur Verfügung.

Nach Bestätigung der Dialogseite ruft das Refactoring-Framework die Methode `checkFinalConditions` auf, welche alle weiteren Vorbedingungen zu prüfen hat. Bei der Implementierung dieser Methode werden jedoch, wie in Abschnitt 5.1.1 bereits angedeutet, neben der Überprüfung von Bedingungen auch sämtliche Code-Änderungen bestimmt, die für eine spätere Durchführung des Refactorings notwendig sind. Da dieser Teil des Refaktorisierungswerkzeugs einen großen Umfang einnimmt, wurde diese Funktionalität in eine eigene Klasse `CheckFinalConditions` ausgelagert.

Als Einstiegspunkt dieser Klasse dient die gleichnamige Methode `checkFinalConditions`. Der Ablauf dieser Methode lässt sich dabei grob in folgende Punkte unterteilen:

1. Zunächst finden weitere Prüfungen bzgl. der selektierten Methode statt. Hierbei wird als erstes ein AST für den deklarierenden Typ der selektierten Methode erzeugt. Daraufhin wird geprüft, ob die Methode eine abstrakte Methode oder ein Interface implementiert (falls dies der Fall ist, darf sie nicht gelöscht werden). Bei der Prüfung der Methode wird ebenfalls ermittelt, ob, und wenn ja, welche Instanzvaria-

- ble sie zurückgibt. Diese wird in Schritt 4 bei der Erzeugung der Vermittlermethoden benötigt. Umgesetzt wird dieser Schritt über die Methode `checkSelectedMethod`.
2. Anschließend werden alle Referenzen zur selektierten Methode gesucht. Hierzu wird die Klasse `SearchEngine` der JDT verwendet.
 3. Pro `ICompilationUnit` (d.h. pro Java-Datei), zu der ein oder mehrere Suchtreffer vorliegen, wird anschließend ein AST (mit Hilfe eines `ASTRequestor`) erzeugt.
 4. Für jeden Suchtreffer innerhalb einer `ICompilationUnit` wird dann unter Zuhilfenahme des AST geprüft, ob die Voraussetzungen erfüllt sind, dass der gefundene Methodenaufruf geändert und eine neue Vermittlermethode eingefügt werden kann. Ist dies der Fall, werden die notwendigen Code-Änderungen bestimmt. Ein Suchtreffer, bei dem keine Änderung durchgeführt werden kann (weil bspw. kein verketteter Methodenaufruf vorliegt) führt dazu, dass die selektierte Methode nicht gelöscht werden darf.
 5. Nachdem alle Suchtreffer überprüft wurden, wird ggf. die Code-Änderung für eine Löschung der selektierten Methode vermerkt, falls weder der Benutzer dies durch die Option des Dialogs verhindert hat noch im Schritt 1 oder 4 eine Bedingung gefunden wurde, die dies nicht erlaubt.

Die Punkte 2 und 3 bestehen hierbei aus allgemeinen Routinen, wie sie häufiger im Zusammenhang mit Refactorings benötigt werden und auf die daher hier nicht weiter eingegangen wird. Für eine ausführliche Darstellung sei beispielsweise auf [Wid07] verwiesen. Im folgenden soll der Ablaufschritt 4 detaillierter vorgestellt werden.

Einstieg für den Ablaufschritt 4 ist die Methode `rewriteCompilationUnit`, die von demjenigen `ASTRequestor` aufgerufen wird, der beim Erzeugen der AST verwendet wurde (vgl. Schritt 3). Als Parameter bekommt diese Methode neben dem `ASTRequestor` die Suchtreffer, den AST und die dazugehörige `ICompilationUnit`. Zunächst wird eine Instanz der Klasse `ASTRewrite` erzeugt (vgl. Abschnitt 5.1.2). Jeder Suchtreffer, der im AST zu einem Knoten vom Typ `MethodInvocation` führt, wird an die Methode `checkMethodInvocation` weitergereicht. Führt ein Suchtreffer zu einem Knoten vom Typ `SuperMethodInvocation`, so wird lediglich vermerkt, dass ein Löschen der selektierten Methode nicht möglich ist.

Die Methode `checkMethodInvocation` prüft zunächst, ob sich der gefundene Methodenaufruf innerhalb eines verketteten Methodenaufrufs befindet. Ist dies nicht der Fall, wird vermerkt, dass die selektierte Methode nicht gelöscht werden darf und zur aufrufenden Methode zurückgekehrt. Andernfalls wird zunächst diejenige Methode ermittelt, die in der Kette unmittelbar auf dem gefundenen Methodenaufruf folgt. Hierbei handelt es sich um die neu einzufügende Vermittlermethode. Wurde diese bislang (bei der Überprüfung der vorherigen Suchtreffer) noch nicht eingefügt, so wird zunächst ein für die betreffende Typhierarchie eindeutiger Name für die Vermittlermethode ermittelt (`HideDelegateUtil.generateDelegateName`). Anschließend erfolgt die Erzeugung einer neuen Methode (hierfür stellt die Klasse `org.intellij.hidedelegate.refactoring.Modifications` einige Methoden zur Verfügung) sowie die Änderung der gefundenen Aufrufstelle dahingehend, dass nun die neue Vermittlermethode aufgerufen wird (ebenfalls über die zuvor genannte Klasse). Die neue Vermittlermethode wird zudem in einer Liste gespeichert, so dass im weiteren Verlauf bei der Überprüfung der nächsten Suchtreffer festgestellt werden kann, ob für eine bestimmte Methode bereits eine Vermittlermethode hinzugefügt wurde.

Für die Erzeugung einer neuen Methode werden folgende Fälle in Abhängigkeit des Typs, an den die zu verbergende Methode an der Aufrufstelle statisch gebunden ist, unterschieden:

- A) Ist dieser Typ identisch mit dem deklarierenden Typ der selektierten Methode, so wird die neue Vermittlermethode in diesem Typ eingefügt. Hierfür wird der unter Schritt 1 bereits erzeugte AST bzw. der dazugehörige ASTRewriter verwendet. Ebenfalls aus Schritt 1 wird berücksichtigt, ob die Vermittlermethode Aufrufe an eine Instanzvariable oder an die selektierte Methode weiterleiten soll.
- B) Handelt es sich bei dem Typ um einen Subtyp des deklarierenden Typs der selektierten Methode¹⁴, so wird die neue Vermittlermethode auch in dem deklarierenden Typ der gewählten Methode eingefügt. Hierbei werden jedoch in jedem Fall Methodenaufrufe an die selektierte Methode weitergeleitet (für die folglich vermerkt wird, dass sie nicht gelöscht werden darf).
- C) Ist der Typ hingegen ein Supertyp des deklarierenden Typs der selektierten Methode, so muss die neue Vermittlermethode in diesem Supertyp eingefügt werden, da andernfalls syntaktische Fehler entstehen würden. Handelt es sich bei dem Supertyp

¹⁴ Die `SearchEngine` liefert solche Suchtreffer in denjenigen Fällen, in denen ein Subtyp die selektierte Methode erbt (und damit nicht überschreibt).

um ein Interface (und der Benutzer hat die Option zur Erweiterung von Interfaces nicht deaktiviert), so wird in dem Interface entsprechend eine Vermittlermethode deklariert. Dieser Fall wird von der Methode `changeSupertypes` abgedeckt. Anschließend werden alle Klassen, die das Interface implementieren, ermittelt und jeweils um eine Implementierung dieser Methode erweitert (`createDelegatesForImplementingClasses`).

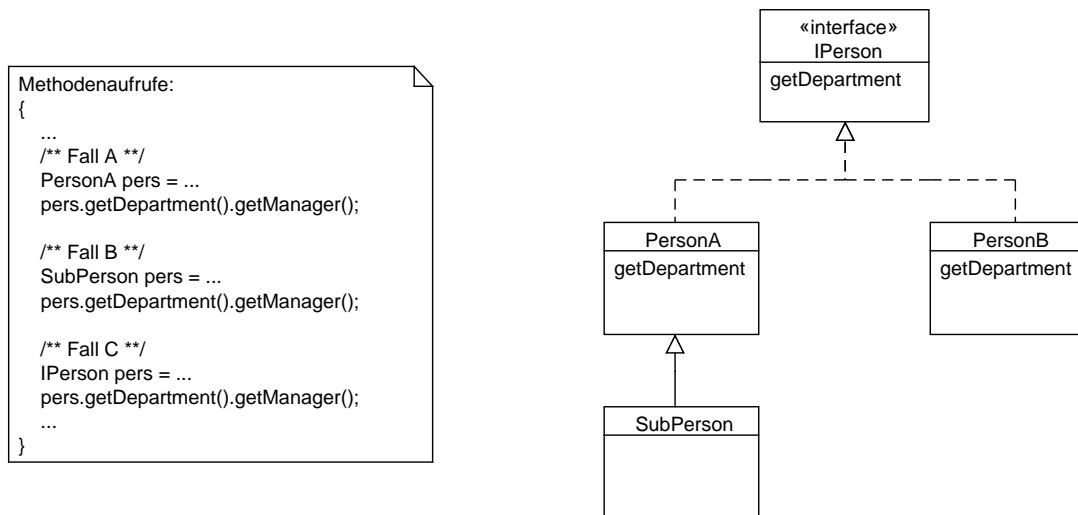


Abbildung 10: Fallunterscheidungen je nach Typ, an den der gefundene Methodenaufruf statisch gebunden ist. Ausgangslage ist in dem Beispiel die Selektion der Methode `PersonA.getDepartment()`.

Abbildung 10 veranschaulicht noch einmal die verschiedenen Fälle. Ausgangslage ist die selektierte Methode `PersonA.getDepartment()`. In der UML-Notiz sind die verschiedenen Fälle möglicher Suchtreffer aufgeführt.

Wurden alle Suchtreffer und Bedingungen überprüft und alle notwendigen Code-Änderungen bestimmt, so liefert die Methode `createChange` der Klasse `HideDelegateRefactoring` alle vorgesehenen Code-Änderungen in Form eines Objekts vom Typ `org.eclipse.ltk.core.refactoring.Change` an das Refactoring-Framework zurück (vgl. Abbildung 6). Dieses präsentiert dem Benutzer die geplanten Änderungen in einer Vorschau. Wird diese bestätigt, werden die Änderungen dann tatsächlich auf den beteiligten Ressourcen durchgeführt.

Die Klasse `HideDelegateRefactoring` besitzt darüber hinaus noch eine innere Klasse

`Statistics`, über die alle für eine Evaluation des Refaktorisierungswerkzeugs wichtigen Kennzahlen festgehalten und ausgegeben werden können.

Eine ausführliche Dokumentation aller Klassen findet sich zudem auf der beiliegenden CD als *Javadoc* im HTML-Format.

6. Evaluation

Dieses Kapitel beschreibt die Evaluierung des erstellten Refaktorisierungswerkzeugs. Eine formale Verifikation, dass das Refaktorisierungswerkzeug für alle denkbaren Eingaben stets korrekt arbeitet in dem Sinne, dass die modifizierten Programme sich weiterhin ausführen lassen und ein unverändertes beobachtbares Verhalten zeigen, ist extrem aufwändig und fehleranfällig [Ste10]. Daher wurde das erstellte Refactoring-Plug-in mit Hilfe des nachfolgend vorgestellten *Refactoring Tool Testers (RTT)* einem systematischen Test unterzogen.

Abschnitt 6.1 stellt zunächst die Kriterien vor, die im Rahmen der Evaluation des Refaktorisierungswerkzeugs überprüft wurden. Darauf folgend beschreibt Abschnitt 6.2 den RTT, dessen Grundidee und seine Architektur. Anschließend geht Abschnitt 6.3 näher darauf ein, wie die Prüfung mit dem RTT durchgeführt wurde. Die Testergebnisse werden abschließend in Abschnitt 6.4 vorgestellt. Eine Interpretation der Ergebnisse schließt sich im nachfolgenden Kapitel 7 an.

6.1. Beschreibung der untersuchten Kriterien

Das Ziel der Evaluierung des Refaktorisierungswerkzeugs ist die Überprüfung folgender Kriterien:

1. Anwendbarkeit:

Um die Praxistauglichkeit des Refactoring-Plug-ins zu überprüfen, wurden bei jeder Anwendung des Refactorings im Rahmen der Evaluierung statistische Informationen vom Plug-in protokolliert, die u.a. zu jeder Vorbedingung darüber Auskunft geben, ob sie erfüllt ist oder nicht. Hierüber lassen sich dann Werte ermitteln, in wie vielen Fällen das Refactoring angewandt werden kann und welche Vorbedingungen jeweils für die Nicht-Anwendbarkeit verantwortlich sind.

2. Korrektheit:

Definitionsgemäß muss ein Refactoring dafür Sorge tragen, dass nach Durchführung das refaktorierte Programm weiterhin ausführbar ist und ein unverändertes beobachtbares Verhalten zeigt. Hierzu wurde nach jeder Refaktorisierung festgestellt, ob das für den Test verwendete Programm Compile-Fehler enthält und ob alle Testfälle

des Programms weiterhin fehlerfrei durchgeführt werden konnten.

3. Auswirkung:

Um die Auswirkungen der Anwendung des Refactorings festzustellen, wurden eine Reihe von statistischen Informationen vom Plug-in protokolliert. Diese beinhalten insbesondere die Anzahl an neu hinzugefügten Vermittlermethoden, die Anzahl geänderter Methodenaufrufe und die Anzahl an Methoden, die gelöscht werden konnten. Des Weiteren wurde pro getestetem Programm vor und nach Durchführung aller möglichen Refaktorisierungen die Anzahl an Programmzeilen gemessen. Eines der wesentlichen Vorteile einer Anwendung des Law of Demeter ist nach deren Autoren die Erzielung einer geringeren Kopplung zwischen den Klassen eines objektorientierten Programms [LH89a] (vgl. Abschnitt 3.3). Um dies zu überprüfen, wurde die Metrik *Coupling Between Objects (CBO)* [CK94] von Chidamber und Kemerer gemessen. Diese gibt an, mit wie vielen anderen Klassen eine Klasse gekoppelt ist. Nach der Definition von Chidamber und Kemerer sind 2 Klassen dann miteinander gekoppelt, wenn Methoden der einen Klasse auf Methoden oder Instanzvariablen der anderen Klasse zugreifen [CK94].

6.2. Beschreibung des Refactoring Tool Testers

Der Refactoring Tool Tester (RTT) wurde am Lehrgebiet für Programmiersysteme der Fernuniversität in Hagen als Eclipse-Plug-in entwickelt¹⁵. Mit Hilfe des RTT kann ein Refaktorisierungswerkzeug systematisch getestet werden. Die Grundidee besteht darin, für den Test quelloffene Programme als Probanden einzusetzen. Für die Probanden sind dann für eine Testdurchführung (des Refaktorisierungswerkzeugs) diejenigen Programmstellen anzugeben, die für eine Refaktorisierung in Frage kommen. Der RTT wendet dann der Reihe nach für jede dieser Programmstellen die Refaktorisierung des zu überprüfenden Werkzeugs an. Nach jeder durchgeführten Refaktorisierung prüft der RTT zum einen, ob das modifizierte Programm noch ausführbar ist, indem es feststellt, ob es noch kompiliert. Zum anderen prüft er, ob das geänderte Programm ein für Refaktorisierungen definitionsgemäß unverändertes beobachtbares Verhalten zeigt. Diese Prüfung wird mit Hilfe der automatischen Tests der Probanden durchgeführt, die das Verhalten eines Programmes dokumentieren. Daher ist bei der Auswahl der Probanden darauf zu achten, dass die-

¹⁵ Der RTT kann von der folgenden Projektseite bezogen werden:
<http://www.fernuni-hagen.de/ps/prjs/rtt>

se eine möglichst hohe Testabdeckung (bspw. in Form von JUnit-Tests) aufweisen. Nach Durchführung der beiden Prüfungen wird die Refaktorisierung wieder rückgängig gemacht und mit der nächsten Programmstelle, auf die das Refactoring anwendbar ist, fortgefahren.

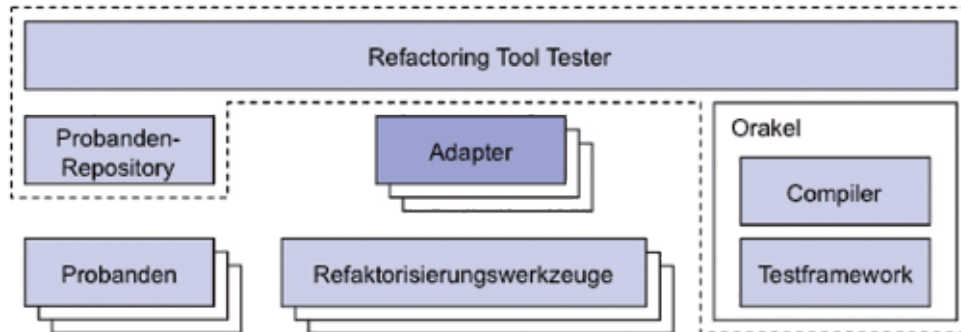


Abbildung 11: Architektur des Refactoring Tool Testers (Quelle: [Ste10])

Die Verwendung des RTT bietet also einige wesentliche Vorteile für eine Testdurchführung gegenüber einer manuellen Erstellung von Testprogrammen:

1. Der (Programmier-)Aufwand wird deutlich reduziert. Ein manuelles Erstellen von Testprogrammen, die die zu refaktorisierenden Programmstellen enthalten, führt zu einem erheblichen Aufwand. Insbesondere sind dabei diese Programmstellen jeweils in einen Kontext einzubetten, der geeignet ist, Fehler bei der Refaktorisierung zu provozieren.
2. Es wird eine deutlich bessere Testabdeckung durch Wahl geeigneter Probanden erzielt, die ihrerseits über eine hohe Testabdeckung verfügen. Je höher die Testabdeckung der Probanden, desto höher ist die Wahrscheinlichkeit, Verhaltensänderungen, die durch das Refaktorisierungswerkzeug verursacht werden, zu finden [Ste10].

Abbildung 11 veranschaulicht die Architektur des RTT. Die festen Bestandteile des RTT sind dabei gestrichelt umrandet. Das Probanden-Repository dient dazu, den Quellcode der Probanden inklusive ihrer automatischen Tests in einem Archiv hinterlegen zu können. Die (Test-)Orakel geben im Kontext des RTT darüber Auskunft, ob nach der Durchführung einer Refaktorisierung das modifizierte Programm noch ausführbar ist und ob es ein unverändertes beobachtbares Verhalten zeigt. Über die Ausführbarkeit des Programms gibt der in Eclipse integrierte Compiler Auskunft. Die Ausführung der Regressionstests der Probanden erfolgt über das Testframework (im Regelfall JUnit). Probanden und zu tes-

tende Refaktorisierungswerkzeuge können unverändert verwendet werden. Es ist lediglich ein Adapter zu schreiben, der ein zu testendes Refaktorisierungswerkzeug in den RTT einbindet und diejenigen Programmstellen der Probanden angibt, an denen die Refaktorisierung durchgeführt werden soll. [Ste10]

6.3. Prüfung mit dem Refactoring Tool Tester

Als Probanden kommen Java-Projekte in Frage, deren Quellcode frei zugänglich ist. Ausgewählt wurden Projekte, die allgemein bekannt und vielfach benutzt werden. Dabei wurde einerseits darauf geachtet, dass diese ein breites Spektrum an Java-Sprachkonstrukten (beispielsweise komplexe Vererbungshierarchien, Generics, innere und anonyme Klassen) abdecken. Auf der anderen Seite sollten die Projekte ihrerseits über eine möglichst hohe Testabdeckung verfügen. Dadurch wird eine höhere Wahrscheinlichkeit für die Entdeckung von semantischen Fehlern erreicht.

Als Testprobanden wurden folgende frei verfügbare Java-Projekte verwendet:

- *JUnit 4.8.1* (<http://sourceforge.net/projects/junit/>)
- *HTMLParser 1.6*, eine Java-Bibliothek zum Parsen von HTML-Seiten (<http://sourceforge.net/projects/htmlparser/>)
- *JPaul 2.5.1*, enthält eine Sammlung von Algorithmen zur Programmanalyse (<http://sourceforge.net/projects/jpaul/>)
- *Apache Ivy 2.2.0*, ein Tool zum Einbinden und Verwalten von abhängigen Java-Bibliotheken (<http://ant.apache.org/ivy/>)

Für die Durchführung der Tests wurde ein Adapter für den RTT implementiert, der diejenigen Programmstellen der Probanden auswählt, auf denen das Refactoring angewandt werden soll und das zu testende Refaktorisierungswerkzeug in den RTT einbindet. Da der Aufsetzpunkt für das Refactoring eine Methode ist, wurden für den RTT zunächst alle nicht-privaten Methoden selektiert, die einen Referenztypen zurückgeben. Hierbei wurden nur Methoden der Probanden selbst untersucht, nicht jedoch Methoden der zugehörigen Testfälle.

Zeile		JUnit	HTML	JPaul	Ivy	Gesamt
1	Anzahl untersuchter Methoden	233	441	263	1461	2398
2	davon verworfen	182	369	195	1137	1883
3	verbleibende Kandidaten	47	72	68	318	505
4	Refactoring durchführbar	16	59	7	187	269
5	in Prozent	34	82	10	59	53
6	Gründe für Nichtanwendbarkeit:					
7	(a) Rückgabetyp = dekl. Typ	22	13	0	129	164
8	(b) nur class-File	5	0	10	2	17
9	(c) Rückgabetyp = Typvariable	4	0	51	0	55
10	gefundene Compile-Fehler	0	2	0	0	2
11	Anzahl vorhandener Testfälle	583	673	21	832	2109
12	fehlgeschlagene Testfälle	0	0	0	0	0

Tabelle 1: Anwendbarkeit und Korrektheit des Refactorings

6.4. Ergebnisse

Tabelle 1 zeigt die ermittelten Werte für die Anwendbarkeit und Korrektheit des Refactorings. Zeile 1 enthält zunächst die Anzahl der Methoden, die für den RTT selektiert wurden. Viele Methoden sind jedoch gar nicht in einen verketteten Methodenaufruf eingebunden, so dass sie für die weitere Untersuchung verworfen wurden (Zeile 2). Zeile 3 zeigt die Anzahl der verbleibenden Kandidaten, bei denen die Ausgangslage des Refactorings tatsächlich vorliegt. In durchschnittlich 53% der Fälle konnte das Refactoring durchgeführt werden.

Dabei variiert die Anwendbarkeitsrate zwischen den Projekten sehr stark: Bei *JPaul* war ein Refactoring lediglich in ca. 10% der Fälle möglich. Dies liegt hauptsächlich daran, dass das Projekt sehr intensiv Generics verwendet und der Rückgabetyp der untersuchten Methoden oftmals über eine Typvariable deklariert ist (vgl. Zeile 9). Im Gegensatz dazu erreichte das Projekt *HTMLParser* die größte Anwendbarkeitsrate mit ca. 82%.

Ein Hauptgrund für die Nichtanwendbarkeit liegt in der Bedingung begründet, dass der Rückgabetyp der selektierten Methode nicht identisch mit demjenigen Typ sein darf, der diese Methode deklariert (vgl. Zeile 7). Lässt man bezüglich der Anwendbarkeitsrate diese Bedingung außen vor, da in diesen Fällen eine Anwendung des Refactorings keinen Sinn machen, so liegt die durchschnittliche Rate der untersuchten Projekte bei ca. 79%.

Ein weiterer Grund für eine Nichtanwendbarkeit liegt vor, wenn diejenige Klasse (oder

Interface), in die die neue(n) Vermittlermethode(n) eingefügt werden soll(en), nur als Class-File vorliegen (Zeile 8). Dies ist beispielsweise dann der Fall, wenn die zu untersuchende Methode aus einer Klasse stammt, die ein Interface der Java-Sprachbibliotheken implementiert und eine gefundene Methodenreferenz statisch an dieses Interface gebunden ist (vgl. hierzu auch Abschnitt 5.3).

Bei dem Projekt *HTMLParser* sind durch das Refactoring insgesamt 2 Compile-Fehler aufgetreten, die beide die gleiche Ursache haben. Sie wurden durch eine einzige Methode verursacht¹⁶. Listing 18 zeigt eine verallgemeinerte Darstellung des Problems und der Ursache.

```
1 public interface IA {
2     public Delegate getDelegate() throws Exception;
3 }
4
5 public interface IB extends IA {
6     public Delegate getDelegate(); // overrides IA.getDelegate()
7 }
8
9 public class Client1 {
10    public void m() throws Exception {
11        IA a = ...
12        a.getDelegate().doSomething(); // 1. Suchtreffer
13    }
14 }
15
16 public class Client2 {
17    public void n() {
18        IB b = ...
19        b.getDelegate().doSomething(); // 2. Suchtreffer
20    }
21 }
```

Listing 18: Ursache des Compile-Fehlers

Gegeben seien 2 Interfaces *IA* und *IB*, wobei *IB* eine Methode von *IA* überschreibt und dabei im Gegensatz zur überschriebenen Methode keine Exception in der **throws**-Klausel deklariert. Wird das Refactoring auf der Methode *IB.getDelegate()* angewandt, so werden alle Methodenreferenzen nacheinander untersucht. Wird die erste Referenz in der Klasse *Client1* gefunden, so erzeugt das Refactoring-Plug-in in Interface *IA* eine Deklaration für die Methode *doSomething()* inklusive der **throws**-Klausel (siehe Listing 19), da die Vermittlermethode in diesen Fällen Aufrufe immer an die selektierte Methode wei-

¹⁶ Es handelt sich hierbei um die Methode `SimpleNodeIterator.nextNode()`.

terleitet. Wird anschließend dann die zweite Referenz in der Klasse *Client2* untersucht, stellt das Plug-in fest, dass die Methodensignatur der zu erstellenden Vermittlermethode identisch ist mit der bereits erzeugten Vermittlermethode `doSomething()` und ändert daraufhin lediglich den Methodenaufruf. Dies führt in dem Kontext zu einem Compile-Fehler der Art „Unhandled Exception...“.

```

1 public interface IA {
2     public Delegate getDelegate() throws Exception;
3     public void doSomething() throws Exception; // neue Vermittlermethode
4 }
5
6 public class Client2 {
7     public void n() {
8         IB b = ...
9         // Ersetzung des Methodenaufrufs führt zu einem Compile-Fehler:
10        // "Unhandled Exception..."
11        b.doSomething();
12    }
13 }

```

Listing 19: Ausschnitt nach dem Refactoring

Die Ergebnisse zeigen, dass alle vorhandenen Testfälle weiterhin fehlerfrei durchlaufen. Anzumerken ist hierbei, dass alle Testfälle der Projekte herangezogen wurden, die im Originalzustand der Projekte fehlerfrei durchliefen. Im Projekt *Apache Ivy* gab es beispielsweise eine Testfallklasse¹⁷, die Tests zu konkurrierenden Threads durchführt und die im Originalzustand zu sowohl fehlerfreien als auch fehlerhaften Durchläufen führt. Insgesamt konnten also keinerlei semantische Veränderungen festgestellt werden.

Tabelle 2 stellt die gemessenen Auswirkungen des Refactorings zusammen. Die erste Zeile der Tabelle wiederholt noch einmal die Anzahl der Fälle, in denen das Refactoring durchführbar war. Es folgt die Angabe der Anzahl erzeugter Vermittlermethoden sowie eine Aufteilung dieser Zahl je nachdem, in welcher Klasse die Methode eingefügt wurde - in Relation zu derjenigen Klasse, die die untersuchte Methode deklariert. Wird ein Interface erweitert, so muss in jeder Klasse, die dieses Interface implementiert, ebenfalls die Vermittlermethode eingefügt werden (Zeile 7).

Weiterhin wurden die Anzahl geänderter Methodenaufrufe sowie die Anzahl gelöschter Methoden ausgewertet. Eine Methode wird nur dann gelöscht, wenn mindestens eine Ver-

¹⁷ Es handelt sich hierbei um die Klasse `ArtifactLockStrategyTest`.

Zeile		JUnit	HTML	JPaul	Ivy
1	Refactoring möglich	16	59	7	187
2	erzeugte Vermittlermethoden	25	204	7	807
3	davon				
4	in derselben Klasse	18	43	3	213
5	in Supertyp (ohne Interface)	3	11	4	4
6	in Interface	2	38	0	147
7	in Interface impl. Klassen	2	112	0	443
8	geänderte Methodenaufrufe	30	156	7	771
9	gelöschte Methoden	0	0	0	15
10	ELOC vorher	6317	18049	6877	41163
11	ELOC nachher	6396	18372	6887	42448
12	Differenz	79	323	10	1285
13	ELOC pro Refactoring	4,9	5,5	1,4	6,9
14	CBO vorher	2,63	3,29	2,93	7,05
15	CBO nachher	2,63	3,13	2,93	6,72

Tabelle 2: Auswirkungen des Refactorings

mittlermethode eingefügt werden konnte und alle für das Löschen erforderlichen Bedingungen (vgl. Abschnitt 4.4) erfüllt sind. Wird keine einzige Methodenreferenz gefunden, wird die untersuchte Methode auch nicht gelöscht, da der Benutzer beim Anwenden des Refactorings eher in der Erwartung steht, dass Methodenreferenzen vorliegen und für das alleinige Löschen von Methoden anderweitige Mittel zur Verfügung stehen.

Die bisher beschriebenen Werte wurden mit Hilfe des Refactoring-Plug-ins gesammelt. Darüber hinaus wurden 2 weitere Softwaremetriken gemessen¹⁸:

- ELOC (*Executable Lines of Code*): Gibt die Anzahl an ausführbaren Anweisungen eines Programms an. Die Kennzahl gibt Auskunft darüber, inwieweit sich der Umfang der Programme durch das Einfügen der Methoden verändert hat.
- CBO (*Coupling between Objects*): Zählt zu den objektorientierten Softwaremetriken von Chidamber und Kemerer und gibt an, inwieweit die Klassen des Programms miteinander gekoppelt sind (vgl. Abschnitt 6.1). Angegeben ist jeweils der durchschnittliche Wert aller Klassen des untersuchten Java-Projekts.

Diese Werte wurden jeweils einmal vor und einmal nach der Ausführung aller durchführ-

¹⁸ Die Werte wurden jeweils mit dem Eclipse-Plug-in *Stan* (Version 2.0.2) gemessen, welches auf der Internetseite <http://stan4j.com/eclipse/eclipse-integration.html> zur Verfügung steht.

baren Refactorings für das jeweilige Programm gemessen.

7. Diskussion

Dieses Kapitel stellt eine Interpretation der Ergebnisse der Evaluation vor und ordnet anschließend die Ergebnisse und diese Arbeit in den Kontext verwandter Arbeiten ein.

7.1. Interpretation der Ergebnisse

Die im vorangegangenen Kapitel vorgestellten Ergebnisse weisen eine Anwendbarkeitsrate von durchschnittlich 53% auf. Jedoch bilden diejenigen Fälle, bei denen der Rückgabebetyp identisch ist mit dem deklarierenden Typ der selektierten Methode, keine sinnvollen Anwendungsfälle des Refactorings. Werden diese Fälle ausgeklammert, so liegt die durchschnittliche Anwendbarkeitsrate bei einem akzeptablen Wert von ca. 79%.

Dabei spiegeln die Ergebnisse natürlich nur eine technische Sichtweise wieder. Aus den Zahlen, in denen das Refactoring anwendbar ist, läßt sich nicht ableiten, in welchen Fällen eine Anwendung auch tatsächlich sinnvoll ist und zu einem besseren Design führt. Die klassenbasierte Definition des Law of Demeter nennt in ihrer *minimierten Form* bereits Gründe, in denen es nicht geboten erscheint bzw. sinnvoll ist, das Gesetz anzuwenden (vgl. Abschnitt 3.2.2). So zum Beispiel beim Methodenaufruf von Klassen, die als stabil gelten können, deren Klasseninterface sich also den Erwartungen nach nicht ändern wird. Prominentes Beispiel hierfür ist die Klasse `java.lang.String`, bei der es nicht sinnvoll ist, eine Methode, die diese Klasse als Rückgabebetyp deklariert, zu verbergen und etliche Vermittlermethoden einzufügen.

Die Ergebnisse zeigen weiterhin, dass ein Löschen derjenigen Methode, auf die das Refactoring angewandt wird und die ja das zu verbergende Delegate-Objekt preisgibt, nur in den allerseltensten Fällen möglich war.

Durch das Einfügen der Vermittlermethoden steigt erwartungsgemäß auch der Umfang der Programme. Pro durchführbarem Refactoring wurden im Schnitt ca. 6 Codezeilen hinzugefügt. Würde man ausschließlich (und stark vereinfachend) die Anzahl an Codezeilen als Indiz für die Komplexität und Wartbarkeit eines Programmes heranziehen, so müßte man schlußfolgern, dass dieses Refactoring die allgemeinen Ziele von Refactorings (vgl. Abschnitt 2.1) verfehlt. Allerdings besagt das Gesetz von Demeter, dass bei Einhaltung des Gesetzes eine geringere Kopplung der Klassen erreicht und dadurch eine bessere Wart-

barkeit erzielt werden kann. Für die beiden kleineren untersuchten Projekte, *JUnit* und *JPaul*, läßt sich dies mit Hilfe der verwendeten Metrik CBO nicht nachweisen. Allerdings dürfte dies daran liegen, dass das Refactoring hier nur in sehr wenigen Fällen anwendbar war und eine Entkoppelung zweier Klassen in einer Methode noch nicht eine mögliche Kopplung in einer anderen Methode der beiden Klassen auflöst. Etwas anders sieht dies schon in den beiden größeren Projekten *HTMLParser* und *Apache Ivy* aus: Hier zeigen die Werte für CBO nach Anwendung aller Refactorings eine geringere, durchschnittliche Kopplung aller Klassen des Projekts an.

Wie in Kapitel 6 bereits erwähnt, ist es sehr schwierig, einen formalen Beweis für die Korrektheit eines Refaktorisierungswerkzeugs zu führen [Ste10]. Auch der im Rahmen der Evaluation durchgeführte systematische Test kann natürlich keinen Beweis dafür liefern, dass - abgesehen von dem gefundenen Fehler - keinerlei weitere Fehler mehr im entwickelten Refactoring-Plug-in vorhanden sind. Während der Entwicklungsphase des Plug-ins wurde der RTT ebenfalls eingesetzt, so dass viele anfängliche Fehler unmittelbar aufgedeckt und behoben werden konnten.

7.2. Verwandte Arbeiten

Palm et al. haben ein Eclipse-Plug-in mit dem Namen *DemeterCop*¹⁹ entwickelt, welches Verletzungen des Law of Demeter innerhalb von Java-Quellcodes kenntlich macht. Es ist im Rahmen einer Untersuchung entstanden, die zum Ziel hatte, festzustellen, inwieweit die Wartbarkeit von Software und die Anzahl an Verletzungen des Law of Demeter korrelieren [PAL03b] [Pal03a]. Zu diesem Zweck wurde ein sogenannter *Maintainability Index (MI)* verwendet, der einige bekannte Metriken wie beispielsweise die *Cyclomatic Complexity* von McCabe kombiniert. Mit Hilfe des Plug-ins wurde die Anzahl an Verletzungen des Law of Demeter für eine Reihe von quelloffenen Java-Projekten gemessen, für die dann ebenfalls auch der MI berechnet wurde.

Die vorläufigen Ergebnisse von Palm et al. deuteten zunächst an, dass durchaus ein Zusammenhang zwischen einer hohen Anzahl an Verletzungen und einem schlechten Wert für die Wartbarkeit (MI) existieren könne [PAL03b]. Die weiteren Untersuchungen, bei

¹⁹ Dieses Plug-in wurde in den Jahren 2002-2003 für die Eclipse-Version 2.1 entwickelt, ist danach aber nicht weiter gepflegt worden. Die Projektseite ist unter <http://sourceforge.net/projects/demetercop/> erreichbar.

denen eine Vielzahl weiterer Software-Metriken (auch für aufeinander folgende Releases der untersuchten Java-Projekte) gemessen wurden, konnten jedoch keinen Zusammenhang zwischen den Metriken und dem Law of Demeter sicher nachweisen [Pal03a].

DemeterCop beinhaltet jedoch nicht die Möglichkeit, automatisiert die gefundenen Verletzungen des Gesetzes von Demeter zu beheben. Im Gegensatz zur vorliegenden Arbeit konnten also keinerlei Werte nach Behebung aller gefundenen Verletzungen gemessen werden.

[LW] ist ein Eclipse-Plug-in, welches den Refactoring-Katalog von Eclipse um 3 Refactorings erweitert, u.a. um das Hide Delegate Refactoring. Die Entwickler wählten jedoch für die Realisierung einen anderen Ansatz: Hier ist ein verketteter Methodenaufruf zu selektieren, auf dem dann das Refactoring aufsetzt. Dem Empfänger des ersten Methodenaufrufs innerhalb der Kette wird dann eine Vermittlermethode hinzugefügt (die ihrerseits jedoch wieder einen mehrfach verketteten Methodenaufruf enthalten kann, wenn die ausgewählte Verkettung entsprechend viele Methodenaufrufe beinhaltet). Bezüglich des Namens der einzufügenden Vermittlermethode finden zudem keinerlei Prüfungen oder Warnungen statt, so dass der Benutzer Gefahr läuft, dass eine der unter Abschnitt 4.3 aufgeführten Vorbedingungen nicht erfüllt ist und somit Syntaxfehler erzeugt werden oder aber im schlimmeren Fall semantische Fehler entstehen, die gegebenenfalls unentdeckt bleiben.

Das Plug-in sucht zudem nach weiteren Vorkommen des selektierten verketteten Methodenaufrufs und passt die gefundenen Stellen entsprechend an. Handelt es sich bei dem Empfänger des ersten Methodenaufrufs um ein Interface, so ist das Refactoring nicht anwendbar.

Vorteilhaft bei diesem Ansatz ist der Aspekt, dass der Benutzer leichter darauf gestoßen wird, in welchen Fällen er das Refactoring anwenden kann: Bemerkt er einen verketteten Methodenaufruf im Programmcode, kann er unmittelbar auf diesem das Refactoring anwenden. Der im Rahmen dieser Abschlussarbeit gewählte Ansatz hat jedoch den Vorteil, dass in den Fällen, in denen unterschiedliche Methoden auf dem Delegate-Objekt aufgerufen werden, für diese in einer einzigen Anwendung unmittelbar alle notwendigen Vermittlermethoden erzeugt werden (bei Anwendung auf derjenigen Methode, die das Delegate-Objekt zurückgibt). Im Plug-in von [LW] muss stattdessen nacheinander für alle betroffenen Code-Stellen einzeln das Refactoring durchgeführt werden.

An der Technischen Universität Darmstadt wurde ein Eclipse-Projekt mit dem Namen

Eclipse Code Recommenders [Cod] initiiert, welches Tools entwickelt, die dem Entwickler beim Erlernen der (korrekten) Verwendung von Frameworks helfen sollen. Hierzu untersuchen die Tools die Verwendung von Frameworks²⁰ anhand von Beispiel-Anwendungen und bringen die Ergebnisse in einer vom Projekt genannten *Intelligent Code Completion* ein.

```
1  public class SampleView extends ViewPart {
2
3  private void showStatusMessage(String message) {
4      IStatusLineManager manager = // (*)
5      ...
6  }
7  ...
8  }
```

Listing 20: Beispiel für eine Java Chain Completion; Quelle: [Cod]

Ein Bestandteil hiervon ist die *Java Chain Completion*, die dem Entwickler Vorschläge anbietet, wie er Referenzen auf Objekte über mehrfach verkettete Methodenaufrufe erhalten kann. Listing 20 zeigt hierfür ein kurzes Beispiel: in Zeile 4 wird eine Variable vom Typ `IStatusLineManager` deklariert. Ruft der Entwickler an der markierten Code-Stelle in Zeile 4 (*) nun den Codeassistenten der *Java Chain Completion* auf, so wird ihm folgender verketteter Methdenaufruf vorgeschlagen:

```
getViewSite().getActionBars().getStatusLineManager().
```

Somit schlägt dieses Plug-in genau solche Muster vor, die im Rahmen des Hide Delegate Refactorings entfernt werden.

²⁰ Derzeit haben die Tools hauptsächlich Eclipse-eigene Frameworks untersucht und können bei der Verwendung dieser entsprechende Vorschläge zur *Code completion* geben.

8. Schlussbetrachtungen

8.1. Zusammenfassung und Fazit

Die Bedeutung des Refaktorisierens hat in den letzten Jahren zugenommen. Mit dazu beigetragen haben sicherlich auch die an Popularität zunehmenden so genannten agilen Prozeßmodelle, von denen das Extreme Programming eines der bekanntesten Vertreter ist. Ein Prinzip dieser agilen Prozeßmodelle besteht im ständigen Refaktorisieren. Somit steigt der Bedarf nach verlässlichen Refaktorisierungswerkzeugen.

Mit der vorliegenden Arbeit wurde ein Refaktorisierungswerkzeug zur Umsetzung des Law of Demeter implementiert. Das Gesetz verspricht bei Befolgung die Erzielung einer geringeren Kopplung zwischen den Klassen eines objektorientierten Systems und soll sich damit positiv auf die Wartbarkeit der Software auswirken.

Für das umgesetzte Refactoring Hide Delegate, mit dem Verletzungen des Law of Demeter behoben werden können, konnte eine Vielzahl an Vorbedingungen für die Programmiersprache Java beschrieben werden, die insbesondere auf die Frage abzielen, was zu beachten ist, wenn eine gegebene Methode in eine existierende Klasse hinzugefügt werden soll. Dass das Erkennen und Aufstellen der Vorbedingungen eines Refactorings alles andere als trivial ist [Ste07][Ste10], hat diese Arbeit ebenfalls bestätigen können.

Die Ergebnisse im Rahmen der Evaluation haben eine akzeptable Anwendbarkeitsrate des Werkzeugs gezeigt. Dies wurde sicherlich positiv durch die Entscheidung beeinflusst, für die einzufügenden Methoden einen eindeutigen Namen zu vergeben.

Weiterhin hat die Evaluierung erste Hinweise dafür liefern können, dass nach einem Befolgen des Law of Demeter tatsächlich eine geringere Kopplung erzielt wird. Die Zahl der untersuchten Java-Projekte reicht jedoch für eine Verallgemeinerung der Ergebnisse nicht aus.

8.2. Ausblick

Das umgesetzte Refactoring-Plug-in erlaubt die Anwendung des Refactorings Hide Delegate. Allerdings gilt hier (wie für andere Refaktorisierungswerkzeuge oftmals auch) die

Tatsache, dass der Benutzer zunächst einmal einen *Code smell* [FBB⁺99] erkannt haben muß, im vorliegenden Fall also eine Verletzung des Law of Demeter. Daher wäre es wünschenswert, das Plug-in beispielsweise um so genannte *Marker* zu erweitern, die dem Benutzer im Quellcode Verletzungen des Law of Demeter anzeigen und damit auf die Stellen hinweisen, an denen das Refactoring durchgeführt werden kann. Hierfür ist möglicherweise auch das in 2003 entwickelte Plug-in *DemeterCop* (vgl. Abschnitt 7.2) eine hilfreiche Quelle (auch wenn es seitdem nicht mehr gepflegt wurde).

Weiterhin kann mit Hilfe des entwickelten Plug-ins dieser Arbeit eine breit angelegtere Untersuchung für eine Vielzahl von Java-Projekten durchgeführt werden, die die Effekte der Berücksichtigung des Law of Demeter auf die Qualität der Software untersucht. Da das Gesetz ja nicht nur ausschließlich Vorteile mit sich bringt, könnte das Ziel einer solchen Untersuchung sein, festzustellen, ob im allgemeinen eher die positiven oder die negativen Aspekte überwiegen. Im Gegensatz zu den Untersuchungen von [PAL03b] und [Pal03a] ist hierbei ein „Vorher-Nachher-Vergleich“ möglich.

A. Installation und Bedienungshinweise

Das Plug-in Hide Delegate wurde mit der Eclipse-Version 3.5.1 entwickelt und getestet. Weitere Voraussetzung ist eine JRE 6. Für eine Installation ist lediglich die Datei `org.intoj.HideDelegate_1.0.0.jar` in das Verzeichnis `plugins` der Eclipse-Installation zu kopieren. Nach einem anschließenden Start von Eclipse steht die Funktionalität des Plug-ins zur Verfügung.

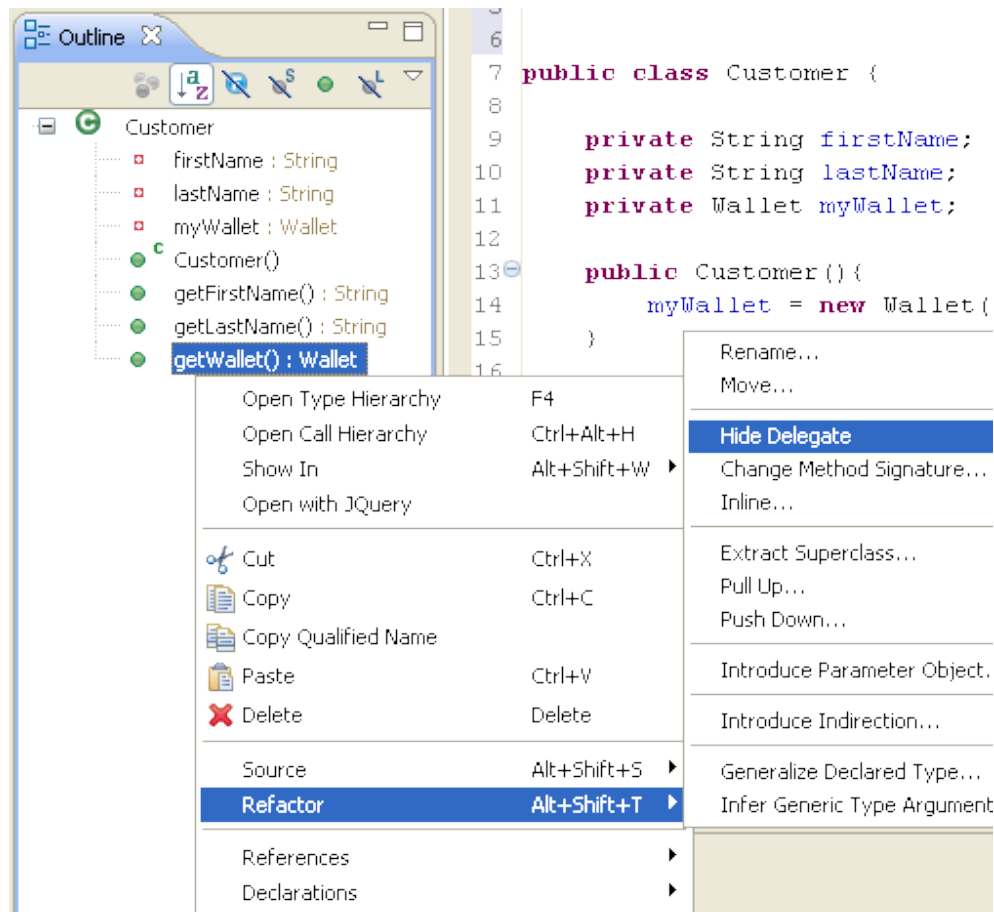


Abbildung 12: Aufruf des Refactorings Hide Delegate

Für eine Anwendung des Plug-ins ist diejenige Methode, die innerhalb eines verketteten Methodenaufrufs enthalten ist, in einer View wie beispielsweise der Outline-View von Eclipse zu selektieren. Aus dem Kontext-Menü heraus kann das Refactoring anschließend unter dem Punkt „Refactor > Hide Delegate“ aufgerufen werden (siehe Abbildung 12).

Anschließend erscheint die erste Seite des Refactoring-Dialogs (siehe Abbildung 13), auf

der dem Benutzer 2 Optionen angeboten werden. Zum einen kann er angeben, ob die von ihm selektierte Methode mit dem Refactoring-Vorgang gelöscht werden soll, falls hierfür alle erforderlichen Bedingungen erfüllt sind (vgl. Abschnitt 4.4). Zum anderen kann er angeben, ob auch Interfaces um neue Vermittlermethoden erweitert werden sollen, falls diese involviert sind (d.h. es existiert eine Methodenreferenz, die statisch an ein Interface gebunden ist).

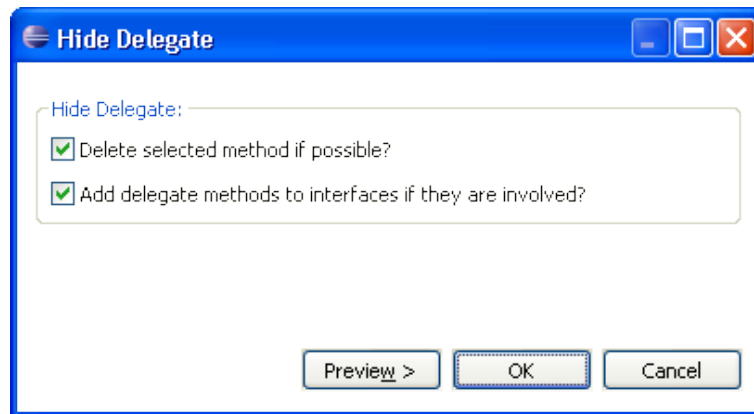


Abbildung 13: Erste Seite des Refactoring-Dialogs

Nach Betätigen des Buttons „Preview“ wird dem Benutzer anschließend eine Vorschau für die durchzuführenden Code-Änderungen angezeigt. An dieser Stelle hat der Benutzer die Möglichkeit, den Refactoring-Vorgang abubrechen oder aber durchführen zu lassen.

B. Beschreibung der beiliegenden CD

Dieser Abschlussarbeit liegt eine CD mit folgendem Inhalt bei:

- **Ein Refaktorisierungswerkzeug zur Umsetzung des LoD.pdf**

Der Text dieser Arbeit in Form eines PDF-Dokuments.

- **org.intoj.HideDelegate_1.0.0.jar**

Das entwickelte Eclipse-Plug-in in Form einer unmittelbar installierbaren JAR-Datei (vgl. Anhang A).

- **Ordner src**

In diesem Ordner befindet sich der Quellcode des umgesetzten Refactoring-Plug-ins in Form eines gepackten ZIP-Archivs, welches sich direkt in Eclipse importieren läßt („HideDelegate.zip“). Ebenfalls in dem Ordner ist der Quellcode des Testadapters, mit dem das Plug-in in den Refactoring Tool Tester eingebunden wurde („TestAdapter.zip“).

- **Ordner doc**

In diesem Ordner befindet sich die Quellcode-Dokumentation zum Plug-in im Javadoc-HTML-Format.

Abbildungsverzeichnis

1.	Ausgangssituation des Refactorings <i>Hide Delegate</i>	9
2.	Zieldesign des Refactorings <i>Hide Delegate</i>	10
3.	Klassendiagramm der Klassen Paperboy, Customer und Wallet	14
4.	Klassendiagramm nach Einführung einer Vermittlermethode	15
5.	Abstrakte Sicht auf die wesentlichen Elemente des Refactoring-Frameworks (Quelle: [Pet07])	40
6.	Lebenszyklus eines Refactorings (Quelle: [Pet07])	41
7.	Elemente des Java-Modells (Quelle: [Ecl09])	43
8.	Kindelemente von <code>ICmpilationUnit</code> (Quelle: [Ecl09])	44
9.	Visualisierte Darstellung eines AST anhand eines Hello-World-Programms	45
10.	Fallunterscheidungen für gefundene Methodenreferenzen	50
11.	Architektur des Refactoring Tool Testers (Quelle: [Ste10])	54
12.	Aufruf des Refactorings <i>Hide Delegate</i>	67
13.	Erste Seite des Refactoring-Dialogs	68

Listings

1.	Beispiel zum Law of Demeter	5
2.	Klassen Customer und Wallet	12
3.	Client-Klasse Paperboy	13
4.	Modifizierte Klassen Customer und Paperboy	14
5.	Überschreiben einer als <code>final</code> deklarierten Methode nicht möglich	24
6.	Einfügen einer Methode würde zur Reduzierung der Sichtbarkeit einer ge- erbten Methode führen.	25
7.	Eingefügte Methode <code>m</code> überschreibt eine andere und kann dadurch zu einer semantischen Änderung führen	26
8.	Das Einfügen einer Methode verursacht Syntaxfehler in Subklasse	27
9.	Ein <i>name clash</i> wird verursacht.	29
10.	Ein weiterer Fall, in dem ein <i>name clash</i> verursacht wird.	29
11.	Semantischer Fehler durch Überladen	30
12.	Beispiel für Auto boxing	31
13.	Semantischer Fehler durch Überladen in einem Subtyp.	32
14.	Syntaktische Fehler (<i>method is ambiguous</i>) im Zusammenhang mit dem Überladen	33
15.	Beispiel für geschachtelte Klassen	33
16.	Semantikänderung bei geschachtelten Klassen	34
17.	Verletzung des Gesetzes von Demeter auch ohne verketteten Methodenaufruf	38
18.	Ursache des Compile-Fehlers	57
19.	Ausschnitt nach dem Refactoring	58
20.	Beispiel für eine Java Chain Completion; Quelle: [Cod]	64

Literatur

- [Aes08] AESCHLIMANN, Martin: *JDT fundamentals Become a JDT tool smith*. http://www.eclipsecon.org/2008/sub/attachments/JDT_fundamentals.ppt.
Version: 2008
- [Boc] BOCK, David: *The Paperboy, The Wallet, and The Law Of Demeter*. <http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf>
- [CK94] CHIDAMBER, S. R. ; KEMERER, C. F.: A Metrics Suite for Object Oriented Design. In: *IEEE Trans. Softw. Eng.* 20 (1994), June, 476–493. <http://dx.doi.org/10.1109/32.295895>. – DOI 10.1109/32.295895. – ISSN 0098–5589
- [Cod] *Eclipse Code Recommenders*. <http://www.eclipse.org/recommenders/>
- [CR08] CLAYBERG, Eric ; RUBEL, Dan: *Eclipse Plug-ins*. 3. Addison-Wesley Professional, 2008. – ISBN 0321553462, 9780321553461
- [DFK⁺04] D’ANJOU, Jim ; FAIRBROTHER, Scott ; KEHN, Dan ; KELLERMAN, John ; MCCARTHY, Pat: *Java(TM) Developer’s Guide to Eclipse, The (2nd Edition)*. Addison-Wesley Professional, 2004. – ISBN 0321305027
- [Ecl09] ECLIPSE: *Help - Eclipse SDK*. <http://help.eclipse.org/galileo/index.jsp>. Version: 2009
- [FBB⁺99] FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS, Don: *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts : Addison Wesley, 1999. – ISBN 0201485672
- [Fow07] FOWLER, Martin: *Mocks Aren’t Stubs*. <http://www.martinfowler.com/articles/mocksArentStubs.html>. Version: 2007
- [Fre05] FRENZEL, Leif: Neutral im Sinne der Qualität. In: *Eclipse-Magazin* Vol. 5 (2005), S. 62–65
- [GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley,

1995. – ISBN 0201633612

- [GJSB05] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. – ISBN 0321246780
- [Kno01] KNOERNSCHILD, Kirk: *Java Design: Objects, UML, and Process*. Pearson Education, 2001. – ISBN 0201750449
- [Lar04] LARMAN, Craig: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2004. – ISBN 0131489062
- [LH89a] LIEBERHERR, Karl J. ; HOLLAND, Ian: Assuring Good Style for Object-Oriented Programs. In: *ieee-software* (1989), September, S. 38–48
- [LH89b] LIEBERHERR, Karl J. ; HOLLAND, Ian: Formulations and Benefits of the Law of Demeter. In: *SIGPLAN Notices* 24 (1989), March, Nr. 3, S. 67–78
- [LHR88] LIEBERHERR, K. ; HOLLAND, I. ; RIEL, A.: Object-oriented programming: an objective sense of style. In: *Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA : ACM, 1988 (OOPSLA '88). – ISBN 0-89791-284-5, 323–334
- [LW] LANCE WALTON, Channing W.: *Tane Eclipse Refactorings*. <http://www.stateofflow.com/projects/81/tane-eclipse-refactorings>
- [Mar03] MARTIN, Robert C.: *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2003. – ISBN 0135974445
- [Pal03a] PALM, Jeffrey: *On the Relationship Between Object-Oriented Metrics and Software Evolution*, University of Colorado at Boulder, Diplomarbeit, May 2003. <http://demetercop.sourceforge.net/JeffreyPalm-thesis.pdf>
- [PAL03b] PALM, Jeffrey ; ANDERSON, Kenneth M. ; LIEBERHERR, Karl: Investigating the Relationship Between Violations of the Law of Demeter and Software Maintainability. In: *Workshop on Software-Engineering Properties of Languages for*

Aspect Technologies, 2003

- [PCW85] PARNAS, D. L. ; CLEMENTS, P. C. ; WEISS, D. M.: The Modular Structure of Complex Systems. In: *IEEE Trans. Softw. Eng.* 11 (1985), March, 259–266. <http://dx.doi.org/10.1109/TSE.1985.232209>. – DOI 10.1109/TSE.1985.232209. – ISSN 0098–5589
- [Pet07] PETITO, Michael: *Eclipse Refactoring*. <http://people.clarkson.edu/~dhou/courses/EE564-s07/Eclipse-Refactoring.pdf>. Version: 2007
- [Rie96] RIEL, Arthur J.: *Object-Oriented Design Heuristics*. 1st. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1996. – ISBN 020163385X
- [Ste07] STEIMANN, Friedrich: *Moderne Programmier-techniken und -methoden*. Sommersemester 2009. Hagen, Germany : FernUniversität in Hagen, 2007 (Kurs 01853)
- [Ste08] STEIMANN, Friedrich: *Kurs 01814: Objektorientierte Programmierung*. Wintersemester 2009/2010. Hagen, Germany : FernUniversität in Hagen, 2008 (Kurs 01814)
- [Ste10] STEIMANN, Friedrich: Korrekte Refaktorisierungen: Der Bau von Refaktorisierungswerkzeugen als eigenständige Disziplin. In: *OBJEKTspektrum* (2010), Nr. 04/2010, 24-29. http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2010/04/steimann_OS_04_10.pdf
- [TK06] THOMAS KUHN, Olivier T.: *Abstract Syntax Tree*. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. Version: 2006
- [Wid07] WIDMER, Tobias: *Unleashing the Power of Refactoring*. <http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html>. Version: 2007

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Köln, im Oktober 2011

Diethelm Berens